El Proceso Unificado de Desarrollo de Software

IVAR JACOBSON
GRADY BOOCH
JAMES RUMBAUGH

La guía completa del Proceso Unificado escrita por sus creadores

Addison Wesley

The Addison-Wesley Object Technology Series

Grady Booch, Ivar Jacobson, and James Rumbaugh, Series Editors

Para tener más información, consulte el sitio web de la serie [http://www.awl.com/cseng/otseries/], así como las páginas de cada libro [http://www.awl.com/cseng/I-S-B-N/] (I-S-B-N- es el número de ISBN del libro en inglés, incluyendo los guiones).

David Bellin and Susan Suchman Simone, *The CRC Card Book*
ISBN 0-201-89535-8

Grady Booch, *Object Solutions: Managing the Object-Oriented Project*
ISBN 0-8053-0594-7

Grady Booch, *Object-Oriented Analysis and Design with Applications, Second Edition*
ISBN 0-8053-5340-2

Grady Booch, James Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide*
ISBN 0-201-57168-4

Don Box, *Essential COM*
ISBN 0-201-63446-5

Don Box, Keith Brown, Tim Ewald, and Chris Selis, *Effective COM: 50 Ways to Improve Your COM and MTS-based Applications*
ISBN 0-201-37968-6

Alistair Cockburn, *Surviving Object-Oriented Projects: A Manager's Guide*
ISBN 0-201-49834-0

Dave Collins, *Designing Object-Oriented User Interfaces*
ISBN 0-8053-5350-X

Bruce Powel Douglass, *Doing Hard Time:Designing and Implementing Embedded Systems with UML*
ISBN 0-201-49837-5

Bruce Powel Douglass, *Real-Time UML: Developing Efficient Objects for Embedded Systems*
ISBN 0-201-32579-9

Desmond F. D'Souza and Alan Cameron Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach*
ISBN 0-201-31012-0

Martin Fowler, *Analysis Patterns: Reusable Object Models*
ISBN 0-201-89542-0

Martin Fowler with Kendall Scott, *UML Distilled: Applying the Standard Object Modeling Language*
ISBN 0-201-32563-2

Peter Heinckiens, *Building Scalable Database Applications: Object-Oriented Design, Architectures, and Implementations*
ISBN 0-201-31013-9

Ivar Jacobson, Grady Booch, and James Rumbaugh, *The Unified Software Development Process*
ISBN 0-201-57169-2

Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*
ISBN 0-201-54435-0

Ivar Jacobson, Maria Ericsson, and Agneta Jacobson, *The Object Advantage: Business Process Reengineering with Object Technology*
ISBN 0-201-42289-1

Ivar Jacobson, Martin Griss, and Patrik Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*
ISBN 0-201-92476-5

David Jordan, *C++ Object Databases: Programming with the ODMG Standard*
ISBN 0-201-63488-0

Philippe Kruchten, *The Rational Unified Process: An Introduction*
ISBN 0-201-60459-0

Wilf LaLonde, *Discovering Smalltalk*
ISBN 0-8053-2720-7

Lockheed Martin Advanced Concepts Center and Rational Software Corporation, *Succeeding with the Booch and OMT Methods: A Practical Approach*
ISBN 0-8053-2279-5

Thomas Mowbray and William Ruh, *Inside CORBA: Distributed Object Standards and Applications*
ISBN 0-201-89540-4

Ira Pohl, *Object-Oriented Programming Using C++, Second Edition*
ISBN 0-201-89550-1

Rob Pooley and Perdita Stevens, *Using UML: Software Engineering with Objects and Components*
ISBN 0-201-36067-5

Terry Quatrani, *Visual Modeling with Rational Rose and UML*
ISBN 0-201-31016-3

Brent E. Rector and Chris Sells, *ATL Internals*
ISBN 0-201-69589-8

Doug Rosenberg with Kendall Scott, *Use Case Driven Object Modeling with UML: A Practical Approach*
ISBN 0-201-43289-7

Walker Royce, *Software Project Management: A Unified Framework*
ISBN 0-201-30958-0

William Ruh, Thomas Herron, and Paul Klinker, *IIOP Complete: Middleware Interoperability and Distributed Object Standards*
ISBN 0-201-37925-2

James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*
ISBN 0-201-30998-X

Geri Schneider and Jason P. Winters, *Applying Use Cases: A Practical Guide*
ISBN 0-201-30981-5

Yen-Ping Shan and Ralph H. Earle, *Enterprise Computing with Objects: From Client/Server Environments to the Internet*
ISBN 0-201-32566-7

David N. Smith, *IBM Smalltalk: The Language*
ISBN 0-8053-0908-X

Daniel Tkach, Walter Fang, and Andrew So, *Visual Modeling Technique: Object Technology Using Visual Programming*
ISBN 0-8053-2574-3

Daniel Tkach and Richard Puttick, *Object Technology in Application Development, Second Edition*
ISBN 0-201-49833-2

Jos Warmer and Anneke Kleppe, *The Object Constraint Language: Precise Modeling with UML*
ISBN 0-201-37940-6

EL PROCESO UNIFICADO DE DESARROLLO DE SOFTWARE

Ivar JACOBSON
Grady BOOCH
James RUMBAUGH
RATIONAL SOFTWARE CORPORATION

Traducción:

Salvador Sánchez
Universidad Pontificia de Salamanca en Madrid
Miguel Ángel Sicilia
Universidad Pontificia de Salamanca en Madrid
Carlos Canal
Universidad de Málaga
Francisco Javier Durán
Universidad de Málaga

Coordinación de la traducción y revisión técnica:

Luis Joyanes
Director del Departamento de Lenguajes y Sistemas Informáticos
Universidad Pontificia de Salamanca en Madrid
Ernesto Pimentel
Director del Departamento de Lenguajes y Sistemas Informáticos
Universidad de Málaga

ADDISON WESLEY
Madrid • México • Santafé de Bogotá • Buenos Aires • Caracas • Lima • Montevideo • San Juan
San José • Santiago • São Paulo • White Plains

Nº Registro:

Datos de catalogación bibliográfica

I. Jacobson, G. Booch, J. Rumbaugh
EL PROCESO UNIFICADO DE DESARROLLO DE SOFTWARE
PEARSON EDUCACIÓN, S. A., Madrid, 2000

ISBN: 84-7829-036-2
Materia: Informática 681.3

Formato 195 × 250 Páginas: 464

I. Jacobson, G. Booch, J. Rumbaugh
EL PROCESO UNIFICADO DE DESARROLLO DE SOFTWARE

No está permitida la reproducción total o parcial de esta obra
ni su tratamiento o transmisión por cualquier medio o método
sin autorización escrita de la Editorial.

DERECHOS RESERVADOS
© 2000 respecto a la primera edición en español por:
PEARSON EDUCACIÓN, S. A.
Núñez de Balboa, 120
28006 Madrid

ISBN: 84-7829-036-2
Depósito legal: M. 20.385-2000

ADDISON WESLEY es un sello editorial de PEARSON EDUCACIÓN, S. A.

Traducido de:
THE UNIFIED SOFTWARE DEVELOPMENT PROCESS
Copyright © 1999 Addison Wesley Longman Inc.

ISBN: 0-201-57169-2

Edición en español:
Editor: Andrés Otero
Asistente editorial: Ana Isabel García
Diseño de cubierta: DIGRAF, S. A.
Composición: COPIBOOK, S. L.
Impreso por: Imprenta FARESO, S. A.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Este libro ha sido impreso con papel y tintas ecológicos

Contenido

Prefacio XV

Parte I: El Proceso Unificado de Desarrollo de Software

Capítulo 1: El Proceso Unificado: dirigido por casos de uso, centrado en la arquitectura, iterativo e incremental 1

1.1. El Proceso Unificado en pocas palabras 4
1.2. El Proceso Unificado está dirigido por casos de uso 5
1.3. El Proceso Unificado está centrado en la arquitectura 5
1.4. El Proceso Unificado es iterativo e incremental 6
1.5. La vida del Proceso Unificado 8
1.5.1. El producto 9
1.5.2. Fases dentro de un ciclo 10
1.6. Un Proceso integrado 12

Capítulo 2: Las cuatro "P" en el desarrollo de software: Personas, Proyecto, Producto y Proceso 13

2.1. Las personas son decisivas 14
2.1.1. Los procesos de desarrollo afectan a las personas 14
2.1.2. Los papeles cambiarán 15
2.1.3. Convirtiendo "recursos" en "trabajadores" 16
2.2. Los proyectos construyen el producto 17
2.3. El producto es más que código 18
2.3.1. ¿Qué es un sistema software? 18
2.3.2. Artefactos 18

2.3.3. Un sistema posee una colección de modelos 19
2.3.4. ¿Qué es un modelo? 20
2.3.5. Cada modelo es una vista autocontenida del sistema 20
2.3.6. Dentro de un modelo 21
2.3.7. Relaciones entre modelos 21
2.4. El proceso dirige los proyectos 22
2.4.1. El proceso: una plantilla 22
2.4.2. Las actividades relacionadas conforman flujos de trabajo 22
2.4.3. Procesos especializados 24
2.4.4. Méritos del proceso 25
2.5. La herramientas son esenciales en el proceso 25
2.5.1. Las herramientas influyen en el proceso 25
2.5.2. El proceso dirige las herramientas 26
2.5.3. El equilibrio entre el proceso y las herramientas 27
2.5.4. El modelado visual soporta UML 27
2.5.5. Las herramientas dan soporte al ciclo de vida completo 28
2.6. Referencias 29

Capítulo 3: Un proceso dirigido por casos de uso 31

3.1. Desarrollo dirigido por casos de uso en pocas palabras 33
3.2. ¿Por qué casos de uso? 35
3.2.1. Para capturar los requisitos que aportan valor añadido 35
3.2.2. Para dirigir el proceso 36
3.3.3. Para idear la arquitectura y más... 37
3.3. La captura de casos de uso 38
3.3.1. El modelo de casos de uso representa los requisitos funcionales. 38
3.3.2. Los actores son el entorno del sistema 39
3.3.3. Los casos de uso especifican el sistema 39
3.4. Análisis, diseño e implementación para realizar los casos de uso 40
3.4.1. Creación del modelo de análisis a partir de los casos de uso ... 41
3.4.2. Cada clase debe cumplir todos sus roles de colaboración 45
3.4.3. Creación del modelo de diseño a partir del modelo de análisis .. 46
3.4.4. Los subsistemas agrupan a las clases 49
3.4.5. Creación del modelo de implementación a partir del modelo de diseño 50
3.5. Prueba de los casos de uso 52
3.6. Resumen 53
3.7. Referencias 54

Capítulo 4: Un proceso centrado en la arquitectura 55

4.1. La Arquitectura en pocas palabras 56
4.2. Por qué es necesaria la arquitectura 58
4.2.1. Comprensión del sistema 58
4.2.2. Organización del desarrollo 59
4.2.3. Fomento de la reutilización 59
4.2.4. Evolución del sistema 60
4.3. Casos de uso y arquitectura 61
4.4. Los pasos hacia una arquitectura 64

4.4.1. La línea base de la arquitectura es un sistema "pequeño y flaco" . . 65
4.4.2. Utilización de patrones arquitectónicos 67
4.4.3. Descripción de la arquitectura 69
4.4.4. El arquitecto crea la arquitectura 71
4.5. Por fin, una descripción de la arquitectura 72
4.5.1. La vista de la arquitectura del modelo de casos de uso 73
4.5.2. La vista de la arquitectura del modelo de diseño 74
4.5.3. La vista de la arquitectura del modelo de despliegue 76
4.5.4. La vista de la arquitectura del modelo de implementación 77
4.6. Tres conceptos interesantes 78
4.6.1. ¿Qué es una arquitectura? 78
4.6.2. ¿Cómo se obtiene? 78
4.6.3. ¿Cómo se describe? 78
4.7. Referencias 78

Capítulo 5. Un proceso iterativo e incremental 81

5.1. Iterativo e incremental en breve 82
5.1.1. Desarrollo en pequeños pasos 83
5.1.2. Lo que no es una iteración 84
5.2. ¿Por qué un desarrollo iterativo e incremental? 85
5.2.1. Atenuación de riesgos 85
5.2.2. Obtención de una arquitectura robusta 87
5.2.3. Gestión de requisitos cambiantes 87
5.2.4. Permitir cambios tácticos 88
5.2.5. Conseguir una integración continua 88
5.2.6. Conseguir un aprendizaje temprano 90
5.3. La aproximación iterativa es dirigida por los riesgos 90
5.3.1. Las iteraciones alivian los riesgos técnicos 91
5.3.2. La dirección es responsable de los riesgos no técnicos 93
5.3.3. Tratamiento de los riesgos 93
5.4. La iteración genérica 94
5.4.1. Lo que es una iteración 94
5.4.2. Planificación de las iteraciones 96
5.4.3. Secuenciación de las iteraciones 96
5.5. El resultado de una iteración es un incremento 97
5.6. Las iteraciones sobre el ciclo de vida 98
5.7. Los modelos evolucionan con las iteraciones 100
5.8 Las iteraciones desafían a la organización 101
5.9 Referencias 102

Parte II: Los flujos de trabajo fundamentales

Capítulo 6: Captura de requisitos: de la visión a los requisitos 105

6.1. Por qué la captura de requisitos es complicada 106
6.2. El objeto del flujo de trabajo de los requisitos 107
6.3. Visión general de la captura de requisitos 107
6.4. El papel de los requisitos en el ciclo de vida del software 111

6.5. La comprensión del contexto del sistema mediante un modelo del dominio ... 112
6.5.1. ¿Qué es un modelo del dominio? ... 112
6.5.2. Desarrollo de un modelo del dominio ... 114
6.5.3. Uso del modelo del dominio ... 115
6.6. La comprensión del contexto del sistema mediante un modelo del negocio. 115
6.6.1. ¿Qué es un modelo del negocio? ... 115
6.6.2. Cómo desarrollar un modelo del negocio ... 118
6.6.3. Búsqueda de casos de uso a partir de un modelo del negocio ... 120
6.7. Requisitos adicionales ... 121
6.8. Resumen ... 123
6.9. Referencias ... 123

Capítulo 7: Captura de requisitos como casos de uso ... 125

7.1. Introducción ... 125
7.2. Artefactos ... 127
7.2.1. Artefacto: modelo de casos de uso ... 127
7.2.2. Artefacto: actor ... 128
7.2.3. Caso de uso ... 129
7.2.4. Artefacto: descripción de la arquitectura (vista del modelo de casos de uso) ... 132
7.2.5. Artefacto: glosario ... 133
7.2.6. Artefacto: prototipo de interfaz de usuario ... 133
7.3. Trabajadores ... 133
7.3.1. Trabajador: analista del sistema ... 134
7.3.2. Trabajador: especificador de casos de uso ... 135
7.3.3. Diseñador de interfaces de usuario ... 135
7.3.4. Trabajador: arquitecto ... 136
7.4. Flujo de trabajo ... 136
7.4.1. Actividad: encontrar actores y casos de uso ... 138
7.4.2. Actividad: priorizar casos de uso ... 146
7.4.3. Actividad: detallar un caso de uso ... 147
7.4.4. Actividad: prototipar la interfaz de usuario ... 152
7.4.5. Actividad: estructurar el modelo de casos de uso ... 158
7.5. Resumen del flujo de trabajo de los requisitos ... 162
7.6. Referencias ... 163

Capítulo 8: Análisis ... 165

8.1. Introducción ... 165
8.2. El análisis en pocas palabras ... 168
8.2.1. Por qué el análisis no es diseño ni implementación ... 168
8.2.2. El objeto del análisis: resumen ... 169
8.2.3. Ejemplos concretos de cuándo hacer análisis ... 170
8.3. El papel del análisis en el ciclo de vida del software ... 171
8.4. Artefactos ... 172
8.4.1. Artefacto: modelo de análisis ... 172
8.4.2. Artefacto: clase del análisis ... 173
8.4.3. Artefacto: realización de caso de uso-análisis ... 177

8.4.4. Artefacto: paquete del análisis ... 181
8.4.5. Artefacto: descripción de la arquitectura (vista del modelo de análisis) ... 183
8.5. Trabajadores ... 184
8.5.1. Trabajador: arquitecto ... 184
8.5.2. Trabajador: ingeniero de casos de uso ... 185
8.5.3. Trabajador: ingeniero de componentes ... 186
8.6. Flujo de trabajo ... 187
8.6.1. Actividad: análisis de la arquitectura ... 187
8.6.2. Actividad: analizar un caso de uso ... 194
8.6.3. Actividad: analizar una clase ... 197
8.6.4. Actividad: analizar un paquete ... 201
8.7. Resumen del análisis ... 203
8.8. Referencias ... 204

Capítulo 9: Diseño ... 205

9.1. Introducción ... 205
9.2. El papel del diseño en el ciclo de vida del software ... 207
9.3. Artefactos ... 208
9.3.1. Artefacto: modelo de diseño ... 208
9.3.2. Artefacto: clase del diseño ... 209
9.3.3. Artefacto: realización de caso de uso-diseño ... 210
9.3.4. Artefacto: subsistema del diseño ... 213
9.3.5. Artefacto: interfaz ... 215
9.3.6. Artefacto: descripción de la arquitectura (vista del modelo de diseño) ... 216
9.3.7. Artefacto: modelo de despliegue ... 217
9.3.8. Artefacto: descripción de la arquitectura (vista del modelo de despliegue) ... 218
9.4. Trabajadores ... 218
9.4.1. Trabajador: arquitecto ... 218
9.4.2. Trabajador: ingeniero de casos de uso ... 219
9.4.3. Trabajador: ingeniero de componentes ... 220
9.5. Flujo de trabajo ... 220
9.5.1. Actividad: diseño de la arquitectura ... 221
9.5.2. Actividad: diseñar un caso de uso ... 237
9.5.3. Actividad: diseñar una clase ... 243
9.5.4. Actividad: diseñar un subsistema ... 250
9.6. Resumen del diseño ... 251
9.7. Referencias ... 253

Capítulo 10: Implementación ... 255

10.1. Introducción ... 255
10.2. El papel de la implementación en el ciclo de vida del software ... 256
10.3. Artefactos ... 257
10.3.1. Artefacto: modelo de implementación ... 257
10.3.2. Artefacto: componente ... 257

10.3.3. Artefacto: subsistema de la implementación 260
10.3.4. Artefacto: interfaz 262
10.3.5. Artefacto: descripción de la arquitectura (vista del modelo de implementación) 263
10.3.6. Artefacto: plan de integración de construcciones 264
10.4. Trabajadores 265
10.4.1. Trabajador: arquitecto 265
10.4.2. Trabajador: ingeniero de componentes 266
10.4.3. Trabajador: integrador de sistemas 266
10.5. Flujo de trabajo 267
10.5.1. Actividad: implementación de la arquitectura 268
10.5.2. Actividad: integrar el sistema 270
10.5.3. Actividad: implementar un subsistema 272
10.5.4. Actividad: implementar una clase 274
10.5.5. Actividad: realizar prueba unidad 276
10.6. Resumen de la implementación 279
10.7. Referencias 279

Capítulo 11: Prueba 281

11.1. Introducción 281
11.2. El papel de la prueba en el ciclo de vida del software 282
11.3. Artefactos 283
11.3.1. Artefacto: modelo de pruebas 283
11.3.2. Artefacto: caso de prueba 283
11.3.3. Artefacto: procedimiento de prueba 286
11.3.4. Artefacto: componente de prueba 287
11.3.5. Artefacto: plan de prueba 288
11.3.6. Artefacto: defecto 288
11.3.7. Artefacto: evaluación de prueba 288
11.4. Trabajadores 288
11.4.1. Trabajador: diseñador de pruebas 288
11.4.2. Trabajador: ingeniero de componentes 289
11.4.3. Trabajador: ingeniero de pruebas de integración 289
11.4.4. Trabajador: ingeniero de pruebas del sistema. 289
11.5. Flujo de trabajo 290
11.5.1. Actividad: planificar prueba 291
11.5.2. Actividad: diseñar prueba 292
11.5.3. Actividad: implementar prueba 295
11.5.4. Actividad: realizar pruebas de integración 296
11.5.5. Actividad: realizar prueba de sistema 297
11.5.6. Actividad: evaluar prueba 297
11.6. Resumen de la prueba 299
11.7. Referencias 299

Parte III: El Desarrollo iterativo e incremental

Capítulo 12: El flujo de trabajo de iteración genérico 303

12.1. La necesidad de equilibrio 304

12.2. Las fases son la primera división del trabajo 305
12.2.1. La fase de inicio establece la viabilidad 305
12.2.2. La fase de elaboración se centra en la factibilidad 306
12.2.3. La fase de construcción construye el sistema 307
12.2.4. La fase de transición se mete dentro del entorno del usuario . 308
12.3. La iteración genérica 308
12.3.1. Los flujos de trabajo fundamentales se repiten en cada iteración 308
12.3.2. Los trabajadores participan en los flujos de trabajo 309
12.4. El planificar precede al hacer 310
12.4.1. Planear las cuatro fases 311
12.4.2. Plan de iteraciones 312
12.4.3. Pensar a largo plazo 313
12.4.4. Planear los criterios de evaluación 313
12.5. Los riesgos influyen en la planificación del proyecto 314
12.5.1. Administrar la lista de riesgos 314
12.5.2. Los riesgos influyen en el plan de iteración 315
12.5.3. Planificar la acción sobre los riesgos 316
12.6. Asignación de prioridades a los casos de uso 316
12.6.1. Riesgos específicos de un producto particular 317
12.6.2. Riesgo de no conseguir la arquitectura correcta 317
12.6.3. Riesgo de no conseguir los requisitos correctos 319
12.7. Recursos necesitados 319
12.7.1. Los proyectos difieren enormemente 320
12.7.2. Un proyecto típico tiene este aspecto 321
12.7.3. Los proyectos más grandes tienen mayores necesidades 321
12.7.4. Una nueva línea de productos requiere experiencia 322
12.7.5. El pago del coste de los recursos utilizados 323
12.8. Evaluar las iteraciones y las fases 324
12.8.1. Criterios no alcanzados 324
12.8.2. Los criterios mismos 325
12.8.3. La siguiente iteración 325
12.8.4. Evolución del conjunto de modelos 326

Capítulo 13: La fase de inicio pone en marcha el proyecto 327

13.1. La fase de inicio en pocas palabras 327
13.2. Al comienzo de la fase de inicio 328
13.2.1. Antes de comenzar la fase de inicio 328
13.2.2. Planificación de la fase de inicio 329
13.2.3. Ampliación de la descripción del sistema 330
13.2.4. Establecimiento de los criterios de evaluación 330
13.3. Flujo de trabajo arquetípico de una iteración en la fase de inicio 332
13.3.1. Introducción a los cinco flujos de trabajo fundamentales 332
13.3.2. Ajuste del proyecto al entorno de desarrollo 334
13.3.3. Identificación de los riesgos críticos 334
13.4. Ejecución de los flujos de trabajo fundamentales, de requisitos a pruebas 334
13.4.1. Recopilación de requisitos 335

13.4.2. Análisis 337
13.4.3. Diseño 338
13.4.4. Implementación 339
13.4.5. Pruebas 339
13.5. Realización del análisis inicial de negocio 340
13.5.1. Esbozar la apuesta económica 340
13.5.2. Estimar la recuperación de la inversión 341
13.6. Evaluación de la iteración o iteraciones de la fase de inicio 341
13.7. Planificación de la fase de elaboración 342
13.8. Productos de la fase de inicio 343

Capítulo 14: La fase de elaboración construye la línea base de la arquitectura 345

14.1. La fase de elaboración en pocas palabras 345
14.2. Al comienzo de la fase de elaboración 346
14.2.1. Planificación de la fase de elaboración 346
14.2.2. Formación del equipo 347
14.2.3. Modificación del entorno de desarrollo 347
14.2.4. Establecimiento de criterios de evaluación 347
14.3. Flujo de trabajo arquetípico de una iteración en la fase de elaboración 348
14.3.1. Recopilación y refinamiento de la mayor parte de los requisitos 349
14.3.2. Desarrollo de la línea base de la arquitectura 349
14.3.3. Iterando mientras el equipo es pequeño 350
14.4. Ejecución de los flujos de trabajo fundamentales, de requisitos a pruebas 350
14.4.1. Recopilar los requisitos 351
14.4.2. Análisis 353
14.4.3. Diseño 357
14.4.4. Implementación 360
14.4.5. Pruebas 361
14.5. Desarrollo del análisis del negocio 363
14.5.1. Preparar la apuesta económica 363
14.5.2. Actualizar la recuperación de la inversión 363
14.6. Evaluación de las iteraciones de la fase de elaboración 364
14.7. Planificación de la fase de construcción 364
14.8. Productos clave 365

Capítulo 15: La construcción lleva a la capacidad operación inicila . 367

15.1. La fase de construcción en pocas palabras 367
15.2. Al comienzo de la fase de construcción 368
15.2.1. Asignación de personal para la fase 368
15.2.2. Establecimiento de los criterios de evaluación 369
15.3. Flujo de trabajo arquetípico de una iteración en la fase de construcción . 370
15.4. Ejecución de los flujos de trabajo fundamentales, de requisitos a pruebas 371
15.4.1. Requisitos 372

15.4.2. Análisis ... 373
15.4.3. Diseño ... 374
15.4.4. Implementación ... 375
15.4.5. Pruebas ... 377
15.5. Control del análisis de negocio ... 378
15.6. Evaluación de las iteraciones y de la fase de construcción ... 378
15.7. Planificación de la fase de transición ... 379
15.8. Productos clave ... 379

Capítulo 16: La transición completa la versión del producto ... 381

16.1. La fase de transición en pocas palabras ... 382
16.2. Al comienzo de la fase de transición ... 383
16.2.1. Planificación de la fase de transición ... 383
16.2.2. Asignación de personal para la fase ... 384
16.2.3. Establecimiento de los criterios de evaluación ... 385
16.3. Los flujos de trabajo fundamentales desempeñan un papel pequeño en esta fase ... 385
16.4. Lo que se hace en la fase de transición ... 386
16.4.1. Preparación de la versión beta ... 387
16.4.2. Instalación de la versión beta ... 387
16.4.3. Reacción a los resultados de las pruebas ... 388
16.4.4. Adaptación del producto a entornos de usuario variados ... 388
16.4.5. Finalización de los artefactos ... 390
16.4.6. ¿Cuándo acaba el proyecto? ... 390
16.5. Finalización del análisis del negocio ... 391
16.5.1. Control del progreso ... 391
16.5.2. Revisión del plan del negocio ... 391
16.6. Evaluación de la fase de transición ... 391
16.6.1. Evaluación de las iteraciones y de la fase ... 392
16.6.2. Autopsia del proyecto ... 392
16.7. Planificación de la próxima versión o generación ... 393
16.8. Productos clave ... 393

Capítulo 17: Cómo hacer que el Proceso Unificado funcione ... 395

17.1. El Proceso Unificado ayuda a manejar la complejidad ... 395
17.1.1. Los objetivos del ciclo de vida ... 396
17.1.2. La arquitectura del ciclo de vida ... 396
17.1.3. Capacidad operativa inicial ... 397
17.1.4. Lanzamiento del producto ... 397
17.2. Los temas importantes ... 397
17.3. La dirección lidera la conversión al Proceso Unificado ... 398
17.3.1. La necesidad de actuar ... 399
17.3.2. La directriz de reingeniería ... 399
17.3.3. Implementación de la transición ... 400
17.4. Especialización del Proceso Unificado ... 402
17.4.1. Adaptación del proceso ... 402
17.4.2. Completando el marco de trabajo del proceso ... 403
17.5. Relación con comunidades más amplias ... 403

17.6. Obtenga los beneficios del Proceso Unificado 404
17.7. Referencias 405

Apéndice A: Visión general del UML 407

A.1. Introducción 407
A.1.1. Vocabulario 408
A.1.2. Mecanismos de extensibilidad 408
A.2. Notación gráfica 409
A.2.1. Cosas estructurales 409
A.2.2. Elementos de comportamiento 410
A.2.3. Elementos de agrupación 411
A.2.4. Elementos de anotación 411
A.2.5. Relaciones de dependencia 411
A.2.6. Relaciones de asociación 411
A.2.7. Relaciones de generalización 412
A.2.8. Mecanismos de extensibilidad 412
A.3. Glosario de términos 412
A.4. Referencias 418

Apéndice B: Extensiones del UML específicas del Proceso Unificado 419

B.1. Introducción 419
B.2. Estereotipos 419
B.3. Valores etiquetados 422
B.4. Notación gráfica 424
B.5. Referencias 424

Apéndice C: Glosario general 425

C.1. Introducción 425
C.2. Términos 425

Índice 435

Prefacio

Hay gente que cree que las empresas profesionales deberían organizarse en torno a las habilidades de individuos altamente cualificados, que saben cómo hacer el trabajo y lo hacen bien, y que raramente necesitan dirección sobre las políticas y procedimientos de la organización para la que trabajan.

Esta creencia es una equivocación en la mayoría de los casos, y una grave equivocación en el caso del desarrollo de software. Por supuesto, los desarrolladores de software están altamente cualificados, pero la profesión es aún joven. En consecuencia, los desarrolladores necesitan dirección organizativa, a la cual, en este libro, llamamos "proceso de desarrollo de software". Además, debido a que el proceso que ponemos en marcha en este libro representa la unión de metodologías antes separadas, nos sentimos justificados al llamarlo "Proceso Unificado". No sólo reúne el trabajo de tres autores, sino que incorpora numerosas aportaciones de otras personas y empresas que han contribuido a UML, así como un número significativo de aportaciones fundamentales de personas en *Rational Software Corporation*. Surge de manera especial de la experiencia directa de cientos de organizaciones que han trabajado en sus oficinas con las primeras versiones del proceso.

Un director de una orquesta sinfónica durante un concierto hace poco más que decir a los músicos cuándo comenzar y ayudarles a tocar juntos. Él o ella no puede hacer más porque ha dirigido a la orquesta durante los ensayos y la preparación de las partituras, y porque cada músico está muy preparado en su propio instrumento y en realidad lo toca de manera independiente del resto de los otros miembros de la orquesta. Lo que es más importante para nuestros propósitos, cada músico sigue un "proceso" diseñado hace mucho tiempo por el compositor. Es la partitura musical la que proporciona el grueso de "la política y el procedimiento" que guían el concierto. En contraste, los desarrolladores de software no trabajan de manera independiente; interaccionan unos con otros y con los usuarios. No tienen una partitura —mientras no tengan un *proceso*.

La necesidad de un proceso promete hacerse más crítica, especialmente en empresas u organizaciones en las cuales los sistemas software son esenciales, tales como las financieras, las de control de tráfico aéreo, las de defensa y las de sistemas de telecomunicaciones. Con esto queremos decir que la dirección con éxito del negocio o la ejecución de la misión pública depende del software que la soporta. Estos sistemas software se hacen más complejos, su tiempo de salida al mercado necesita reducirse, y su desarrollo, por tanto, se hace más difícil. Por razones como éstas, la industria del software necesita un proceso para guiar a los desarrolladores, al igual que una orquesta necesita la partitura de un compositor para dirigir el concierto.

¿Qué es un proceso de desarrollo de software?

Un proceso define *quién* está haciendo *qué*, *cuándo*, y *cómo* alcanzar un determinado objetivo. En la ingeniería del software el objetivo es construir un producto software o mejorar uno existente. Un proceso efectivo proporciona normas para el desarrollo eficiente de software de calidad. Captura y presenta las mejores prácticas que el estado actual de la tecnología permite. En consecuencia, reduce el riesgo y hace el proyecto más predecible. El efecto global es el fomento de una visión y una cultura comunes.

Es necesario un proceso que sirva como guía para todos los participantes —clientes, usuarios, desarrolladores y directores ejecutivos. No nos sirve ningún proceso antiguo; necesitamos uno que sea el *mejor* proceso que la industria pueda reunir en este punto de su historia. Por último, necesitamos un proceso que esté ampliamente disponible de forma que todos los interesados puedan comprender su papel en el desarrollo en el que se encuentran implicados.

Un proceso de desarrollo de software debería también ser capaz de evolucionar durante muchos años. Durante esta evolución debería limitar su alcance, en un momento del tiempo dado, a las realidades que permitan las tecnologías, herramientas, personas y patrones de organización.

- *Tecnologías*. El proceso debe construirse sobre las tecnologías —lenguajes de programación, sistemas operativos, computadores, estructuras de red, entornos de desarrollo, etc.— disponibles en el momento en que se va a emplear el proceso. Por ejemplo, hace veinte años el modelado visual no era realmente de uso general. Era demasiado caro. En aquellos tiempos, un creador de un proceso prácticamente tenía que asumir que se usarían diagramas hechos a mano. Esa suposición limitaba mucho el grado en el cual el creador del proceso podía establecer el modelado dentro del proceso.

- *Herramientas*. Los procesos y las herramientas deben desarrollarse en paralelo. Las herramientas son esenciales en el proceso. Dicho de otra forma, un proceso ampliamente utilizado puede soportar la inversión necesaria para crear las herramientas que lo soporten.

- *Personas*. Un creador del proceso debe limitar el conjunto de habilidades necesarias para trabajar en el proceso a las habilidades que los desarrolladores actuales poseen, o apuntar aquellas que los desarrolladores puedan obtener rápidamente. Hoy es posible empotrar en herramientas software técnicas que antes requerían amplios conocimientos, como la comprobación de la consistencia en los diagramas del modelo.

- *Patrones de organización*. Aunque los desarrolladores de software no pueden ser expertos tan independientes como los músicos de una orquesta, están muy lejos de los trabajadores autómatas en los cuales Frederick W. Taylor basó su "dirección científica" hace cien años. El creador del proceso debe adaptar el proceso a las realidades del momento

—hechos como las empresas virtuales; el trabajo a distancia a través de líneas de alta velocidad; la mezcla (en empresas pequeñas recién montadas) de socios de la empresa, empleados asalariados, trabajadores por obra, y subcontratas de *outsourcing*; y la prolongada escasez de desarrolladores de software.

Los ingenieros del proceso deben equilibrar estos cuatro conjuntos de circunstancias. Además, el equilibrio debe estar presente no sólo ahora, sino también en el futuro. El creador del proceso debe diseñar el proceso de forma que pueda evolucionar, de igual forma que el desarrollador de software intenta desarrollar un sistema que no sólo funciona este año, sino que evoluciona con éxito en los años venideros. Un proceso debe madurar durante varios años antes de alcanzar el nivel de estabilidad y madurez que le permitirá resistir a los rigores del desarrollo de productos comerciales, manteniendo a la vez un nivel razonable de riesgo en su utilización. El desarrollo de un producto nuevo es bastante arriesgado en sí mismo como para añadirle el riesgo de un proceso que esté poco validado por la experiencia de su uso. En estas circunstancias, un proceso puede ser estable. Sin este equilibrio de tecnologías, herramientas, personas y organización, el uso del proceso sería bastante arriesgado.

Objetivos de este libro

Este libro presenta el proceso de desarrollo que estuvo constantemente en nuestras cabezas mientras desarrollábamos el Lenguaje Unificado de Modelado. Aunque UML nos ofrece un modo estándar de visualizar, especificar, construir, documentar y comunicar los artefactos de un sistema muy basado en el software, por supuesto somos conscientes de que un lenguaje como éste debe utilizarse en el contexto de un proceso de software completo. UML es un medio, y no un fin. El objetivo final es una aplicación software robusta, flexible y escalable. Es necesario tanto un proceso como un lenguaje para poder obtenerla, y el objetivo de este libro es mostrar la parte del proceso. Aunque proporcionamos un breve apéndice sobre UML, no pretende ser completo ni detallado. Para un tutorial detallado sobre UML, remitimos a *El Lenguaje Unificado de Modelado*, traducción de la *Guía de Usuario del Lenguaje Unificado de Modelado* [11]. Para una referencia completa de UML remitimos al *Lenguaje Unificado de Modelado Manual de Referencia* [12].

Audiencia

Este libro está destinado a cualquier persona implicada en el desarrollo de software. Se dirige principalmente a miembros del equipo de desarrollo que se dedican a las siguientes actividades del ciclo de vida: requisitos, análisis, diseño, implementación y pruebas —es decir, a trabajos que producen modelos UML. Así, por ejemplo, este libro es útil para analistas y usuarios finales (que especifican la estructura y comportamiento requeridos por el sistema), para desarrolladores de aplicaciones (que diseñan los sistemas que satisfacen esos requisitos), para programadores (que convierten esos diseños en código ejecutable), para ingenieros de prueba (que verifican y validan la estructura y comportamiento del sistema), para desarrolladores de componentes (que crean y catalogan componentes), y para directores de proyecto y de producto.

Este libro presupone un conocimiento básico de conceptos de orientación a objetos. También es útil, pero no se requiere, experiencia en desarrollo de software y en algún lenguaje orientado a objetos.

Método del libro

Hemos dedicado la mayoría del espacio de este libro a aquellas actividades —requisitos, análisis y diseño— sobre las cuales UML hace mayor hincapié. Es en esas áreas de mayor énfasis en las que el proceso desarrolla la *arquitectura* de sistemas software complejos. Sin embargo, tratamos el proceso completo, aunque con menos detalle. No en vano, es el programa ejecutable el que se ejecuta finalmente. Para llegar hasta él, un proyecto depende de los esfuerzos de cada miembro del equipo, así como del soporte de los usuarios. Como se verá, el proceso descansa en una tremenda variedad de actividades. Es necesario producir y hacer el seguimiento de muchos artefactos. Todas las actividades deben gestionarse.

El tratamiento completo del ciclo del ciclo de vida queda fuera de la intención de cualquier libro. Un libro que lo hiciese tendría que cubrir normas de diseño, plantillas para los artefactos, indicadores de calidad, gestión del proyecto, gestión de la configuración, métricas y más, ¡mucho más! Con el desarrollo del acceso interactivo, ese "más" está hoy disponible, y puede actualizarse según dicten los nuevos avances. Por esto, remitimos al lector al Proceso Unificado de Rational, un producto software que puede utilizarse desde la Web, que orienta a los equipos de desarrollo hacia prácticas de desarrollo de software más efectivas. (Consultar para más información http://www.rational.com.) Al cubrir el ciclo de vida completo, el Proceso Unificado de Rational extiende el Proceso Unificado más allá de las áreas descritas en este libro y ofrece flujos de trabajo adicionales que este libro no cubre o que menciona sólo de pasada, como el modelado del negocio, la gestión del proyecto, y la gestión de la configuración.

Historia del Proceso Unificado

El Proceso Unificado está equilibrado por ser el producto final de tres décadas de desarrollo y uso práctico. Su desarrollo como producto sigue un camino (véase la Figura P.1) desde el Proceso Objectory (primera publicación en 1987) pasando por el Proceso Objectory de Rational (publicado en 1997) hasta el Proceso Unificado de Rational (publicado en 1998). Su desarrollo

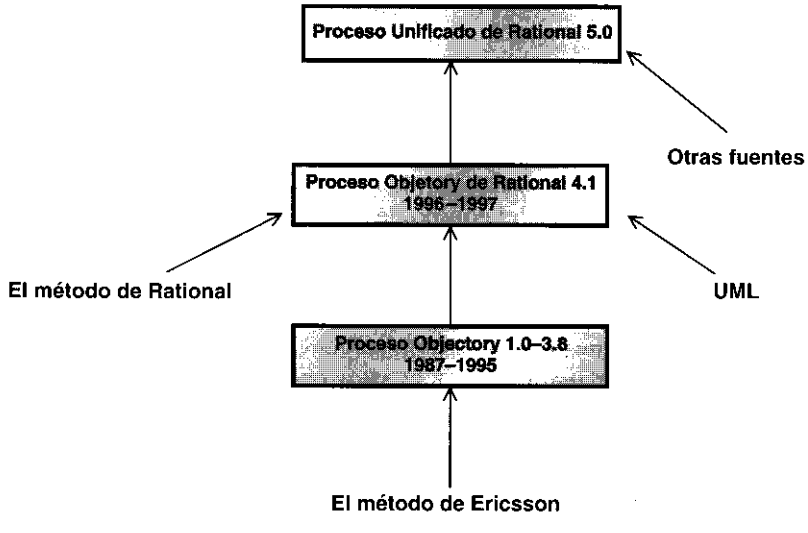


Figura P.1. El desarrollo del Proceso Unificado (las versiones del producto se muestran en rectángulos coloreados en gris).

ha recibido influencias de muchas fuentes. No pretendemos tratar de identificarlas todas (realmente no sabemos cuáles son todas), trabajo que dejamos a la investigación de los arqueólogos del software. Sin embargo, describiremos la influencia sobre el producto de los métodos de Ericsson y de Rational, así como el de varias otras fuentes.

El método de Ericsson

El Proceso Unificado posee raíces profundas. En palabras de Peter F. Drucker, es una "innovación basada en el conocimiento". Él nos informa de que "hay un lapso de tiempo prolongado entre la emergencia del nuevo conocimiento y su destilación en tecnología utilizable". "Entonces, sucede otro largo periodo antes de que esta nueva tecnología aparece en el mercado en productos, procesos o servicios." [1]

Un motivo para este largo tiempo de aparición es que la innovación basada en el conocimiento se cimienta en la unión de muchos tipos de conocimiento, y esto lleva su tiempo. Otra razón es que las personas que tienen que hacer efectiva la nueva idea necesitan tiempo para digerirla y comunicarla a los demás.

Como primer paso hacia el alumbramiento del desarrollo del Proceso Unificado, nos remontaremos a 1967 para esbozar los logros de Ericsson [14], [15], [16]. Ericsson modelaba el sistema entero como un conjunto de bloques interconectados (en UML, se los conoce como "subsistemas" y se implementan mediante "componentes"). Después, ensamblaba los bloques de más bajo nivel en subsistemas de más alto nivel, para hacer el sistema más manejable. Identificaban los bloques estudiando los casos de negocio —hoy conocidos como "casos de uso"—, previamente especificados. Para cada caso de uso, identificaban los bloques que debían cooperar para realizarlo. Con el conocimiento de las responsabilidades de cada bloque, preparaban su especificación. Sus actividades de diseño producían un conjunto de diagramas de bloques estáticos con sus interfaces, agrupados en subsistemas. Estos diagramas de bloques se corresponden directamente con una versión simplificada de los diagramas de clases o paquetes de UML —simplificados en que sólo mostraban las asociaciones que se utilizaban para comunicaciones.

El primer producto del trabajo de las actividades de diseño era una *descripción de arquitectura* software. Se basaba en la comprensión de los requisitos más críticos, y describía brevemente cada bloque y su agrupamiento en subsistemas. Un conjunto de diagramas de bloques describían a los bloques y a sus interconexiones. Sobre las interconexiones se comunicaban señales, es decir, un tipo de mensaje. Todos los mensajes quedaban descritos, uno por uno, en una biblioteca de mensajes. La descripción de la arquitectura software y la biblioteca de mensajes eran los documentos fundamentales que guiaban el trabajo de desarrollo, pero también se utilizaban para presentar el sistema a los clientes. En aquellos momentos (1968) los clientes no estaban acostumbrados a que les presentasen los productos software por medios similares a los datos de proyectos de ingeniería.

Para cada caso de uso, los ingenieros preparaban bien un diagrama de secuencia o bien un diagrama de colaboración (hoy desarrollados adicionalmente en UML). Estos diagramas mostraban cómo los bloques se comunicaban dinámicamente para llevar a cabo el caso de uso. Preparaban una especificación en forma de grafo de estados (que incluía sólo estados y transiciones) y un grafo de transición de estados (una versión simplificada de los diagramas de actividad de UML). Este método, el diseñar a partir de bloques con interfaces bien definidos, fue la clave del éxito. De ese modo, se podía crear una nueva configuración del sistema —por ejemplo, para un cliente nuevo —intercambiando un bloque por otro que proporcionase las mismos interfaces.

Por tanto, los bloques no eran sólo subsistemas y componentes de código fuente; se compilaban en bloques ejecutables, se instalaban en la máquina destino uno por uno, y se comprobaba que funcionaban con el resto de los bloques ejecutables. Es más, debía ser posible instalar cada bloque ejecutable, nuevo o modificado, sobre la marcha en un sistema en ejecución, mientras éste gestionaba llamadas de sistemas de telefonía en operación durante el 100 por ciento del tiempo. No se puede parar sistemas de ese tipo sólo para hacer cambios. Sería como cambiar las ruedas a un coche que circula a 100 kilómetros por hora.

En esencia, el método que utilizaban era el que hoy conocemos como *desarrollo basado en componentes*. Ivar Jacobson fue el creador de este método de desarrollo. Él dirigió su evolución hacia un proceso de desarrollo de software durante muchos años en el periodo anterior a Objectory.

El Lenguaje de Descripción y Especificación

Un avance significativo durante este periodo fue la publicación en 1976 por parte del CCITT, el organismo internacional para la estandarización en el área de las telecomunicaciones, del *Lenguaje de Especificación y Descripción* (*Specification and Description Language, SDL*) para el comportamiento funcional de los sistemas de telecomunicación. Este estándar, influenciado significativamente por el método de Ericsson, especificaba un sistema como un conjunto de bloques interconectados que se comunicaban unos con otros únicamente a través de mensajes (llamados "señales" en el estándar). Cada bloque poseía un conjunto de "procesos", que era el término SDL para designar las clases activas. Un proceso poseía instancias de manera muy parecida a cómo lo hacen las clases en términos de orientación a objetos. Las instancias de los procesos interactuaban mediante mensajes. SDL proponía diagramas que eran especializaciones de lo que hoy UML llama diagramas de clases, diagramas de actividad, diagramas de colaboración y diagramas de secuencia.

Por tanto, SDL era un estándar de modelado de objetos especializado. Se actualiza periódicamente, y todavía lo utilizan más de 10.000 desarrolladores y cuenta con el soporte de varios fabricantes de herramientas. Fue desarrollado inicialmente hace veinte años, y estaba muy por delante de su tiempo. Sin embargo, se desarrolló en un momento en el cual el modelado de objetos no había madurado. Probablemente, SDL será sustituido por UML, que se estandarizó en 1997.

Objectory

En 1987 Ivar Jacobson dejó Ericsson y fundó Objectory AB en Estocolmo. Durante los siguientes ocho años, él y sus colaboradores desarrollaron un proceso denominado Objectory ("Objectory" es una abreviatura de "Object Factory", fábrica de objetos). Extendieron su uso en otras industrias además de las de telecomunicaciones, y en otros países aparte de Suiza.

Aunque el concepto de *caso de uso* había estado presente en el trabajo de Ericsson, ahora se le había dado un nombre (que se presentó en la conferencia OOPSLA de 1987), se había desarrollado una técnica de representación, y la idea se había ampliado para abarcar una variedad de aplicaciones. Es decir, los casos de uso que dirigían el desarrollo se hicieron más claros. Dicho de otro modo, la arquitectura que conduce a los desarrolladores e informa a los usuarios comenzó a destacar.

Los flujos de trabajo sucesivos se representaron en una serie de modelos: requisitos-casos de uso, análisis, diseño, implementación y prueba. Un modelo es una perspectiva del sistema. Las relaciones entre los modelos de esta serie eran importantes para los desarrolladores como forma de hacer el seguimiento de una característica de un extremo a otro de la serie de modelos. De hecho, la trazabilidad se convirtió en un prerrequisito del desarrollo dirigido por casos de uso. Los desarrolladores podían seguir la traza de un caso de uso a través de la secuencia de modelos hasta el código fuente o bien, cuando surgían problemas, volver hacia atrás.

El desarrollo del proceso Objectory continuó en una serie de versiones, desde Objectory 1.0 en 1988 a la primera versión interactiva, Objectory 3.8, en 1995 (puede consultarse una visión general de Objectory en [2]).

Es importante hacer notar que el propio producto Objectory llegó a ser visto como un sistema. Esta forma de describir un proceso —como un producto en forma de sistema— proporcionaba una manera mejor de desarrollar una nueva versión de Objectory a partir de una anterior. Este modo de desarrollar Objectory hizo más fácil el ajustarlo para cubrir las necesidades específicas de diferentes organizaciones de desarrollo. El hecho de que el propio proceso de desarrollo de software Objectory era un producto de ingeniería era una característica única.

La experiencia en el desarrollo de Objectory también aportó ideas sobre cómo diseñar los procesos generales sobre los cuales opera un negocio. Eran aplicables los mismos principios y estos se recogieron en un libro en 1995 [3].

El método de Rational

Rational Software Corporation compró Objectory AB a finales de 1995 y la tarea de unificar los principios básicos subyacentes en los procesos de desarrollo existentes adquirió una urgencia especial. Rational había desarrollado algunas prácticas de desarrollo de software, la mayoría de ellas complementarias a las contenidas en Objectory.

Por ejemplo, como recordaron James E. Archer Junior y Michael T. Devlin en 1986 [4], "En 1981, Rational se dispuso a crear un entorno interactivo que mejoraría la productividad en el desarrollo de grandes sistemas software". A continuación dijeron que en este esfuerzo, eran importantes el diseño orientado a objetos, la abstracción, la ocultación de la información, la reutilización y el prototipado.

Muchos libros, artículos y documentos internos detallan los desarrollos de Rational desde 1981, pero quizá las dos contribuciones más importantes al *proceso* fueron los énfasis en la arquitectura y en el desarrollo iterativo. Por ejemplo, en 1990, Mike Devlin escribió un artículo introductorio sobre un proceso de desarrollo iterativo dirigido por la arquitectura. Philippe Kruchten, a cargo de la división de Prácticas de Arquitectura en Rational, firmó artículos sobre la iteración y la arquitectura.

Mencionaremos uno de ellos, un artículo sobre una representación de la arquitectura con cuatro vistas: la vista lógica, la vista de procesos, la vista física, y la vista de desarrollo, más una vista adicional que ilustraba las primeras cuatro vistas mediante casos de uso o escenarios [6]. La idea de tener un conjunto de vistas, en lugar de tratar de meter todo en un único tipo de diagrama, nació de la experiencia de Kruchten en varios proyectos grandes. Las vistas múltiples permitieron encontrar, tanto a los usuarios como a los desarrolladores, lo que necesitaban para sus diferentes objetivos con la vista adecuada.

Hay gente que percibe el desarrollo iterativo como algo caótico o anárquico. El método de cuatro fases (comienzo, elaboración, construcción y transición) se diseñó para estructurar mejor y controlar el proceso durante las iteraciones. La planificación detallada de las fases y la ordenación de las iteraciones dentro de las fases fue un esfuerzo conjunto entre Walker Royce y Rich Reitman, junto con la participación continua de Grady Booch y Philippe Kruchten.

Booch estaba presente desde los principios de Rational, y en 1996 en uno de sus libros mencionó dos "principios fundamentales" sobre la arquitectura y la iteración:

- "Un estilo de desarrollo dirigido por la arquitectura es normalmente la mejor aproximación para la creación de la mayoría de los proyectos complejos muy basados en el software."
- "Para que un proyecto orientado a objetos tenga éxito, debe aplicarse un proceso iterativo e incremental." [7]

El Proceso Objectory de Rational: 1995-1997

En el momento de la fusión, Objectory 3.8 había demostrado que se puede crear y modelar un proceso de desarrollo software como si fuese un producto. Había diseñado la arquitectura original de un proceso de desarrollo software. Había identificado un conjunto de modelos que documentaban el resultado del proyecto. Estaba correctamente desarrollado en áreas como el modelado de casos de uso, análisis y diseño, aunque en otras áreas —gestión de requisitos aparte de los casos de uso, implementación y pruebas—no estaba tan bien desarrollado. Además, decía poco sobre gestión del proyecto, gestión de la configuración, distribución, y sobre la preparación del entorno de desarrollo (obtención del proceso y las herramientas).

Por ello se le añadieron la experiencia y prácticas de Rational para formar el Proceso Objectory de Rational 4.1. Se añadieron, en concreto, las fases y la aproximación iterativa *controlada*. Se hizo explícita la arquitectura en forma de una descripción de la arquitectura —la "Biblia" de la organización de desarrollo de software.

Se desarrolló una definición precisa de la arquitectura, considerada como la parte más significativa de la organización del sistema. Representaba la arquitectura como vistas arquitectónicas de los modelos. Se amplió el desarrollo iterativo, pasando de ser un concepto relativamente general a ser un método dirigido por los riesgos que consideraba la arquitectura en primer lugar.

En estos momentos, UML estaba en fase de desarrollo y se incorporó como el lenguaje de modelado del Proceso Objectory de Rational (Rational Objectory Process, ROP). Los autores de este libro colaboraron como creadores originales de UML. El equipo de desarrollo del proceso, liderado por Philippe Kruchten, corrigió algunas de las debilidades del ROP reforzando, por ejemplo, la gestión del proyecto, basada en aportaciones de Royce [8].

El Lenguaje Unificado de Modelado

Era evidente desde hace algún tiempo la necesidad de un lenguaje de modelado visual y consistente, en el cual expresar los resultados de las bastante numerosas metodologías de orientación a objetos existentes a principios de los noventa.

Durante este periodo, Grady Booch, por ejemplo, era el autor del método Booch [9], y James Rumbaugh era el desarrollador principal de OMT *(Object Modelling Technique)* [10] creado en

el Centro de Investigación y Desarrollo de General Electric. Cuando este último se incorporó a Rational en Octubre de 1994, los dos comenzaron el trabajo de unificar sus métodos, en coordinación con muchos de los clientes de Rational. Publicaron la versión 0.8 del Método Unificado en Octubre de 1995, casi en el mismo momento de la incorporación de Ivar Jacobson a Rational.

Los tres, trabajando juntos, publicaron la versión 0.9 del Lenguaje Unificado de Modelado. El esfuerzo se amplió para incluir a otros metodologistas, así como a diversas empresas, que incluían a IBM, HP y Microsoft, cada una de las cuales contribuyó al estándar en evolución. En noviembre de 1997, después de pasar por el proceso de estandarización, el Object Management Group publicó como estándar la versión 1.1 del Lenguaje Unificado de Modelado. Remitimos a la *Guía de Usuario* [11] y al *Manual de Referencia* [12] para obtener información detallada.

El Proceso Objectory de Rational utilizó UML en todos sus modelos.

El Proceso Unificado de Rational

Durante este periodo, Rational compró o se fusionó a otras empresas fabricantes de herramientas. Cada una de ellas aportó a la mezcla su experiencia en áreas del proceso que ampliaron más aún el proceso Objectory de Rational:

- Requisite Inc. aportó su experiencia en gestión de requisitos.
- SQA Inc. había desarrollado un proceso de prueba para acompañar a su producto de pruebas, y lo añadió a la dilatada experiencia de Rational en este campo.
- Pure-Atria añadió su experiencia en gestión de configuración a la de Rational.
- Performance Awareness añadió las pruebas de rendimiento y las de carga.
- Vigortech añadió su experiencia en ingeniería de datos.

El proceso también se amplió con un nuevo flujo de trabajo para el modelado del negocio, basado en [3], que se utiliza para obtener los requisitos a partir de los procesos de negocio que el software va a cubrir. También se extendió con diseño de interfaces de usuario dirigido por los casos de uso (basado en el trabajo de Objectory AB).

A mediados de 1998 el Proceso Objectory de Rational se había convertido en un proceso hecho y derecho, capaz de soportar el ciclo de vida del desarrollo en su totalidad. Para ello, integraba una amplia variedad de aportaciones, no sólo de los tres autores de este libro, sino de las muchas fuentes sobre las cuales sobre las cuales se basaron Rational y UML. En junio, Rational publicó una nueva versión del producto, el Proceso Unificado de Rational 5.0 [13]. En ese momento, por primera vez, se pusieron a disposición del público en general muchos elementos de ese proceso propietario a través de este libro.

El cambio de nombre refleja el hecho de que la unificación ha tenido lugar en muchas dimensiones: unificación de técnicas de desarrollo, a través del Lenguaje Unificado de Modelado, y unificación del trabajo de muchos metodologistas —no sólo en Rational sino también en las oficinas de los cientos de clientes que llevaban utilizando el proceso muchos años.

Agradecimientos

Un proyecto de esta magnitud es el fruto del trabajo de mucha gente, y nos gustaría dar las gracias personalmente a tantos como sea posible.

Por aportaciones a este libro

Birgitte Lønvig preparó el ejemplo del sistema Interbank y lo desarrolló en todos los modelos. Éste es el ejemplo principal que utilizamos a lo largo del libro.

Patrik Jonsson extrajo material de la documentación del Proceso Objectory de Rational y lo dispuso en el orden de los capítulos propuestos. También ayudó en la preparación de los ejemplos, aportando muchas ideas sobre la mejor manera de presentar el Proceso Unificado.

Ware Myers participó en el desarrollo de este libro desde el esquema inicial en adelante. Convirtió los primeros borradores preparados por el autor principal en prosa inglesa más legible.

De los revisores agradecemos especialmente a Kurt Bittner, Cris Kobryn y Earl Ecklund, Jr. Además, agradecemos principalmente las revisiones de Walker Royce, Philippe Kruchten, Dean Leffingwell, Martin Griss, Maria Ericsson y Bruce Katz. También fueron revisores Pete McBreen, Glenn Jones, Johan Galle, N. Venu Gopal, David Rine, Mary Loomis, Marie Lenzi, Janet Gardner, y algunos revisores anónimos, a todos los cuales queremos dar las gracias.

Terry Quatrani de Rational mejoró el inglés de los Capítulos 1 al 5. Karen Tongish corrigió el libro entero. Nuestros agradecimientos para ambos.

Queremos dar las gracias en particular a Stefan Bylund que revisó a conciencia los borradores y sugirió mejoras detalladas, muchas de las cuales se han incorporado. Sus aportaciones han aumentado considerablemente la calidad del libro.

Durante los años

También queremos dar las gracias a bastantes personas que nos han ayudado a "mantener al día el proceso" y que han apoyado el trabajo de diversas maneras. En concreto, queremos dar las gracias a las siguientes personas: Stefan Ahlquist, Ali Ali, Gunilla Andersson, Kjell S. Andersson, Sten-Erik Bergner, Dave Bernstein, Kurt Bittner, Per Bjork, Hans Brandtberg, Mark Broms, Stefan Bylund, Ann Carlbrand, Ingemar Carlsson, Margaret Chan, Magnus Christerson, Geoff Clemm, Catherine Connor, Hakan Dahl, Stephane Desjardins, Mike Devlin, Hakan Dyrhage, Susanne Dyrhage, Staffan Ehnebom, Christian Ehrenborg, Maria Ericsson, Gunnar M. Eriksson, Iain Gavin, Carlo Goti, Sam Guckenheimer, Bjorn Gullbrand, Sunny Gupta, Marten Gustafsson, Bjorn Gustafsson, Lars Hallmarken, David Hanslip, Per Hedfors, Barbara Hedlund, Jorgen Hellberg, Joachim Herzog, Kelli Houston, Agneta Jacobson, Sten Jacobson, Paer Jansson, Hakan Jansson, Christer Johansson, Ingemar Johnsson, Patrik Jonsson, Dan Jonsson, Bruce Katz, Kurt Katzeff, Kevin Kelly, Anthony Kesterton, Per Kilgren, Rudi Koster, Per Kroll, Ron Krubeck, Mikael Larsson, Bud Lawson, Dean Leffingwell, Rolf Leidhammar, Hakan Lidstrom, Lars Lindroos, Fredrik Lindstrom, Chris Littlejohns, Andrew Lyons, Jas Madhur, Bruce Malasky, Chris McClenaghan, Christian Meck, Sue Mickel, Jorma Mobrin, Christer Nilsson, Rune Nilsson, Anders Nordin, Jan-Erik Nordin, Roger Oberg, Benny Odenteg, Erik Ornulf, Gunnar Overgaard, Karin Palmkvist, Fabio Peruzzi, Janne Pettersson, Gary Pollice, Tonya Prince, Leslee Probasco, Terry Quatrani, Anders Rockstrom, Walker Royce, Goran Schefte, Jeff Schuster, John Smith, Kjell Sorme, Ian Spence, Birgitta Spiridon, Fredrik Stromberg, Goran Sundelof, Per Sundquist, Per-Olof Thysselius, Mike Tudball, Karin Villers, Ctirad Vrana, Stefan Wallin, Roland Wester, Lars Wetterborg, Brian White, Lars Wiktorin, Charlotte Wranne, y Jan Wunsche.

Además, las siguientes personas han ofrecido al autor principal su apoyo personal durante años, por lo cual me siento muy agradecido: Dines Bjorner, Tore Bingefors, Dave Bulman, Larry Constantine, Goran Hemdal, Bo Hedfors, Tom Love, Nils Lennmarker, Lars-Olof Noren, Dave Thomas, y Lars-Erik Thorelli.

Por último, queremos dar las gracias en particular

A Mike Devlin, presidente de Rational Software Corporation, por su confianza en el proceso Objectory como un producto que ayudará a todos los desarrolladores de software del mundo, y por su apoyo constante en el uso de un proceso de software eficaz como guía para el desarrollo de herramientas.

Y por último, queremos dar las gracias a Philippe Kruchten, director del Proceso Unificado de Rational, y a todos los miembros del equipo del proceso Rational por haber integrado lo mejor de Objectory con las mejores técnicas de Rational y con UML, manteniendo a la vez sus valores individuales. Además, no podríamos haber conseguido este objetivo sin contar con el compromiso personal de Philippe y con su perseverancia en la tarea de construir, sencillamente, el mejor proceso de software nunca visto en el mundo.

El proceso se abre camino

A lo largo de este libro, y del resto de los libros, versiones interactivas y herramientas relacionados, el proceso de desarrollo de software se hace mayor. El Proceso Unificado tomó su inspiración de muchas fuentes, y ya está ampliamente difundido. Proporciona un sustrato común de comprensión del proceso del cual pueden partir los desarrolladores, los directores y los usuarios.

Todavía queda mucho trabajo por hacer. Los desarrolladores deben aprender maneras unificadas de trabajar. Los clientes y los directivos deben apoyarlas. Para muchas empresas de desarrollo, el adelanto es sólo potencial. Usted puede hacerlo real.

Ivar Jacobson
Palo Alto, California
Diciembre de 1998
ivar@rational.com

Referencias

[1] Peter F. Drucker, "The Discipline of Innovation," *Harvard Business Review,* May-June, 1985; reprinted Nov.-Dec. 1998, pp. 149-157.

[2] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard, *Object-Oriented Software Engineering: A Use-Case Driven Approach,* Reading, MA: Addison-Wesley, 1992.

[3] Ivar Jacobson, Maria Ericsson, and Agneta Jacobson, *The Object Advantage: Business Process Reengineering with Object Technology,* Reading, MA: Addison-Wesley, 1995.

[4] James E. Archer Jr. and Michael T. Devlin, "Rational's Experience Using Ada for Very Large Systems," *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station,* June, 1986.

[6] Philippe B. Kruchten, "The 4 + 1 View Model of Architecture", *IEEE Software,* November 1995, pp. 42-50.

[7] Grady Booch, *Object Solutions: Managing the Object-Oriented Project,* Reading, MA: Addison-Wesley, 1996.

[8] Walker Royce, *Software Project Management: A Unified Framework,* Reading, MA: Addison-Wesley, 1998.

[9] Grady Booch, *Object-Oriented Analysis and Design with Applications,* Redwood City, CA: Benjamin/Cummings, 1994.

[10] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design,* Englewood Cliffs, NJ: Prentice Hall, 1991.

[11] Grady Booch, James Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide,* Reading, MA: Addison-Wesley, 1998.

[12] James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual,* Reading, MA: Addison-Wesley, 1998.

[13] Philippe Kruchten, *The Rational Unified Process: An Introduction,* Reading, MA: Addison-Wesley, 1998.

[14] Ivar Jacobson, *Concepts for Modeling Large Real Time Systems,* Chapter 2, Dissertation, Department of Computer Systems, The Royal Institute of Technology, Stockholm, Sept. 1985.

[15] Ivar Jacobson, "Object-Orientation as a Competitive Advantage", *American Programmer,* Oct. 1992.

[16] Ivar Jacobson, "A Large Commercial Success Story with Objects, Succeeding with Objects", *Object Magazine,* May 1996.

Parte I

El Proceso Unificado de Desarrollo de Software

En esta Parte I presentamos las ideas fundamentales.

El Capítulo 1 describe el Proceso Unificado de Desarrollo de Software brevemente, centrándose en su carácter dirigido por los casos de uso, centrado en la arquitectura, iterativo e incremental. El proceso utiliza el Lenguaje Unificado de Modelado (UML), un lenguaje que produce dibujos comparables en sus objetivos a los esquemas que se utilizan desde hace mucho tiempo en otras disciplinas técnicas. El proceso pone en práctica el basar gran parte del proyecto de desarrollo en componentes reutilizables, es decir, en piezas de software con una interfaz bien definida.

El Capítulo 2 introduce las cuatro "P": personas, proyecto, producto, y proceso, y describe sus relaciones, que son esenciales para la comprensión del resto del libro. Los conceptos clave necesarios para comprender el proceso que cubre este libro son: *artefacto, modelo, trabajador* y *flujo de trabajo*.

El Capítulo 3 trata el concepto de desarrollo dirigido por casos de uso con mayor detalle. Los casos de uso son un medio para determinar los requisitos correctos y utilizarlos para conducir el proceso de desarrollo.

El Capítulo 4 describe el papel de la arquitectura en el Proceso Unificado. La arquitectura establece lo que se tiene que hacer; esquematiza los niveles significativos de la organización del software y se centra en el armazón del sistema.

El Capítulo 5 enfatiza la importancia de adoptar una aproximación *iterativa e incremental* en el desarrollo de software. En la práctica, esto se traduce en atacar primero las partes del sistema

cargadas de riesgo, obteniendo pronto una arquitectura estable, y completando después las partes más rutinarias en iteraciones sucesivas, cada una de las cuales lleva a un incremento del progreso hasta la versión final.

En la Parte II profundizaremos más. Dedicamos un capítulo a cada flujo de trabajo fundamental: requisitos, análisis, diseño, implementación y prueba. Estos flujos de trabajo se utilizarán después en la Parte III como actividades importantes en los diferentes tipos de iteración durante las cuatro fases en las que dividimos el proceso.

En la Parte III describimos en concreto cómo se lleva a cabo el trabajo en cada fase: en la de inicio para obtener un análisis del negocio, en la de elaboración para crear la arquitectura y hacer un plan, en la de construcción para aumentar la arquitectura hasta conseguir un sistema entregable, y en la de transición para garantizar que el sistema funciona correctamente en el entorno del usuario. En esta parte, volvemos a utilizar los flujos de trabajo fundamentales y los combinamos de un modo ajustado a cada fase de manera que seamos capaces de alcanzar los resultados deseados.

La intención subyacente de una organización no es, sin embargo, tener un software bueno, sino administrar sus procesos de negocio, o sistemas empotrados, de forma que le permita producir rápidamente bienes y servicios de alta calidad con costes razonables, en respuesta a las demandas del mercado. El software es el arma estratégica con el cual las empresas o los gobiernos pueden conseguir enormes reducciones en costes y tiempos de producción tanto para bienes como para servicios. Es imposible reaccionar con rapidez frente al dinamismo del mercado sin unos buenos procesos de organización establecidos. En el entorno de una economía global que opera veinticuatro horas al día, siete días a la semana, muchos de esos procesos no pueden funcionar sin el software. Un buen proceso de desarrollo de software es, por tanto, un elemento crítico para el éxito de cualquier organización.

Capítulo 1

El Proceso Unificado: dirigido por casos de uso, centrado en la arquitectura, iterativo e incremental

La tendencia actual en el software lleva a la construcción de sistemas más grandes y más complejos. Esto es debido en parte al hecho de que los computadores son más potentes cada año, y los usuarios, por tanto, esperan más de ellos. Esta tendencia también se ha visto afectada por el uso creciente de Internet para el intercambio de todo tipo de información —de texto sin formato a texto con formato, fotos, diagramas y multimedia. Nuestro apetito de software aún más sofisticado crece a medida que vemos cómo pueden mejorarse los productos de una versión a otra. Queremos un software que esté mejor adaptado a nuestras necesidades, pero esto, a su vez, simplemente hace el software más complejo. En breve, querremos más.

También lo queremos más rápido. El tiempo de salida al mercado es otro conductor importante.

Conseguirlo, sin embargo, es difícil. Nuestra demanda de software potente y complejo no se corresponde con cómo se desarrolla el software. Hoy, la mayoría de la gente desarrolla software mediante los mismos métodos que llevan utilizándose desde hace 25 años. Esto es un problema. A menos que renovemos nuestros métodos, no podremos cumplir con el objetivo de desarrollar el software complejo que se necesita actualmente.

El problema del software se reduce a la dificultad que afrontan los desarrolladores para coordinar las múltiples cadenas de trabajo de un gran proyecto de software. La comunidad de desarrolladores necesita una forma coordinada de trabajar. Necesita un proceso que integre las múltiples facetas del desarrollo. Necesita un método común, un proceso que:

- Proporcione una guía para ordenar las actividades de un equipo.
- Dirija las tareas de cada desarrollador por separado y del equipo como un todo.
- Especifique los artefactos que deben desarrollarse.
- Ofrezca criterios para el control y la medición de los productos y actividades del proyecto.

La presencia de un proceso bien definido y bien gestionado es una diferencia esencial entre proyectos hiperproductivos y otros que fracasan. (*Véase* la Sección 2.4.4 para más motivos por los cuales es necesario un proceso.) El Proceso Unificado de Desarrollo —el resultado de más de 30 años de experiencia— es una solución al problema del software. Este capítulo proporciona una visión general del Proceso Unificado completo. En posteriores capítulos, examinaremos cada elemento del proceso en detalle.

1.1. El Proceso Unificado en pocas palabras

En primer lugar, el Proceso Unificado es un proceso de desarrollo de software. Un *proceso* de desarrollo de software es el conjunto de actividades necesarias para transformar los requisitos de un usuario en un sistema software (*véase* la Figura 1.1). Sin embargo, el Proceso Unificado es más que un simple proceso; es un marco de trabajo genérico que puede especializarse para una gran variedad de sistemas software, para diferentes áreas de aplicación, diferentes tipos de organizaciones, diferentes niveles de aptitud y diferentes tamaños de proyecto.

El Proceso Unificado está *basado en componentes*, lo cual quiere decir que el sistema software en construcción está formado por **componentes** software (Apéndice A) interconectados a través de **interfaces** (Apéndice A) bien definidas.

El Proceso Unificado utiliza el *Lenguaje Unificado de Modelado* (Unified Modeling Language, UML) para preparar todos los esquemas de un sistema software. De hecho, UML es una parte esencial del Proceso Unificado —sus desarrollos fueron paralelos.

No obstante, los verdaderos aspectos definitorios del Proceso Unificado se resumen en tres frases clave —dirigido por casos de uso, centrado en la arquitectura, e iterativo e incremental. Esto es lo que hace único al Proceso Unificado.

En las tres secciones siguientes describiremos esas tres frases clave. Después, en el resto del capítulo, daremos una breve visión general del proceso: su ciclo de vida, fases, versiones, iteraciones, flujos de trabajo, y artefactos. Toda la motivación de este capítulo es la de presentar las ideas más importantes y dar una "vista de pájaro" del proceso en su totalidad. Después de este capítulo, el lector debería conocer, pero no necesariamente comprender en su totalidad, de qué trata el Proceso Unificado. El resto del libro cubrirá los detalles. En el Capítulo 2 establecemos el contexto de las cuatro "P" del desarrollo de software: personas, proyecto, producto y proceso. Después, dedicamos un capítulo a cada una de las tres ideas clave antes mencionadas. Todo esto constituirá la primera parte del libro. Las Partes II y III —el grueso del libro— describirán los distintos flujos de trabajo del proceso en detalle.

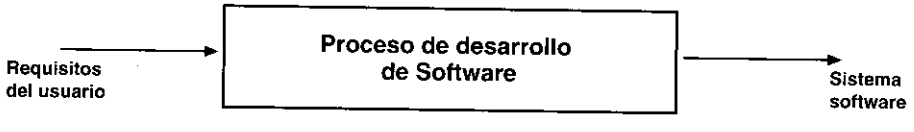


Figura 1.1. Un proceso de desarrollo de software.

1.2. El Proceso Unificado está dirigido por casos de uso

Un sistema software ve la luz para dar servicio a sus usuarios. Por tanto, para construir un sistema con éxito debemos conocer lo que sus futuros usuarios necesitan y desean.

El término *usuario* no sólo hace referencia a usuarios humanos sino a otros sistemas. En este sentido, el término *usuario* representa alguien o algo (como otro sistema fuera del sistema en consideración) que interactúa con el sistema que estamos desarrollando. Un ejemplo de interacción sería una persona que utiliza un cajero automático. Él (o ella) inserta la tarjeta de plástico, responde a las preguntas que le hace la máquina en su pantalla, y recibe una suma de dinero. En respuesta a la tarjeta del usuario y a sus contestaciones, el sistema lleva a cabo una secuencia de **acciones** (Apéndice A) que proporcionan al usuario un resultado importante, en este caso, la retirada del efectivo.

Una interacción de este tipo es un **caso de uso** (Apéndice A; *véase* también el Capítulo 3). Un caso de uso es un fragmento de funcionalidad del sistema que proporciona al usuario un resultado importante. Los casos de uso representan los requisitos funcionales. Todos los casos de uso juntos constituyen el **modelo de casos de uso** (Apéndice B; *véase* también la Sección 2.3), el cual describe la funcionalidad total del sistema. Puede decirse que una especificación funcional contesta a la pregunta: ¿Qué debe hacer el sistema?. La estrategia de los casos de uso puede describirse añadiendo tres palabras al final de esta pregunta: *¿...para cada usuario?* Estas tres palabras albergan una implicación importante. Nos fuerzan a pensar en términos de importancia para el usuario y no sólo en términos de funciones que sería bueno tener. Sin embargo, los casos de uso no son sólo una herramienta para especificar los requisitos de un sistema. También guían su diseño, implementación, y prueba; esto es, *guían el proceso de desarrollo.* Basándose en el modelo de casos de uso, los desarrolladores crean una serie de modelos de diseño e implementación que llevan a cabo los casos de uso. Los desarrolladores revisan cada uno de los sucesivos modelos para que sean conformes al modelo de casos de uso. Los ingenieros de prueba prueban la implementación para garantizar que los componentes del modelo de implementación implementan correctamente los casos de uso. De este modo, los casos de uso no sólo inician el proceso de desarrollo sino que le proporcionan un hilo conductor. *Dirigido por casos de uso* quiere decir que el proceso de desarrollo sigue un hilo —avanza a través de una serie de flujos de trabajo que parten de los casos de uso. Los casos de uso se especifican, se diseñan, y los casos de uso finales son la fuente a partir de la cual los ingenieros de prueba construyen sus casos de prueba.

Aunque es cierto que los casos de uso guían el proceso, no se desarrollan aisladamente. Se desarrollan a la vez que la arquitectura del sistema. Es decir, los casos de uso guían la arquitectura del sistema y la arquitectura del sistema influye en la selección de los casos de uso. Por tanto, tanto la arquitectura del sistema como los casos de uso maduran según avanza el ciclo de desarrollo.

1.3. El Proceso Unificado está centrado en la arquitectura

El papel de la arquitectura software es parecido al papel que juega la arquitectura en la construcción de edificios. El edificio se contempla desde varios puntos de vista: estructura, servicios, conducción de la calefacción, fontanería, electricidad, etc. Esto permite a un constructor ver una imagen completa antes de que comience la construcción. Análogamente, la arquitectura en un sistema software se describe mediante diferentes vistas del sistema en construcción.

El concepto de arquitectura software incluye los aspectos estáticos y dinámicos más significativos del sistema. La arquitectura surge de las necesidades de la empresa, como las perciben los usuarios y los inversores, y se refleja en los casos de uso. Sin embargo, también se ve influida por muchos otros factores, como la plataforma en la que tiene que funcionar el software (arquitectura hardware, sistema operativo, sistema de gestión de base de datos, protocolos para comunicaciones en red), los bloques de construcción reutilizables de que se dispone (por ejemplo, un **marco de trabajo** (Apéndice C) para interfaces gráficas de usuario), consideraciones de implantación, sistemas heredados, y requisitos no funcionales (por ejemplo, rendimiento, fiabilidad). La arquitectura es una vista del diseño completo con las características más importantes resaltadas, dejando los detalles de lado. Debido a que lo que es significativo depende en parte de una valoración, que a su vez, se adquiere con la experiencia, el valor de una arquitectura depende de las personas que se hayan responsabilizado de su creación. No obstante, el proceso ayuda al arquitecto a centrarse en los objetivos adecuados, como la comprensibilidad, la capacidad de adaptación al cambio, y la reutilización.

¿Cómo se relacionan los casos de uso y la arquitectura? Cada producto tiene tanto una función como una forma. Ninguna es suficiente por sí misma. Estas dos fuerzas deben equilibrarse para obtener un producto con éxito. En esta situación, la función corresponde a los casos de uso y la forma a la arquitectura. Debe haber interacción entre los casos de uso y la arquitectura. Es un problema del tipo "el huevo y la gallina". Por un lado, los casos de uso deben encajar en la arquitectura cuando se llevan a cabo. Por otro lado, la arquitectura debe permitir el desarrollo de todos los casos de uso requeridos, ahora y en el futuro. En realidad, tanto la arquitectura como los casos de uso deben evolucionar en paralelo.

Por tanto, los arquitectos moldean el sistema para darle una *forma*. Es esta forma, la arquitectura, la que debe diseñarse para permitir que el sistema evolucione, no sólo en su desarrollo inicial, sino también a lo largo de las futuras generaciones. Para encontrar esa forma, los arquitectos deben trabajar sobre la comprensión general de las funciones clave, es decir, sobre los casos de uso claves del sistema. Estos casos de uso clave pueden suponer solamente entre el 5 y el 10 por ciento de todos los casos de uso, pero son los significativos, los que constituyen las funciones fundamentales del sistema. De manera resumida, podemos decir que el arquitecto:

- Crea un esquema en borrador de la arquitectura, comenzando por la parte de la arquitectura que no es específica de los casos de uso (por ejemplo, la plataforma). Aunque esta parte de la arquitectura es independiente de los casos de uso, el arquitecto debe poseer una comprensión general de los casos de uso antes de comenzar la creación del esquema arquitectónico.
- A continuación, el arquitecto trabaja con un subconjunto de los casos de uso especificados, con aquellos que representen las funciones clave del sistema en desarrollo. Cada caso de uso seleccionado se especifica en detalle y se realiza en términos de **subsistemas** (Apéndice A; *véase* también Sección 3.4.4), **clases** (Apéndice A), y **componentes** (Apéndice A).
- A medida que los casos de uso se especifican y maduran, se descubre más de la arquitectura. Esto, a su vez, lleva a la maduración de más casos de uso.

Este proceso continúa hasta que se considere que la arquitectura es estable.

1.4. El Proceso Unificado es iterativo e incremental

El desarrollo de un producto software comercial supone un gran esfuerzo que puede durar entre varios meses hasta posiblemente un año o más. Es práctico dividir el trabajo en partes más

pequeñas o miniproyectos. Cada miniproyecto es una iteración que resulta en un incremento. Las iteraciones hacen referencia a pasos en el flujo de trabajo, y los incrementos, al crecimiento del producto. Para una efectividad máxima, las iteraciones deber estar *controladas*; esto es, deben seleccionarse y ejecutarse de una forma planificada. Es por esto por lo que son mini-*proyectos*.

Los desarrolladores basan la selección de lo que se implementará en una iteración en dos factores. En primer lugar, la iteración trata un grupo de casos de uso que juntos amplían la utilidad del producto desarrollado hasta ahora. En segundo lugar, la iteración trata los riesgos más importantes. Las iteraciones sucesivas se construyen sobre los artefactos de desarrollo tal como quedaron al final de la última iteración. Al ser miniproyectos, comienzan con los casos de uso y continúan a través del trabajo de desarrollo subsiguiente —análisis, diseño, implementación y prueba—, que termina convirtiendo en código ejecutable los casos de uso que se desarrollaban en la iteración. Por supuesto, un incremento no necesariamente es aditivo. Especialmente en las primeras fases del ciclo de vida, los desarrolladores pueden tener que reemplazar un diseño superficial por uno más detallado o sofisticado. En fases posteriores, los incrementos son típicamente aditivos.

En cada iteración, los desarrolladores identifican y especifican los casos de uso relevantes, crean un diseño utilizando la arquitectura seleccionada como guía, implementan el diseño mediante componentes, y verifican que los componentes satisfacen los casos de uso. Si una iteración cumple con sus objetivos —como suele suceder— el desarrollo continúa con la siguiente iteración. Cuando una iteración no cumple sus objetivos, los desarrolladores deben revisar sus decisiones previas y probar con un nuevo enfoque.

Para alcanzar el mayor grado de economía en el desarrollo, un equipo de proyecto intentará seleccionar sólo las iteraciones requeridas para lograr el objetivo del proyecto. Intentará secuenciar las iteraciones en un orden lógico. Un proyecto con éxito se ejecutará de una forma directa, sólo con pequeñas desviaciones del curso que los desarrolladores planificaron inicialmente. Por supuesto, en la medida en que se añadan iteraciones o se altere el orden de las mismas por problemas inesperados, el proceso de desarrollo consumirá más esfuerzo y tiempo. Uno de los objetivos de la reducción del riesgo es minimizar los problemas inesperados.

Son muchos los beneficios de un proceso iterativo controlado:

- La iteración controlada reduce el coste del riesgo a los costes de un solo incremento. Si los desarrolladores tienen que repetir la iteración, la organización sólo pierde el esfuerzo mal empleado de la iteración, no el valor del producto entero.
- La iteración controlada reduce el riesgo de no sacar al mercado el producto en el calendario previsto. Mediante la identificación de riesgos en fases tempranas del desarrollo, el tiempo que se gasta en resolverlos se emplea al principio de la planificación, cuando la gente está menos presionada por cumplir los plazos. En el método "tradicional", en el cual los problemas complicados se revelan por primera vez en la prueba del sistema, el tiempo necesario para resolverlos normalmente es mayor que el tiempo que queda en la planificación, y casi siempre obliga a retrasar la entrega.
- La iteración controlada acelera el ritmo del esfuerzo de desarrollo en su totalidad debido a que los desarrolladores trabajan de manera más eficiente para obtener resultados claros a corto plazo, en lugar de tener un calendario largo, que se prolonga eternamente.
- La iteración controlada reconoce una realidad que a menudo se ignora —que las necesidades del usuario y sus correspondientes requisitos no pueden definirse completamente al

principio. Típicamente, se refinan en iteraciones sucesivas. Esta forma de operar hace más fácil la adaptación a los requisitos cambiantes.

Estos conceptos —los de desarrollo dirigido por los casos de uso, centrado en la arquitectura, iterativo e incremental— son de igual importancia. La arquitectura proporciona la estructura sobre la cual guiar las iteraciones, mientras que los casos de uso definen los objetivos y dirigen el trabajo de cada iteración. La eliminación de una de las tres ideas reduciría drásticamente el valor del Proceso Unificado. Es como un taburete de tres patas. Sin una de ellas, el taburete se cae.

Ahora que hemos presentado los tres conceptos clave, es el momento de echar un vistazo al proceso en su totalidad, su ciclo de vida, artefactos, flujos de trabajo, fases e iteraciones.

1.5. La vida del Proceso Unificado

El Proceso Unificado se repite a lo largo de una serie de ciclos que constituyen la vida de un sistema, como se muestra en la Figura 1.2. Cada ciclo concluye con una **versión** del producto (Apéndice C; *véase* también el Capítulo 5) para los clientes.

Cada ciclo consta de cuatro fases: inicio[1], elaboración, construcción y transición. Cada fase (Apéndice C) se subdivide a su vez en iteraciones, como se ha dicho anteriormente. *Véase* la Figura 1.3.

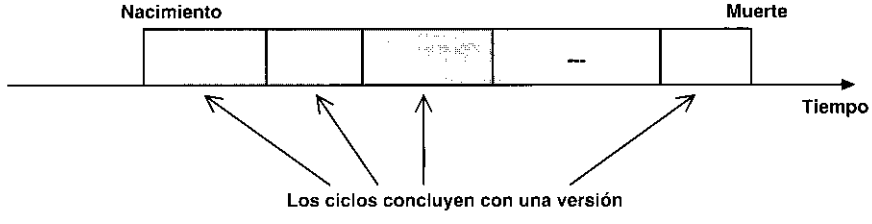


Figura 1.2. La vida de un proceso consta de ciclos desde su nacimiento hasta su muerte.

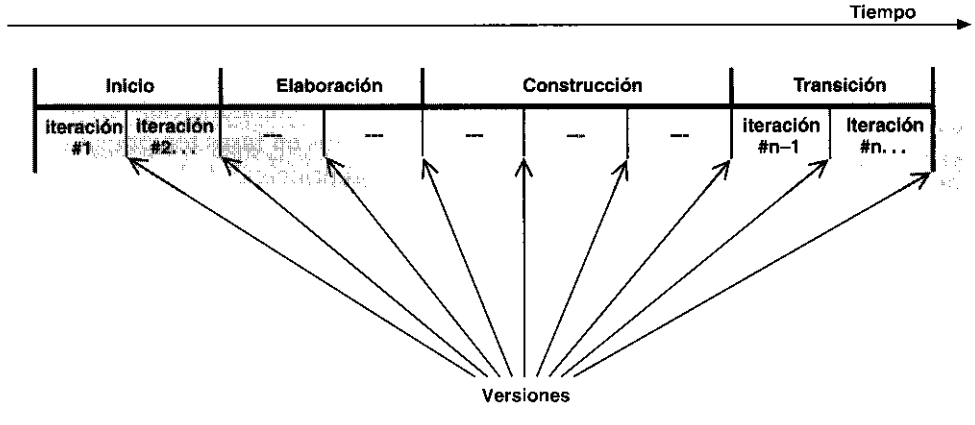


Figura 1.3. Un ciclo con sus fases e iteraciones.

[1] *N. del T.* Tambien se suele utilizar el término "fase de comienzo".

1.5.1. El producto

Cada ciclo produce una nueva versión del sistema, y cada versión es un producto preparado para su entrega. Consta de un cuerpo de código fuente incluido en componentes que puede compilarse y ejecutarse, además de manuales y otros productos asociados. Sin embargo, el producto terminado no sólo debe ajustarse a las necesidades de los usuarios, sino también a las de todos los interesados, es decir, toda la gente que trabajará con el producto. El producto software debería ser algo más que el código máquina que se ejecuta.

El producto terminado incluye los requisitos, casos de uso, especificaciones no funcionales y casos de prueba. Incluye el modelo de la arquitectura y el modelo visual —artefactos modelados con el Lenguaje Unificado de Modelado. De hecho, incluye todos los elementos que hemos mencionado en este capítulo, debido a que son esos elementos los que permiten a los interesados— clientes, usuarios, analistas, diseñadores, programadores, ingenieros de prueba, y directores —especificar, diseñar, implementar, probar y utilizar un sistema. Es más, son esos elementos los que permiten a los usuarios utilizar y modificar el sistema de generación en generación.

Aunque los componentes ejecutables sean los artefactos más importantes desde la perspectiva del usuario, no son suficientes por sí solos. Esto se debe a que el entorno cambia. Se mejoran los sistemas operativos, los sistemas de bases de datos y las máquinas que los soportan. A medida que el objetivo del sistema se comprende mejor, los propios requisitos pueden cambiar. De hecho, el que los requisitos cambien es una de las constantes del desarrollo de software. Al final, los desarrolladores deben afrontar un nuevo ciclo, y los directores deben financiarlo. Para llevar a cabo el siguiente ciclo de manera eficiente, los desarrolladores necesitan todas las representaciones del producto software (Figura 1.4):

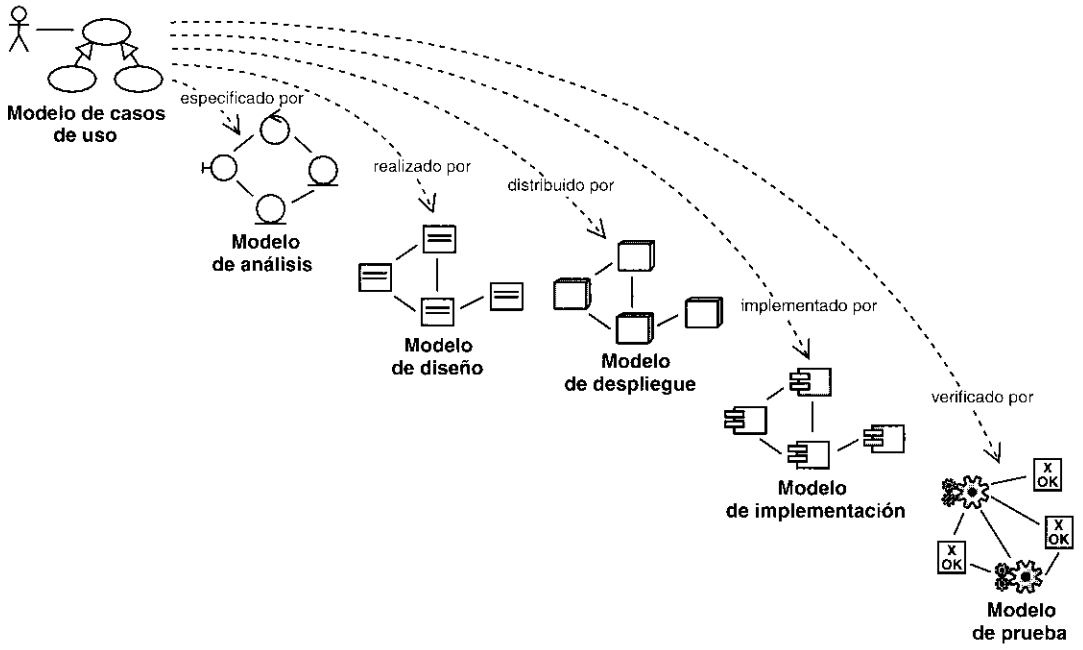


Figura 1.4. Modelo del Proceso Unificado. Existen dependencias entre muchos de los modelos. Como ejemplo, se indican las dependencias entre el modelo de casos de uso y los demás modelos.

- Un modelo de casos de uso, con todos los casos de uso y su relación con los usuarios.
- Un modelo de análisis, con dos propósitos: refinar los casos de uso con más detalle y establecer la asignación inicial de funcionalidad del sistema a un conjunto de objetos que proporcionan el comportamiento.
- Un modelo de diseño que define: (a) la estructura estática del sistema en la forma de subsistemas, clases e interfaces y (b) los casos de uso reflejados como **colaboraciones** (Apéndice A; *véase* también la Sección 3.1) entre los subsistemas, clases, e interfaces.
- Un modelo de implementación, que incluye componentes (que representan al código fuente) y la correspondencia de las clases con los componentes.
- Un modelo de despliegue[2], que define los nodos físicos (ordenadores) y la correspondencia de los componentes con esos nodos.
- Un modelo de prueba, que describe los casos de prueba que verifican los casos de uso.
- Y, por supuesto, una representación de la arquitectura.

El sistema también debe tener un modelo del dominio o modelo del negocio que describa el contexto del negocio en el que se halla el sistema.

Todos estos modelos están relacionados. Juntos, representan al sistema como un todo. Los elementos de un modelo poseen dependencias de **traza** (Apéndice A; *véase* también la Sección 2.3.7) hacia atrás y hacia adelante, mediante enlaces hacia otros modelos. Por ejemplo, podemos hacer el seguimiento de un caso de uso (en el modelo de casos de uso) hasta una realización de caso de uso (en el modelo de diseño) y hasta un caso de prueba (en el modelo de prueba). La trazabilidad facilita la comprensión y el cambio.

1.5.2. Fases dentro de un ciclo

Cada ciclo se desarrolla a lo largo del tiempo. Este tiempo, a su vez, se divide en cuatro fases, como se muestra en la Figura 1.5. A través de una secuencia de modelos, los implicados visualizan lo que está sucediendo en esas fases. Dentro de cada fase, los directores o los desarrolladores pueden descomponer adicionalmente el trabajo —en iteraciones con sus incrementos resultantes. Cada fase termina con un **hito** (Apéndice C; *véase* también Capítulo 5). Cada hito se determina por la disponibilidad de un conjunto de artefactos; es decir, ciertos modelos o documentos han sido desarrollados hasta alcanzar un estado predefinido.

Los hitos tienen muchos objetivos. El más crítico es que los directores deben tomar ciertas decisiones cruciales antes de que el trabajo pueda continuar con la siguiente fase. Los hitos también permiten a la dirección, y a los mismos desarrolladores, controlar el progreso del trabajo según pasa por esos cuatro puntos clave. Al final, se obtiene un conjunto de datos a partir del seguimiento del tiempo y esfuerzo consumido en cada fase. Estos datos son útiles en la estimación del tiempo y los recursos humanos para otros proyectos, en la asignación de los recursos durante el tiempo que dura el proyecto, y en el control del progreso contrastado con las planificaciones.

La Figura 1.5. muestra en la columna izquierda los flujos de trabajo —requisitos, análisis, diseño, implementación y prueba. Las curvas son una aproximación (no deben considerarse muy literalmente) de hasta dónde se llevan a cabo los flujos de trabajo en cada fase. Recuérdese que cada fase se divide normalmente en iteraciones o mini-proyectos. Una iteración típica pasa por los cinco flujos de trabajo, como se muestra en la Figura 1.5 para una iteración en la fase de elaboración.

---

[2] *N. del T.* También se suele utilizar el término "modelo de distribución".

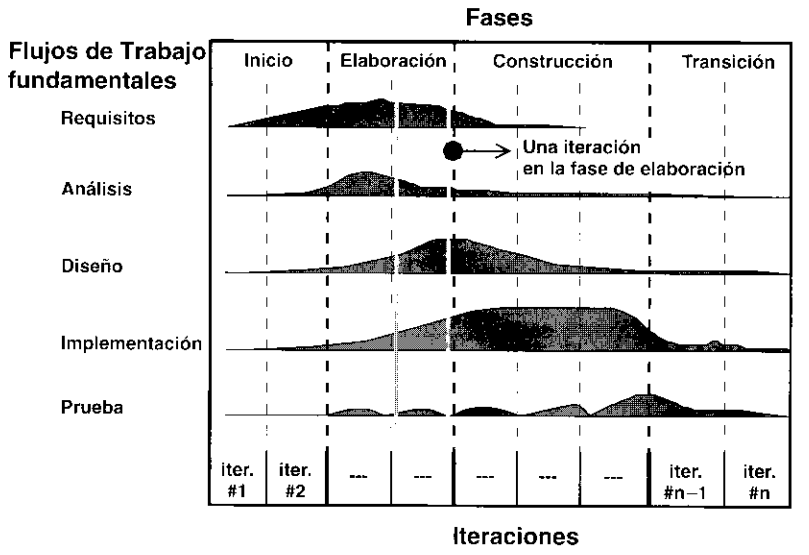


Figura 1.5. Los cinco flujos de trabajo —requisitos, análisis, diseño, implementación y prueba— tienen lugar sobre las cuatro fases: inicio, elaboración, construcción y transición.

Durante la *fase de inicio*, se desarrolla una descripción del producto final a partir de una buena idea y se presenta el análisis de negocio para el producto. Esencialmente, esta fase responde a las siguientes preguntas:

- ¿Cuáles son las principales funciones del sistema para sus usuarios más importantes?
- ¿Cómo podría ser la arquitectura del sistema?
- ¿Cuál es el plan de proyecto y cuánto costará desarrollar el producto?

La respuesta a la primera pregunta se encuentra en un modelo de casos de uso simplificado que contenga los casos de uso más críticos. Cuando lo tengamos, la arquitectura es provisional, y consiste típicamente en un simple esbozo que muestra los subsistemas más importantes. En esta fase, se identifican y priorizan los riesgos más importantes, se planifica en detalle la fase de elaboración, y se estima el proyecto de manera aproximada.

Durante la *fase de elaboración*, se especifican en detalle la mayoría de los casos de uso del producto y se diseña la arquitectura del sistema. La relación entre la arquitectura del sistema y el propio sistema es primordial. Una manera simple de expresarlo es decir que la arquitectura es análoga al esqueleto cubierto por la piel pero con muy poco músculo (el software) entre los huesos y la piel —sólo lo necesario para permitir que el esqueleto haga movimientos básicos. El sistema es el cuerpo entero con esqueleto, piel, y músculos.

Por tanto, la arquitectura se expresa en forma de vistas de todos los modelos del sistema, los cuales juntos representan al sistema entero. Esto implica que hay vistas arquitectónicas del modelo de casos de uso, del modelo de análisis, del modelo de diseño, del modelo de implementación y modelo de despliegue. La vista del modelo de implementación incluye componentes para probar que la arquitectura es ejecutable. Durante esta fase del desarrollo, se realizan los casos de uso más críticos que se identificaron en la fase de comienzo. El resultado de esta fase es una **línea base** de la arquitectura (Apéndice C; *véase* también Sección 4.4).

Al final de la fase de elaboración, el director de proyecto está en disposición de planificar las actividades y estimar los recursos necesarios para terminar el proyecto. Aquí la cuestión fundamental es: ¿son suficientemente estables los casos de uso, la arquitectura y el plan, y están los riesgos suficientemente controlados como para que seamos capaces de comprometernos al desarrollo entero mediante un contrato?

Durante la *fase de construcción* se crea el producto —se añaden los músculos (software terminado) al esqueleto (la arquitectura). En esta fase, la línea base de la arquitectura crece hasta convertirse en el sistema completo. La descripción evoluciona hasta convertirse en un producto preparado para ser entregado a la comunidad de usuarios. El grueso de los recursos requeridos se emplea durante esta fase del desarrollo. Sin embargo, la arquitectura del sistema es estable, aunque los desarrolladores pueden descubrir formas mejores de estructurar el sistema, ya que los arquitectos recibirán sugerencias de cambios arquitectónicos de menor importancia. Al final de esta fase, el producto contiene todos los casos de uso que la dirección y el cliente han acordado para el desarrollo de esta versión. Sin embargo, puede que no está completamente libre de defectos. Muchos de estos defectos se descubrirán y solucionarán durante la fase de transición. La pregunta decisiva es: ¿cubre el producto las necesidades de algunos usuarios de manera suficiente como para hacer una primera entrega?

La *fase de transición* cubre el periodo durante el cual el producto se convierte en versión beta. En la versión beta un número reducido de usuarios con experiencia prueba el producto e informa de defectos y deficiencias. Los desarrolladores corrigen los problemas e incorporan algunas de las mejoras sugeridas en una versión general dirigida a la totalidad de la comunidad de usuarios. La fase de transición conlleva actividades como la fabricación, formación del cliente, el proporcionar una línea de ayuda y asistencia, y la corrección de los defectos que se encuentren tras la entrega. El equipo de mantenimiento suele dividir esos defectos en dos categorías: los que tienen suficiente impacto en la operación para justificar una versión incrementada (versión *delta*) y los que pueden corregirse en la siguiente versión normal.

1.6. Un proceso integrado

El Proceso Unificado está basado en componentes. Utiliza el nuevo estándar de modelado visual, el Lenguaje Unificado de Modelado (UML), y se sostiene sobre tres ideas básicas —casos de uso, arquitectura, y desarrollo iterativo e incremental. Para hacer que estas ideas funcionen, se necesita un proceso polifacético, que tenga en cuenta ciclos, fases, flujos de trabajo, gestión del riego, control de calidad, gestión de proyecto y control de la configuración. El Proceso Unificado ha establecido un marco de trabajo que integra todas esas diferentes facetas. Este marco de trabajo sirve también como paraguas bajo el cual los fabricantes de herramientas y los desarrolladores pueden construir herramientas que soporten la automatización del proceso entero, de cada flujo de trabajo individualmente, de la construcción de los diferentes modelos, y de la integración del trabajo a lo largo del ciclo de vida y a través de todos los modelos.

El propósito de este libro es describir el Proceso Unificado con un énfasis particular en las facetas de ingeniería, en las tres ideas clave (casos de uso, arquitectura, y desarrollo iterativo e incremental), y en el diseño basado en componentes y el uso de UML. Describiremos las cuatro fases y los diferentes flujos de trabajo, pero no vamos a cubrir con gran detalle temas de gestión, como la planificación del proyecto, planificación de recursos, gestión de riesgos, control de configuración, recogida de métricas y control de calidad, y sólo trataremos brevemente la automatización del proceso.

Capítulo 2

Las cuatro "P" en el desarrollo de software: Personas, Proyecto, Producto y Proceso

El resultado final de un **proyecto** software (Apéndice C) es un producto que toma forma durante su desarrollo gracias a la intervención de muchos tipos distintos de personas. Un proceso de desarrollo de software guía los esfuerzos de las personas implicadas en el proyecto, a modo de plantilla que explica los pasos necesarios para terminar el proyecto. Típicamente, el proceso está automatizado por medio de una herramienta o de un conjunto de ellas. *Véase* la Figura 2.1.

A lo largo de este libro utilizaremos los términos *personas, proyecto, producto*, **proceso** (Apéndice C) y *herramientas*, que definimos a continuación:

- *Personas*. Los principales autores de un proyecto software son los arquitectos, desarrolladores, ingenieros de prueba, y el personal de gestión que les da soporte, además de los usuarios, clientes, y otros interesados. Las personas son realmente seres humanos, a diferencia del término abstracto *trabajadores*, que introduciremos más adelante.
- *Proyecto*. Elemento organizativo a través del cual se gestiona el desarrollo de software. El resultado de un proyecto es una versión de un producto.
- *Producto*. Artefactos que se crean durante la vida del proyecto, como los **modelos** (Apéndice A), código fuente, ejecutables, y documentación.
- *Proceso*. Un proceso de ingeniería de software es una definición del conjunto completo de actividades necesarias para transformar los requisitos de usuario en un producto. Un proceso es una plantilla para crear proyectos.
- *Herramientas*. Software que se utiliza para automatizar las actividades definidas en el proceso.

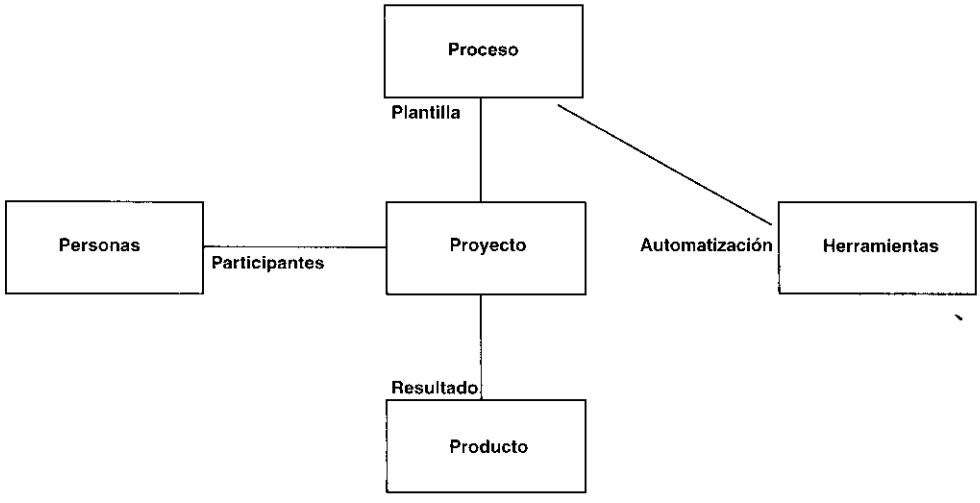


Figura 2.1. Las cuatro "P" en el desarrollo de software.

2.1. Las personas son decisivas

Hay personas implicadas en el desarrollo de un producto software durante todo su ciclo de vida. Financian el producto, lo planifican, lo desarrollan, lo gestionan, lo prueban, lo utilizan y se benefician de él. Por tanto, el proceso que guía este desarrollo debe orientarse a las personas, es decir, debe funcionar bien para las personas que lo utilizan.

2.1.1. Los procesos de desarrollo afectan a las personas

El modo en que se organiza y gestiona un proyecto software afecta profundamente a las personas implicadas en él. Conceptos como la viabilidad, la gestión del riesgo, la organización de los equipos, la planificación del proyecto y la facilidad de comprensión del proyecto tienen un papel importante:

- *Viabilidad del proyecto.* La mayoría de la gente no disfruta trabajando en proyectos que parecen imposibles —nadie quiere hundirse con la nave. Como vimos en el Capítulo 1, una aproximación iterativa en el desarrollo permite juzgar pronto la viabilidad del proyecto. Los proyectos que no son viables pueden detenerse en una fase temprana, aliviando así los problemas de moral.
- *Gestión del riesgo.* De igual forma, cuando la gente siente que los riesgos no han sido analizados y reducidos, se sienten incómodos. La exploración de los riesgos significativos en las primeras fases atenúa este problema.
- *Estructura de los equipos.* La gente trabaja de manera más eficaz en grupos pequeños de seis a ocho miembros. Un proceso que produce trabajo significativo para grupos pequeños, como el análisis de un determinado riesgo, el desarrollo de un subsistema (Apéndice A), o el llevar a cabo una iteración, proporciona esta oportunidad. Una buena arquitectura, con interfaces bien definidas (Apéndice A; *véase* también el Capítulo 9) entre subsistemas y componentes (Apéndice A; *véase* también el Capítulo 10) hace posible una división del esfuerzo de este tipo.

- *Planificación del proyecto.* Cuando la gente cree que la planificación de un proyecto no es realista, la moral se hunde —la gente no quiere trabajar a sabiendas de que por mucho que lo intenten, nunca podrán obtener los resultados deseados. Las técnicas que se utilizan en las fases de inicio y de elaboración permiten a los desarrolladores tener una buena noción de cuál debería ser el resultado del proyecto— es decir, de lo que debería hacer la versión del producto. Debido a que hay una sensación adecuada de lo que debería hacer el producto, puede crearse un plan de proyecto realista para el esfuerzo requerido, así como una definición del tiempo necesario para lograr los objetivos, atenuando el problema del "nunca acabaremos".
- *Facilidad de comprensión del proyecto.* A la gente le gusta saber lo que está haciendo; quieren tener una comprensión global. La descripción de la arquitectura proporciona una visión general para cualquiera que se encuentre implicado en el proyecto.
- *Sensación de cumplimiento.* En un ciclo de vida iterativo, la gente recibe retroalimentación frecuentemente, lo cual a su vez, hace llegar a conclusiones. La retroalimentación frecuente y las conclusiones obtenidas aceleran el ritmo de trabajo. Un ritmo de trabajo rápido combinado con conclusiones frecuentes aumenta la sensación de progreso de la gente.

2.1.2. Los papeles cambiarán

Debido a que son las personas las que ejecutan las actividades clave del desarrollo de software, es necesario un proceso de desarrollo uniforme que esté soportado por herramientas y un Lenguaje Unificado de Modelado (hoy disponible en UML) (Apéndice C), para hacer que las personas sean más eficaces. Ese proceso permitirá a los desarrolladores construir un mejor software en términos de tiempo de salida al mercado, calidad y costes. Les permite especificar los requisitos que mejor se ajusten a las necesidades de los usuarios. Les permite elegir una arquitectura que permita construir los sistemas de forma económica y puntual. Un buen proceso de software tiene otra ventaja: nos ayuda a construir sistemas más complejos. Señalamos en el primer capítulo que a medida que el mundo real se hace más complejo, los clientes requerirán sistemas software más complejos. Los procesos de negocio y su correspondiente software tendrán una vida más larga. Debido a que los cambios en el mundo real seguirán sucediendo durante estos ciclos de vida, los sistemas software tendrán que diseñarse de un modo que les permita crecer durante largos periodos de tiempo.

Para comprender y dar soporte a esos procesos de negocio más complejos y para implementarlos en software, los desarrolladores deberán trabajar con muchos otros desarrolladores. Para trabajar eficazmente en equipos cada vez más grandes, se necesita un proceso que sirva como guía. Esta guía tendrá como resultado desarrolladores "que trabajan de manera más inteligente", es decir, que limitan sus esfuerzos individuales a aquello que proporcione un valor al cliente. Un paso en esta dirección es el modelado de casos de uso, que centra el esfuerzo en lo que el usuario necesita hacer. Otro paso es una arquitectura que permitirá a los sistemas el continuar su evolución durante los años venideros. Un tercer paso es la compra o reutilización de tanto software como sea posible. Esto, a su vez, puede lograrse sólo si hay un modo consistente de integrar componentes reutilizables con elementos de nuevo desarrollo.

En los años venideros, la mayoría de los informáticos van a empezar a trabajar más estrechamente con sus objetivos, y serán capaces de desarrollar software más complejo gracias a un proceso automatizado y a los componentes reutilizables. Las personas serán decisivas en el desarrollo de software en el futuro que esperamos. En último término, el contar con las personas

adecuadas es lo que nos lleva al éxito. El problema radica en hacerlas eficaces y permitir que se dediquen a lo que sólo los seres humanos pueden hacer —ser creativos, encontrar nuevas oportunidades, utilizar la razón, comunicarse con los clientes y usuarios, y comprender rápidamente un mundo cambiante.

2.1.3. Convirtiendo "recursos" en "trabajadores"

La gente llega a ocupar muchos puestos diferentes en una organización de desarrollo de software. Su preparación para esos puestos requiere una formación y un entrenamiento preciso, seguidos de una cuidadosa asignación guiada por el análisis y por una adecuada supervisión. Una organización se enfrenta con una tarea esencial siempre que hace que una persona pase de recurso "latente" a un puesto concreto de "trabajador".

Hemos elegido la palabra **trabajador** (Apéndice C) para denominar a los puestos a los cuales se pueden asignar personas, y los cuales esas personas aceptan [4]. Un *tipo de trabajador* es un papel que un individuo puede desempeñar en el desarrollo de software, como puede ser un especificador de casos de uso, un arquitecto, un ingeniero de componentes, o un ingeniero de pruebas de integración. No utilizamos el término *rol* (en lugar de *trabajador*) principalmente por dos motivos: tiene un significado preciso y diferente en UML, y el concepto de trabajador tiene que ser muy concreto; debemos pensar en términos de trabajadores individuales como los puestos que asumen las personas. También debemos emplear el término *rol* para hablar de los papeles que cumple un trabajador. Un trabajador puede asumir roles en relación con otros trabajadores en diferentes flujos de trabajo. Por ejemplo, el trabajador ingeniero de componentes puede participar en varios flujos de trabajo y puede asumir un rol diferente en cada uno de ellos.

Cada trabajador (un trabajador concreto) es responsable de un conjunto completo de actividades, como las actividades necesarias para el diseño de un subsistema. Para trabajar eficazmente, los trabajadores necesitan la información requerida para llevar a cabo sus actividades. Necesitan comprender cuáles son sus roles en relación con los de otros trabajadores. Al mismo tiempo, si tienen que hacer su trabajo, las herramientas que emplean deben ser las adecuadas. Las herramientas no sólo deben ayudar a los trabajadores a llevar a cabo sus propias actividades, sino que también deben aislarles de la información que no les sea relevante. Para lograr estos objetivos, el Proceso Unificado describe formalmente los puestos —es decir, los trabajadores— que las personas pueden asumir en el proceso.

La Figura 2.2 ilustra cómo una persona puede ser varios trabajadores dentro de un proyecto.

Un trabajador también puede representar a un conjunto de personas que trabajan juntas. Por ejemplo, un trabajador arquitecto puede ser un grupo de arquitectura.

Cada trabajador tiene un conjunto de responsabilidades y lleva a cabo un conjunto de actividades en el desarrollo de software.

Al asignar los trabajadores a un proyecto, el jefe del proyecto debe identificar las aptitudes de las personas y casarlas con las aptitudes requeridas de los trabajadores. Ésta no es una tarea fácil, en especial si es la primera vez que se utiliza el Proceso Unificado. Se deben casar las habilidades de los recursos (las personas reales) con las competencias especificadas por los diferentes trabajadores que el proyecto necesita. Las aptitudes requeridas por algunos trabajadores pueden conseguirse mediante formación, mientras que otras sólo pueden obtenerse por medio de la experiencia. Por ejemplo, las habilidades necesarias para ser un especificador de casos de uso

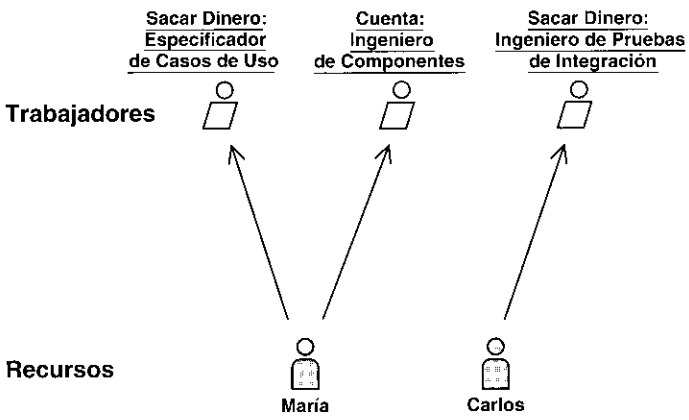


Figura 2.2. Trabajadores y recursos que los realizan.

pueden obtenerse mediante formación, pero las de un arquitecto se adquieren normalmente de la experiencia.

Una persona puede ser muchos trabajadores durante la vida de un proyecto. Por ejemplo, María puede comenzar como especificador de casos de uso, y después pasar a ser ingeniero de componentes.

Al asignar recursos, el jefe de proyecto debería minimizar el trasiego de artefactos de un recurso a otro de manera que el flujo del proceso sea tan continuo como se pueda. Por ejemplo, el ingeniero de casos de uso del caso de uso Sacar Dinero (Apéndice A; *véase* también el Capítulo 7) obtendrá gran cantidad de conocimiento sobre las responsabilidades de la clase Cuenta (Apéndice A), así que él o ella sería una elección lógica para ser el ingeniero de componentes de la clase Cuenta. La alternativa sería la de formar a una nueva persona para hacer el trabajo, lo cual es posible, pero sería menos eficiente por la pérdida de información, por el riesgo de una mala comprensión, etc.

2.2. Los proyectos construyen el producto

Un proyecto de desarrollo da como resultado una nueva versión de un producto. El primer proyecto dentro del ciclo de vida (es decir, el primer ciclo de desarrollo, llamado a veces en inglés "green-field project", *proyecto inicial o innovador* en castellano) desarrolla y obtiene el sistema o producto inicial. *Véase* [9] y [10] para una presentación más completa de la gestión de proyecto.

A través de su ciclo de vida, un equipo de proyecto debe preocuparse del cambio, las iteraciones, y el patrón organizativo dentro del cual se conduce el proyecto:

- *Una secuencia de cambio:* Los proyectos de desarrollo de sistemas obtienen productos como resultados, pero el camino hasta obtenerlos es una serie de cambios. Este hecho de la vida del proyecto debe tenerse en mente mientras los trabajadores progresan a través de las fases e iteraciones. Cada ciclo, cada fase y cada iteración modifican el sistema de ser una cosa a otra distinta. El primer ciclo de desarrollo es un caso especial que convierte el sistema de nada en algo. Cada ciclo lleva a una *versión*, y más allá de una secuencia de ciclos, el cambio continúa por *generaciones*.

- *Una serie de iteraciones:* Dentro de cada fase de un ciclo, los trabajadores llevan a cabo las actividades de la fase a través de una serie de iteraciones. Cada iteración implementa un conjunto de casos de uso relacionados o atenúa algunos riesgos. En una iteración los desarrolladores progresan a través de una serie de flujos de trabajo: requisitos, diseño, implementación y prueba. Debido a que cada iteración pasa por cada uno de esos flujos de trabajo, podemos pensar en una iteración como si fuese un *miniproyecto.*
- *Un patrón organizativo*: Un proyecto implica a un equipo de personas asignadas para lograr un resultado dentro de las restricciones del negocio, es decir, tiempo, coste y calidad. La gente trabaja como diferentes trabajadores. La idea de "proceso" es proporcionar un patrón dentro del cual las personas en su papel de trabajadores ejecutan un proyecto. Este patrón o plantilla indica los tipos de trabajadores que el proyecto necesita y los artefactos por los cuales hay que trabajar. El proceso también ofrece un montón de guías, heurísticas y prácticas de documentación que ayudan a hacer su trabajo a las personas asignadas.

2.3. El producto es más que código

En el contexto del Proceso Unificado, el producto que se obtiene es un sistema software. El término *producto* aquí hace referencia al sistema entero, y no sólo al código que se entrega.

2.3.1. ¿Qué es un sistema software?

¿Un sistema software es el código máquina, los ejecutables? Lo es, por supuesto, pero ¿qué es el código máquina? ¡Es una descripción! Una descripción en forma binaria que puede "ser leída" y "ser comprendida" por un computador.

¿Un sistema software es el código fuente? Es decir, ¿es una *descripción* escrita por programadores que puede leer y comprender un compilador? Sí, puede que ésta sea la respuesta.

Podemos seguir de esta forma haciéndonos preguntas similares sobre el diseño de un sistema software en términos de subsistemas, clases, **diagramas de interacción** (Apéndice A), **diagramas de estados** (Apéndice A), y otros artefactos. ¿Son ellos el sistema? Sí, son parte de él. ¿Qué hay de los requisitos, pruebas, venta, producción, instalación y operación? ¿Son el sistema? Sí, también son parte del sistema.

Un sistema es todos los artefactos que se necesitan para representarlo en una forma comprensible por máquinas u hombres, para las máquinas, los trabajadores y los interesados. Las máquinas son las herramientas, compiladores, u ordenadores destinatarios. Entre los trabajadores tenemos directores, arquitectos, desarrolladores, ingenieros de prueba, personal de marketing, administradores, y otros. Los interesados son los inversores, usuarios, comerciales, jefes de proyecto, directores de línea de producto, personal de producción, agencias de regulación, etc.

En este libro, utilizaremos el término *trabajador* para estas tres categorías a la vez, a menos que indiquemos explícitamente otra cosa.

2.3.2. Artefactos

Artefacto (Apéndice C) es un término general para cualquier tipo de información creada, producida, cambiada o utilizada por los trabajadores en el desarrollo del sistema. Algunos ejemplos

de artefactos son los diagramas UML y su texto asociado, los bocetos de la interfaz de usuario y los **prototipos** (Apéndice C; *véase* también los Capítulos 7 y 13), los componentes, los planes de prueba (*véase* Capítulo 11) y los procedimientos de prueba (*véase* Capítulo 11).

Básicamente, hay dos tipos de artefactos: artefactos de ingeniería y artefactos de gestión. Este libro se centra en los artefactos de ingeniería creados durante las distintas fases del proceso (requisitos, análisis, diseño, implementación y prueba).

Sin embargo, el desarrollo de software también requiere artefactos de gestión. Varios de estos artefactos tienen un tiempo de vida corto —sólo duran lo que dure la vida del proyecto. A este conjunto pertenecen artefactos como el análisis del negocio, el plan de desarrollo (incluyendo el plan de versiones e iteraciones), un plan para la asignación de personas concretas a trabajadores (es decir, a diferentes puestos o responsabilidades en el proyecto), y el diseño de las actividades de los trabajadores en el plan. Estos artefactos se describen mediante texto o diagramas, utilizando cualquier tipo de visualización requerida para especificar el compromiso asumido por el equipo de desarrollo con los inversores. Los artefactos de gestión también incluyen las especificaciones del entorno de desarrollo— el software de automatización del proceso así como la plataforma hardware necesaria para los desarrolladores y necesaria como repositorio de los artefactos de ingeniería.

2.3.3. Un sistema posee una colección de modelos

El tipo de artefacto más interesante utilizado en el Proceso Unificado es el modelo. Cada trabajador necesita una perspectiva diferente del sistema (*véase* la Figura 2.3). Cuando diseñamos el Proceso Unificado, identificamos todos los trabajadores y cada una de las perspectivas que posiblemente podrían necesitar. Las perspectivas recogidas de todos los trabajadores se estructuran en unidades más grandes, es decir, modelos, de modo que un trabajador puede tomar una perspectiva concreta del conjunto de modelos.

La construcción de un sistema es por tanto un proceso de construcción de modelos, utilizando distintos modelos para describir todas las perspectivas diferentes del sistema. La elec-

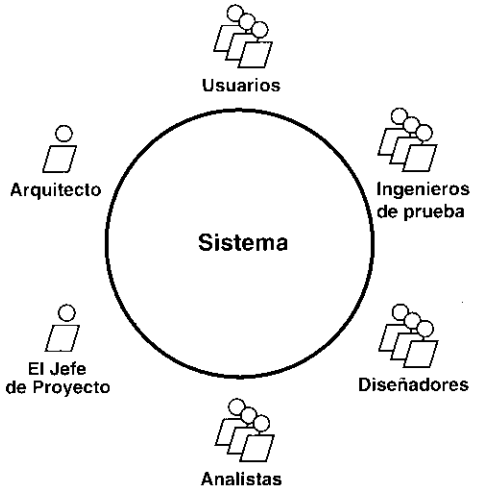


Figura 2.3. Trabajadores participando en el desarrollo de software. (Algunos son trabajadores individuales; otros son multitipos y multiobjetos.)

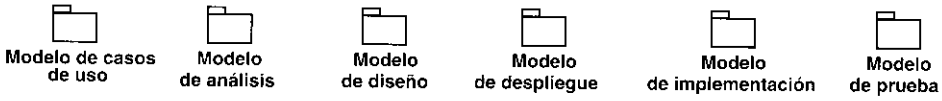


Figura 2.4. El conjunto fundamental de modelos del Proceso Unificado[1].

ción de los modelos para un sistema es una de las decisiones más importantes del equipo de desarrollo.

En el Capítulo 1 presentamos los modelos principales del Proceso Unificado (*véase* la Figura 2.4).

El Proceso Unificado proporciona un conjunto de modelos cuidadosamente seleccionado con el cual comenzar. Este conjunto de modelos hace claro el sistema para todos los trabajadores, incluyendo a los clientes, usuarios y jefes de proyecto. Fue elegido para satisfacer las necesidades de información de esos trabajadores.

2.3.4. ¿Qué es un modelo?

Un modelo es una abstracción del sistema, especificando el sistema modelado desde un cierto punto de vista y en un determinado nivel de abstracción [1]. Un punto de vista es, por ejemplo, una vista de especificación o una vista de diseño del sistema.

Los modelos son abstracciones del sistema que construyen los arquitectos y desarrolladores. Por ejemplo, los trabajadores que modelan los requisitos funcionales piensan en el sistema con los usuarios fuera de él y con los casos de uso en su interior. No se preocupan de cómo es el sistema por dentro, sólo se preocupan de lo que puede hacer para sus usurarios. Los trabajadores que construyen el diseño piensan en los elementos estructurales como subsistemas y clases; piensan en términos de cómo esos elementos funcionan en un contexto dado y cómo colaboran para proporcionar los casos de uso. Comprenden cómo funcionan esos elementos abstractos, y poseen en sus mentes una interpretación particular.

2.3.5. Cada modelo es una vista autocontenida del sistema

Un modelo es una abstracción semánticamente cerrada del sistema. Es una vista autocontenida en el sentido de que un usuario de un modelo no necesita para interpretarlo más información (de otros modelos).

La idea de ser autocontenido significa que los desarrolladores trataron de que hubiera una sola interpretación de lo que ocurrirá en el sistema cuando se dispare un evento descrito en el modelo. Además del sistema, un modelo debe describir las interacciones entre el sistema y los que le rodean. Por tanto, aparte del sistema que se está modelando, el modelo también debería incluir elementos que describan partes relevantes de su entorno, es decir, **actores** (Apéndice A; *véase* también el Capítulo 7).

---

[1] En términos de UML, estos "paquetes" representan *entidades del negocio*, "business entities" (o *unidades de trabajo*, "work units") en el Proceso Unificado, y no elementos del modelo para modelar un sistema concreto. *Véase* también la explicación de la Sección 7.1.

La mayoría de los modelos de ingeniería se definen mediante un subconjunto de UML cuidadosamente seleccionado. Por ejemplo, el modelo de casos de uso comprende a los casos de uso y los actores. Esto es básicamente lo que un lector necesita para comprenderlo. El modelo de diseño describe los subsistemas y las clases del sistema y cómo interactúan para llevar a cabo los casos de uso. Tanto el modelo de casos de uso como el modelo de diseño describen dos interpretaciones, diferentes pero mutuamente consistentes, de lo que el sistema hará dado un conjunto de estímulos externos provinientes de los actores. Son diferentes debido a que están pensados para ser utilizados por diferentes trabajadores con diferentes tareas y objetivos. El modelo de casos de uso es una vista externa del sistema, el modelo de diseño es una vista interna. El modelo de casos de uso captura los usos del sistema, mientras que el modelo de diseño representa la construcción del sistema.

2.3.6. Dentro de un modelo

Un modelo siempre identifica el sistema que está modelando. Este elemento del sistema es por tanto el contenedor de los demás elementos. El **subsistema de más alto nivel** (Apéndice B) representa al sistema en construcción. En el modelo de casos de uso, el sistema contiene casos de uso; en el modelo de diseño, contiene subsistemas, interfaces y clases. También contiene colaboraciones (Apéndice A) que identifican a todos los subsistemas o clases participantes, y puede contener más cosas, como diagramas de estado o diagramas de interacción. En el modelo de diseño cada subsistema puede ser contenedor de construcciones similares. Esto implica que hay una jerarquía de elementos en este modelo.

2.3.7. Relaciones entre modelos

Un sistema contiene todas las **relaciones** (Apéndice A) y restricciones entre elementos incluidos en diferentes modelos [1]. Por tanto un sistema no es sólo la colección de sus modelos sino que contiene también las relaciones entre ellos.

Por ejemplo, cada caso de uso en el modelo de casos de uso tiene una relación con una colaboración en el modelo de análisis (y viceversa). Esa relación se llama en UML dependencia de traza, o simplemente traza (Apéndice A). *Véase* la Figura 2.5, en la que se indican las trazas en un solo sentido.

También hay, por ejemplo, trazas entre las colaboraciones del modelo de diseño y las colaboraciones en el modelo de análisis, y entre los componentes en el modelo de implementación y los subsistemas en el modelo de diseño. Por tanto, podemos conectar elementos en un modelo a elementos en otro mediante trazas.

El hecho de que los elementos en dos modelos estén conectados no cambia lo que hacen en los modelos a los que pertenecen. Las relaciones de traza entre elementos en modelos distintos

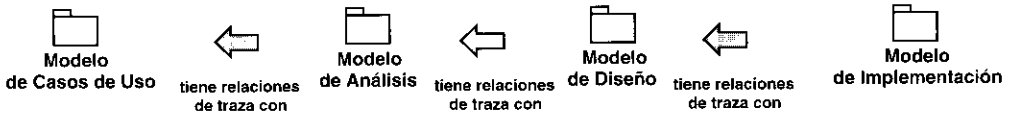


Figura 2.5. Los modelos están estrechamente enlazados unos con otros mediante trazas.

no añaden información semántica para ayudar a comprender los propios modelos relacionados; simplemente los conectan. La posibilidad de crear trazas es muy importante en el desarrollo de software por razones como la comprensibilidad y la propagación de los cambios.

2.4. El proceso dirige los proyectos

La palabra *proceso* es un término demasiado utilizado. Se utiliza en muchos contextos diferentes, como cuando decimos proceso de negocio, proceso de desarrollo y proceso software, con muchos significados diferentes. En el contexto del Proceso Unificado, el término se refiere a los procesos "de negocio" claves en una empresa de desarrollo de software, es decir, en una organización que desarrolla y da soporte al software (sobre el diseño de una empresa de desarrollo de software, *véase* [2]). En estos negocios también existen otros procesos, como el proceso de soporte, que interactúa con los usuarios de los productos, y un proceso de venta, que comienza con un pedido y entrega un producto. Sin embargo, en este libro nos centramos en el proceso de desarrollo [3].

2.4.1. El proceso: una plantilla

En el Proceso Unificado, *proceso* hace referencia a un contexto que sirve como plantilla que puede reutilizarse para crear instancias de ella. Es comparable a una clase, que puede utilizarse para crear objetos en el paradigma de la orientación a objetos. *Instancia del proceso* es un sinónimo de *proyecto*.

En este libro, un *proceso de desarrollo de software* es una definición del conjunto completo de actividades necesarias para convertir los requisitos de usuario en un conjunto consistente de artefactos que conforman un producto software, y para convertir los cambios sobre esos requisitos en un nuevo conjunto consistente de artefactos.

La palabra *requisito* se utiliza con un sentido general, refiriéndose a "necesidades". Al principio, estas necesidades no necesariamente se entienden en su totalidad. Para capturar estos requisitos, o necesidades, de una forma más completa, tenemos que comprender con mayor amplitud el negocio de los clientes y el entorno en que trabajan sus usuarios.

El resultado de valor añadido del proceso es un conjunto consistente de artefactos, una línea base que conforma una aplicación o una familia de ellas que forman un producto software.

Un proceso es una definición de un conjunto de actividades, no su ejecución.

Por último, un proceso cubre no solamente el primer ciclo de desarrollo (la primera versión) sino también los ciclos posteriores más comunes. En versiones posteriores, una instancia del proceso toma cambios incrementales en los requisitos y produce cambios incrementales en el conjunto de artefactos.

2.4.2. Las actividades relacionadas conforman flujos de trabajo

El modo en que describimos un proceso es en términos de flujos de trabajo, donde un flujo de trabajo es un conjunto de actividades. ¿Cuál es el origen de esos flujos de trabajo? No los obtenemos mediante división del proceso en un número de subprocesos más pequeños que in-

teractúan. No utilizamos diagramas de flujo tradicionales para describir cómo descomponemos el proceso en partes más pequeñas. Éstas no son formas eficaces de diseñar la estructura de flujos de trabajo.

En cambio, identificamos primero los distintos tipos de trabajadores que participan en el proceso. Después identificamos los artefactos que necesitamos crear durante el proceso para cada tipo de trabajador. Esta identificación, por supuesto, no es algo que se pueda hacer en un abrir y cerrar de ojos. El Proceso Unificado se basa en una amplia experiencia para encontrar el conjunto factible de artefactos y trabajadores. Una vez que lo hemos identificado, podemos describir cómo fluye el proceso a través de los diferentes trabajadores, y cómo ellos crean, producen y utilizan los artefactos de los demás. En la Figura 2.6 mostramos un **diagrama de actividad** (Apéndice A) que describe el flujo de trabajo en el modelado de casos de uso. Obsérvense las **"calles"** (Apéndice A) —hay una para cada trabajador—, cómo fluye el trabajo de un trabajador a otro, y cómo los trabajadores de este flujo llevan a cabo las actividades (representadas por ruedas dentadas).

A partir de aquí podemos identificar fácilmente las actividades que estos trabajadores deben realizar cuando se activan. Estas actividades por trabajador son trabajos significativos para una persona que actúe como trabajador. Además, podemos ver inmediatamente a partir de esas descripciones si algún trabajador concreto necesita participar más de una vez en el flujo de trabajo.

En otras palabras, describimos el proceso entero en partes llamadas **flujos de trabajo** (Apéndice C). En términos de UML, un flujo de trabajo es un **estereotipo** (Apéndice A) de colaboración, en el cual los trabajadores y los artefactos son los participantes. Por tanto, los trabajadores y artefactos que participan en un flujo de trabajo pueden participar (y lo suelen hacer) también en otros flujos de trabajo. Usaremos para los flujos de trabajo la notación de la Figura 2.7.

Un ejemplo de un flujo de trabajo es el flujo de trabajo de los requisitos. Incluye a los siguientes trabajadores: analista de sistemas, arquitecto, especificador de casos de uso y diseñador de interfaz de usuario. Y los siguientes artefactos: modelo de casos de uso, caso de uso, y otros. Otros ejemplos de trabajadores son: ingenieros de componentes e ingenieros de pruebas de integración. Otros ejemplos de artefacto son las **realizaciones de caso de uso** (Apéndice B; *véase* también los Capítulos 8 y 9), clases, subsistemas e interfaces.

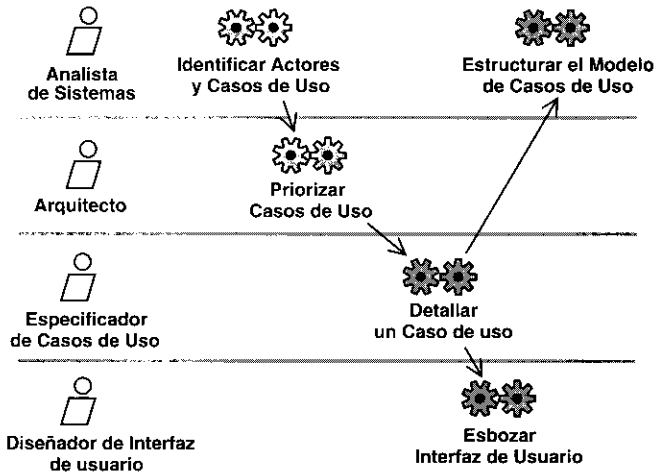


Figura 2.6. Un flujo de trabajo con trabajadores y actividades en "calles".

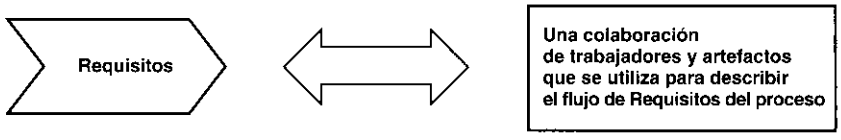


Figura 2.7. La notación "en forma de pez" es la abreviatura de un flujo de trabajo.

2.4.3. Procesos especializados

¡Ningún proceso de desarrollo de software es de aplicabilidad universal! Los procesos varían porque tienen lugar en diferentes contextos, desarrollan diferentes tipos de sistemas, y se ajustan a diferentes tipos de restricciones del negocio (plazos, costes, calidad y fiabilidad). Por consiguiente, un proceso de desarrollo de software del mundo real debe ser adaptable y configurable para cumplir con las necesidades reales de un proyecto y/o organización concreta. El Proceso Unificado se diseñó para poder ser adaptado (sobre el diseño del proceso, *véase* [6]). Es un proceso genérico, es decir, un marco de trabajo de proceso. Cada organización que utilice el Proceso Unificado en algún momento lo especializará para ajustarlo a su situación (es decir, su tipo de aplicación, su plataforma, etc.) (sobre la especialización de un proceso, *véase* [8]).

El Proceso Unificado puede especializarse para cumplir diferentes necesidades de aplicación o de organización. Al mismo tiempo, es deseable que el proceso sea, al menos, completamente consistente dentro de una organización. Esta consistencia permitirá el intercambio de componentes, una transición eficaz de personas y directivos entre proyectos, y el logro de conseguir que las métricas sean comparables.

Los factores principales que influyen en cómo se diferenciará el proceso son:

- *Factores organizativos*: La estructura organizativa, la cultura de la empresa, la organización y la gestión del proyecto, las aptitudes y habilidades disponibles, experiencias previas y sistemas software existentes.
- *Factores del dominio*: El dominio de la aplicación, procesos de negocio que se deben soportar, la comunidad de usuarios y las ofertas disponibles de la competencia.
- *Factores del ciclo de vida*: El tiempo de salida al mercado, el tiempo de vida esperado del software, la tecnología y la experiencia de las personas en el desarrollo de software, y las versiones planificadas y futuras.
- *Factores técnicos*: Lenguaje de programación, herramientas de desarrollo, base de datos, marcos de trabajo y arquitecturas "estándar" subyacentes, comunicaciones y distribución.

Éstas son las causas. ¿Qué efecto tendrán? Bien, puede decidir eliminar trabajadores y artefactos del Proceso Unificado para que se ajuste mejor a organizaciones de desarrollo menos maduras. También puede ocurrir que extienda el proceso con trabajadores o artefactos nuevos —aún no especificados— ya que esas extensiones pueden hacer el proceso más eficiente para su proyecto. También puede cambiar la forma en que piensa que un determinado artefacto debería describirse; puede imponer una estructura diferente a su definición. Nuestra experiencia es que la gente en los primeros proyectos utiliza mucho lo que sugiere el Proceso Unificado. A medida que transcurre el tiempo y ganan más experiencia, desarrollan sus propias extensiones secundarias.

¿Qué tiene el diseño del Proceso Unificado que le permite ser especializado [6]? La respuesta es simple pero no es fácil de comprender al principio. El propio Objectory está diseñado con lo que son, en efecto, objetos: casos de uso, colaboraciones y clases. Por supuesto, las clases aquí no son software sino objetos de negocio, es decir, trabajadores y artefactos. Pueden especializarse o intercambiarse con otros sin cambiar el diseño del proceso. En capítulos posteriores cuando describamos los flujos de trabajo, veremos cómo utilizamos objetos para describirlos. Esos objetos son trabajadores y artefactos.

2.4.4. Méritos del proceso

Un proceso común dentro y a lo largo de los equipos de desarrollo proporciona muchos beneficios:

- Todo el mundo en el equipo de desarrollo puede comprender lo que tiene que hacer para desarrollar el producto.
- Los desarrolladores pueden comprender mejor lo que los otros están haciendo —en momentos al principio o al final del proyecto, en proyectos similares en la misma empresa, en diferentes ubicaciones geográficas, e incluso en proyectos de otras empresas.
- Los supervisores y directores, incluso los que no entienden el código, pueden comprender lo que los desarrolladores están haciendo gracias a los esquemas de la arquitectura.
- Los desarrolladores, los supervisores y los directores pueden cambiar de proyecto o de división sin tener que aprender un nuevo proceso.
- La formación puede estandarizarse dentro de la empresa, y puede obtenerse de compañeros o de cursos breves.
- El devenir del desarrollo del software es repetible, es decir, puede planificarse y estimarse en coste con suficiente exactitud como para cumplir las expectativas.

A pesar de estas ventajas de un proceso común, hay quien insiste en que un proceso común no resuelve los "problemas verdaderamente difíciles". A esto simplemente contestamos: "por supuesto que no". Es la gente la que realmente resuelve los problemas. Pero un buen proceso ayuda a la gente a sobresalir como equipo. Comparémoslo con una organización de operaciones militares. La batalla siempre se reduce a personas que hacen cosas, pero el resultado también se decide por la eficacia de la organización.

2.5. Las herramientas son esenciales en el proceso

Las herramientas soportan los procesos de desarrollo de software modernos. Hoy, es impensable desarrollar software sin utilizar un proceso soportado por herramientas. El proceso y las herramientas vienen en el mismo paquete: las herramientas son esenciales en el proceso [5], [7].

2.5.1. Las herramientas influyen en el proceso

El proceso se ve influido fuertemente por las herramientas. Las herramientas son buenas para automatizar procesos repetitivos, mantener las cosas estructuradas, gestionar grandes cantidades de información y para guiarnos a lo largo de un camino de desarrollo concreto.

Con poco soporte por herramientas, un proceso debe sostenerse sobre gran cantidad de trabajo manual y será por tanto menos formal. En la práctica, la mayoría del trabajo formal debe posponerse a las actividades de implementación. Sin un soporte por herramientas que automatice la consistencia a lo largo del ciclo de vida, será difícil mantener actualizados los modelos y la implementación. El desarrollo iterativo e incremental será más difícil, acabará siendo inconsistente, o requerirá una gran cantidad de trabajo manual para actualizar documentos y mantener así la consistencia. Esto último podría disminuir la productividad del equipo de manera significativa. El equipo tendría que hacer manualmente todas las comprobaciones de consistencia. Esto es muy difícil, si no imposible, por lo que tendríamos muchas fisuras en los artefactos desarrollados. Y hacerlo de ese modo requeriría más tiempo de desarrollo.

Las herramientas se desarrollan para automatizar actividades, de manera completa o parcial, para incrementar la productividad y la calidad, y para reducir el tiempo de desarrollo. A medida que introducimos soporte por herramientas, obtenemos un proceso diferente, más formal. Podemos incluir nuevas actividades que sería poco práctico realizar sin herramientas. Podemos trabajar de una manera más precisa durante el ciclo de vida entero: podemos utilizar un lenguaje de modelado formal como UML para asegurar que cada modelo es consistente internamente y en relación con otros modelos. Podemos utilizar un modelo y a partir de él generar partes de otro modelo (por ejemplo, del diseño a la implementación y viceversa).

2.5.2. El proceso dirige las herramientas

El proceso, tanto si se define explícita como implícitamente, especifica la funcionalidad de las herramientas, es decir, los casos de uso de las herramientas. El hecho de que existe un proceso es, por supuesto, el único motivo para necesitar herramientas. Las herramientas están ahí para automatizar tanto como sea posible del proceso.

La capacidad de automatizar un proceso depende de que tengamos una visión clara de qué casos de uso necesita cada usuario y qué artefactos necesita manejar. Un proceso automatizado proporciona un medio eficiente de permitir el trabajo concurrente del conjunto completo de los trabajadores, y proporciona una manera de comprobar la consistencia de todos los artefactos.

Las herramientas que implementan un proceso automatizado deben ser *fáciles de usar*. Hacerlas muy manejables significa que los desarrolladores de herramientas deben considerar seriamente la forma en que se lleva a cabo el desarrollo de software. Por ejemplo, ¿cómo afrontará un trabajador una determinada tarea? ¿Cómo se dará cuenta de cómo le puede ayudar la herramienta? ¿Qué tareas hará con frecuencia y por tanto, merecen la pena ser automatizadas? ¿Qué tareas son poco frecuentes y quizá no merece la pena incluir en una herramienta? ¿Cómo puede la herramienta guiar a un trabajador para que ocupe su tiempo en las tareas importantes que solo él puede hacer, ocupándose la herramienta del resto?

Para responder a preguntas como éstas, la herramienta debe ser sencilla de comprender y manejar para los trabajadores. Es más, para que merezca la pena el tiempo que se tarda en aprenderla, debe proporcionar un incremento de productividad sustancial.

Hay razones especiales para el objetivo de la facilidad de uso. Los trabajadores deben ser capaces de probar diferentes alternativas y deberían poder tantear los diseños candidatos para cada alternativa con facilidad. Deberían ser capaces de elegir una aproximación y probarla. Si resulta ser inviable, deberían poder pasar fácilmente de esa aproximación a otra. Las herramientas de-

berían permitir a los trabajadores reutilizar tanto como sea posible; no deberían comenzar todo de nuevo para cada alternativa probada. En resumen, es esencial que las herramientas sean capaces tanto de soportar la automatización de actividades repetitivas como de gestionar la información representada por la serie de modelos y artefactos, fomentando y soportando las actividades creativas que son el núcleo fundamental del desarrollo.

2.5.3. El equilibrio entre el proceso y las herramientas

Por tanto, desarrollar un proceso sin pensar cómo se automatizará es un mero ejercicio académico. Desarrollar herramientas sin saber qué proceso (marco de trabajo) van a soportar puede ser una experimentación infructuosa. Debe haber un equilibrio entre proceso y herramientas.

Por un lado, el proceso dirige el desarrollo de las herramientas. Por otro lado, las herramientas dirigen el desarrollo del proceso. El desarrollo del proceso y su soporte por herramientas debe tener lugar de manera simultánea. En cada versión del proceso debe haber también una versión de las herramientas. En cada versión debe darse este equilibrio. Alcanzar casi el ideal de este equilibrio nos llevará varias iteraciones, y las sucesivas iteraciones deben guiarse por la retroalimentación de los usuarios entre versiones.

Esta relación proceso-herramienta es otro problema del "huevo y la gallina". ¿Cuál existe primero? En el caso de las herramientas, muchas de ellas en las últimas décadas han llegado antes de tiempo. El proceso aún no estaba bien desarrollado. En consecuencia, las herramientas no funcionaban tan bien como se esperaba en los procesos, bastante mal planificados, en los cuales los usuarios intentaron aplicarlas. Muchos de nosotros habíamos perdido la fe en las herramientas. En otras palabras, el proceso debe aprender de las herramientas, y las herramientas deben soportar un proceso bien pensado.

Queremos explicar este tema con la mayor claridad: el desarrollo con éxito de una automatización de un proceso (herramientas) no puede hacerse sin el desarrollo paralelo de un marco de trabajo de proceso en el cual vayan a funcionar las herramientas. Este tema debe quedar claro para todo el mundo. Si aún alberga una sombra de duda, pregúntese si sería posible desarrollar un soporte informático para los procesos de negocio en un banco sin conocer cuáles son.

2.5.4. El modelado visual soporta UML

Hasta aquí sólo hemos dicho que las herramientas son importantes para llevar a cabo el propósito del proceso.

Vamos a examinar un ejemplo significativo de una herramienta en el contexto del soporte del Proceso Unificado. Nuestro ejemplo es la herramienta de modelado para UML. UML es un lenguaje visual. Como tal, se le suponen características comunes en muchos productos de dibujo, como edición, dar formato, hacer *zoom*, imprimir, dar color y diseño automático. Además de esas características, UML define reglas sintácticas que especifican cómo combinar elementos del lenguaje. Por tanto, la herramienta debe ser capaz de garantizar que se cumplen esas reglas. Esta posibilidad está fuera del alcance de los productos de dibujo habituales, donde no se comprueban reglas.

Desafortunadamente, hacer cumplir reglas sintácticas de este tipo sin excepción haría que la herramienta fuese imposible de utilizar. Por ejemplo, durante la edición de un modelo, el modelo

será con frecuencia sintácticamente incorrecto, y la herramienta debe poder permitir incorecciones sintácticas en este modo. Por ejemplo, debe permitirse incluir mensajes en un **diagrama de secuencia** (Apéndice A; *véase* también el Capítulo 9) antes de que se haya definido ninguna operación para la clase.

UML incluye un cierto número de reglas semánticas que también requieren soporte. Estas herramientas pueden incluirse en la herramienta de modelado, bien como de cumplimiento obligado inmediato o como rutinas bajo demanda que recorren el modelo y comprueban si hay errores comunes, o buscan faltas de compleción semánticas o sintácticas. En resumen, la herramienta de modelado debe incorporar más que conocimiento de UML; debe permitir que los desarrolladores trabajen creativamente con UML.

Mediante el uso de UML como lenguaje estándar, el mercado experimentará un soporte por herramientas mucho mejor que el que nunca haya tenido ningún lenguaje de modelado. Esta oportunidad de un soporte mejor es debida en parte a la definición precisa de UML. También puede atribuirse al hecho de que UML es hoy un estándar industrial ampliamente utilizado. En lugar de haber fabricantes de herramientas compitiendo para dar soporte a muchos lenguajes de modelado diferentes, el juego está ahora en encontrar al que mejor soporte UML. Este nuevo juego es mejor para los usuarios y clientes del software.

UML es sólo el lenguaje de modelado. No define un proceso que diga cómo utilizar UML para desarrollar sistemas software. La herramienta de modelado no tiene que obligar a utilizar un proceso, pero si el usuario utiliza uno, la herramienta puede soportarlo.

2.5.5. Las herramientas dan soporte al ciclo de vida completo

Hay herramientas que soportan todos los aspectos del **ciclo de vida del software** (Apéndice C):

- *Gestión de requisitos*: Se utiliza para almacenar, examinar, revisar, hacer el seguimiento y navegar por los diferentes requisitos de un proyecto software. Un requisito debería tener un estado asociado, y la herramienta debería permitir hacer el seguimiento de un requisito desde otros artefactos del ciclo de vida, como un caso de uso o un caso de prueba (*véase* el Capítulo 11).
- *Modelado visual*: Se utiliza para automatizar el uso de UML, es decir, para modelar y ensamblar una aplicación visualmente. Con esta herramienta conseguimos la integración con entornos de programación y aseguramos que el modelo y la implementación siempre son consistentes.
- *Herramientas de programación*: Proporciona una gama de herramientas, incluyendo editores, compiladores, depuradores, detectores de errores y analizadores de rendimiento.
- *Aseguramiento de la calidad*: Se utiliza para probar aplicaciones y componentes, es decir, para registrar y ejecutar casos de prueba que dirigen la prueba de un IGU y de las interfaces de un componente. En un ciclo de vida iterativo, la prueba de regresión es aún más esencial que en el desarrollo convencional. La automatización de los casos de prueba es esencial para conseguir una productividad alta. Además, muchas aplicaciones también necesitan ser expuestas a pruebas de estrés y de carga. ¿Cómo responderá la arquitectura de la aplicación cuando la están utilizando 10.000 usuarios concurrentes? Queremos saber la respuesta a esta pregunta antes de entregarla al usuario número 10.000.

Además de estas herramientas orientadas a la funcionalidad, hay otras herramientas que abarcan todo el ciclo de vida. Estas herramientas incluyen las de control de versiones, gestión de la configuración, seguimiento de defectos, documentación, gestión de proyecto y automatización de procesos.

2.6. Referencias

[1] OMG Unified Modeling Language Specification. Object Management Group, Framingham, MA, 1998. Internet: www.omg.org.

[2] Ivar Jacobson, Martin Griss, and Patrik Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success,* Reading, MA: Addison-Wesley, 1997.

[3] Watts S. Humphrey, *Managing the Software Process,* Reading, MA: Addison-Wesley, 1989.

[4] Ivar Jacobson, Maria Ericsson, and Agneta Jacobson, *The Object Advantage: Business Process Reengineering with Object Technology,* Reading, MA: Addison-Wesley, 1995.

[5] Ivar Jacobson and Sten Jacobson, "Beyond methods and CASE: The software engineering process with its integral support environment", *Object Magazine,* January 1995.

[6] Ivar Jacobson and Sten Jacobson, "Designing a Software Engineering Process", *Object Magazine,* June 1995.

[7] Ivar Jacobson and Sten Jacobson, "Designing an integrated SEPSE", *Object Magazine,* September 1995.

[8] Ivar Jacobson and Sten Jacobson, "Building your own methodology by specializing a methodology framework", *Object Magazine,* November-December 1995.

[9] Grady Booch, *Object Solutions: Managing the Object-Oriented Project,* Reading, MA: Addison-Wesley, 1996.

[10] Walker Royce, *Software Project Management: A Unified Framework,* Reading, MA: Addison-Wesley, 1998.

Capítulo 3

Un proceso dirigido por casos de uso

El objetivo del Proceso Unificado es guiar a los desarrolladores en la implementación y distribución eficiente de sistemas que se ajusten a las necesidades de los clientes. La eficiencia se mide en términos de coste, calidad, y tiempo de desarrollo. El paso desde la determinación de las necesidades del cliente hasta la implementación no es trivial. En primer lugar, las necesidades del cliente no son fáciles de discernir. Esto nos obliga a que tengamos algún modo de capturar las necesidades del usuario de forma que puedan comunicarse fácilmente a todas las personas implicadas en el proyecto. Después, debemos ser capaces de diseñar una implementación funcional que se ajuste a esas necesidades. Por último, debemos verificar que las necesidades del cliente se han cumplido mediante la prueba del sistema. Debido a esta complejidad, el proceso se describe como una serie de flujos de trabajo que construyen el sistema gradualmente.

Como dijimos en el Capítulo 1, el Proceso Unificado está dirigido por los casos de uso, centrado en la arquitectura y es iterativo e incremental. En este capítulo examinaremos el aspecto dirigido-por-casos-de-uso del Proceso Unificado, presentado por primera vez en [1] y explicado en más profundidad en [2]. Los aspectos de centrado en la arquitectura e iterativo e incremental son el tema de los Capítulos 4 y 5, respectivamente. Gracias a la división de la explicación, esperamos comunicar la idea de desarrollo dirigido por casos de uso más simple y claramente. Con el mismo objetivo, restamos importancia a cómo preparar el modelo de despliegue, a cómo diseñar subsistemas con una alta cohesión, a cómo desarrollar buenos componentes de implementación (*véase* la Sección 10.3.2), y a cómo llevar a cabo otros tipos de pruebas. Estos temas

no contribuyen a la explicación de los casos de uso y a cómo estos dirigen el trabajo de desarrollo, por lo que posponemos su explicación detallada hasta la Parte II.

La Figura 3.1 muestra la serie de flujos de trabajo y modelos del Proceso Unificado. Los desarrolladores comienzan capturando los requisitos de cliente en la forma de casos de uso en el modelo de casos de uso. Después analizan y diseñan el sistema para cumplir los casos de uso, creando en primer lugar un modelo de análisis, después uno de diseño y después otro de implementación, el cual incluye todo el código, es decir, los componentes. Por último, los desarrolladores preparan un modelo de prueba que les permite verificar que el sistema proporciona la funcionalidad descrita en los casos de uso. Todos los modelos se relacionan con los otros mediante dependencias de traza. El modelo de implementación es el más formal, mientras que el modelo de casos de uso es el menos formal, en el sentido de poder ser interpretado por la máquina, es decir, las partes del modelo de implementación pueden compilarse y enlazarse produciendo ejecutables, mientras que el modelo de casos de uso se describe fundamentalmente mediante lenguaje natural.

Los casos de uso han sido adoptados casi universalmente para la captura de requisitos de sistemas software en general, y de sistemas basados en componentes en particular [6], pero los casos de uso son mucho más que una herramienta para capturar requisitos. Dirigen el proceso de desarrollo en su totalidad. Los casos de uso son la entrada fundamental cuando se identifican y especifican clases, subsistemas e interfaces (Apéndice A), cuando se identifican y especifican casos de prueba, y cuando se planifican las iteraciones del desarrollo y la integración del sistema (*véase* Capítulo 10). Para cada iteración, nos guían a través del conjunto completo de flujos de trabajo, desde la captura de requisitos, pasando por el análisis, diseño e implementación, hasta la prueba, enlazando estos diferentes flujos de trabajo.

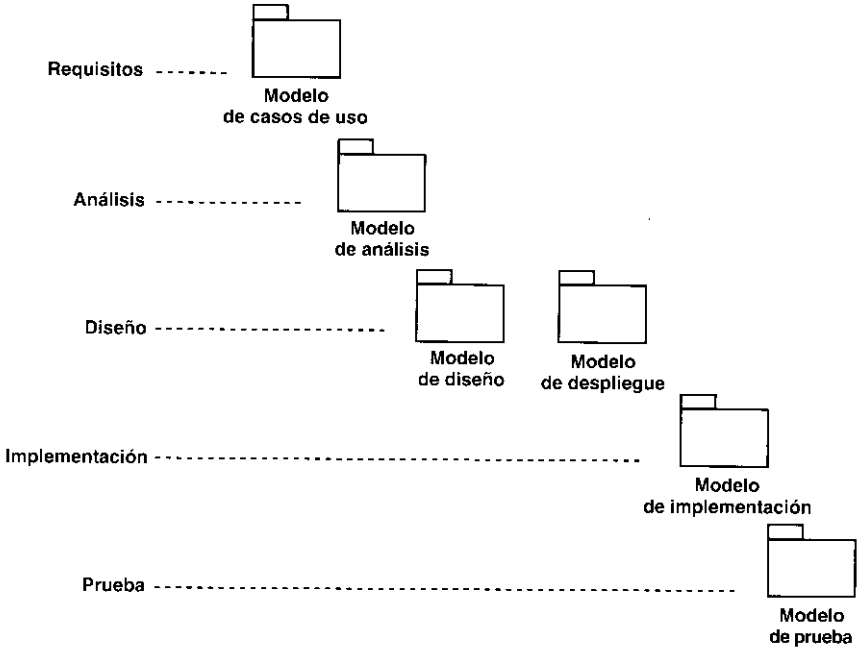


Figura 3.1. El Proceso Unificado consiste en una serie de flujos de trabajo que van desde los requisitos hasta las pruebas (de izquierda a derecha y de arriba hacia abajo). Los flujos de trabajo desarrollan modelos, desde el modelo de casos de uso hasta el modelo de prueba.

3.1. Desarrollo dirigido por casos de uso en pocas palabras

La captura de requisitos tiene dos objetivos: encontrar los verdaderos requisitos (*véase* Capítulo 6), y representarlos de un modo adecuado para los usuarios, clientes y desarrolladores. Entendemos por "verdaderos requisitos" aquellos que cuando se implementen añadirán el valor esperado para los usuarios. Con "representarlos de un modo adecuado para los usuarios, clientes y desarrolladores" queremos decir en concreto que la descripción obtenida de los requisitos debe ser comprensible por usuarios y clientes. Éste es uno de los retos principales del flujo de trabajo de los requisitos.

Normalmente, un sistema tiene muchos tipos de usuarios. Cada tipo de usuario se representa por un actor. Los actores utilizan el sistema interactuando con los casos de uso. Un caso de uso es una secuencia de acciones que el sistema lleva a cabo para ofrecer algún resultado de valor para un actor. El modelo de casos de uso está compuesto por todos los actores y todos los casos de uso de un sistema [3], [4].

Durante el análisis y el diseño, el modelo de casos de uso se transforma en un modelo de diseño a través de un modelo de análisis. En breve, tanto un modelo de análisis como un modelo de diseño son estructuras compuestas por **clasificadores** (Apéndice A) y por un conjunto de realizaciones de casos de uso (*véase* Capítulos 8 y 9) que describen cómo esa estructura realiza los casos de uso. Los clasificadores son, en general, elementos "parecidos a clases". Por ejemplo, tienen **atributos** y **operaciones** (Apéndice A), se les puede describir con **diagramas de estados** (Apéndice A), algunos de ellos pueden instanciarse, pueden ser participantes en colaboraciones, etc. UML define muchos tipos de clasificadores. Subsistemas, clases e interfaces son ejemplos de clasificadores en esta estructura. También son clasificadores los actores, casos de uso, componentes y **nodos** (Apéndice A; *véase* también la Sección 9.3.7).

El modelo de análisis es una especificación detallada de los requisitos y funciona como primera aproximación del modelo de diseño, aunque es un modelo con entidad propia. Los desarrolladores lo utilizan para comprender de manera más precisa los casos de uso descritos en el flujo de trabajo de los requisitos, refinándolos en forma de colaboraciones entre clasificadores conceptuales (diferentes de los clasificadores de diseño que serán objeto de implementación). El modelo de análisis también se utiliza para crear un sistema robusto y flexible (incluyendo una arquitectura) que emplea la reutilización de componentes de manera considerable. El modelo de análisis es diferente del de diseño en que es un modelo conceptual, en lugar de ser un esquema de la implementación. El modelo de análisis puede ser transitorio y sobrevivir sólo al primer par de iteraciones. Sin embargo, en algunos casos, especialmente para sistemas grandes y complejos, el modelo de análisis debe mantenerse durante toda la vida del sistema. Es estos casos, existe una relación directa (mediante dependencias de traza) entre una realización de caso de uso en el modelo de análisis y la correspondiente realización de caso de uso en el modelo de diseño. Cada elemento del modelo de análisis es trazable a partir de elementos del modelo de diseño que lo realizan. (El modelo de análisis, su propósito y su relación con el modelo de diseño se explican en profundidad en las Secciones 8.1 a 8.3).

El modelo de diseño posee las siguientes características:

- El modelo de diseño es jerárquico, pero también contiene relaciones que atraviesan la jerarquía. Las relaciones son las habituales en UML: **asociaciones, generalizaciones,** y **dependencias** (Apéndice A).

- Las realizaciones de los casos de uso son estereotipos de colaboraciones. Una colaboración representa cómo los clasificadores participan y desempeñan papeles en hacer algo útil, como la realización de un caso de uso.
- El modelo de diseño es también un esquema de la implementación. Existe una correspondencia directa entre subsistemas del modelo de diseño y componentes del modelo de implementación.

Los desarrolladores crean un modelo de análisis que utiliza el modelo de casos de uso como entrada [5]. Cada caso de uso en el modelo de casos de uso se traducirá en una realización de caso de uso en el modelo de análisis. La dualidad caso de uso/realización de caso de uso es la base de una trazabilidad directa entre los requisitos y el análisis. Tomando los casos de uso uno a uno, los desarrolladores pueden identificar las clases que participan en la realización de los casos de uso. Por ejemplo, el caso de uso Sacar Dinero puede realizarse por parte de las clases de análisis Retirada de Efectivo, Cuenta, Cajero, y otras que no necesitamos identificar para esta explicación. Los desarrolladores asignan **responsabilidades** (Apéndice A) definidas en el caso de uso a responsabilidades de las clases. En nuestro ejemplo, la clase Cuenta podría tener responsabilidades como "restar cantidad de la cuenta". De esta forma, podemos garantizar que obtenemos un conjunto de clases que juntas realizan los casos de uso y son verdaderamente necesarias para los usuarios.

Después, los desarrolladores diseñan las clases y las realizaciones de casos de uso para obtener mejor partido de los productos y tecnologías (*object request brokers*[1], kits de construcción de IGU, y sistemas de gestión de base de datos) que se utilizarán para implementar el sistema. Las clases de diseño se agrupan en subsistemas, y pueden definirse interfaces entre ellos. Los desarrolladores también preparan el modelo de despliegue, donde definen la organización física del sistema en términos de nodos de cómputo, y verifican que los casos de uso pueden implementarse como componentes que se ejecutan en esos nodos.

A continuación, los desarrolladores implementan las clases diseñadas mediante un conjunto de ficheros (código fuente) en el modelo de implementación, a partir de los cuales se pueden producir (compilar y enlazar) ejecutables, como DLL's, JavaBeans, y componentes ActiveX. Los casos de uso ayudan a los desarrolladores a determinar el orden de implementación e integración de los componentes.

Por último, durante el flujo de trabajo de prueba los ingenieros de prueba verifican que el sistema implementa de verdad la funcionalidad descrita en los casos de uso y que satisface los requisitos del sistema. El modelo de prueba se compone de casos de prueba (y otros elementos que explicaremos más adelante en el Capítulo 11). Un caso de prueba define una colección de entradas, condiciones de ejecución, y resultados. Muchos de los casos de prueba se pueden obtener directamente de los casos de uso y por tanto tenemos una dependencia de traza entre el caso de prueba y el caso de uso correspondiente. Esto significa que los ingenieros de prueba verificarán que el sistema puede hacer lo que los usuarios quieren que haga, es decir, que ejecuta los casos de uso. Cualquiera que haya probado software en el pasado en realidad ha probado casos de uso — incluso si su trabajo no los describía en esos términos en su momento [8]. Lo novedoso y diferente del Proceso Unificado es que la prueba puede planificarse al principio del ciclo de desarrollo. Tan pronto como se hayan capturado los casos de uso, es posible especificar los casos de prueba (pruebas de "caja negra") y determinar el orden en el cual

[1] *N. del T.* Por su amplia difusión en la industria del software, mantenemos el término original en inglés.

realizarlos, integrarlos y probarlos. Más adelante, según se vayan realizando los casos de uso en el diseño, pueden detallarse las pruebas de los casos de uso (pruebas de "caja blanca"). Cada forma de ejecutar un caso de uso —es decir, cada camino a través de una realización de un caso de uso— es un caso de prueba candidato.

Hasta aquí, hemos presentado el Proceso Unificado como una secuencia de pasos, muy parecida al antiguo método en cascada. Pero lo hemos hecho así sólo para mantener la simplicidad hasta este momento. En el Capítulo 5 veremos cómo estos pasos pueden desarrollarse de una forma mucho más interesante mediante una aproximación iterativa e incremental. Realmente, lo que hemos descrito hasta aquí es una sola iteración. Un proyecto de desarrollo completo será una serie de iteraciones, en la cual cada una de ellas (con la posible excepción de la primera) consiste en una pasada a través de los flujos de trabajo de requisitos, análisis, diseño, implementación y prueba.

Vamos a examinar con más detalle los beneficios de los casos de uso antes de estudiar los flujos de trabajo más en profundidad.

3.2. ¿Por qué casos de uso?

Existen varios motivos por los cuales los casos de uso son buenos, se han hecho populares y se han adoptado universalmente [6]. Las dos razones fundamentales son:

- Proporcionan un medio sistemático e intuitivo de capturar **requisitos funcionales** (Apéndice C; *véase* también los Capítulos 6 y 7) centrándose en el valor añadido para el usuario.
- Dirigen todo el proceso de desarrollo debido a que la mayoría de las actividades como el análisis, diseño y prueba se llevan a cabo partiendo de los casos de uso. El diseño y la prueba pueden también planificarse y coordinarse en términos de casos de uso. Esta característica es aún más evidente cuando la arquitectura se ha estabilizado en el proyecto, después del primer conjunto de iteraciones.

3.2.1. Para capturar los requisitos que aportan valor añadido

Según Karl Wieger, "la perspectiva que proporcionan los casos de uso refuerza el objetivo último de la ingeniería del software: la creación de productos que permitan a los clientes realizar un trabajo útil." [9]. Existen varias razones por las cuales esto es cierto.

En primer lugar, la idea de caso de uso permite la identificación del software que cumple con los objetivos del usuario. Los casos de uso son las funciones que proporciona un sistema para añadir valor a sus usuarios. Tomando la perspectiva de cada tipo de usuario, podemos capturar los casos de uso que necesitan para hacer su trabajo. Por otro lado, si comenzamos pensando en un conjunto de buenas funciones del sistema sin pensar en los casos de uso que emplean los usuarios concretos, será difícil decir si esas funciones son importantes o incluso si son buenas. ¿A quién ayudan? ¿Qué necesidades del negocio satisfacen? ¿Cuánto valor añaden al negocio?

Los mejores casos de uso son aquellos que añaden el mayor valor al negocio que implanta el sistema. Un caso de uso que aporta un valor negativo o que permite hacer al usuario cosas que no debería poder hacer no es un caso de uso. Podríamos llamarlo un "caso de abuso", ya que especifica formas de utilizar el sistema que deben prohibirse. Un ejemplo de un caso de uso de este tipo sería permitir a un cliente de un banco transferir dinero de la cuenta de otro a la suya. Los casos de uso que aportan poco o ningún valor se utilizarán con menos frecuencia; son "casos sin-uso" superfluos.

Como dijimos en el Capítulo 1, mucha gente ha llegado a la conclusión de que preguntarse a sí mismos sobre qué quieren que haga el sistema no les ayuda a obtener las respuestas correctas. En su lugar, obtienen una lista de funciones del sistema, que a primera vista puede parecer valiosa, pero cuando se examina más en profundidad se ve que no necesariamente está relacionada con las necesidades del usuario. Puede parecer que la estrategia de los casos de uso que reformula la pregunta añadiendo tres palabras al final —¿qué se quiere que haga el sistema *para cada usuario*?— es sólo un poco diferente, pero obtiene un resultado muy distinto. Nos mantiene centrados en la comprensión de cómo el sistema debe dar soporte a cada uno de sus usuarios. Nos guía en la identificación de las funciones que cada usuario necesita. También nos ayuda a abstenernos de sugerir funciones superfluas que ninguno de los usuarios necesita.

Además, los casos de uso son intuitivos. Los usuarios y los clientes no tienen que aprender una notación compleja. En su lugar, se puede usar simple castellano (es decir, lenguaje natural) la mayor parte del tiempo, lo que hace más fácil la lectura de los casos de uso y la propuesta de cambios.

La captura de los casos de uso implica a los usuarios, a los clientes y a los desarrolladores. Los usuarios y los clientes son los expertos en los requisitos. El papel de los desarrolladores es el de facilitar las discusiones y ayudar a los usuarios y a los clientes a comunicar sus necesidades.

El modelo de casos de uso se utiliza para conseguir un acuerdo con los usuarios y clientes sobre qué debería hacer el sistema para los usuarios. Podemos pensar en el modelo de casos de uso como en una especificación completa de todas las formas posibles de utilizar el sistema (los casos de uso). Esta especificación puede utilizarse como parte del contrato con el cliente. El modelo de casos de uso nos ayuda a delimitar el sistema definiendo todo lo que debe hacer para sus usuarios. Una técnica interesante para estructurar los casos de uso se encuentra en [12], y [11] proporciona una buena introducción general a los casos de uso.

3.2.2. Para dirigir el proceso

Como hemos dicho, el que un proyecto de desarrollo esté dirigido por los casos de uso significa que progresa a través de una serie de flujos de trabajo que se inician a partir de los casos de uso. Los casos de uso ayudan a los desarrolladores a encontrar las clases. Las clases se recogen de las descripciones de los casos de uso a medida que las leen los desarrolladores, buscando clases que sean adecuadas para la realización de los casos de uso[2]. Los casos de uso también nos

[2] Esto es una simplificación. En realidad, cada caso de uso puede tocar varias clases (subsistemas, etc.) que ya se han desarrollado, de modo que modificaremos los casos de uso para que se ajusten a esos bloques de construcción reutilizables. Esto se explicará en la Sección 4.3.

Figura 3.2. Los casos de uso enlazan los flujos de trabajo fundamentales. La elipse sombreada en el fondo simboliza cómo los casos de uso enlazan esos flujos de trabajo.

ayudan a desarrollar interfaces de usuario que hacen más fácil a los usuarios el desempeño de sus tareas. Más adelante, las realizaciones de los casos de uso se prueban para verificar que las **instancias** (Apéndice A) de las clases pueden llevar a cabo los casos de uso [8].

Los casos de uso no sólo inician un proceso de desarrollo sino que lo enlazan, como se muestra en la Figura 3.2.

También tenemos que estar seguros de que capturamos los casos de uso correctos de forma que los usuarios obtengan los que realmente quieren. La mejor forma de conseguir esto es, por supuesto, hacer un buen trabajo durante el flujo de los requisitos. Pero con frecuencia no es suficiente. Un sistema en ejecución nos permite una validación adicional de los casos de uso con las necesidades reales del usuario.

Los casos de uso ayudan a los jefes de proyecto a planificar, asignar y controlar muchas de las tareas que los desarrolladores realizan. Por cada caso de uso, un jefe de proyecto puede identificar unas cuantas tareas. Cada caso de uso a especificar es una tarea, cada caso de uso a diseñar es una tarea, y cada caso de uso a probar es una tarea. El jefe de proyecto puede incluso estimar el esfuerzo y el tiempo necesario para llevar a cabo esas tareas. Las tareas identificadas a partir de los casos de uso ayudan a los jefes a estimar el tamaño del proyecto y los recursos necesarios. Entonces esas tareas pueden asignarse a desarrolladores concretos que se hacen responsables de ellas. Un jefe de proyecto podría hacer responsable de la especificación de cinco casos de uso a una persona durante el flujo de requisitos, a otra responsable de diseñar tres casos de uso y a un tercero responsable de especificar los casos de prueba para dos casos de uso.

Los casos de uso son un importante mecanismo para dar soporte a la trazabilidad a través de todos los modelos. Un caso de uso en el modelo de requisitos es trazable a su realización en el análisis y en el diseño, a todas las clases participantes en su realización, a componentes (indirectamente), y finalmente, a los casos de prueba que lo verifican. Esta trazabilidad es un aspecto importante de la gestión de un proyecto. Cuando se cambia un caso de uso, las realizaciones, clases, componentes y casos de prueba correspondientes tienen que comprobarse para ser actualizadas. De igual forma, cuando un componente de fichero(código fuente) se modifica, las clases, casos de uso y casos de prueba correspondientes que se ven afectados también deben comprobarse (*véase* [10]). La trazabilidad entre los casos de uso y el resto de los elementos del modelo hace más fácil mantener la integridad del sistema y conservar actualizado al sistema en su conjunto cuando tenemos requisitos cambiantes.

3.2.3. Para idear la arquitectura y más...

Los casos de uso nos ayudan a llevar a cabo el desarrollo iterativo. Cada iteración, excepto quizá la primera de todas en un proyecto, se dirige por los casos de uso a través de todos los flujos de trabajo, de los requisitos al diseño y a la prueba, obteniendo un **incremento** (Apéndice C). Cada incremento del desarrollo es por tanto una realización funcional de un conjunto de casos de uso. En otras palabras, en cada iteración se identifican e implementan unos cuantos casos de uso.

Los casos de uso también nos ayudan a idear la arquitectura. Mediante la selección del conjunto correcto de casos de uso —los casos de uso significativos arquitectónicamente— para llevarlo a cabo durante las primeras iteraciones, podemos implementar un sistema con una arquitectura estable, que pueda utilizarse en muchos ciclos de desarrollo subsiguientes. Volveremos a este tema en el Capítulo 4.

Los casos de uso también se utilizan como punto de partida para escribir el manual de usuario. Ya que cada caso de uso describe una forma de utilizar el sistema, son el punto de partida ideal para explicar cómo puede el usuario interactuar con el sistema.

Mediante la estimación de la frecuencia de uso de los diferentes caminos en los casos de uso, podemos estimar qué caminos necesitan el mejor rendimiento. Esas estimaciones pueden utilizarse para dimensionar la capacidad de proceso del hardware subyacente o para optimizar el diseño de la base de datos para ciertos usos. Pueden utilizarse estimaciones parecidas para conseguir una buena facilidad de uso, seleccionando los caminos más importantes para centrarnos en ellos durante el diseño de la interfaz de usuario.

3.3. La captura de casos de uso

Pasamos ahora a dar una visión general de cómo se desarrolla el trabajo a través de todos los flujos de trabajo. Nos centraremos, como se dijo anteriormente, en el hecho de que está dirigido-por-casos-de-uso. En la siguiente sección nos centramos en la captura de requisitos funcionales en la forma de casos de uso, aunque también es necesario capturar otros tipos de requisitos.

Durante el flujo de trabajo de los requisitos identificamos las necesidades de usuarios y clientes como requisitos. Los requisitos funcionales se expresan como casos de uso en un modelo de casos de uso, y los demás requisitos o bien se "adjuntan" a los casos de uso a los que afectan, o bien se guardan en una lista aparte o se describen de alguna otra forma.

3.3.1. El modelo de casos de uso representa los requisitos funcionales

El modelo de casos de uso ayuda al cliente, a los usuarios y a los desarrolladores a llegar a un acuerdo sobre cómo utilizar el sistema. La mayoría de los sistemas tienen muchos tipos de usuarios. Cada tipo de usuario se representa mediante un *actor*. Los actores utilizan el sistema al interactuar con los casos de uso. Todos los actores y casos de uso del sistema forman un modelo de casos de uso [2], [3]. Un **diagrama de casos de uso** (Apéndice A; *véase* también la Sección 7.4.1) describe parte del modelo de casos de uso y muestra un conjunto de casos de uso y actores con una asociación entre cada par actor/caso de uso que interactúan (*véase* la Figura 3.3).

Ejemplo **Un modelo de casos de uso para un sistema de cajero automático (CA)**

El actor Cliente de Banco utiliza un sistema CA para retirar e ingresar dinero desde y en cuentas, y para transferir dinero entre cuentas. Esto se representa por los tres casos de uso que se muestran en la Figura 3.3, que poseen asociaciones con el actor para indicar su interacción.

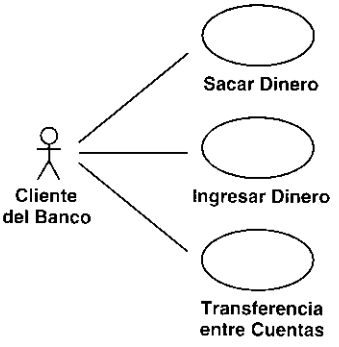


Figura 3.3. Un ejemplo de diagrama de casos de uso con un actor y tres casos de uso.

3.3.2. Los actores son el entorno del sistema

No todos los actores representan a personas. Pueden ser actores otros sistemas o hardware externo que interactuará con el sistema. Cada actor asume un conjunto coherente de papeles cuando interactúa con el sistema. Un usuario físico puede actuar como uno o varios actores, desempeñando los papeles de esos actores en su interacción con el sistema. Varios usuarios concretos pueden actuar como diferentes ocurrencias del mismo actor. Por ejemplo, puede haber miles de personas que actúan como el actor Cliente de Banco.

Los actores se comunican con el sistema mediante el envío y recepción de **mensajes** (Apéndice A) hacia y desde el sistema según éste lleva a cabo los casos de uso. A medida que definimos lo que hacen los actores y lo que hacen los casos de uso, trazaremos una clara separación entre las responsabilidades de los actores y las del sistema. Esta separación nos ayuda a delimitar el alcance del sistema.

Podemos encontrar y especificar todos los actores examinando a los usuarios que utilizarán el sistema y a otros sistemas que deben interactuar con él. Cada categoría de usuarios o sistemas que interactúan se representan por tanto como actores.

3.3.3. Los casos de uso especifican el sistema

Los casos de uso están diseñados para cumplir los deseos del usuario cuando utiliza el sistema. El modelo de casos de uso captura todos los requisitos funcionales del sistema. Definiremos un caso de uso de manera precisa como sigue:

> *Un caso de uso especifica una secuencia de acciones, incluyendo variantes, que el sistema puede llevar a cabo, y que producen un resultado observable de valor para un actor concreto.*

Identificamos los casos de uso examinando cómo los usuarios necesitan utilizar el sistema para realizar su trabajo (para un modo de estructurar los casos de uso basado en objetivos, *véase* [10]). Cada uno de esos modos de utilizar el sistema que añaden valor al usuario es un caso de uso candidato. Estos candidatos se ampliarán, se cambiarán, se dividirán en casos de uso más pequeños, o se integrarán en casos de uso más completos. El modelo de casos de uso está casi terminado cuando recoge todos los requisitos funcionales correctamente de un modo

que puedan comprender los clientes, usuarios y desarrolladores. La secuencia de acciones realizada por un caso de uso durante su operación (es decir, una instancia del caso de uso) es un camino específico a través del caso de uso. Puede haber muchos caminos, muchos de ellos muy parecidos —son variantes de la ejecución de la secuencia de acciones especificada en el caso de uso. Para hacer que un modelo de casos de uso sea más comprensible, podemos agrupar descripciones de caminos variantes parecidas en un solo caso de uso. Cuando decimos que identificamos y describimos un caso de uso, realmente estamos diciendo que identificamos y describimos los diferentes caminos que es práctico definir como un solo caso de uso.

Ejemplo **El caso de uso sacar dinero**

La secuencia de acciones para un camino a través de este caso de uso es (de forma muy simplificada):

El Cliente del Banco se identifica.

El Cliente del Banco elige de qué cuenta sacar dinero y especifica qué cantidad.

El sistema deduce la cantidad de la cuenta y entrega el dinero.

Los casos de uso también se utilizan como "contenedores" de los **requisitos no funcionales** (Apéndice C; *véase* también el Capítulo 6), tales como los requisitos de rendimiento, disponibilidad, exactitud y seguridad que son específicos de un caso de uso. Por ejemplo, puede asociarse al caso de uso Sacar Dinero el siguiente requisito: el tiempo de respuesta para un Cliente de Banco medido desde la selección de la cantidad a sacar hasta la entrega de los recibos debe ser menor de 30 segundos en el 95 por ciento de los casos.

Resumiendo, todos los requisitos funcionales quedan atados mediante casos de uso, y muchos de los requisitos no funcionales pueden asociarse a los casos de uso. De hecho, el modelo de casos de uso es un vehículo para organizar los requisitos de un modo fácil de gestionar. Clientes y usuarios pueden comprenderlo y utilizarlo para comunicar sus necesidades de un modo consistente y no redundante. Los desarrolladores pueden dividir el trabajo de captura de requisitos entre ellos, y después utilizar los resultados (fundamentalmente los casos de uso) como entrada para el análisis, diseño, implementación y prueba del sistema.

3.4. Análisis, diseño e implementación para realizar los casos de uso

Durante el análisis y el diseño, transformamos el modelo de casos de uso mediante un modelo de análisis en un modelo de diseño, es decir, en una estructura de clasificadores y realizaciones de casos de uso. El objetivo es realizar los casos de uso de una forma económica de manera que el sistema ofrezca un rendimiento adecuado y pueda evolucionar en el futuro.

En esta sección, veremos cómo pasar por un modelo de análisis para desarrollar un diseño que realice los casos de uso. En los Capítulos 4 y 5, examinaremos cómo la arquitectura y el desarrollo iterativo e incremental nos ayudan a desarrollar un sistema económico que pueda incorporar nuevos requisitos.

3.4.1. Creación del modelo de análisis a partir de los casos de uso

El modelo de análisis crece incrementalmente a medida que se analizan más y más casos de uso. En cada iteración, elegimos un conjunto de casos de uso y los reflejamos en el modelo de análisis. Construimos el sistema como una estructura de clasificadores (clases de análisis) y relaciones entre ellas. También describimos las colaboraciones que llevan a cabo los casos de uso, es decir, las realizaciones de los casos de uso. Después, en la siguiente iteración, tomamos otro conjunto de casos de uso para desarrollar, y los añadimos a la iteración anterior. En las Secciones 5.3 y 12.6, discutiremos cómo identificar y seleccionar los conjuntos más "importantes" de casos de uso para las primeras iteraciones con el objetivo de construir pronto una arquitectura estable dentro del ciclo de vida del sistema.

Una forma práctica de trabajar es identificar y describir en primer lugar los casos de uso para una iteración, después leer la descripción de cada caso de uso (ejemplificada en la Sección 3.3.3), y proponer los clasificadores y asociaciones necesarios para llevar a cabo el caso de uso. Lo hacemos para todos los casos de uso de una iteración a modo de esfuerzo coordinado. Dependiendo de dónde nos encontremos en el ciclo de vida y del tipo de iteración que estemos tratando, puede que haya ya una arquitectura establecida para guiarnos en la identificación de nuevos clasificadores y en la reutilización de los existentes (*véase* la Sección 4.3). Cada clasificador desempeña uno o varios roles en una realización de caso de uso. Cada papel de un clasificador especifica las responsabilidades, atributos y demás que el clasificador debe tener para participar en la realización del caso de uso. Podemos pensar en esos roles como en "embriones" de clasificadores. De hecho, un rol en UML es un clasificador en sí mismo. Por ejemplo, podemos pensar que un rol de una clase es una vista de la clase. Por tanto, incluye lo que incluya la clase, es decir, sus responsabilidades, atributos y demás —pero sólo aquellos que sean de interés para el rol en una realización de caso de uso. Otra forma de describir un rol de una clase es verlo como lo que queda de la clase cuando se le pone un filtro, un filtro que bloquea el resto de los roles de la clase que no tienen responsabilidades comunes. Este concepto de rol se describe brevemente en este capítulo, aunque por simplicidad no se desarrolla completamente hasta la Parte II de este libro donde se explican todos los clasificadores en detalle.

Ejemplo **La realización de un caso de uso en el modelo de análisis**

En la Figura 3.4 describimos cómo se lleva a cabo el caso de uso Sacar Dinero mediante una colaboración (es decir, una realización de caso de uso) con una dependencia de "traza" (una traza es un

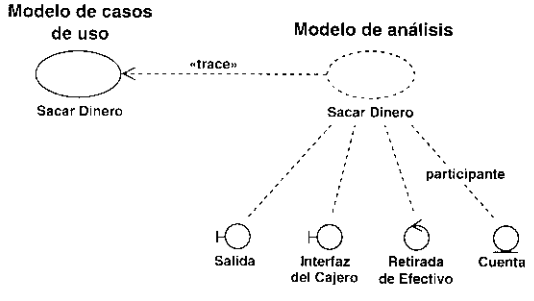


Figura 3.4. Las clases de análisis que participan en la realización de Sacar Dinero. Salida e Interfaz del Cajero son clases de interfaz, Retirada de Efectivo es una clase de control, y Cuenta es una clase de entidad.

estereotipo de dependencia que se indica mediante las comillas << y >>) entre ellos, y cómo cuatro clases participan y desempeñan roles en la realización del caso de uso. Como puede verse en la figura, la notación para una realización de caso de uso o colaboración es una elipse con una línea de trazo discontinuo.

Comenzamos normalmente examinando unos pocos casos de uso, creando sus realizaciones, e identificando los roles de los clasificadores. Después hacemos lo mismo para más casos de uso y proponemos nuevos roles de clasificador. Algunos de estos últimos pueden especificarse como roles nuevos o modificados de clasificadores ya identificados, mientras que otros roles necesitarán nuevos clasificadores. Después observamos de nuevo los primeros casos de uso, alternando de esta forma entre los casos de uso y construyendo gradualmente un modelo de análisis estable. Al final, cada clasificador puede participar y desempeñar papeles en varias realizaciones de caso de uso.

Estereotipos de análisis

En el modelo de análisis, se utilizan tres estereotipos diferentes sobre las clases: "boundary class", "control class",y "entity class"[3] (traducidos al castellano, clase de interfaz, clase de control y clase de entidad, respectivamente). Salida e Interfaz del Cajero son clases de interfaz que se utilizan generalmente para modelar la interacción entre el sistema y sus actores (es decir, los usuarios y los sistemas externos). Retirada de Efectivo es una clase de control que se utiliza generalmente para representar coordinación, secuenciamiento, transacciones y control de otros objetos —y se utiliza con frecuencia para encapsular el control relativo a un caso de uso (en el ejemplo, el caso de uso Sacar Dinero). Cuenta es una clase de entidad, que en general se utilizan para modelar información que tiene una vida larga y que a veces es persistente. Por tanto, cada uno de estos estereotipos de clase encapsula un tipo diferente de comportamiento (o funcionalidad, si se quiere). En consecuencia, los estereotipos nos ayudan a construir un sistema robusto, algo a lo cual volveremos, para explicarlo en más detalle, en el Capítulo 8. También nos ayudan a encontrar activos reutilizables, ya que las clases de entidad son normalmente genéricas para muchos casos de uso, y por tanto para muchas aplicaciones diferentes. La distinción de las clases de análisis en esos tres estereotipos se describe con más detalle en [2]. Se lleva utilizando desde hace muchos años y hoy está muy difundida e está incluida en UML [12].

Ejemplo Una clase que participa en varias realizaciones de caso de uso en el modelo de análisis

En la parte izquierda de la Figura 3.5 tenemos un conjunto de casos de uso para un sistema de Cajero Automático (el mismo de la Figura 3.3), y a la derecha tenemos la correspondiente estructura del sistema, en este caso, las clases de análisis que realizan los casos de uso. La estructura del sistema se modela en un diagrama de clases (Apéndice A). Los **diagramas de clases** se utilizan generalmente para mostrar clases y sus relaciones, pero también pueden utilizarse para mostrar subsistemas e interfaces (como veremos cuando expliquemos el diseño en la Sección 3.4.3). Por sencillez, hemos

[3] *N. del T.* Hemos optado por no traducir los estereotipos en los diagramas UML.

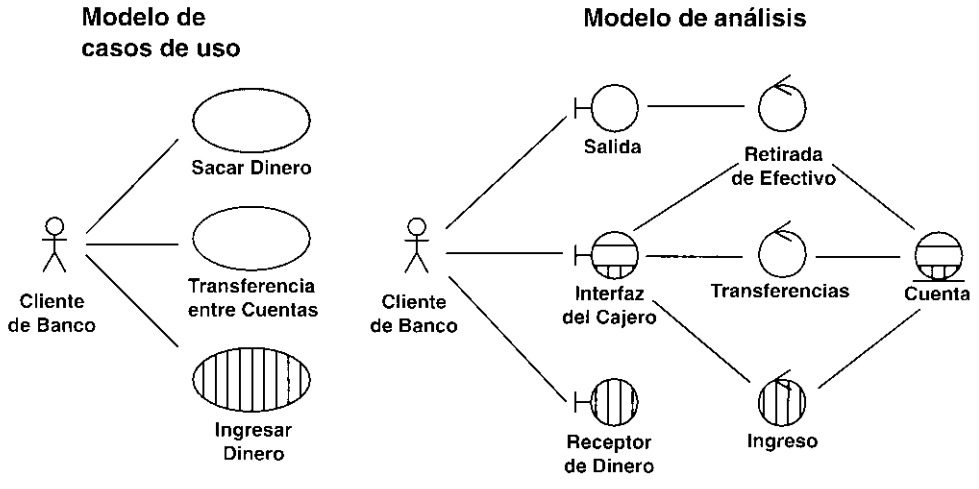


Figura 3.5. El diagrama muestra cómo cada caso de uso (izquierda) se realiza como una estructura de clases del análisis (derecha). Por ejemplo, las clases Interfaz del Cajero, Retirada de Efectivo, Cuenta y Salida participan en la realización del caso de uso Sacar Dinero. Las clases Interfaz del Cajero y Cuenta participan y desempeñan roles en las tres realizaciones de caso de uso. Las otras clases participan en una sola realización de caso de uso, es decir, desempeñan un solo rol.

utilizado diferentes sombreados para indicar en qué realizaciones de caso de uso participa y desempeña algún rol cada clase.

El diagrama de clases para el sistema CA (Figura 3.5) se obtuvo a partir del estudio de las descripciones de los tres casos de uso y de la posterior búsqueda de formas de llevar a cabo cada uno de ellos. Podríamos haber hecho algo como lo siguiente:

- La realización de los tres casos de uso, Sacar Dinero, Transferencia entre Cuentas, e Ingresar Dinero implica a la clase Interfaz del Cajero y a la clase de entidad Cuenta. La ejecución de cada caso de uso comienza con un **objeto** (Apéndice A) de la clase Interfaz del Cajero. Después, el trabajo continúa en un objeto de control que coordina la mayor parte del caso de uso en cuestión. La clase de este objeto es específica de cada caso de uso. La clase Retirada de Efectivo participa por tanto en el caso de uso Sacar Dinero, etc. El objeto Retirada de Efectivo pide a la clase Salida que entregue el dinero, y éste pide al objeto Cuenta que disminuya el saldo.
- El objeto Transferencia pide a los dos objetos Cuenta implicados en la realización del caso de uso Transferencia entre Cuentas que actualicen sus saldos.
- El objeto Ingreso acepta dinero a través del Receptor de Dinero y pide al objeto Cuenta que incremente el saldo.

Hasta aquí hemos trabajado para obtener una estructura del sistema estable para la iteración en curso. Hemos identificado las responsabilidades de los clasificadores participantes, y hemos encontrado relaciones entre los clasificadores. Sin embargo, no hemos identificado en detalle la interacción que debe tener lugar en el desarrollo de los casos de uso. Hemos encontrado la estructura, pero ahora es necesario sobreponer sobre esa estructura los diferentes patrones de interacción necesarios para la realización de cada caso de uso.

Como dijimos anteriormente, cada caso de uso se desarrolla como una realización de caso de uso, y cada realización de caso de uso tiene un conjunto de clasificadores participantes, que desempeñan diferentes roles. Comprender los patrones de interacción significa que describimos cómo se lleva a cabo o se ejecuta (o se instancia) una realización de caso de uso. Por ejemplo, qué ocurre cuando un Cliente de Banco concreto saca dinero, es decir, cuando él o ella ejecuta un caso de uso Sacar Dinero. Sabemos que las clases Interfaz del Cajero, Retirada de Efectivo, Salida y Cuenta participarán en la realización del caso de uso. También sabemos las responsabilidades que deberían cumplir. Sin embargo, aún no sabemos cómo ellas —o más correctamente, cómo objetos de esas clases— interactuarán para llevar a cabo la realización del caso de uso. Esto es lo siguiente que debemos averiguar. Utilizamos fundamentalmente **diagramas de colaboración** (Apéndice A) para modelar las interacciones entre objetos en el análisis. (UML también proporciona diagramas de secuencia para modelar interacciones, volveremos a hablar de ellos cuando tratemos el diseño en la Sección 3.4.3.) Un diagrama de colaboración recuerda a un diagrama de clases, pero contiene instancias y **enlaces** (Apéndice A) en lugar de clases y asociaciones. Muestra cómo interactúan los objetos secuencialmente o en paralelo, numerando los mensajes que se envían unos a otros.

Ejemplo Uso de diagramas de colaboración para describir una realización de caso de uso en el modelo de análisis

En la Figura 3.6 utilizamos un diagrama de colaboración para describir cómo una sociedad de objetos de análisis lleva a cabo la realización del caso de uso Sacar Dinero.

El diagrama muestra cómo el control pasa de un objeto a otro a medida que se lleva a cabo el caso de uso, y los mensajes que se envían entre los objetos. Un mensaje de un objeto dispara al objeto receptor para que tome el control y lleve a cabo una de las responsabilidades de su clase.

El nombre de un mensaje indica el motivo del objeto que realiza la llamada en su interacción con el objeto invocado. Más adelante, en el diseño, estos mensajes se refinarán en una o más operaciones proporcionadas por las clases de diseño correspondientes (como veremos en la Sección 3.4.3).

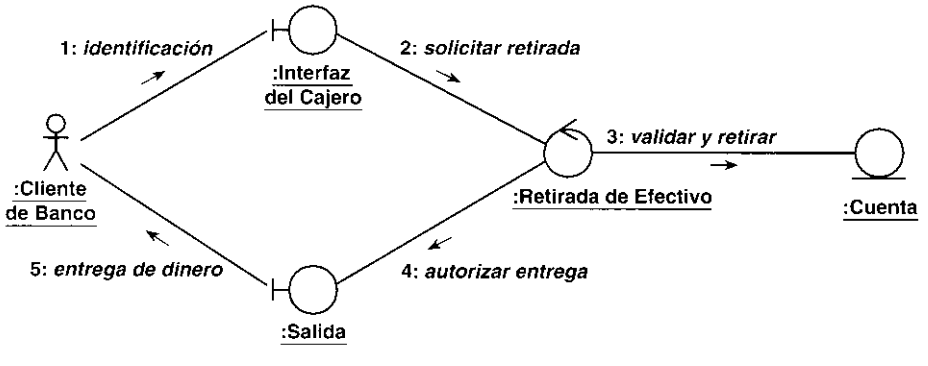


Figura 3.6. Un diagrama de colaboración para la realización del caso de uso Sacar Dinero en el modelo de análisis.

Como complemento a un diagrama de colaboración, los desarrolladores pueden usar también texto para explicar cómo interactúan los objetos para llevar a cabo el flujo de eventos del caso de uso. Existen muchas otras formas de describir una realización, como el texto estructurado o el pseudocódigo.

Ejemplo **Descripción del flujo de sucesos de una realización de caso de uso**

Describimos ahora la realización del caso de uso Sacar Dinero en términos de objetos y actores que interactúan, como se muestra en la Figura 3.6.

Un Cliente de Banco decide sacar dinero y activa el objeto Interfaz del Cajero. El Cliente de Banco se identifica y especifica la cantidad a retirar y la cuenta de la cual hacerlo (1). Interfaz del Cajero verifica la identidad del Cliente de Banco y solicita al objeto Retirada de Efectivo que lleve a cabo la transacción (2).

Si la identidad del Cliente de Banco es válida, se le solicita al objeto Retirada de Efectivo que confirme si el cliente del banco tiene derecho a sacar la cantidad especificada de la Cuenta. El objeto Retirada de Efectivo lo confirma solicitando al objeto Cuenta que valide la petición y, si la petición es válida, que reste la cantidad (3).

Después el objeto Retirada de Efectivo autoriza a Salida a que entregue al Cliente de Banco la cantidad solicitada (4). Entonces es cuando el Cliente de Banco recibe la cantidad de dinero solicitada (5).

Obsérvese que este sencillo ejemplo sólo muestra un camino a través de la realización del caso de uso, cuando todo sucede sin complicaciones. Podrían ocurrir complicaciones, por ejemplo, si el saldo de la Cuenta es demasiado bajo para permitir la retirada de efectivo.

En este momento, hemos analizado cada caso de uso y por tanto hemos identificado todos los roles de clase que participan en cada realización de caso de uso. Ahora pasamos a cómo analizar cada clase.

3.4.2. Cada clase debe cumplir todos sus roles de colaboración

Las responsabilidades de una clase son sencillamente la recopilación de todos los roles que cumple en todas las realizaciones de caso de uso. Juntándolas y eliminando repeticiones entre los roles, obtenemos una especificación de todas las responsabilidades y atributos de la clase.

Los desarrolladores responsables de analizar y realizar los casos de uso son los responsables de especificar los roles de las clases. Un desarrollador responsable de una clase agrupa todos los roles de la clase en un conjunto completo de responsabilidades para la clase, y después los integra en un conjunto consistente de responsabilidades.

Los desarrolladores responsables de analizar los casos de uso deben asegurarse de que las clases realizan los casos de uso con la calidad adecuada. Si se cambia una clase, su desarrollador debe verificar que la clase sigue cumpliendo sus roles en las realizaciones de casos de uso. Si se cambia un rol en una realización de caso de uso, el desarrollador del caso de uso

debe comunicar el cambio al desarrollador de la clase. Por tanto, los roles ayudan a mantener la integridad del análisis tanto a los desarrolladores de las clases, como a los de los casos de uso.

3.4.3. Creación del modelo de diseño a partir del modelo de análisis

El modelo de diseño se crea tomando el modelo de análisis como entrada principal, pero se adapta al entorno de implementación elegido, por ejemplo, a un *object request broker*, un kit de construcción de IGU, o a un sistema de gestión de bases de datos. También debe adaptarse para reutilizar **sistemas heredados** (Apéndice C) u otros marcos de trabajo desarrollados para el proyecto. Por tanto, mientras que el modelo de análisis sirve como una primera aproximación del modelo de diseño, el modelo de diseño funciona como esquema para la implementación.

De igual forma que el modelo de análisis, el modelo de diseño también define clasificadores (clases, subsistemas e interfaces), relaciones entre esos clasificadores, y colaboraciones que llevan a cabo los casos de uso (las realizaciones de casos de uso). Sin embargo, los elementos definidos en el modelo de diseño son las "contrapartidas de diseño" de los elementos, más conceptuales, definidos en el modelo de análisis, en el sentido de que los primeros (diseño) se adaptan al entorno de la implementación mientras que los últimos (análisis) no. Dicho de otra forma, el modelo de diseño es más "físico" por naturaleza, mientras que el modelo de análisis es más "conceptual".

Puede hacerse la traza de una realización de caso de uso en el modelo de análisis a partir de una realización de caso de uso en el modelo de diseño.

Ejemplo **Realizaciones de casos de uso en los modelos de análisis y diseño**

En la Figura 3.7 describimos cómo el caso de uso Sacar Dinero se lleva a cabo mediante una realización de caso de uso tanto en el modelo de análisis como en el modelo de diseño.

Las realizaciones de caso de uso en los diferentes modelos sirven para cosas distintas. Recuérdese de la Sección 3.4.1 (Figura 3.4) que las clases de análisis Interfaz del Cajero, Retirada de Efectivo, Cuenta, y Salida participan en la realización del caso de uso Sacar Dinero en el modelo de análisis. Sin embargo, cuando se diseñan esas clases del análisis, todas ellas especifican y hacen surgir clases de diseño más refinadas que se adaptan al entorno de implementación, como se ejemplifica en la Figura 3.8.

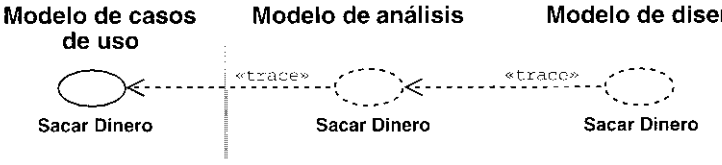


Figura 3.7. Realizaciones de caso de uso en diferentes modelos.

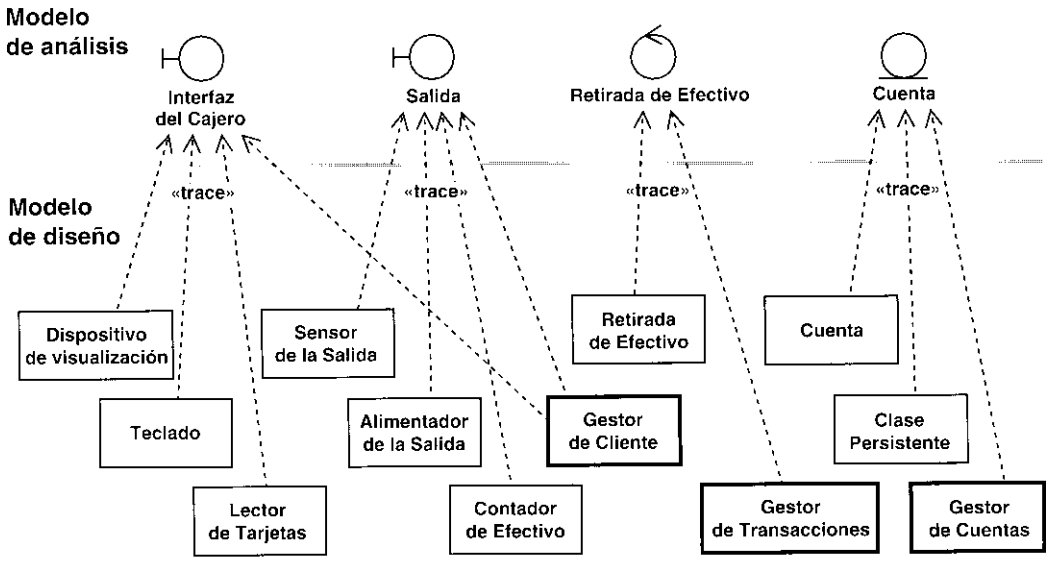


Figura 3.8. Clases de diseño del modelo de diseño con sus trazas hacia clases del modelo de análisis.

Por ejemplo, la clase del análisis llamada Interfaz del Cajero se diseña mediante cuatro clases del diseño: Dispositivo de visualización, Teclado, Lector de Tarjetas, y Gestor de Cliente (ésta última es una clase activa y por tanto se dibuja con borde grueso; *véase* el Apéndice A).

Obsérvese en la Figura 3.8 que la mayoría de las clases de diseño normalmente tienen una sola traza a una clase de análisis. Esto es habitual para las clases de diseño que son específicas de la aplicación, diseñadas para dar soporte a una aplicación o a un conjunto de ellas. Por tanto, la estructura del sistema definida por el modelo de análisis se conserva de forma natural durante el diseño, aunque pueden ser necesarios algunos cambios (como por ejemplo, el Gestor de Cliente que participa en el diseño de las dos clases Interfaz del Cajero y Salida). Además, las clases activas (Gestor de Cliente, Gestor de Transacciones y Gestor de Cuentas) representan procesos que organizan el trabajo de las otras clases (no activas) cuando el sistema es distribuido (volveremos a este tema en la Sección 4.5.3).

Como consecuencia, la realización del caso de uso Sacar Dinero en el modelo de diseño debe describir cómo se realiza el caso de uso en términos de las clases de diseño correspondientes. La Figura 3.9 muestra un diagrama de clases que es parte de la realización del caso de uso.

Es evidente que este diagrama de clases incluye más detalles que el diagrama de clases del modelo de análisis (Figura 3.5). Este detalle es necesario debido a la adaptación del modelo de diseño al entorno de la implementación.

De manera parecida a cómo lo hacíamos en el análisis (Figura 3.6), debemos identificar la interacción detallada entre los objetos de diseño que tiene lugar cuando se lleva a cabo el caso de uso en el modelo de diseño. Utilizamos principalmente diagramas de secuencia para modelar las interacciones entre objetos del diseño, como se muestra en la Figura 3.10.

El diagrama de secuencia muestra cómo el control —que comienza en la esquina superior izquierda— pasa de un objeto a otro a medida que se ejecuta el caso de uso y a medida que se envían mensajes entre objetos. Un mensaje enviado por un objeto dispara la toma del control en el objeto receptor y la realización de las operaciones de su clase.

Es un ejercicio interesante comparar este diagrama de secuencia con su "contrapartida de análisis" —es decir, el diagrama de colaboración— de la Figura 3.6. Como puede comprobarse, los dos

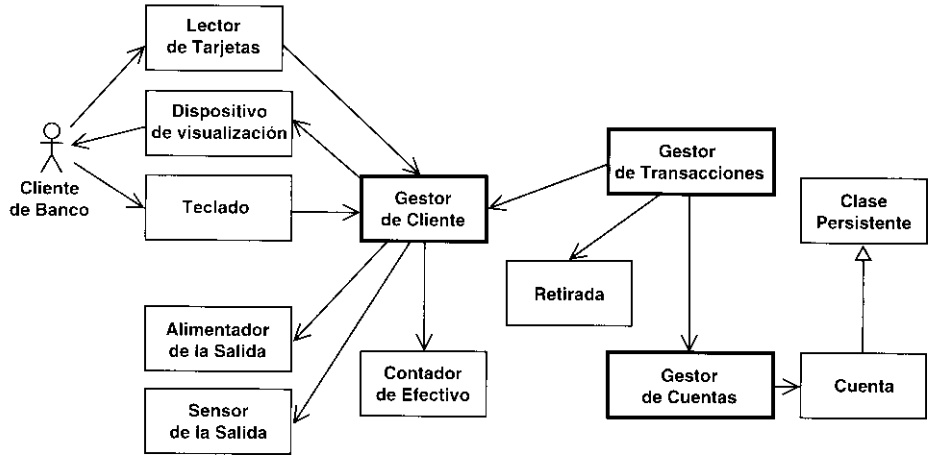


Figura 3.9. Un diagrama de clases que es parte de la realización del caso de uso Sacar Dinero en el modelo de diseño. Cada clase de diseño participa y asume roles en la realización del caso de uso.

primeros mensajes del diagrama de colaboración ("1: identificación" y "2: solicitar retirada") se diseñan mediante todos los mensajes en el diagrama de secuencia de la Figura 3.10. Esto nos proporciona una idea de la complejidad y de la cantidad de detalle que se incluye en el modelo de diseño en comparación con la del modelo de análisis.

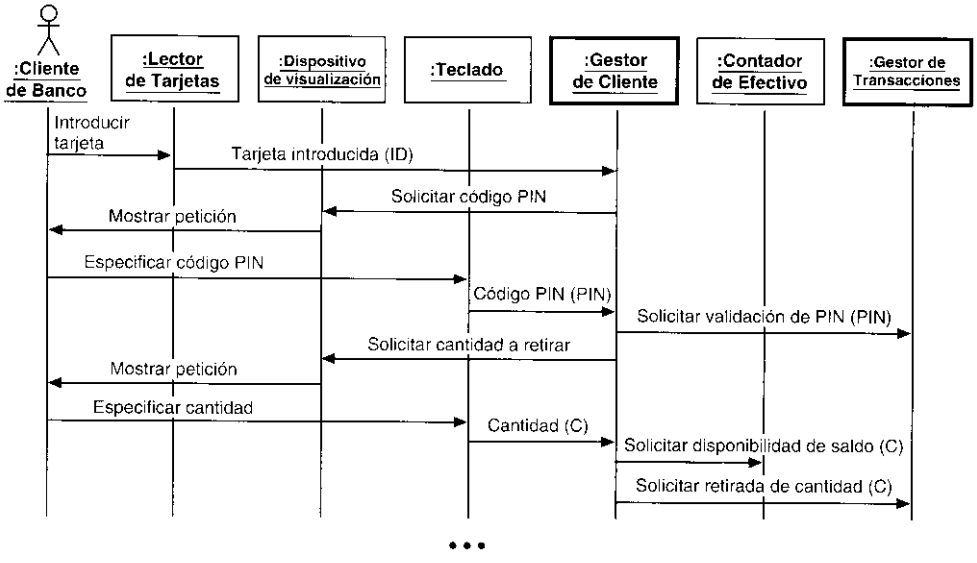


Figura 3.10. Un diagrama de secuencia que es parte de la realización del caso de uso Sacar Dinero en el modelo de diseño.

De igual forma que en los diagramas de colaboración, los desarrolladores pueden también utilizar texto como complemento a los diagramas de secuencia —explicando cómo interactúan los objetos de diseño para llevar a cabo el flujo de eventos del caso de uso.

Como puede observarse en este ejemplo, el modelo de diseño contendrá probablemente muchas clases. Por tanto, es necesaria una forma de organizarlas. Esto se hace mediante subsistemas, que tratamos en la siguiente sección.

3.4.4. Los subsistemas agrupan a las clases

Es imposible utilizar sólo clases para realizar los casos de uso en un sistema grande con cientos o miles de clases: el sistema es demasiado grande para poder comprenderlo sin una organización de más alto nivel. Las clases se agrupan en subsistemas. Un subsistema es un agrupamiento semánticamente útil de clases o de otros subsistemas. Un subsistema posee un conjunto de interfaces que ofrece a sus usuarios. Estas interfaces definen el contexto del subsistema (actores y otros subsistemas y clases).

Los subsistemas de bajo nivel se denominan **subsistemas de servicio** (Apéndice B; *véase* también el Capítulo 9) debido a que sus clases llevan a cabo un servicio (para una descripción más detallada del concepto de servicio, *véase* la Sección 8.4.5.1). Los subsistemas de servicio constituyen una unidad manejable de funcionalidad opcional (o potencialmente opcional). Sólo es posible instalar un subsistema en un sistema del cliente si se hace en su totalidad. Los subsistemas de servicio también se utilizan para modelar grupos de clases que tienden a cambiar juntas.

Los subsistemas pueden diseñarse descendente o ascendentemente. Cuando se hace de manera ascendente, los desarrolladores proponen subsistemas basados en las clases que ya han identificado; proponen subsistemas que empaquetan las clases en unidades con unas funciones claramente definidas. Si en cambio los desarrolladores eligen un enfoque descendente, el arquitecto identifica los subsistemas de más alto nivel y las interfaces antes de que se hayan identificado las clases. Por tanto, se asigna a los desarrolladores el trabajo con determinados subsistemas para identificar y diseñar las clases dentro de sus subsistemas. Presentamos los subsistemas en el Capítulo 1, y los explicaremos con más profundidad en los Capítulos 4 y 9.

Ejemplo **Los subsistemas agrupan clases**

Los desarrolladores agrupan las clases en los tres subsistemas que se muestran en la Figura 3.11. Estos subsistemas se eligieron de forma que todas las clases que proporcionan la interfaz gráfica se ubican en un subsistema, todas las que tienen que ver con las cuentas en otro, y las clases específicas de los casos de uso en un tercero. La ventaja de colocar todas las clases de interfaz gráfica en un subsistema de Interfaz CA es que podemos reemplazar este subsistema por otro que ofrezca la misma funcionalidad al subsistema de Gestión de Transacciones. Un subsistema de Interfaz CA alternativo podría ofrecer una implementación de la interfaz de usuario muy diferente, quizá diseñada para aceptar y entregar monedas en lugar de billetes.

Cada una de las clases específicas de los casos de uso en el subsistema de Gestión de Transacciones, como la clase Retirada de Efectivo, se colocan en un subsistema de servicio aparte. Cada uno de estos subsistemas de servicio podría contener más de una clase en realidad, pero no lo hemos reflejado en nuestro sencillo ejemplo.

La Figura 3.11 también muestra las interfaces entre los subsistemas. Un círculo representa una interfaz. La línea continua de una clase a una interfaz significa que la clase proporciona la interfaz.

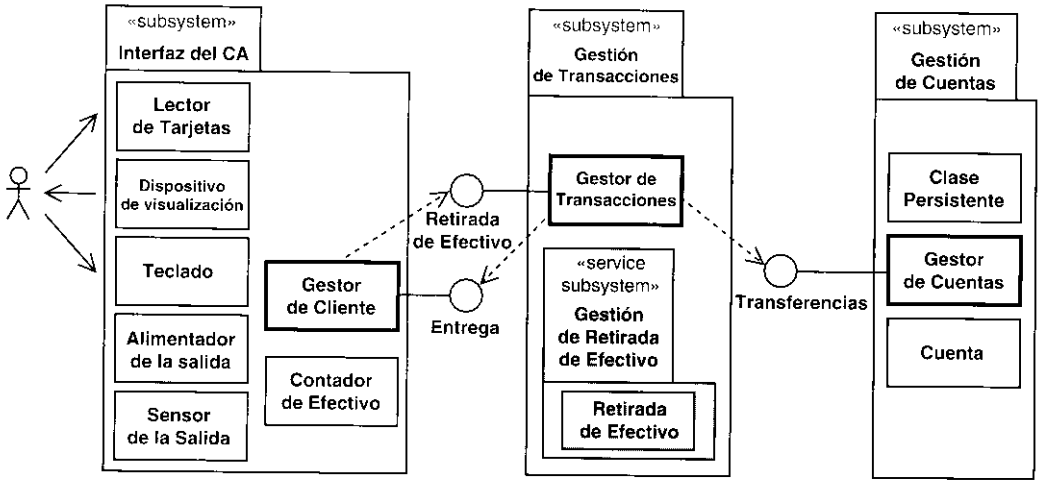


Figura 3.11. Tres subsistemas y un subsistema de servicio (en gris dentro del subsistema de Gestión de Transacciones) para nuestro ejemplo sencillo del CA.

Una línea de trazo discontinuo de una clase a una interfaz significa que la clase usa la interfaz. Por sencillez, no mostramos las interfaces que proporcionan o utilizan los actores; en su lugar, empleamos asociaciones normales.

La interfaz Transferencias define operaciones para transferir dinero entre cuentas, retirar dinero, e ingresar dinero en cuentas. La interfaz Retirada de Efectivo define operaciones para solicitar retiradas de efectivo de una cuenta. La interfaz Entrega define operaciones que otros subsistemas, como el de Gestión de Transacciones, pueden utilizar para entregar dinero al cliente del banco.

3.4.5. Creación del modelo de implementación a partir del modelo de diseño

Durante el flujo de trabajo de implementación desarrollamos todo lo necesario para obtener un sistema ejecutable: componentes ejecutables, componentes de fichero (código fuente, guiones *shell*, etc), componentes de tabla (elementos de la base de datos), etc. Un componente es una parte física y reemplazable del sistema que cumple y proporciona la realización de un conjunto de interfaces. El modelo de implementación está formado por componentes, que incluyen todos los **ejecutables** (Apéndice A; *véase* también el Capítulo 10) tales como componentes ActiveX y JavaBeans, así como otros tipos de componentes.

Ejemplo **Componentes en el modelo de implementación**

En la Figura 3.12 se muestran los componentes que implementan las clases de diseño (como se definió en la Sección 3.4.3).

Por ejemplo, el componente fichero salida.c contiene el código fuente de tres clases (y por tanto las implementa): Alimentador de la Salida, Gestor de Cliente, y Contador de Efectivo. Este compo-

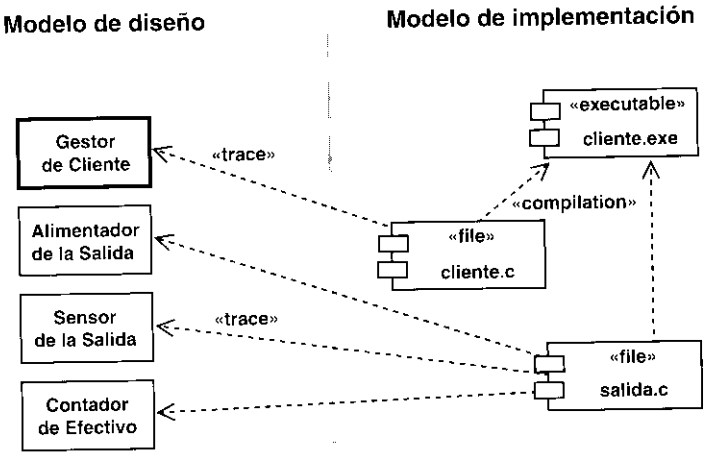


Figura 3.12. Componentes que implementan clases de diseño.

nente fichero se compilará y enlazará junto con el componente fichero cliente.c para obtener cliente.exe, que es un ejecutable.

Un componente presupone un contexto de la arquitectura definido por sus interfaces. También es reemplazable, es decir, los desarrolladores pueden intercambiar un componente con otro, quizás mejor, siempre que el nuevo proporcione y requiera las mismas interfaces. Normalmente existe una forma directa de implementar un subsistema de servicio del modelo de diseño mediante componentes que pueden asignarse a nodos del modelo de despliegue. Cada subsistema de servicio se implementará mediante un componente si siempre se asigna a un mismo tipo de nodo en el modelo de despliegue. Si se asigna a más de un nodo, podemos dividir el subsistema de servicio —normalmente separando algunas clases— en tantas partes como tipos de nodo. En este caso, cada parte del subsistema de servicio se implementará como un componente.

Ejemplo **Un subsistema de servicio implementado mediante componentes**

Supongamos que hemos elegido una solución **cliente/servidor** (*véase* la Sección 9.5.1.1) para nuestro ejemplo del Cajero Automático. Podríamos distribuir tanto en el cliente como en el servidor parte del subsistema de servicio Gestión de Retirada de Efectivo (Figura 3.11) que contiene a la clase Retirada. El subsistema de servicio Gestión de Retirada de Efectivo podría implementarse mediante dos componentes: "Retirada en el Cliente" y "Retirada en el Servidor".

Si los componentes se implementan en un lenguaje de programación orientado a objetos, la implementación de las clases es también directa. Cada clase de diseño se corresponde con una clase en la implementación, por ejemplo, clases C++ o Java. Cada componente fichero puede implementar varias de esas clases, dependiendo de las convenciones del lenguaje de programación.

Pero la implementación es más que el desarrollo del código para crear un sistema ejecutable. Los desarrolladores responsables de implementar un componente son también responsables de hacer su prueba de unidad antes de enviarlo a las pruebas de integración y del sistema.

3.5. Prueba de los casos de uso

Durante la prueba, verificamos que el sistema implementa correctamente su especificación. Desarrollamos un modelo de prueba compuesto por **casos de prueba** y **procedimientos de prueba** (Apéndice C; *véase* también el Capítulo 11) y después ejecutamos los casos de prueba para estar seguros de que el sistema funciona como esperamos. Un caso de prueba es un conjunto de entradas de prueba, condiciones de ejecución, y resultados esperados, desarrollados para un objetivo concreto, tal como probar un camino concreto a través de un caso de uso, o verificar que se cumple un requisito específico. Un procedimiento de prueba es una especificación de cómo llevar a cabo la preparación, ejecución, y evaluación de los resultados de un caso de prueba particular. Los procedimientos de prueba también pueden derivarse de los casos de uso. Los defectos hallados se analizan para localizar el problema. Después estos problemas se priorizan y se corrigen por orden de importancia.

En la Sección 3.3, comenzamos con la captura de los casos de uso, y después en la Sección 3.4 analizamos, diseñamos, e implementamos un sistema que llevaba a cabo los casos de uso. Ahora, describiremos cómo probar que los casos de uso se han implementado correctamente. De alguna forma, esto no es nada nuevo. Los desarrolladores siempre han probado los casos de uso, incluso antes de que se acuñara el término *caso de uso*. La forma práctica de probar las funciones de un sistema es la prueba de que el sistema puede utilizarse de maneras que tengan sentido para los usuarios. Sin embargo, por otro lado, ésta es una técnica nueva. Es nueva ya que identificamos los casos de prueba (como casos de uso durante el flujo de trabajo de los requisitos) antes de que tan siquiera comencemos a diseñar el sistema, y después nos aseguramos de que nuestro diseño realmente implementa los casos de uso.

Ejemplo Identificación de un caso de prueba a partir de un caso de uso

En la Figura 3.13 mostramos un caso de prueba, Sacar Dinero-Flujo Básico, que especifica cómo probar el flujo básico del caso de uso Sacar Dinero.

Obsérvese el nuevo estereotipo que presentamos aquí para los casos de prueba, un símbolo de caso de uso con una cruz dentro. Hacemos esto para poder dibujar los casos de prueba en los diagramas (*véase* el Capítulo 11).

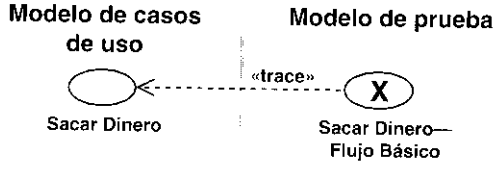


Figura 3.13. Un caso de prueba del modelo de prueba que especifica cómo probar un caso de uso (Sacar Dinero) del modelo de casos de uso.

El caso de prueba especifica la entrada, los resultados esperados, y otras condiciones relevantes para verificar el flujo básico del caso de uso Sacar Dinero:

Entradas:

- La cuenta 12-121-1211 del Cliente de Banco tiene un saldo de 350 dólares.
- El Cliente de Banco se identifica correctamente.
- El Cliente de Banco solicita la retirada de 200 dólares de la cuenta 12-121-1211.
- Hay suficiente dinero (por lo menos 200 dólares) en el Cajero Automático.

Resultados:

- El saldo de la cuenta 12-121-1211 del Cliente de Banco disminuye a 150 dólares.
- El Cliente de Banco recibe 200 dólares del Cajero Automático.

Condiciones:

No se permite a ningún otro caso de uso (instancias de) acceder a la cuenta 12-121-1211 durante la ejecución del caso de prueba.

Obsérvese que este caso de prueba se basa en la descripción del caso de uso Sacar Dinero que se dio en la Sección 3.3.3. Mediante la identificación temprana de los casos de uso, podemos comenzar pronto la planificación de las actividades de prueba, y podemos proponer casos de prueba desde el comienzo. Estos casos de prueba podrán detallarse más durante el diseño, cuando sepamos más sobre cómo el sistema llevará a cabo los casos de uso. Algunas herramientas generan los casos de prueba a partir del modelo de diseño —todo lo que hay que hacer es introducir manualmente los datos necesarios para ejecutar las pruebas.

Las pruebas de los casos de uso pueden llevarse a cabo bien desde la perspectiva de un actor que considera el sistema como una caja negra, o bien desde una perspectiva de diseño, en la cual el caso de prueba se construye para verificar que las instancias de las clases participantes en la realización del caso de uso hacen lo que deberían hacer. Las pruebas de caja negra pueden identificarse, especificarse y planificarse tan pronto como los requisitos sean algo estables.

También hay otro tipo de pruebas, como las del sistema, las de aceptación, y las de la documentación de usuario. Hablaremos más sobre las pruebas en el Capítulo 11.

3.6. Resumen

Los casos de uso dirigen el proceso. Durante el flujo de trabajo de los requisitos, los desarrolladores pueden representar los requisitos en la forma de casos de uso. Los jefes de proyecto pueden después planificar el proyecto en términos de los casos de uso con los cuales trabajan los desarrolladores. Durante el análisis y el diseño, los desarrolladores crean realizaciones de casos de uso en términos de clases y subsistemas. Los componentes se incorporan en los incrementos, y cada uno de ellos realiza un conjunto de casos de uso. Por último, los ingenieros de prueba verifican que el sistema implementa los casos de uso correctos para los usuarios. En otras palabras, los casos de uso enlazan todas las actividades del desarrollo y dirigen el proceso de desarrollo —éste es quizá el beneficio más importante de la aproximación dirigida por los casos de uso.

Los casos de uso proporcionan muchos beneficios al proyecto. Sin embargo, no son todo. En el siguiente capítulo hablaremos sobre otro aspecto muy importante del Proceso Unificado —el estar centrado en la arquitectura.

3.7. Referencias

[1] Ivar Jacobson, "Object-oriented development in an industrial environment", *Proceedings of OOPSLA'87,* Special issue of *SIGPLAN Notices* 22(12): 183–191, December 1987.

[2] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard, *Object-Oriented Software Engineering: A Use-Case Driven Approach,* Reading, MA: Addison-Wesley, 1992 (Revised fourth printing, 1993).

[3] Ivar Jacobson, "Basic use case modeling," *ROAD* 1(2), July–August 1994.

[4] Ivar Jacobson, "Basic Use Case Modeling (continued)", *ROAD* 1 (3), September–October 1994.

[5] Ivar Jacobson, "Use cases and objects", *ROAD* 1(4), November–December 1994.

[6] Ivar Jacobson and Magnus Christerson, "A growing consensus on use cases", *Journal of Object-Oriented Programming,* March–April 1995.

[7] Ivar Jacobson and Sten Jacobson, "Use-case engineering: Unlocking the power", *Object Magazine,* October 1996.

[8] Karl Wieger, "Use cases: Listening to the customer's voice", *Software Development,* March 1997.

[9] E. Ecklund, L. Delcambre, and M. Freiling, "Change cases: Use cases that identify future requirements", *Proceedings, Conference on Object-Oriented Programming Systems, Languages, & Applications (OOPSLA'96),* ACM, 1996, pp. 342–358.

[10] Alistair Cockburn, "Structuring use cases with goals", *Report on Analysis & Design* (ROAD), 1997.

[11] Geri Schneider and Jason Winters, *Applying Use Cases: A Practical Approach,* Reading, MA: Addison-Wesley, 1998.

[12] OMG Unified Modeling Language Specification. Object Management Group, Framingham, MA, 1998. Internet: www.omg.org.

Capítulo 4

Un proceso centrado en la arquitectura

En el Capítulo 3, comenzamos con una simplificación, diciendo que sólo los casos de uso mostrarían el camino a través de los requisitos, análisis, diseño, implementación y pruebas para producir un sistema. No obstante, para desarrollar software hay que hacer algo más que conducirse a ciegas a través de los flujos de trabajo guiado exclusivamente por los casos de uso.

Los casos de uso solamente no son suficientes. Se necesitan más cosas para conseguir un sistema de trabajo. Esas "cosas" son la arquitectura. Podemos pensar que la arquitectura de un sistema es la visión común en la que todos los empleados (desarrolladores y otros usuarios) deben estar de acuerdo, o como poco, deben aceptar. La arquitectura nos da una clara perspectiva del sistema completo, necesaria para controlar el desarrollo.

Necesitamos una arquitectura que describa los elementos del modelo que son más importantes para nosotros. ¿Cómo podemos determinar qué elementos son importantes? Su importancia reside en el hecho de que nos guían en nuestro trabajo con el sistema, tanto en este ciclo como a través del ciclo de vida completo. Estos elementos significativos, arquitectónicamente hablando, incluyen algunos de los subsistemas, dependencias, interfaces, colaboraciones, nodos y clases activas (Apéndice A; *véase* también la Sección 9.3.2). Describen los cimientos del sistema, que son necesarios como base para comprenderlo, desarrollarlo y producirlo económicamente.

Vamos a comparar un proyecto software con la construcción de un garaje de uso individual. En primer lugar, el constructor debería considerar cómo quieren utilizar el garaje los usuarios.

Un caso de uso podría ser ciertamente Guardar el Coche, es decir, conducir el coche dentro del garaje, y dejarlo allí para sacarlo más tarde.

¿Tendrá el usuario algún otro uso en mente? Supongamos que quiere utilizarlo también como taller. Esto lleva al constructor a pensar en las necesidades de luz —algunas ventanas y luz eléctrica—. Muchas herramientas funcionan con electricidad, así que el constructor debe planear poner media docena de enchufes y el suficiente voltaje para soportarlos. De alguna forma, el constructor está desarrollando una arquitectura simple. Puede hacerlo mentalmente porque ya ha visto un garaje antes. Ha visto un banco de trabajo antes. Él sabe que una persona necesita un banco de trabajo para trabajar. El enfoque del constructor no es ciego porque ya está familiarizado con la arquitectura típica de un pequeño garaje. Simplemente ha juntado las partes para adecuar el garaje para su uso. ¿Y si el constructor nunca ha visto un garaje y se centra ciegamente en la forma en que se utilizaría? Construiría un garaje muy extraño. Así que necesita considerar no solamente la funcionalidad del garaje, sino su forma.

Construir una casa con 10 habitaciones, una catedral, un centro comercial o un rascacielos, son tareas diferentes. Hay muchas formas de construir grandes edificios como éstos. Para diseñarlos se requiere un equipo de arquitectos. Los miembros del equipo tendrán que mantenerse informados del progreso de la arquitectura. Esto significa que tendrán que registrar su trabajo de forma que sea comprensible para los otros miembros del equipo. También tendrán que presentárselo de forma comprensible a una persona no experta —ya sea el propietario, el usuario u otros interesados—. Por último, también deben informar de la arquitectura al constructor y a los proveedores de materiales mediante los planos del edificio.

De forma similar, el desarrollo de muchos sistemas software —o de sistemas software con el hardware correspondiente sobre el que se ejecuta— requiere el pensar y registrar las ideas de forma compresible, no sólo para los siguientes desarrolladores, sino para otro tipo de usuarios. Es más, estas ideas, esta arquitectura, no brotan hechas y derechas del monte de Zeus: los arquitectos las desarrollan iterando repetidas veces durante la fase de inicio y elaboración. De hecho, el primer objetivo de la fase de elaboración es establecer una arquitectura sólida de forma que sea una arquitectura base ejecutable (Apéndice C). Como resultado, se entra en la fase de construcción con unos fundamentos sólidos para construir el resto del sistema.

4.1. La arquitectura en pocas palabras

Necesitamos una arquitectura, bien. Pero, ¿qué quiere decir exactamente "arquitectura de sistemas software"? A medida que uno busca en la literatura sobre arquitectura software, nos viene a la cabeza la parábola del ciego y el elefante. Un elefante es todo lo que el hombre ciego va encontrando —una gran serpiente (la trompa), un trozo de cuerda (la cola), un árbol pequeño (la pata)—. De forma similar, la idea de arquitectura, al menos reducida a una sencilla frase definitoria, es lo que se encuentra en la mente del autor en ese punto.

Vamos a comparar de nuevo la arquitectura software con la construcción de casas. Un edificio es habitualmente una sola unidad desde la perspectiva del cliente. El arquitecto del edificio, por tanto, puede encontrar útil el hacer una maqueta a escala del edificio, junto con los dibujos del edificio visto desde distintas perspectivas. Estos planos no son muy detallados por lo general, pero son comprensibles para el cliente.

No obstante, la construcción de edificios implica a otro tipo de trabajadores durante la fase de construcción, como carpinteros, albañiles, peones, soladores, fontaneros y electricistas.

Todos ellos necesitan más detalles y planos especializados del edificio, y todos estos planos deben ser consistentes unos con otros. Las tuberías de ventilación y las tuberías de agua, por ejemplo, no deben estar situadas en el mismo espacio físico. El papel del arquitecto es crear los aspectos más significativos del diseño del edificio en su conjunto. Así que el arquitecto hace un conjunto de planos del edificio que describen muchos de los bloques del edificio, tales como los pilares en la excavación. Un ingeniero de estructuras determina el tamaño de las vigas que soportan la estructura. Los pilares soportan las paredes, los suelos y el techo. Esta estructura contiene sistemas para ascensores, agua, electricidad, aire acondicionado, sanitarios, etc. Sin embargo, estos planos de la arquitectura no son lo suficientemente detallados para que los constructores trabajen a partir de ellos. Los delineantes preparan planos y especificaciones que proporcionan detalles sobre la elección de materiales, subsistemas de ventilación, y cosas así. El arquitecto tiene la responsabilidad global sobre el proyecto, pero estos otros tipos de diseñadores completan los detalles. En general, el arquitecto es un experto en integrar todos los aspectos del edificio, pero no es un experto en cada aspecto. Cuando todos los planos están hechos, los planos de la arquitectura cubren únicamente las partes principales del edificio. Los planos son vistas de todos los otros planos, siendo consistentes con todos ellos.

Durante la construcción, distintos trabajadores utilizarán los planos de la arquitectura —las vistas de los planos detallados— para tener una buena visión global del edificio, pero se fiarán de los planos de la construcción más detallados para realizar su trabajo.

Como un edificio, un sistema software es una única entidad, pero al arquitecto del software y a los desarrolladores les resulta útil presentar el sistema desde diferentes perspectivas para comprender mejor el diseño. Estas perspectivas son **vistas** (Apéndices A y C) del modelo del sistema. Todas las vistas juntas representan la arquitectura.

La arquitectura software abarca decisiones importantes sobre:

- La organización del sistema software.
- Los elementos estructurales que compondrán el sistema y sus interfaces, junto con sus comportamientos, tal y como se especifican en las colaboraciones entre estos elementos.
- La composición de los elementos estructurales y del comportamiento en subsistemas progresivamente más grandes.
- El **estilo de la arquitectura** (Apéndice C) que guía esta organización: los elementos y sus interfaces, sus colaboraciones y su composición.

Sin embargo, la arquitectura software está afectada no sólo por la estructura y el comportamiento, sino también por el uso, la funcionalidad, el rendimiento, la flexibilidad, la reutilización, la facilidad de comprensión, las restricciones y compromisos económicos y tecnológicos, y la estética.

En la Sección 4.4, discutiremos el concepto de arquitectura software en términos más concretos y se describirá cómo representarla utilizando el Proceso Unificado. No obstante, aquí insinuaremos una **descripción de la arquitectura** (Apéndice C). Ya hemos dicho que la arquitectura se representa mediante vistas del modelo: una vista del modelo de casos de uso, una vista del modelo de análisis, una vista del modelo de diseño, etc. Este conjunto de vistas concuerda con las 4+1 vistas discutidas en [3]. Ya que una vista de un modelo es un extracto, o una parte de ese modelo, una vista del modelo de casos de uso, por ejemplo, se parece al propio modelo de casos de uso. Tiene actores y casos de uso, pero solamente aquellos que son arquitectónicamente

significativos. De forma similar, la vista de la arquitectura del modelo de diseño se parece al modelo de diseño, pero contiene exclusivamente aquellos elementos del diseño que comprenden los casos de uso importantes desde el punto de vista de la arquitectura (*véase* la Sección 12.6.2).

No hay nada mágico en la descripción de la arquitectura. Es como un descripción completa del sistema con todos sus modelos (hay algunas diferencias que comentaremos más tarde), pero más pequeño. ¿Cómo de pequeño? No hay un tamaño absoluto para la descripción de la arquitectura, pero según nuestra experiencia una gran clase de sistemas suele estar entre 50 y 100 páginas. Ese rango es aplicable en **sistemas de una sola aplicación** (Apéndice C); las descripciones de arquitectura para **sistemas de conjuntos de aplicaciones** será mayor (Apéndice C).

4.2. Por qué es necesaria la arquitectura

Un sistema software grande y complejo requiere una arquitectura para que los desarrolladores puedan progresar hasta tener una visión común. Un sistema software es difícil de abarcar visualmente porque no existe en un mundo de tres dimensiones. Es a menudo único y sin precedente en determinados aspectos. Suele utilizar tecnología poco probada o una mezcla de tecnologías nuevas. Tampoco es raro que el sistema lleve a sus últimos límites la tecnología existente. Además, debe ser construido para acomodar gran cantidad de clases que sufrirán cambios futuros. A medida que los sistemas se hacen más complejos, "los problemas de diseño van más allá de los algoritmos y las estructuras de datos para su computación: para diseñar y especificar una estructura del sistema global surgen nuevos tipos de problemas" [1].

Además, existe con frecuencia un sistema que ya realiza algunas de las funciones del sistema propuesto. El saber identificar qué hace este sistema, casi siempre con poca o ninguna documentación, y qué código pueden reutilizar los desarrolladores, añade complejidad al desarrollo.

Se necesita una arquitectura para:

- Comprender el sistema.
- Organizar el desarrollo.
- Fomentar la reutilización.
- Hacer evolucionar el sistema.

4.2.1. Comprensión del sistema

Para que una organización desarrolle un sistema, dicho sistema debe ser comprendido por todos los que vayan a intervenir en él. El hacer que los sistemas modernos sean comprensibles es un reto importante por muchas razones:

- Abarcan un comportamiento complejo.
- Operan en entornos complejos.
- Son tecnológicamente complejos.
- A menudo combinan computación distribuida, productos y plataformas comerciales (como sistemas operativos y sistemas gestores de bases de datos) y reutilizan componentes y marcos de trabajo.
- Deben satisfacer demandas individuales y de la organización.

- En algunos casos son tan grandes que la dirección tiene que dividir el trabajo de desarrollo en varios proyectos, que están a menudo separados geográficamente, añadiendo dificultades a la hora de coordinarlos.

Por otra parte, estos factores cambian constantemente. Todo esto añade dificultad potencial para comprender la situación.

Para realizar un desarrollo centrado en la arquitectura (Apéndice C) hay que prevenir estos fallos en la compresión del sistema. Por consiguiente, el primer requisito que tiene lugar en una descripción de la arquitectura es que se debe capacitar a los desarrolladores, directivos, clientes y otros usuarios para comprender qué se está haciendo con suficiente detalle como para facilitar su propia participación. Los modelos y diagramas que mencionamos en el Capítulo 3 ayudan a esto, y deben utilizarse para describir la arquitectura. A medida que la gente se vaya familiarizando con UML, encontrarán más fácil de comprender la arquitectura modelada en ese lenguaje.

4.2.2. Organización del desarrollo

Cuanto mayor sea la organización del proyecto software, mayor será la sobrecarga de comunicación entre los desarrolladores para intentar coordinar sus esfuerzos. Esta sobrecarga se incrementa cuando el proyecto está geográficamente disperso. Dividiendo el sistema en subsistemas, con las interfaces claramente definidas y con un responsable o un grupo de responsables establecido para cada subsistema, el arquitecto puede reducir la carga de comunicación entre los grupos de trabajo de los diferentes subsistemas, tanto si están en el mismo edificio como si están en diferentes continentes. Una "buena" arquitectura es la que define explícitamente estas interfaces, haciendo que sea posible la reducción en la comunicación. Una interfaz bien definida "comunica" eficientemente a los desarrolladores de ambas partes qué necesitan saber sobre lo que los otros equipos están haciendo.

Las interfaces estables permiten que el software de ambas partes progrese independientemente. Una arquitectura y unos patrones de diseño (Apéndice C) adecuados nos ayudan a encontrar las interfaces correctas entre los subsistemas. Un ejemplo es el patrón Boundary-Control-Entity (*véase* la Sección 4.3.1), que nos ayuda a distinguir el comportamiento específico de los casos de uso, las clases de interfaz y las clases genéricas.

4.2.3. Fomento de la reutilización

Permítanos usar una analogía para explicar cómo la arquitectura es importante para la reutilización. La industria de la fontanería está estandarizada desde hace tiempo. Los contratistas de fontanería se benefician de componentes estándar. En lugar de esforzarse en encajar la dimensión de componentes "creativos", obtenidos de aquí y de allá, el fontanero los selecciona de un conjunto de componentes estandarizados que siempre se ajustarán.

Como el fontanero, los desarrolladores capaces de reutilizar conocen el **dominio del problema** (Apéndice C) y qué componentes especifica como adecuados la arquitectura. Los desarrolladores piensan en cómo conectar esos componentes para cumplir con los requisitos del sistema y realizar el modelo de casos de uso. Cuando tienen disponibles componentes reutilizables, los usan. Como los elementos estándar de fontanería, los componentes software reutilizables están diseñados y probados para *encajar*, y así el tiempo de construcción y el coste son menores. El resultado es predecible. Como en la industria de la fontanería, donde la estandarización llevó

siglos, en la estandarización del software se va avanzando con la experiencia —pero esperamos que se incremente la "componentización" de aquí a un par de años—. De hecho, ya ha comenzado.

La industria del software todavía tiene que alcanzar el nivel de estandarización que muchos dominios hardware han conseguido, pero las buenas arquitecturas y las interfaces bien definidas son pasos en esa dirección. Una buena arquitectura ofrece a los desarrolladores un andamio estable sobre el que trabajar. El papel de los arquitectos es definir ese andamiaje y los subsistemas reutilizables que el desarrollador pueda utilizar. Se obtienen subsistemas reutilzables diseñándolos con cuidado para que puedan ser utilizados conjuntamente [2]. Un buen arquitecto ayuda a los desarrolladores para que sepan dónde buscar elementos reutilizables de manera poco costosa, y para que puedan encontrar los componentes adecuados para ser reutilizados. El UML acelerará el proceso de "construcción de componentes", porque un lenguaje de modelado estándar es un prerrequisito para construir componentes específicos del dominio que puedan estar disponibles para su reutilización.

4.2.4. Evolución del sistema

Si hay algo de lo que podemos estar seguros, es de que cualquier sistema de un tamaño considerable evolucionará. Evolucionará incluso aunque aún esté en desarrollo. Más tarde, cuando esté en uso, el entorno cambiante provocará futuras evoluciones. Hasta que esto ocurra, el sistema debe ser fácil de modificar; esto quiere decir que los desarrolladores deberían ser capaces de modificar partes del diseño e implementación sin tener que preocuparse por los efectos inesperados que puedan tener repercusión en el sistema. En la mayoría de los casos, deberían ser capaces de implementar nuevas funcionalidades (es decir, casos de uso) en el sistema sin tener que pensar en un impacto dramático en el diseño e implementación existentes. En otras palabras, el sistema debe ser en sí mismo flexible a los cambios o tolerante a los cambios. Otra forma de enunciar este objetivo es afirmar que el sistema debe ser capaz de evolucionar sin problemas. Las arquitecturas del sistema pobres, por el contrario, suelen degradarse con el paso del tiempo y necesitan ser "parcheadas" hasta que al final no es posible actualizarlas con un coste razonable.

Ejemplo **El Sistema AXE de Ericsson - Sobre la importancia de la arquitectura**

El sistema de conmutación de telecomunicaciones de Ericsson se desarrolló inicialmente a principio de los setenta usando una primera versión de nuestros principios de las arquitecturas. La descripción de la arquitectura software es un artefacto importante que ha guiado el desarrollo completo del trabajo a lo largo de la vida del sistema. La arquitectura estaba guiada por un par de principios que ahora se han incorporado al Proceso Unificado.

Uno de estos principios era el de la modularización de funciones. Las clases o los elementos de diseño equivalentes se agruparon en bloques funcionales, o subsistemas de servicio, que los clientes podían considerar como opcionales (incluso si se entregaban a todos los clientes). Un subsistema de servicio tenía una fuerte cohesión interna. Los cambios en el sistema solían localizarse en un subsistema de servicio y raramente estos cambios afectaban a más de un servicio.

Otro de los principios era el de separar el diseño de interfaces del diseño de subsistemas de servicio. El objetivo era conseguir diseños "conectables" [1], donde muchos subsistemas de servicio po-

[1] Se ha optado por esta traducción para el término inglés "plug-able", aunque podría haberse mantenido el original debido a su amplia difusión.

dían soportar la misma interfaz. Cambiar un subsistema de servicio por otro podía hacerse sin cambiar los clientes del subsistema de servicio (los cuales dependían solamente de las interfaces, no del código del subsistema de servicio).

Un tercer principio era hacer corresponder los subsistemas de servicio en el diseño a uno o más componentes de la implementación. Los componentes de los subsistemas de servicio podían ser distribuidos en diferentes nodos de computación. Había exactamente un componente por cada nodo de proceso en el cual el subsistema de servicio podía ser ejecutado. Así, si el subsistema de servicio se ejecutaba en el ordenador central (servidor), entonces habría exactamente un componente para el subsistema. Si el subsistema estaba siendo implementado en ambos, clientes y servidor, habría dos componentes. Este principio simplificó la gestión de los cambios en el software en las diferentes instalaciones.

Todavía hay otro principio, que era el bajo acoplamiento entre los subsistemas de servicio. La única comunicación entre los subsistemas de servicio eran las señales. Aunque las señales eran asíncronas (semántica de envío sin respuesta), éstas no sólo soportaban encapsulación, sino también distribución.

Dado que su comienzo y su consiguiente desarrollo fue guiado por una arquitectura bien diseñada, el sistema AXE continua hoy en uso, con más de un centenar de clientes y varios miles de instalaciones. Se espera que continúe funcionando durante décadas, con los cambios necesarios.

4.3. Casos de uso y arquitectura

Ya hemos señalado que existe cierta interacción entre los casos de uso y la arquitectura. En el Capítulo 3 mostramos por primera vez cómo desarrollar un sistema que proporciona los casos de uso correctos a sus usuarios. Si el sistema proporciona los casos de uso correctos —casos de uso de alto rendimiento, calidad y facilidad de utilización— los usuarios pueden emplearlo para llevar a cabo sus objetivos. Pero, ¿cómo podemos conseguirlo? La respuesta, como ya hemos sugerido, es construir una arquitectura que nos permita implementar los casos de uso de una forma económica, ahora y en el futuro.

Vamos a clarificar cómo sucede esta interacción, observando primero qué influye en la arquitectura (*véase* la Figura 4.1) y después qué influye en los casos de uso.

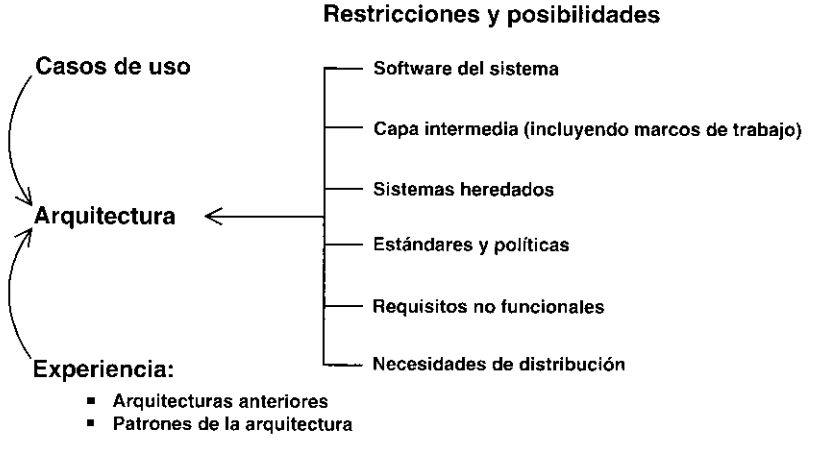


Figura 4.1. Existen diferentes tipos de requisitos y productos que influyen en la arquitectura, aparte de los casos de uso. También son de ayuda en el diseño de una arquitectura la experiencia de trabajos anteriores y las estructuras que podamos identificar como patrones de la arquitectura.

Como ya hemos dicho, la arquitectura está condicionada por los casos de uso que queremos que soporte el sistema; los casos de uso son *directores* de la arquitectura. Después de todo, queremos una arquitectura viable a la hora de implementar nuestros casos de uso. En las primeras iteraciones, elegimos unos pocos casos de uso, que pensamos que son los que nos ayudarán mejor en el diseño de la arquitectura. Estos casos de uso arquitectónicamente significativos incluyen los que son más necesarios para los clientes en la próxima versión y quizá para versiones futuras.

Sin embargo, la arquitectura no sólo se ve condicionada por los casos de uso arquitectónicamente significativos, sino también por los siguientes factores:

- Sobre qué productos software del sistema queremos desarrollar, como sistemas operativos o sistemas de gestión de bases de datos concretos.
- Qué productos de *middleware* (**capa de intermedia**; Apéndice C) queremos utilizar. Por ejemplo, tenemos que seleccionar un object request broker (ORB), que es un mecanismo para la conversión y envío de mensajes a objetos en entornos heterogéneos [6], o un marco de trabajo independiente de la plataforma, es decir, un subsistema "prefabricado", para construir interfaces gráficas.
- Qué sistemas heredados queremos utilizar en nuestro sistema. La utilización en nuestra arquitectura de un sistema heredado, como por ejemplo un sistema bancario existente, nos permite reutilizar gran parte de la funcionalidad existente, pero también tenemos que ajustar nuestra arquitectura para que encaje con el producto "antiguo".
- A qué estándares y políticas corporativas debemos adaptarnos. Por ejemplo, podemos elegir el Lenguaje de Definición de Interfaces (*Interface Definition Language*, IDL) [7] para especificar todas las interfaces de las clases, o el estándar TMN de telecomunicaciones [8] para especificar objetos en nuestro sistema.
- Requisitos no funcionales generales (no específicos de los casos de uso), como los requisitos de disponibilidad, tiempo de recuperación, o uso de memoria.
- Las necesidades de distribución especifican cómo distribuir el sistema, quizá a través de una arquitectura cliente/servidor.

Podemos pensar en los elementos de la parte derecha de la Figura 4.1 como en restricciones y posibilidades que nos llevan a forjar la arquitectura de una forma determinada.

La arquitectura se desarrolla en iteraciones de la fase de elaboración. La siguiente podría ser una aproximación simplificada, de algún modo ingenua. Comenzamos determinando un diseño de alto nivel para la arquitectura, a modo de una **arquitectura en capas** (Apéndice C). Después formamos la arquitectura en un par de **construcciones** (Apéndice C; *véase* también el Capítulo 10) dentro de la primera iteración.

En la primera construcción, trabajamos con las partes **generales de la aplicación** (Apéndice C), que son generales en cuanto al dominio, y que no son específicas del sistema que pensamos desarrollar (es decir, seleccionamos el software del sistema (capa de software del sistema; Apéndice C; *véase* también la Sección 9.5.1.2.2), el middleware, los sistemas heredados, los estándares y las políticas de uso). Decidimos qué nodos contendrá nuestro modelo de desarrollo y cómo deben interactuar entre ellos. También decidimos cómo manejar los requisitos generales no funcionales, así como la disponibilidad de estos requisitos. Con la primera pasada es suficiente para tener una visión general del funcionamiento de la aplicación.

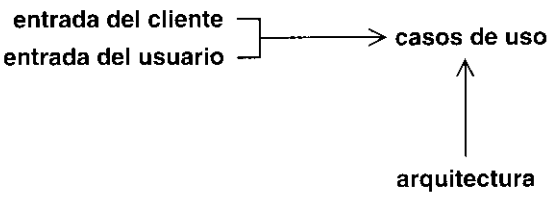


Figura 4.2. Los casos de uso pueden ser desarrollados de acuerdo a las entradas de los clientes y de los usuarios. No obstante, también se ven influenciados por la arquitectura seleccionada.

En la segunda construcción, trabajamos con los aspectos de la arquitectura específicos de la aplicación (capa específica de la aplicación; Apéndice C). Escogemos un conjunto de casos de uso relevantes en cuanto a la arquitectura, capturamos los requisitos, los analizamos, los diseñamos, los implementamos y los probamos. El resultado serán nuevos subsistemas implementados como componentes del desarrollo que soportan los casos de uso seleccionados. Pueden existir también algunos cambios en los componentes significativos de la arquitectura que implementamos en la primera entrega (cuando no pensamos en términos de casos de uso). Los componentes nuevos o cambiados se desarrollan para realizar los casos de uso, y de esta forma la arquitectura se adapta para ajustarse mejor a los casos de uso. Entonces elaboraremos otra construcción, y así sucesivamente hasta terminar con las iteraciones. Si este final de las iteraciones tiene lugar en el final de la fase de elaboración, habremos conseguido una arquitectura estable.

Cuando tenemos una arquitectura estable, podemos implementar la funcionalidad completamente realizando el resto de casos de uso durante la fase de construcción. Los casos de uso implementados durante la fase de construcción se desarrollan utilizando como entradas los requisitos de los clientes y de los usuarios (*véase* la Figura 4.2), pero los casos de uso están también influenciados por la arquitectura elegida en la fase de elaboración.

Según vayamos capturando nuevos casos de uso, vamos utilizando el conocimiento que ya tenemos de la arquitectura existente para hacer mejor nuestro trabajo. Cuando calculamos el valor y el coste de los casos de uso que se sugieren, lo hacemos a la luz de la arquitectura que tenemos. Algunos casos de uso serán más fáciles de implementar, mientras que otros serán más difíciles.

Ejemplo **Adaptación de los casos de uso a la arquitectura ya existente**

El cliente ha requerido una función que supervise el proceso de carga. Esto se especificó como un caso de uso que midiera la carga, con un nivel de prioridad alto en el computador. La implementación de ese caso de uso podría haber requerido algunos cambios en sistema operativo en tiempo real que se estaba utilizando. Entonces, el equipo de desarrollo sugirió que la funcionalidad requerida fuera implementada en un dispositivo externo separado que hiciera llamadas al sistema y midiera el tiempo de respuesta. El cliente obtuvo mayor fiabilidad en las medidas y el equipo de desarrollo evitó tener que cambiar partes críticas de la arquitectura subyacente.

Negociamos con el cliente y decidimos si los casos de uso podrían cambiarse para hacer la implementación más sencilla, ajustando los casos de uso y el diseño resultante con la arqui-

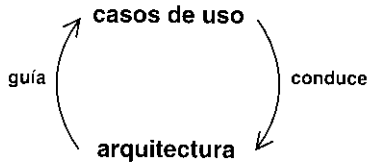


Figura 4.3. Los casos de uso conducen el desarrollo de la arquitectura, y la arquitectura indica qué casos de uso pueden realizarse.

tectura que ya tenemos. Este ajuste significa que debemos considerar que ya tenemos los subsistemas, interfaces, casos de uso, realizaciones de casos de uso, clases y demás. Ajustando los casos de uso con la arquitectura, podemos crear nuevos casos de uso, subsistemas y clases con poco esfuerzo, partiendo de las que ya existen.

Así, por una parte, la arquitectura está influenciada por aquellos casos de uso que queremos que el sistema soporte. Los casos de uso conducen la arquitectura. Por otra parte, utilizamos nuestro conocimiento de la arquitectura para hacer mejor el trabajo de captura de requisitos, para obtener casos de uso. La arquitectura guía los casos de uso (*véase* la Figura 4.3).

¿Qué es primero, la arquitectura o los casos de uso? Tenemos otra vez el típico problema del "huevo y la gallina". La mejor forma de resolver estos problemas es mediante una iteración. Primero, construimos una arquitectura tentativa básica a partir de una buena comprensión del **área del dominio** (Apéndice C), pero sin considerar los casos de uso detallados. Entonces, escogemos un par de casos de uso y ampliamos la arquitectura adaptándola para que soporte esos casos de uso. Después, escogemos algunos casos de uso más y construimos una arquitectura todavía mejor, y así sucesivamente. Con cada iteración, escogemos e implementamos un conjunto de casos de uso para validar, y si es necesario, mejorar la arquitectura. Con cada iteración también implementamos además las partes de la arquitectura específicas de la aplicación basadas en los casos de uso que hemos seleccionado. Los casos de uso entonces nos ayudan a mejorar gradualmente la arquitectura según vamos iterando para completar el sistema. Éste es uno de los beneficios de los desarrollos conducidos por casos de uso. Volveremos sobre este enfoque en el Capítulo 5.

Resumiendo, una buena arquitectura es algo que nos permite obtener los casos de uso correctos, de manera económica, hoy y en el futuro.

4.4. Los pasos hacia una arquitectura

La arquitectura se desarrolla mediante iteraciones, principalmente durante la fase de elaboración. Cada iteración se desarrolla como se esbozó en el Capítulo 3, comenzando con los requisitos y siguiendo con el análisis, diseño, implementación y pruebas, pero centrándonos en los casos de uso relevantes desde el punto de vista de la arquitectura y en otros requisitos. El resultado al final de la fase de elaboración es una línea base de la arquitectura —un esqueleto del sistema con pocos "músculos" de software.

¿Qué casos de uso son relevantes arquitectónicamente hablando? Plantearemos esta cuestión en la Sección 12.6. Por ahora, es suficiente con decir que los casos de uso arquitectónicamente relevantes son aquellos que nos ayudan a mitigar los riesgos más importantes, aquellos que

son los más importantes para los usuarios del sistema, y aquellos que nos ayudan a cubrir todas las funcionalidades significativas, de forma que nada quede en penumbra. La implementación, integración y prueba de la línea base de la arquitectura proporciona seguridad al arquitecto y a otros trabajadores de su equipo, por lo que comprender estos puntos es algo francamente operativo. Esto es algo que no puede obtenerse mediante un análisis y diseño "sobre el papel". La línea base de la arquitectura de operación proporciona una demostración que funciona para que los trabajadores puedan proporcionar sus retroalimentaciones.

4.4.1. La línea base de la arquitectura es un sistema "pequeño y flaco"

Al final de la fase de elaboración hemos desarrollado modelos del sistema que representan los casos de uso más importantes y sus realizaciones, desde la perspectiva de la arquitectura. También hemos decidido, como ya se discutió en la Sección 4.3, "Casos de uso y arquitectura", con qué estándares contamos, qué software del sistema y qué middleware utilizar, qué sistemas heredados reutilizar y qué necesidades de distribución tenemos. Así, tendremos una primera versión de los modelos de casos de uso, de análisis, de diseño y demás. Esta agregación de modelos (*véase* la Figura 4.4) es la línea base de la arquitectura; es un sistema pequeño y flaco[2]. Tiene las versiones de todos los modelos que un sistema terminado contiene al final de la fase de construcción. Incluye el mismo esqueleto de subsistemas, componentes y nodos que un sistema definitivo, pero no existe toda la musculatura. No obstante, contiene comportamiento y código ejecutable. El sistema flaco se desarrollará para convertirse en un sistema hecho y derecho, quizás con algunos cambios sin importancia en su estructura y comportamiento. Los cambios son menores porque al final de la fase de elaboración hemos definido una arquitectura estable; si no, la fase de elaboración debe continuar hasta que alcance su objetivo.

En la Figura 4.4, la parte sombreada de cada modelo representa la versión del mismo que está desarrollada al final de la fase de elaboración, es decir, la versión del modelo que es parte de la línea base de la arquitectura. El rectángulo completo (la parte sombreada y sin sombrear) representa la versión del modelo como desarrollo al final de la fase de transición, por tanto, la línea base representa la versión del cliente (el lector no debería sacar ninguna conclusión del tamaño de las áreas sombreadas que se muestran en la Figura 4.4, que sólo tienen finalidad ilustrativa). Desde la línea base de la arquitectura hasta la versión de la línea base del cliente, habrá algunas otras que representen las **versiones internas** de los modelos (Apéndice C). Podríamos haber mostrado esas nuevas versiones del modelo como incrementos partiendo de la línea base de la arquitectura. Cada nueva versión de un modelo puede desarrollarse a partir de la anterior versión. Los diferentes modelos de la Figura 4.4, como es natural, no se desarrollan independientemente los unos de los otros. Cada caso de uso del modelo de casos de uso corresponde, por ejemplo, a una realización de caso de uso en los modelos de análisis y diseño, y a una prueba en el modelo de pruebas. Los procesos y la estructura de los nodos deben ajustarse a la ejecución requerida por los casos de uso (en caso contrario, el modelo de casos de

[2] Esto no es del todo correcto. Al final de la fase de elaboración, tenemos una versión del modelo de casos de uso que contiene los casos de uso arquitectónicamente significativos y los casos de uso (más del 80 por ciento) que necesitamos tener especificados para hacer un análisis del negocio. Así, la línea base de la arquitectura tiene más del modelo de casos de uso y del modelo de análisis que lo que la Figura 4.4 indica. No obstante, para nuestro propósito en este momento, podemos hacer esta simplificación.

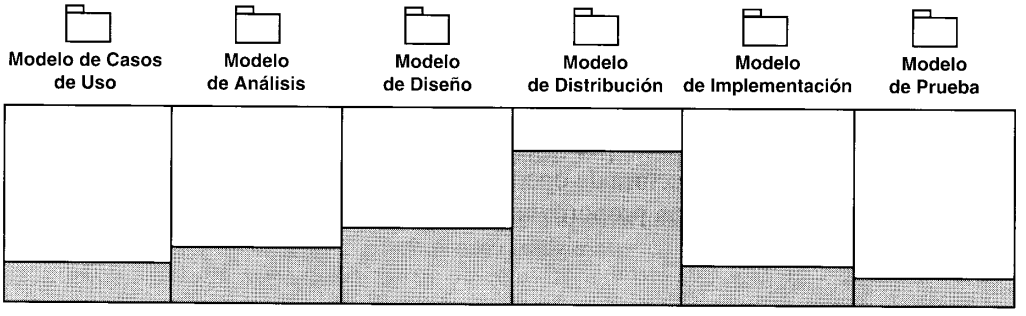


Figura 4.4. La línea base de la arquitectura es una versión interna del sistema, que está centrada en la descripción de la arquitectura.

uso o el modelo de despliegue deben modificarse, quizás cambiando la forma en que las clases activas se asignan a los nodos para una mejor ejecución. Tales cambios en el desarrollo o en el modelo de diseño pueden conducir a alteraciones en el modelo de casos de uso, si es que los cambios lo requieren). Los elementos del modelo, en los diferentes modelos, están, como se dijo en la Sección 2.3.7, relacionados entre sí a través de las dependencias de traza.

No obstante, la línea base de la arquitectura, esto es, la versión interna del sistema al final de la fase de elaboración, se representa por algo más que los artefactos del modelo. También se incluye la descripción de la arquitectura. Esta descripción se desarrolla habitualmente de forma concurrente, a menudo incluso antes que las actividades que obtienen las versiones del modelo que son parte de la línea base de la arquitectura. El papel de la descripción de la arquitectura es guiar al equipo de desarrollo a través del ciclo de vida del sistema —no sólo por las iteraciones del ciclo actual, sino por todos los ciclos que vengan—. Éste es el estándar a seguir por todos los desarrolladores, ahora y en el futuro. Como la arquitectura debería ser estable, el estándar debería ser estable también.

La descripción de la arquitectura puede adoptar diferentes formas. Puede ser un extracto (de las versiones) de los modelos que son parte de la línea base de la arquitectura, o puede ser una reescritura de los extractos de forma que sea más fácil leerlos. Volveremos a esto en la Sección 4.4.3, "Descripción de la arquitectura". En cualquier caso, incluye extractos o vistas de los modelos que son parte de la línea base de la arquitectura. A medida que se desarrolla el sistema y los modelos se van haciendo más voluminosos en las últimas fases, la arquitectura seguirá incluyendo vistas de las nuevas versiones de los modelos. Asumiendo que la línea base de la arquitectura ha desarrollado una arquitectura estable —esto es, los elementos del modelo relevantes arquitectónicamente no cambian en las sucesivas iteraciones—, la descripción de la arquitectura también será estable, e incluirá en todo momento vistas de los modelos del sistema.

Es fascinante observar que resulta posible desarrollar una arquitectura estable durante la fase de elaboración del primer ciclo de vida, cuando solamente se ha invertido aproximadamente un 30 por ciento de la primera versión. Esta arquitectura constituirá los pilares del sistema el resto de su vida. Aunque los cambios en los pilares resultarán costosos, y en algunos casos muy difíciles, es importante obtener una arquitectura estable pronto en el desarrollo del trabajo. El desarrollo de una arquitectura para un sistema en particular es, por una parte, la creación de algo nuevo. Por otro lado, la gente lleva desarrollando arquitecturas muchos años. Se tiene experiencia y conocimiento en el desarrollo de buenas arquitecturas. Hay muchas "soluciones" genéricas —estructuras, colaboraciones y arquitecturas físicas— que han evolucionado a lo

largo de muchos años y con las que todo arquitecto con experiencia debería estar familiarizado. Estas soluciones se llaman habitualmente patrones, como los patrones de arquitectura descritos en [4] y los patrones de diseño descritos en [5]. Los patrones genéricos son recursos con los que los arquitectos pueden contar.

4.4.2. Utilización de patrones de la arquitectura

Las ideas del arquitecto Christopher Alexander sobre cómo los "lenguajes de patrones" se utilizan para sistematizar principios y prácticas importantes en el diseño de edificios y comunidades, han inspirado a muchos miembros de la comunidad de la orientación a objetos a definir, coleccionar y probar una gran variedad de patrones software [10]. La "comunidad de patrones" define un patrón como "una solución a un problema de diseño que aparece con frecuencia". Muchos de los patrones de diseño están documentados en libros, que presentan los patrones utilizando plantillas estándar. Estas plantillas asignan un nombre a un patrón y presentan un resumen de los problemas y las fuerzas que lo hacen surgir, una solución en términos de colaboración de clases participantes e interacción entre objetos de esas clases. Las plantillas también proporcionan ejemplos de cómo se utiliza el patrón en algunos lenguajes de programación, junto con variantes del patrón, un resumen con las ventajas y las consecuencias de la utilización de patrones, y referencias a estos. Según Alexander, sería bueno que los ingenieros de software aprendiesen los nombres y el objetivo de muchos patrones estándar, y que los aplicasen para hacer diseños mejores y más compresibles. Existen patrones de diseño como Facade, Decorator, Proxy Observer, Strategy y Visitor ampliamente citados y utilizados.

La comunidad de patrones también ha aplicado esta idea, con una plantilla de documento ligeramente modificada, para recoger soluciones estándar a problemas de la arquitectura que ocurren frecuentemente. Algunos de estos patrones incluyen Layers, Pipes and Filters, Broker Blackboard, Horizontal-Vertical Metadata y MVC. Otros han desarrollado patrones para que se utilicen durante el análisis ("patrones de análisis"), durante la implementación ("idiomas" que hacen corresponder estructuras comunes de orientación a objetos con aspectos peculiares de los lenguajes, como C ++ o Smalltalk), e incluso para estructuras de organización efectivas ("patrones de organización"). Típicamente los patrones de diseño se implementan de una forma muy directa en lenguajes orientados a objetos. Algunos ejemplos pueden ser C++, Java y Smalltalk, mientras que los patrones de arquitectura se manejan mejor con sistemas o subsistemas e interfaces, y los ejemplos no incluyen habitualmente código. Para ver un buen esquema de clasificación *véase* [9].

Desde nuestra perspectiva dirigida por modelos, definiremos *patrón* como una plantilla de colaboración, que es una colaboración general que puede especializarse según lo definido en la **plantilla** (Apéndice A). Por tanto, consideramos los patrones de diseño como colaboraciones entre clases e instancias, con su comportamiento explicado en los diagramas de colaboración. Utilizaremos plantillas de colaboración ya que entendemos que las soluciones son bastante generales. Utilizaremos herencia, extensión y otros mecanismos para especializar el patrón (especificando los nombres de clases, número de clases, etc., que aparecen en la plantilla). En muchos casos, cuando especializamos las plantillas de colaboración, surgen colaboraciones concretas que pueden trazarse directamente con los casos de uso. *Véase* [5] para un tratamiento extenso de los patrones de diseño.

Los patrones de las arquitecturas se utilizan de una forma parecida pero se centran en estructuras e interacciones de grano más grueso, entre subsistemas e incluso entre sistemas.

Existen muchos patrones de arquitectura, pero aquí sólo trataremos brevemente algunos de los más interesantes.

El patrón Broker [4] es un mecanismo genérico para la gestión de objetos distribuidos. Permite que los objetos hagan llamadas a otros objetos remotos a través de un gestor que redirige la llamada al nodo y al proceso que guardan al objeto deseado. Esta redirección se hace de manera transparente, lo cual quiere decir que el llamante no necesita saber si el objeto llamado es remoto. El patrón Broker suele utilizar el patrón de diseño Proxy, que proporciona un objeto sustituto local con la misma interfaz que el objeto remoto para hacer transparentes el estilo y los detalles de la comunicación distribuida.

Hay otros patrones que nos ayudan a comprender el hardware de los sistemas que construimos y que nos ayudan a diseñar nuestro sistema sobre él, como por ejemplo Client/Server, Three-Tier, y Peer-to-Peer. Estos patrones definen una estructura para el modelo de despliegue y sugieren cómo se deben asignar los componentes a los nodos. En la Sección 4.5, ilustraremos cómo se pudo aplicar el patrón Client/Server al sistema CA descrito en el Capítulo 3. En nuestro ejemplo, la distribución cliente/servidor tiene un nodo cliente que ejecuta el código de interfaces de usuario y parte de la lógica de negocio (clases de control) en cada CA "físico". El nodo servidor mantiene las cuentas y las reglas de negocio que permiten verificar las transacciones.

El patrón Layers es aplicable a muchos tipos de sistemas. Este patrón define cómo organizar el modelo de diseño en capas, lo cual quiere decir que los componentes de una capa sólo pueden hacer referencia a componentes en capas inmediatamente inferiores. Este patrón es importante porque simplifica la comprensión y la organización del desarrollo de sistemas complejos, reduciendo las dependencias de forma que las capas más bajas no son conscientes de ningún detalle o interfaz de las superiores. Además, nos ayuda a identificar qué puede reutilizarse, y proporciona una estructura que nos ayuda a tomar decisiones sobre qué partes comprar y qué partes construir.

Un sistema con una arquitectura en capas pone a los subsistemas de aplicación individuales en lo más alto. Estos se construyen a partir de subsistemas en las capas más bajas, como son los marcos de trabajo y las bibliotecas de clases. Observemos la Figura 4.5. La capa general de aplicación contiene los subsistemas que no son específicos de una sola aplicación, sino que pueden

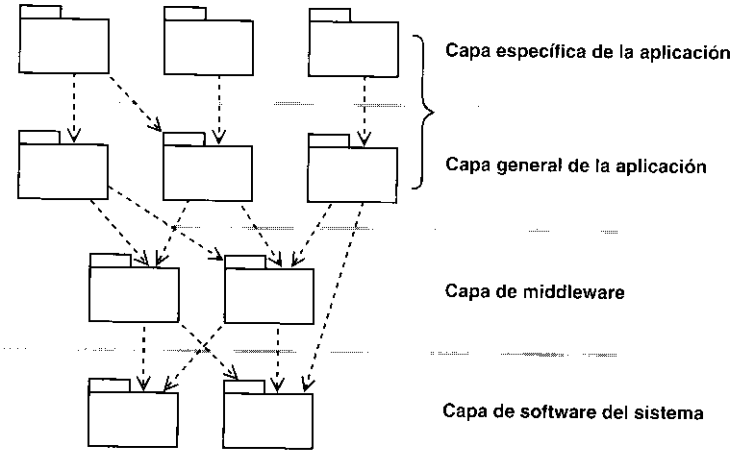


Figura 4.5. La arquitectura en capas organiza los sistemas en capas de subsistemas.

ser reutilizados por muchas aplicaciones diferentes dentro del mismo dominio o negocio. La arquitectura de las dos capas inferiores puede establecerse sin considerar los casos de uso debido a que no son dependientes del negocio. La arquitectura de las dos capas superiores se crea a partir de los casos se uso significativos para la arquitectura (estas capas son dependientes del negocio).

Una **capa** (Apéndice C) es un conjunto de subsistemas que comparten el mismo grado de generalidad y de volatilidad en las interfaces: las capas inferiores son de aplicación general a varias aplicaciones y deben poseer interfaces más estables, mientras que las capas más altas son más dependientes de la aplicación y pueden tener interfaces menos estables. Debido a que las capas inferiores cambian con menor frecuencia, los desarrolladores que trabajan en las capas superiores pueden construir sobre capas inferiores estables. Subsistemas en diferentes capas pueden reutilizar casos de uso, otros subsistemas de más bajo nivel, clases, interfaces, colaboraciones, y componentes de las capas inferiores. Podemos aplicar sobre un mismo sistema muchos patrones de arquitectura. Los patrones que estructuran el modelo de despliegue (es decir, Client/Server, Three-Tier, o Peer-to-Peer) pueden combinarse con el patrón Layers, lo cual nos ayuda a estructurar el modelo de diseño. Los patrones que tratan estructuras en diferentes modelos son a menudo independientes unos de otros. Incluso los patrones que tratan el mismo modelo suelen poder combinarse bien mutuamente. Por ejemplo, el patrón Broker se combina correctamente con el patrón Layers, y ambos se utilizan en el modelo de diseño. El patrón Broker se encarga de cómo tratar con la distribución transparente de objetos, mientras que el patrón Layers nos indica cómo organizar el diseño entero. De hecho, el Patrón Broker puede interpretarse como un subsistema en la capa intermedia[3].

Obsérvese que a veces un patrón es predominante. Por ejemplo, en un sistema en capas, el patrón Layers define la arquitectura general y la descomposición del trabajo (cada capa asignada a un grupo diferente), mientras que se pueden utilizar Pipes y Filters dentro de una o más capas. En contraste, en un sistema basado en Pipes y Filters, mostraríamos la arquitectura general como un flujo entre filtros, mientras que la división en capas podría utilizarse de forma explícita para algunos filtros.

4.4.3. Descripción de la arquitectura

La línea base de la arquitectura desarrollada en la fase de elaboración sobrevive, como dijimos en la Sección 4.4.1, en forma de una descripción de la arquitectura. Esta descripción se obtiene de versiones de los diferentes modelos que son resultado de la fase de elaboración, como se muestra en la Figura 4.6. La descripción de la arquitectura es un extracto o, en nuestros términos, un conjunto de vistas —quizá con una reescritura cuidada para hacerlas más legibles— de los modelos que están en la línea base de la arquitectura. Estas vistas incluyen los elementos arquitectónicamente significativos. Por supuesto, muchos de los elementos del modelo que son parte de la línea base de la arquitectura aparecerán también en la descripción de la arquitectura. Sin embargo, no lo harán todos ellos, debido a que para obtener una línea base operativa puede ser necesario el desarrollo de algunos elementos del modelo que no son arquitectónicamente interesantes, pero que se necesitan para generar código ejecutable. Debido a que la línea base de la arquitectura no sólo se usa para desarrollar una arquitectura, sino también para especificar los requisitos del sistema en un nivel que permita el desarrollo de un plan detallado, el modelo de casos de uso de esta línea base puede contener también más casos de uso aparte de los interesantes desde el punto de vista de la arquitectura.

---

[3] *N. del T.* También se suele utilizar el término "capa middleware".

La descripción de la arquitectura debe mantenerse actualizada a lo largo de la vida del sistema para reflejar los cambios y las adiciones que son relevantes para la arquitectura. Estos cambios son normalmente secundarios y pueden incluir:

- La identificación de nuevas clases abstractas e interfaces.
- La adición de nueva funcionalidad a los subsistemas existentes.
- La actualización a nuevas versiones de los componentes reutilizables.
- La reordenación de la estructura de procesos.

Puede que tengamos que modificar la propia descripción de la arquitectura, pero su tamaño no debe crecer. Sólo se actualiza para ser relevante (*véase* la Figura 4.6).

Como dijimos anteriormente, la descripción de la arquitectura presenta vistas de los modelos. Esto incluye casos de uso, subsistemas, interfaces, algunas clases y componentes, nodos y colaboraciones. La descripción de la arquitectura también incluye requisitos significativos para la arquitectura que no están descritos por medio de los casos de uso. Estos otros requisitos son no funcionales y se especifican como requisitos adicionales, como aquellos relativos a la seguridad, e importantes restricciones acerca de la **distribución** y la **concurrencia** (Apéndice C; *véase* también la Sección 9.3.2). La descripción de la arquitectura debería incluir también una breve descripción de la plataforma, los sistemas heredados, y el software comercial que se utilizará, como por ejemplo la invocación de métodos remotos de Java (*Remote Method Invocation*, RMI) para la distribución de objetos. Es más, es importante describir los marcos de trabajo que implementan **mecanismos** (Apéndice C; *véase* también 9.5.1.4) genéricos, como el almacenamiento y recuperación de un objeto en una base de datos relacional. Estos mecanis-

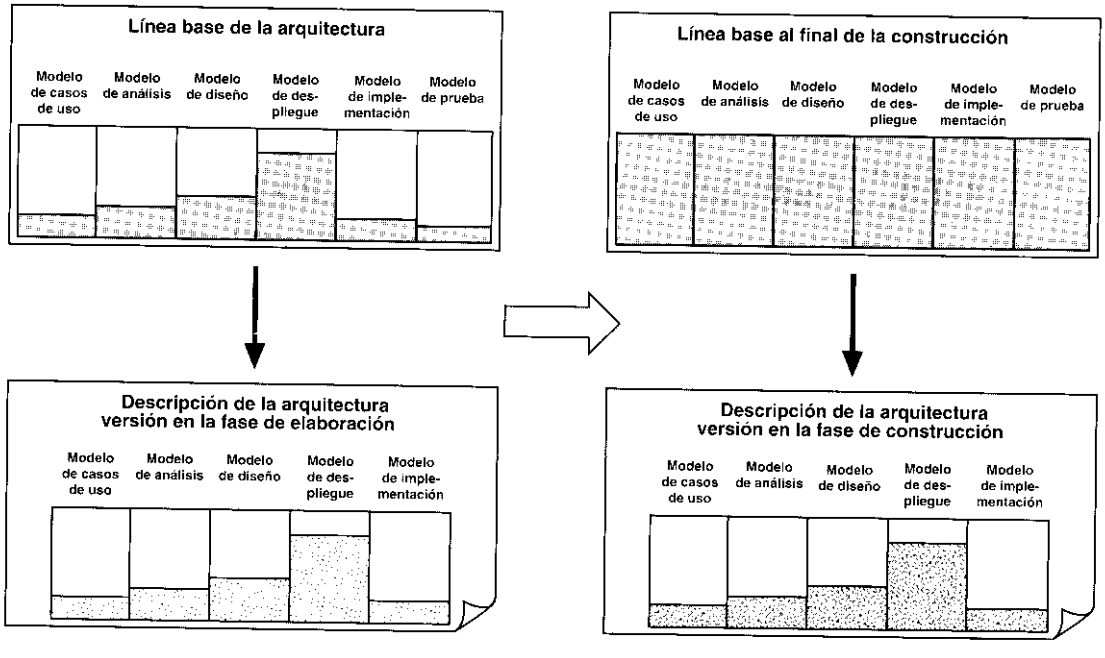


Figura 4.6. Durante la construcción, los diversos modelos van creciendo hasta completarse (según se muestra con las formas rellenas en la esquina superior derecha). La descripción de la arquitectura, sin embargo, no crece significativamente (abajo a la derecha) debido a que la mayor parte de la arquitectura se definió durante la fase de elaboración. Se incorporan pocos cambios secundarios a la arquitectura (indicados por un relleno punteado).

mos pueden reutilizarse en varias realizaciones de caso de uso ya que se han diseñado para llevar a cabo colaboraciones reutilizables. La descripción de la arquitectura también debería documentar todos los patrones de arquitectura que se han utilizado.

La descripción de la arquitectura subraya los temas de diseño más importantes y los expone para ser considerados y para obtener la opinión de otros. Después se deben tratar, analizar y resolver estos temas. Estos análisis pueden, por ejemplo, incluir una estimación de la carga del rendimiento, o requisitos de memoria, e imaginar requisitos futuros que podrían romper la arquitectura.

Aunque esté detallada en lo necesario, la descripción de la arquitectura es aún una vista de alto nivel. Por un lado, no se pretende que cubra todo; no debería inundar a los participantes con una cantidad desbordante de detalle. Es un mapa de carreteras, no una especificación detallada del sistema entero. Por otro lado, debe representar lo que cada participante necesita, por lo que incluso puede que 100 páginas no sean excesivas. La gente utilizará un documento grande si contiene lo que necesitan en una forma que sea rápidamente comprensible. Después de todo, eso es lo que se espera de una descripción de la arquitectura: debería contener lo que los desarrolladores necesitan para hacer sus trabajos.

Cuando leemos una descripción de la arquitectura, nos puede parecer que trata algunos de los subsistemas de manera superficial, mientras que especifica en detalle las interfaces y colaboraciones de un puñado de otros subsistemas. La razón para este tratamiento distinto es que los subsistemas muy especificados son significativos para la arquitectura, y deberían mantenerse bajo el control del arquitecto (*véase* las Secciones 12.4.2 y 14.4.3.1).

Puede ser útil el tratar qué no es una arquitectura. La mayoría de las clases, con operaciones, interfaces, y atributos que son privadas en los subsistemas o en los subsistemas de servicio (ocultas al resto del sistema), no son significativas para la arquitectura. Los subsistemas que son variantes de otros subsistemas no son importantes desde una perspectiva de la arquitectura. La experiencia indica que menos del 10 por ciento de las clases son relevantes para la arquitectura. El 90 por ciento restante no es significativo porque no es visible al resto del sistema. Un cambio en una de ellas no afecta a nada esencial fuera del subsistema de servicio. Tampoco son arquitectónicamente relevantes la mayoría de las realizaciones de casos de uso debido a que no imponen ninguna restricción adicional al sistema. Éste es el motivo por el cual los arquitectos pueden planificar una arquitectura partiendo sólo de una fracción de los casos de uso y de otros requisitos. La mayoría de las realizaciones de caso de uso representan simple comportamiento añadido fácil de implementar incluso a pesar de que constituyen la mayoría de las funciones que el sistema ofrece. Y ésta es la clave: la mayoría de la funcionalidad del sistema es en realidad fácil de implementar una vez que hemos establecido la arquitectura.

La descripción de la arquitectura no incluye información que sea sólo necesaria para validar o verificar la arquitectura. Por tanto no tiene casos o procedimientos de prueba, y no incluye una vista de la arquitectura del modelo de prueba. Estas cuestiones no son de arquitectura. Sin embargo, como puede verse en la Figura 4.6, la línea base de la arquitectura contiene una versión de todos los modelos, incluida una versión del modelo de prueba. Por tanto, la línea base subyacente a la descripción de la arquitectura contiene pruebas realizadas —todas las líneas base las incluyen.

4.4.4. El arquitecto crea la arquitectura

El arquitecto crea la arquitectura junto con otros desarrolladores. Trabajan para conseguir un sistema que tendrá un alto rendimiento y una alta calidad, y será completamente funcional,

verificable, amigable para el usuario, fiable, de alta disponibilidad, preciso, extensible, tolerante a cambios, robusto, mantenible, portable, confiable, seguro, y económico. Ellos saben que han de convivir con esas restricciones y que tendrán que tomar soluciones de compromiso entre ellas —éste es el motivo por el que hay un arquitecto—. El arquitecto posee la responsabilidad técnica más importante en estos aspectos y selecciona entre patrones de arquitectura y entre productos para establecer las dependencias entre subsistemas para cada uno de esos distintos intereses. Aquí la separación de intereses significa la creación de un diseño donde un cambio en un subsistema no retumba en otros varios subsistemas.

El verdadero objetivo es cumplir con las necesidades de la aplicación de la mejor forma posible con el estado actual de la tecnología y con un coste que la aplicación pueda soportar, en otras palabras, ser capaz de implementar la funcionalidad de la aplicación (es decir, los casos de uso) de manera económica, ahora y en el futuro. En este punto el arquitecto tiene el soporte de UML y del Proceso Unificado. UML posee construcciones potentes para la formulación de la arquitectura, y el Proceso Unificado nos ofrece directrices detalladas sobre lo que constituye una buena arquitectura. Incluso así, al final, la arquitectura seleccionada es el resultado de un juicio basado en aptitudes y experiencia. El arquitecto es el responsable de emitir este juicio. Cuando el arquitecto presenta la descripción de la arquitectura, al término de la fase de elaboración, al jefe de proyecto, está queriendo decir: "Ahora sé que podemos construir el sistema sin encontrar ninguna sorpresa técnica importante."

Un arquitecto cualificado se consigue mediante dos tipos de aptitudes. Una es el conocimiento del dominio en el que trabaja, por lo que debe trabajar adquiriendo experiencia con todos los usuarios —no sólo con los desarrolladores—. El otro es el conocimiento del desarrollo de software, incluso debe ser capaz de escribir código, ya que debe comunicar la arquitectura a los desarrolladores, coordinar sus esfuerzos, y obtener su retroalimentación. También es valioso que el arquitecto tenga experiencia con sistemas similares al que se está desarrollando.

El arquitecto ocupa un puesto difícil en la organización de desarrollo. No debería ser jefe de proyecto, ya que ese puesto tiene muchas dificultades además de la arquitectura. Debe contar con el compromiso incondicional de la dirección, tanto para crear la arquitectura por primera vez, como para forzar a que se cumpla. Además debe ser lo bastante flexible como para encajar las opiniones útiles de los desarrolladores y otros implicados. Éste es un breve resumen de lo que el arquitecto pone sobre la mesa. Puede que un solo arquitecto no sea suficiente para sistemas grandes. En su lugar, puede ser una sabia decisión el tener un grupo de arquitectura para desarrollarla y mantenerla.

El desarrollo de la arquitectura consume un tiempo de calendario considerable. Este tiempo está al principio del calendario de desarrollo y puede incomodar a los directores que están acostumbrados a ver dedicado el tiempo de desarrollo a la implementación y pruebas en su mayor parte. Sin embargo, la experiencia indica que la duración total del desarrollo desciende marcadamente cuando es una buena arquitectura la que guía las últimas fases. Esto es algo que comentaremos en el Capítulo 5.

4.5. ¡Por fin una descripción de la arquitectura!

Hemos estado hablando bastante tiempo sobre lo que es la arquitectura sin ofrecer un ejemplo significativo. Presentamos ahora un ejemplo concreto de la apariencia de una descripción de arquitectura. Sin embargo, antes tenemos que explicar por qué no es fácil hacerlo.

Recuérdese que la descripción de la arquitectura es sencillamente un extracto adecuado de los modelos del sistema (es decir, no añade nada nuevo). La primera versión de la descripción de la arquitectura es un extracto de la versión de los modelos que tenemos al término de la fase de elaboración en el primer ciclo de vida. Dado que no intentamos hacer una reescritura más legible de esos extractos, la descripción de la arquitectura se parece mucho a los modelos normales del sistema. Esta apariencia significa que la vista de la arquitectura del modelo de casos de uso es muy parecida a un modelo de casos de uso normal. La única diferencia reside en que la vista de la arquitectura sólo contiene los casos de uso significativos para la arquitectura (*véase* la Sección 12.6.2), mientras que el modelo de casos de uso final contiene todos los casos de uso. Lo mismo ocurre con la vista de la arquitectura del modelo de diseño. Es igual que un modelo de diseño, pero sólo representa los casos de uso que son interesantes para la arquitectura.

Otra razón por la cual es difícil ofrecer un ejemplo es que sólo es interesante hablar de la arquitectura en sistemas reales, y cuando queremos hablar aquí sobre un sistema en detalle, debe ser por necesidad un sistema pequeño. Sin embargo, vamos a emplear el ejemplo del CA del Capítulo 3 para ilustrar lo que podrían llevar las vistas de la arquitectura. Lo haremos comparando lo que debería estar en las vistas y lo que debería estar en los modelos completos del sistema.

La descripción de la arquitectura tiene cinco secciones, una para cada modelo. Tiene una vista del modelo de casos de uso, una vista del modelo de análisis (que no siempre se mantiene), una vista del modelo de diseño, una vista del modelo de despliegue, y una vista del modelo de implementación. No incluye una vista del modelo de prueba porque no desempeña ningún papel en la descripción de la arquitectura, y sólo se utiliza para verificar la línea base de la arquitectura.

4.5.1. La vista de la arquitectura del modelo de casos de uso

La vista de la arquitectura del modelo de casos de uso presenta los actores y casos de uso más importantes (o escenarios de esos casos de uso). *Véase* la Sección 3.3, "La captura de casos de uso", en lo relativo al modelo de casos de uso del sistema de CA.

Ejemplo **La vista de la arquitectura del modelo de casos de uso del sistema de CA**

En el ejemplo del CA, el caso de uso más importante es Sacar Dinero. Sin él, no existiría un verdadero sistema de CA. Los casos de uso Ingresar Dinero y Transferencia entre Cuentas se consideran menos importantes para un cliente de banco normal.

Para definir la arquitectura, el arquitecto sugiere por tanto que el caso de uso Sacar Dinero se implemente en su totalidad durante la fase de elaboración, pero ningún otro caso de uso (o parte de caso de uso) se considera interesante para la arquitectura. (En la práctica, esta decisión podría ser un poco precipitada, pero la utilizamos para los propósitos de esta explicación.)

La vista de la arquitectura del modelo de casos de uso debería por tanto, mostrar la descripción completa del caso de uso Sacar Dinero.

4.5.2. La vista de la arquitectura del modelo de diseño

La vista de la arquitectura del modelo de diseño presenta los clasificadores más importantes para la arquitectura pertenecientes al modelo de diseño: los subsistemas e interfaces más importantes, así como algunas pocas clases muy importantes, fundamentalmente las clases activas. También presenta cómo se realizan los casos de uso en términos de esos clasificadores, por medio de realizaciones de casos de uso. Las clases activas también las trataremos en la Sección 4.5.3 cuando hablemos del modelo de despliegue (en el cual las clases activas se asignan a nodos).

Ejemplo La vista de la arquitectura del modelo de diseño del sistema de CA

En la Sección 3.4.3, identificamos tres clases activas: Gestor de Clientes, Gestor de Transacciones, y Gestor de Cuentas (Figura 4.7). Estas clases activas se incluyen en la vista de la arquitectura del modelo de diseño.

Además, en la Sección 3.4.4, se describieron los tres subsistemas: Interfaz del CA, Gestión de Transacciones, y Gestión de Cuentas; *véase* la Figura 4.8. Estos subsistemas son necesarios para la realización del caso de uso Sacar Dinero, por lo cual son subsistemas significativos para la arquitectura. El modelo de diseño incluye muchos otros subsistemas, pero no los consideramos.

El subsistema Interfaz del CA se encarga de todas las entradas y salidas del cliente del banco, tales como la impresión de los recibos y la recepción de los comandos del cliente. El subsistema de Gestión de Cuentas mantiene toda la información persistente sobre las cuentas, y se utiliza en todas las transacciones relativas a ellas. El subsistema de Gestión de Transacciones contiene las clases para el comportamiento específico de los casos de uso, como el comportamiento específico del caso de uso Sacar Dinero. En el ejemplo de la Sección 3.4.4, dijimos que las clases específicas de los casos de uso acaban a menudo en distintos subsistemas de servicio, como los subsistemas de servicio para cada una de las clases Retirada de Efectivo, Transferencia e Ingreso

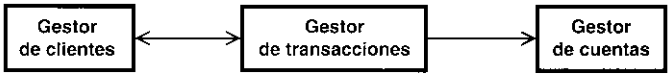


Figura 4.7. La estructura estática de la vista de la arquitectura del modelo de diseño para el sistema de CA. Este diagrama de clases muestra las clases activas.

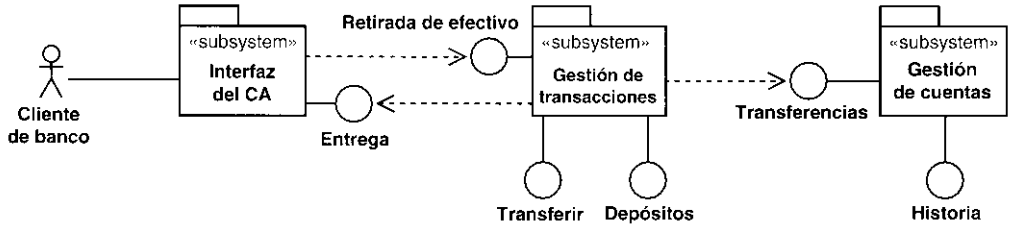


Figura 4.8. La estructura estática de la vista de la arquitectura del modelo de diseño para el sistema de CA. Este diagrama de clases muestra los subsistemas y las interfaces entre ellos.

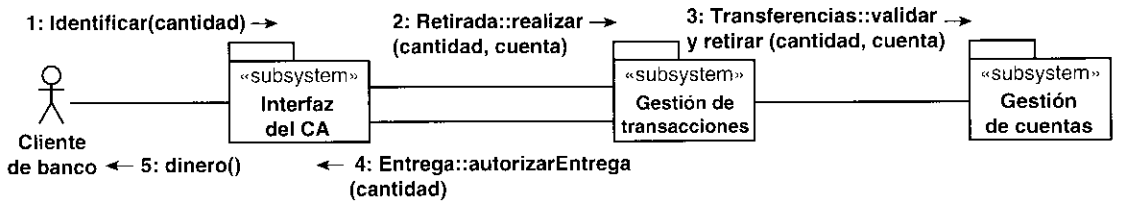


Figura 4.9. Los subsistemas que colaboran en la ejecución del caso de uso Sacar Dinero.

dentro del subsistema de Gestión de Transacciones (no se muestran en la Figura 4.8). En realidad, cada uno de esos subsistemas de servicio contiene normalmente varias clases, pero nuestro ejemplo es muy sencillo.

Los subsistemas de la Figura 4.8 proporcionan comportamiento unos a otros mediante interfaces, como la interfaz Transferencias que proporciona la Gestión de Transacciones. Las interfaces Transferencias, Retirada, y Entrega se describen en la Sección 3.4.4. También tenemos interfaces Transferir, Depósitos, e Historia, pero no se ven implicadas en el caso de uso que tratamos en este ejemplo, por lo que no las hemos explicado.

No basta con la estructura estática. También tenemos que mostrar cómo se llevan a cabo, por parte de los subsistemas del modelo de diseño, los casos de uso significativos para la arquitectura. Por tanto, describiremos una vez más el caso de uso Sacar Dinero, esta vez en términos de subsistemas y actores que interactúan utilizando un diagrama de colaboración (Apéndice A), como se muestra en la Figura 4.9. Los objetos de las clases que poseen los subsistemas interactúan unos con otros para ejecutar una instancia de un caso de uso. Los objetos se envían mensajes; el diagrama muestra estos intercambios. Los mensajes llevan nombres que especifican operaciones contenidas en las interfaces de los subsistemas. Esto se indica mediante la notación :: (por ejemplo, Retirada::realizar(cantidad, cuenta), donde Retirada es una interfaz proporcionada por una clase dentro del subsistema de Gestión de Transacciones).

La siguiente lista explica brevemente el flujo de la ejecución del caso de uso. El texto es casi el mismo que el de la Sección 3.4.1 (la descripción de la ejecución del caso de uso), pero aquí lo presentamos en términos de subsistemas en lugar de clases.

Precondición: El cliente del banco tiene una cuenta bancaria válida para el CA.

1. El actor Cliente de Banco selecciona sacar dinero y se identifica en la Interfaz del CA, quizás a través de una tarjeta de banda magnética, mediante un número y una contraseña. El Cliente de Banco también indica la cantidad a retirar y de qué cuenta hacerlo. Suponemos que el subsistema Interfaz de CA es capaz de verificar la identidad.

2. La Interfaz de CA solicita al subsistema de Gestión de Transacciones que retire el dinero. Este subsistema es responsable de llevar a cabo la secuencia entera de retirada a modo de transacción atómica, de manera que el dinero se deduce de la cuenta y también se entrega al Cliente de Banco.

3. Gestión de Transacciones solicita al subsistema Gestión de Cuentas que retire el dinero. El subsistema Gestión de Cuentas decide si puede retirarse el dinero y, si es así, deduce la suma de la cuenta y devuelve una respuesta que indica que es posible ejecutar la retirada.

4. Gestión de Transacciones autoriza al Interfaz de CA a entregar el dinero.

5. Interfaz de CA entrega el dinero al Cliente de Banco.

4.5.3. La vista de la arquitectura del modelo de despliegue

El modelo de despliegue define la arquitectura física del sistema por medio de nodos interconectados. Estos nodos son elementos hardware sobre los cuales pueden ejecutarse los elementos software. Con frecuencia conocemos cómo será la arquitectura física del sistema antes de comenzar su desarrollo. Por tanto, podemos modelar los nodos y las conexiones del modelo de despliegue tan pronto como comience el flujo de trabajo de los requisitos.

Durante el diseño, decidiremos qué clases son activas, es decir, son hilos o procesos. Determinaremos lo que debería hacer cada clase activa, cómo debería ser el ciclo de vida de cada una de ellas, y cómo deberían comunicarse, sincronizarse, y compartir información. Los objetos activos se asignan a los nodos del modelo de despliegue. Al hacer esta asignación, debemos considerar la capacidad de los nodos, como su capacidad de proceso y tamaño de memoria, y las características de las conexiones, como el ancho de banda y la disponibilidad.

Los nodos y conexiones del modelo de despliegue y la asignación de los objetos activos a los nodos pueden mostrarse en **diagramas de despliegue** (Apéndice A). Estos diagramas también puede mostrar cómo se asignan los componentes ejecutables a los nodos. El sistema de CA de nuestro ejemplo se distribuye en tres nodos distintos.

Ejemplo **La vista de la arquitectura del modelo de despliegue del sistema de CA**

El Cliente de Banco accede al sistema a través del nodo Cliente CA, que accede al Servidor de Aplicaciones CA para realizar las transacciones (Figura 4.10). El Servidor de Aplicaciones CA utiliza, a su vez, al Servidor de Datos CA para ejecutar transacciones concretas sobre cuentas, por ejemplo. Esto es así no sólo para el caso de uso Sacar Dinero, que hemos clasificado como significativo para la

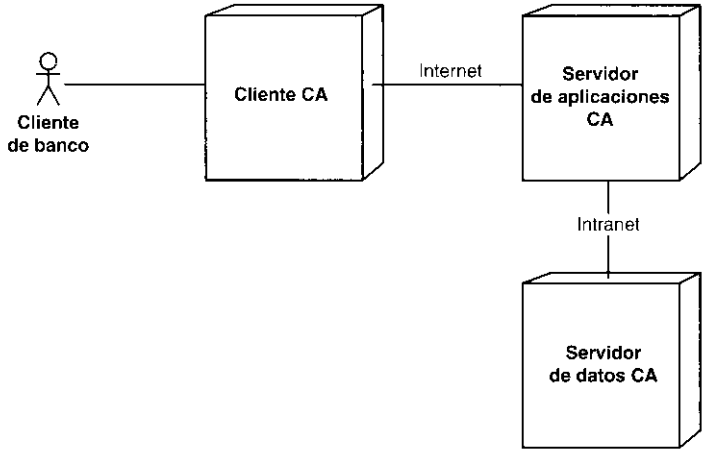


Figura 4.10. El modelo de despliegue define tres nodos: Cliente CA, Servidor de Aplicaciones CA, y Servidor de Datos CA.

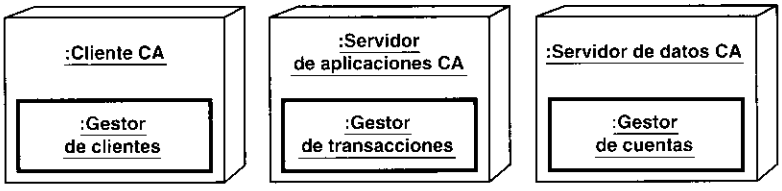


Figura 4.11. La vista arquitectónica del modelo de despliegue. Las clases activas del sistema de CA se distribuyen sobre los nodos. Los objetos activos se muestran mediante rectángulos con bordes gruesos.

arquitectura, sino también para los demás casos de uso, como Ingresar Dinero y Transferir entre Cuentas. En la Sección 3.4.3, describimos qué clases habíamos seleccionado para llevar a cabo el caso de uso Sacar Dinero.

Cuando se han definido los nodos, podemos distribuir la funcionalidad sobre ellos. Por sencillez, lo haremos distribuyendo cada subsistema (*véase* la Figura 4.8) como un todo a un único nodo. El subsistema Interfaz de CA se implanta sobre el nodo Cliente CA, el subsistema Gestión de Transacciones sobre el Servidor de Aplicaciones CA, y el subsistema de Gestión de Cuentas sobre el Servidor de Datos CA. En consecuencia, cada clase activa dentro de estos subsistemas (*véase* la Sección 3.4.4 y la Figura 3.10) se implanta en su correspondiente nodo —mediante un proceso ejecutándose sobre el mismo—. Cada uno de estos procesos gestiona, y mantiene en su espacio de direcciones, objetos del resto de las clases dentro del subsistema (clases ordinarias, no activas). La Figura 4.11 muestra la distribución de los objetos activos.

Éste es un ejemplo simplista de distribución de un sistema. En un sistema real, la distribución es por supuesto más compleja. Una solución alternativa al problema de la distribución habría sido el uso de alguna capa intermedia para la distribución de objetos como un object request broker (ORB).

4.5.4. La vista de la arquitectura del modelo de implementación

El modelo de implementación es una correspondencia directa de los modelos de diseño y de despliegue. Cada subsistema de servicio del diseño normalmente acaba siendo un componente por cada tipo de nodo en el que deba instalarse —pero no siempre es así—. A veces el mismo componente (por ejemplo, el componente de gestión de buffers) puede instanciarse y ejecutarse sobre varios nodos. Hay lenguajes que proporcionan construcciones para el empaquetado de los componentes, como es el caso de los JavaBeans. En otros casos, las clases se organizan en ficheros con el código que representa los diferentes componentes.

En la Sección 3.4.5, indicamos que el subsistema de servicio Gestión de Retiradas posiblemente podría implementarse mediante dos componentes, "Retirada en Servidor", y "Retirada en Cliente". El componente "Retirada en Servidor" podría implementar la clase Retirada de Efectivo, y "Retirada en Cliente" podría implementar una clase Proxy de Retirada. En nuestro ejemplo sencillo, estos componentes sólo implementarían una clase cada uno. En un sistema real, habría varias clases más en cada subsistema de servicio, de forma que un componente podría implementar varias clases.

4.6. Tres conceptos interesantes

4.6.1. ¿Qué es una arquitectura?

Es lo que especifica el arquitecto en la descripción de la arquitectura. La descripción de la arquitectura permite al arquitecto controlar el desarrollo del sistema desde la perspectiva técnica. La arquitectura software se centra tanto en los elementos estructurales significativos del sistema, como subsistemas, clases, componentes y nodos, como en las colaboraciones que tienen lugar entre estos elementos a través de las interfaces.

Los casos de uso dirigen la arquitectura para hacer que el sistema proporcione la funcionalidad y uso deseados, alcanzando a la vez objetivos de rendimiento razonables. Una arquitectura debe ser completa, pero también debe ser suficientemente flexible como para incorporar nuevas funciones, y debe soportar la reutilización del software existente.

4.6.2. ¿Cómo se obtiene?

La arquitectura se desarrolla de forma iterativa durante la fase de elaboración pasando por los requisitos, el análisis, el diseño, la implementación y las pruebas. Utilizamos los casos de uso significativos para la arquitectura y un conjunto de otras entradas para implementar la línea base de la arquitectura, o "esqueleto" del sistema. Ese conjunto de entradas incluye requisitos del software del sistema, middleware, qué sistemas heredados deben utilizarse, requisitos no funcionales, y demás.

4.6.3. ¿Cómo se describe?

La descripción de la arquitectura es una vista de los modelos del sistema, de los modelos de casos de uso, análisis, diseño, implementación y despliegue. La descripción de la arquitectura describe las partes del sistema que es importante que comprendan todos los desarrolladores y otros interesados.

4.7. Referencias

[1] David Garlan and Mary Shaw, *Software Architecture: Perspectives on an Emerging Discipline,* Upper Saddle River, NJ: Prentice-Hall, 1996.

[2] Ivar Jacobson, Martin Griss, and Patrik Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success,* Reading, MA: Addison-Wesley, 1997.

[3] P.B. Kruchten, "The 4+1 view model of architecture", *IEEE Software,* November 1995.

[4] F. Buschmann, R. Meurier, H. Rohnert, P. Sommerlad, M. Stal, *A System of Patterns,* New York: John Wiley and Sons, 1996.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Reading, MA: Addison-Wesley, 1994.

[6] OMG, Inc. The Common Object Request Broker: Architecture and Specification (CORBA), Framingham, MA. 1996.

[7] ISO/IEC International Standard 10165-4 = ITU-T Recommendation X.722.

[8] ITU-T Recommendation M.3010, Principles for a Telecommunication Management Network.

[9] Thomas J. Mowbray and Raphael C. Malveau, *CORBA Design Patterns*, New York: John Wiley and Sons, 1997.

[10] Christopher Alexander, Sara Ishikawa, Murray Silverstein, with Max Jacobsen, Ingrid Fiksdahi-King, Shlomo Angel, *A Pattern Language: Towns, Buildings, Construction*, New York: Oxford University Press, 1977.

Capítulo 5

Un proceso iterativo e incremental

Un proceso de desarrollo de software debe tener una secuencia de **hitos** (Apéndice C) claramente articulados para ser eficaz, que proporcionen a los directores y al resto del equipo del proyecto los criterios que necesitan para autorizar el paso de una fase a la siguiente dentro del ciclo del producto.

Dentro de cada fase, el proceso pasa por una serie de **iteraciones** e incrementos (Apéndice C) que nos conducen a esos criterios.

En la fase de inicio el criterio esencial es la viabilidad, que llevamos a cabo mediante:

- La identificación y la reducción de los **riesgos** (Apéndice C; *véase* también la Sección 12.5) críticos para la viabilidad del sistema.
- La creación de una arquitectura candidata a partir del desarrollo de un subconjunto clave de los requisitos, pasando por el modelado de los casos de uso.
- La realización de una estimación inicial de coste, esfuerzo, calendario y calidad del producto con límites amplios.
- El inicio del análisis del negocio *(véase* más sobre el análisis del negocio en los Capítulos 12 a 16) por el cual el proyecto parece que merece la pena económicamente, una vez más con límites amplios.

En la fase de elaboración, el criterio esencial es la capacidad de construir el sistema dentro de un marco de trabajo económico, que llevamos a cabo mediante:

- La identificación y la reducción de los riesgos que afectan de manera significativa a la construcción del sistema.
- La especificación de la mayoría de los casos de uso que representan la funcionalidad que ha de desarrollarse.
- La extensión de la arquitectura candidata hasta las proporciones de una línea base.
- La preparación del **plan del proyecto** (Apéndice C) con suficiente detalle como para guiar la fase de construcción.
- La realización de una estimación con unos límites suficientemente ajustados como para justificar la inversión.
- La terminación del análisis del negocio —el proyecto merece la pena.

En la fase de construcción, el criterio esencial es un sistema capaz de una operatividad inicial en el entorno del usuario, y lo llevamos a cabo mediante:

- Una serie de iteraciones, que llevan a incrementos y entregas periódicos, de forma que a lo largo de esta fase, la viabilidad del sistema siempre es evidente en la forma de ejecutables.

En la fase de transición, el criterio esencial es un sistema que alcanza una operatividad final, llevado a cabo mediante:

- La modificación del producto para subsanar problemas que no se identificaron en fases anteriores.
- La corrección de **defectos** (Apéndice C; *véase* también la Sección 11.3.6).

Uno de los objetivos del Proceso Unificado es hacer que los arquitectos, desarrolladores e interesados en general comprendan la importancia de las primeras fases. Acerca de esto, no podemos hacer nada mejor que citar el consejo de Barry Boehm de hace unos años [1]:

> *«No puedo recalcar suficiente cuán crítico es para su proyecto y para su carrera el hito de la Arquitectura del Ciclo de Vida (ACV) [que se corresponde con nuestro hito de la fase de elaboración]. Si no ha cumplido con el criterio del hito ACV, no prosiga en un desarrollo a escala completa. Reúna de nuevo a los usuarios y obtenga un nuevo plan del proyecto que cumpla con éxito los criterios ACV.»*

Las fases, y las iteraciones dentro de ellas, se tratarán con más detalle en la Parte III.

5.1. Iterativo e incremental en pocas palabras

Como indicamos en los Capítulos 3 y 4, dos de las tres claves del Proceso Unificado son que el proceso de desarrollo software debería estar dirigido por los casos de uso y centrado en la arquitectura. Estos aspectos tienen un impacto técnico evidente sobre el producto del proceso. El estar dirigido por los casos de uso significa que cada fase en el camino al producto final está relacionada con lo que los usuarios hacen realmente. Lleva a los desarrolladores a garantizar que

el sistema se ajusta a las necesidades reales del usuario. El estar centrado en la arquitectura significa que el trabajo de desarrollo se centra en obtener el patrón de la arquitectura que dirigirá la construcción del sistema en las primeras fases, garantizando un progreso continuo no sólo para la versión en curso del producto, sino para la vida entera del mismo.

Conseguir el equilibrio correcto entre los casos de uso y la arquitectura es algo muy parecido al equilibrado de la forma y la función en el desarrollo de cualquier producto. Se consigue con el tiempo. Lo que nos encontramos primero, como dijimos en la Sección 4.3, es un problema de la "gallina y el huevo". La gallina y el huevo se consiguen a lo largo de iteraciones casi sin fin durante el largo proceso de la evolución. De manera similar, durante el más corto proceso de desarrollo de software, los desarrolladores elaboran conscientemente este equilibrio (entre casos de uso y arquitectura) a lo largo de una serie de iteraciones. Por tanto, la técnica de desarrollo iterativo e incremental constituye el tercer aspecto clave del Proceso Unificado.

5.1.1. Desarrollo en pequeños pasos

La tercera clave proporciona la estrategia para desarrollar un producto software en pasos pequeños manejables:

- Planificar un poco.
- Especificar, diseñar, e implementar un poco.
- Integrar, probar, y ejecutar un poco cada iteración.

Si estamos contentos con un paso, continuamos con el siguiente. Entre cada paso, obtenemos retroalimentación que nos permite ajustar nuestros objetivos para el siguiente paso. Después se da el siguiente paso, y después otro. Cuando se han dado todos los pasos que habíamos planificado, tenemos un producto desarrollado que podemos distribuir a nuestros clientes y usuarios.

Las iteraciones en las primeras fases tratan en su mayor parte con la determinación del ámbito del proyecto, la eliminación de los riesgos críticos, y la creación de la línea base de la arquitectura. Después, a medida que avanzamos a lo largo del proyecto y vamos reduciendo gradualmente los riesgos restantes e implementando los componentes, la forma de las iteraciones cambia, dando incrementos como resultados.

Un proyecto de desarrollo software transforma una "delta" (un cambio) de los requisitos de usuario en una delta del producto software (*véase* la Sección 2.2). Con el método de desarrollo iterativo e incremental esta adaptación de los cambios se realiza poco a poco. Dicho de otra forma, dividimos el proyecto en un número de miniproyectos, siendo cada uno de ellos una iteración. Cada iteración tiene todo lo que tiene un proyecto de desarrollo de software: planificación, desarrollo en una serie de flujos de trabajo (requisitos, análisis y diseño, implementación y prueba), y una preparación para la entrega.

Pero una iteración no es una entidad completamente independiente. Es una etapa dentro de un proyecto, y se ve fuertemente condicionada por ello. Decimos que es un miniproyecto porque no es por sí misma algo que nuestros usuarios nos hayan pedido que hagamos. Además, cada uno de estos miniproyectos se parece al antiguo ciclo de vida en cascada debido a que se desarrolla a través de actividades en cascada. Podríamos llamar una "minicascada" a cada iteración.

El ciclo de vida iterativo produce resultados tangibles en forma de versiones internas (aunque preliminares), y cada una de ellas aporta un incremento y demuestra la reducción de los riesgos con los que se relaciona. Estas versiones pueden presentarse a los clientes y usuarios, en cuyo caso proporcionan una retroalimentación valiosa para la validación del trabajo.

Los jefes de proyecto tratarán de ordenar las iteraciones para conseguir una vía directa en la cual las primeras iteraciones proporcionen la base de conocimiento para las siguientes. Las primeras iteraciones del proyecto consiguen incrementar la comprensión de los requisitos, el problema, los riesgos y el **dominio de la solución** (Apéndice C), mientras que las restantes añaden incrementos que al final conformarán la **versión externa** (Apéndice C), es decir, el producto para el cliente. El éxito fundamental —para los jefes de proyecto— es conseguir una serie de iteraciones que siempre avanzan; es decir, que nunca hay que volver dos o tres iteraciones atrás para corregir el modelo debido a algo que hemos aprendido en la última iteración. No queremos escalar una montaña de nieve que se funde, dando dos pasos adelante y resbalando uno hacia atrás.

En resumen, un ciclo de vida se compone de una secuencia de iteraciones. Algunas de ellas, especialmente las primeras, nos ayudan a comprender los riesgos, determinar la viabilidad, construir el núcleo interno del software, y realizar el análisis del negocio. Otras, en especial las últimas, incorporan incrementos hasta conseguir un producto preparado para su entrega.

Las iteraciones ayudan a la dirección a planificar, a organizar, a hacer el seguimiento y a controlar el proyecto. Las iteraciones se organizan dentro de las cuatro fases, cada una de ellas con necesidades concretas de personal, financiación y planificación, y con sus propios criterios de comienzo y fin. Al comienzo de cada fase, la dirección puede decidir cómo llevarla a cabo, los resultados que deben entregarse, y los riesgos que deben reducirse.

5.1.2. Lo que no es una iteración

Algunos directores creen que "iterativo e incremental" es un eufemismo de "hacking" [1], y temen que esas palabras solamente ocultan la realidad de que los desarrolladores no saben lo que están haciendo. En la fase de inicio, e incluso al principio de la fase de elaboración, puede que haya algo de verdad en ello. Por ejemplo, si los desarrolladores no han resuelto riesgos críticos o significativos, entonces la proposición es cierta. Si aún no han probado el concepto subyacente, o no han establecido la línea base de la arquitectura, también es cierta. Lo es también si aún no han decidido cómo implementar los requisitos más importantes. En realidad, podría ser que no supieran lo que están haciendo.

¿Ayuda en algo el pretender que saben lo que están haciendo? ¿Es de algún provecho basar un plan sobre una información insuficiente? ¿Sirve para algo comenzar ese plan no fiable? Por supuesto que no.

Para concretar, vamos a recalcar lo que no es un ciclo de vida iterativo:

- No es un desarrollo aleatorio.
- No es un parque de juegos para los desarrolladores.

---

[1] *N. del T.* El verbo *hack* tiene difícil traducción al castellano, y se utiliza mucho en la jerga de Internet, por lo que hemos decidido no traducirlo.

- No es algo que afecte sólo a los desarrolladores.
- No es rediseñar una y otra vez lo mismo hasta que al final los desarrolladores prueben algo que funciona.
- No es algo impredecible.
- No es una excusa para fracasar en la planificación y en la gestión.

De hecho, la iteración controlada está muy lejos de ser aleatoria. Se planifica. Es una herramienta que pueden utilizar los directores para controlar el proyecto. Reduce, al principio del ciclo de vida, los riesgos que pueden amenazar el progreso del desarrollo. Las **versiones internas** (Apéndice C) tras las iteraciones posibilitan la retroalimentación de los usuarios, y esto lleva a su vez a una corrección temprana de la trayectoria del proyecto.

5.2. ¿Por qué un desarrollo iterativo e incremental?

En dos palabras: para obtener un software mejor. Dicho con unas cuantas palabras más, para cumplir los hitos principales y secundarios con los cuales controlamos el desarrollo. Y dicho con algunas palabras más:

- Para tomar las riendas de los riesgos críticos y significativos desde el principio.
- Para poner en marcha una arquitectura que guíe el desarrollo del software.
- Para proporcionar un marco de trabajo que gestione de mejor forma los inevitables cambios en los requisitos y en otros aspectos.
- Para construir el sistema a lo largo del tiempo en lugar de hacerlo de una sola vez cerca del final, cuando el cambiar algo se ha vuelto costoso.
- Para proporcionar un proceso de desarrollo a través del cual el personal puede trabajar de manera más eficaz.

5.2.1. Atenuación de riesgos

El desarrollo de software se enfrenta con riesgos, al igual que cualquier otra actividad de ingeniería. Según la visión del profeta de la dirección Peter F. Drucker [2]: "El riesgo es inherente en el empleo de los recursos disponibles para las expectativas futuras." En el desarrollo de software, afrontamos esta realidad identificando los riesgos tan pronto como sea posible dentro del desarrollo y tratándolos rápidamente. Un riesgo es una exposición que nos puede acarrear pérdidas o daños. El riesgo es un factor, cosa, elemento o camino que constituye un peligro, cuyo grado es incierto. En el desarrollo de software, podemos definir un riesgo como un asunto que tiene cierto grado de probabilidad de poner en peligro el éxito de un proyecto. Por ejemplo,

- El **object request broker** (Apéndice C) que consideramos inicialmente, puede que no sea capaz de dar servicio a 1.000 búsquedas de objetos remotos, que representan cuentas de cliente, por segundo.
- Un sistema de tiempo real puede tener que tomar un cierto número de entradas de datos que no se especificaron en la fase de inicio. Puede que tenga que procesar los datos me-

diante cálculos intensivos que aún no se han definido con detalle, o puede que tenga que enviar una señal de control en un tiempo corto que aún no se ha especificado.

- Un sistema de conmutación telefónica puede tener que responder a varias entradas en un número de milisegundos especificado por la empresa de operación de telecomunicaciones cliente.

Lo que necesita la disciplina del software, como escribió Barry Boehm hace ya muchos años, es un modelo de proceso que "cree una aproximación al proceso de desarrollo de software *dirigida por los riesgos* en lugar de un proceso fundamentalmente *dirigido por los documentos* o *dirigido por el código*" [3]. El Proceso Unificado cumple con estos criterios porque trata los riesgos importantes en las dos primeras fases, inicio y elaboración, y cualquier riesgo restante al principio de la fase de construcción, por orden de importancia. Identifica, gestiona, y reduce los riesgos en las primeras fases mediante las iteraciones. En consecuencia, los riesgos no identificados o ignorados no emergen más tarde, poniendo en peligro el proyecto entero.

La técnica iterativa para la reducción de los riesgos recuerda muy poco al **método en cascada** (Apéndice C). El modelo en cascada muestra el desarrollo pasando en un solo sentido por una serie de pasos: requisitos, análisis, diseño, implementación y prueba. En este método, el proyecto debe implicar a todos sus desarrolladores cuando llega a la implementación, la **integración** (Apéndice C) y la prueba. Durante la integración y la prueba, los problemas pueden empezar a estallar a su alrededor. El jefe de proyecto se verá entonces forzado a reasignar personas —habitualmente a los desarrolladores con más experiencia— para resolver esos problemas antes de que el trabajo pueda continuar. Sin embargo, como todos los desarrolladores ya están ocupados, los jefes de proyecto tendrán difícil el "liberar" a los pocos que estén más preparados para resolver los problemas que se acaban de descubrir. Para tratar estas dificultades, la reasignación de los desarrolladores con más experiencia para "resolver los problemas" suele dejar a los desarrolladores con menos experiencia sentados esperando. Los plazos pasan, y los costes se rebasan. En el peor caso, nuestra competencia se hace con el mercado antes que nosotros.

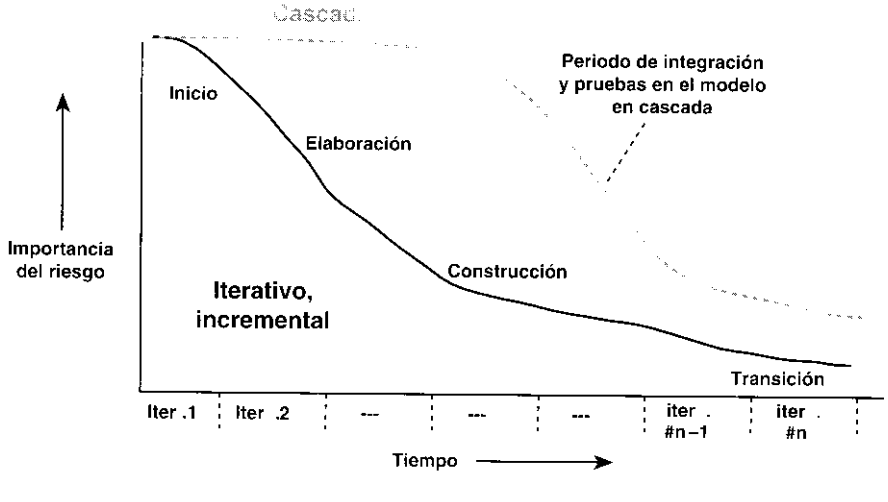


Figura 5.1. Los riesgos importantes se identifican y se reducen al principio del desarrollo iterativo, a diferencia del desarrollo en cascada. En este último, los riesgos más importantes permanecen hasta que la integración y las pruebas del sistema se ocupan de ellos (como indica la línea discontinua). Las iteraciones indicadas en la parte inferior de la figura son, por supuesto, solamente relevantes para el enfoque iterativo e incremental.

Si realizamos un gráfico del riesgo comparándolo con el tiempo de desarrollo, como en la Figura 5.1, el desarrollo iterativo empieza a reducir riesgos importantes en las primeras iteraciones. En el momento en que el trabajo alcanza la fase de construcción, quedan pocos riesgos importantes, y el trabajo prosigue sin problemas. En contraste, si utilizamos el modelo en cascada, los riesgos importantes no se tratan hasta la integración del código al estilo de un "big bang".

Cerca de las dos terceras partes de los proyectos software grandes fracasan en la realización de un análisis de riesgos adecuado, según Capers Jones [4], por tanto, ¡hay lugar para grandes mejoras! Atacar los riesgos al comienzo del proceso de desarrollo es el primer paso.

5.2.2. Obtención de una arquitectura robusta

La consecución de una arquitectura robusta es en sí misma el resultado de las iteraciones en las primeras fases. En la fase de inicio, por ejemplo, encontramos una arquitectura esencial que satisface los requisitos clave, evita los riesgos críticos, y resuelve los problemas de desarrollo principales. En la fase de elaboración, establecemos la línea base de la arquitectura que guiará el resto del desarrollo.

Por un lado, la inversión en esas fases es todavía pequeña, y podemos permitirnos las iteraciones que garantizan que la arquitectura es robusta. Por ejemplo, después de la primera iteración en la fase de elaboración, estamos en disposición de hacer una evaluación inicial de la arquitectura. En este momento, aún podemos permitirnos cambiarla, si eso es lo que hemos decidido, para ajustarla a las necesidades de los casos de uso significativos y a los requisitos no funcionales.

Si seguimos el método en cascada, en el momento en que descubrimos la necesidad de un cambio en la arquitectura, ya hemos invertido tanto en el desarrollo que hacer un cambio en la arquitectura supone un revés económico importante. Además, podríamos estar cerca de la fecha de entrega comprometida. Atrapados entre los costes y el calendario, no tendremos la motivación necesaria para hacer un cambio en la arquitectura. Podemos evitar este dilema centrándonos en la arquitectura en la fase de elaboración. Estabilizamos la arquitectura hasta un nivel de línea base al principio del ciclo de vida, cuando los costes aún son bajos y el calendario todavía es generoso con nosotros.

5.2.3. Gestión de requisitos cambiantes

Los usuarios pueden comprender más fácilmente un sistema en funcionamiento, aunque aún no funcione perfectamente, que un sistema que sólo existe en forma de cientos de páginas de documentos. Además, tienen dificultades en reconocer el avance del proyecto si todo lo que existe son documentos. Por tanto, desde la perspectiva de los usuarios y de otros interesados, es más productivo hacer evolucionar el producto a lo largo de una serie de versiones ejecutables, o "construcciones", que presentar montañas de documentación difíciles de leer. Una construcción es una versión operativa de parte de un sistema que demuestra un subconjunto de posibilidades del sistema. Cada iteración debe progresar mediante una serie de construcciones hasta alcanzar el resultado esperado, es decir, el incremento.

El tener un sistema con un funcionamiento parcial en una fase inicial permite que los usuarios y otros interesados hagan sugerencias sobre él o señalen requisitos que se nos pueden haber es-

capado. El plan —presupuesto y calendario— aún no es inamovible, de forma que los desarrolladores pueden incorporar revisiones con más facilidad. En el modelo unidireccional en cascada, los usuarios no ven un sistema funcionando hasta la integración y la prueba. En ese momento, los cambios, incluso aquellos que son valiosos o parecen ser pequeños, introducen una desviación en el presupuesto o en el calendario de manera casi inevitable. Por tanto, el ciclo de vida iterativo hace más sencillo a los clientes el observar la necesidad de requisitos adicionales o modificados pronto dentro del ciclo de desarrollo, y hace más sencillo también a los desarrolladores el trabajar en ellos. No en vano, están construyendo el sistema mediante una serie de iteraciones, por lo que responder a una retroalimentación o incluir una revisión sólo significa un cambio incremental.

5.2.4. Permitir cambios tácticos

Con el método iterativo e incremental, los desarrolladores pueden resolver los problemas y los aspectos no cubiertos por las primeras construcciones e incluir cambios para corregirlos casi a la vez. Mediante esta técnica, los problemas se van descubriendo con un ritmo de goteo constante que los desarrolladores pueden tratar fácilmente. La tromba de informes de fallo que aparece en la integración en "big bang" del ciclo en cascada a menudo rompe la marcha del proyecto. Si la ruptura es grave, el proyecto puede llegar a pararse, con los desarrolladores aplastados por la presión, los jefes de proyecto dando vueltas en círculo, y con el pánico de otros directivos. En contraste, una serie de construcciones funcionales ofrece a todos la sensación de que las cosas van bien.

Los ingenieros de prueba, los escritores de manuales, los encargados de las herramientas, el personal de gestión de configuración, y los encargados del aseguramiento de la calidad pueden adaptar sus propios planes al calendario en evolución del proyecto. Tienen conocimiento de la existencia de retrasos serios en las primeras fases del proyecto, cuando los desarrolladores encuentran por primera vez los problemas que los originan. Tienen tiempo de adaptar sus propios calendarios. Cuando los problemas descansan ocultos hasta la prueba, es demasiado tarde para que aquéllos puedan rehacer sus calendarios de forma eficaz.

Cuando la iteración ha sido validada por el aseguramiento de la calidad, los jefes de proyecto, los arquitectos, y otros interesados pueden evaluarla según los criterios preestablecidos. Pueden decidir si la iteración ha obtenido el incremento correcto y si los riesgos se han tratado adecuadamente. Esta evaluación permite a los directores determinar si la iteración tuvo éxito. Si fue así, pueden autorizar la siguiente. Si la iteración sólo tuvo un éxito parcial, pueden ampliarla para cubrir aspectos no resueltos o para realizar las correcciones necesarias para la siguiente iteración. En un caso extremo, cuando la evaluación sea totalmente negativa, pueden cancelar el proyecto entero.

5.2.5. Conseguir una integración continua

Al final de cada iteración, el equipo del proyecto demuestra que ha reducido algunos riesgos. El equipo entrega nuevas funcionalidades en cada iteración, que se hacen evidentes para los usuarios, los cuales pueden observar el progreso del proyecto.

Las construcciones frecuentes fuerzan a los desarrolladores a concretar a intervalos regulares —concreción en forma de un fragmento de código ejecutable—. La experiencia de las construcciones les hace difícil a ellos o a cualquiera sostener la actitud del "90 por ciento completo". Esta actitud aparece cuando un número de fragmentos de código o de otros artefactos (Apéndice C) hace parecer que el producto está casi terminado. Sin embargo, en ausencia de

construcciones funcionales, es posible que la parte más difícil aún esté por venir. Puede que aún no se hayan descubierto los problemas mediante intentos de integrar y probar el sistema. En contraste, las sucesivas iteraciones, debido a que funcionan, producen una serie de resultados que nos indican exactamente el estado del proyecto.

Incluso si los desarrolladores no consiguen obtener los resultados previstos en una de las primeras iteraciones, todavía tienen tiempo de intentarlo de nuevo y mejorar los modelos en las subsiguientes versiones internas. Ya que primero trabajan sobre los aspectos mas críticos, cuentan con varias oportunidades para mejorar sus resultados.

La línea gruesa de la Figura 5.2 (presentado originalmente en [5]) muestra cómo funciona el desarrollo iterativo. En puntos al comienzo del calendario, en este caso con sólo del 2 al 4 por ciento del proyecto codificado, se consigue un incremento (o construcción). Este intento muestra algunos problemas, que se denotan en la figura por los pequeños descensos en la línea del avance, pero se resuelven rápidamente y el avance continúa. Después de ello, el proyecto realiza construcciones frecuentes. Cada una de ellas puede llevar a una parada temporal del avance. Debido a que los incrementos son relativamente pequeños, comparados con la integración final del producto entero (en la línea inferior), la recuperación se lleva a cabo rápidamente.

Como sugiere el diagrama, en el método en cascada, una sola integración cercana a la fecha de entrega descubre una legión de problemas. El gran volumen de problemas y la precipitación inevitable del momento tiene como resultado el que muchas de las correcciones no estén bien pensadas. Apresurarse a corregir los problemas suele retrasar la entrega más allá de la fecha planificada. En consecuencia, el desarrollo iterativo termina en mucho menos tiempo que el desarrollo en cascada. Es más, el producto de un "proyecto en cascada" puede ser frágil, es decir, difícil de mantener.

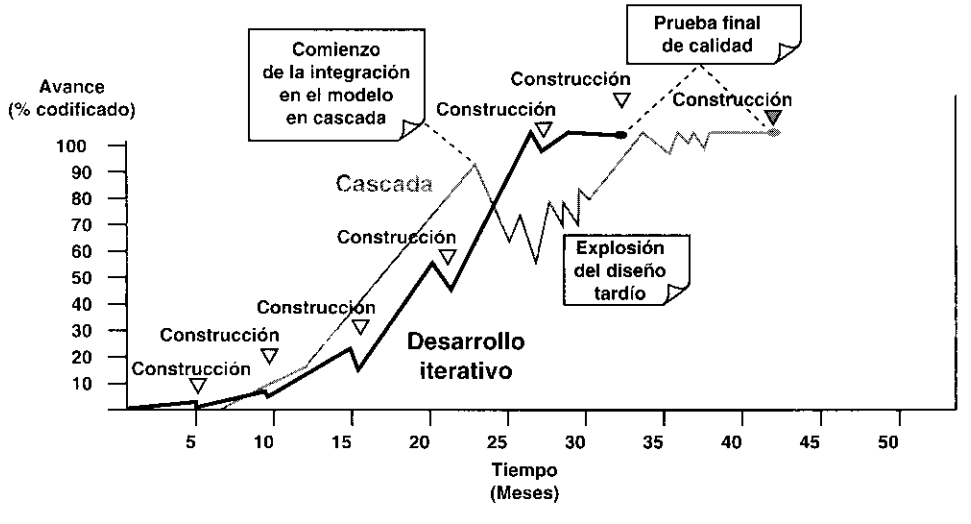


Figura 5.2. En el método en cascada (la línea fina), los desarrolladores no comienzan la implementación hasta haber terminado los requisitos, y el diseño. Informan de un buen progreso en la implementación porque no tienen construcciones intermedias que les indiquen otra cosa. Los problemas están aún ocultos bajo tierra hasta que la integración y las pruebas los descubran todos a la vez (Explosión del Diseño Tardío). En el desarrollo iterativo, la implementación comienza más al principio y las construcciones frecuentes no sólo descubren pronto los problemas, sino que también los solucionan en pequeños lotes más fáciles de manejar.

5.2.6. Conseguir un aprendizaje temprano

Después de un par de iteraciones, todas las personas del equipo tienen una buena comprensión de lo que significan los diferentes flujos de trabajo. Saben lo que viene después de los requisitos y lo que viene después del análisis. Se reduce en gran medida el riesgo de "parálisis del análisis" (demasiado tiempo gastado en el análisis).

Además, es fácil formar a gente nueva debido a que pueden formarse con el propio trabajo. El proyecto no está forzado a diseñar proyectos piloto especiales sólo para que la gente entienda qué es el proceso. Pueden comenzar directamente con el trabajo real. Dado que han recibido la formación adecuada y que trabajan con alguien que ya lo ha hecho antes, rápidamente se ajustan a la velocidad adecuada. Si alguien nuevo no consigue entender algo o comete un error, su fallo no es crítico para el progreso a largo plazo del proyecto, debido a que se revela en el siguiente intento de hacer una construcción.

El método iterativo también ayuda al proyecto a tratar los riesgos de naturaleza no técnica, como los riesgos de la organización. Por ejemplo, puede que los desarrolladores no aprendan suficientemente rápido:

- A construir aplicaciones utilizando un object request broker.
- A utilizar las herramientas de prueba de **gestión de la configuración** (Apéndice C).
- A trabajar de acuerdo al proceso de desarrollo de software.

A medida que itera un proyecto, un grupo pequeño se pone al día con esas tecnologías, herramientas y procesos nuevos. En las siguientes iteraciones, a medida que el grupo las utiliza más, adquiere más habilidad. El equipo va creciendo gradualmente a medida que el proyecto pasa por las iteraciones, quizá empezando con unas 5 a 10 personas, pasando a 25, y finalmente a unas 100 personas. A medida que el grupo va creciendo paso a paso, el equipo inicial está disponible para tutelar a los nuevos miembros que suben a bordo. El método iterativo permite al equipo inicial ajustar el proceso y las herramientas antes de que la mayoría de los desarrolladores se unan al equipo.

Mediante el trabajo dividido en fases e iteraciones, los desarrolladores son más capaces de cumplir con las necesidades reales de los clientes y de reducir los riesgos. Mediante la construcción por incrementos, todos los implicados pueden observar su nivel de progreso. Mediante la reducción de los problemas de última hora, aceleran el tiempo de salida al mercado. Es más, este método iterativo es beneficioso, no sólo para los desarrolladores, y fundamentalmente para los clientes, sino también para los directores. Los directores pueden notar el verdadero avance fijándose en las iteraciones terminadas.

5.3. La aproximación iterativa está dirigida por los riesgos

Un riesgo es una variable del proyecto que pone en peligro o impide el éxito del proyecto. Es la "probabilidad de que un proyecto experimente sucesos no deseables, como retrasos en las fechas, excesos de coste, o la cancelación directa" (*véase* el glosario en [4]).

Identificamos, priorizamos, y llevamos a cabo las iteraciones sobre la base de los riesgos y su orden de importancia. Esto se hace así cuando evaluamos tecnologías nuevas, y cuando traba-

jamos para cumplir con las necesidades de los usuarios —los requisitos—, sean éstos funcionales o no funcionales. También es así cuando, en las primeras fases, vamos estableciendo una arquitectura que será robusta, es decir, que podrá incorporar los cambios con un riesgo bajo de tener que rediseñar algo. Sí, organizamos las iteraciones para conseguir una reducción del riesgo.

Otros riesgos importantes son temas de rendimiento (velocidad, capacidad, precisión), **fiabilidad**, disponibilidad, integridad de las interfaces de usuario, adaptabilidad, y **portabilidad** (Apéndice C). Muchos de estos riesgos no son visibles hasta que se implementa y prueba el software que implementa las funciones subyacentes. Éste es el motivo por el cual se deben llevar a cabo iteraciones que examinen los riesgos, incluso en las fases de inicio y elaboración, y hasta la codificación y la prueba. El objetivo es acabar con los riesgos en una iteración temprana.

Una observación interesante sobre esto es que, en principio, todos los riesgos técnicos pueden hacerse corresponder con un caso de uso o un escenario de un caso de uso. Aquí, *correspondencia* quiere decir que el riesgo se atenúa si se desarrolla el caso de uso con sus requisitos funcionales y no funcionales. Esto es cierto no sólo para los riesgos relativos a los requisitos y a la arquitectura, sino también para la verificación del hardware y del software subyacentes. Mediante una cuidadosa selección de los casos de uso, podemos probar todas las funciones de la arquitectura subyacente.

La reducción del riesgo es un eje central de las iteraciones que hacemos en las fases de inicio y de elaboración. Más adelante, en la fase de construcción, la mayoría de los riesgos se han reducido hasta un nivel de rutina, queriendo decir con esto que se someten a prácticas de desarrollo ordinarias. Intentamos ordenar las iteraciones de manera que cada una de ellas se construya sobre la anterior. Intentamos reducir así el riesgo concreto de que si no ordenamos bien las iteraciones, podríamos tener que rehacer varias iteraciones previas.

5.3.1. Las iteraciones alivian los riesgos técnicos

Los riesgos han sido clasificados en muchas categorías [3] y [4]. Sin embargo, es suficiente para nuestros propósitos el dar sugerencias, sin ser exhaustivos. Hemos identificado cuatro categorías amplias:

1. Riesgos relacionados con nuevas tecnologías:
   - Puede que haya que distribuir los procesos en muchos nodos, lo cual posiblemente origine problemas de sincronización.
   - Algunos casos de uso pueden depender de técnicas informáticas que aún no están bien conseguidas, como el reconocimiento del lenguaje natural o el uso de la tecnología Web.
2. Riesgos relativos a la arquitectura. Estos riesgos son tan importantes que hemos diseñado el Proceso Unificado para que los trate de manera estándar; es decir, la fase de elaboración y las iteraciones para la arquitectura dentro de ella proporcionan un lugar explícito en el proceso para tratarlos. Mediante el establecimiento temprano de una arquitectura que acomode los riesgos, eliminamos el riesgo de no ser capaces de incorporar fácilmente los cambios. Eliminamos el riesgo de tener que rehacer después una buena parte del trabajo. Esta arquitectura resistente a los riesgos es robusta. La adaptación elegante al cambio es característica de la **robustez** (Apéndice C) de la arquitectura.

Otra ventaja de obtener pronto una arquitectura robusta incluye el mostrar dónde encajan los componentes reutilizables, lo que nos permite pensar en comprar en lugar de desarrollar al principio del proyecto. También reduce el riesgo de descubrir demasiado tarde que el sistema es demasiado caro de construir. Por ejemplo,

- Los casos de uso que seleccionamos inicialmente no sirven para ayudarnos a obtener la estructura de subsistemas que necesitamos para hacer evolucionar el sistema con los casos de uso que irán llegando. En las primeras iteraciones, digamos durante la fase de elaboración, puede que no nos demos cuenta de que varios actores utilizan el mismo caso de uso a través de diferentes interfaces. Un ejemplo de esta situación es la de tener varios interfaces para sacar dinero: uno emplea un interfaz gráfico de usuario y un computador personal; y otro utiliza un protocolo de comunicaciones sobre una red. Si nuestro diseño trata de cumplir sólo uno de estos casos de uso, podemos acabar con una arquitectura que no tenga un interfaz interno que nos permita añadir nuevos tipos de interacción. El riesgo reside en que será difícil hacer que un sistema de ese tipo evolucione.
- Ciertos marcos de trabajo (Apéndice C) planificados para su reutilización, puede que en realidad no se hayan usado fuera del proyecto original para el cual se construyeron. El riesgo reside en que un marco de trabajo como ese no funcionará bien con otros marcos de trabajo, o en que no será fácil de reutilizar.
- La nueva versión del sistema operativo que pretendemos utilizar puede no haber alcanzado el nivel de calidad necesario para que confiemos en él. El riesgo reside en que podemos vernos obligados a retrasar la entrega de nuestro propio software mientras esperamos que el fabricante actualice el sistema operativo.

3. Riesgos relativos a construir el sistema adecuado, es decir, que cumpla con su misión y sus usuarios. Este riesgo subraya la importancia de identificar los requisitos funcionales y no funcionales, lo cual significa esencialmente identificar los casos de uso correctos con las interfaces de usuario correctas. Es importante encontrar las funciones más importantes al principio para estar seguros de que se implementan al principio. Para ello, disponemos los casos de uso por orden de importancia relativa al cumplimiento de las necesidades de los clientes y de los requisitos de rendimiento. Consideramos tanto el comportamiento como las capacidades, tales como el rendimiento. Cuando seleccionamos casos de uso, basamos el orden en que los tratamos en su riesgo potencial, como la posibilidad de encontrar problemas con el rendimiento del caso de uso. Especialmente en las fases de inicio y de elaboración, existe una estrecha relación entre ciertos requisitos (y los casos de uso que los expresan) y los riesgos que descansan en su interior. Los casos de uso que el equipo seleccione tendrán su impacto en la arquitectura que están desarrollando. Por ejemplo,
   - El caso de uso Sígueme permite a un abonado a una línea redirigir sus llamadas a otro número. ¿Debería aplicarse esta redirección a todas las llamadas? ¿Qué se debe hacer con las llamadas del despertador? El abonado estará en esos casos probablemente en su número básico, y no querrá que se redireccione la llamada.
4. Algunos riesgos son relativos al rendimiento. Por ejemplo,
   - El tiempo de respuesta de un caso de uso debe ser menor de 1 segundo.
   - El número de instancias concurrentes de un caso de uso sobrepasa las 100.000 en una hora.

La identificación de áreas problemáticas como éstas depende en gran medida de personas con una dilatada experiencia. Debido a que es probable que ninguna persona tenga la experiencia necesaria, tendrán que estudiar el sistema varias personas, hacer listas de posibles problemas, y reunirse en sesiones de identificación de riesgos. No se pretende que estas sesiones resuelvan los problemas, simplemente que los identifiquen y priorizen el orden en que se van a estudiar más en profundidad en iteraciones dentro de las fases de inicio y elaboración.

5.3.2. La dirección es responsable de los riesgos no técnicos

Riesgos no técnicos son aquellos que una dirección atenta puede detectar y desviar. Los siguientes son ejemplos de esta categoría:

- La organización a fecha de hoy no cuenta con gente con experiencia en ciertos aspectos poco usuales del proyecto propuesto.
- La organización pretende implementar partes del sistema propuesto en un lenguaje que le es nuevo.
- El calendario propuesto por el cliente parece ser demasiado corto, a menos que cada paso encaje en su lugar sin problemas.
- La organización sólo puede cumplir sus plazos si terceras empresas subcontratadas, con las que no se ha trabajado antes, pueden entregar a tiempo ciertos subsistemas.
- Puede que el cliente no sea capaz de devolver ciertas aprobaciones dentro de los límites de tiempo necesarios para llegar a la fecha de entrega.

Los riesgos de este tipo caen fuera del propósito de este libro. Baste decir que la organización de desarrollo debería identificarlos, poner los medios administrativos necesarios para seguir los desarrollos en cada una de las áreas de riesgo, y garantizar que los directivos responsables emprendan acciones cuando uno de estos riesgos aparece.

5.3.3. Tratamiento de los riesgos

Una vez que se han identificado y priorizado los riesgos, a continuación el equipo decide cómo tratar cada uno de ellos. Cuentan fundamentalmente con cuatro elecciones: evitarlo, limitarlo, atenuarlo, o controlarlo.

- Algunos riesgos pueden y deberían evitarse, quizá mediante una replanificación del proyecto o un cambio en los requisitos.
- Otros riesgos deberían limitarse, es decir, restringirse de modo que sólo afecten a una pequeña parte del proyecto.
- Algunos riesgos pueden atenuarse ejercitándolos y observando si aparecen o no. Si aparece, el aspecto positivo es que el equipo ha aprendido más sobre él. Puede que el equipo esté en disposición de encontrar una forma de evitarlo, limitarlo o controlarlo.

- Sin embargo, hay riesgos que no pueden atenuarse. Lo único que puede hacer el equipo es controlarlos y observar si aparecen. Si esto ocurre, el equipo debe seguir sus **planes de contingencia** (Apéndice C). Si aparece un riesgo "asesino de proyectos", debemos respirar hondo y analizar la situación. ¿Queremos continuar, o deberíamos parar el proyecto? Hasta el momento sólo hemos gastado un tiempo y dinero limitados. Sabíamos que podía suceder uno de estos riesgos —éste es el motivo por el cual estamos haciendo las primeras iteraciones—. Por tanto, hicimos un buen trabajo encontrando un riesgo de esta magnitud antes de poner a trabajar a todos los desarrolladores en el proyecto.

El tratamiento de un riesgo lleva su tiempo. Evitarlo o limitarlo conlleva hacer una replanificación, o rehacer algún trabajo. La atenuación de un riesgo podría requerir que el equipo construyese algo que lo haga evidente. El controlar un riesgo implica la selección de un mecanismo de control, su puesta en marcha, y su ejecución. A su vez, la atenuación o control de los riesgos conlleva un esfuerzo de desarrollo importante, es decir, tiempo. Debido a que el tratamiento de los riesgos consume tiempo, una organización raramente puede tratarlos todos a la vez. Éste es el motivo por el cual es necesaria la priorización de las iteraciones. Esto es lo que queremos expresar con desarrollo iterativo dirigido por los riesgos. Y es una gestión de riesgos sólida.

5.4. La iteración genérica

Como hemos visto, las iteraciones difieren marcadamente en las diferentes fases del ciclo de desarrollo, como consecuencia de que los desafíos que afrontan los desarrolladores son diferentes en cada fase. Nuestra intención en esta sección es presentar el concepto de iteración a un nivel genérico: qué es, cómo planificarla, cómo organizarla, y cuál es su resultado. En la Parte III, trataremos las iteraciones de cada una de las cuatro fases en capítulos separados.

5.4.1. Lo que es una iteración

Una *iteración* es un miniproyecto —un recorrido más o menos completo a lo largo de todos los flujos de trabajo fundamentales— que obtiene como resultado una versión interna. Ésta es una explicación intuitiva de lo que es una iteración. Sin embargo, hemos ampliado esta definición para ser capaces de describir el trabajo que tiene lugar en una iteración por debajo de su superficie.

Podemos imaginar una iteración como un flujo de trabajo, lo cual significa que es una colaboración entre trabajadores (Apéndice C) que utilizan y producen artefactos. En el Proceso Unificado distinguimos entre **flujos de trabajo fundamentales** y **flujos de trabajo de iteración** (Apéndice C). Por ahora, nos son familiares los cinco flujos de trabajo fundamentales: requisitos, análisis, diseño, implementación y prueba. Estos flujos de trabajo fundamentales sólo sirven para ayudarnos a describir los flujos de trabajo de iteración, por razones pedagógicas. Por tanto, no hay nada mágico en lo que constituye un flujo de trabajo fundamental; fácilmente podríamos haber utilizado otro conjunto de flujos de trabajo fundamentales, como por ejemplo, uno que integrase el análisis y el diseño [2]. Se utilizan para simplificar la descripción de flujos de trabajo más concretos, de igual forma que una clase abstracta nos ayuda a describir

[2] No deben confundirse flujos de trabajo con procesos concurrentes. Los flujos de trabajo son colaboraciones que son útiles para la creación de descripciones.

clases concretas. Estos flujos de trabajo más concretos son los flujos de trabajo de iteración. Los flujos de trabajo fundamentales se describen en detalle en los Capítulos 6 al 11, y los flujos de trabajo de iteración en los Capítulos 12 al 16, partiendo de los fundamentales.

En la Figura 5.3, se describen los elementos genéricos del flujo de trabajo de cada iteración. Todas pasan por los cinco flujos de trabajo fundamentales. Todos comienzan con una actividad de planificación y terminan con un análisis. En la Parte III, describiremos cuatro iteraciones arquetípicas, una por cada fase del Proceso Unificado. Cada una de ellas reutiliza las descripciones de los flujos de trabajo fundamentales, pero de forma diferente.

¿En qué se diferencia esto de un modelo en cascada tradicional? Cada uno de los flujos de trabajo fundamentales es una colaboración entre un conjunto de trabajadores y artefactos. Sin embargo, hay coincidencias entre las iteraciones. Los trabajadores y los artefactos pueden participar en más de un flujo de trabajo. Por ejemplo, el ingeniero de componentes participa en tres flujos de trabajo: análisis, diseño e implementación. Por último, el flujo de trabajo de la iteración se crea superponiendo, uno encima de otro, un subconjunto elegido de los flujos de trabajo fundamentales, y añadiendo lo extra, como la planificación y el análisis.

Las primeras iteraciones se centran en la comprensión del problema y de la tecnología. En la fase de inicio, las iteraciones se preocupan de producir un análisis del negocio [3]. En la fase de elaboración, las iteraciones se orientan al desarrollo de la línea base de la arquitectura. En la fase de construcción, las iteraciones se dedican a la construcción del producto por medio de una serie de construcciones dentro de cada iteración, que acaban en un producto preparado para su distribución a la comunidad de usuarios. Sin embargo, como se muestra en la Figura 5.3, todas las iteraciones siguen el mismo patrón.

Cada iteración se analiza cuando termina. Uno de los objetivos es determinar si han aparecido nuevos requisitos o han cambiado los existentes, afectando a las iteraciones siguientes. Durante la planificación de los detalles de la siguiente iteración, el equipo también examina cómo afectarán los riesgos que aún quedan al trabajo en curso.

Una función que merece un énfasis especial en este momento son las **pruebas de regresión** (Apéndice C). Antes de finalizar una iteración, debemos estar seguros de que no hemos estro-

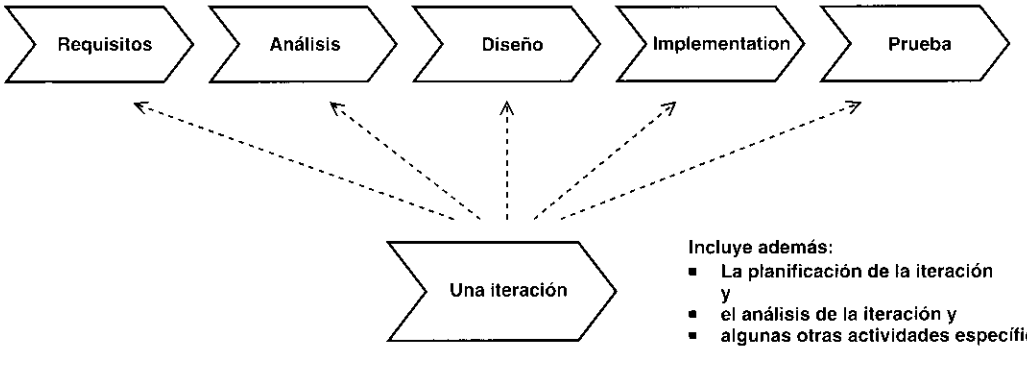


Figura 5.3. Cada iteración constituye una pasada a través de los cinco flujos de trabajo fundamentales. Se inicia con una actividad de planificación y se concluye con un análisis.

[3] Durante la fase de inicio, una iteración puede seguir una variante simplificada de los flujos de trabajo para estudiar problemas particulares relativos a la tecnología.

peado ninguna otra parte del sistema que funcionaba en anteriores iteraciones. La prueba de regresión es especialmente importante en un ciclo de vida iterativo e incremental, debido a que cada iteración produce una adición significativa al incremento previo, al mismo tiempo que una cantidad importante de cambios. Debemos señalar que no es práctico llevar a cabo las pruebas de regresión a tan gran escala —cada construcción de cada iteración— sin las herramientas de prueba adecuadas.

Los jefes de proyecto no deberían permitir que comience la siguiente iteración a menos que se hayan conseguido los objetivos de la presente. Si no se hace así, la planificación tendrá que cambiar para adecuarse a la nueva situación.

5.4.2. Planificación de las iteraciones

En cualquier caso, el ciclo de vida iterativo requiere más planificación y más reflexión que el método en cascada. En el modelo en cascada toda la planificación se hace al principio, a menudo antes de que se hayan reducido los riesgos y antes de que se haya asentado la arquitectura. Las planificaciones que se obtienen están basadas en mucha incertidumbre y falta de fidelidad. En contraste, el método iterativo no planifica el proyecto entero en detalle durante la fase de inicio, sólo da los primeros pasos. El equipo de proyecto no intenta planificar las fases de construcción y transición hasta que se ha establecido una base de hechos durante la fase de elaboración. Naturalmente, hay un plan de trabajo durante las dos primeras fases, pero no es muy detallado.

El esfuerzo de planificación tiene en cuenta normalmente (excepto al principio del todo del proyecto) los resultados de las iteraciones precedentes, la selección de los casos de uso relevantes en la nueva iteración, el estado actual de los riesgos que afectan a la nueva iteración, y el estado de la última versión del conjunto de modelos. Termina con la preparación de la versión interna.

Al término de la fase de elaboración, por tanto, existe una base para planificar el resto del proyecto y para poner en marcha un plan detallado para cada iteración de la fase de construcción. El plan para la primera iteración estará muy claro. Las posteriores aparecerán en el plan menos detalladas, y estarán sujetas a modificación, de acuerdo con los resultados y con el conocimiento que se adquiera en las iteraciones anteriores. De igual forma, debería haber un plan para la fase de transición, pero puede que tenga que ser modificado a la luz de lo que el equipo aprenda de las iteraciones de la fase de construcción. Este tipo de planificación nos permite un desarrollo iterativo controlado.

5.4.3. Secuenciación de las iteraciones

La evolución en la naturaleza sucede sin un plan que la preceda. Éste no es el caso de la iteración en el desarrollo de software. Para entendernos, los casos de uso establecen una meta, y la arquitectura establece un patrón. Con esta meta y este patrón en mente, los desarrolladores planifican la secuencia que seguirán en el desarrollo del producto.

Los planificadores tratan de ordenar las iteraciones para obtener un camino directo en el cual las primeras iteraciones proporcionan la base de conocimiento para las siguientes. Las primeras iteraciones del proyecto dan como resultado un mayor conocimiento de los requisitos, los

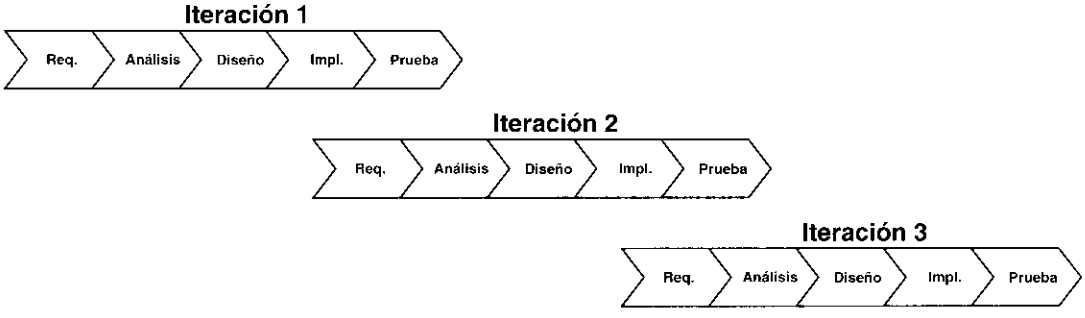


Figura 5.4. Las iteraciones discurren a lo largo de los flujos de trabajo desde la captura de requisitos hasta la prueba.

problemas, los riesgos, y el dominio de la solución, mientras que las siguientes dan como resultado incrementos aditivos que finalmente conforman la versión externa, es decir, el producto para el cliente. Para los planificadores, el éxito fundamental es una secuencia de iteraciones que siempre avanza, sin tener que volver nunca a los resultados de una iteración previa para corregir el modelo debido a algo que se descubrió en una iteración posterior.

Las iteraciones pueden solaparse en el sentido de que una iteración está a punto de terminar cuando otra está comenzando, como se muestra en la Figura 5.4. La planificación y el trabajo inicial de la siguiente iteración debe comenzar a medida que terminamos y preparamos para entregar la iteración anterior. Recuérdese que el final de una iteración significa que hemos conseguido una conclusión dentro del equipo de desarrollo. Se ha integrado todo el software de la iteración y puede crearse la versión interna.

El orden en el cual planificamos las iteraciones depende, en un grado considerable, de factores técnicos. Sin embargo, el objetivo más importante es ordenar el trabajo en secuencia de modo que puedan desarrollarse antes las decisiones más importantes, aquéllas que implican tecnologías, casos de uso y arquitecturas nuevas.

5.5. El resultado de una iteración es un incremento

Un incremento es la diferencia entre la versión interna de una iteración y la versión interna de la siguiente.

Al final de una iteración, el conjunto de modelos que representa al sistema queda en un estado concreto. Llamamos a este estado la *línea base*. Cada modelo ha alcanzado una línea base; cada elemento fundamental del modelo está en un estado de línea base. Por ejemplo, el modelo de casos de uso al final de cada iteración contiene un conjunto de casos de uso que representan el grado en que la iteración ha desarrollado los requisitos. Algunos de los casos de uso de este conjunto están terminados, mientras que otros sólo lo están parcialmente. A su vez, el modelo de diseño ha llegado a un estado de línea base consistente[4] con el modelo de casos de uso. Los subsistemas, interfaces, y realizaciones de casos de uso del modelo de diseño se en-

[4] No es necesario que se hayan diseñado todos los casos de uso, así que el término *consistente* hace referencia sólo a aquellos que se hayan diseñado.

cuentran también en líneas base que son mutuamente consistentes unas con otras. La organización del desarrollo debe mantener versiones consistentes y compatibles de todos los artefactos de cada línea base para poder trabajar de manera eficiente con muchas de ellas dentro del proyecto. Al emplear desarrollo iterativo, nunca es suficiente el énfasis que podamos hacer sobre la necesidad de herramientas eficientes de gestión de configuración.

En un momento dado de la secuencia de iteraciones, estarán terminados algunos subsistemas. Estos contendrán toda la funcionalidad requerida, y estarán implementados y probados. Otros subsistemas están parcialmente terminados, y otros están todavía vacíos, aunque contendrán resguardos de forma que puedan funcionar e integrarse con otros subsistemas. Por tanto, en términos más precisos, un incremento es la diferencia entre dos líneas base sucesivas.

Durante la fase de elaboración, como ya hemos señalado, construimos la línea base de la arquitectura. Identificamos los casos de uso que tienen un impacto significativo sobre la arquitectura, y los representamos como colaboraciones. Ésta es la forma en que identificamos los subsistemas y las interfaces —por lo menos, los que son interesantes para la arquitectura—. Una vez que hemos identificado la mayoría de los subsistemas e interfaces, les damos forma, es decir, escribimos el código que los implementa. Parte de este trabajo se hace antes de obtener la línea base de la arquitectura, y continúa a lo largo de todos los flujos de trabajo. Sin embargo, la mayor parte de la creación tiene lugar en las iteraciones de la fase de construcción.

A medida que nos acercamos a la fase de transición, incrementa el nivel de consistencia entre los modelos y dentro de ellos. Construimos incrementos dando forma a los modelos de manera iterativa, y la integración del último incremento se convierte en el sistema final.

5.6. Las iteraciones sobre el ciclo de vida

Cada una de las cuatro fases termina con un hito principal, como se muestra en la Figura 5.5. [1]:

- Inicio: objetivos del ciclo de vida.
- Elaboración: arquitectura del ciclo de vida.
- Construcción: funcionalidad operativa inicial.
- Transición: versión del producto.

El objetivo de cada hito principal es garantizar que los diferentes modelos de flujo de trabajo evolucionan de manera equilibrada durante el ciclo de vida del producto. Entendemos "equilibrada" en el sentido de que se toman pronto dentro del ciclo de vida las decisiones más importantes que influyen en esos modelos, las relativas a los riesgos, casos de uso, y arquitectura. Más adelante, deberíamos ser capaces de continuar el trabajo con niveles de detalle crecientes obteniendo una mayor calidad.

Los objetivos fundamentales de la fase de inicio son el establecimiento del ámbito de lo que debería hacer el producto, la reducción de los peores riesgos, y la preparación del análisis del negocio inicial, que indique que merece la pena realizar el proyecto desde la perspectiva del negocio. Dicho de otra forma, pretendemos establecer los objetivos del ciclo de vida para el proyecto.

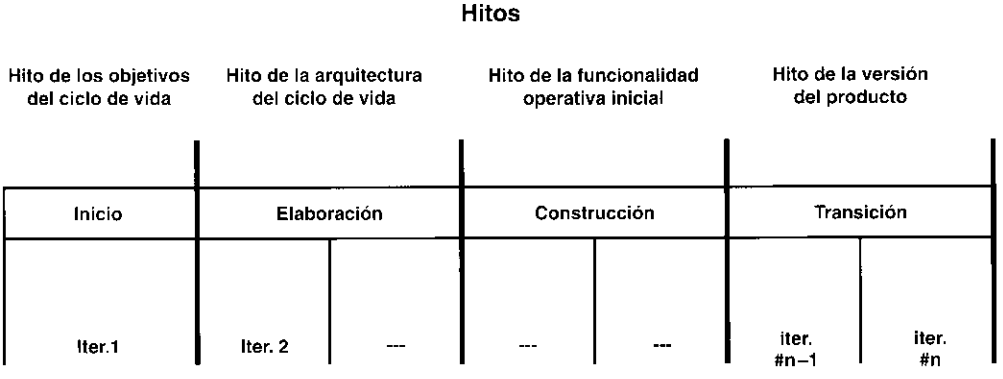


Figura 5.5. Las fases reúnen iteraciones que dan como resultado los hitos principales, sobre los cuales la dirección toma decisiones de negocio importantes. (El número de iteraciones no es fijo, sino que varía para diferentes proyectos.)

Los objetivos fundamentales de la fase de elaboración son obtener la línea base de la arquitectura, capturar la mayoría de los requisitos, y reducir los siguientes peores riesgos, es decir, establecer la arquitectura del ciclo de vida. Al final de esta fase, somos capaces de estimar los costes y las fechas y de planificar la fase de construcción con algún detalle. En este momento, deberíamos ser capaces de hacer nuestra apuesta.

Los objetivos fundamentales de la fase de construcción son el desarrollo del sistema entero y la garantía de que el producto puede comenzar su transición a los clientes, es decir, que tiene una funcionalidad operativa inicial.

El objetivo fundamental de la fase de transición es garantizar que tenemos un producto preparado para su entrega a la comunidad de usuarios. Durante esta fase del desarrollo, se enseña a los usuarios a utilizar el software.

Dentro de cada fase hay hitos menores, en concreto, los criterios aplicables a cada iteración. Cada iteración produce resultados, *artefactos del modelo*. Por tanto, al final de cada iteración, tendremos un nuevo incremento del modelo de casos de uso, del modelo de análisis, del modelo de diseño, del modelo de despliegue, del modelo de implementación, y del modelo de pruebas. El nuevo incremento se integrará con el resultado de la anterior iteración, obteniendo una nueva versión del conjunto de modelos.

Al llegar a los hitos secundarios, los jefes y los desarrolladores deciden cómo continuar en las subsiguientes iteraciones, de la forma que hemos explicado en las secciones anteriores. Al llegar a los hitos principales del final de las fases, los jefes toman decisiones cruciales de tipo continuar/no continuar y determinan el calendario, el presupuesto, y los requisitos.

Un hito secundario (en el momento de una versión interna, al final de una iteración) es un paso planificado hacia el hito principal al final de la fase. La distinción entre hitos principales y secundarios se hace esencialmente en el nivel de gestión. Los desarrolladores tratan los riesgos de manera iterativa y construyen artefactos de software hasta alcanzar el hito principal. En cada hito principal, la dirección evalúa lo que han conseguido los desarrolladores. Cada transición al pasar un hito principal representa por tanto una decisión de gestión importante y el compromiso de financiar el trabajo de (por lo menos) la siguiente fase de acuerdo al plan. Podemos imaginar los hitos principales como puntos de sincronización donde se juntan los dominios de gestión y técnicos.

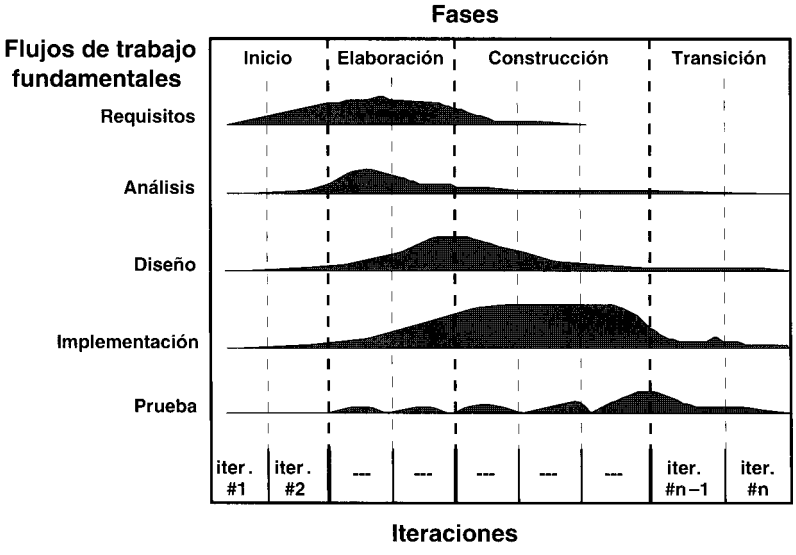


Figura 5.6. El énfasis se desplaza en las iteraciones, desde la captura de requisitos y el análisis hacia el diseño, la implementación y la prueba.

Estas divisiones ayudan a la dirección y a otros usuarios implicados a evaluar lo hecho durante las fases de bajo coste de inicio y elaboración antes de tomar la decisión de comprometerse con la fase de construcción de alto coste.

Un proyecto de desarrollo de software puede dividirse aproximadamente en dos trozos: las fases de inicio y elaboración y las fases de construcción y transición. Durante las fases de inicio y elaboración, construimos el análisis del negocio, atenuamos los peores riesgos, creamos la línea base de la arquitectura, y planificamos el resto del proyecto con una alta precisión. Este trabajo lo realiza un equipo pequeño, de bajo coste.

Después, el proyecto pasa a la fase de construcción donde el objetivo es la economía de escala. En este momento, el número de personas en el proyecto aumenta. Desarrollan el grueso de la funcionalidad del sistema construyendo sobre la arquitectura cuya línea base se obtuvo durante la fase de elaboración. Reutilizan el software existente tanto como es posible.

Aunque cada iteración discurre a lo largo de los flujos de trabajo de requisitos, análisis, diseño, implementación y prueba, las iteraciones tienen énfasis diferentes en las distintas fases, como se muestra en la Figura 5.6. Durante las fases de inicio y elaboración, la mayoría del esfuerzo se dedica a la captura de requisitos y a un análisis y diseño preliminares. Durante la construcción el énfasis pasa al diseño detallado, la implementación, y la prueba. Aunque no se muestra en la Figura 5.6, las primeras fases tienen una gran carga de gestión de proyecto y de preparación del entorno de desarrollo para el proyecto.

5.7. Los modelos evolucionan con las iteraciones

Las iteraciones construyen los modelos resultantes incremento por incremento. Cada iteración añade algo más a cada modelo, a medida que la iteración fluye a lo largo de requisitos, análisis, di-

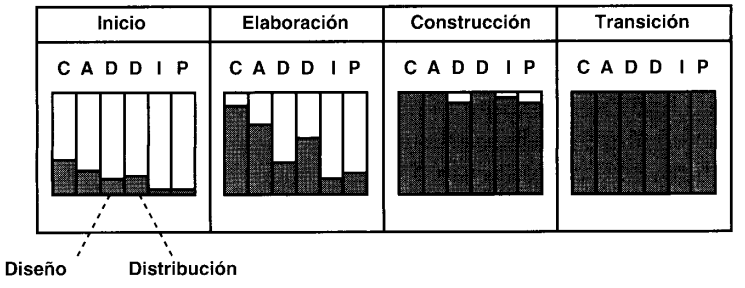


Figura 5.7. El trabajo en todos los modelos continúa en todas las fases, como se indica con el relleno creciente de los modelos. La fase de construcción finaliza con un conjunto de modelos (casi) completo. Sin embargo, estos modelos necesitan ajustes durante la transición a medida que se distribuyen en la comunidad de usuarios.

seño, implementación y prueba. Algunos de estos modelos, como el de casos de uso, reciben más atención en las primeras fases, mientras que otros, como el de implementación, la reciben durante la fase de construcción, como muestra el gráfico de la Figura 5.7. En la fase de inicio, el equipo crea quizá las partes de los modelos que son necesarias para soportar un prototipo que sirva como prueba de concepto. Estas partes incluyen los elementos más importantes del modelo de casos de uso (C), el modelo de análisis (A), el modelo de diseño (D), así como algo de los modelos de distribución (D), implementación (I), y prueba (P). La mayoría del material de implementación es preliminar en esta etapa. Como muestra la Figura 5.7, aún queda mucho trabajo por hacer.

En la fase de elaboración, el área sombreada, que denota el trabajo terminado, avanza de manera bastante significativa. Al final de esta fase, sin embargo, mientras que el equipo ha capturado más o menos un 80 por ciento de los requisitos (U) y del modelo de despliegue (la segunda D), se ha "incorporado" al sistema menos del 10 por ciento en cuanto a implementación (I) y prueba (P). Los modelos de casos de uso y de implantación deben estar así de avanzados tras la fase de elaboración. De otro modo, no conocemos suficientemente bien los requisitos ni las precondiciones de implementación (incluida la arquitectura) como para planificar con precisión la fase de construcción[5].

La fase de construcción termina con la mayoría de las partes C, A, D, D, I y T, como era de esperar, ya que el criterio de finalización es tener una implementación completa del sistema preparada para su transición a la comunidad de usuarios. Más adelante, a medida que se pasa el sistema a su uso operativo en la fase de transición, tendrán lugar correcciones secundarias y algunos ajustes.

5.8. Las iteraciones desafían a la organización

Muchas organizaciones de software tienden a lanzarse a escribir código porque son las líneas de código lo que los jefes tienen en cuenta. Tienden a resistirse al cambio, debido a que el cambio disminuye la cuenta del código. No están interesadas en reutilizar el análisis, el diseño o el código, porque sus jefes sólo tienen en cuenta el código nuevo.

---

[5] En el Capítulo 4, indicamos que los modelos de casos de uso y de análisis habían alcanzado un menor nivel de avance al final de la fase de elaboración de lo que indica la Figura 5.7. La razón de esta discrepancia es que en el Capítulo 4 nos centrábamos exclusivamente en la arquitectura, y no considerábamos qué otro trabajo debía hacerse (es decir, comprender más los casos de uso para ser capaces de hacer un análisis del negocio).

El pasar a un desarrollo iterativo desafía las prácticas de estas organizaciones. Requiere un cambio de actitud. El énfasis de la organización debe pasar de contar líneas de código a reducir los riesgos y a crear la línea base de funcionalidad para la arquitectura. Los jefes deben tener una mirada nueva sobre lo que miden. Deberán demostrar por sus acciones que miden el progreso en términos de los riesgos tratados, los casos de uso preparados, y los componentes que realizan esos casos de uso. De otro modo, los desarrolladores pronto volverán a aquello a lo cual estaban acostumbrados, las líneas de código.

La aplicación del método iterativo e incremental tiene algunas consecuencias importantes:

- Para crear el análisis del negocio en la fase de inicio, la organización se ha centrado en la reducción de riesgos críticos y en demostrar una prueba de concepto.
- Para hacer una apuesta que merezca la pena para el negocio al término de la fase de elaboración, la organización debe saber qué va a contratar para desarrollo (representado por la línea base de la arquitectura unida a los requisitos), y debe tener confianza en que no contiene riesgos ocultos (es decir, costes insuficientemente examinados y posibilidades de salirse del calendario).
- Para minimizar los costes, los defectos, y el tiempo de salida al mercado, la organización debe emplear componentes reutilizables (resultado del desarrollo temprano de la arquitectura, basado en el estudio del dominio sobre el que se sitúa el sistema propuesto).
- Para evitar el retraso en la entrega, el sobrepasar los costes, y el obtener un producto de baja calidad, la organización debe "hacer primero la parte difícil".
- Para evitar construir un producto fuera de plazos, la organización no puede decir no a todos los cambios tozudamente. La técnica por fases, iterativa, permite acomodar los cambios en el desarrollo hasta mucho más adelante en la secuencia del desarrollo.

El desarrollo iterativo e incremental requiere no sólo una nueva forma de gestionar los proyectos, sino también herramientas que soporten este nuevo método. Es prácticamente imposible tratar con todos los artefactos del sistema que están sujetos a cambios concurrentemente en cada construcción y en cada incremento sin el apoyo de las herramientas. Una organización que afronte esta forma de desarrollo necesita el soporte de herramientas para los diferentes flujos de trabajo, así como herramientas para gestión de la configuración y control de versiones.

5.9. Referencias

[1] Barry Boehm, "Anchoring the software process", IEEE Software, July 1996, pp. 73–82.

[2] Peter F. Drucker, Management: Tasks, Responsibilities, Practices, New York: Harper & Row, 1973.

[3] Barry W. Boehm, "A spiral model of software development and enhancement", Computer, May 1988, pp. 61–72. (Reprinted in Barry W. Boehm, Tutorial: Software Risk Management, IEEE Computer Society Press, Los Alamitos, CA, 1989.)

[4] Capers Jones, Assessment and Control of Software Risks, Upper Saddle River, NJ: Prentice-Hall, 1993.

[5] Walker Royce, "TRW's Ada process model for incremental development of large software systems", Proceedings, 12th International Conference on Software Engineering, 1990, pp. 2–11.

Parte II

Los flujos de trabajo fundamentales

Ahora que comprendemos las nociones básicas que subyacen en las prácticas fundamentales del Proceso Unificado, describiremos cada flujo de trabajo en detalle. La Parte III cubrirá las iteraciones y las fases.

En esta Parte II presentamos los flujos de trabajo *fundamentales* uno por uno en capítulos separados: los Capítulos 6 y 7 tratan los requisitos; el Capítulo 8, el análisis; el Capítulo 9, el diseño; el Capítulo 10, la implementación; y el Capítulo 11, la prueba. El término *flujo de trabajo fundamental* es una abstracción, y se explica en detalle en el Capítulo 5. Durante las iteraciones, nos centramos en desarrollar los flujos de trabajo de forma precisa, a este tema se dedica la Parte III.

La descripción separada de los flujos de trabajo fundamentales, como estamos a punto de hacer, podría confundir al lector, y queremos estar seguros de que esto no suceda. En primer lugar, mediante la descripción separada de los flujos de trabajo uno detrás de otro, damos la impresión de que el proceso de desarrollo de software general, del comienzo al fin del proyecto, pasa por una secuencia de flujos de trabajo sólo una vez. Un lector podría llegar a pensar que los flujos de trabajo fundamentales son un proceso de una pasada, como el antiguo proceso en cascada. En segundo lugar, un lector descuidado podría concluir que cada flujo de trabajo fundamental es un paso monolítico en el proceso.

Ninguna de estas impresiones es acertada. Describimos los flujos de trabajo en capítulos separados solamente como una forma de explicar por completo, por motivos pedagógicos, la totalidad de cada flujo de trabajo.

En referencia al primer tema, la posibilidad de parecerse al ciclo de vida en cascada, recorremos los cinco flujos de trabajo secuencialmente, pero lo hacemos una vez por cada iteración,

no una vez para el proyecto completo. Por tanto, si tenemos siete iteraciones sobre cuatro fases, podríamos llevar a cabo los flujos de trabajo siete veces. Para ser un poco más precisos, podríamos no utilizar los cinco flujos de trabajo al principio de la fase de inicio; es decir, podríamos no llegar a los últimos flujos de trabajo, como la implementación y la prueba, en la primera iteración. El principio es claro: llevamos a cabo los flujos de trabajo en cada iteración mientras sea necesario para cada iteración en concreto.

En referencia al segundo tema, el paso monolítico, describimos cada flujo de trabajo fundamental de forma bastante independiente de los otros. Sin embargo, hemos intentado simplificar un poco cada flujo de trabajo centrándonos en sus actividades básicas, una vez más, por motivos pedagógicos. No hemos entrado en las alternancias con las actividades en otros flujos de trabajo. Por supuesto, estas alternancias son esenciales en un proceso de desarrollo de software iterativo, y las tratamos en detalle en la Parte III. Por ejemplo, mientras estamos trabajando en un determinado diseño, un desarrollador podría considerar deseable alternar entre los flujos de trabajo de análisis y diseño.

En la Parte III describimos cómo los flujos de trabajo, que se describen por separado en esta parte, se combinan en un proyecto en ejecución. Por ejemplo, puede ser apropiado para una primera fase el tener un conjunto de flujos de trabajo limitado, mientras que en la fase de construcción es necesario el conjunto completo de flujos de trabajo.

Capítulo 6

Captura de requisitos: de la visión a los requisitos

Desde hace generaciones, ciertas tribus de americanos nativos construyen una especie de canoas, llamadas *dugouts,* hechas con un tronco ahuecado. Los constructores de canoas comienzan buscando un árbol que mida varios pies de diámetro y que esté ya derribado cerca del agua. A su lado, encienden un fuego y cubren el tronco con las brasas ardientes. La madera carbonizada es mucho más fácil de vaciar con las herramientas de piedra. Después de varios días tallando, la canoa aparenta estar terminada, y los constructores pueden empujarla y meterla en aguas poco profundas. Más que probablemente, al primer esfuerzo sencillamente dará vueltas. No estará equilibrada. Hará falta más trabajo con aquellas torpes herramientas de piedra, hasta obtener una barca que no zozobre cuando alguien se incline para sacar un pez del agua. Sólo entonces se consideraba terminada. Este conocimiento ha pasado de generación en generación, y los constructores lo han convertido en algo propio.

Cuando una "tribu" de desarrolladores de software escucha la llamada para desarrollar un nuevo sistema, se enfrentan a una situación muy diferente. En primer lugar, los desarrolladores no serán los futuros usuarios del sistema. No obtendrán de sí mismos la retroalimentación de qué tal funcionan sus "dugouts". En segundo lugar, los requisitos y restricciones del sistema no están profundamente arraigados en sus "médulas" a través de un uso continuo del producto desde la infancia. En su lugar, tienen que descubrir por sí mismos lo que se necesita.

Llamamos *captura de requisitos* a este acto de descubrimiento. Es el proceso de averiguar, normalmente en circunstancias difíciles, lo que se debe construir. De hecho, es tan difícil que

todavía no es poco común para los equipos de proyecto el comenzar a escribir código (lo cual es bastante fácil) antes de que hayan firmado simplemente lo que se supone que debe hacer el código (lo cual es difícil de determinar).

6.1. Por qué la captura de requisitos es complicada

Los desarrolladores de software profesionales normalmente crean código para otros y no para sí mismos —lo hacen para los usuarios del software—. "¡Ajá!", suelen decir los desarrolladores, "los usuarios deben saber lo que quieren". Sin embargo, una mínima experiencia intentando recoger requisitos de los usuarios pronto los revela como una fuente imperfecta de información. De un modo u otro, normalmente cualquier sistema tiene muchos usuarios (o tipos de usuarios), y mientras que cada uno de ellos puede saber lo que él o ella hacen, ninguno tiene una visión global. Los usuarios no saben cómo puede hacerse más eficiente la operación en su conjunto. La mayoría de los usuarios no sabe qué partes de su trabajo pueden transformarse en software. Francamente, con frecuencia los usuarios no saben cuáles son los requisitos ni tampoco cómo especificarlos de una forma precisa.

La aproximación tradicional a este problema ha sido asignar intermediarios —analistas— para obtener una lista de requisitos de cada usuario con la esperanza de que el analista fuese capaz de tener una visión de conjunto y de componer una especificación de requisitos completa, correcta y consistente. Típicamente, los analistas registraban los requisitos en documentos que llegaban a cientos o a veces miles de páginas. Pero es absurdo pensar que la mente humana es capaz de conseguir una lista consistente y relevante de requisitos del tipo "el sistema debe..." Es más, estas especificaciones de requisitos no se transformaban fácilmente en especificaciones de diseño e implementación.

Incluso con la ayuda de los analistas, los usuarios no comprendían del todo lo que el sistema software debería hacer hasta que el sistema estaba casi terminado. A medida que el proyecto avanzaba, los usuarios, los intermediarios y los propios desarrolladores podían ver cómo parecería el sistema, y por tanto llegaban a comprender mejor las verdaderas necesidades, y se sugería una gran cantidad de cambios. Muchos de esos cambios eran deseables, pero su implementación tenía con frecuencia un impacto importante en los plazos y en los costes.

Durante años nos hemos engañado a nosotros mismos creyendo que los usuarios saben cuáles son los requisitos y que lo único que tenemos que hacer es entrevistarnos con ellos. Es cierto que los sistemas que construimos deberían dar soporte a los usuarios, y que podemos aprender de ellos sobre sus interacciones. Sin embargo, es aún más importante que los sistemas den soporte a la *misión* para la cual se construyen. Por ejemplo, el sistema debería proporcionar valor al negocio que lo utiliza y a sus clientes. Con frecuencia, es difícil identificar o comprender qué es este valor, y a veces es imposible hacer que el sistema lo proporcione. Aún peor, como reflejo del mundo real en constante cambio, este valor escurridizo probablemente cambiará durante la vida del proyecto: puede cambiar el negocio en sí, puede cambiar la tecnología disponible para construir el sistema, pueden cambiar los recursos (personas, dinero) disponibles para construir el sistema, etc.

Aún con estas ideas, la captura de requisitos sigue siendo difícil, y la industria lleva buscando un proceso bueno, sistemático para llevarla a cabo desde hace mucho tiempo. Nos centraremos en esto en este capítulo y en el siguiente.

6.2. El objeto del flujo de trabajo de los requisitos

El propósito fundamental del flujo de trabajo de los requisitos es guiar el desarrollo hacia el sistema correcto. Esto se consigue mediante una descripción de los requisitos del sistema (es decir, las condiciones o capacidades que el sistema debe cumplir) suficientemente buena como para que pueda llegarse a un acuerdo entre el cliente (incluyendo a los usuarios) y los desarrolladores sobre qué debe y qué no debe hacer el sistema.

Un reto fundamental para conseguirlo es que el cliente, que asumimos que la mayor parte de las veces será un especialista no informático, debe ser capaz de leer y comprender el resultado de la captura de requisitos. Para alcanzar este objetivo debemos utilizar *el lenguaje del cliente* para describir esos resultados. Consecuentemente, deberíamos tener mucho cuidado a la hora de introducir en los resultados formalidad y estructura, y cuando incluyamos detalles sobre el funcionamiento interno del sistema.

Los resultados del flujo de trabajo de los requisitos también ayudan al jefe de proyecto a planificar las iteraciones y las versiones del cliente (lo explicaremos en la Parte III).

6.3. Visión general de la captura de requisitos

Cada proyecto software es diferente. Esta singularidad proviene de las diferencias en el tipo de sistema, en el cliente, en la organización de desarrollo, en la tecnología, etc. De igual forma, hay diferentes puntos de partida para la captura de requisitos. En algunas ocasiones, comenzamos haciendo un modelo del negocio, o comenzamos con un modelo del negocio que ya está en desarrollo por parte de alguna otra empresa (*véase* la Sección 6.6.1, "¿Qué es un modelo del negocio?"). En otros casos, el software es un sistema empotrado que no da soporte directamente al negocio. Entonces podríamos tener como entrada un modelo de objetos sencillo, como un modelo del dominio (*véase* la Sección 6.5.1, "¿Qué es un modelo del dominio?") Y en otros casos, el cliente incluso puede haber ya desarrollado una especificación de requisitos completa y detallada que no esté basada en un modelo de objetos, a partir de la cual podemos comenzar y negociar los cambios.

En el otro extremo tenemos clientes que sólo tienen una vaga noción de lo que debería ser su sistema —quizás derivada de un informe de visión general creado por la alta dirección—. Entre estos extremos, tenemos toda la variedad de combinaciones. Tomaremos uno de estos puntos de partida, el de la "vaga noción", y presentaremos el ejemplo que utilizaremos en el resto del libro.

Ejemplo El Consorcio Interbank estudia un sistema informático

El Consorcio Interbank, una institución financiera ficticia, se enfrenta a cambios importantes debidos a la desregulación, la nueva competencia, y las posibilidades abiertas por la World Wide Web. El Consorcio quiere desarrollar aplicaciones nuevas que soporten los rápidamente cambiantes mercados financieros. Ha encargado a su subsidiaria de desarrollo de software, Interbank Software, que comience el desarrollo de esas aplicaciones.

Interbank Software decide diseñar el Sistema de Pagos y Facturación en colaboración con algunos de sus principales clientes. El sistema utilizará Internet para el envío de pedidos, facturas, y

pagos entre compradores y vendedores. La motivación del banco para el desarrollo del sistema es atraer nuevos clientes ofreciendo comisiones bajas por el proceso de los pagos. El banco también podrá reducir sus costes laborales procesando las solicitudes de cambio automáticamente por Internet, en lugar de manualmente mediante cajeros.

Las motivaciones para compradores y vendedores son reducir los costes, el papeleo y el tiempo de proceso. Por ejemplo, ya no tendrán que enviar pedidos o facturas por correo ordinario. El pago de las facturas se llevará a cabo entre los computadores del comprador y del vendedor. Ambos tendrán también una visión general mejor del estado de sus facturas y pagos.

La posibilidad de tener puntos de partida tan dispares como una vaga noción y una especificación de requisitos detallada sugiere que los analistas necesitan ser capaces de adaptar sus técnicas a la captura de requisitos en cada situación. Los diferentes puntos de partida plantean tipos diferentes de riesgos, por lo que los analistas deberían elegir las técnicas que reduzcan esos riesgos de la mejor forma. La reducción de los riesgos se trata en detalle en la Parte III.

Aparte de las diferencias en los puntos de partida, ciertos pasos son factibles en la mayoría de los casos, lo que nos permite sugerir un flujo de trabajo arquetípico. Este flujo de trabajo incluye los siguientes pasos, que en la realidad no se llevan a cabo separadamente:

- Enumerar los requisitos candidatos.
- Comprender el contexto del sistema.
- Capturar requisitos funcionales.
- Capturar requisitos no funcionales.

Describiremos brevemente cada uno de estos pasos en los párrafos siguientes.

Enumerar los requisitos candidatos Durante la vida del sistema, los clientes, usuarios, analistas y desarrolladores aparecen con muchas buenas ideas que podrían convertirse en verdaderos requisitos. Mantenemos una lista de estas ideas, que consideramos como un conjunto de requisitos candidatos que podemos decidir implementar en una versión futura del sistema. Esta *lista de características* crece a medida que se añaden nuevos elementos y mengua cuando algunas características se convierten en requisitos y se transforman en otros artefactos como casos de uso. La lista de características se utiliza sólo para la planificación del trabajo.

Cada característica tiene un nombre corto y una breve explicación o definición, información suficiente para poder hablar de ella durante la planificación del producto. Cada característica tiene también un conjunto de valores de planificación que podríamos incluir:

- Estado (propuesto, aprobado, incluido o validado).
- Coste estimado de implementación (en términos de tipos de recursos y horas-persona).
- Prioridad (crítico, importante o secundario).
- Nivel de riesgo asociado a la implementación de la característica (crítico, significativo u ordinario; *véase* el Capítulo 5).

Estos valores de planificación se utilizan junto con otros aspectos (como se tratará en el Capítulo 12) para estimar el tamaño del proyecto y para decidir cómo dividir el proyecto en una secuencia de iteraciones.

La prioridad y el nivel de riesgo asociados con una característica se utilizan, por ejemplo, para decidir la iteración en la que se implementará la característica (como se trata en la Parte III). Además, cuando se planifica la implementación de una característica, se le añadirán trazas a uno o más casos de uso o requisitos adicionales (lo trataremos en breve).

Comprender el contexto del sistema Muchas de las personas implicadas en el desarrollo de software son especialistas en temas relativos al software. Sin embargo, para capturar los requisitos correctos y para construir el sistema correcto, los desarrolladores clave —en particular el arquitecto y algunos de los analistas senior— requieren un firme conocimiento del contexto en el que se emplaza el sistema.

Hay por lo menos dos aproximaciones para expresar el contexto de un sistema en una forma utilizable para desarrolladores de software: modelado del dominio y modelado del negocio. Un modelo del dominio describe los conceptos importantes del contexto como objetos del dominio, y enlaza estos objetos unos con otros. La identificación y la asignación de un nombre para estos objetos nos ayuda a desarrollar un glosario de términos que permitirán comunicarse mejor a todos los que están trabajando en el sistema. Más adelante, los objetos del dominio nos ayudarán a identificar algunas de las clases a medida que analizamos y diseñamos el sistema. Como se verá, un modelo del negocio puede describirse como un supraconjunto de un modelo del dominio, e incluye algo más que sólo los objetos del dominio.

El objetivo del modelado del negocio es describir los procesos —existentes u observados— con el objetivo de comprenderlos. El modelado del negocio es la única parte de la ingeniería de negocio que hemos utilizado en este libro [3]. Baste decir por ahora que la ingeniería de negocio es muy parecida al modelado del negocio, pero también tiene el objetivo de mejorar los procesos de negocio de la organización.

A medida que los analistas modelan el negocio aprenden mucho sobre el contexto del sistema software, y lo describen en un modelo del negocio. El modelo del negocio especifica qué procesos de negocio soportará el sistema. Aparte de identificar los objetos del dominio o del negocio implicados en el negocio, este modelado también establece las competencias requeridas en cada proceso: sus trabajadores, sus responsabilidades, y las operaciones que llevan a cabo. Por supuesto, este conocimiento es decisivo en la identificación de los casos de uso, como se tratará en breve. De hecho, la técnica de ingeniería de negocio es un proceso más sistemático para capturar los requisitos en las aplicaciones empresariales [3].

El arquitecto y el jefe de proyecto juntos deciden si se prepara un modelo del dominio, si se recorre todo el camino y se prepara un modelo del negocio entero, o si no se hace ninguno de esos modelos.

Capturar requisitos funcionales La técnica inmediata para identificar los requisitos del sistema se basa en los casos de uso (los casos de uso se tratan con profundidad en el Capítulo 7). Estos casos de uso capturan tanto los requisitos funcionales como los no funcionales que son específicos de cada caso de uso.

Vamos a recapitular brevemente cómo los casos de uso nos ayudan a capturar los requisitos adecuados. Cada usuario quiere que el sistema haga algo para él o ella, es decir, que lleve a cabo ciertos casos de uso. Para el usuario, un caso de uso es un modo de utilizar el sistema. En consecuencia, si los analistas pueden describir todos los casos de uso que necesita el usuario, entonces saben lo que debe hacer el sistema.

Cada caso de uso representa una forma de usar el sistema (de dar soporte a un usuario durante un proceso de negocio). Cada usuario necesita varios casos de uso distintos, cada uno de los cuales representa los modos diferentes en los cuales él o ella utiliza el sistema. La captura de los casos de uso que realmente se quieren para el sistema, como aquellos que soportarán el negocio y que el usuario piensa que le permiten trabajar "cómodamente", requiere que conozcamos en profundidad las necesidades del usuario y del cliente. Para hacerlo tenemos que comprender el contexto del sistema, entrevistar a los usuarios, discutir propuestas, etc.

Como accesorio de los casos de uso, los analistas deben especificar también cuál será la apariencia de la interfaz de usuario cuando se lleven a cabo los casos de uso. La mejor forma de desarrollar esta especificación de interfaz de usuario es esbozar varias versiones que muestren la información que se transferirá, discutir los esbozos con los usuarios, y construir visualizaciones o prototipos concretos para que los usuarios los prueben.

Capturar requisitos no funcionales Los requisitos no funcionales especifican propiedades del sistema, como restricciones del entorno o de la implementación, rendimiento, dependencias de la plataforma, facilidad de mantenimiento, extensibilidad, y fiabilidad —todas las "-ades". La *fiabilidad* hace referencia a características como la disponibilidad, exactitud, tiempo medio entre fallos, defectos por miles de líneas de código (KLDC), y defectos por clase. Un requisito de rendimiento impone condiciones sobre los requisitos funcionales como la velocidad, rendimiento, tiempo de respuesta, y uso de memoria. La mayoría de los requisitos de rendimiento afectan sólo a ciertos casos de uso y por tanto deberían conectarse (como valores etiquetados) a ese caso de uso (Apéndice A). En la práctica, esto significa que estos requisitos se describirán "en la parte derecha", es decir, en la descripción del caso de uso (quizá en una sección aparte de Requisitos).

Ejemplo **Requisitos especiales para el caso de uso pagar factura**

Requisitos de rendimiento

Cuando un comprador envía una factura para su pago, el sistema debería responder con una verificación de la solicitud en menos de 1.0 segundos en el 90 por ciento de los casos. La duración de esta verificación nunca deberá exceder los 10.0 segundos a menos que la conexión de red no funcione (en cuyo caso se debe informar al usuario).

Algunos requisitos no funcionales hacen referencia a fenómenos del mundo real, como las cuentas en un sistema bancario. Estos requisitos pueden capturarse al principio en el objeto del dominio o del negocio correspondiente en el modelo de contexto del sistema.

Más adelante, cuando se determinen los casos de uso y los "conceptos" sobre los que realmente operan, estos requisitos no funcionales pasarán a relacionarse con los conceptos. Entendemos por "conceptos" bien los términos informales en un glosario utilizado para los casos de uso (*véase* el Capítulo 7), o, más formalmente, las clases en un modelo de análisis (*véase* el Capítulo 8). Por sencillez, asumimos la primera situación en este capítulo, es decir, estos requisitos se relacionan con conceptos del glosario.

Trabajo a realizar	Artefactos resultantes	
Enumerar requisitos candidatos	Lista de características	
Comprender el contexto del sistema	Modelo del dominio o del negocio	
Capturar los requisitos funcionales	Modelo de casos de uso	Definen una especificación de requisitos tradicional
Capturar los requisitos no funcionales	Requisitos adicionales o casos de uso concretos (para requisistos específicos de un caso de uso)	

Figura 6.1. El conjunto de todos los requisitos está formado por los diferentes artefactos que se muestran en la columna derecha. El trabajo a realizar influye en uno o más de estos artefactos. Obsérvese que los casos de uso también contienen los requisitos no funcionales que son específicos de los casos de uso.

Por último, algunos requisitos no funcionales son más genéricos y no pueden relacionarse con un caso de uso concreto o con una clase concreta del mundo real. Estos deberían gestionarse aparte en una lista de **requisitos adicionales** (Apéndice C).

Resumen Para capturar los requisitos de manera eficaz, los analistas necesitan un conjunto de técnicas y artefactos que les ayude a obtener una visión suficientemente buena del sistema para avanzar en los flujos de trabajo subsiguientes. Llamamos a este conjunto de artefactos colectivamente el *conjunto de requisitos*. La especificación de requisitos tradicional en el pasado se reemplaza por tanto por un conjunto de artefactos: el modelo de casos de uso y los requisitos adicionales. Los artefactos necesarios para establecer el contexto del sistema son los modelos del dominio y del negocio. Esto se muestra en la Figura 6.1.

Debido a que los requisitos cambian constantemente, necesitamos alguna forma de actualizarlos de manera controlada. Lo hacemos en las iteraciones, donde cada iteración reflejará algún cambio en el conjunto de requisitos, pero el número de cambios normalmente disminuirá a medida que nos adentremos en la fase de construcción y a medida que se estabilicen los requisitos. Esto se trata en profundidad en la Sección 6.4. Después nos concentramos en la descripción del contexto del sistema como modelo del dominio (Sección 6.5) o como modelo del negocio (Sección 6.6). Por último, explicaremos los requisitos adicionales (Sección 6.7).

La captura de requisitos en la forma de casos de uso es un tema mucho más largo, al cual volveremos en el Capítulo 7.

6.4. El papel de los requisitos en el ciclo de vida del software

El modelo de casos de uso se desarrolla a lo largo de varios incrementos del desarrollo, donde las iteraciones añadirán nuevos casos de uso y/o añadirán detalle a las descripciones de los casos de uso existentes.

La Figura 6.2 ilustra cómo el flujo de trabajo de captura de requisitos y sus artefactos resultantes adquieren diferentes formas durante las distintas fases y sus iteraciones (*véase* el Capítulo 12):

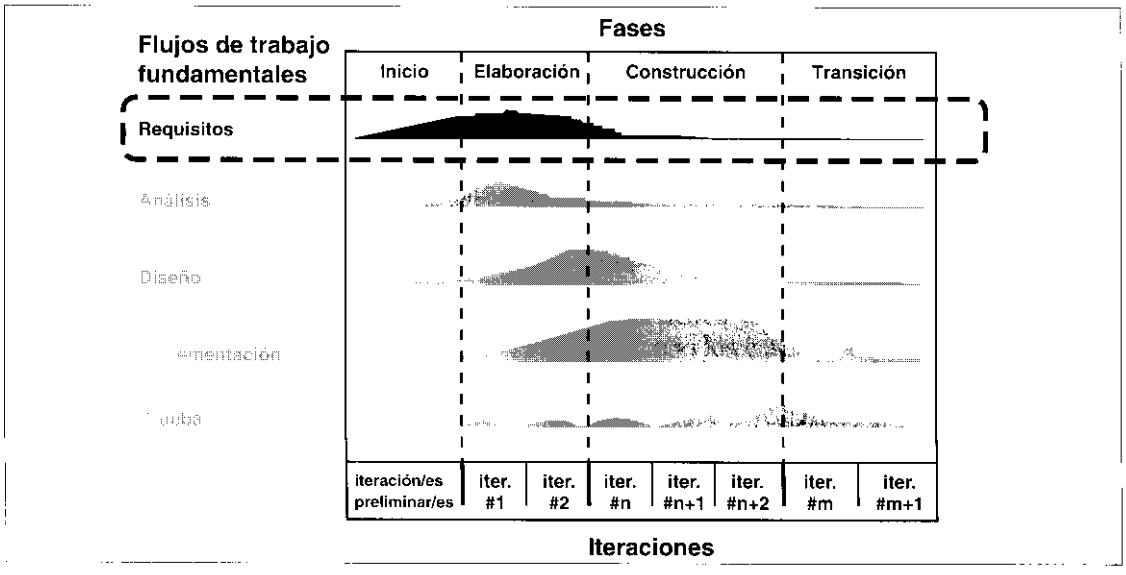


Figura 6.2. El trabajo de los requisitos se hace fundamentalmente durante el inicio y la elaboración.

- Durante la fase de inicio, los analistas identifican la mayoría de los casos de uso para delimitar el sistema y el alcance del proyecto y para detallar los más importantes (menos del 10 por ciento).
- Durante la fase de elaboración, los analistas capturan la mayoría de los requisitos restantes para que los desarrolladores puedan estimar el tamaño del esfuerzo de desarrollo que se requerirá. El objetivo es haber capturado sobre un 80 por ciento de los requisitos y haber descrito la mayoría de los casos de uso al final de esta fase de elaboración. (Obsérvese que sólo entre un 5 y un 10 por ciento de los requisitos debería estar implementado en la línea base en este momento.)
- Los requisitos restantes se capturan (e implementan) durante la fase de construcción.
- Casi no hay captura de requisitos en la fase de transición, a menos que haya requisitos que cambien.

6.5. La comprensión del contexto del sistema mediante un modelo del dominio

6.5.1. ¿Qué es un modelo del dominio?

Un modelo del dominio captura los tipos más importantes de objetos en el contexto del sistema. Los objetos del dominio representan las "cosas" que existen o los eventos que suceden en el entorno en el que trabaja el sistema [2,5].

Muchos de los objetos del dominio o clases (para emplear una terminología más precisa) pueden obtenerse de una especificación de requisitos o mediante la entrevista con los expertos del dominio. Las clases del dominio aparecen en tres formas típicas:

- Objetos del negocio que representan cosas que se manipulan en el negocio, como pedidos, cuentas y contratos.
- Objetos del mundo real y conceptos de los que el sistema debe hacer un seguimiento, como la aviación enemiga, misiles y trayectorias.
- Sucesos que ocurrirán o han ocurrido, como la llegada de un avión, su salida y la hora de la comida.

El modelo del dominio se describe mediante diagramas de UML (especialmente mediante diagramas de clases).

Estos diagramas muestran a los clientes, usuarios, revisores y a otros desarrolladores las clases del dominio y cómo se relacionan unas con otras mediante asociaciones.

Ejemplo Las clases del dominio pedido, factura, artículo y cuenta

El sistema utilizará Internet para enviar pedidos, facturas y pagos entre compradores y vendedores. El sistema ayuda al comprador a confeccionar sus pedidos, al vendedor a evaluar los pedidos y a enviar las facturas, y al comprador a validar las facturas y a hacer efectivos los pagos de su cuenta a la del vendedor.

Un pedido es la solicitud de un comprador a un vendedor de un número de artículos. Cada artículo "ocupa una línea" en el pedido. Un pedido posee atributos como la fecha de emisión y la dirección de entrega. *Véase* el diagrama de clases de la Figura 6.3.

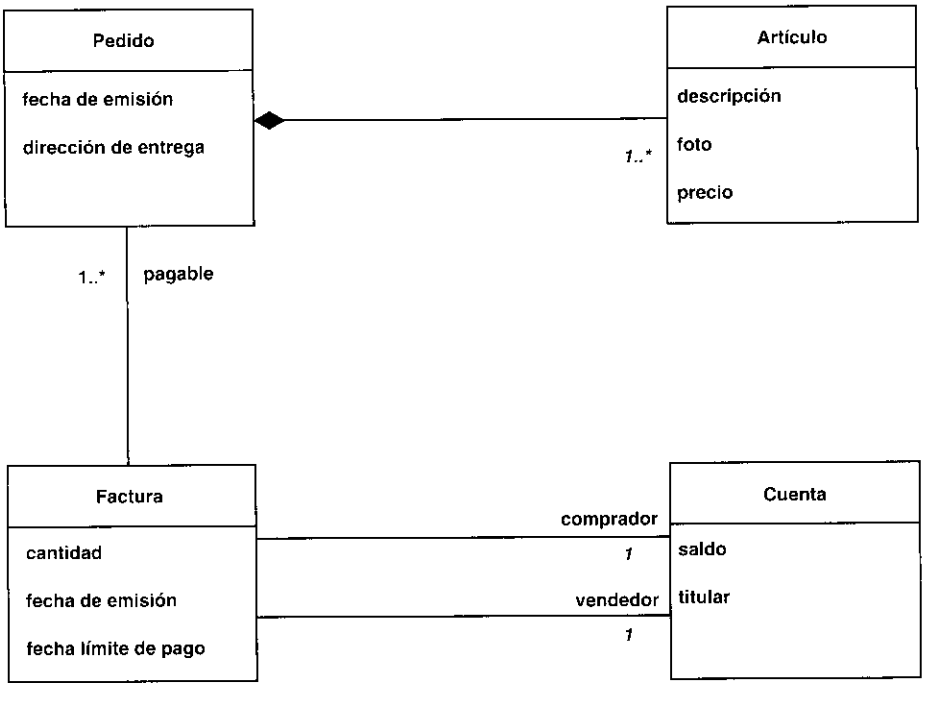


Figura 6.3. Un diagrama de clases en un modelo del dominio, que captura los conceptos más importantes del contexto del sistema.

Una factura es una solicitud de pago del vendedor al comprador, en respuesta a un pedido de bienes o servicios. Una factura posee atributos como la cantidad, fecha de emisión, y fecha límite de pago. Una factura puede ser la solicitud de pago de varios pedidos.

Las facturas se pagan mediante la transferencia de dinero de la cuenta del comprador a la del vendedor. Una Cuenta posee atributos como el saldo y el titular. El atributo titular identifica a la persona a la cual pertenece la cuenta.

Notación UML

Clases (rectángulos), atributos (texto en la parte inferior de los rectángulos de las clases), y asociaciones (las líneas entre los rectángulos de las clases). El texto al final de una línea de asociación explica el rol de la clase en relación con la otra, es decir, el rol de la asociación. La multiplicidad —los números y asteriscos al final de una línea de asociación— indican cuántos objetos de la clase de ese extremo pueden enlazarse a un objeto en el otro extremo. Por ejemplo, la asociación que conecta las clases Factura y Pedido en la Figura 6.3 tiene una multiplicidad 1..*, dibujada en el extremo de la clase Pedido. Esto significa que cada objeto Factura puede ser una solicitud de pago de uno o más objetos Pedido, como se indica en el **rol** (Apéndice A) de la asociación pagable.

6.5.2. Desarrollo de un modelo del dominio

El modelado del dominio se realiza habitualmente en reuniones organizadas por los analistas del dominio, que utilizan UML y otros lenguajes de modelado para documentar los resultados. Para formar un equipo eficaz, estas reuniones deberían incluir tanto a expertos del dominio como a gente con experiencia en modelado.

El objetivo del modelado del dominio es comprender y describir las clases más importantes dentro del contexto del sistema. Los dominios de tamaño moderado normalmente requieren entre 10 y 50 de esas clases. Los dominios más grandes pueden requerir muchas más.

Los restantes cientos de clases candidatas que los analistas pueden extraer del dominio se guardan como definiciones en un glosario de términos; de otra manera, el modelo del dominio se haría demasiado grande y requeriría más esfuerzo del necesario para esta parte del proceso.

Algunas veces, como en los dominios de negocio muy pequeños, no es necesario desarrollar un modelo de objetos para el dominio; en su lugar, puede ser suficiente un glosario de términos.

El glosario y el modelo del dominio ayudan a los usuarios, clientes, desarrolladores, y otros interesados a utilizar un vocabulario común. La terminología común es necesaria para compartir el conocimiento con los otros. Cuando abunda la confusión, el proceso de ingeniería se hace difícil, si no imposible. Para construir un sistema software de cualquier tamaño, los ingenieros de hoy en día deben "fundir" el lenguaje de todos los participantes en uno solo consistente.

Por último, es necesaria una llamada de atención sobre el modelado del dominio. Puede ser bastante fácil el comenzar modelando las partes internas de un sistema y no su contexto [7]. Por ejemplo, algunos objetos del dominio podrían tener una representación inmediata en el sistema, y algunos analistas del dominio podrían a su vez caer en la trampa de especificar los detalles relativos a esa representación. En casos como éstos, es muy importante recordar que el objetivo del modelado del dominio es contribuir a la comprensión del contexto del sistema, y por lo

tanto también contribuir a la comprensión de los requisitos del sistema que se desprenden de este contexto. En otras palabras, el modelado del dominio debería contribuir a una comprensión del *problema* que se supone que el sistema resuelve en relación a su contexto. El modo interno por el cual el sistema resuelve este problema se tratará en los flujos de trabajo de análisis, diseño, e implementación (*véase* los Capítulos 8, 9 y 10).

6.5.3. Uso del modelo del dominio

Las clases del dominio y el glosario de términos se utilizan en el desarrollo de los modelos de casos de uso y de análisis. Se utilizan:

- Al describir los casos de uso y al diseñar la interfaz de usuario, a lo que volveremos en el Capítulo 7.
- Para sugerir clases internas al sistema en desarrollo durante el análisis, a lo que volveremos en el Capítulo 8.

Sin embargo, existe una forma más sistemática aún de identificar los casos de uso y encontrar las clases dentro del sistema: desarrollar un modelo del negocio. Como veremos, un modelo del dominio es en realidad un caso especial de un modelo del negocio más completo. Por tanto, desarrollar un modelo del negocio es una alternativa más potente que desarrollar un modelo del dominio.

6.6. La comprensión del contexto del sistema mediante un modelo del negocio

El modelado del negocio es una técnica para comprender los procesos de negocio de la organización. Pero, ¿qué pasa si tratamos con un sistema que no tiene nada que ver con lo que la mayoría de nosotros considera un negocio? Por ejemplo, ¿qué deberíamos hacer en el desarrollo de un marcapasos, de un sistema de frenos antibloqueo, de un controlador de cámaras, o de un sistema de telecomunicaciones? En estos casos, también podemos modelar el sistema que rodea el sistema software que vamos a desarrollar. Este sistema (parte del cuerpo humano, parte de un coche, la cámara, el conmutador) es el "sistema de negocio" del sistema software empotrado. Participa en casos de uso de más alto nivel que deberíamos esquematizar brevemente. El objetivo es identificar los casos de uso del software y las entidades de negocio relevantes que el software debe soportar, de forma que podríamos modelar sólo lo necesario para comprender el contexto. El resultado de esta actividad es un modelo del dominio derivado de la comprensión del funcionamiento del "sistema de negocio" que lo rodea.

El modelado del negocio está soportado por dos tipos de modelos de UML: modelos de casos de uso y modelos de objetos [6]. Ambos se definen en la extensión de UML relativa al negocio.

6.6.1. ¿Qué es un modelo del negocio?

En primer lugar, un modelo de casos de uso del negocio describe los procesos de negocio de una empresa en términos de casos de uso del negocio y actores del negocio que se correspon-

den con los procesos del negocio y los clientes, respectivamente. Al igual que el modelo de casos de uso para un sistema sofware, el modelo de casos de uso del negocio presenta un sistema (en este caso, el negocio) desde la perspectiva de su uso, y esquematiza cómo proporciona valor a sus usuarios (en este caso, sus clientes y socios) [3, 4, 6].

Ejemplo **Casos de uso del negocio**

El ejemplo del Consorcio Interbank tiene un caso de uso del negocio que comprende el envío de pedidos, facturas, y pagos entre un comprador y un vendedor —Ventas: desde el Pedido a la Entrega—. En este caso de uso del negocio, un comprador sabe lo que tiene que comprar y a quién. En la siguiente secuencia, Interbank actúa como el agente de negocios en el caso de uso del negocio, conectando al comprador y al vendedor entre sí y proporcionando rutinas seguras para el pago de las facturas:

1. El comprador hace el pedido de bienes o servicios.
2. El vendedor entrega los bienes o servicios.
3. El vendedor envía la factura al comprador.
4. El comprador paga.

En este contexto, el comprador y el vendedor son los actores del negocio de Interbank, y utilizan el caso de uso de negocio que Interbank les proporciona.

Un negocio proporciona normalmente muchos casos de uso de negocio. Interbank no es una excepción. Describiremos dos de estos casos de uso aquí, sólo para situarnos en el contexto adecuado, pero no describiremos el resto de los procesos.

En el caso de uso de negocio Gestión de Préstamo: de la Solicitud al Desembolso, un cliente del banco envía una solicitud de préstamo a Interbank y recibe los fondos del banco.

El cliente del banco representa un cliente genérico para el banco. El comprador y el vendedor son dos tipos más específicos de clientes.

En los casos de uso de negocio Hacer Transferencia, y Sacar e Ingresar Dinero, un cliente del banco realiza transferencias entre cuentas, y retira o ingresa dinero. Este caso de uso de negocio también permitirá a un cliente del banco establecer transferencias automáticas futuras.

El modelo de casos de uso del negocio se describe mediante diagramas de casos de uso (*véase* los Capítulos 4 y 7).

Un modelo de objetos del negocio es un modelo interno a un negocio. Describe cómo cada caso de uso de negocio es llevado a cabo por parte de un conjunto de trabajadores que utilizan un conjunto de entidades del negocio y de unidades de trabajo. Cada realización de un caso de uso del negocio puede mostrarse en diagramas de interacción (*véase* los Capítulos 4 y 9) y diagramas de actividad (como los diagramas de flujo de trabajo en los Capítulos 7 a 11).

Una *entidad* del negocio representa algo, como una factura, que los trabajadores toman, inspeccionan, manipulan, producen o utilizan en un caso de uso del negocio. Una *unidad de trabajo* es un conjunto de esas entidades que conforma un todo reconocible para un usuario final.

Las entidades del negocio y las unidades de trabajo se utilizan para representar los mismos tipos de conceptos que las clases del dominio, como Pedido, Artículo, Factura y Cuenta. Por

tanto podemos confeccionar un diagrama de las entidades del negocio, muy parecido al de la Figura 6.3. También tendremos otros diagramas para mostrar los trabajadores, sus interacciones, y cómo utilizan las entidades de negocio y las unidades de trabajo, como en la Figura 6.4.

Cada trabajador, entidad del negocio, y unidad de trabajo pueden participar en la realización de más de un caso de uso del negocio. Por ejemplo, la clase Cuenta participará probablemente en las tres siguientes realizaciones de caso de uso del negocio:

- En Gestión de Préstamo: de la Solicitud al Desembolso, el dinero que se adquiere por el préstamo se desembolsa en una cuenta.
- En Hacer Transferencia, y Sacar e Ingresar Dinero: el dinero se retira o se ingresa en cuentas, y se transfiere entre cuentas.
- Ventas: del Pedido a la Entrega implica la transferencia de dinero de la cuenta del comprador a la del vendedor.

Ejemplo El caso de uso del negocio ventas: Del pedido a la entrega

Los trabajadores dan los siguientes pasos en el caso de uso del negocio Ventas: del Pedido a la Entrega (*véase* la Figura 6.4):

1. Un comprador solicita bienes o servicios contactando con el vendedor.
2. El vendedor envía una factura al comprador a través del gestor de pagos.
3. El vendedor entrega los bienes o servicios al comprador.
4. El comprador paga mediante el gestor de pagos. Esto implica la transferencia de dinero de la cuenta del comprador a la del vendedor.

El gestor de pagos es un empleado del banco que se encarga de los pasos 2 y 4. Estas tareas pueden automatizarse mediante un sistema de información.

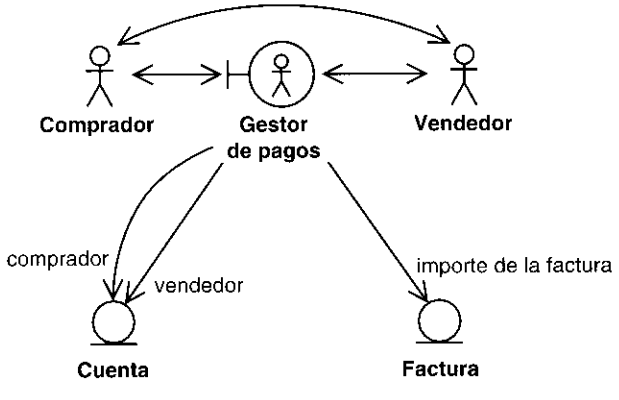


Figura 6.4. El comprador, el vendedor y el gestor de pagos están implicados en el caso de uso del negocio Ventas: del Pedido a la Entrega. El gestor de pagos transfiere el dinero de una cuenta a otra tal como especifica la factura.

El comprador y el vendedor utilizan el gestor de pagos (automatizado) debido a que ese trabajador les proporciona un valor añadido. El trabajador gestor de pagos aporta valor al vendedor enviando la factura al comprador y haciendo el seguimiento de los pagos pendientes. Y aporta valor al comprador simplificando los pagos y ofreciendo una visión general y una disponibilidad mejores del pago de las facturas.

6.6.2. Cómo desarrollar un modelo del negocio

Un modelo del negocio se desarrolla, por tanto, en dos pasos:

1. Los modeladores del negocio deben confeccionar un modelo de casos de uso del negocio que identifique los actores del negocio y los casos de uso del negocio que utilicen los actores. Este modelo de casos de uso del negocio permite a los modeladores comprender mejor qué valor proporciona el negocio a sus actores.

2. Los modeladores deben desarrollar un modelo de objetos del negocio compuesto por trabajadores, entidades del negocio, y unidades de trabajo que juntos realizan los casos de uso del negocio. Se asocian a estos diferentes objetos las reglas del negocio y otras normas impuestas por el negocio. El objetivo es crear trabajadores, entidades del negocio, y unidades de trabajo que realicen los casos de uso del negocio de la manera más eficaz posible —es decir, rápidamente, con precisión, y con un coste bajo.

El modelado del negocio y el modelado del dominio se parecen en muchos aspectos. De hecho, podemos pensar en el modelado del dominio como en una variante simplificada del modelado del negocio, en la cual nos centramos sólo en las "cosas", es decir, las clases del dominio o entidades del negocio que necesitan usar los trabajadores[1]. Por tanto las clases del dominio y las entidades del negocio son conceptos muy parecidos, y utilizamos ambos términos indistintamente.

Sin embargo, existen algunas diferencias importantes entre el modelado del negocio y el modelado del dominio que hacen mucho más recomendable el realizar el más completo modelado del negocio:

- Las clases del dominio se obtienen de la base del conocimiento de unos pocos expertos del dominio, o posiblemente del conocimiento (otras clases del dominio, especificaciones de requisitos, etc.) asociado con sistemas similares al que estamos desarrollando. Las entidades del negocio, por otro lado, se derivan a partir de los clientes del negocio, identificando los casos de uso del negocio, y después buscando las entidades. En la técnica de modelado del negocio, cada entidad debe venir motivada por su utilización en un caso de uso del negocio. Estas dos técnicas normalmente acaban con conjuntos diferentes de clases, asociaciones, atributos y operaciones. La técnica de modelado del dominio puede hacer la traza de las clases hasta la experiencia de los expertos del dominio. La técnica de modelado del negocio puede hacer la traza de la necesidad de cada elemento del modelo hasta los clientes.

- Las clases del dominio tienen atributos pero normalmente ninguna o muy pocas operaciones. No es así para las entidades del negocio. La técnica de modelado del negocio no

[1] Debido a que un modelo del dominio es una variante simplificada de un modelo del negocio, sólo mencionamos este último como entrada en los siguientes flujos de trabajo que se tratan en los Capítulos del 7 al 11.

sólo identifica las entidades sino también todos los trabajadores que participarán en la realización de los casos de uso del negocio que utilizan a las entidades. Además, se identifica cómo utilizarán esos trabajadores las entidades a través de operaciones que debe ofrecer cada entidad. Al igual que para las propias entidades, estas operaciones también se derivarán de los clientes y podrán ser trazadas hasta ellos.

- Los trabajadores identificados en el modelado del negocio se utilizan como punto de partida para derivar un primer conjunto de actores y casos de uso para el sistema de información en construcción. Esto nos permite hacer la traza de cada caso de uso del sistema de información, a través de los trabajadores y los casos de uso del negocio, hasta los clientes del negocio. Esto se tratará en profundidad en el Capítulo 7. Además, puede hacerse la traza de cada caso de uso hasta los componentes que implementan el sistema, como se describió en el Capítulo 3. Por tanto podemos concluir que el modelado del negocio y la técnica de ingeniería del software combinados en el Proceso Unificado nos permite hacer el seguimiento de las necesidades del cliente a lo largo del camino completo a través de procesos del negocio, trabajadores y casos de uso, hasta el código del software. Sin embargo, cuando se utiliza solamente un modelo del dominio, no hay una forma evidente de hacer la traza entre el modelo del dominio y los casos de uso del sistema.

Uso de la técnica de modelado del negocio para la descripción del proceso unificado (Primera Parte)

La técnica del modelado del negocio que presentamos para modelar la empresa del cliente es básicamente la misma técnica que hemos utilizado al describir el Proceso Unificado para la ingeniería de software de este libro. Por tanto hemos utilizado la extensión específica del negocio de UML al describir el Proceso Unificado para la ingeniería de software (*véase* el Capítulo 2). Aunque posee unas bases teóricas sólidas, es también muy práctica. Es un tipo de bootstrapping o de trabajo reflexivo. Revela las fortalezas y debilidades de la técnica.

Por tanto, el Proceso Unificado es un caso de uso del negocio del desarrollo de software. Dentro del negocio del desarrollo de software, el proceso se organiza, o como decimos "se realiza", como una secuencia de flujos de trabajo entrelazados: requisitos (como se explica en este capítulo y en el Capítulo 7), análisis (Capítulo 8), diseño (Capítulo 9), implementación (Capítulo 10), y prueba (Capítulo 11). Cada flujo de trabajo es una realización de parte del caso de uso del negocio del Proceso Unificado, que se describe en términos de:

- Trabajadores, como el analista de sistemas y los especificadores de casos de uso.
- Entidades del negocio, o como nosotros les llamamos, artefactos, como casos de uso y casos de prueba.
- Unidades de trabajo —que también son artefactos— como el modelo de casos de uso y la descripción de la arquitectura.

Los trabajadores, las entidades del negocio, y las unidades de trabajo del Proceso Unificado también se muestran en diagramas de clase UML junto con las relaciones más importantes entre ellas.

(Esta sección resaltada continuará en el Capítulo 7).

6.6.3. Búsqueda de casos de uso a partir de un modelo del negocio

Mediante la utilización de un modelo del negocio como entrada, un analista emplea una técnica sistemática para crear un modelo de casos de uso tentativo.

En primer lugar, el analista identifica un *actor*[2] por cada trabajador y por cada actor del negocio (es decir, cada cliente) que se convertirá en usuario del sistema de información.

Ejemplo **El actor comprador**

El comprador utiliza el Sistema de Facturación y Pagos para solicitar bienes o servicios y para pagar facturas. Por tanto el comprador es tanto un cliente como un actor ya que utiliza el sistema para pedir y pagar mediante dos casos de uso: Solicitar Bienes o Servicios y Pagar Facturas.

Cada trabajador (y cada actor del negocio) que vaya a ser usuario del sistema de información requerirá un soporte por parte del mismo. El soporte necesario se determina tratando cada uno de los trabajadores, uno detrás de otro. Para cada trabajador, identificamos todas las realizaciones de casos de uso del negocio diferentes en las que participa. El trabajador cumple un papel en cada una, de forma muy parecida al papel que desempeña una clase en cada realización de caso de uso.

Una vez que hemos encontrado todos los roles de un trabajador o de un actor del negocio, uno por cada caso de uso del negocio en el que participa, podemos encontrar los casos de uso de los actores del sistema en el sistema de información. Cada trabajador y cada actor del negocio se corresponde con un actor del sistema de información. Por cada papel de un trabajador o un actor del negocio, necesitamos un caso de uso para el actor del sistema correspondiente.

Por tanto la manera más directa de identificar un conjunto tentativo de casos de uso es crear un caso de uso para el actor correspondiente a cada rol de cada trabajador y de cada actor del negocio. Como resultado, obtendremos para cada caso de uso del negocio un caso de uso por cada trabajador y por cada actor del sistema. Los analistas pueden detallar y ajustar después estos casos de uso tentativos.

Los analistas deben también decidir cuántas de las tareas de los trabajadores o de los actores del sistema deberían automatizarse mediante sistemas de información (en la forma de casos de uso) y reorganizar los casos de uso para que se ajusten mejor a las necesidades de los actores. Obsérvese que puede que no todas las tareas sean apropiadas para ser automatizadas.

Ejemplo **Identificación de casos de uso a partir de un modelo del negocio**

Prosiguiendo con el ejemplo precedente, podríamos proponer un caso de uso tentativo llamado Compra de Bienes o Servicios, que podría dar soporte al actor comprador en su papel de actor del ne-

[2] Utilizaremos el término *actor* para referirnos a un *actor* del sistema cuando no haya riesgo de confusión con los actores del negocio.

gocio durante el caso de uso del negocio Ventas: del Pedido a la Entrega. Después de un análisis más profundo, parece claro que Compra de Bienes o Servicios se realizaría de mejor forma como varios casos de uso diferentes, tales como Solicitar Bienes o Servicios y Pagar Factura. La razón de descomponer el caso de uso tentativo en varios casos de uso más pequeños es que el comprador no desea llevar a cabo el caso de uso Compra de Bienes o Servicios en una secuencia ininterrumpida de acciones. En cambio, el comprador quiere esperar a la llegada de los bienes o servicios antes de pagar la factura. La secuencia de pago se representa por tanto mediante un caso de uso Pagar Factura aparte, que se lleva a cabo cuando se han entregado los bienes.

Hasta este momento hemos visto cómo podemos modelar el contexto del sistema mediante un modelo del dominio o del negocio. Después vimos cómo podía derivarse un modelo de casos de uso a partir de un modelo del negocio, es decir, como un modelo de casos de uso que captura todos los requisitos funcionales de un sistema así, como la mayoría de los requisitos no funcionales. Algunos requisitos no pueden asociarse a ningún caso de uso en concreto y se conocen como requisitos adicionales.

6.7. Requisitos adicionales

Los requisitos adicionales son fundamentalmente requisitos no funcionales que no pueden asociarse a ningún caso de uso en concreto —en cambio cada uno de estos requisitos tiene impacto en varios casos de uso o en ninguno—. Algunos ejemplos son el rendimiento, las interfaces y los requisitos de diseño físico, así como las restricciones arquitectónicas, de diseño y de implementación [1]. Los requisitos adicionales se capturan de forma muy parecida a como se hacía en la especificación de requisitos tradicional, es decir, como una lista de requisitos. Después se utilizan durante el análisis y el diseño junto al modelo de casos de uso.

Un *requisito de interfaz* especifica la interfaz con un elemento externo con el cual debe interactuar el sistema, o que establece restricciones condicionantes en formatos, tiempos, u otros factores de relevancia en esa interacción.

Un *requisito físico* especifica una característica física que debe poseer un sistema, como su material, forma, tamaño o peso. Por ejemplo, puede utilizarse este tipo de requisito para representar requisistos hardware como las configuraciones físicas de red requeridas.

Ejemplo **Requisitos de plataforma hardware**

Servidores

SUN SPARC 20 o PC Pentium.

Clientes

PC (procesador mínimo Intel 486) o Sun Sparc 5.

Una *restricción de diseño* limita el diseño de un sistema, como lo hacen las restricciones de extensibilidad y mantenibilidad, o las restricciones relativas a la reutilización de sistemas heredados o partes esenciales de los mismos.

Una *restricción de implementación* especifica o limita la codificación o construcción de un sistema. Son ejemplos los estándares requeridos, las normas de codificación, los lenguajes de programación, políticas para la integridad de la base de datos, limitaciones de recurso, y entornos operativos.

Ejemplo **Restricciones en los formatos de fichero**

La Versión 1.2 del Sistema de Facturación y Pagos debe soportar nombres largos de fichero.

Ejemplo **Restricciones en la plataforma software**

Software del Sistema

Sistemas operativos de cliente: Windows NT 4.0, Windows 95, o Solaris 2.6.

Sistemas operativos de servidor: Windows NT 4.0 o Solaris 2.6.

Software para Internet

Netscape Communicator 4.0 o Microsoft Internet Explorer 4.0.

Además, existen *otros requisitos*, como los requisitos legales y las normativas.

Ejemplo **Otros requisitos**

Seguridad

La transmisión debe ser segura, entendiendo por esto que sólo las personas autorizadas pueden tener acceso a la información. Las únicas personas autorizadas son el cliente del banco que posee las cuentas y los actores administradores del sistema.

Disponibilidad

El Sistema de Facturación y Pagos no debe tener más de 1 hora por mes de tiempo de no disponibilidad.

Facilidad de Aprendizaje

El tiempo de aprendizaje (mediante instrucciones paso a paso proporcionadas) para enviar pedidos simples y para pagar facturas simples no debe exceder de 10 minutos para el 90 por ciento de los compradores. Un pedido simple es un pedido con un solo artículo. Una factura simple es una factura para el pago de un pedido simple.

6.8. Resumen

Hasta aquí, hemos ofrecido unas buenas bases de la captura de requisitos. Hemos visto como los modelos del negocio y del dominio nos ayudan a definir el contexto del sistema y cómo pueden derivarse los casos de uso a partir de un modelo del negocio. Hemos visto que los casos de uso se utilizan para capturar los requisitos, y volveremos a este tema en el siguiente capítulo. En capítulos sucesivos veremos cómo los casos de uso y los requisitos adicionales nos ayudan a analizar, crear la arquitectura, diseñar, implementar y probar el sistema para asegurar que cumple con los requisitos del cliente.

6.9. Referencias

[1] IEEE Std 610.12.1990.

[2] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard, *Object-Oriented Software Engineering: A Use-Case-Driven Approach*, Reading, MA: Addison-Wesley, 1992 (Revised fourth printing, 1993).

[3] Ivar Jacobson, Maria Ericsson, and Agneta Jacobson, *The Object Advantage: Business Process Reengineering with Object Technology*, Reading, MA: Addison-Wesley, 1994.

[4] Ivar Jacobson, "Business process reengineering with object technology", *Object Magazine*, May 1994.

[5] James Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*, Englewood Cliffs, NJ: Prentice Hall, 1991.

[6] OMG Unified Modeling Language Specification. Object Management Group, Framingham, MA, 1998. Internet: www.omg.org.

[7] Alan M. Davis, *Software Requirements: Objects, Functions, and States*, Englewood Cliffs, NJ: Prentice Hall, 1993.

Capítulo 7

Captura de requisitos como casos de uso

7.1. Introducción

El esfuerzo principal en la fase de requisitos es desarrollar un modelo del sistema que se va a construir, y la utilización de los casos de uso es una forma adecuada de crear ese modelo. Esto es debido a que los requisitos funcionales se estructuran de forma natural mediante casos de uso, y a que la mayoría de los otros requisitos no funcionales son específicos de un solo caso de uso, y pueden tratarse en el contexto de ese caso de uso.

Los requisitos no funcionales restantes, aquellos que son comunes para muchos o para todos los casos de uso, se mantienen en un documento aparte y se denominan *requisitos adicionales*. Ya tratamos estos requisitos en el Capítulo 6 y no volveremos a ellos hasta que lleguemos a su utilización en los flujos de trabajo de análisis, diseño, implementación y prueba.

Los casos de uso proporcionan un medio intuitivo y sistemático para capturar los requisitos funcionales con un énfasis especial en el valor añadido para cada usuario individual o para cada sistema externo. Mediante la utilización de los casos de uso, los analistas se ven obligados a pensar en términos de quiénes son los usuarios y qué necesidades u objetivos de la empresa pueden cumplir. Sin embargo, como dijimos en el Capítulo 4, los casos de uso no habrían tenido la amplia aceptación de que gozan si eso fuese todo lo que hacen. Su papel clave en la dirección del resto del trabajo de desarrollo ha sido un motivo importante para su aceptación en la mayoría de los métodos de la ingeniería moderna del software [8].

En este capítulo detallaremos nuestra comprensión de los casos de uso y los actores, y presentaremos definiciones más precisas que las empleadas en el Capítulo 3.

Vamos a describir el flujo de trabajo de los requisitos en tres pasos (y describiremos de igual manera todos los flujos de trabajo en los Capítulos 8 al 11):

- Los artefactos creados en el flujo de trabajo de los requisitos.
- Los trabajadores participantes en el flujo de trabajo de los requisitos.
- El flujo de trabajo de captura de requisitos, incluyendo cada actividad en más detalle.

Para comenzar, examinaremos los trabajadores y artefactos que se muestran en la Figura 7.1.

Uso de la técnica de modelado del negocio para la descripción del proceso unificado (Segunda Parte)

Identificamos los trabajadores y los artefactos que participan en cada flujo de trabajo. Un *trabajador* representa un puesto que puede ser asignado a una persona o a un grupo, y especifica las responsabilidades y habilidades requeridas (*véase* también la Sección 2.1.3).

Artefacto es un término general para cualquier tipo de descripción o información creada, producida, cambiada o utilizada por los trabajadores durante su trabajo con el sistema. Un artefacto puede ser un modelo, un elemento de un modelo, o un documento. Por ejemplo, en el flujo de trabajo de los requisitos, los artefactos son fundamentalmente el modelo de casos de uso y sus casos de uso. Obsérvese que en el modelo del negocio del Proceso Unificado, un artefacto es una entidad del negocio o una unidad de trabajo.

Cada trabajador es responsable de un conjunto de artefactos. Esto se muestra en los diagramas mediante una asociación etiquetada como "responsable de" desde el trabajador hacia los correspondientes artefactos (por ejemplo, *véase* la Figura 7.1). Para hacer más intuitivos esos diagramas, utilizamos símbolos especiales para la mayoría de los artefactos. Los artefactos que representan documentos se muestran con un símbolo especial de documento. Los artefactos que representan modelos o elementos de un modelo se muestran con su correspondiente símbolo UML.

Obsérvese que para poder utilizar esos símbolos especiales en el modelo del negocio del Proceso Unificado, hemos incluido estereotipos para documentos, modelos y elementos de modelo. Cada uno de estos estereotipos es un subtipo de los estereotipos "business entity" o "work unit".

Mostraremos cómo colaboran los trabajadores en un *flujo de trabajo*, y así veremos cómo pasa la atención del trabajo de trabajador a trabajador, y cómo estos trabajan con artefactos a medida que realizan sus actividades (*véase* también la Sección 2.4.2). En este contexto también observamos cada actividad en más detalle. Una *actividad* es un fragmento de trabajo realizado por un trabajador en el flujo de trabajo, es decir, es la ejecución de una de las operaciones de los trabajadores.

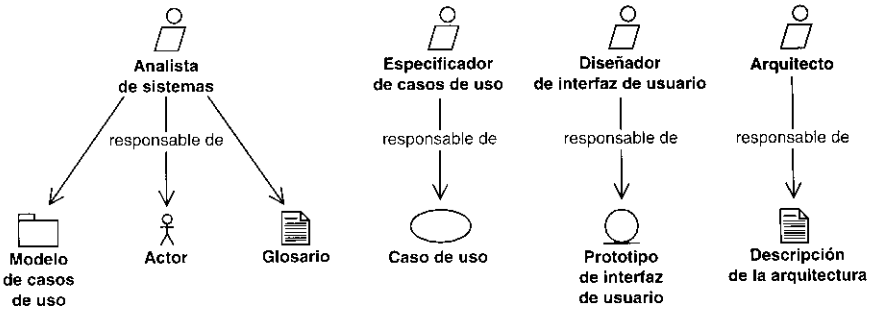


Figura 7.1. Los trabajadores y artefactos implicados durante la captura de los requisitos mediante casos de uso.

7.2. Artefactos

Los artefactos fundamentales que se utilizan en la captura de requisitos son el modelo de casos de uso, que incluye los casos de uso y los actores. También puede haber otros tipos de artefactos, como prototipos de interfaz de usuario.

Estos artefactos se presentaron en el Capítulo 3, pero ahora les dotaremos de definiciones más precisas, consistentes con las que se ofrecen en [12]. Después, suavizaremos este formalismo y mostraremos cómo aplicar estas construcciones en la práctica del Proceso Unificado. Ofrecemos aquí las definiciones para proporcionar unos fundamentos sólidos, pero no es necesario aplicar el formalismo en la práctica. Las hemos incluido por las siguientes razones:

- Podrían ser importantes en la descripción formal de algunos casos de uso, como cuando utilizamos diagramas de actividad o diagramas de estados. Esto es particularmente útil para casos de uso con muchos estados y con transiciones complejas entre ellos.
- Facilita la identificación de los casos de uso correctos y su descripción consistente. En realidad, incluso si decidimos no utilizar el formalismo disponible en la descripción de, por ejemplo, los actores o los casos de uso, es bueno tenerlo en mente para que nos ayude a ser completos y consistentes.
- Es importante para poder explicar los actores y casos de uso en relación con otras construcciones de UML.

7.2.1. Artefacto: modelo de casos de uso

El modelo de casos de uso permite que los desarrolladores de software y los clientes lleguen a un acuerdo sobre los requisitos [6], es decir, sobre las condiciones y posibilidades que debe cumplir el sistema. El modelo de casos de uso sirve como acuerdo entre clientes y desarrolladores, y proporciona la entrada fundamental para el análisis, el diseño y las pruebas.

Un modelo de casos de uso es un modelo del sistema que contiene actores, casos de uso y sus relaciones (*véase* la Figura 7.2).

El modelo de casos de uso puede hacerse bastante grande y difícil de digerir de un solo mordisco, de forma que es necesario algún medio de abordarlo en trozos más pequeños. UML nos permite presentar el modelo en diagramas que muestran los actores y los casos de uso desde

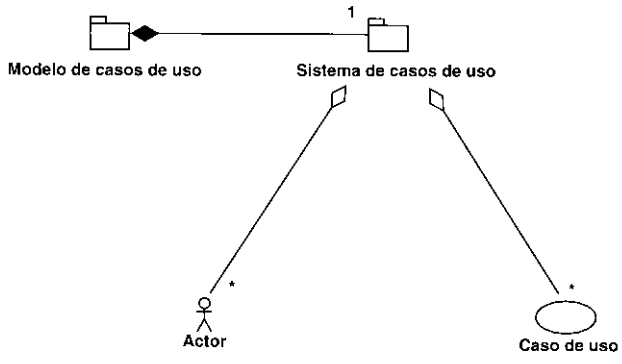


Figura 7.2. El modelo de casos de uso y sus contenidos. El sistema de casos de uso representa el paquete de más alto nivel del modelo.

diferentes puntos de vista y con diferentes propósitos. Obsérvese también que si el modelo de casos de uso es grande, es decir, si contiene un gran número de casos de uso y/o actores, también puede ser útil introducir paquetes en el modelo para tratar su tamaño. Esto es una extensión más o menos trivial del modelo de casos de uso, y no la trataremos en este libro.

7.2.2. Artefacto: actor

El modelo de casos de uso describe lo que hace el sistema para cada tipo de usuario. Cada uno de éstos se representa mediante uno o más actores. También se representa mediante uno o más actores cada sistema externo con el que interactúa el sistema, incluyendo dispositivos externos como temporizadores, que se consideran externos al sistema. Por tanto, los actores representan terceros fuera del sistema que colaboran con el sistema. Una vez que hemos identificado todos los actores del sistema, tenemos identificado el entorno externo al sistema.

Los actores suelen corresponderse con trabajadores (o actores del negocio) en un negocio, como se trató en el Capítulo 6. Recuérdese que cada rol (de un trabajador) define lo que hace el trabajador en un proceso de negocio concreto. Los roles que desempeña un trabajador pueden emplearse para obtener (o para generar realmente si contamos con las herramientas apropiadas) los roles que cumple el actor del sistema correspondiente. Después, como se indicó en el Capítulo 6, dotamos a cada trabajador con un caso de uso del sistema para cada uno de sus roles. Ese caso de uso proporciona un valor al actor cuando representa el papel del trabajador.

Ejemplo **Actor**

El Sistema de Pagos y Facturación interactúa con un tipo de usuario que empleará el sistema para pedir bienes, confirmar pedidos, pagar facturas, y demás. Este tipo de usuario se representa mediante el actor Comprador (Figura 7.3).

Figura 7.3. El actor Comprador.

Un actor juega un papel por cada caso de uso con el que colabora. Cada vez que un usuario en concreto (un humano u otro sistema) interactúa con el sistema, la instancia correspondiente del actor está desarrollando ese papel. Una instancia de un actor es por tanto un usuario concreto que interactúa con el sistema. Cualquier entidad que se ajuste a un actor puede actuar como una instancia del actor.

7.2.3. Caso de uso

Cada forma en que los actores usan el sistema se representa con un caso de uso. Los casos de uso son "fragmentos" de funcionalidad que el sistema ofrece para aportar un resultado de valor para sus actores. De manera más precisa, un caso de uso especifica una secuencia de acciones que el sistema puede llevar a cabo interactuando con sus actores, incluyendo alternativas dentro de la secuencia.

Por tanto, un *caso de uso* es una especificación. Especifica el comportamiento de "cosas" dinámicas, en este caso, de instancias de los casos de uso.

Ejemplo **El caso de uso Sacar Dinero**

En el Capítulo 3 describimos el caso de uso Sacar Dinero, que permite a las instancias del actor Cliente de Banco el sacar dinero mediante un CA (Figura 7.4).

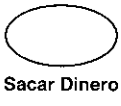


Figura 7.4. El caso de uso Sacar Dinero.

El caso de uso Sacar Dinero especifica las posibles instancias de ese caso de uso, es decir, las diferentes formas válidas de llevar a cabo el caso de uso por parte del sistema y la interacción requerida con las instancias de actores implicadas. Supongamos que una persona concreta, de nombre Jack, introduce en primer lugar su contraseña 1234, selecciona sacar 220 dólares, y toma el dinero. El sistema habrá llevado a cabo una instancia del caso de uso. Si en cambio Jack introduce su contraseña, elige sacar 240 dólares, y después toma el dinero, el sistema habrá llevado a cabo otra instancia del caso de uso. Una tercera instancia del caso de uso podría ser lo que haría el sistema si Jack solicita sacar 480 dólares, y el sistema no se lo permite debido a un saldo insuficiente o a una contraseña errónea, y casos similares.

Según el vocabulario de UML, un *caso de uso* es un clasificador, lo cual quiere decir que tiene operaciones y atributos. Una descripción de un caso de uso puede por tanto incluir diagramas de estados, diagramas de actividad, colaboraciones, y diagramas de secuencia (para consultar detalles, *véase* el Apéndice A, "Visión general de UML").

Los diagramas de estados especifican el ciclo de vida de las instancias de los casos de uso en términos de estados y transiciones entre los estados. Cada transición es una secuencia de

acciones. Los diagramas de actividad describen el ciclo de vida con más detalle describiendo también la secuencia temporal de acciones que tiene lugar dentro de cada transición. Los diagramas de colaboración y los de secuencia se emplean para describir las interacciones entre, por ejemplo, una instancia típica de un actor y una instancia típica de un caso de uso. En la práctica, no siempre es necesario ser tan formal en la descripción de casos de uso, como trataremos en la Sección 7.4.3, "Detallar un caso de uso". Sin embargo, el tener en mente esta comprensión más precisa de los casos de uso nos ayuda a estructurar las descripciones de los mismos.

Una *instancia de caso de uso* es la realización (o ejecución) de un caso de uso. Otra forma de decirlo es que una instancia de un caso de uso es lo que el sistema lleva a cabo cuando "obedece a un caso de uso". Cuando se lleva a cabo una instancia de un caso de uso, ésta interactúa con instancias de actores, y ejecuta una secuencia de acciones según se especifica en el caso de uso. Esta secuencia se especifica en un diagrama de estados o un diagrama de actividad; es un camino a lo largo del caso de uso. Puede haber muchos caminos, y muchos de ellos pueden ser muy parecidos. Éstas son alternativas de la secuencia de acciones para el caso de uso. Un camino como ése a través de un caso de uso puede ser algo parecido a lo siguiente:

1. La instancia del caso de uso se inicia y pasa a estado de comienzo.
2. El caso de uso es invocado por un mensaje externo de un actor.
3. Transita a otro estado realizando una secuencia de acciones. Una secuencia de este tipo contiene cálculos internos, selección del camino, y mensajes de salida (hacia algún actor).
4. Queda a la espera (en el nuevo estado) de otro mensaje externo de un actor.
5. Es invocado (otra vez) por un nuevo mensaje, etc. Esto puede continuar sobre muchos estados hasta que se termina la instancia del caso de uso.

La mayoría de las veces es una instancia de un actor la que invoca a la instancia del caso de uso, como se ha descrito, pero también puede ser un evento interno al sistema el que invoque a la instancia, como cuando un temporizador programado se dispara (siempre que el temporizador se considere interno al sistema).

Los casos de uso, como todos los clasificadores, tienen atributos. Estos atributos representan los valores que una instancia de un caso de uso utiliza y manipula durante la ejecución de su caso de uso. Estos valores son locales a la instancia del caso de uso, es decir, no pueden ser utilizados por otras instancias del caso de uso. Por ejemplo, puede considerarse que el caso de uso Sacar Dinero posee atributos como la contraseña, la cuenta, y la cantidad a retirar.

Las instancias de los casos de uso no interactúan con otras instancias de casos de uso. El único tipo de interacciones en el modelo de casos de uso tiene lugar entre instancias de actores e instancias de casos de uso [10]. El motivo es que queremos que el modelo de casos de uso sea simple e intuitivo para permitir discusiones fructíferas con los usuarios finales y otros interesados, sin quedar atrapados en los detalles. No queremos tratar con interfaces entre casos de uso, concurrencia y otros conflictos (como la compartición de otros objetos) entre instancias de casos de uso distintas. Consideramos atómicas las instancias de casos de uso, es decir, cada una de ellas se ejecuta por completo o no se ejecuta nada, sin interferencias por parte de otras instancias de casos de uso. Por tanto, el comportamiento de cada caso de uso puede interpretarse independientemente del de los otros casos de uso, lo cual hace más sencillo el modelado de casos de uso. El considerar los casos de uso como atómicos no tiene nada que ver con que haya un gestor de transacciones subyacente que se encargue de los conflictos. Lo hacemos solamente para garantizar que podemos leer y comprender el modelo de casos de uso.

Sin embargo, reconocemos que ciertamente existen temas de interferencia entre los diferentes usos de un sistema. Estos temas no pueden resolverse en el modelado de casos de uso sino que se posponen al análisis y al diseño (descritos respectivamente en los Capítulos 8 y 9), en los cuales realizamos los casos de uso como colaboraciones entre clases y/o subsistemas. En el modelo de análisis, podemos, por ejemplo, describir claramente cómo una clase puede participar en varias realizaciones de casos de uso y cómo puede resolverse cualquier tema de interferencia implícita entre casos de uso.

Modelado de grandes sistemas

En este libro nos centramos en cómo modelar un sistema independiente. Sin embargo, en muchos casos es necesario desarrollar sistemas más grandes, sistemas que realmente se componen de otros sistemas. Los llamaremos sistemas de sistemas.

Ejemplo **Uno de los sistemas más grandes del mundo**

El sistema más grande jamás construido por seres humanos podría ser un sistema con cerca de un billón de usuarios, a saber, la red de telecomunicaciones global.

Cuando hacemos una llamada telefónica, por ejemplo, de San Francisco a Estocolmo, la llamada pasará probablemente a través de unos 20 sistemas, incluyendo nodos de conmutación locales, internacionales, sistemas por satélite, sistemas de transmisión y demás. Cada uno de estos sistemas tuvo un coste de desarrollo aproximado de 1.000 personas-año, y el esfuerzo de desarrollo de software constituyó un alto porcentaje de esos costes.

Es sorprendente que cuando hacemos esas llamadas, ¡normalmente funciona!

Dada la complejidad y todas las diferentes personas, empresas y naciones implicadas, ¿por qué funciona? La razón fundamental es que cada interfaz de la red entera (es decir, la arquitectura de la red) ha sido estandarizada por una misma organización, la UIT (la Unión Internacional de Telecomunicaciones).

La UIT ha especificado las interfaces entre todos los tipos de nodos en la red y la semántica precisa de todos ellos.

La construcción de sistemas de sistemas se basa en técnicas similares a las utilizadas para construir la red global de telecomunicaciones [9]. Primero se especifica el sistema entero con sus casos de uso. Se diseña en términos de subsistemas que colaboran. Los casos de uso del sistema en su globalidad se dividen en casos de uso de los subsistemas que colaboran, y los subsistemas se interconectan mediante interfaces. Estas interfaces se definen de manera precisa, después de lo cual cada subsistema por separado puede desarrollarse independientemente (como un sistema en sí mismo) por una empresa diferente. UML soporta este tipo de arquitectura, y el Proceso Unificado puede ampliarse para desarrollar este tipo de sistemas [13].

En realidad, los casos de uso puede utilizarse no sólo para especificar sistemas, sino también otras entidades más pequeñas, como subsistemas o clases. Por tanto, un subsistema o una clase puede tener dos partes, cada una de las cuales describe una perspectiva: una especificación y una implementación. La especificación describe lo que el subsistema o la clase proporciona a su entorno en términos de casos de uso. La parte de implementación describe

(continúa)

cómo se estructura internamente el subsistema o la clase para llevar a cabo su especificación. Este entorno se compone normalmente de otros subsistemas o clases. Sin embargo, si queremos tratar el entorno de forma anónima, podemos representarlo también mediante actores.

Este enfoque se emplea cuando queremos tratar un subsistema como si fuese un sistema de pleno derecho, como por ejemplo:

- Cuando queremos desarrollar el subsistema utilizando una tecnología diferente de la que utilizamos para otros subsistemas. Podemos hacerlo siempre que proporcione los usos y casos de uso adecuados y siempre que soporte las interfaces especificados.
- Cuando queremos gestionar el sistema de forma separada de los otros —quizá en ubicaciones geográficas distintas.

7.2.3.1. Flujo de sucesos El flujo de sucesos para cada caso de uso puede plasmarse como una descripción textual de la secuencia de acciones del caso de uso. Por tanto, el flujo de sucesos especifica lo que el sistema hace cuando se lleva a cabo el caso de uso especificado. El flujo de sucesos también especifica cómo interactúa el sistema con los actores cuando se lleva a cabo el caso de uso.

Desde la perspectiva de la gestión, una descripción de un flujo de sucesos incluye un conjunto de secuencias de acciones que pueden ser modificadas, revisadas, diseñadas, implementadas y probadas juntas, y que pueden ser descritas como una sección o subsección del manual de usuario.

Ofrecemos ejemplos de descripción de un flujo de sucesos de un caso de uso en la Sección 7.4.3, "Detallar un caso de uso".

7.2.3.2. Requisitos especiales Llamamos requisitos especiales a una descripción textual que agrupa todos los requisitos del tipo de los requisitos no funcionales sobre el caso de uso. Son fundamentalmente requisitos no funcionales relacionados con el caso de uso y que deben tratarse en flujos de trabajo posteriores como análisis, diseño o implementación.

Ofrecemos ejemplos de requisitos especiales relacionados con un caso de uso en la Sección 7.4.3, "Detallar un caso de uso".

7.2.4. Artefacto: descripción de la arquitectura (vista del modelo de casos de uso)

La descripción de la arquitectura contiene una vista de la arquitectura del modelo de casos de uso, que representa los casos de uso significativos para la arquitectura (Figura 7.5).

La vista de la arquitectura del modelo de casos de uso debería incluir los casos de uso que describan alguna funcionalidad importante y crítica, o que impliquen algún requisito importante que deba desarrollarse pronto dentro del ciclo de vida del software. Esta vista de la arquitectura se utiliza como entrada cuando se priorizan los casos de uso para su desarrollo (análisis, diseño, implementación) dentro de cada iteración. Este tema se trata con más detalle dentro de las Partes I y III (*véase* el Capítulo 4, Sección 4.3, y el Capítulo 12, Sección 12.6, respectivamente).

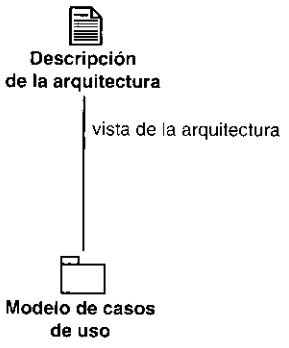


Figura 7.5. La descripción de la arquitectura.

Normalmente, las realizaciones de caso de uso correspondiente pueden encontrarse en las vistas arquitectónicas de los modelos de análisis y de diseño (*véase* el Capítulo 4, Sección 4.5, el Capítulo 8, Sección 8.4.5, y el Capítulo 9, Sección 9.3.6).

7.2.5. Artefacto: glosario

Podemos utilizar un glosario para definir términos comunes importantes que los analistas (y otros desarrolladores) utilizan al describir el sistema. Un glosario es muy útil para alcanzar un consenso entre los desarrolladores relativo a la definición de los diversos conceptos y nociones, y para reducir en general el riesgo de confusiones.

Habitualmente podemos obtener un glosario a partir de un modelo del negocio o de un modelo del dominio, pero debido a que es menos formal (no incluye clases o relaciones explícitas), es más fácil de mantener y es más intuitivo para utilizarlo con terceras personas externas, como usuarios y clientes. Además, un glosario tiende a estar más centrado en el sistema que se va a construir, en lugar de en su contexto, como es el caso de los modelos del negocio o del dominio.

7.2.6. Artefacto: prototipo de interfaz de usuario

Los prototipos de interfaz de usuario nos ayudan a comprender y especificar las interacciones entre actores humanos y el sistema durante la captura de requisitos. No sólo nos ayuda a desarrollar una interfaz gráfica mejor, sino también a comprender mejor los casos de uso. A la hora de especificar la interfaz de usuario también pueden utilizarse otros artefactos, como los modelos de interfaz gráfica y los esquemas de pantallas.

Véase también [2, 4, 6, 10, 11, 12] para obtener más detalles sobre los actores y los casos de uso y [14] para encontrar información sobre el diseño de la interfaz de usuario.

7.3. Trabajadores

Al comienzo de este capítulo, explicamos los artefactos que se producen durante el modelado de casos de uso. El paso siguiente es examinar los trabajadores responsables de esos artefactos.

Como dijimos en el Capítulo 2, un trabajador es un puesto al cual se puede asignar una persona "real". Con cada trabajador tenemos una descripción de las responsabilidades y el comportamiento esperado del mismo. Un trabajador no es lo mismo que un individuo; una misma persona puede estar asignada a diferentes trabajadores durante un proyecto. Un trabajador tampoco se corresponde con un puesto o cargo concreto en una empresa —ése es un tema diferente—. En cambio, podemos decir que un trabajador representa una abstracción de un ser humano con ciertas capacidades que se requieren en un caso de uso del negocio, en nuestro caso, en el Proceso Unificado para el desarrollo de software. Cuando se asignan los recursos humanos a un proyecto, un trabajador representa el conocimiento y las habilidades que alguien necesita para hacerse cargo del trabajo como trabajador del proyecto. Podemos identificar tres trabajadores que participan en el modelado de casos de uso, cada uno con su propio conjunto de operaciones y con diferentes responsabilidades requeridas: analista de sistemas, especificador de casos de uso, y diseñador de interfaz de usuario. En este libro, llamaremos analistas a todos esos trabajadores. También hay otros trabajadores, como los revisores, pero no los consideraremos en este libro.

7.3.1. Trabajador: analista de sistemas

El analista del sistema es el responsable del conjunto de requisitos que están modelados en los casos de uso, lo que incluye todos los requisitos funcionales y no funcionales que son casos de uso específicos. El analista de sistemas es responsable de delimitar el sistema, encontrando los actores y los casos de uso y asegurando que el modelo de casos de uso es completo y consistente. Para la consistencia, el analista de sistemas puede utilizar un glosario para conseguir un acuerdo en los términos comunes, nociones y conceptos durante la captura de los requisitos. Las responsabilidades del analista de sistemas se muestran en la Figura 7.6.

Aunque el analista de sistemas es responsable del modelo de casos de uso y de los actores que contiene, no es responsable de cada caso de uso en particular. Esto es una responsabilidad aparte, que pertenece al trabajador especificador de casos de uso (Sección 7.3.2). El analista de sistemas es también el que dirige el modelado y el que coordina la captura de requisitos.

Hay un analista de sistemas para cada sistema. No obstante, en la práctica, este trabajador está respaldado por un equipo (en talleres o eventos similares) que incluye otras personas que también trabajan como analistas.

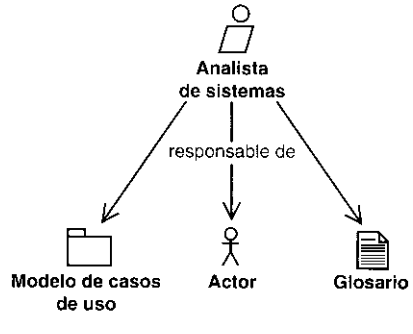


Figura 7.6. Las responsabilidades del analista de sistemas durante la captura de requisitos en forma de casos de uso.

7.3.2. Trabajador: especificador de casos de uso

Habitualmente, el trabajo de captura de requisitos puede no estar dirigido por un solo individuo. De hecho, el analista de sistemas está asistido por otros trabajadores que asumen las responsabilidades de las descripciones detalladas de uno o más casos de uso. Estos trabajadores se denominan *especificadores de casos de uso* (Figura 7.7).

Cada especificador de casos de uso necesita trabajar estrechamente con los usuarios reales de sus casos de uso.

Figura 7.7. Las responsabilidades de un especificador de casos de uso durante la captura de requisitos en forma de casos de uso.

7.3.3. Diseñador de interfaz de usuario

Los diseñadores de interfaces de usuario dan forma[1] visual a las interfaces de usuario. Esto puede implicar el desarrollo de prototipos de interfaces de usuario para algunos casos de uso, habitualmente un prototipo para cada actor (Figura 7.8). Por tanto, es conveniente que cada diseñador de interfaces de usuario dé forma a las interfaces de usuario de uno o más actores.

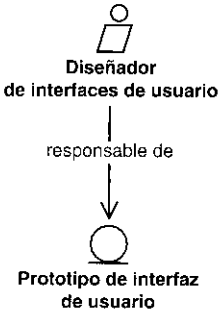


Figura 7.8. Las responsabilidades de un diseñador de interfaces de usuario durante la captura de requisitos en forma de casos de uso.

[1] Entendemos por *diseño de interfaces de usuario* la forma visual de la interfaz de usuario, no la implementación real de la interfaz de usuario. La implementación real la hacen los desarrolladores durante los flujos de trabajo de diseño e implementación.

7.3.4. Trabajador: arquitecto

El arquitecto participa en el flujo de trabajo de los requisitos para describir la vista de la arquitectura del modelo de casos de uso (Figura 7.9).

La vista de la arquitectura del modelo de casos de uso es una entrada importante para planificar las iteraciones, como se describe en la Sección 7.4.2, "Priorizar casos de uso".

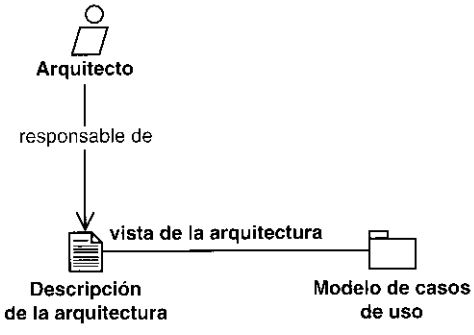


Figura 7.9. Las responsabilidades de un arquitecto durante la captura de requisitos en forma de casos de uso.

7.4. Flujo de trabajo

En la sección anterior, describimos la captura de requisitos en términos estáticos. Ahora vamos a utilizar un diagrama de actividad (Figura 7.10) para describir el comportamiento dinámico.

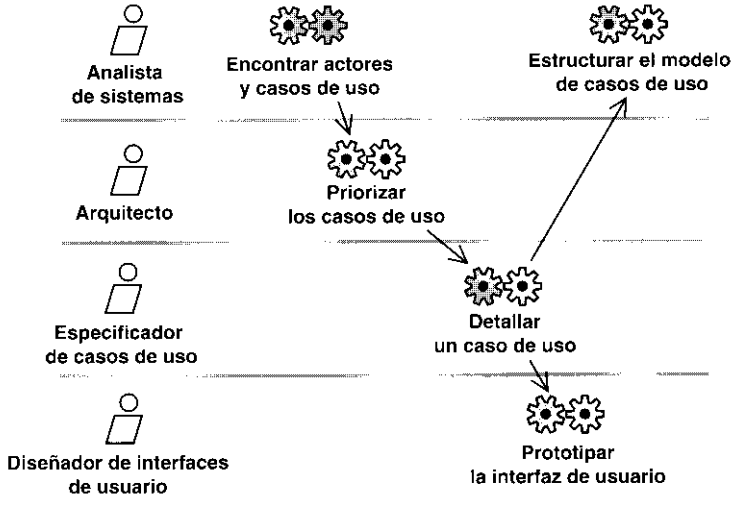


Figura 7.10. El flujo de trabajo para la captura de requisitos en forma de casos de uso, incluyendo trabajadores participantes y sus actividades.

El diagrama utiliza calles para mostrar qué trabajadores ejecutan qué actividades; cada actividad (representada por ruedas dentadas) se sitúa en el mismo campo que el trabajador que la ejecuta. Cuando los trabajadores ejecutan las actividades, crean y modifican artefactos. Describimos los flujos de trabajo como una secuencia de actividades que están ordenadas, así que una actividad produce una salida que sirve de entrada a la siguiente actividad. No obstante, el diagrama de actividad presenta solamente el flujo lógico. En el mundo real, no es necesario trabajar mediante actividades en secuencia. De hecho, podemos trabajar por múltiples vías que producen un resultado final equivalente. Podemos, por ejemplo, comenzar encontrando algunos casos de uso (la actividad de Encontrar Actores y Casos de Uso), después diseñar las interfaces de usuario (la actividad de Prototipar la Interfaz de Usuario), solamente para darnos cuenta de que necesitamos añadir un nuevo caso de uso (así que retrocederemos a la actividad de Encontrar Actores y Casos de Uso, rompiendo la secuencia estricta marcada), y así sucesivamente.

Una actividad puede ser retomada muchas veces, y cada una de éstas puede acarrear la ejecución de una sola fracción de la actividad. Por ejemplo, cuando retomamos la actividad de Encontrar los Actores y Casos de Uso, el nuevo resultado puede ser solamente una identificación adicional de un caso de uso. Por tanto, los caminos de una actividad a otra actividad ilustran tan sólo la secuencia lógica de actividades utilizando los resultados de la ejecución de una actividad como entrada para ejecutar otra.

Primero, el analista de sistemas (una persona respaldada por un equipo de analistas) ejecuta la actividad de Encontrar Actores y Casos de Uso para preparar una primera versión del modelo de casos de uso, con los actores y casos de uso identificados. El analista de sistemas debe asegurar que el desarrollo del modelo de casos de uso captura todos los requisitos que son entradas del flujo de trabajo, es decir, la lista de características y el modelo de dominio o de negocio. Entonces, el arquitecto/s identificará/n los casos de uso relevantes arquitectónicamente hablando, para proporcionar entradas a la priorización de los casos de uso (y posiblemente otros requisitos) que van a ser desarrollados en la iteración actual. Hecho esto, los especificadores de casos de uso (varios individuos) describen todos los casos de uso que se han priorizado. Más o menos en paralelo con ellos, los diseñadores de interfaz de usuario (varios individuos) sugieren las interfaces de usuario adecuadas para cada actor basándose en los casos de uso. Entonces el analista de sistemas reestructura el modelo de casos de uso definiendo generalizaciones entre los casos de uso para hacerlo lo más comprensible posible (comentaremos brevemente las generalizaciones en la actividad de Estructurar el Modelo de Casos de Uso).

Los resultados de la primera iteración a través de este flujo de trabajo consisten en una primera versión del modelo de casos de uso, los casos de uso y cualquier prototipo de interfaz de usuario asociado. Los resultados de cualquier iteración subsiguiente consistirán entonces en nuevas versiones de estos artefactos. Hay que recordar que todos los artefactos se completan y mejoran incrementalmente a través de las iteraciones.

Las distintas actividades en el modelado de casos de uso adoptan formas diferentes en diferentes fases del proyecto (*véase* la Sección 6.4). Por ejemplo, cuando un analista de sistemas ejecuta la actividad de Encontrar Actores y Casos de Uso durante la fase de inicio, identificará muchos actores y casos de uso nuevos. Pero cuando la actividad se realice durante la fase de construcción, el analista hará sobre todo cambios secundarios en el conjunto de actores y casos de uso, tales como la creación de un diagrama de casos de uso que describa mejor el modelo de casos de uso desde una perspectiva en particular. A continuación, vamos a describir las actividades que típicamente aparecen en la iteración de elaboración.

7.4.1. Actividad: encontrar actores y casos de uso

Identificamos los actores y los casos de uso para:

- Delimitar el sistema de su entorno.
- Esbozar quién y qué (actores) interactuarán con el sistema, y qué funcionalidad (casos de uso) se espera del sistema.
- Capturar y definir un glosario de términos comunes esenciales para la creación de descripciones detalladas de las funcionalidades del sistema (es decir, de los casos de uso).

La identificación de actores y casos de uso es la actividad más decisiva para obtener adecuadamente los requisitos, y es responsabilidad del analista de sistemas (Figura 7.11). Pero el analista de sistemas[2] no puede hacer este trabajo solo. El analista requiere entradas de un equipo que incluye al cliente, los usuarios, y otros analistas que participan en talleres de modelado dirigidos por el analista de sistemas.

Algunas veces puede que tengamos un modelo del negocio del cual partir. Si es así, el equipo puede preparar un primer borrador del modelo de casos de uso más o menos "automáticamente". Otras veces, pueden partir del modelo del dominio, o la entrada puede ser solamente una breve descripción general o la especificación de requisitos detallada que incluye las características generales que se requieren. También podemos tener como entrada los requisitos adicionales que no pueden ubicarse en casos de uso individuales. Remitimos al Capítulo 6 para una descripción de esos distintos artefactos de entrada.

Esta actividad consta de cuatro pasos:

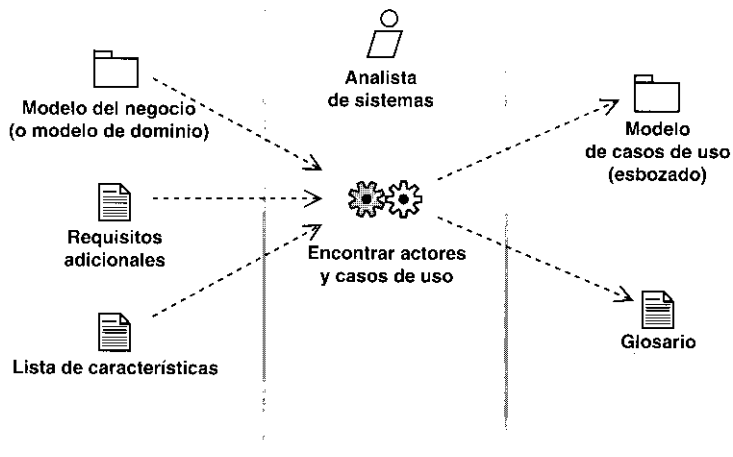


Figura 7.11. Las entradas y los resultados de identificar actores y casos de uso.

[2] Realmente, queremos decir aquí, "la persona que actúa como analista del sistema". Llega a ser un poco pesado el distinguir entre una persona real y el papel de trabajador que representa, pero esto hace al final la descripción más clara. También utilizamos el mismo enfoque para los otros trabajadores que se tratan.

- Encontrar los actores.
- Encontrar los casos de uso.
- Describir brevemente cada caso de uso.
- Describir el modelo de casos de uso completo (este paso también incluye la preparación de un glosario de términos).

Estos pasos no tienen por qué ser ejecutados en ningún orden en particular y a menudo se hacen simultáneamente. Por ejemplo, el diagrama de casos de uso puede actualizarse tan pronto como identifiquemos un nuevo actor o caso de uso.

El resultado de esta actividad es una nueva versión del modelo de casos de uso con actores y casos de uso nuevos o cambiados. Podemos describir y dibujar superficialmente el artefacto modelo de casos de uso resultante, hasta el punto de poder describir cada caso de uso en detalle, que es lo que hace la siguiente actividad: detallar un Caso de Uso. La Figura 7.12 es una ilustración de un diagrama de caso de uso de ese tipo (madurado y reestructurado a través de algunas iteraciones). Pronto lo describiremos con más detalle.

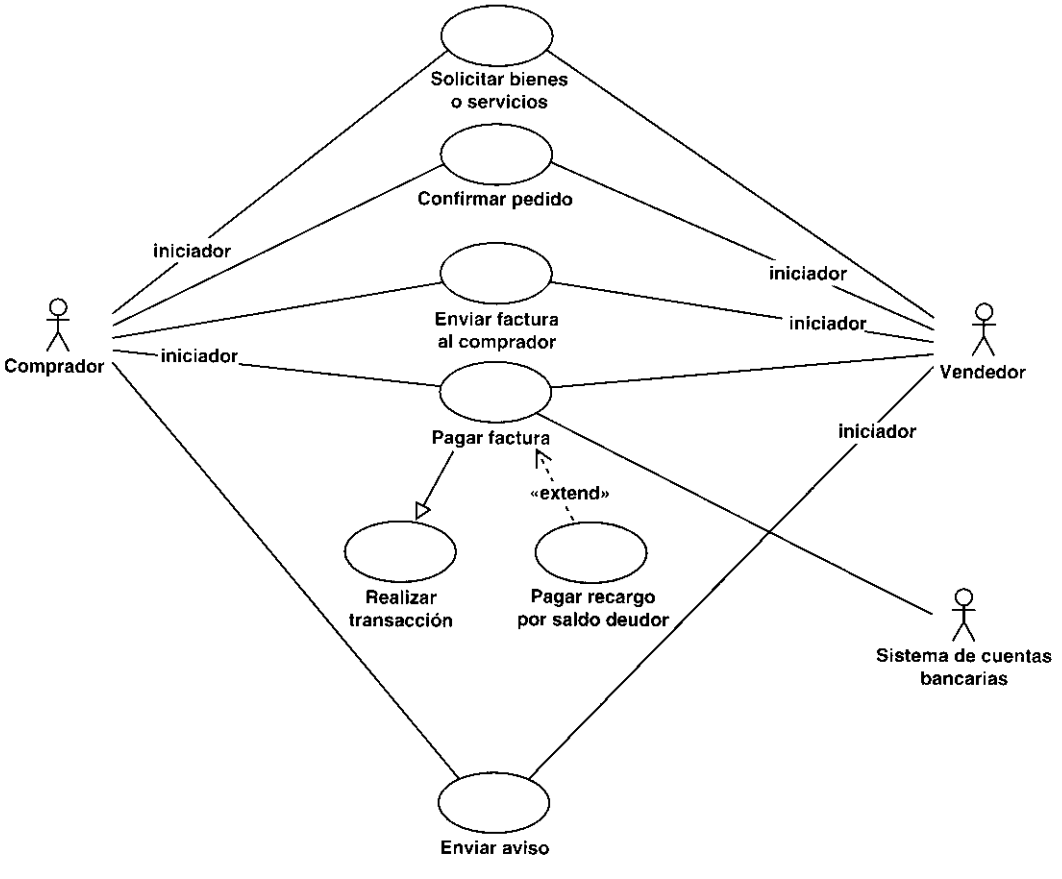


Figura 7.12. Casos de uso en el Sistema de Pagos y Facturación que soporta el caso de uso del negocio Ventas: del pedido a la entrega. El papel *iniciador*, conectado a las asociaciones, indica qué actor comienza el caso de uso.

7.4.1.1. Encontrar los actores

La tarea de encontrar los actores depende de nuestro punto de partida.

Cuando tenemos un modelo del negocio del cual partir, encontrar los actores resulta sencillo. El analista de sistemas puede asignar un actor a cada trabajador del negocio y un actor a cada actor del negocio (es decir, a cada cliente del negocio) que utilizará la información del sistema (*véase* también el Capítulo 6, Sección 6.6.3).

En otro caso, con o sin un modelo del dominio, el analista del sistema, junto con el cliente, identifica los usuarios e intenta organizarlos en categorías representadas por actores.

En ambos casos, debemos identificar los actores que representan sistemas externos y los actores para el mantenimiento y operación del sistema.

Hay dos criterios útiles a la hora de elegir los candidatos a actores: primero, debería ser posible identificar al menos a un usuario que pueda representar al actor candidato. Esto nos ayudará a encontrar solamente a los actores relevantes, y eliminará a los actores que sólo son fantasmas en nuestra imaginación. Segundo, debería existir una coincidencia mínima entre los roles que desempeñan las instancias de los diferentes actores en relación con el sistema. No queremos dos o más actores que tengan en esencia los mismos roles. Si esto ocurre, debemos intentar combinar los papeles en un conjunto de roles que asignaremos exclusivamente a un actor, o bien encontrar un actor generalizado que tenga asignados los roles comunes a los actores que se solapan. Este nuevo actor puede ser especializado utilizando generalizaciones. Por ejemplo, el comprador y el vendedor pueden ser especializaciones del actor Cliente del Banco. En un primer momento, es habitual encontrar muchos actores que se solapan. Esto lleva a algunas discusiones antes de encontrar el conjunto de actores adecuado y de la definición de las generalizaciones.

El analista de sistemas da nombre a los actores y describe brevemente los papeles de cada actor y para qué utiliza el sistema el actor. Encontrar nombres relevantes para los actores es importante para comunicar la semántica deseada. La descripción breve de cada actor debe esbozar sus necesidades y responsabilidades.

Ejemplo **Los actores Comprador, Vendedor y Sistema de Cuentas Bancarias**

Comprador

Un Comprador representa a una persona que es responsable de adquirir bienes o servicios como se describe en el caso de uso Ventas: del pedido a la entrega. Esta persona puede ser un individuo (es decir, no asociado a una compañía), o alguien dentro de una empresa. El Comprador de bienes y servicios necesita el Sistema de Facturación y Pagos para enviar pedidos y pagar las facturas.

Vendedor

Un Vendedor representa a una persona que vende y distribuye bienes o servicios. El Vendedor utiliza el sistema para conseguir nuevos pedidos y entregar las confirmaciones de pedido, facturas y avisos de pago.

Sistema de Cuentas Bancarias

El Sistema de Facturación y Pagos envía verificaciones de transacciones al Sistema de Cuentas Bancarias.

El resultado de este paso es una nueva versión del artefacto modelo de casos de uso con un conjunto de actores actualizado, cada uno con una breve descripción. Estos actores brevemente descritos pueden utilizarse ahora como punto de partida para encontrar los casos de uso.

7.4.1.2. Encontrar los casos de uso Cuando el punto de partida es el modelo del negocio, encontramos los actores y casos de la forma descrita en la Sección 6.6.3, "Búsqueda de casos de uso a partir de un modelo del negocio", del Capítulo 6. Se propone un caso de uso para cada rol de cada trabajador que participa en la realización de casos de uso del negocio y que utilizará información del sistema. En otros casos, el analista de sistemas identificará los casos de uso a través de los talleres con los clientes y los usuarios. El analista de sistemas va repasando los actores uno por uno y proponiendo los casos de uso para cada actor. Por ejemplo, pueden utilizarse las entrevistas y el storyboarding para comprender qué casos de uso son necesarios: *véase* [16]. El actor necesitará normalmente casos de uso para soportar su trabajo de creación, cambio, rastreo, eliminación o estudio de los objetos del negocio, como Pedidos y Cuentas, que se utilizan en los casos de uso del negocio. El actor también puede informar al sistema acerca de algunos sucesos externos u otras formas de representación —el actor puede necesitar al sistema para informarle de algunos sucesos que han tenido lugar, como cuando una factura ha vencido y no se ha pagado—. Puede haber también actores adicionales que ejecuten el inicio del sistema, su mantenimiento o su terminación.

Algunos de los candidatos no llegarán a ser casos de uso por sí mismos; en cambio, podrán ser partes de otros casos de uso. Hay que recordar que intentamos crear casos de uso fáciles de modificar, revisar, probar y manejar unitariamente.

Elegimos un nombre para cada caso de uso de forma que nos haga pensar en la secuencia de acciones concreta que añade valor a un actor. El nombre de un caso de uso a menudo comienza con un verbo, y debe reflejar cuál es el objetivo de la interacción entre el actor y el sistema. En nuestro ejemplo, tenemos casos de uso como Pagar Factura y Solicitar Bienes o Servicios.

Algunas veces es difícil decidir el ámbito de un caso de uso. Una secuencia de interacciones usuario-sistema se puede especificar en un caso de uso o en varios, los cuales el actor invoca uno tras otro. Cuando decidimos si un caso de uso candidato debe ser un caso de uso como tal, tenemos que considerar si es completo por sí mismo o si siempre se ejecuta como continuación de otro caso de uso. Hay que recordar que los casos de uso añaden valor a los actores (*véase* la Sección 7.2.3, "Caso de uso"). Para ser más específicos, un caso de uso entrega un resultado que se puede observar y que añade valor a un actor en concreto. Esta norma práctica para identificar un "buen" caso de uso puede ayudar a determinar el ámbito apropiado de uno de ellos.

Obsérvese que hay dos frases claves en estas directrices que constituyen criterios útiles para la identificación de casos de uso, *resultado de valor* y *un actor en concreto:*

- *Resultado de valor:* Cada ejecución satisfactoria de un caso de uso debe proporcionar algún valor al actor para alcanzar su objetivo [3]. En algunos casos, el actor quiere pagar por el valor devuelto. Nótese que una instancia de caso de uso, como la de una llamada telefónica, puede implicar a más de un actor. En este caso, el criterio para "un resultado de valor observable" debe ser aplicado al actor *iniciador.* Este criterio de "resultado de valor" nos ayuda a evitar encontrar casos de uso demasiado pequeños.

Ejemplo **El ámbito del caso de uso Pagar Facturas**

El Sistema de Facturación y Pago ofrece un caso de uso llamado Pagar Facturas, que lo utiliza un comprador para planificar los pagos de las facturas por los bienes que él o ella ha solicitado y recibido.

El caso de uso Pagar Facturas incluye tanto la planificación como la ejecución de un pago. Si dividimos el caso de uso en dos partes, una para la planificación y otra para la ejecución de pagos, "Planificación de Pagos" no añadiría valor por sí mismo. El valor se obtiene una vez que la factura ha sido pagada (en este punto no nos tenemos que preocupar por los recordatorios de pago).

- *Un actor en concreto:* Identificando casos de uso que proporcionen valores a usuarios individuales reales, nos aseguramos de que los casos de uso no se harán demasiado grandes.

Como pasa con los actores, los casos de uso que identificamos primero necesitan a menudo ser reestructurados y reevaluados un par de veces antes de que el modelo de casos de uso se estabilice.

Como ya tratamos en el Capítulo 4, los casos de uso y la arquitectura del sistema se desarrollan mediante iteraciones. Una vez que ya tenemos la arquitectura, los nuevos casos de uso que capturemos deben ser adaptados a la arquitectura existente. Los casos de uso que no se ajusten a la arquitectura elegida deben ser modificados para que encajen mejor con la arquitectura (podemos también mejorar la arquitectura para acomodar los nuevos casos de uso). Por ejemplo, podemos haber hecho el trabajo inicial de especificación de un caso de uso con una interacción particular del usuario en mente. Una vez que hemos elegido un determinado marco de trabajo de IGU puede que tengamos que modificar los casos de uso, aunque estas adaptaciones sean, normalmente, muy pequeñas.

No obstante, se pueden requerir cambios más drásticos. El analista de sistemas puede proponer una forma de supervisión de la carga del sistema especificando un simulador que actúe como un actor relevante, que requiere casos de uso del sistema. De esta forma, se puede medir los tiempos de respuesta y otros requisitos de ejecución. También se puede medir el tiempo que un caso de uso permanece en colas internas del sistema. Estos dos enfoques pueden dar valores similares, pero el coste de su implementación puede ser muy diferente dependiendo de la arquitectura existente. Así que el analista de sistemas puede necesitar que se renegocien los requisitos (casos de uso y demás) con el cliente para construir un sistema mejor, más fácil de implementar y mantener para los desarrolladores. El cliente también gana con la renegociación de los requisitos y consigue la funcionalidad del sistema antes, con una mayor calidad y un menor coste.

7.4.1.3. Describir brevemente cada caso de uso

A medida que los analistas van identificando los casos de uso, algunas veces garabatean unas pocas palabras explicando cada caso de uso, o algunas veces sólo escriben los nombres. Después, describen brevemente cada caso de uso, en primer lugar con algunas frases que resumen las acciones y más tarde, con una descripción paso a paso de lo que el sistema necesita hacer cuando interactúa con sus actores.

Ejemplo **Descripción inicial del caso de uso Pagar Factura**

Breve descripción

El Comprador utiliza el caso de uso Pagar Factura para planificar los pagos de las facturas. El caso de uso Pagar Factura efectúa el pago el día de vencimiento de la misma.

Descripción paso a paso inicial

Después de que el caso de uso comience, el Comprador ya ha recibido una factura (enviada por otro caso de uso llamado Enviar Factura al Comprador), y también ha recibido los bienes y servicios demandados:

1. El Comprador estudia la factura a pagar y verifica que se corresponde con el pedido original.
2. El Comprador planifica el pago de la factura por banco.
3. Cuando vence el día de pago, el sistema revisa si hay suficiente dinero en la cuenta del Comprador. Si tiene suficiente dinero disponible, se hace la transacción.

Hasta aquí hemos descrito brevemente los actores y los casos de uso. No obstante, no es suficiente con describir y comprender cada caso de uso aisladamente. También necesitamos ver el cuadro completo. Necesitamos explicar cómo están relacionados entre sí los casos de uso y los actores y cómo juntos constituyen el modelo de casos de uso.

7.4.1.4. Descripción del modelo de casos de uso en su totalidad

Preparamos diagramas y descripciones para explicar el modelo de casos de uso en su totalidad, especialmente cómo se relacionan los casos de uso entre sí y con los actores.

No hay una regla estricta sobre lo que se debe incluir en el diagrama. De hecho, elegimos el conjunto de diagramas que describan más claramente el sistema. Por ejemplo, podemos dibujar diagramas para mostrar los casos de uso que participan en un caso de uso del negocio (*véase* la Figura 7.12) o quizás para ilustrar los casos de uso que ejecuta un actor.

Para asegurar la consistencia cuando se describen varios casos de uso concurrentemente, resulta práctico desarrollar un glosario de términos. Estos términos pueden derivar en (y llevar la traza de) clases en un modelo del dominio o un modelo de negocio (descrito en el Capítulo 6).

El modelo de casos de uso requiere ser un modelo plano, como el que se describe aquí. También puede organizarse en conjuntos de casos de uso llamados *paquetes de casos de uso* [12].

La salida de este paso es también una descripción general del modelo de casos de uso. Esta descripción explica el modelo de casos de uso como un todo. Describe cómo interactúan los actores y los casos de uso y como se relacionan entre sí los casos de uso. En la representación del modelo de casos de uso de UML, la descripción general es un valor etiquetado del propio modelo (*véase* el Apéndice A, Sección A.1.2).

Ejemplo **Descripción general**

La descripción general para el modelo de casos de uso del Sistema de Facturación y Pagos (*véase* la Figura 7.12) podría parecerse a lo siguiente. En esta descripción, hemos incluido comentarios numerados, que se explican al final del ejemplo.

El comprador utiliza el caso de uso Solicitar Bienes y Servicios para buscar los productos y precios, para realizar un pedido y después enviarlo.

Tarde o temprano, los bienes o servicios le llegarán al comprador junto con una factura.

El comprador activa el caso de uso Pagar Factura para dar el visto bueno a la factura recibida y planificar el pago requerido. En la fecha planificada, el caso de uso Pagar Factura transferirá automáticamente el dinero desde la cuenta del comprador a la cuenta del vendedor (comentario 1).

Es más, el caso de uso Pagar Cargos Saldo Deudor extiende al caso de uso Pagar Factura si se produce un descubierto en el saldo (comentario 2).

Vamos a pasar ahora a cómo utiliza el sistema el vendedor. El vendedor puede estudiar, proponer cambios y confirmar los pedidos recibidos utilizando el caso de uso Confirmar Pedidos. Un pedido confirmado estará seguido de la entrega de los bienes o servicios (no descritos en nuestro modelo de casos de uso de ejemplo; de hecho se hace fuera del Sistema de Facturación y Pagos).

Más tarde, cuando hayan sido entregados los bienes o servicios, el vendedor facturará al comprador a través del caso de uso Enviar Factura al Comprador. Al llevar a cabo la facturación, el vendedor puede que tenga que aplicar un descuento y puede que también elija combinar varias facturas en una.

Si el comprador no ha pagado en la fecha de vencimiento, se informará al vendedor y éste puede utilizar el caso de uso Enviar Aviso. El sistema podría enviar avisos automáticamente, pero tenemos que elegir una solución en la que el vendedor tenga la oportunidad de revisar los avisos antes de que sean enviados para evitar violentar a algún cliente (comentario 3).

Comentarios

1. Recuérdese que el modelo de casos de uso es más que una lista de casos de uso. También describe generalizaciones entre esos casos de uso. Por ejemplo, la subsecuencia de acciones para transacciones de pago en el caso de uso Pagar Factura puede ser compartida por muchos casos de uso (incluso aunque solamente se muestre una generalización en la Figura 7.12). Esta compartición puede representarse con un caso de uso separado llamado Realizar Transacción que se reutiliza, mediante generalizaciones, por parte de varios casos de uso, como Pagar Facturas. La generalización significa que la subsecuencia de acciones descrita en el caso de uso Realizar Transacción se inserta en la secuencia descrita en Pagar Facturas. Cuando el sistema ejecuta una instancia del caso de uso Pagar Factura, la instancia también seguirá el comportamiento descrito en el caso de uso Ejecutar Transacción.
2. A medida que el sistema ejecuta la instancia del caso de uso Pagar Factura, la secuencia puede extenderse para incluir la secuencia de acciones descrita en el caso de uso extendido Pagar Cargos Saldo Deudor. Hemos mencionado brevemente las relaciones de generalización y extensión para mostrar que el modelo de casos de uso puede estructurarse para hacer más fácil la especificación y comprensión del conjunto total de requisitos funcionales; para más información *véase* [7].
3. Enviar Aviso muestra un caso de uso que simplifica los caminos correctivos en los casos de uso del negocio. Cada camino correctivo ayuda al proceso "a ponerse sobre la pista de nuevo", para prevenir un pequeño problema en la iteración con el cliente, que podría convertirse en un problema grande. Por tanto los actores también necesitan casos de uso (o alternativas de los casos de uso) que les ayuden a corregir desviaciones en los caminos. Esta clase de casos de uso a menudo constituyen una gran cantidad de las responsabilidades totales del sistema para tratar las muchas posibles desviaciones.

Descripción

Hay varios modos de dar forma a un modelo de casos de uso, y este ejemplo sólo ilustra una de ellas. Vamos a tratar algunas de las soluciones de compromiso que hemos tomado. ¿Qué pasa si el comprador hojea un catálogo de bienes y servicios *disponibles* por Internet y después hace un pedido

interactivo, y obtiene la confirmación al instante? ¿Seguiría siendo necesario un caso de uso separado Confirmar Pedido? La respuesta es que no, ya que la confirmación directa podría incluirse en el caso de uso Solicitar Bienes y Servicios.

En este ejemplo, no obstante, hemos asumido que después que un pedido haya sido analizado, o bien se confirma, o bien el vendedor realiza una propuesta alternativa. Por ejemplo, el vendedor puede sugerir un conjunto de bienes alternativos de igual utilidad, más barato, o que se entrega antes. La secuencia real de realización de un pedido puede comprender realmente en este caso una serie de pasos en los cuales el vendedor propone un pedido alternativo y después lo confirma, como se describe a continuación:

1. Un comprador realiza un pedido inicial.
2. El vendedor sugiere un pedido alternativo y se lo envía al comprador.
3. El comprador realiza el pedido final.
4. El vendedor envía al comprador una confirmación de pedido.

Esto pasos se cubren con dos casos de uso: Solicitar Bienes y Servicios y Confirmar Pedido. ¿Por qué no hay cuatro casos de uso diferentes —o sólo un caso de uso—? Vamos a ver primero por qué estos pasos no se describen como un caso de uso grande: no queremos forzar al vendedor y al comprador a interactuar en tiempo real. De hecho, queremos que sean capaces de enviarse peticiones el uno al otro sin que tengan la necesidad de esperar o de estar sincronizados. Podemos asumir que el vendedor quiere almacenar los nuevos pedidos de diferentes compradores y entonces analizarlos y confirmarlos todos al mismo tiempo. Esto no podría ser posible si los cuatro pasos estuvieran en un solo caso de uso, porque cada caso de uso se considera atómico y, por tanto siempre se termina antes de comenzar otro caso de uso.

Como resultado, un vendedor no podría tener varios pedidos iniciales pendientes (después del paso 1), y habría que esperar para que los pedidos alternativos fuesen enviados (paso 2, en el ejemplo).

Por supuesto que los cuatro pasos podrían llegar a ser cuatro casos de uso distintos, pero un pedido inicial y un pedido final (pasos 1 y 3) son tan similares que podrían expresarse como alternativas del caso de uso Solicitar Bienes y Servicios. Después de todo, no queremos que el conjunto de casos de uso crezca hasta hacerse incomprensible. Demasiados casos de uso hacen que el modelo de casos de uso sea difícil de comprender y de utilizar como entrada del análisis y el diseño.

De forma similar, proponer un pedido alternativo (paso 2) implica la confirmación de las partes de un pedido y la sugerencia de alternativas para otras partes, mientras que la confirmación de un pedido (paso 4) implica la confirmación de todas las partes. Ambos pasos pueden expresarse en un mismo caso de uso, Confirmar Pedido, que permite al vendedor tanto confirmar las partes del pedido como sugerir alternativas. Así que habrá un caso de uso llamado Solicitar Bienes y Servicios, que proporciona los servicios correspondientes a los pasos 1 y 3, y otro caso de uso llamado Confirmar Pedido, que se corresponde con los pasos 2 y 4.

Cuando la descripción del modelo de casos de uso esté preparada, dejamos que dé el visto bueno del modelo de casos de uso la gente que no forma parte del equipo de desarrollo (es decir, clientes y usuarios), convocando una revisión informal para determinar si:

- Se han capturado como casos de uso todos los requisitos funcionales necesarios.
- La secuencia de acciones es correcta, completa y comprensible para cada caso de uso.
- Se identifica algún caso de uso que no proporcione valor. Si es así, ese caso de uso debería reconsiderarse.

7.4.2. Actividad: priorizar casos de uso

El propósito de esta actividad es proporcionar entradas a la priorización de los casos de uso para determinar cuáles son necesarios para el desarrollo (es decir, análisis, diseño, implementación, etc.) en las primeras iteraciones, y cuáles pueden dejarse para más tarde (Figura 7.13).

Los resultados se recogen en la vista de la arquitectura del modelo de casos de uso. Después, esta vista se revisa con el jefe de proyecto, y se utiliza como entrada al hacer la planificación de lo que debe desarrollarse dentro de una iteración. Hay que darse cuenta de que esta planificación también necesita la consideración de otros aspectos no técnicos, como los aspectos económicos o del negocio del sistema que va a ser desarrollado (*véase* el Capítulo 12, Sección 12.4.2).

La vista de la arquitectura del modelo de casos de uso debe mostrar los casos de uso significativos desde el punto de vista de la arquitectura. Remitimos a la Sección 7.2.4, "Descripción de la arquitectura (vista del modelo de casos de uso)" para más detalles.

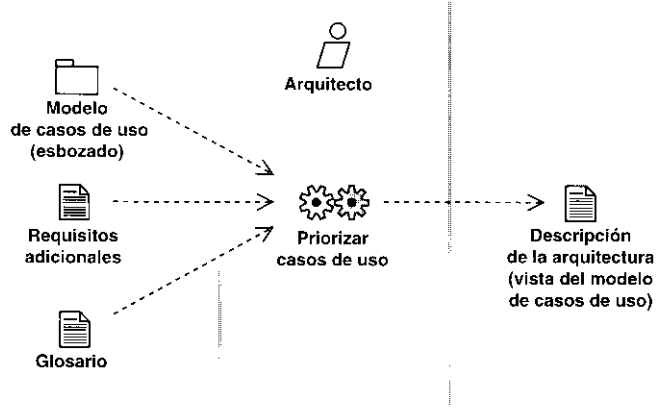


Figura 7.13. Las entradas y los resultados de priorizar los casos de uso.

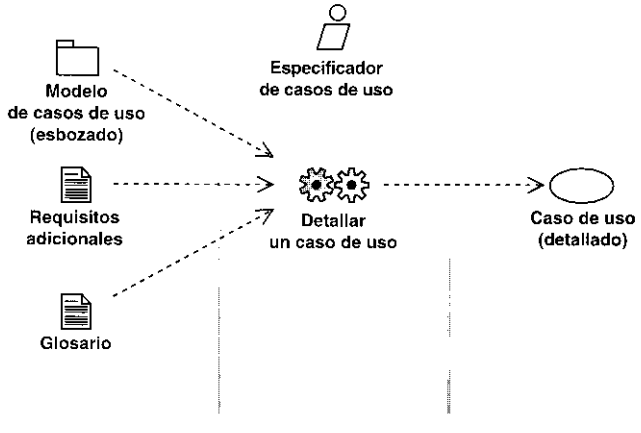


Figura 7.14. Las entradas y los resultados de detallar cada caso de uso.

7.4.3. Actividad: detallar un caso de uso

El objetivo principal de detallar cada caso de uso es describir su flujo de sucesos en detalle, incluyendo cómo comienza, termina e interactúan con los actores (Figura 7.14).

Con el modelo de casos de uso y los diagramas de casos de uso asociados como punto de comienzo, el especificador de un caso de uso individual puede ya describir cada caso de uso en detalle. El especificador de casos de uso detalla paso a paso la descripción de cada caso de uso en una especificación precisa de la secuencia de acciones. En esta sección veremos:

- Cómo estructurar la descripción para especificar todas las vías alternativas del caso de uso.
- Qué incluir en una descripción de caso de uso.
- Cómo formalizar la descripción del caso de uso cuando sea necesario.

Cada especificador de casos de uso debe trabajar estrechamente con los usuarios reales de los casos de uso. El especificador de casos de uso necesita entrevistarse con los usuarios, quizás anotar su comprensión de los casos de uso y discutir propuestas con ellos, y solicitarles que revisen la descripción de los casos de uso.

El resultado de esta actividad es la descripción detallada de un caso de uso en particular en forma de texto y diagramas.

7.4.3.1. Estructuración de la descripción de casos de uso

Ya hemos mencionado que un caso de uso define los estados que las instancias de los casos de uso pueden tener y la posible transición entre estos estados (*véase* la Figura 7.15). Cada transición es una secuencia de acciones que se ejecutan en una instancia del caso de uso cuando ésta se dispara por efecto de un suceso, como podría ser un mensaje.

El gráfico de transición de estados ilustrado en la Figura 7.15 puede llegar a ser bastante intrincado. Por ello, debemos describir la posible transición de estados de manera simple y precisa. Una técnica probada es elegir un camino básico completo (las flechas rectas en la Figura 7.15) desde el estado de inicio al estado final, y describir ese camino en una sección de la descripción. Entonces podemos describir en secciones separadas el resto de los caminos (flechas curvas) como caminos alternativos o desviaciones del camino básico. Algunas veces, no obstante, las alternativas o desviaciones son lo suficientemente pequeñas como para explicarlas "en línea" como parte de la descripción del camino básico. El sentido común determina

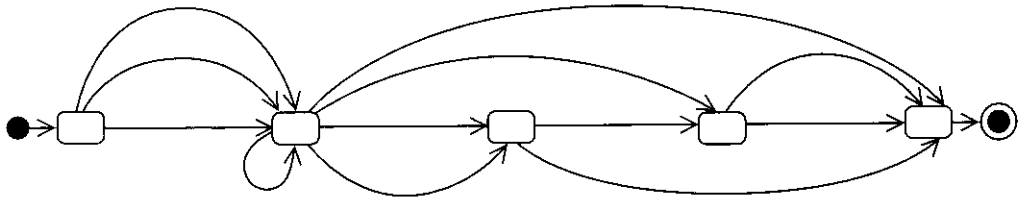


Figura 7.15. Un caso de uso puede imaginarse como si tuviera un estado de comienzo (el rectángulo redondeado de más a la izquierda), estados intermedios (los subsiguientes rectángulos redondeados), estados finales (el rectángulo redondeado de más a la derecha) y transiciones de un estado a otro. (*Véase* el Apéndice A para la notación de diagramas de estados.) Las flechas rectas ilustran el camino básico, y las curvas, otros caminos.

si la descripción debe ser "en línea" o se debe crear una sección separada para ella. Hay que recordar que el objetivo es hacer una descripción precisa pero fácil de leer. Con cualquier técnica que elijamos, tendremos que describir todas las alternativas, sino no habremos especificado el caso de uso.

Las alternativas, desviaciones, o excepciones del camino básico pueden ocurrir por muchas razones:

- El actor puede elegir entre diferentes caminos en el caso de uso. Por ejemplo, durante el caso de uso Pagar Factura, el actor puede decidir pagar una factura o rechazarla.
- Si está implicado más de un actor en el caso de uso, las acciones de uno de ellos pueden influenciar el camino de las acciones del resto de actores.
- El sistema puede detectar entradas erróneas de los actores.
- Algunos recursos del sistema pueden tener un mal funcionamiento, y así impedir que el sistema realice de forma adecuada su trabajo.

El camino básico elegido debe ser el camino "normal", esto es, el que el usuario percibe como el que más habitualmente va a seguir y aquél que proporciona el valor más obvio al actor. Generalmente, cada camino básico debe abarcar un par de excepciones y un par de peculiaridades que el sistema raramente necesita manejar.

Ejemplo Caminos en el caso de uso Pagar Factura

Por favor, nótese cómo ha cambiado el texto desde el primer borrador de este capítulo cuando teníamos solamente un esbozo de la descripción de un caso de uso (*véase* la Sección 7.4.1.3, "Describir brevemente cada caso de uso"). Este cambio refleja cómo detallamos los casos de uso a medida que los modelamos, aunque las descripciones completas de los casos de uso en realidad es más probable que sean más grandes y que cubran más caminos.

Precondición: *el comprador ha recibido los bienes y servicios y al menos una factura del sistema. El comprador ahora planifica el pago de las facturas.*

Flujo de sucesos

Camino básico

1. El comprador invoca al caso de uso para comenzar a hojear las facturas recibidas del sistema. El sistema verifica que el contenido de cada factura es consistente con las confirmaciones de pedido recibidas anteriormente (como parte del caso de uso Confirmar Pedido) y así indicárselo al comprador. La confirmación de pedido describe qué elementos serán enviados, cuándo, dónde y a qué precio.
2. El comprador decide planificar una factura para pagarla por banco, y el sistema genera una petición de pago para transferir el dinero a la cuenta del vendedor. Nótese que un comprador no puede planificar el pago de la misma factura dos veces.
3. Más tarde, si hay suficiente dinero en la cuenta del comprador, se hace un pago mediante transacción en la fecha planificada. Durante la transacción, el dinero se transfiere de la cuenta del comprador a la cuenta del vendedor, como se describe en el caso de uso abstracto Realizar Transacción (que es utilizado por Pagar Factura). Tanto el comprador como el

vendedor tienen notificación del resultado de la transacción. El banco recoge unos cargos por la transacción, que se retiran de la cuenta del comprador.

4. La instancia del caso de uso finaliza.

Caminos alternativos

En el paso 2, el comprador puede, en cambio, pedir al sistema que devuelva un rechazo de factura al vendedor.

En el paso 3, si no hay suficiente dinero en la cuenta, el caso de uso cancelará el pago y se lo notificará al comprador.

Postcondición: la instancia del caso de uso termina cuando la factura ha sido pagada o cuando el pago de la factura ha sido cancelado y no se ha hecho ninguna transferencia.

Debido a que los casos de uso deben ser entendidos por los desarrolladores y por los clientes y usuarios, deberían describirse siempre utilizando un castellano corriente como el que se utiliza en este ejemplo.

7.4.3.1.1. ¿Qué incluir en una descripción de caso de uso? Como hemos mostrado en el ejemplo anterior, la descripción de un caso de uso debe incluir lo siguiente:

- La descripción de un caso de uso debe definir el estado inicial (*véase* la Figura 7.15) como precondición.
- Cómo y cuándo comienza el caso de uso (es decir, la primera acción a ejecutar; paso 1).
- El orden requerido (si hay alguno) en el que las acciones se deben ejecutar. En este punto el orden se define por una secuencia numerada (pasos 1-4).
- Cómo y cuándo terminan los casos de uso (paso 4).
- Una descripción de caso de uso debe definir los posibles estados finales (Figura 7.15) como postcondiciones.
- Los caminos de ejecución que no están permitidos. La nota del paso 2 nos indica que ese camino no es posible —pagar una factura dos veces—. Es un camino que el usuario no puede tomar.
- Las descripciones de caminos alternativos que están incluidos junto con la descripción del camino básico. Todo lo que hay en el paso 3 son acciones que se ejecutan solamente si hay suficiente dinero en la cuenta.
- Las descripciones de los caminos alternativos que han sido extraídas de la descripción del camino básico (paso 5).
- La interacción del sistema con los actores y qué cambios producen (pasos 2 y 3). Los ejemplos más típicos son cuando el comprador decide planificar el pago de la factura en el paso 2 y cuando se notifica el resultado de la transacción al comprador y al vendedor en el paso 3. En otras palabras, describimos la secuencia de acciones del caso de uso, cómo estas acciones son invocadas por los actores y cómo su ejecución resulta en solicitudes a los actores.

- La utilización de objetos, valores y recursos de sistema (paso 3). Dicho de otra forma, hemos descrito la secuencia de acciones en la utilización de un caso de uso, y hemos asignado valores a los atributos de los casos de uso. Un ejemplo claro es cuando se transfiere el dinero de la cuenta del comprador a la cuenta del vendedor en el paso 3. Otro ejemplo podría ser la utilización de facturas y confirmaciones de pedidos en el paso 1.

- Obsérvese que debemos describir explícitamente qué hace el sistema (las acciones que ejecuta). Debemos ser capaces de separar las responsabilidades del sistema y las de los actores, sino, la descripción del caso de uso no será suficientemente precisa para poder utilizarla como especificación del sistema. Por ejemplo, en el paso 1, escribimos "el *sistema* verifica que el contenido de cada factura es consistente con la confirmación de pedido recibida", y en paso 3, "estos cargos se retiran de la cuenta del comprador por el *sistema*".

Los atributos de los casos de uso pueden utilizarse como inspiración para encontrar más tarde clases y atributos en el análisis y diseño. Por ejemplo, podemos identificar una clase de diseño llamada Factura, derivada de un atributo *factura* de un caso de uso. Durante el análisis y el diseño, también consideraremos algunos objetos que se utilizarán en varios casos de uso, aunque no es necesario considerarlo en el modelo de casos de uso. De hecho (como se ha discutido en la Sección 7.2.3, "Caso de uso"), mantenemos simple el modelo de casos de uso prohibiendo las interacciones entre las instancias de casos de uso y prohibiendo a las instancias acceder a los atributos de las otras.

Hemos hablado a menudo sobre los requisitos funcionales y cómo representarlos mediante casos de uso, pero también necesitamos especificar los requisitos no funcionales. Muchos de los requisitos no funcionales están relatados con casos de uso específicos, como los requisitos para especificar la velocidad, disponibilidad, exactitud, tiempo de respuesta o utilización de memoria que el sistema necesita para ejecutar los casos de uso. Estos requisitos se asocian como requisitos especiales (valores etiquetados en UML) con el caso de uso en cuestión. Esto puede documentarse, por ejemplo, en una sección separada dentro de la descripción de casos de uso.

Ejemplo **Requisito especial (de rendimiento)**

Cuando un comprador envía una factura para su pago, el sistema debe responder con una verificación de la petición en 1,0 segundo en el 90 por ciento de los casos. El tiempo para la verificación no debe exceder nunca 10,0 segundos, a menos que se rompa la conexión de red (en cuyo caso deberíamos notificárselo al usuario).

Si el sistema interactúa con algunos otros sistema (actores no humanos), es necesario especificar esta interacción (esto es, haciendo referencia a un protocolo de comunicaciones estándar). Esto se debe hacer en las primeras iteraciones, durante la fase de elaboración, debido a que la realización de comunicaciones entre sistemas tiene habitualmente influencia en la arquitectura.

Las descripciones de casos de uso se dan por finalizadas cuando se consideran comprensibles, correctas (es decir, capturan adecuadamente los requisitos), completas (es decir, describen

todos los caminos posibles) y consistentes. Las descripciones son evaluadas por los analistas, los usuarios y los clientes en reuniones de revisión que se establecen al final de la captura de requisitos. Solamente los clientes y los usuarios pueden verificar si los casos de uso son los correctos.

7.4.3.2. Formalización de la descripción de casos de uso

La figura 7.15 muestra cómo las transiciones hacen pasar a las instancias de los casos de uso de un estado a otro. A menudo, como cuando el caso de uso tiene solamente unos pocos estados, no necesitamos describir explícitamente los estados. En su lugar, podemos elegir utilizar el estilo utilizado en el ejemplo de Pagar Factura. Aun así, es una buena idea tener una máquina de estados en nuestras mentes cuando describimos un caso de uso, para asegurarnos de que cubre todas las posibles situaciones. No obstante, en algunas ocasiones, como cuando tenemos un sistema en tiempo real complejo, los casos de uso pueden ser muy complejos, por eso puede llegar a ser necesaria una técnica de descripción más estructurada. La interacción entre los actores y los casos de uso puede transitar, por ejemplo, por tantos estados y tantas transiciones alternativas que es casi imposible mantener consistente la descripción textual de los casos de uso. Entonces puede ser útil utilizar una técnica de modelado visual para describir los casos de uso. Estas técnicas pueden ayudar al analista de sistemas a mejorar la comprensión de los casos de uso:

- Pueden utilizarse los diagramas de estado de UML para describir los estados de los casos de uso y las transiciones entre esos estados; *véase* la Figura 7.16.
- Pueden utilizarse los diagramas de actividad para describir las transiciones entre estados con más detalle como secuencias de acciones. Los diagramas de actividad pueden describirse como una generalización de los diagramas de transición de estados de SDL [15], que son una técnica bien probada que se utiliza en telecomunicaciones.
- Se pueden utilizar los diagramas de interacción para describir cómo interactúa una instancia de caso de uso con la instancia de un actor. Los diagramas de interacción, entonces, muestran el caso de uso y el actor (o actores) participante.

Véase [2, 11, 12 17] para la explicación de diagramas de estados, interacción y actividad.

Ejemplo **Utilización de diagramas de estado para describir casos de uso**

La figura 7.16 es un diagrama de estados para el caso de uso de Pagar Factura. El punto negro que está en la parte superior del diagrama indica el comienzo del caso de uso. Esto quiere decir que es ahí donde comienza la máquina de estados cuando se inicia un caso de uso. La flecha que parte del punto negro muestra a dónde transita la máquina de estados inmediatamente cuando se inicia, en este caso, al primer estado, Hojeando. Los estados se muestran como rectángulos con las esquinas redondeadas. La transición entre estados se muestra con flechas que van de un estado a otro.

Primero, el usuario hojea la factura (Paso 1 en el anterior ejemplo de Pagar Factura) y entonces decide planificar (Paso 2) o rechazar (Paso 5) una factura.

El caso de uso transita del estado Factura Planificada a Factura Pagada cuando la factura planificada se paga en la fecha de vencimiento (Paso 3). El caso de uso termina inmediatamente (el

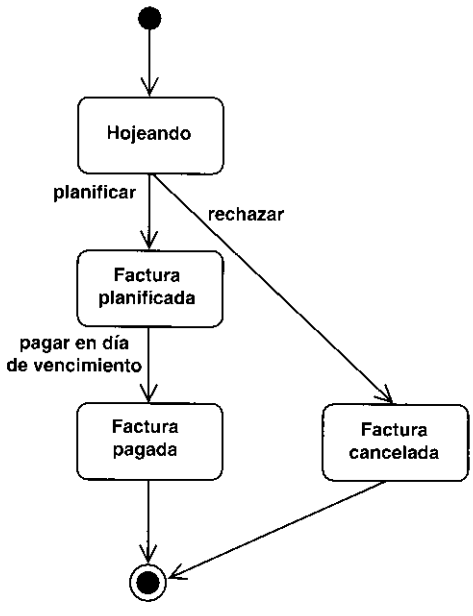


Figura 7.16. El diagrama de estados para el caso de uso Pagar Factura muestra cómo una instancia del caso de uso Pagar Factura transita por diferentes estados (rectángulos redondeados) en una secuencia de transiciones de estado (flechas).

círculo con un punto negro dentro) después de que se ha alcanzado el estado de Factura Pagada o el de Factura Cancelada.

Nótese que el uso de estos diagramas en el contexto de los casos de uso puede llevarnos algunas veces a largos y complejos diagramas que son muy difíciles de leer y comprender. Por ejemplo, un único caso de uso puede implicar muchos estados que son difíciles de nombrar de forma comprensible. Esto es especialmente delicado si los interesados que no son desarrolladores de software deben leer los diagramas. También es muy costoso desarrollar diagramas detallados y mantenerlos consistentes con otros modelos del sistema.

Así que nuestra recomendación básica es utilizar esta clase de diagramas con cuidado, y que las descripciones simplemente textuales (es decir, descripciones de flujos de sucesos) de los casos de uso son con frecuencia suficientes. También, en muchos casos las descripciones textuales y los diagramas pueden complementarse unos con otros.

7.4.4. Actividad: prototipar la interfaz de usuario

El objetivo de esta actividad es construir un prototipo de interfaz de usuario (*véase* la Figura 7.17).

Hasta ahora, el analista de sistemas ha desarrollado un modelo de casos de uso que especifica qué usuarios hay y para qué necesitan utilizar el sistema. Esto se ha presentado en los diagramas de casos de uso, en las descripciones generales del modelo de casos de uso y en las descripciones detalladas para cada caso de uso.

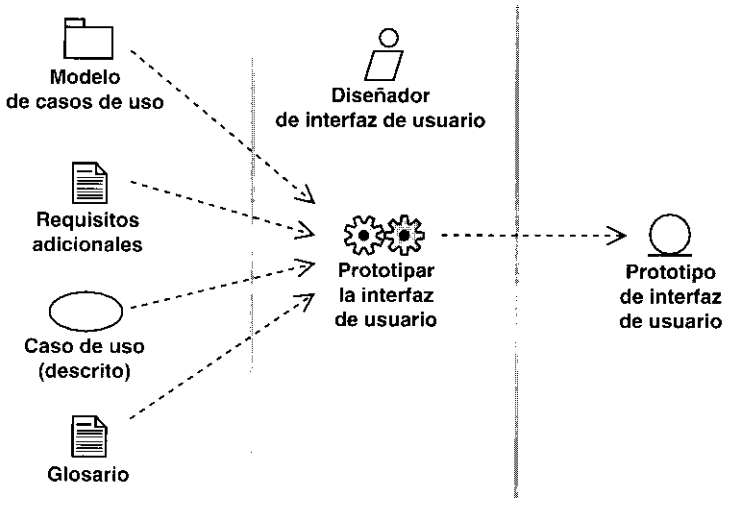


Figura 7.17. Las entradas y los resultados de prototipar la interfaz de usuario.

Ahora necesitamos diseñar la interfaz de usuario que permita al usuario llevar a cabo los casos de uso de manera eficiente. Lo haremos en varios pasos. Comenzamos con los casos de uso e intentamos discernir qué se necesita de las interfaces de usuario para habilitar los casos de uso para cada actor. Esto es, hacemos un diseño lógico de la interfaz de usuario. Después, creamos el diseño físico de la interfaz de usuario y desarrollamos prototipos que ilustren cómo pueden utilizar el sistema los usuarios para ejecutar los casos de uso [1]. Mediante la especificación de qué se necesita antes de decidir cómo realizar la interfaz de usuario, llegamos a comprender las necesidades antes de intentar realizarlas [1].

El resultado final de esta actividad es un conjunto de esquemas de interfaces de usuario y prototipos de interfaces de usuario que especifican la apariencia de esas interfaces de cara a los actores más importantes.

7.4.4.1. Crear el diseño lógico de una interfaz de usuario

Cuando los actores interactúen con el sistema, utilizarán y manipularán elementos de interfaces de usuario que representan atributos (de los casos de uso). A menudo estos son términos del glosario (por ejemplo, *balance de cuenta, fecha de vencimiento* y *titular de cuenta*). Los actores pueden experimentar estos elementos de las interfaces de usuario como iconos, listas de elementos u objetos en un mapa de 2D, y pueden manipularlos por selección, arrastre o hablando con ellos. El diseñador de interfaces de usuario identifica y especifica estos elementos actor por actor, recorriendo todos los casos de uso a los que el actor puede acceder, e identificando los elementos apropiados de la interfaz de usuario para cada caso de uso. Un único elemento de interfaz de usuario puede intervenir en muchos casos de uso [3], desempeñando un papel diferente en cada uno. Así, los elementos de la interfaz de usuario pueden diseñarse para jugar varios roles. Deberíamos responder a las siguientes preguntas para cada actor:

[3] De la misma forma en que una clase puede participar en varias colaboraciones para realizar diferentes casos de uso, la clase desempeña un papel en cada colaboración.

- ¿Qué elementos de interfaz de usuario se necesitan para posibilitar el caso de uso?
- ¿Cómo deberían relacionarse unos con otros?
- ¿Cómo se utilizarán en los diferentes casos de uso?
- ¿Cuál debería ser su apariencia?
- ¿Cómo deberían manipularse?

Para determinar qué elementos de interfaz de usuario necesitan ser accesibles al actor en cada caso de uso, podemos contestar a las siguientes preguntas:

- ¿Qué clases del dominio, entidades del negocio o unidades de trabajo son adecuadas como elementos de la interfaz de usuario para cada caso de uso?
- ¿Con qué elementos de la interfaz de usuario va a trabajar el actor?
- ¿Qué acciones puede invocar el actor, y qué decisiones puede tomar?
- ¿Qué guía o información va a necesitar el actor antes de invocar cada acción de los casos de uso?
- ¿Qué información debe proporcionar el actor al sistema?
- ¿Qué información debe proporcionar el sistema al actor?
- ¿Cuáles son los valores medios de todos los parámetros de entrada o salida? Por ejemplo, ¿cuántas facturas manejará habitualmente un actor durante una sesión y cuál es el saldo de cuenta medio? Necesitamos estimaciones aproximadas de estas cosas porque así se optimizará la interfaz gráfica de usuario para ellas (incluso aunque tengamos que permitir un rango suficientemente grande).

Ejemplo Elementos de interfaz de usuario empleados en el caso de uso Pagar Factura

Para el caso de uso Pagar Factura ilustraremos cómo podemos encontrar los elementos de la interfaz de usuario que el actor necesita para trabajar sobre la pantalla, así como qué clase de guía podría necesitar el actor mientras trabaja.

El actor trabajará seguramente con elementos de la interfaz de usuario como Facturas (identificadas a partir de una clase del dominio o de una entidad del negocio). Factura es por tanto un elemento de la interfaz de usuario como se ilustra en la Figura 7.18. Nótese que las facturas tiene una fecha de vencimiento, una cantidad a pagar y una cuenta destino. Todos estos atributos son necesarios para un actor, que debe decidir si paga o no la factura.

Además, como el actor está decidiendo qué facturas pagar, él o ella puede querer consultar la cantidad de dinero en su cuenta para evitar quedarse en números rojos. Esto es un ejemplo de la clase de guía o información que un actor necesita. La interfaz de usuario debe pues mostrar las facturas según se planifican en el tiempo y cómo afectará el pago planificado de las facturas al saldo de la cuenta (como se indica en la asociación *cuenta comprador*, que se deriva de la asociación de la clase del dominio *comprador* del Capítulo 6). La cuenta es por eso otro aspecto de la interfaz de usuario. El saldo de la cuenta y su variación esperada en el tiempo a medida que se paguen las facturas se indican en la Figura 7.18 mediante el atributo *cuenta* y mediante la asociación *factura a pagar* entre los elementos de la interfaz de usuario Cuenta y Factura (*véase* la Figura 7.18).

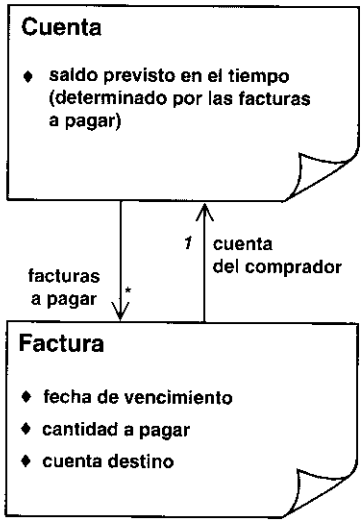


Figura 7.18. Elementos de la interfaz de usuario para Cuenta y Factura, con algunos de sus atributos indicados.

Una forma práctica de trabajo es representar los elementos de la interfaz de usuario mediante una nota adhesiva[4] (como se muestra en la Figura 7.18), pegarlas en una pizarra, y ordenarlas para ilustrar la apariencia de la interfaz de usuario. Seguidamente, los diseñadores de interfaces de usuario describen cómo pueden utilizar los actores estos elementos cuando trabajen con los casos de uso. La ventaja de utilizar notas adhesivas es que pueden representar la cantidad necesaria de datos. Además, Las notas adhesivas tampoco parecen permanentes —es fácil moverlas o simplemente eliminarlas—, lo que hace que el usuario se sienta cómodo proponiendo cambios.

De esta forma, los diseñadores de interfaz de usuario aseguran que cada caso de uso es accesible a través de sus elementos de las interfaces de usuario. No obstante, también deben asegurarse de que el conjunto de casos de uso accesibles para el actor tiene una interfaz de usuario bien integrada, fácil de utilizar y consistente.

Hasta aquí, sólo hemos estudiado las necesidades del actor respecto al interfaz de usuario. Veremos ahora cómo la interfaz física de usuario puede proporcionárnoslas.

7.4.4.2. Creación de un diseño y un prototipo físicos de interfaz de usuario

Los diseñadores de interfaz de usuario primero preparan unos esquemas aproximados de la configuración de elementos de las interfaces de usuario que constituyen interfaces de usuario útiles para los actores. Después, bosquejan elementos adicionales necesarios para combinar varios elementos de interfaces de usuario en interfaces de usuario completos. Estos elementos adicionales pueden incluir contenedores de los elementos de la interfaz de usuario (por ejemplo, carpetas), ventanas, herramientas y controles; *véase* la Figura 7.19. Estos esquemas pueden prepararse después (o a veces concurrentemente) del desarrollo de las notas adhesivas identificadas durante el diseño de la interfaz de usuario lógica.

[4] Para ser formales, cada "nota adhesiva" puede entenderse como un estereotipo sobre la clase en UML. No obstante, está fuera del alcance de este libro el entrar en detalles del modelo (de objetos) formal de los elementos de la interfaz de usuario.

Modelado esencial de casos de uso

Las descripciones detalladas de los casos de uso son un buen punto de partida cuando diseñamos la interfaz de usuario —algunas veces un punto de partida demasiado bueno—. El problema es que las descripciones a menudo contienen decisiones implícitas sobre las interfaces de usuario. Más tarde, cuando los diseñadores de interfaces de usuario sugieren interfaces adecuadas para los casos de uso, pueden estar limitadas por esas decisiones implícitas. No obstante, para crear la mejor interfaz de usuario par un caso de uso, debemos evitar caer en esta trampa. Por ejemplo, el caso de uso Pagar Factura comienza: "El comprador invoca el caso de uso comenzando a hojear las facturas recibidas...". Tal descripción puede engañar al diseñador de interfaces de usuario, que creará una interfaz de usuario que incluya una lista de las facturas recibidas para que el comprador pueda hojearlas. Pero puede que ésa no sea la mejor forma de estudiar las facturas recibidas. De hecho, el comprador puede encontrar más fácil hojear las facturas de una forma menos obvia, por ejemplo a través de iconos puestos horizontalmente de acuerdo a la *fecha de pago* y verticalmente de acuerdo a la *cantidad a pagar*.

Larry Constantine propone un remedio para el problema de las decisiones implícitas en las interfaces de usuario [14]. Sugiere que los especificadores de caso de uso primero preparen una descripción somera de casos de uso —los casos de uso esenciales— que no contengan ninguna decisión implícita de interfaz de usuario. El ejemplo anterior puede entonces reescribirse: "El comprador invoca el caso de uso para comenzar a estudiar las facturas recibidas...". Entonces los diseñadores de interfaces de usuario pueden utilizar estos casos de uso esenciales como entrada para crear la interfaz de usuario, sin estar limitados por ninguna decisión implícita.

Este ejemplo es una ilustración simplificada de qué debe conseguirse para destilar el significado esencial de una descripción de caso de uso. En realidad, hacerlo podría requerir más que reemplazar una palabra en una descripción. Lo importante es evitar tomar decisiones prematuras sobre:

- La técnica que se utilizará para presentar algunos elementos de la interfaz de usuario, como utilizar una lista o un campo de texto.
- La secuencia de entradas del actor, como introducir un atributo antes que otro.
- Los dispositivos necesarios para la entrada y salida, como puede ser la utilización del ratón para seleccionar algo o el monitor para presentar cualquier cosa.

Después, si procede, los especificadores de caso de uso pueden hacer una segunda pasada sobre la descripción de los casos de uso para añadir los detalles que quedaron fuera en la descripción con el estilo esencial.

Ejemplo Diseño y prototipo físicos de una interfaz de usuario

El diseñador de interfaces de usuario bosqueja el siguiente diseño físico de cómo el saldo de cuenta visualizado se ve afectado en el tiempo por las facturas planificadas. Las facturas se muestran como trapecios blancos que reducirán el saldo de la cuenta cuando llegue el momento de pagarlas. El diseñador de interfaces de usuario elige trapezoides porque son suficientemente anchos para ser visibles y seleccionables, pero también tienen una punta afilada que se utiliza para indicar cuándo tiene lugar el pago. Las facturas planificadas, como las del alquiler, provocarán una

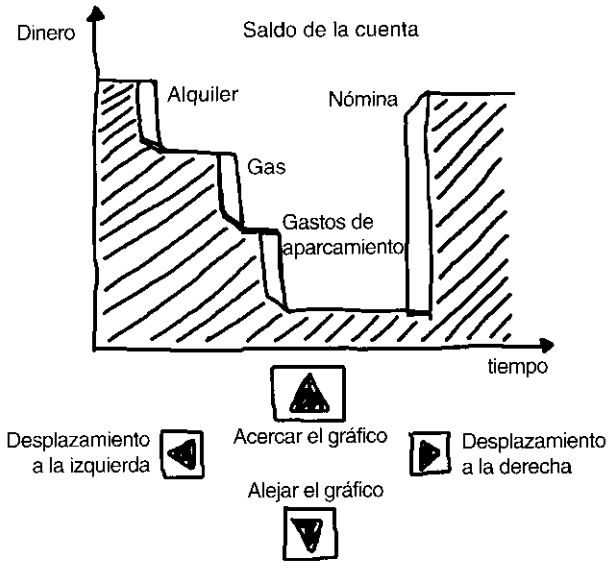


Figura 7.19. Una interfaz de usuario propuesta para los elementos de interfaz relacionados Cuenta y Factura. La interfaz muestra cómo incrementan o reducen la cuenta los pagos de facturas e ingresos de depósitos. Los botones de desplazamiento izquierdo y derecho mueven la ventana de tiempo en la que el actor puede estudiar el saldo de la cuenta. Los botones de alejar y acercar cambian la escala del diagrama.

reducción de saldo de la cuenta en la fecha de vencimiento, como se ilustra en la Figura 7.19. De igual forma, el dibujo muestra cómo se añade el dinero a la cuenta, quizás cuando el usuario ingresa su nómina. La figura también indica los controles de la interfaz de usuario, como los botones de desplazamiento y zoom.

Ahora estamos preparados para construir prototipos ejecutables de las configuraciones más importantes de elementos de interfaz de usuario. Estos prototipos pueden construirse con una herramienta de prototipado rápido.

Puede haber varios prototipos, quizá uno para cada actor, para verificar que cada actor puede ejecutar el caso de uso que necesita. El esfuerzo de prototipado debe ser proporcional al valor de retorno esperado. Desarrollamos prototipos ejecutables de IGU cuando tenemos mucho que ganar en facilidad de uso (por ejemplo, un prototipo para los actores más importantes), y utilizamos bocetos en papel cuando no tengamos tanto que ganar.

La validación de las interfaces de usuarios a través de revisiones de prototipos y esquemas en los primeros momentos pueden prevenir muchos errores que serán más caros de corregir después. Los prototipos también pueden revisarse superficialmente en la descripción de casos de uso, y permitir que se corrijan después de que los casos de uso pasen a su diseñado. Los revisores deben verificar que cada interfaz de usuario

- Permite que el actor navegue de forma adecuada.
- Proporciona una apariencia agradable y una forma consistente de trabajo con la interfaz de usuario, como por ejemplo, en cuanto a orden de tabulación y teclas rápidas.

- Cumple con estándares relevantes como el color, tamaño de los botones y situación de las barras de herramientas.

Nótese que la implementación de las interfaces de usuario reales (al contrario que el prototipo que hemos desarrollado aquí) se construye en paralelo con el resto del sistema, esto es, durante los flujos de trabajo de análisis, diseño e implementación. El prototipo de interfaz de usuario que hemos desarrollado aquí deberá entonces servir como especificación de la interfaz de usuario. Esta especificación será realizada en términos de componentes de calidad profesional.

7.4.5. Actividad: estructurar el modelo de casos de uso

El modelo de casos de uso se estructura para:

- Extraer descripciones de funcionalidad (de casos de uso) generales y compartidas que pueden ser utilizadas por descripciones (de casos de uso) más específicas.
- Extraer descripciones de funcionalidad (de casos de uso) adicionales u opcionales que pueden extender descripciones (de casos de uso) más específicas.

Antes de que tenga lugar esta actividad, el analista de sistemas ha identificado los actores y los casos de uso, dibujándolos en diagramas, y explicando el modelo de casos de uso como un todo. Los especificadores de casos de uso han desarrollado una descripción detallada de cada caso de uso. En este punto el analista de sistemas puede reestructurar el conjunto completo de casos de uso para hacer que el modelo sea más fácil de entender y de trabajar con él (Figura 7.20). El analista debe buscar comportamientos compartidos y extensiones.

7.4.5.1. Identificación de descripciones de funcionalidad compartidas

A medida que identificamos y esbozamos las acciones de cada caso de uso, debemos también ir buscando acciones o parte de acciones comunes o compartidas por varios casos de uso. Con

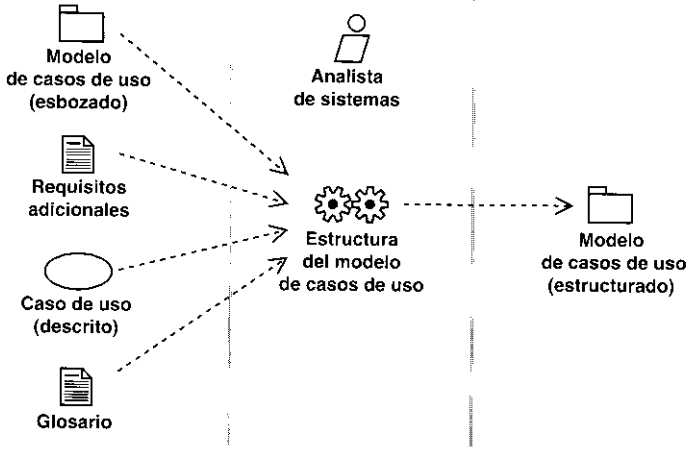


Figura 7.20. Las entradas y los resultados de encontrar generalizaciones en el modelo de casos de uso.

el fin de reducir la redundancia, esta compartición puede extraerse y describirse en un caso de uso separado que puede ser después reutilizado por el caso de uso original. Mostramos la relación de reutilización mediante una generalización (llamada *relación de uso* en [7]). La generalización entre casos de uso es una clase de herencia, en la que las instancias de los casos de uso generalizados pueden ejecutar todo el comportamiento descrito en el caso de uso generalizador. Dicho de otra forma, una generalización de un caso de uso A hacia un caso de uso B indica que una instancia del caso de uso A incluirá también el comportamiento especificado en B.

Ejemplo Generalización entre casos de uso

Recordemos la Figura 7.12, explicada anteriormente en este capítulo, en la cual generalizamos el caso de uso Pagar Factura con el caso de uso Ejecutar Transacción. La secuencia de acciones descrita en el caso de uso Ejecutar Transacción es, por tanto, heredada en la secuencia descrita en Pagar Factura (Figura 7.21).

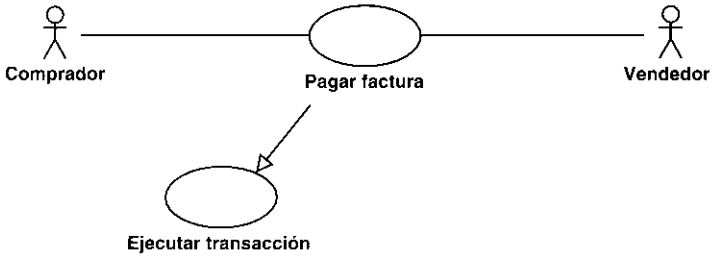


Figura 7.21. La relación de generalización entre los casos de uso Pagar Factura y Ejecutar Transacción.

La generalización se emplea para simplificar la forma de trabajo y la comprensión del modelo de casos de uso y para reutilizar casos de uso "semifabricados" cuando reunimos casos de uso terminados requeridos por el usuario. Cada caso de uso terminado se denomina *caso de uso concreto*. Los inicia un actor, y sus instancias constituyen una secuencia de acciones completa ejecutada por el sistema. El caso de uso "semifabricado" existe solamente para que otros casos de uso lo reutilicen y se les llama *casos de uso abstractos*. Un caso de uso abstracto no puede instanciarse por sí mismo, pero una instancia de un caso de uso concreto también exhibe el comportamiento especificado por un caso de uso abstracto que lo (re)utiliza. Para entendernos, llamamos a esta instancia el caso de uso "real" que el actor(es) percibe cuando interactúa con el sistema.

Ejemplo Caso de uso "real"

Podemos conceptualizar un caso de uso "real", como se muestra en la Figura 7.22, donde Ejecutar Transacción generaliza a Pagar Factura.

Este caso de uso "real" es el resultado que obtenemos después de la aplicación de la generalización a los dos casos de uso, uno concreto y otro abstracto. Este caso de uso real representa el comportamiento de la instancia del caso de uso en la que se percibe la interacción de un actor con el sistema. Si

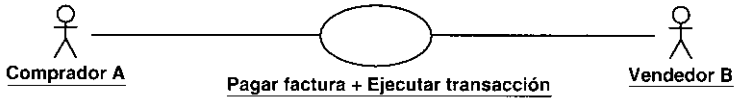


Figura 7.22. La instancia del caso de uso "real" formada por los casos de uso Pagar Factura y Ejecutar Transacción, tal y como la perciben las instancias de actor Comprador A y Vendedor B.

el modelo contiene más casos de uso concretos generalizados del caso de uso Ejecutar Transacción, entonces habría más casos de uso reales. Estos casos de uso reales tendrían especificaciones solapadas, donde el solapamiento consiste en lo especificado en el caso de uso Ejecutar Transacción.

Obsérvese que este ejemplo no expone la verdad completa, ya que hay también un caso de uso (Pagar Cargos Saldo Deudor) que extiende del caso de uso Pagar Factura, ofreciendo de esta forma otros casos de uso reales. Prestaremos atención a esto en la siguiente sección.

7.4.5.2. Identificación de descripciones de funcionalidad adicionales y opcionales

La otra relación entre casos de uso en la *relación de extensión* (*extend relationship*) [7]. Esta relación modela la adición de una secuencia de acciones a un caso de uso. Una extensión se comporta como si fuera algo que se añade a la descripción original de un caso de uso. Dicho de otra forma, una relación de extensión desde un caso de uso A hacia un caso de uso B indica que una instancia del caso de uso B puede incluir (para especificar condiciones específicas en las extensiones) el comportamiento especificado por A. El comportamiento especificado por varias extensiones de un único caso de uso destino puede darse dentro una única instancia de caso de uso.

La relación de extensión incluye tanto una condición para la extensión como una referencia a un punto de extensión en el caso de uso destino, es decir, una posición en el caso de uso donde se pueden hacer las adiciones. Una vez que una instancia de un caso de uso (destino) llega al punto de extensión al cual se refiere una relación de extensión, se evalúa la condición de la relación. Si la condición se cumple, la secuencia seguida por la instancia del caso de uso incluye la secuencia del caso de uso extendido.

Podemos decir que los casos de uso reales, que son casos de uso en toda regla, se obtienen aplicando las relaciones de generalización y extensión para utilizar casos de uso en el modelo.

Ejemplo **Relación de extensión entre casos de uso**

Recordamos la Figura 7.12 situada anteriormente en este capítulo y el ejemplo expuesto en la Sección 7.4.1.4, "Descripción del modelo de casos de uso en su totalidad", en la cual el caso de uso Pagar Factura es extendido por el caso de uso Pagar Cargos Saldo Deudor. La secuencia de acciones descritas en el caso de uso Pagar Cargos Saldo Deudor (Figura 7.23) se inserta de esta forma en la secuencia descrita en Pagar Factura, si se da un descubierto en la cuenta (ésta es la condición para la extensión).

Nótese que con la relación de extensión podemos ir un paso más allá cuando discutimos de los casos de uso percibidos por el actor. Aplicando la relación de extensión desde el caso de uso Pagar Cargos Saldo Deudor al caso de uso destino (es decir, Pagar Facturas generalizado de Ejecutar Transacción), obtenemos otro caso de uso real que es la fusión de los tres casos de uso (Figura 7.24).

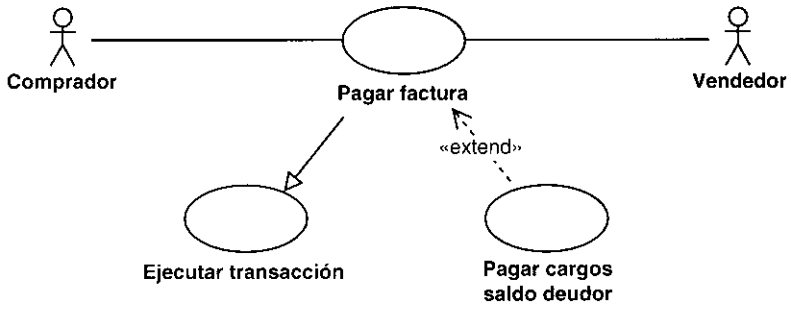


Figura 7.23. La relación de extensión entre los casos de uso Pagar Factura y Pagar Cargos Saldo Deudor.

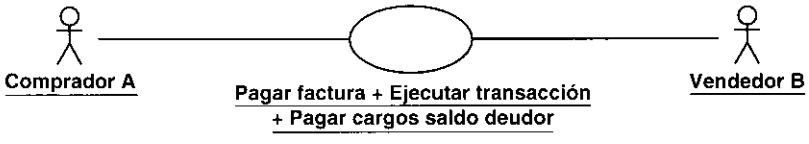


Figura 7.24. La instancia del caso de uso real formado por los casos de uso Pagar Factura y Ejecutar Transacción extendidos mediante el caso de uso Pagar Cargos Saldo Deudor, tal y como lo perciben las instancias de los actores Comprador A y Vendedor B.

Los casos de uso reales son aquellos que aportan valor a los usuarios, así, el criterio para obtener buenos casos de uso que se mencionó en la Sección 7.4.1, "Encontrar actores y casos de uso" (un caso de uso obtiene valores de resultados observables para un actor en particular) es relevante solamente para casos de uso reales. Así que debemos identificar criterios separados para realizar buenos casos de uso concretos, buenos casos de uso abstractos y buenas extensiones de casos de uso. Los casos de uso concretos no deben describir comportamientos (significantivos) que son compartidos con otros casos de uso concretos. Un caso de uso abstracto hace más fácil de especificar los casos de uso concretos ofreciendo comportamiento reutilizable y compartido. Una extensión de una caso de uso especifica comportamiento adicional para otros casos de uso, independientemente de si su comportamiento es opcional u obligatorio.

Así, para comprender las relaciones de generalización y extensión, hemos introducido la noción de caso de uso real. Una vez que hemos identificado los casos de uso concretos, los abstractos y las extensiones de los casos de uso, podemos combinarlos para obtener un caso de uso real. No obstante, cuando comenzamos el modelado de un nuevo sistema, normalmente seguimos el camino contrario, comenzando por los casos de uso reales e identificando comportamientos comunes, que distinguen a los casos de uso concretos de los casos de uso abstractos, y comportamientos adicionales, que tratamos como extensiones de otros casos de uso.

Véase [7] par una discusión más profunda de las relaciones de generalización (que también llamaremos relaciones de uso, *uses relationships*) y extensión, y de cuándo utilizarlas.

7.4.5.3. Identificación de otras relaciones entre casos de uso

También existen otras relacione entre casos de uso, como las relaciones de inclusión (*include relationship*) [12]. Podemos imaginar esta relación, para simplificar, como una relación inversa a la de

extensión, que proporciona una extensión explícita e incondicional a un caso de uso. Es más, cuando incluimos un caso de uso, la secuencia de comportamiento y los atributos del caso de uso incluido se encapsulan y no pueden modificarse o accederse —solamente puede utilizarse el resultado (o función) de un caso de uso incluido—; esto es una diferencia en comparación con la utilización de relaciones de generalización. No obstante, en este libro no entraremos en muchos detalles sobre este tipo de relación. Pero sí decir algunas palabras de precaución:

- La estructura de los casos de uso y de sus relaciones debe reflejar los casos de uso reales (como ya hemos tratado) tanto como sea posible. Cuanto más diverjan estas estructuras de los casos de uso reales, más complicado será comprender los casos de uso y sus propósitos, no solamente para terceras partes externas como usuarios y clientes, sino también para los mismos desarrolladores.
- Cada caso de uso individual necesita ser tratado como un artefacto separado. Alguien, es decir, un especificador de casos de uso, debe ser responsable de su descripción; y en los flujos de trabajos subsiguientes —análisis y diseño— debe representarse al caso de uso mediante realizaciones de casos de uso separadas (como veremos en los Capítulos 8 y 9). Por este motivo los casos de uso no deberían ser demasiados o demasiado pequeños, conllevando así una sobrecarga de gestión significativa.
- Evite descomponer funcionalmente los casos de uso en el modelo de casos de uso. Esto se hace mucho mejor mediante el refinamiento de cada caso de uso en el modelo de análisis. Como veremos en el Capítulo 8, esto se debe a que la funcionalidad que definen los casos de uso se descompondrá a su vez, de una forma orientada a objetos, en colaboraciones de objetos de análisis conceptuales. Esta descomposición nos proporcionará una comprensión en profundidad de los requisitos, si fuese necesario.

7.5. Resumen del flujo de trabajo de los requisitos

En este capítulo y en el anterior, hemos descrito cómo capturar los requisitos de un sistema en forma de:

- Un modelo del negocio o un modelo del dominio para establecer el contexto del sistema.
- Un modelo de casos de uso que capture los requisitos funcionales, y los no funcionales que son específicos de casos de uso concretos. El modelo de casos de uso se realiza mediante una descripción general, un conjunto de diagramas, y una descripción detallada de cada caso de uso.
- Un conjunto de esbozos de interfaces de usuario y de prototipos para cada actor, que representan el diseño de las interfaces de usuario.
- Una especificación de requisitos adicionales para los requisitos que son genéricos y no específicos de un caso de uso en particular.

Estos resultados son un muy buen punto de partida para los siguientes flujos de trabajo: análisis, diseño, implementación y prueba. Los casos de uso dirigirán el trabajo a lo largo de estos flujos de trabajo iteración por iteración. Para cada caso de uso del modelo de casos de uso identificaremos una realización de caso de uso correspondiente en las fases de análisis y diseño y un conjunto de casos de prueba en la fase de pruebas. Por tanto, los casos de uso enlazarán directamente los diferentes flujos de trabajo.

En el Capítulo 8 pasamos al siguiente paso de nuestra cadena de flujos de trabajo —análisis— donde reformularemos los casos de uso como objetos para obtener una mejor comprensión de los requisitos y también para prepararnos para el diseño e implementación del sistema.

7.6. Referencias

[1] Ahlqvist Stefan and Jonsson Patrik, "Techniques for systematic design of graphical user interfaces based on use cases", *Proceedings OOPSLA 96.*

[2] Grady Booch, James Rumbaugh, and Ivar Jacobson, *Unified Modeling Language User Guide.* Reading, MA: Addison-Wesley, 1998.

[3] Alistair Cockburn, "Structuring use cases with goals", *Report on Analysis and Design (ROAD),* 1997.

[4] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard, *Object-Oriented Software Engineering: A Use-Case–Driven Approach.* Menlo Park, CA: Addison-Wesley, 1992. (Revised fourth printing, 1993).

[5] Ivar Jacobson, Maria Ericsson, and Agneta Jacobson, *The Object Advantage—Business Process Reengineering with Object Technology.* Menlo Park, CA: Addison-Wesley, 1994.

[6] Jacobson I., "Basic use case modeling", *Report on Analysis and Design (ROAD),* July–August, vol. 1 no. 2, 1994.

[7] Jacobson I., "Basic use case modeling (continued)", *Report on Analysis and Design (ROAD),* vol. 1 no. 3, September–October 1994.

[8] Ivar Jacobson and Magnus Christerson, "Modeling with use cases—A growing consensus on use cases". *Journal of Object-Oriented Programming,* March–April 1995

[9] Jacobson I., K Palmqvist, and S. Dyrhage, "Systems of interconnected systems", *Report on Analysis and Design (ROAD),* May–June 1995.

[10] Ivar Jacobson, "Modeling with use cases—Formalizing use-case modeling", *Journal of Object-Oriented Programming,* June 1995.

[11] James Rumbaugh, Ivar Jacobson, and Grady Booch, *Unified Modeling Language Reference Manual,* Reading, MA: Addison-Wesley, 1998.

[12] OMG Unified Modeling Language Specification. Object Management Group, Framingham, MA, 1998. Internet: www.omg.org.

[13] Ivar Jacobson, Martin Griss, and Patrik Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success.* Reading, MA: Addison-Wesley, 1997.

[14] L.L. Constantine and L.A.D. Lockwood, *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design.* Reading, MA: Addison-Wesley, 1999.

[15] CCITT, Specification and Description Language (SDL), Recommendation Z.100. Geneva, 1988.

[16] John Carrol, *Scenario-Based Design,* New York: John Wiley & Sons, 1995.

[17] David Harel, Michal Politi, *Modeling Reactive Systems With Statecharts: The STATEMATE Approach*, New York: McGraw-Hill, 1998.

Capítulo 8

Análisis

8.1. Introducción

Durante el análisis, analizamos los requisitos que se describieron en la captura de requisitos, refinándolos y estructurándolos. El objetivo de hacerlo es conseguir una comprensión más precisa de los requisitos y una descripción de los mismos que sea fácil de mantener y que nos ayude a estructurar el sistema entero —incluyendo su arquitectura.

Antes de explicar exactamente lo que esto significa, permítasenos reflexionar un poco sobre los resultados de la captura de requisitos. Recordemos que la regla número uno de la captura de requisitos es utilizar el lenguaje del cliente (*véase* Sección 6.2). Creemos también, como explicamos en el Capítulo 7, que los casos de uso son una buena base para este lenguaje. Pero incluso si conseguimos llegar a un acuerdo con el cliente sobre lo que debería hacer el sistema, *es probable que aún queden aspectos sin resolver relativos a los requisitos del sistema*. Éste es el precio que hay que pagar por el uso del lenguaje intuitivo pero impreciso del cliente durante la captura de requisitos. Para arrojar alguna luz sobre qué "temas sin resolver" pueden haber quedado, relativos a los requisitos del sistema descritos en la captura de requisitos, recuérdese que para comunicar de manera eficiente las funciones del sistema al cliente:

1. *Los casos de uso deben mantenerse tan independientes unos de otros como sea posible*. Esto se consigue no quedándose atrapado en detalles relativos a las interfe-

rencias, concurrencia, y conflictos entre los casos de uso cuando éstos, por ejemplo, compiten por recursos compartidos que son internos al sistema (*véase* la Sección 7.2.3). Por ejemplo, los casos de uso Ingresar y Sacar Dinero acceden ambos a la misma cuenta del cliente. O bien puede darse un conflicto si un actor combina casos de uso que dan como resultado un comportamiento no deseado, como cuando un abonado a una línea telefónica utiliza un caso de uso Llamada de Despertador, seguido de un caso de uso Redirigir Llamadas Entrantes, para solicitar una llamada de despertador para otro abonado. Por tanto, los aspectos relativos a la interferencia, la concurrencia, y los conflictos entre casos de uso pueden quedar sin resolver en la captura de requisitos.

2. *Los casos de uso deben describirse utilizando el lenguaje del cliente.* Esto se consigue fundamentalmente mediante el uso del lenguaje natural en las descripciones de casos de uso, y siendo cuidadosos al utilizar notaciones más formales, como diagramas de estado, actividad o interacción (*véase* la Sección 7.4.3.2). Sin embargo, con la utilización solamente de lenguaje natural, perdemos poder expresivo, y en la captura de requisitos pueden quedar sin tratar —o quedar sólo vagamente descritos— muchos detalles que podríamos haber precisado con notaciones más formales.

3. *Debe estructurarse cada caso de uso para que forme una especificación de funcionalidad completa e intuitiva.* Esto se consigue estructurando los casos de uso (y por tanto, los requisitos) de manera que reflejen intuitivamente los casos de uso "reales" que el sistema proporciona. Por ejemplo, no deberían estructurarse en casos de uso pequeños, abstractos y no intuitivos para reducir las redundancias. Aunque puede hacerse así, debemos llegar a un equilibrio entre comprensibilidad y mantenibilidad en las descripciones de casos de uso (*véase* la Sección 7.4.5.3). Por tanto, los aspectos relativos a esas redundancias entre los requisitos descritos puede que queden sin resolver durante la captura de requisitos.

Dados esos temas sin tratar, el propósito fundamental del análisis es resolverlos analizando los requisitos con mayor profundidad, pero con la gran diferencia (comparado con la captura de requisitos) de que puede utilizarse *el lenguaje de los desarrolladores* para describir los resultados.

En consecuencia, en el análisis podemos razonar más sobre los aspectos internos del sistema, y por tanto resolver aspectos relativos a la interferencia de casos de uso y demás (señalados en el punto 1 anterior). También podemos utilizar un lenguaje más formal para apuntar detalles relativos a los requisitos del sistema (*véase* el punto 2 anterior). Llamaremos a esto "refinar los requisitos".

Además, en el análisis podemos estructurar los requisitos de manera que nos facilite su comprensión, su preparación, su modificación, y, en general, su mantenimiento (el punto 3 anterior). Esta estructura (basada en clases de análisis y paquetes) es independiente de la estructura que se dio a los requisitos (basada en casos de uso). Sin embargo, existe una trazabilidad directa entre esas distintas estructuras, de forma que podemos hacer la traza de diferentes descripciones —en diferentes niveles de detalle— del mismo requisito y mantener su consistencia mutua con facilidad. De hecho, esta trazabilidad directa se define entre casos de uso del modelo de casos de uso y realizaciones de casos de uso en el modelo de análisis; esto lo explicaremos en detalle más adelante en este capítulo (*véase* también la Tabla 8.1).

Tabla 8.1. Breve comparación del modelo de casos de uso con el modelo de análisis

Modelo de casos de uso	Modelo de análisis
Descrito con el lenguaje del cliente.	Descrito con el lenguaje del desarrollador.
Vista externa del sistema.	Vista interna del sistema.
Estructurado por los casos de uso; proporciona la estructura a la vista externa.	Estructurado por clases y paquetes estereotipados; proporciona la estructura a la vista interna.
Utilizado fundamentalmente como contrato entre el cliente y los desarrolladores sobre qué debería y qué no debería hacer el sistema.	Utilizado fundamentalmente por los desarrolladores para comprender cómo debería darse forma al sistema, es decir, cómo debería ser diseñado e implementado.
Puede contener redundancias, inconsistencias, etc., entre requisitos.	No debería contener redundancias, inconsistencias, etc., entre requisitos.
Captura la funcionalidad del sistema, incluida la funcionalidad significativa para la arquitectura.	Esboza cómo llevar a cabo la funcionalidad dentro del sistema, incluida la funcionalidad significativa para la arquitectura; sirve como una primera aproximación al diseño.
Define casos de uso que se analizarán con más profundidad en el modelo de análisis.	Define realizaciones de casos de uso, y cada una de ellas representa el análisis de un caso de uso del modelo de casos de uso.

Por último, la estructura de los requisitos que proporciona el análisis también sirve como entrada fundamental para dar forma al sistema en su totalidad (incluida su arquitectura); esto es debido a que queremos construir el sistema como un todo mantenible, y no sólo describir sus requisitos.

En este capítulo presentaremos una explicación más detallada de qué entendemos por análisis y por refinamiento y estructuración de requisitos. Empezaremos con una breve "puesta en situación" del análisis (Sección 8.2) y después describiremos el papel del análisis en las diferentes fases del ciclo de vida del software (Sección 8.3). Después, presentaremos los artefactos (Sección 8.4) y los trabajadores (Sección 8.5) implicados en el análisis (*véase* la Figura 8.1). Para terminar describiremos el flujo de trabajo del análisis (Sección 8.6).

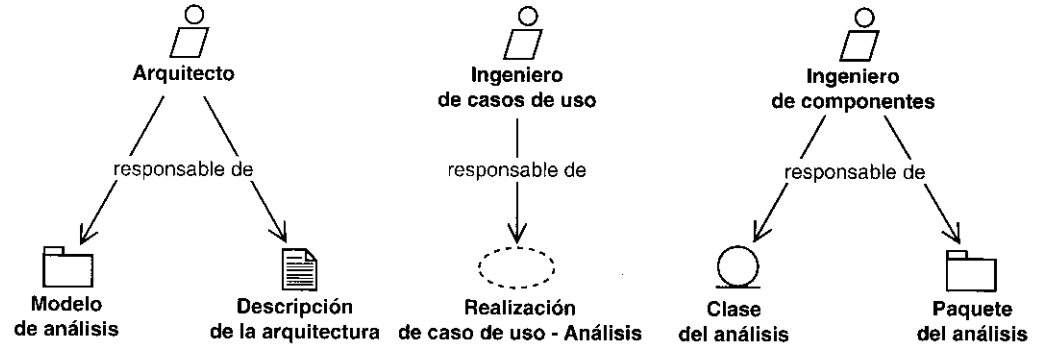


Figura 8.1. Los trabajadores y artefactos implicados en el análisis.

8.2. El análisis en pocas palabras

El lenguaje que utilizamos en el análisis se basa en un modelo de objetos conceptual, que llamamos *modelo de análisis*. El modelo de análisis nos ayuda a refinar los requisitos según las líneas que ya hemos mencionado antes (Sección 8.1) y nos permite razonar sobre los aspectos internos del sistema, incluidos sus recursos compartidos internos. De hecho, un recurso interno puede representarse como un objeto en el modelo de análisis, como por ejemplo la cuenta de cliente que es accedida por los casos de uso Ingresar y Sacar. Además, el modelo de análisis nos ofrece un mayor poder expresivo y una mayor formalización, como por ejemplo, la que nos proporcionan los diagramas de interacción que se utilizan para describir los aspectos dinámicos del sistema.

El modelo de análisis también nos ayuda a estructurar los requisitos como se ha explicado anteriormente (Sección 8.1) y nos proporciona una estructura centrada en el mantenimiento, en aspectos tales como la flexibilidad ante los cambios y la reutilización. (Más adelante en este capítulo trataremos principios que harán el modelo de análisis flexible ante esos cambios y que contendrán elementos reutilizables). Esta estructura no sólo es útil para el mantenimiento de los requisitos como tales, sino que también se utiliza como entrada en las actividades de diseño y de implementación (como explicaremos en los Capítulos 9 y 10). Tratamos de preservar esta estructura a medida que damos forma al sistema y tomamos las decisiones sobre su diseño e implementación. Dicho todo esto, el modelo de análisis puede considerarse una primera aproximación al modelo de diseño, aunque es un modelo por sí mismo. Mediante la conservación de la estructura del modelo de análisis durante el diseño, obtenemos un sistema que debería ser también mantenible como un todo: será flexible a los cambios en los requisitos, e incluirá elementos que podrán ser reutilizados cuando se construyan sistemas parecidos.

Sin embargo, es importante hacer notar aquí que el modelo de análisis hace abstracciones y evita resolver algunos problemas y tratar algunos requisitos que pensamos que es mejor posponer al diseño y a la implementación (Apéndice C; *véase* también la Sección 8.2.1). Debido a esto, no siempre se puede conservar la estructura proporcionada por el análisis, sino que se debe negociar y comprometer durante el diseño y la implementación, como veremos en los Capítulos 9 y 10. La razón por la cual esta "conservación de la estructura" no siempre tiene lugar en la práctica es sencillamente que el diseño debe considerar la plataforma de implementación: lenguaje de programación, sistemas operativos, marcos de trabajo prefabricados, sistemas heredados, y demás. Por economía, puede conseguirse una arquitectura mejor mediante la modificación de la estructura del modelo de análisis durante la transición al modelo de diseño, a ir dando forma al sistema.

8.2.1. Por qué el análisis no es diseño ni implementación

Podría preguntarnos ahora por qué no analizamos los requisitos al mismo tiempo que diseñamos e implementamos el sistema. Nuestra respuesta es que el *diseño y la implementación son mucho más que el análisis (refinamiento y estructuración de los requisitos), por lo que se requiere una separación de intereses.* En el diseño, debemos moldear el sistema y encontrar su forma, incluyendo su arquitectura; una forma que dé vida a todos los requisitos incorporados en el sistema, una forma que incluya componentes de código que se compilan e integran en versiones ejecutables del sistema; y con suerte una forma que podamos mantener a largo plazo —que se mantenga firme bajo las presiones del tiempo, los cambios, y la evolución; una forma con integridad

En el diseño tenemos por tanto que tomar decisiones relativas a cómo debería el sistema tratar, por ejemplo, los requisitos de rendimiento y de distribución, y contestar preguntas tales como: "¿Cómo podemos optimizar este procedimiento para que su ejecución dure un máximo de 5 milisegundos?"; y "¿cómo podemos distribuir este código en ese nodo de la red sin sobrecargar el tráfico de la misma?". Hay muchos más temas parecidos que deben tratarse en el diseño, por ejemplo, cómo explotar de manera eficiente componentes comprados como bases de datos y object request brokers, y cómo integrarlos en la arquitectura del sistema, cómo utilizar el lenguaje de programación de forma adecuada, y demás. No ofreceremos aquí una lista completa de todos los temas adicionales que aparecen durante el diseño y la implementación, en cambio volveremos sobre ello en los Capítulos 9 y 10. Esperamos haber explicado lo que tratábamos de explicar, a saber, que *el diseño y la implementación son mucho más que simplemente el analizar los requisitos refinándolos y estructurándolos; el diseño y la implementación se preocupan en realidad de dar forma al sistema de manera que dé vida a todos los requisitos —incluidos todos los requisitos no funcionales— que incorpora.* Para dar alguna idea sobre las abstracciones hechas en el análisis, en comparación con la riqueza de detalle del modelo de diseño, son comunes entre ambos una proporción de 1 a 5 de sus elementos de modelo.

Dicho todo esto, pensamos simplemente que antes de comenzar a diseñar e implementar, deberíamos contar con una comprensión precisa y detallada de los requisitos —un nivel de detalle que no preocupa al cliente (en la mayoría de los casos)—. Además, si contamos con una estructura de los requisitos que puede utilizarse como entrada para dar forma al sistema, mejor. Todo esto se consigue mediante el análisis.

Dicho simplemente, llevando a cabo el análisis conseguimos una separación de intereses que prepara y simplifica las subsiguientes actividades de diseño e implementación, delimitando los temas que deben resolverse y las decisiones que deben tomarse en esas actividades. Además, mediante esta separación de intereses, los desarrolladores pueden "afrontar la escalada" al comienzo del esfuerzo de desarrollo de software, y por tanto evitar la parálisis que puede ocurrir cuando intentamos resolver demasiados problemas a la vez —incluyendo problemas que puede que no se hayan resuelto en absoluto debido a que los requisitos eran vagos y ¡no se comprendían correctamente!

8.2.2. El objeto del análisis: resumen

Analizar los requisitos en la forma de un modelo de análisis es importante por varios motivos, como ya hemos explicado:

- Un modelo de análisis ofrece una especificación más precisa de los requisitos que la que tenemos como resultado de la captura de requisitos, incluyendo al modelo de casos de uso.
- Un modelo de análisis se describe utilizando el lenguaje de los desarrolladores, y puede por tanto introducir un mayor formalismo y ser utilizado para razonar sobre los funcionamientos internos del sistema.
- Un modelo de análisis estructura los requisitos de un modo que facilita su comprensión, su preparación, su modificación, y en general, su mantenimiento.
- Un modelo de análisis puede considerarse como una primera aproximación al modelo de diseño (aunque es un modelo por sí mismo), y es por tanto una entrada fundamental cuando se da forma al sistema en el diseño y en la implementación. Esto se debe a que debería ser mantenible el sistema en su conjunto, y no sólo la descripción de sus requisitos.

8.2.3. Ejemplos concretos de cuándo hacer análisis

Además de lo que ya hemos dicho, ofrecemos ejemplos más completos de cuándo utilizar el análisis y de cómo explotar su resultado (el modelo de análisis):

- Mediante la realización separada del análisis, en lugar de llevarlo a cabo como parte integrada en el diseño y la implementación, podemos analizar sin grandes costes una gran parte del sistema. Y podemos después utilizar el resultado para planificar el trabajo de diseño e implementación subsiguiente; quizá en varios incrementos sucesivos, donde cada incremento sólo diseña e implementa un *pequeña parte* del sistema; o quizá en varios incrementos concurrentes, posiblemente diseñados e implementados por equipos de desarrollo distribuidos geográficamente. Sin los resultados del análisis, la identificación y planificación de estos incrementos puede ser más difícil de hacer.

- El análisis proporciona una visión general del sistema que puede ser más difícil de obtener mediante el estudio de los resultados del diseño y la implementación, debido a que contienen demasiados detalles (recuérdese la proporción 1 a 5 que se explicó en la Sección 8.2.1). Una visión general como esa puede ser muy valiosa para recién llegados al sistema o para desarrolladores que en general mantienen el sistema. Por ejemplo, una organización ha desarrollado un gran sistema (uno con miles de subsistemas de servicio) utilizando principios parecidos a los que describimos en los Capítulos 3 y 4. Incluimos, *a posteriori*, un modelo de análisis para obtener una mejor comprensión del sistema ya desarrollado. El directivo sintetizó su experiencia con las siguiente palabras: "Gracias al modelo de análisis, ahora somos capaces de formar arquitectos del sistema en dos años en lugar de en cinco." Para un sistema más pequeño, el periodo de formación podría medirse en meses en lugar de en años, pero las proporciones deberían ser las mismas.

- Algunas partes del sistema tienen diseños y/o implementaciones alternativas. Por ejemplo, sistemas críticos como los de control aéreo o ferroviario, pueden constar de varios programas distintos que calculan de manera concurrente las mismas operaciones, y sólo se puede llevar a cabo una maniobra importante en el caso de que esos cálculos arrojen los mismos resultados. Otro ejemplo sería cuando un cliente quiere que dos o más fabricantes —o empresas subcontratadas— le proporcionen software en diferentes ubicaciones; el cliente quiere que dos casas de software competidoras le hagan ofertas basadas en la misma especificación. En general, éste es el caso cuando una parte del sistema se implementa más de una vez utilizando diferentes tecnologías, tales como lenguajes de programación o componentes, que se ejecutan en distintas plataformas. El modelo de análisis puede en este caso proporcionar una vista conceptual, precisa y unificadora de esas implementaciones alternativas. En este caso, el modelo de análisis debería, obviamente, mantenerse a lo largo de la vida del sistema.

- El sistema se construye utilizando un sistema heredado complejo. La reingeniería de este sistema heredado, o de una parte de él, puede hacerse en términos del modelo de análisis de manera que los desarrolladores pueden comprender el sistema heredado sin tener que profundizar en los detalles de su diseño e implementación, y construir el nuevo sistema utilizando el heredado como un bloque de construcción reutilizable. Puede que la reingeniería completa del diseño y la implementación de un sistema heredado de ese tipo sea muy complicada, cara, y de poca utilidad —sobre todo si el sistema heredado no ha de cambiarse y está implementado con tecnologías obsoletas.

8.3. El papel del análisis en el ciclo de vida del software

Las iteraciones iniciales de la elaboración se centran en el análisis (*véase* la Figura 8.2). Eso contribuye a obtener una arquitectura estable y sólida y facilita una comprensión en profundidad de los requisitos. Más adelante, al término de la fase de elaboración y durante la construcción, cuando la arquitectura es estable y se comprenden los requisitos, el énfasis pasa en cambio al diseño y a la implementación.

El propósito y objetivo del análisis debe alcanzarse de algún modo en todo proyecto. Pero la manera exacta de ver y de emplear el análisis puede diferir de un proyecto a otro, y nosotros distinguimos tres variantes básicas:

1. El proyecto utiliza el modelo de análisis (como se describirá más adelante en este capítulo) para describir los resultados del análisis, y mantiene la consistencia de este modelo a lo largo de todo el ciclo de vida del software. Esto puede hacerse, por ejemplo, de manera continua —en cada iteración en el proyecto— para obtener algunos de los beneficios esquematizados en la Sección 8.2.3.

2. El proyecto utiliza el modelo de análisis para describir los resultados del análisis pero considera a este modelo como una herramienta transitoria e intermedia —quizás de más interés durante la fase de elaboración—. Más adelante, cuando el diseño y la implementación están en marcha durante la fase de construcción, se deja de actualizar el análisis. En su lugar, cualquier "tema de análisis" que aún quede se resuelve como parte integrada dentro del trabajo de diseño en el modelo de diseño resultante (al cual volveremos nuestra atención en el Capítulo 9).

3. El proyecto no utiliza en absoluto el modelo de análisis para describir los resultados del análisis. En cambio, el proyecto analiza los requisitos como parte integrada en la captura de requisitos o en el diseño. En el primero de los casos, hará falta un mayor formalismo en el modelo de casos de uso. Esto puede ser justificable si el cliente es

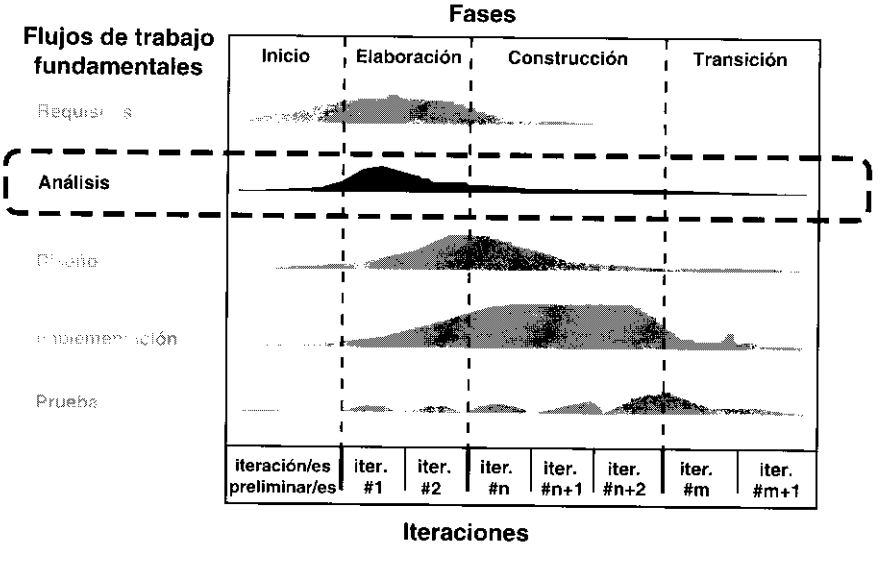


Figura 8.2. El análisis.

capaz de comprender los resultados, aunque creemos que rara vez se da el caso. El segundo caso podría complicar el trabajo de diseño como explicamos en la Sección 8.2.1. Sin embargo, esto puede ser justificable si, por ejemplo, los requisitos son muy simples y/o son bien conocidos, si es fácil identificar la forma del sistema (incluyendo su arquitectura), o si los desarrolladores cuentan con una cierta comprensión, intuitiva pero correcta, de los requisitos, y son capaces de construir un sistema que los encarne de una manera bastante directa. También creemos que rara vez se da este caso.

Al elegir entre las dos primeras variantes, debemos sopesar las ventajas de mantener el modelo de análisis con el coste de mantenerlo durante varias iteraciones y generaciones. Por tanto, tenemos que realizar un análisis coste/beneficio correcto, y decidir si deberíamos dejar de actualizar el modelo de análisis —quizá tan pronto como al final de la fase de elaboración— o si deberíamos conservarlo y mantenerlo durante el resto de la vida del sistema.

En cuanto a la tercera variante, estamos de acuerdo en que el proyecto puede no sólo evitar el coste de mantener el modelo de análisis, sino también el de introducirlo al principio (incluyendo el coste en tiempo y recursos que consume el formar a los desarrolladores —y el hacer que tomen experiencia— en el uso del modelo). Sin embargo, como creemos que ya hemos señalado anteriormente, normalmente las ventajas de trabajar con un modelo de análisis por lo menos al principio del proyecto sobrepasan los costes de emplearlo; por tanto, sólo deberíamos emplear esta variante en casos excepcionales para sistemas extraordinariamente simples.

8.4. Artefactos

8.4.1. Artefacto: modelo de análisis

Presentamos el modelo de análisis al comienzo de la Sección 8.2. La estructura impuesta por el modelo de análisis se define mediante una jerarquía como se muestra en la Figura 8.3.

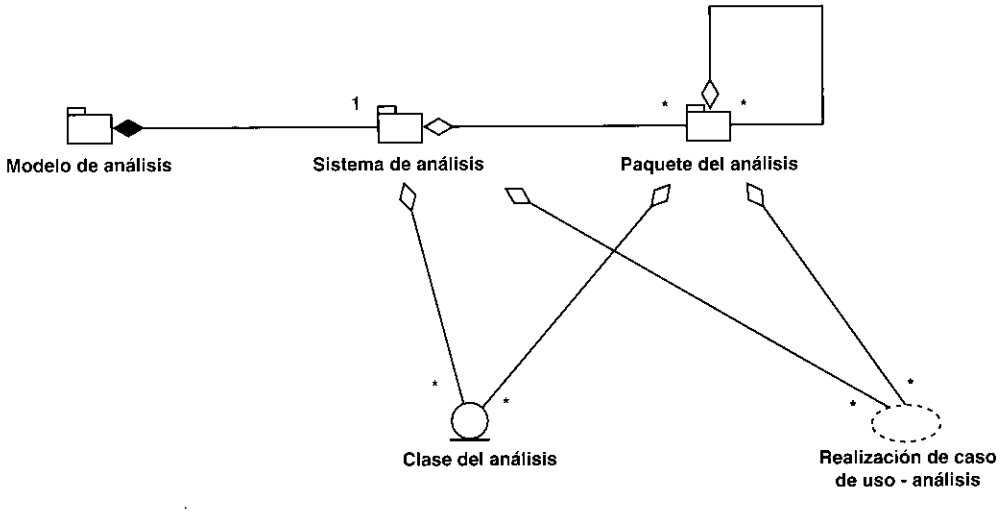


Figura 8.3. El modelo de análisis es una jerarquía de paquetes del análisis que contienen clases del análisis y realizaciones de casos de uso.

El modelo de análisis se representa mediante un sistema de análisis que denota el paquete de más alto nivel del modelo. La utilización de otros paquetes de análisis es por tanto una forma de organizar el modelo de análisis en partes más manejables que representan abstracciones de subsistemas y posiblemente capas completas del diseño del sistema. Las clases de análisis representan abstracciones de clases y posiblemente de subsistemas del diseño del sistema. Dentro del modelo de análisis, los casos de uso se describen mediante clases de análisis y sus objetos. Esto se representa mediante colaboraciones dentro del modelo de análisis que llamamos *realizaciones de caso de uso-análisis*. Los artefactos del modelo de análisis se describen en detalle más adelante en las Secciones 8.4.2 a 8.4.5.

8.4.2. Artefacto: clase del análisis

Una clase de análisis representa una abstracción de una o varias clases y/o subsistemas del diseño del sistema. Esta abstracción posee las siguientes características:

- Una clase de análisis se centra en el tratamiento de los requisitos funcionales y pospone los no funcionales, denominándolos requisitos especiales, hasta llegar a las actividades de diseño e implementación subsiguientes.
- Esto hace que una clase de análisis sea más evidente en el contexto del dominio del problema, más "conceptual", a menudo de mayor granularidad que sus contrapartidas de diseño e implementación.
- Una clase de análisis raramente define u ofrece un interfaz en términos de operaciones y de sus signaturas. En cambio, su comportamiento se define mediante responsabilidades en un nivel más alto y menos formal. Una responsabilidad es una descripción textual de un conjunto cohesivo del comportamiento de una clase.
- Una clase de análisis define atributos, aunque esos atributos también son de un nivel bastante alto. Normalmente los tipos de esos atributos son conceptuales y reconocibles en el dominio del problema, mientras que los tipos de los atributos en las clases de diseño y la implementación suelen ser tipos de lenguajes de programación. Además, los atributos identificados durante el análisis con frecuencia pasan a ser clases en el diseño y la implementación.
- Una clase de análisis participa en relaciones, aunque esas relaciones son más conceptuales que sus contrapartidas de diseño e implementación. Por ejemplo, la navegabilidad de las asociaciones no es muy importante en el análisis, pero es fundamental en el diseño, o por ejemplo, pueden utilizarse generalizaciones durante el análisis, pero podría no ser posible utilizarlas en el diseño si no las soporta el lenguaje de programación.
- Las clases de análisis siempre encajan en uno de tres estereotipos básicos: de interfaz, de control o de entidad (*véase* la Figura 8.4). Cada estereotipo implica una semántica específica (descrita brevemente), lo cual constituye un método potente y consistente de identificar y describir las clases de análisis y contribuye a la creación de un modelo de objetos y una arquitectura robustos. Sin embargo, es mucho más difícil estereotipar las clases de diseño e implementación de esta manera clara e intuitiva. Debido a que tratan requisitos no funcionales, éstas últimas "viven en el contexto de un dominio de la solución", y a menudo se describen mediante sintaxis de lenguajes de programación y tecnologías similares de bajo nivel.

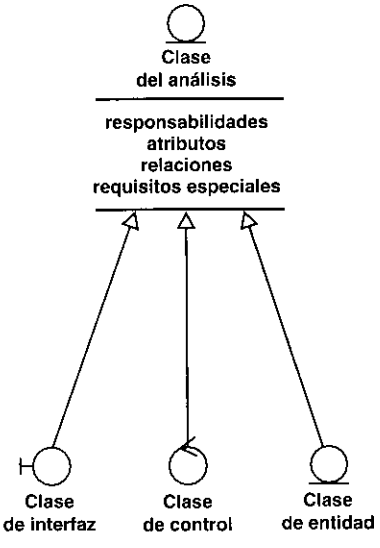


Figura 8.4. Los atributos esenciales y los subtipos (es decir estereotipos) de una clase del análisis.

Estos tres estereotipos están estandarizados en UML y se utilizan para ayudar a los desarrolladores a distinguir el ámbito de las diferentes clases [3]. Cada estereotipo tiene su propio símbolo, como se muestra en la Figura 8.5.

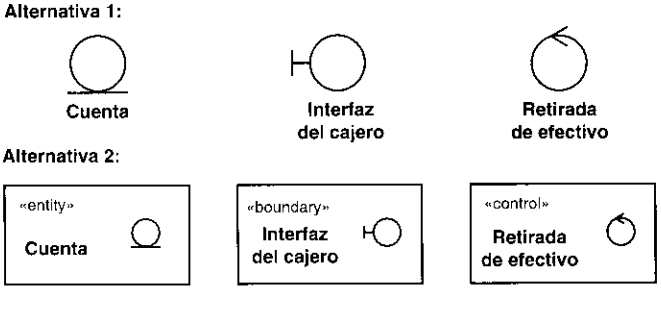


Figura 8.5. UML proporciona tres estereotipos de clases estándar que podemos utilizar en el análisis.

8.4.2.1. Clases de interfaz Las clases de interfaz se utilizan para modelar la interacción entre el sistema y sus actores (es decir, usuarios y sistemas externos). Esta interacción a menudo implica recibir (y presentar) información y peticiones de (y hacia) los usuarios y los sistemas externos.

Las clases de interfaz modelan las partes del sistema que dependen de sus actores, lo cual implica que clarifican y reúnen los requisitos en los límites del sistema. Por tanto, un cambio en un interfaz de usuario o en un interfaz de comunicaciones queda normalmente aislado en una o más clases de interfaz.

Las clases de interfaz representan a menudo abstracciones de ventanas, formularios, paneles, interfaces de comunicaciones, interfaces de impresoras, sensores, terminales, y API (posiblemente

no orientados a objetos). Aún así, las clases de interfaz deberían mantenerse en un nivel bastante alto y conceptual; por ejemplo, no deberíamos profundizar en cada widget de un interfaz de usuario. Obsérvese que es suficiente con que las clases de interfaz describan lo que se obtiene con la interacción (es decir, la información y las peticiones que se intercambian entre el sistema y sus actores). No es necesario que describan cómo se ejecuta físicamente la interacción, ya que esto se considerará en las actividades de diseño e implementación subsiguientes.

Cada clase de interfaz debería asociarse con al menos un actor, y viceversa.

Ejemplo **La clase de interfaz IU Solicitud de Pago**

La siguiente clase de interfaz, llamada IU Solicitud de Pago, se utiliza para cubrir la interacción entre el actor Comprador y el caso de uso Pagar Factura (Figura 8.6).

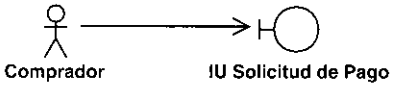


Figura 8.6. La clase de interfaz IU Solicitud de Pago.

IU Solicitud de Pago permite que un usuario consulte las facturas a pagar, después compruebe facturas concretas con más detalle, y por último, solicite al sistema el pago de una factura (planificándola). IU Solicitud de Pago también permite a un usuario descartar una factura que el comprador no quiere pagar.

A continuación daremos ejemplos de cómo esta clase de interfaz se asocia con el "interior" del sistema, es decir, con las clases de control y de entidad.

8.4.2.2. Clases de entidad

Las clases de entidad se utilizan para modelar información que posee una vida larga y que es a menudo persistente. Las clases de entidad modelan la información y el comportamiento asociado de algún fenómeno o concepto, como una persona, un objeto del mundo real, o un suceso del mundo real.

En la mayoría de los casos, las clases de entidad se derivan directamente de una clase de entidad del negocio (o de una clase del dominio) correspondiente, tomada del modelo de objetos del negocio (o del modelo del dominio). Sin embargo, una diferencia fundamental entre clases de entidad y clases de entidad del negocio es que las primeras representan objetos manejados por el sistema en consideración, mientras que las últimas representan objetos presentes en el negocio (y en el dominio del problema) en general. En consecuencia, las clases de entidad reflejan la información de un modo que beneficia a los desarrolladores al diseñar e implementar el sistema, incluyendo su soporte de persistencia. Esto no sucede realmente con las clases de entidad del negocio (o clases del dominio), que por el contrario, describen el contexto del sistema, y por tanto pueden incluir información que el sistema no maneja en absoluto.

Un objeto de entidad no ha de ser necesariamente pasivo y puede tener en ocasiones un comportamiento complejo relativo a la información que representa. Los objetos de entidad aislan los cambios en la información que representan.

Las clases de entidad suelen mostrar una estructura de datos lógica y contribuyen a comprender de qué información depende el sistema.

Ejemplo **La clase de Entidad Factura**

La siguiente clase de entidad, llamada Factura, se utiliza para representar facturas. La clase de entidad se asocia con la clase de interfaz IU Solicitud de Pago por medio de la cual el usuario consulta y gestiona las facturas; *véase* la Figura 8.7.

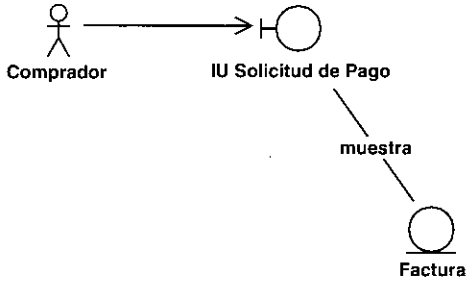


Figura 8.7. La clase de entidad Factura y su relación con la clase de interfaz IU Solicitud de Pago.

8.4.2.3. Clases de control Las clases de control representan coordinación, secuencia, transacciones, y control de otros objetos y se usan con frecuencia para encapsular el control de un caso de uso en concreto. Las clases de control también se utilizan para representar derivaciones y cálculos complejos, como la lógica del negocio, que no pueden asociarse con ninguna información concreta, de larga duración, almacenada por el sistema (es decir, una clase de entidad concreta).

Los aspectos dinámicos del sistema se modelan con clases de control, debido a que ellas manejan y coordinan las acciones y los flujos de control principales, y delegan trabajo a otros objetos (es decir, objetos de interfaz y de entidad).

Obsérvese que las clases de control no encapsulan aspectos relacionados con las interacciones, con los actores, ni tampoco aspectos relacionados con información de larga vida y persistente gestionada por el sistema; esto lo encapsulan las clases de interfaz y de entidad, respectivamente. En cambio, las clases de control encapsulan , y por tanto aíslan, los cambios del control, la coordinación, la secuencia, las transacciones y a veces la lógica del negocio compleja que implica a varios otros objetos.

Ejemplo **La clase de control Planificador de Pagos**

Para refinar el ejemplo anterior, introducimos una clase de control llamada Planificador de Pagos, la cual es responsable de la coordinación entre IU Solicitud de Pago y Factura; *véase* la Figura 8.8.

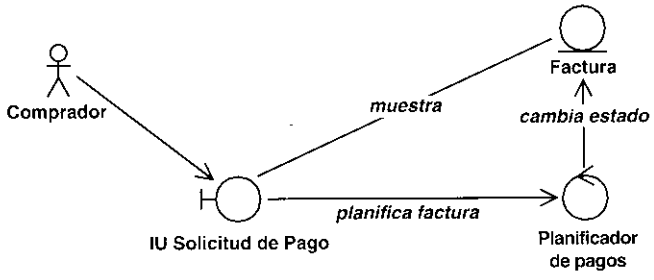


Figura 8.8. La clase de control planificador de pagos y sus relaciones con las clases de interfaz y de entidad.

Planificador de pagos acepta una solicitud de pago, como una solicitud de pagar una factura, y una fecha en la cual debe pagarse la factura. Más adelante, en la fecha de pago, Planificador de pagos lleva a cabo el pago solicitando una transferencia de dinero entre las cuentas apropiadas.

8.4.3. Artefacto: realización de caso de uso-análisis

Una *realización de caso de uso - análisis* es una colaboración dentro del modelo de análisis que describe cómo se lleva a cabo y se ejecuta un caso de uso determinado en términos de las clases del análisis y de sus objetos del análisis en interacción. Una realización de caso de uso proporciona por tanto una traza directa hacia un caso de uso concreto del modelo de casos de uso (Figura 8.9).

Una realización de caso de uso posee una descripción textual del flujo de sucesos, diagramas de clases que muestran sus clases del análisis participantes, y diagramas de interacción que muestran la realización de un flujo o escenario particular del caso de uso en términos de interacciones de objetos del análisis (*véase* la Figura 8.10). Además, debido a que describimos una realización de caso de uso en términos de clases del análisis y de sus objetos, se centra de manera natural en los requisitos funcionales. Por tanto, al igual que las propias clases del análisis, puede posponer el tratamiento de los requisitos no funcionales hasta las actividades subsiguientes de diseño e implementación, llamándoles requisitos especiales en la realización.

8.4.3.1. Diagramas de clases Una clase de análisis y sus objetos normalmente participan en varias realizaciones de casos de uso, y algunas de las responsabilidades, atributos, y asociaciones de una clase concreta suelen ser sólo relevantes para una única realización de

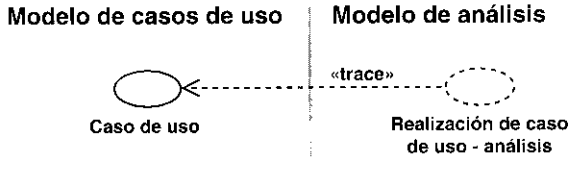


Figura 8.9. Existe una traza entre una realización de caso de uso-análisis en el modelo de análisis y un caso de uso en el modelo de casos de uso.

Figura 8.10. Los atributos y asociaciones fundamentales de una realización de caso de uso-análisis.

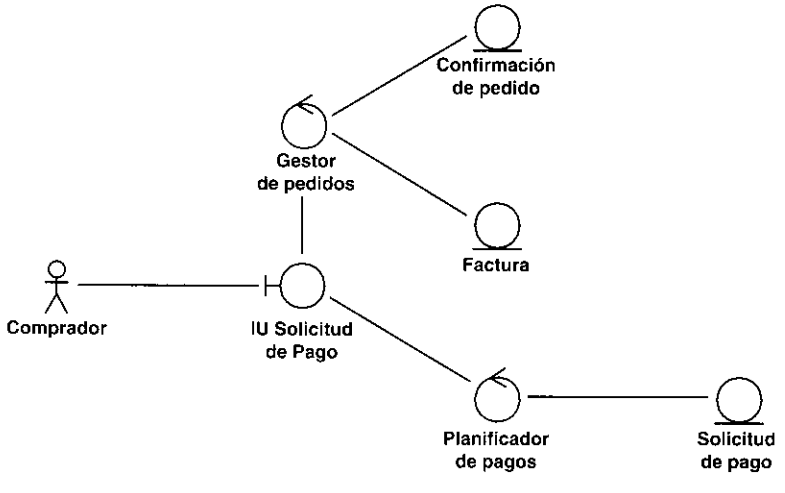


Figura 8.11. Un diagrama de clases de una realización del caso de uso Pagar Factura.

caso de uso. Por tanto, es importante durante el análisis coordinar todos los requisitos sobre una clase y sus objetos que pueden tener diferentes casos de uso. Para hacerlo, adjuntamos diagramas de clases a las realizaciones de casos de uso, mostrando sus clases participantes y sus relaciones (*véase* la Figura 8.11).

8.4.3.2. Diagramas de interacción

La secuencia de acciones en un caso de uso comienza cuando un actor invoca el caso de uso mediante el envío de algún tipo de mensaje al sistema. Si consideramos el "interior" del sistema, un objeto de interfaz recibirá este mensaje del actor. El objeto de interfaz enviará a su vez un mensaje a algún otro objeto, y de esta forma los objetos implicados interactuarán para llevar a cabo el caso de uso. En el análisis preferimos mostrar

esto con diagramas de colaboración ya que nuestro objetivo fundamental es identificar requisitos y responsabilidades sobre los objetos, y no identificar secuencias de interacción detalladas y ordenadas cronológicamente (es ese caso, utilizaríamos en cambio diagramas de secuencia).

En los diagramas de colaboración, mostramos las interacciones entre objetos creando enlaces entre ellos y añadiendo mensajes a esos enlaces. El nombre de un mensaje debería denotar el propósito del objeto invocante en la interacción con el objeto invocado.

Ejemplo Un diagrama de colaboración que muestra la realización de un caso de uso

La Figura 8.12 es un diagrama de colaboración que describe la primera parte del caso de uso Pagar Factura.

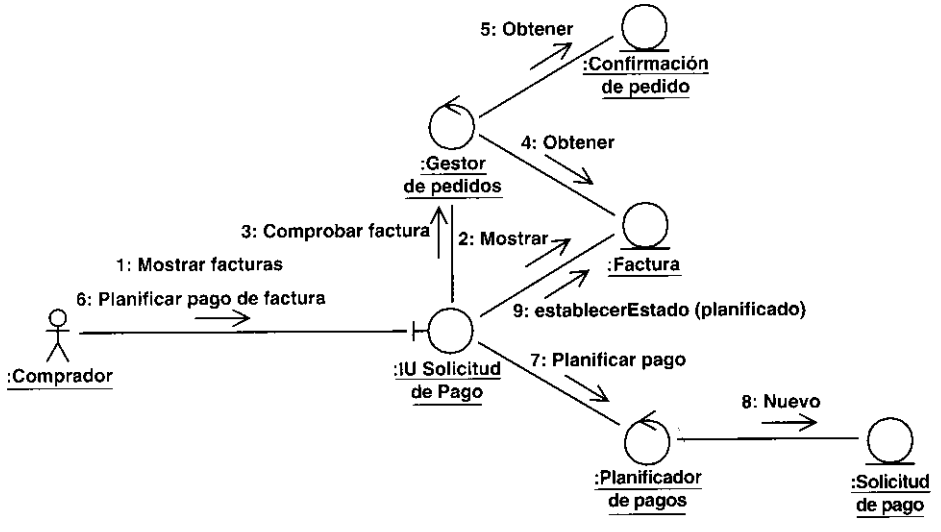


Figura 8.12. Un diagrama de colaboración de una realización del caso de uso Pagar Factura.

En relación con la creación y finalización de los objetos del análisis dentro de una realización de caso de uso, objetos diferentes tienen diferentes ciclos de vida (Apéndice C):

- Un objeto de interfaz no tiene por qué ser particular de una realización de caso de uso si, por ejemplo, debe aparecer en una ventana y participar en dos o más instancias de caso de uso. Sin embargo, los objetos de interfaz a menudo se crean y se finalizan dentro de una sola realización de caso de uso.
- Un objeto de entidad normalmente no es particular de una realización de caso de uso. Al contrario, un objeto de entidad suele tener una vida larga y participa en varias realizaciones de caso de uso antes de su destrucción.
- Las clases de control suelen encapsular el control asociado con un caso de uso concreto, lo cual implica que debería crearse un objeto de control cuando el caso de uso comienza, y ese obje-

to de control debería destruirse cuando termina el caso de uso. Sin embargo, hay excepciones, como cuando un objeto de control participa en más de una realización de caso de uso, cuando varios objetos de control (o diferentes clases de control) participan en una misma realización de caso de uso, y cuando una realización de caso de uso no requiere ningún objeto de control.

8.4.3.3. Flujo de sucesos-análisis Los diagramas —especialmente los diagramas de colaboración— de una realización de caso de uso pueden ser difíciles de leer por sí mismos, de modo que puede ser útil un texto adicional que los explique. Este texto debería escribirse en términos de objetos, particularmente objetos de control que interactúan para llevar a cabo el caso de uso. Sin embargo, el texto no debería mencionar ninguno de los atributos, responsabilidades, y asociaciones del objeto, debido a que cambian con bastante frecuencia y sería difícil mantenerlos.

Ejemplo **Un flujo de sucesos-análisis que explica un diagrama de colaboración**

La siguiente descripción textual complementa el diagrama de colaboración del ejemplo anterior (véase la Figura 8.12):

El comprador consulta a través del IU Solicitud de Pago las facturas gestionadas por el sistema para encontrar las recibidas (1, 2). El IU Solicitud de Pago utiliza el Gestor de Pedidos para comprobar las facturas con sus correspondientes confirmaciones de pedido (3, 4, 5), antes de mostrar la lista de facturas al comprador. Lo que suponga esta comprobación depende de las reglas del negocio establecidas por la organización del comprador, y podría incluir la comparación del precio, la fecha de entrega, y contenidos de la factura con la confirmación de pedido. El objeto Gestor de Pedidos utiliza las reglas del negocio para decidir qué preguntas hacer (representadas por los mensajes Get 4 y 5) a los objetos Confirmación de Pedido y Factura y cómo analizar las respuestas. Cualquier factura sospechosa quedará marcada de alguna forma por el IU Solicitud de Pago, quizá mediante un color diferente que la resalte.

El comprador selecciona una factura mediante el IU Solicitud de Pago y planifica su pago (6). El IU Solicitud de Pago solicita al Planificador de Pagos que planifique el pago de la factura (7). Después, el Planificador de Pagos crea una solicitud de pago (8). El IU Solicitud de Pago cambia a continuación el estado de la factura a "planificada" (9).

El Planificador de Pagos iniciará el pago en el día debido (no se muestra en el diagrama).

Es interesante comparar esta descripción con el flujo de eventos del caso de uso descrito en el Capítulo 7, Sección 7.2.3.1. La descripción del Capítulo 7 es la del comportamiento del caso de uso observable desde el exterior, mientras que la descripción que ofrecemos aquí se centra en cómo el sistema lleva a cabo el caso de uso en términos de objetos (lógicos) que colaboran.

8.4.3.4. Requisitos especiales Los requisitos especiales son descripciones textuales que recogen todos los requisitos no funcionales sobre una realización de caso de uso. Algunos de estos requisitos ya se habían capturado de algún modo durante el flujo de trabajo de los requisitos (como se explicó en los Capítulos 6 y 7), y sólo se cambian a una realización de caso de uso - análisis. Sin embargo, algunos de ellos pueden ser requisitos nuevos o derivados que se encuentran a medida que avanza el trabajo de análisis.

Ejemplo **Requisitos especiales para la realización de un caso de uso**

El siguiente es un ejemplo de requisitos especiales sobre la realización del caso de uso pagar factura:

- Cuando el comprador solicite ver las facturas recibidas, no debería tardar más de medio segundo mostrar las facturas en la pantalla.
- Las facturas deberían pagarse utilizando el estándar SET.

8.4.4. Artefacto: paquete del análisis

Los paquetes del análisis proporcionan un medio para organizar los artefactos del modelo de análisis en piezas manejables. Un paquete de análisis puede constar de clases de análisis, de realizaciones de casos de uso, y de otros paquetes del análisis (recursivamente). *Véase* la Figura 8.13.

Los paquetes del análisis deberían ser cohesivos (es decir, sus contenidos deberían estar fuertemente relacionados), y deberían ser débilmente acoplados (es decir, sus dependencias unos de otros deberían minimizarse).

Además, los paquetes del análisis tienen las siguientes características:

- Los paquetes del análisis pueden representar una separación de intereses de análisis. Por ejemplo, en un sistema grande algunos paquetes del análisis pueden analizarse de manera separada —posiblemente de manera concurrente por diferentes desarrolladores con diferente conocimiento del dominio.
- Los paquetes del análisis deberían crearse basándonos en los requisitos funcionales y en el dominio del problema (es decir, la aplicación o el negocio), y deberían ser reconocibles por las personas con conocimiento del dominio. Los paquetes del análisis no deberían basarse en requisitos no funcionales o en el dominio de la solución.
- Los paquetes del análisis probablemente se convertirán en subsistemas en las dos capas de aplicación superiores del modelo de diseño, o se distribuirán entre ellos. En algunos casos, un paquete del análisis podría incluso reflejar una capa completa de primer nivel en el modelo de diseño.

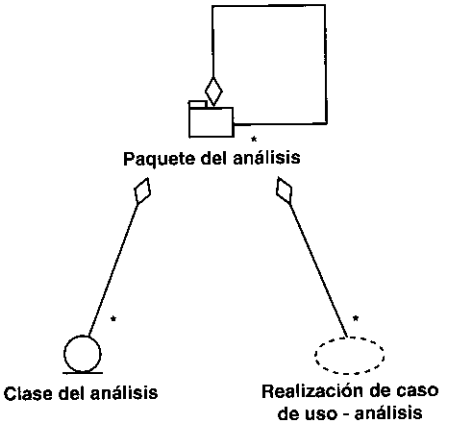


Figura 8.13. Contenidos de un paquete del análisis.

8.4.4.1. Paquetes de servicio Aparte de proporcionar casos de uso a sus actores, todo sistema también proporciona un conjunto de servicios a sus clientes. Un cliente adquiere una combinación adecuada de servicios para ofrecer a sus usuarios los casos de uso necesarios para llevar a cabo su negocio:

- Un caso de uso especifica una secuencia de acciones: un actor inicia un hilo, seguido de interacciones entre el actor y el sistema, que termina tras haber devuelto un valor al actor. Normalmente, los casos de uso no existen aisladamente. Por ejemplo, el caso de uso Enviar Factura al Comprador presupone que otro caso de uso ha creado una factura y que están disponibles la dirección del comprador y otros datos del cliente.
- Un servicio representa un conjunto coherente de acciones relacionadas funcionalmente —un paquete de funcionalidad— que se utiliza en varios casos de uso. Un cliente de un sistema normalmente compra una combinación de servicios para ofrecer a sus usuarios los casos de uso necesarios. Un servicio es indivisible en el sentido de que el sistema necesita ofrecerlo o todo entero o nada en absoluto.
- Los casos de uso son para los usuarios, y los servicios son para los clientes. Los casos de uso atraviesan los servicios, es decir, un caso de uso requiere acciones de varios servicios.

En el Proceso Unificado, el concepto de servicio está soportado por los paquetes de servicio. Los paquetes de servicio se utilizan en un nivel más bajo de la jerarquía (de agregación) de paquetes del análisis para estructurar el sistema de acuerdo a los servicios que proporciona. Podemos observar lo siguiente acerca de los paquetes de servicio:

- Un paquete de servicio contiene un conjunto de clases relacionadas funcionalmente.
- Un paquete de servicio es indivisible. Cada cliente obtiene o bien todas las clases o bien ninguna del paquete de servicio.
- Cuando se lleva a cabo un caso de uso, puede que sean participantes uno o más paquetes de servicio. Además, es frecuente que un paquete de servicio concreto participe en varias realizaciones de caso de uso diferentes.
- Un paquete de servicio, depende a menudo de otro paquete de servicio.
- Un paquete de servicio normalmente sólo es relevante para uno o unos pocos actores.
- La funcionalidad definida por un paquete de servicio, cuando se diseña e implementa, puede gestionarse como una unidad de distribución separada. Un paquete de servicio puede, por tanto, representar cierta funcionalidad "adicional" del sistema. Cuando un paquete de servicio queda excuido, también lo queda todo caso de uso cuya realización requiera el paquete de servicio.
- Los paquetes de servicio pueden ser mutuamente excluyentes, o pueden representar diferentes aspectos o variantes del mismo servicio. Por ejemplo, "corrección ortográfica del inglés británico" y "corrección ortográfica del inglés americano" pueden ser dos paquetes de servicio diferentes que proporciona un sistema.
- Los paquetes de servicio constituyen una entrada fundamental para las actividades de diseño e implementación subsiguientes, dado que ayudarán a estructurar los modelos de diseño e implementación en términos de subsistemas de servicio. En particular, los subsistemas de servicio tienen una influencia decisiva en la descomposición del sistema en componentes binarios y ejecutables.

Mediante la estructuración del sistema, de acuerdo a los servicios que proporciona, nos preparamos para los cambios en servicios individuales, ya que esos cambios probablemente se localizarán en el paquete de servicio correspondiente. Esto da como resultado un sistema robusto que es resistente al cambio.

En la Sección 8.6.1.1.2 ofrecemos un método para identificar paquetes de servicio, junto con ejemplos de paquetes de servicio.

Obsérvese que la manera general de organizar los artefactos del modelo de análisis sigue siendo la utilización de paquetes del análisis ordinarios como explicamos en la sección anterior. Sin embargo, aquí hemos introducido un estereotipo "paquete de servicio" para poder marcar explícitamente aquellos paquetes que representan servicios. En sistemas grandes (que tienen muchos paquetes), es especialmente importante el ser capaces de diferenciar diferentes tipos de paquetes de una manera sencilla. Esto es también un ejemplo de cómo pueden utilizarse y aprovecharse los estereotipos dentro del UML.

*8.4.4.1.1. **Los paquetes de servicio son reutilizables*** Como explicamos en la sección anterior, los paquetes de servicio tienen muchas características buenas, como el ser cohesivos (Apéndice C), indivisibles, débilmente acoplados, distribuidos de manera separada, y demás. Esto hace que los paquetes de servicio sean principales candidatos para la reutilización, tanto dentro de un sistema como entre sistemas (más o menos) relacionados. Más en concreto, los paquetes de servicio cuyos servicios se centran alrededor de una o más clases de entidad (*véase* la Sección 8.4.2.2) son probablemente reutilizables en diferentes sistemas que soporten el mismo negocio o dominio. Esto es debido a que las clases de entidad se obtienen a partir de clases de entidad del negocio o de clases del dominio, lo cual hace a las clases de entidad y sus servicios relacionados candidatos para reutilización dentro del negocio o dominio entero y dentro de la mayoría de los sistemas que los soportan —no sólo candidatos para un sistema en concreto—. Los paquetes de servicio, y los servicios, son independientes de los casos de uso en el sentido de que un paquete de servicio puede emplearse en varias realizaciones de caso de uso diferentes. Esto es particularmente cierto cuando el paquete de servicio reside en una capa general con funcionalidad común y compartida (*véase* la Sección 8.6.1.1.3). Un paquete de servicio de ese tipo es probable que se reutilice en varias aplicaciones diferentes (configuraciones) del sistema, donde cada aplicación proporciona casos de uso cuyas realizaciones requieren el paquete de servicio. Los paquetes de servicio poseen una traza con subsistemas de servicio en el diseño (*véase* la Sección 9.3.4.1) y con componentes en la implementación (*véase* la Sección 10.3.2). Esos componentes son reutilizables por los mismos motivos por los que lo son los paquetes de servicio. Por tanto, los paquetes de servicio se revelan como nuestro instrumento fundamental para la reutilización durante el análisis. Esto tiene sus consecuencias tanto en el diseño como en la implementación del sistema, y finalmente obtiene como resultado componentes reutilizables.

8.4.5. Artefacto: descripción de la arquitectura (vista del modelo de análisis)

La descripción de la arquitectura contiene una **vista de la arquitectura del modelo de análisis** (Apéndice C), que muestra sus artefactos significativos para la arquitectura (Figura 8.14).

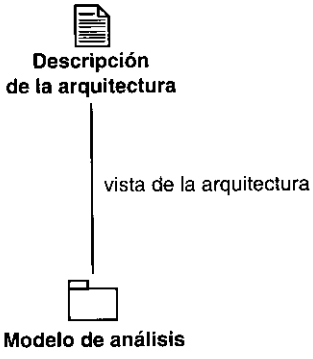


Figura 8.14. La descripción de la arquitectura contiene una vista arquitectónica del modelo de análisis.

Los siguientes artefactos del modelo de análisis normalmente se consideran significativos para la arquitectura:

- La descomposición del modelo de análisis en paquetes de análisis y sus dependencias. Esta descomposición suele tener su efecto en los subsistemas de las capas superiores durante el diseño y la implementación y es por tanto relevante para la arquitectura en general.
- Las clases fundamentales del análisis como las clases de entidad que encapsulan un fenómeno importante del dominio del problema; las clases de interfaz que encapsulan interfaces de comunicación importantes y mecanismos de interfaz de usuario; las clases de control que encapsulan importantes secuencias con una amplia cobertura (es decir, aquéllas que coordinan realizaciones de casos de uso significativas); y clases del análisis que son generales, centrales, y que tienen muchas relaciones con otras clases del análisis. Suele ser suficiente con considerar significativa para la arquitectura una clase abstracta pero no sus subclases.
- Realizaciones de casos de uso que describen cierta funcionalidad importante y crítica; que implican muchas clases del análisis y por tanto tienen una cobertura amplia, posiblemente a lo largo de varios paquetes de análisis; o que se centran en un caso de uso importante que debe ser desarrollado al principio en el ciclo de vida del software, y por tanto es probable que se encuentre en la **vista de la arquitectura del modelo de casos de uso** (Apéndice C).

8.5. Trabajadores

8.5.1. Trabajador: arquitecto

Durante el flujo de trabajo del análisis, el arquitecto es responsable de la integridad del modelo de análisis, garantizando que éste sea correcto, consistente y legible como un todo (*véase* la Figura 8.15). En sistemas grandes y complejos, estas responsabilidades pueden requerir más mantenimiento tras algunas iteraciones, y el trabajo que conllevan puede hacerse bastante rutinario. En esos casos, el arquitecto puede delegar ese trabajo a otro trabajador, posiblemente a un ingeniero de componentes de "alto nivel" (*véase* la Sección 8.5.3). Sin embargo, el

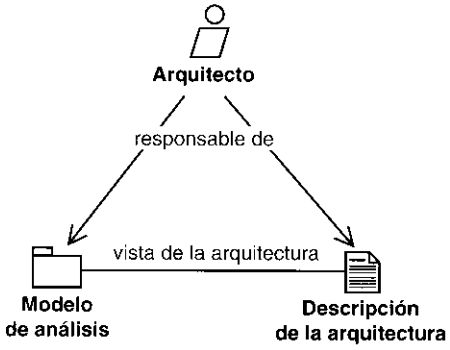


Figura 8.15. Las responsabilidades del arquitecto en el análisis.

arquitecto sigue siendo responsable de lo que es significativo para la arquitectura —la descripción de la arquitectura—. El otro trabajador será responsable del paquete de nivel superior del modelo de análisis, que debe ser conforme con la descripción de la arquitectura.

El modelo de análisis es correcto cuando realiza la funcionalidad descrita en el modelo de casos de uso, y sólo esa funcionalidad.

El arquitecto es también responsable de la arquitectura del modelo de análisis, es decir, de la existencia de sus partes significativas para la arquitectura tal y como se muestran en la vista de la arquitectura del modelo. Recuérdese que esta vista es una parte de la descripción de la arquitectura del sistema.

Obsérvese que el arquitecto no es responsable del desarrollo y mantenimiento continuo de los diferentes artefactos del modelo de análisis. Éstos son responsabilidad de los correspondientes ingenieros de casos de uso e ingenieros de componentes (*véase* las Secciones 8.5.2 y 8.5.3).

8.5.2. Trabajador: ingeniero de casos de uso

Un ingeniero de casos de uso es responsable de la integridad de una o más realizaciones de caso de uso, garantizando que cumplen los requisitos que recaen sobre ellos (*véase* la Figura 8.16).

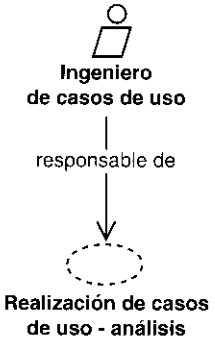


Figura 8.16. Las responsabilidades de un ingeniero de casos de uso en el análisis.

Una realización de caso de uso debe llevar a cabo correctamente el comportamiento de su correspondiente caso de uso del modelo de casos de uso, y sólo ese comportamiento. Esto incluye garantizar que todas las descripciones textuales y los diagramas que describen la realización del caso de uso son legibles y se ajustan a su objetivo.

Obsérvese que el ingeniero de casos de uso no es responsable de las clases del análisis ni de las relaciones que se usan en la realización del caso de uso. Éstas son las responsabilidades correspondientes al ingeniero de componentes (*véase* la Sección 8.5.3).

Como veremos en el siguiente capítulo, el ingeniero de casos de uso también es responsable del diseño de las realizaciones de los casos de uso. Por tanto, el ingeniero de casos de uso es responsable tanto del análisis como del diseño del caso de uso, lo cual resulta en una transición suave.

8.5.3. Trabajador: ingeniero de componentes

El ingeniero de componentes define y mantiene las responsabilidades, atributos, relaciones, y requisitos especiales de una o varias clases del análisis, asegurándose de que cada clase del análisis cumple los requisitos que se esperan de ella de acuerdo a las realizaciones de caso de uso en las que participa (*véase* la Figura 8.17).

El ingeniero de componentes también mantiene la integridad de uno o varios paquetes del análisis. Esto incluye garantizar que sus contenidos (por ejemplo, clases y sus relaciones) son correctos y que sus dependencias de otros paquetes del análisis son correctas y mínimas.

Suele ser apropiado dejar que el ingeniero de componentes responsable de un paquete del análisis lo sea también de las clases del análisis contenidas en él. Además, si existe una correspondencia directa (una traza directa) entre un paquete del análisis y los subsistemas de diseño correspondientes (*véase* la Sección 8.4.4), el ingeniero de componentes debería ser también responsable de esos subsistemas, para utilizar el conocimiento adquirido durante el análisis en el diseño y la implementación del paquete del análisis. Si no existe una correspondencia directa de ese tipo, pueden ocuparse en el diseño e implementación del paquete del análisis a ingenieros de componentes adicionales.

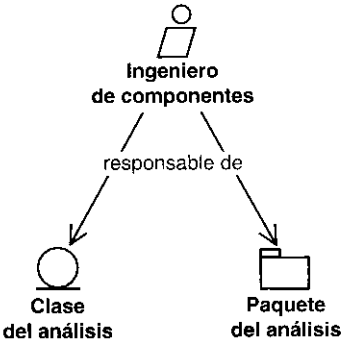


Figura 8.17. Las responsabilidades de un ingeniero de componentes en el análisis.

8.6. Flujo de trabajo

Anteriormente en este capítulo, describimos el trabajo del análisis en términos estáticos. Ahora utilizaremos un diagrama de actividad para razonar acerca de su comportamiento dinámico (*véase* la Figura 8.18).

Los arquitectos comienzan la creación del modelo de análisis (como hemos definido anteriormente en este capítulo), identificando los paquetes de análisis principales, las clases de entidad evidentes, y los requisitos comunes. Después, los ingenieros de casos de uso realizan cada caso de uso en términos de las clases del análisis participantes exponiendo los requisitos de comportamiento de cada clase. Los ingenieros de componentes especifican posteriormente estos requisitos y los integran dentro de cada clase creando responsabilidades, atributos y relaciones consistentes para cada clase. Durante el análisis, el arquitecto identifica de manera continua nuevos paquetes del análisis, clases, y requisitos comunes a medida que el modelo de análisis evoluciona, y los ingenieros de componentes responsables de los paquetes del análisis concretos continuamente los refinan y mantienen.

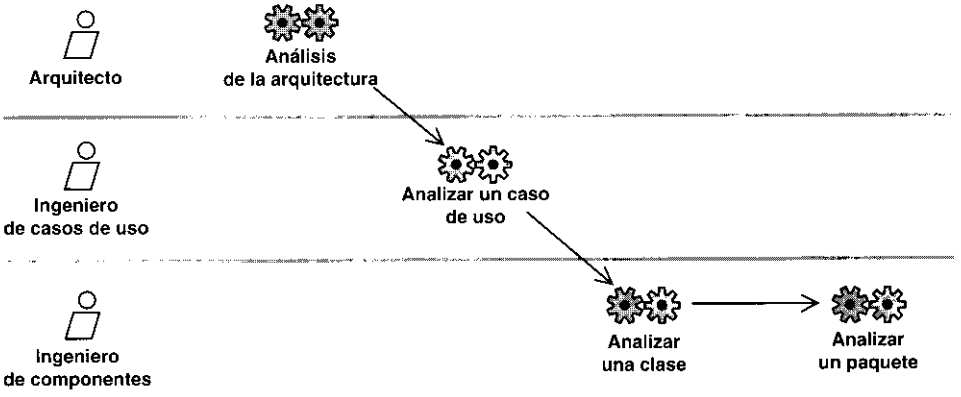


Figura 8.18. El flujo de trabajo en el análisis, incluyendo a los trabajadores participantes y sus actividades.

8.6.1. Actividad: análisis de la arquitectura

El propósito del análisis de la arquitectura es esbozar el modelo de análisis y la arquitectura mediante la identificación de paquetes del análisis, clases del análisis evidentes, y requisitos especiales comunes (*véase* la Figura 8.19).

8.6.1.1. Identificación de paquetes del análisis Los paquetes del análisis proporcionan un medio para organizar el modelo de análisis en piezas más pequeñas y más manejables. Pueden, bien identificarse inicialmente como forma de dividir el trabajo de análisis, o bien encontrarse a medida que el modelo de análisis evoluciona y "crece" convirtiéndose en una gran estructura que debe descomponerse.

Una identificación inicial de los paquetes del análisis se hace de manera natural basándonos en los requisitos funcionales y en el dominio del problema, es decir, en la aplicación o negocio que estamos considerando. Debido a que capturamos los requisitos funcionales en la forma de

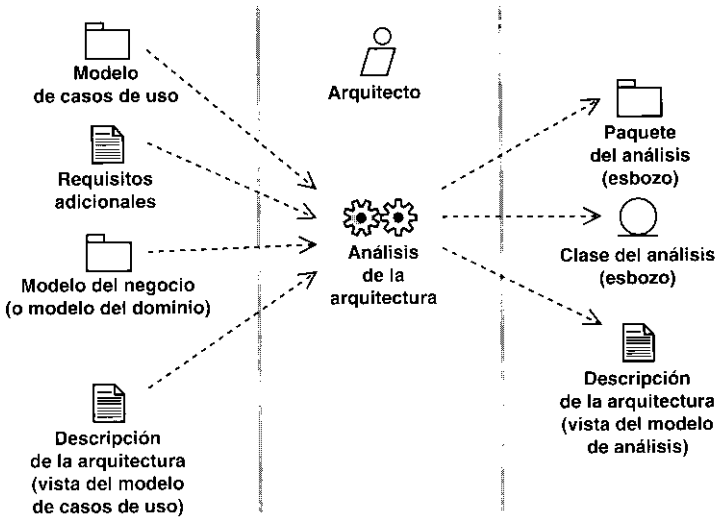


Figura 8.19. Las entradas y los resultados del análisis arquitectónico.

casos de uso, una forma directa de identificar paquetes del análisis es asignar la mayor parte de un cierto número de casos de uso a un paquete concreto y después realizar la funcionalidad correspondiente dentro de ese paquete. Entre las "asignaciones" adecuadas de casos de uso a un paquete en concreto tenemos las siguientes:

- Los casos de uso requeridos para dar soporte a un determinado proceso de negocio.
- Los casos de uso requeridos para dar soporte a un determinado actor del sistema.
- Los casos de uso que están relacionados mediante relaciones de generalización y de extensión. Este tipo de conjunto de casos de uso es coherente en el sentido de que los casos de uso o bien especializan o bien "extienden" a los otros.

Los paquetes de estos tipos localizan los cambios respectivamente en un proceso del negocio, en el comportamiento de un actor, y en un conjunto de casos de uso estrechamente relacionados. Esta técnica simplemente nos ayuda inicialmente a asignar casos de uso a paquetes. Por tanto, a medida que se desarrolla el trabajo del análisis, cuando los casos de uso sean realizados como colaboraciones entre clases, posiblemente en diferentes paquetes, se evolucionará hacia una estructura de paquetes más refinada.

Ejemplo Identificación de paquetes del análisis

Este ejemplo muestra cómo Interbank Software podría identificar algunos de sus paquetes del análisis a partir del modelo de casos de uso. Los casos de uso Pagar Factura, Enviar Aviso, y Enviar factura al comprador están todos implicados en el mismo proceso del negocio, Ventas: del Pedido a la Entrega. Por tanto, pueden incluirse en un mismo paquete del análisis.

Sin embargo, Interbank Software debe ser capaz de poner su sistema a disposición de diferentes clientes con diferentes necesidades. Algunos clientes utilizan el sistema sólo como compradores, otros sólo como vendedores, y algunos como compradores y vendedores. Por tanto, decidieron

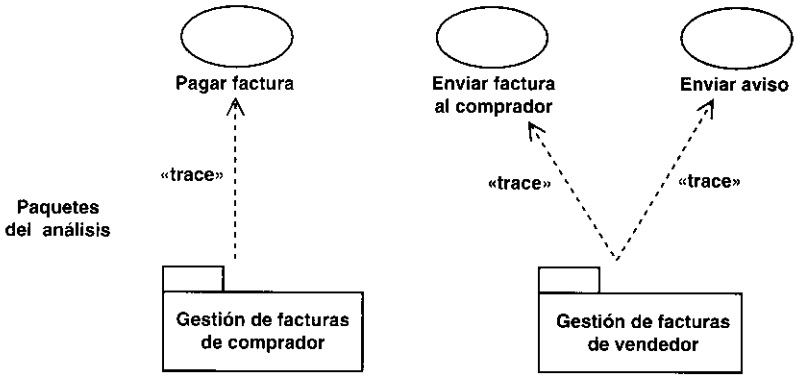


Figura 8.20. Identificación de paquetes del análisis a partir de los casos de uso.

separar la realización de los casos de uso necesarios para el vendedor de la realización de los casos de uso necesarios para el comprador. Aquí vemos cómo la suposición inicial de un paquete del análisis para el proceso del negocio Ventas: del Pedido a la Entrega se ha ajustado para adaptarse a los requisitos del cliente. El resultado es que tenemos dos paquetes del análisis que pueden distribuirse separadamente a los clientes dependiendo de sus necesidades: Gestión de facturas de comprador y gestión de facturas de vendedor (*véase* la Figura 8.20).

Obsérvese que también hay otros casos de uso que soportan el proceso del negocio Ventas: del Pedido a la Entrega, pero los hemos ignorado para hacer más sencillo el ejemplo.

8.6.1.1.1. Gestión de los aspectos comunes entre paquetes del análisis Con frecuencia se da el caso de encontrar aspectos comunes entre los paquetes identificados como en la sección anterior. Un ejemplo es cuando dos o más paquetes del análisis necesitan compartir la misma clase del análisis. Una manera apropiada de tratarlo es extraer la clase compartida, colocarla dentro de su propio paquete o simplemente fuera de cualquier otro paquete, y hacer después que los otros paquetes sean dependientes de este paquete o clase más general.

Las clases compartidas de ese tipo son muy probablemente clases de entidad de las cuales se puede hacer una traza hasta clases del dominio o clases de entidad del negocio. Por tanto merece la pena estudiar las clases del dominio o las clases de entidad del negocio si están compartidas, y si se van a identificar inicialmente los paquetes en el análisis.

Ejemplo Identificación de paquetes del análisis generales

Este ejemplo muestra cómo Interbank Software podría identificar paquetes del análisis generales a partir del modelo del dominio. Cada una de las clases del dominio Cliente de Banco y Cuenta representa una entidad importante y compleja del mundo real. Interbank se da cuenta de que esas clases requieren un soporte sofisticado del sistema de información, y de que están compartidas por otros paquetes del análisis más específicos. Interbank Software crea en consecuencia un paquete aparte para cada clase, Gestión de Cuentas y Gestión de Clientes de Banco (*véase* la Figura 8.21).

Obsérvese que los paquetes Gestión de Cuentas y Gestión de Clientes de Banco probablemente contendrán muchas clases del análisis tales como clases de control y de interfaz relacionadas respectivamente con cuentas y con Gestión de Clientes de Banco. Por tanto, es improbable que esos

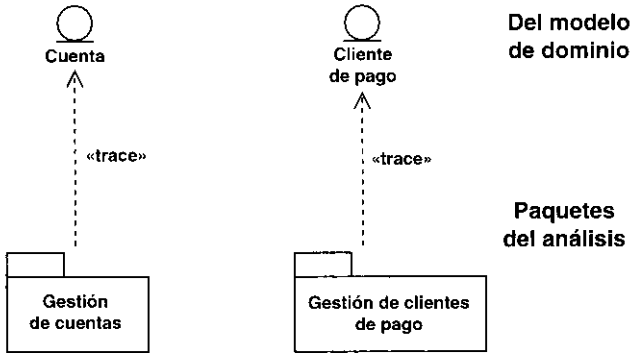


Figura 8.21. Identificación de paquetes del análisis generales a partir de clases del dominio.

paquetes contengan sólo una o unas pocas clases de entidad trazables hasta sus correspondientes clases del dominio.

8.6.1.1.2. Identificación de paquetes de servicio La identificación adecuada de los paquetes de servicio se suele hacer cuando el trabajo de análisis está avanzado, momento en el que los requisitos funcionales se comprenden bien y existen la mayoría de las clases del análisis. Todas las clases del análisis dentro del mismo paquete de servicio contribuyen al mismo servicio.

Al identificar paquetes de servicio, debemos hacer lo siguiente:

- Identificar un paquete de servicio por cada servicio opcional. El paquete de servicio será una unidad de compra.

Ejemplo Paquetes de servicio opcionales

La mayoría de los vendedores que utilizan el sistema de Interbank quieren un servicio para enviar avisos. Este servicio se describe en el caso de uso (opcional) enviar aviso. Algunos vendedores

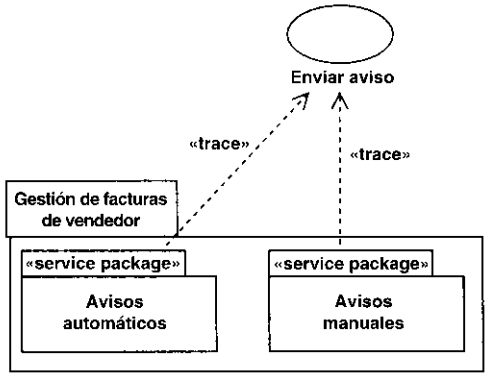


Figura 8.22. Los paquetes de servicio, Avisos automáticos y Avisos manuales, ubicados dentro del paquete Gestión de Facturas de Vendedor.

quieren que los avisos se envíen automáticamente tan pronto como se dé una factura atrasada, mientras que otros prefieren que se les avise cuando haya una factura atrasada, y ellos mismos decidirán después si mandan un aviso. Esta variabilidad se representa mediante dos paquetes de servicio opcionales y mutuamente exclusivos: Avisos automáticos se utiliza para los avisos automáticos, y Avisos manuales notifica al vendedor, el cual decide si contacta con el comprador; *véase* la Figura 8.22. Cuando un vendedor no quiere ningún soporte para avisos, no se entrega ninguno de los paquetes con el sistema. Los paquetes de servicio están dentro del paquete gestión de facturas de vendedor.

- Identificar un paquete de servicio por cada servicio que *podría* hacerse opcional, incluso aunque todos los clientes siempre lo quieran. Debido a que los paquetes de servicio contienen clases relacionadas funcionalmente, obtendremos con ellos una estructura de paquetes que aísla la mayoría de los cambios en paquetes individuales. Podríamos haber descrito este criterio también del siguiente modo: identificar un paquete de servicio por cada servicio proporcionado por clases relacionadas funcionalmente. Los siguientes son ejemplos de cuándo una clase A y una clase B están relacionadas funcionalmente:
  - — Un cambio en A requerirá muy probablemente un cambio en B.
  - — La eliminación de A hace que B sea superflua.
  - — Los objetos de A interactúan intensamente con objetos de B, posiblemente a través de varios mensajes diferentes.

Ejemplo **Identificación de paquetes de servicio que encapsulan clases relacionadas funcionalmente**

El paquete Gestión de Cuentas incluye un paquete de servicio general llamado Cuentas que se utiliza para acceder a las cuentas en actividades tales como la transferencia de dinero y la extracción de históricos de transacciones. El paquete también incluye un paquete de servicio llamado Riesgos para estimar los riesgos asociados con una determinada cuenta. Estos diferentes paquetes de servicio son comunes y los utilizan varias realizaciones de caso de uso distintas. *Véase* la Figura 8.23.

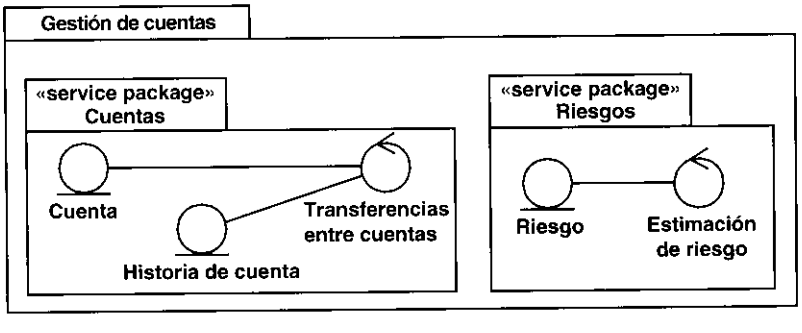


Figura 8.23. Los paquetes de servicio Cuentas y Riesgos, cada uno de ellos encapsulando clases relacionadas funcionalmente.

8.6.1.1.3. Definición de dependencias entre paquetes del análisis Deberían definirse dependencias entre los paquetes del análisis si sus contenidos están relacionados. La dirección de la dependencia debería ser la misma (navegabilidad) dirección de la relación.

El objetivo es conseguir paquetes que sean relativamente independientes y débilmente acoplados pero que posean una cohesión interna alta. La cohesión alta y el acoplamiento débil hace que los paquetes sean más fáciles de mantener debido a que cambiar algunas clases del paquete afectará fundamentalmente a clases dentro del propio paquete. Por tanto, es inteligente intentar reducir el número de relaciones entre clases en paquetes diferentes debido a que ello reduce las dependencias entre paquetes.

Para hacer más claras las dependencias, puede ser útil estratificar el modelo de análisis haciendo que los paquetes específicos de la aplicación queden en una capa de nivel superior y los paquetes generales queden en una capa inferior. Esto aclara la diferencia entre funcionalidad específica y general.

Ejemplo **Dependencias entre paquetes del análisis y capas**

El paquete Gestión de Cuentas contiene varias clases, como Cuenta, que son utilizadas por clases en otros paquetes. Por ejemplo, la clase Cuenta es utilizada por clases en los paquetes Gestión de Facturas de Comprador y Gestión de Facturas de Vendedor. Por tanto, estos paquetes dependen del paquete Gestión de Cuentas (*véase* la Figura 8.24).

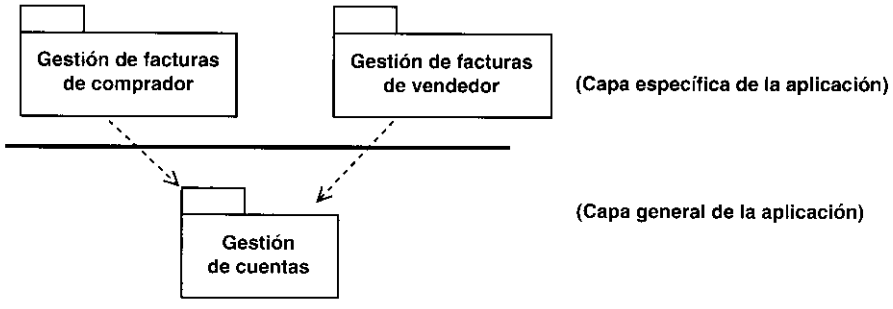


Figura 8.24. Dependencias y capas de paquetes del análisis.

Durante el diseño y la implementación, refinaremos esas capas y añadiremos más capas de bajo nivel a medida que tomemos en consideración el entorno de implementación y los requisitos no funcionales en general.

8.6.1.2. Identificación de clases de entidad obvias

Suele ser adecuado preparar una propuesta preliminar de las clases de entidad más importantes (10 o 20) basado en las clases del dominio o las entidades del negocio que se identificaron durante la captura de requisitos. Sin embargo, la mayoría de las clases se identificarán al crear las realizaciones de los casos de uso (en la actividad de análisis de casos de uso, *véase* la Sección 8.6.2). Debido a esto, deberíamos tener cuidado de no identificar demasiadas clases en esta etapa y quedar atrapados

en demasiados detalles. Debería ser bastante con un esbozo inicial de las clases significativas para la arquitectura (*véase* la Sección 8.4.5). En otro caso, probablemente tendremos que rehacer gran parte del trabajo cuando se utilicen más adelante los casos de uso para identificar las clases de entidad realmente necesarias, es decir, aquellas que participan en la realización de los casos de uso. Una clase de entidad que no participa en una realización de caso de uso no es necesaria.

Las agregaciones y asociaciones entre clases del dominio en el modelo del dominio (o entre entidades del negocio en el modelo del negocio) pueden utilizarse para identificar un conjunto provisional de asociaciones entre las correspondientes clases de entidad.

Ejemplo **Una clase de entidad identificada a partir de una clase del dominio**

Factura es una clase del dominio que mencionamos en el Capítulo 6. Utilizaremos esta clase del dominio para proponer una de las clases de entidad iniciales. Como punto de partida, podemos proponer los mismos atributos para la clase factura: cantidad, fecha de envío, y fecha límite de pago. También podemos definir asociaciones entre clases de entidad del modelo del dominio, tales como la asociación pagable entre un pedido y una factura.

8.6.1.3. Identificación de requisitos especiales comunes Un requisito especial es un requisito que aparece durante el análisis y que es importante anotar de forma que pueda ser tratado adecuadamente en las subsiguientes actividades de diseño e implementación. Como ejemplo citamos las limitaciones o restricciones sobre:

- Persistencia.
- Distribución y concurrencia.
- Características de seguridad.
- Tolerancia a fallos.
- Gestión de transacciones.

El arquitecto es el responsable de identificar los requisitos especiales comunes de forma que los desarrolladores puedan referirse a ellos como requisitos especiales sobre realizaciones de casos de uso y clases del análisis determinadas. En algunos casos, los requisitos especiales no pueden encontrarse al principio y en cambio aparecen a medida que se exploran las realizaciones de casos de uso y las clases del análisis. Obsérvese también que no es raro que una clase o una realización de caso de uso especifique varios requisitos especiales diferentes.

Deberían identificarse las características fundamentales de cada requisito especial común para soportar mejor el diseño y la implementación posteriores.

Ejemplo **Identificación de las características fundamentales de un requisito especial**

Un requisito de persistencia tienen las siguientes características:

- *Rango de tamaño*: Rango de tamaño de los objetos que hay que hacer persistentes.

- *Volumen*: Número de objetos que hay que hacer persistentes.
- *Período de persistencia*: Periodo de tiempo habitual que un objeto debe ser persistente.
- *Frecuencia de actualización*: Frecuencia de actualización de los objetos.
- *Fiabilidad*: Aspectos de fiabilidad tales como si los objetos deberían sobrevivir en caso de una caída del software o del hardware.

Las características de cada requisito especial se calificarán después para cada clase o realización de caso de uso que haga referencia al requisito especial.

8.6.2. Actividad: analizar un caso de uso

Analizamos un caso de uso para (*véase* la Figura 8.25):

- Identificar las clases del análisis cuyos objetos son necesarios para llevar a cabo el flujo de sucesos del caso de uso.
- Distribuir el comportamiento del caso de uso entre los objetos del análisis que interactúan.
- Capturar requisitos especiales sobre la realización del caso de uso.

Otra forma de llamar al análisis de casos de uso podría ser *refinamiento de casos de uso*. Refinamos cada caso de uso como colaboración de clases del análisis.

8.6.2.1. Identificación de clases del análisis En este paso, identificamos las clases de control, entidad, e interfaz necesarias para realizar los casos de uso y esbozamos sus nombres, responsabilidades, atributos y relaciones.

Los casos de uso descritos en los requisitos no siempre están suficientemente detallados como para poder identificar clases del análisis. La información sobre el interior del sistema nor-

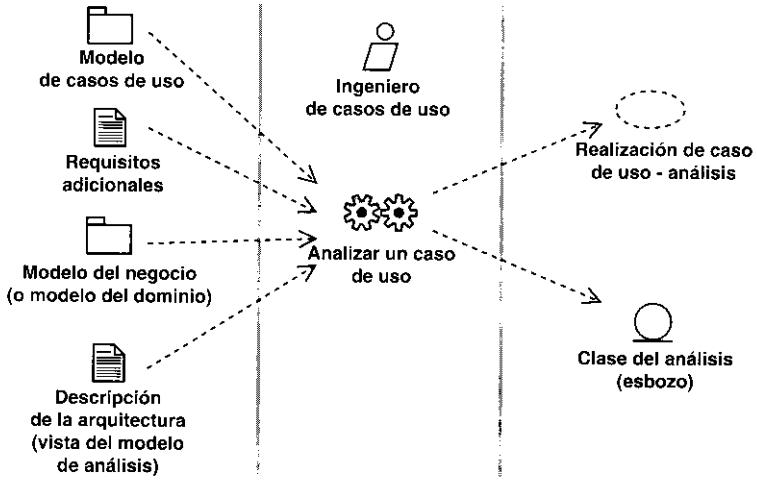


Figura 8.25. Las entradas y los resultados del análisis de un caso de uso.

malmente no es de interés durante la captura de requisitos y por tanto puede haberse dejado de lado. Por tanto, para identificar las clases del análisis puede que tengamos que refinar las descripciones de los casos de uso en lo referente al interior del sistema. Este refinamiento debe recogerse en la descripción textual de flujos de sucesos —análisis de la realización de los casos de uso.

Podemos utilizar las siguientes normas generales para identificar las clases del análisis:

- Identificar clases de entidad mediante el estudio en detalle de la descripción del caso de uso y de cualquier modelo del dominio que se tenga, y después considerar qué información debe utilizarse y manipularse en la realización del caso de uso. Sin embargo, debemos ser conscientes de la información que es mejor capturar como atributo (*véase* la Sección 8.6.3.2, "Identificación de atributos"), la que es preferible asociar a clases de interfaz o de control, o la que simplemente no es necesaria para la realización del caso de uso; las "informaciones" de estos tipos no deberían modelarse como clases de entidad.

- Identificar una clase de interfaz central para cada actor humano, y dejar que esta clase represente la ventana principal del interfaz de usuario con el cual interactúa el actor. Si el actor ya interactúa con una clase de interfaz, deberíamos considerar el reutilizarla para conseguir una buena facilidad de uso de la **interfaz de usuario** (Apéndice C) y para minimizar el número de ventanas principales que cada actor requiere para interactuar con ellas. Además, esas clases de interfaz centrales normalmente se consideran agregados de clases de interfaz más primitivas.

- Identificar una clase de interfaz primitiva para cada clase de entidad que hayamos encontrado anteriormente. Estas clases representan objetos lógicos con las cuales interactúa el actor (humano) en la interfaz de usuario durante el caso de uso. Estas clases de interfaz primitivas pueden refinarse después de acuerdo a diversos criterios de facilidad de uso para contribuir a la creación de una "buena" interfaz de usuario.

- Identificar una clase de interfaz central para cada actor que sea un sistema externo, y dejar que esta clase represente la interfaz de comunicación. Recuérdese que un actor del sistema puede ser cualquier cosa desde unidades software o hardware que interactúan con nuestro sistema, tales como impresoras, terminales, dispositivos de alarma, sensores, etc. Si las comunicaciones del sistema se dividen en varios niveles de protocolo, puede ser necesario que el modelo de análisis distinga algunos de esos niveles. Si esto es así, debemos identificar clases de interfaz separadas para cada nivel de interés.

- Identificar una clase de control responsable del tratamiento del control y de la coordinación de la realización del caso de uso, y después refinar esta clase de control de acuerdo a los requisitos del caso de uso. Por ejemplo, en algunos casos el control se encapsula mejor dentro de una clase de interfaz, especialmente si el actor maneja gran parte del control. En estos casos la clase de control no es necesaria. En otros casos el control es tan complejo que es mejor encapsularlo en dos o más clases de control. En estos casos es necesario dividir la clase de control.

Al llevar a cabo este paso, deberían tenerse en cuenta, naturalmente, las clases del análisis que ya están presentes en el modelo de análisis. Algunas es probable que se reutilicen en la realización de caso de uso que estamos considerando, y se realizan casi simultáneamente varios casos de uso, lo cual hace más fácil identificar clases del análisis que participan en varias realizaciones de casos de uso.

Debemos recoger en un diagrama de clases las clases del análisis que participan en una realización de caso de uso. Utilizaremos este diagrama para mostrar las relaciones que se utilizan en la realización del caso de uso.

8.6.2.2. Descripción de interacciones entre objetos del análisis Cuando tenemos un esbozo de las clases necesarias para realizar el caso de uso, debemos describir cómo interactúan sus correspondientes objetos del análisis. Esto se hace mediante diagramas de colaboración que contienen las instancias de actores participantes, los objetos del análisis, y sus enlaces. Si el caso de uso tiene flujos o subflujos diferenciados y distintos, suele ser útil crear un diagrama de colaboración para cada flujo. Esto contribuye a hacer más clara la realización del caso de uso, y también hace posible extraer diagramas de colaboración que representan interacciones generales y reutilizables.

Un diagrama de colaboración se crea comenzando por el principio del flujo del caso de uso, y continuando el flujo paso a paso decidiendo qué interacciones de objetos del análisis y de instancias de actor son necesarias para realizarlo. Normalmente los objetos encuentran su sitio natural en la secuencia de interacciones de la realización del caso de uso. Podemos observar lo siguiente sobre estos diagramas de colaboración:

- El caso de uso se invoca mediante un mensaje proveniente de una instancia de un actor sobre un objeto de interfaz.
- Cada clase del análisis identificada en el paso anterior debería tener al menos un objeto que participase en un diagrama de colaboración. Si no lo tiene, la clase del análisis es superflua ya que no participa en ninguna realización de caso de uso.
- Los mensajes no se asocian a operaciones debido a que no especificamos operaciones en las clases del análisis. En cambio, un mensaje debería reflejar el propósito del objeto invocante en la interacción con el objeto invocado. Este "propósito" es el origen de una responsabilidad del objeto receptor y podría llegar a ser incluso el nombre de una responsabilidad.
- Los enlaces en el diagrama normalmente deben ser instancias de asociaciones entre clases del análisis. O bien esas asociaciones ya existen o bien los enlaces definen requisitos sobre las asociaciones. Todas las asociaciones obvias deberían esbozarse en este paso y deberían mostrarse en el diagrama de clases asociado con la realización del caso de uso.
- La secuencia en el diagrama no debería ser nuestro objetivo principal y puede eliminarse si es difícil de mantener o crea confusión en el diagrama. Al contrario, el objetivo principal debería estar en las relaciones (enlaces) entre los objetos y en los requisitos (como se recoge en los mensajes) sobre cada objeto en particular.
- El diagrama de colaboración debería tratar todas las relaciones del caso de uso que se está realizando. Por ejemplo, si el caso de uso A es una especialización de otro caso de uso B mediante una relación de especialización, el diagrama de colaboración que realice el caso de uso A puede requerir el hacer referencia a la realización (es decir, al diagrama de colaboración) del caso de uso B.

En algunos casos es apropiado complementar el diagrama de colaboración con descripciones textuales, especialmente si el mismo caso de uso tiene muchos diagramas de colaboración que lo describen o si hay diagramas que representan flujos complejos. Esas descripciones textuales deberían recogerse en el artefacto flujo de sucesos-análisis de la realización del caso de uso.

8.6.2.3. Captura de requisitos especiales En este paso recogeremos todos los requisitos sobre una realización de caso de uso que se identifican en el análisis pero deberían tratarse en el diseño y en la implementación, tales como los requisitos no funcionales.

Ejemplo **Requisitos especiales sobre una realización de caso de uso**

Los requisitos especiales planteados por la realización del caso de uso Pagar Factura incluyen los siguientes:

- La clase Factura debe ser persistente.
- La clase Gestor de Pedidos debe ser capaz de tratar 10.000 transacciones por hora.

Al capturar estos requisitos, debemos hacer referencia si es posible a los requisitos especiales comunes que habían sido identificados por el arquitecto.

8.6.3. Actividad: analizar una clase

Los objetivos de analizar una clase (*véase* la Figura 8.26) son:

- Identificar y mantener las responsabilidades de una clase del análisis, basadas en su papel en las realizaciones de caso de uso.
- Identificar y mantener los atributos y relaciones de la clase del análisis.
- Capturar requisitos especiales sobre la realización de la clase del análisis.

8.6.3.1. Identificar responsabilidades Las responsabilidades de una clase pueden recopilarse combinando todos lo roles que cumple en diferentes realizaciones de caso de

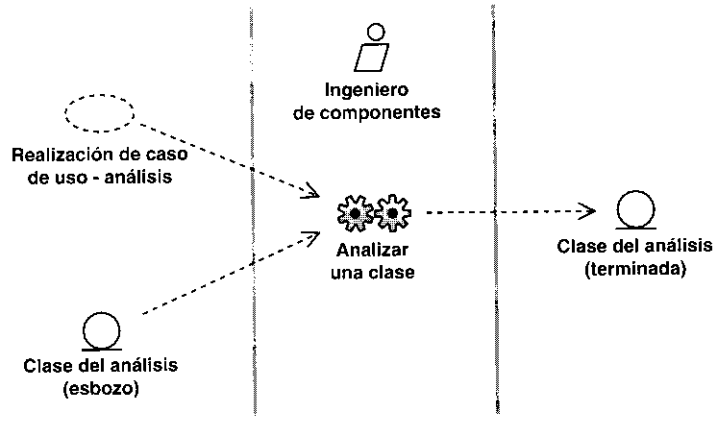


Figura 8.26. Las entradas y los resultados del análisis de una clase.

uso. Podemos identificar todas las realizaciones de caso de uso en las cuales participa la clase mediante el estudio de sus diagramas de clases y de interacción. Recuérdese también que los requisitos de cada realización de caso de uso respecto a sus clases a veces están escritos textualmente en el artefacto flujo de sucesos - análisis de la realización del caso de uso.

Ejemplo **Roles de clase**

Los objetos Factura se crean durante el caso de uso Enviar Factura al Comprador. El vendedor lleva a cabo este caso de uso para solicitar al comprador que pague un pedido (un pedido que fue creado durante los casos de uso Solicitar Bienes y Servicios). Durante este caso de uso, la factura se pasa al comprador, que puede decidir pagarla después.

El pago se lleva a cabo en el caso de uso Pagar Factura, donde el objeto Planificador de Pagos planifica el pago de un objeto Factura. Más adelante se paga la factura, y el objeto Factura se cierra.

Obsérvese, sin embargo, que la misma instancia de factura participa en ambos casos de uso, Enviar Factura al Comprador y Pagar Factura.

Hay varias maneras de recopilar las responsabilidades de una clase. Una técnica simple es extraer las responsabilidades de cada rol uno detrás de otro, añadiendo responsabilidades adicionales o modificando las existentes basándonos en una realización de caso de uso tras otra.

Ejemplo **Responsabilidades de clase**

El Planificador de pagos tiene las siguientes responsabilidades:

- Crear una solicitud de pago.
- Hacer el seguimiento de los pagos que se han planificado y enviar una notificación cuando se ha efectuado o cancelado el pago.
- Iniciar la transferencia de dinero en la fecha debida.
- Notificar una factura cuando se ha planificado para su pago y cuando se ha pagado (es decir, cerrado).

8.6.3.2. Identificación de atributos Un atributo especifica una propiedad de una clase del análisis, y normalmente es necesaria para las responsabilidades de su clase (como se trató en el paso anterior). Deberíamos tener en mente las siguientes normas generales cuando identificamos atributos:

- El nombre de un atributo debería ser un nombre [1,2].
- Recuérdese que el tipo de los atributos debería ser conceptual en el análisis, y, si es posible, no debería verse restringido por el entorno de implementación. Por ejemplo, "cantidad" puede ser un tipo adecuado en el análisis, mientras que su contrapartida en el diseño podría ser "entero".
- Al decidir el tipo de un atributo, debemos intentar reutilizar tipos ya existentes.

- Una determinada instancia de un atributo no puede compartirse por varios objetos del análisis. Si se necesita hacer esto, el atributo debe definirse en su propia clase.
- Si una clase del análisis se hace demasiado difícil de entender por culpa de sus atributos, algunos de esos atributos podrían separarse en clases independientes.
- Los atributos de las clases de entidad suelen ser bastante evidentes. Si una clase de entidad tiene una traza con una clase del dominio o con una clase de entidad del negocio, los atributos de esas clases son una entrada útil.
- Los atributos de las clases de interfaz que interactúan con los actores humanos suelen representar elementos de información manipulados por los actores, tales como campos de texto etiquetados.
- Los atributos de las clases de interfaz que interactúan con los actores que representan sistemas suelen representar propiedades de un interfaz de comunicación.
- Los atributos de las clases de control son poco frecuentes debido a su corto tiempo de vida. Sin embargo, las clases de control pueden poseer atributos que representan valores acumulados o calculados durante la realización de un caso de uso.
- A veces no son necesarios los atributos formales. En su lugar, puede ser suficiente con una sencilla explicación de una propiedad tratada por una clase del análisis, y pueden añadirse a la descripción de las responsabilidades de la clase.
- Si una clase tiene muchos atributos o atributos complejos, podemos mostrarlos en un diagrama de clases aparte, que sólo muestre la sección de atributos.

8.6.3.3. Identificación de asociaciones y agregaciones

Los objetos del análisis interactúan unos con otros mediante enlaces en los diagramas de colaboración. Estos enlaces suelen ser instancias de asociaciones entre sus correspondientes clases. El ingeniero de componentes debería por tanto estudiar los enlaces empleados en los diagramas de colaboración para determinar qué asociaciones son necesarias. Los enlaces pueden implicar la necesidad de referencias y agregaciones entre objetos.

Deberíamos minimizar el número de relaciones entre clases. No son las relaciones del mundo real lo que deberíamos modelar como agregaciones o asociaciones, si no las relaciones que deben existir en respuesta a las demandas de las diferentes realizaciones de caso de uso. En el análisis el centro de atención no debería ser el modelado de rutas de búsqueda óptimas a través de las asociaciones o agregaciones. Eso es mejor tratarlo durante el diseño y la implementación.

El ingeniero de componentes también define la multiplicidad de las asociaciones, los nombres de los roles, autoasociaciones, clases de asociación, roles ordenados, roles cualificados y **asociaciones *n*-arias** (Apéndice A). Puede consultarse [2, 3].

Ejemplo **Una asociación entre clases del análisis**

Una factura es una solicitud de pago de uno o más pedidos (*véase* la Figura 8.27). Esto se representa mediante una asociación con la multiplicidad "1.." (siempre hay al menos un pedido asociado con una factura) y mediante el nombre de rol pedido a pagar.

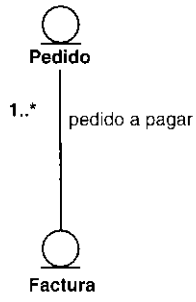


Figura 8.27. Una factura es la solicitud de pago de uno o más pedidos.

Las agregaciones deberían utilizarse cuando los objetos representan:

- Conceptos que se contienen físicamente uno al otro, como un coche que contiene al conductor y a los pasajeros.
- Conceptos que están compuestos uno de otro, como cuando decimos que un coche consta de un motor y ruedas.
- Conceptos que forman una colección conceptual de objetos, como una familia que consta de un padre, una madre, y los niños.

8.6.3.4. Identificación de generalizaciones Las generalizaciones deberían utilizarse durante el análisis para extraer comportamiento compartido y común entre varias clases del análisis diferentes. Las generalizaciones deberían mantenerse en un nivel alto y conceptual, y su objetivo fundamental debería ser hacer el modelo de análisis más fácil de comprender.

Ejemplo **Identificación de generalizaciones**

Facturas y Pedidos tienen responsabilidades similares. Ambas son especializaciones de un más general Objeto de Comercio; *véase* la Figura 8.28.

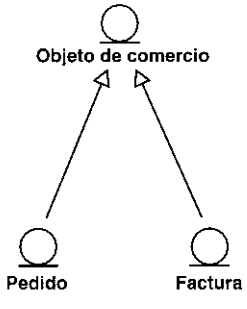


Figura 8.28. Objeto de comercio generaliza factura y pedido.

Durante el diseño, ajustaremos las generalizaciones para que encajen mejor con el entorno de implementación elegido, es decir, con el lenguaje de programación. Una generalización podría desaparecer y convertirse en su lugar en otra relación, como una asociación.

8.6.3.5. Captura de requisitos especiales En este paso recogeremos todos los requisitos de una clase del análisis que se han identificado en el análisis pero que deberían tratarse en el diseño y en la implementación (es decir, requisitos no funcionales). Al llevar a cabo este paso, debemos asegurarnos de estudiar los requisitos especiales de la realización del caso de uso, que pueden contener requisitos adicionales (no funcionales) sobre la clase del análisis.

Ejemplo **Captura de requisitos especiales sobre una clase del análisis**

Las características del requisito de persistencia de la clase Factura podrían definirse de la siguiente manera:

- *Rango de tamaño:* 2 a 24 Kbytes por objeto.
- *Volumen:* hasta 100.000.
- *Frecuencia de actualización:*
  - — Creación/borrado: 1.000 al día.
  - — Actualización: 30 actualizaciones a la hora.
  - — Lectura: 1 acceso a la hora.

Al recoger estos requisitos, debemos hacer referencia si es posible a cualquier requisito especial común identificado por el arquitecto.

8.6.4. Actividad: analizar un paquete

Los objetivos de analizar un paquete, como se muestra en la Figura 8.29, son:

- Garantizar que el paquete del análisis es tan independiente de otros paquetes como sea posible.
- Garantizar que el paquete del análisis cumple su objetivo de realizar algunas clases del dominio o casos de uso.
- Describir las dependencias de forma que pueda estimarse el efecto de los cambios futuros.

Las siguientes son algunas normas generales para esta actividad:

- Definir y mantener las dependencias del paquete con otros paquetes cuyas clases contenidas estén asociadas con él.
- Asegurarnos de que el paquete contiene las clases correctas. Intentar hacer cohesivo el paquete incluyendo sólo objetos relacionados funcionalmente.

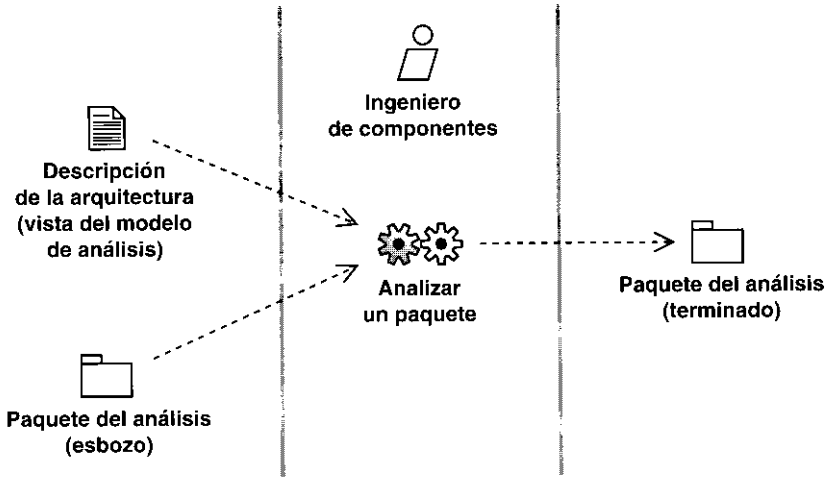


Figura 8.29. Las entradas y los resultados del análisis de un paquete.

- Limitar las dependencias con otros paquetes. Considerar la reubicación de aquellas clases contenidas en paquetes que son demasiado dependientes de otros paquetes.

Ejemplo Dependencias entre paquetes

El paquete Gestión de Facturas de Vendedor contiene una clase Procesamiento de Factura asociada con la clase Cuenta del paquete Gestión de Cuentas. Esto requiere una correspondiente dependencia entre los paquetes (*véase* la Figura 8.30).

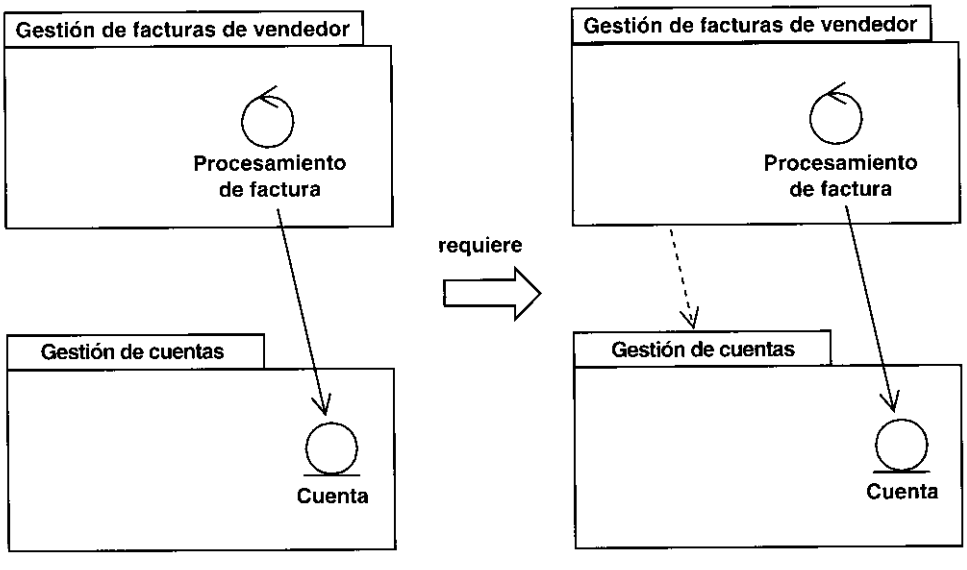


Figura 8.30. Dependencias necesarias entre paquetes.

8.7. Resumen del análisis

El resultado del **flujo de trabajo del análisis** (Apéndice C) es el modelo de análisis, que es un modelo de objetos conceptual que analiza los requisitos mediante su refinamiento y estructuración. El modelo de análisis incluye los siguientes elementos:

- Paquetes del análisis y paquetes de servicio, y sus dependencias y contenidos. Los paquetes del análisis pueden aislar los cambios en un proceso del negocio, el comportamiento de un actor, o en un conjunto de casos de uso estrechamente relacionados. Los paquetes de servicio aislarán los cambios en determinados servicios ofrecidos por el sistema, y constituyen un elemento esencial para construir pensando en la reutilización durante el análisis.

- Clases del análisis, sus responsabilidades, atributos, relaciones, y requisitos especiales. Cada una de las clases de control, entidad e interfaz aislarán los cambios al comportamiento (estereotípico) y a la información que representan. Un cambio en la interfaz de usuario o en una interfaz de comunicación normalmente se ubica en una o más clases de interfaz; un cambio en la información duradera, y a menudo persistente, gestionada por el sistema normalmente se ubica en una o más clases de entidad; un cambio en el control, coordinación, secuencia, transacciones, y a veces en la lógica del negocio compleja, que implica a varios objetos (de interfaz y/o de entidad) normalmente se ubica en una o más clases de control.

- Realizaciones de casos de uso-análisis, que describen cómo se refinan los casos de uso en términos de colaboraciones dentro del modelo de análisis y de sus requisitos especiales. Las realizaciones de casos de uso aislarán los cambios en los casos de uso, debido a que si cambia un caso de uso, debe cambiarse también su realización.

- La vista de la arquitectura del modelo de análisis, incluyendo sus elementos significativos para la arquitectura. La **vista de la arquitectura** (Apéndice C) aislará los cambios de la arquitectura.

Como presentaremos en el siguiente capítulo, el modelo de análisis se considera la entrada fundamental para las actividades de diseño subsiguientes. Cuando utilizamos el modelo de análisis con esta intención, conservamos en todo lo posible la estructura que define durante el diseño del sistema, mediante el tratamiento de la mayor parte de los requisitos no funcionales y otras restricciones relativas al entorno de la implementación. Más en concreto, el modelo de análisis influirá en el modelo de diseño de las siguientes maneras:

- Los paquetes del análisis y los paquetes de servicio tendrán una influencia fundamental en los subsistemas de diseño y en los subsistemas de servicio, respectivamente, en las capas específicas de la aplicación y generales de la aplicación. En muchos casos tendremos una traza uno a uno (isomórfica) entre paquetes y los correspondientes subsistemas.

- Las clases del análisis servirán como especificaciones al diseñar las clases. Se requieren diferentes tecnologías y habilidades al diseñar clases del análisis con diferentes estereotipos; por ejemplo, el diseño de las clases de entidad normalmente requiere el uso de tecnologías de base de datos, mientras que el diseño de clases de interfaz normalmente requiere el uso de tecnologías de interfaz de usuario. Sin embargo, las clases del análisis y sus responsabilidades, atributos, y relaciones sirven como una entrada (lógica) para la creación de las correspondientes operaciones, atributos, y relaciones de las clases de

diseño. Además, la mayoría de los requisitos especiales recogidos sobre una clase del análisis serán tratados por las clases de diseño correspondientes cuando se tienen en cuenta tecnologías como las de bases de datos y de interfaces de usuario.

- Las realizaciones de casos de uso-análisis tienen dos objetivos principales. Uno es ayudar a crear especificaciones más precisas para el caso de uso. En lugar de detallar cada caso de uso en el modelo de casos de uso con diagramas de estado o diagramas de actividad, la descripción de un caso de uso mediante una colaboración entre clases del análisis da como resultado una especificación formal completa de los requisitos del sistema. Las realizaciones de casos de uso-análisis también sirven como entradas al diseño de los casos de uso. Ayudarán a identificar las clases del diseño que deben participar en la correspondiente realización de caso de uso-diseño. También son útiles porque esbozan una secuencia inicial de interacciones entre los objetos del diseño. Además, la mayoría de los requisitos especiales recogidos sobre una realización de caso de uso-análisis se tratarán en la correspondiente realización de caso de uso-diseño al considerar tecnologías como las de bases de datos y de interfaces de usuario.
- La vista de la arquitectura del modelo de análisis se utiliza como entrada en la creación de la vista de la arquitectura del modelo de diseño. Es muy probable que los elementos de las diferentes vistas (de los diferentes modelos) tengan trazas entre ellos. Esto es debido a que la noción de relevancia para la arquitectura tiende a fluir suavemente a lo largo de los diferentes modelos mediante dependencias de traza.

8.8. Referencias

[1] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard, *Object-Oriented Software Engineering: A Use-Case-Driven Approach,* Reading, MA: Addison-Wesley, 1992. (Revised fourth printing, 1993.)

[2] James Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design,* Englewood Cliffs, NJ: Prentice Hall, 1991.

[3] OMG Unified Modeling Language Specification. Object Management Group, Framinham, MA, 1998. Internet: www.omg.org.

Capítulo 9

Diseño

9.1. Introducción

En el diseño modelamos el sistema y encontramos su forma (incluida la arquitectura) para que soporte todos los requisitos —incluyendo los requisitos no funcionales y otras restricciones— que se le suponen. Una entrada esencial en el diseño es el resultado del análisis, esto es, el modelo de análisis (*véase* la Tabla 9.1). El modelo de análisis proporciona una comprensión detallada de los requisitos. Y lo que es más importante, impone una estructura del sistema que debemos esforzarnos por conservar lo más fielmente posible cuando demos forma al sistema (referencia al Capítulo 8, Sección 8.2). En concreto, los propósitos del diseño son:

- Adquirir una comprensión en profundidad de los aspectos relacionados con los requisitos no funcionales y restricciones relacionadas con los lenguajes de programación, componentes reutilizables, sistemas operativos, tecnologías de distribución y concurrencia, tecnologías de interfaz de usuario, tecnologías de gestión de transacciones, etc.
- Crear una entrada apropiada y un punto de partida para actividades de implementación subsiguientes capturando los requisitos o subsistemas individuales, interfaces y clases.
- Ser capaces de descomponer los trabajos de implementación en partes más manejables que puedan ser llevadas a cabo por diferentes equipos de desarrollo, teniendo en cuenta la posible concurrencia. Esto resulta útil en los casos en los que la descomposición no

Tabla 9.1. Breve comparación entre el modelo de análisis y el modelo de diseño

Modelo de análisis	Modelo de diseño
Modelo conceptual, porque es una abstracción del sistema y permite aspectos de la implementación.	Modelo físico, porque es un plano de la implementación.
Genérico respecto al diseño (aplicable a varios diseños).	No genérico, específico para una implemetación.
Tres estereotipos conceptuales sobre las clases: Control, Entidad e Interfaz.	Cualquier número de estereotipos (físicos) sobre las clases, dependiendo del lenguaje de implementación.
Menos formal.	Más formal.
Menos caro de desarrollar (ratio al diseño 1:5).	Más caro de desarrollar (ratio al análisis 5:1).
Menos capas.	Más capas.
Dinámico (no muy centrado en la secuencia).	Dinámico (muy centrado en las secuencias).
Bosquejo del diseño del sistema, incluyendo su arquitectura.	Manifiesto del diseño del sistema, incluyendo su arquitectura (una de sus vistas).
Creado principalmente como "trabajo de a pie" en talleres o similares.	Creado principalmente como "programación visual" en ingeniería de ida y vuelta; el modelo de diseño es realizado según la ingeriería de ida y vuelta con el modelo de implemetación (descrito en el Capítulo 10).
Puede no estar mantenido durante todo el ciclo de vida del software.	Debe ser mantenido durante todo el ciclo de vida del software.
Define una estructura que es una entrada esencial para modelar el sistema —incluyendo la creación del modelo de diseño.	Da forma al sistema mientras que intenta preservar la estructura definida por el modelo de análisis lo más posible.

puede ser hecha basándose en los resultados de la captura de requisitos (incluyendo el modelo de casos de uso) o análisis (incluyendo el modelo de análisis). Un ejemplo podría ser aquellos casos en los que la implementación de estos resultados no es directa.

- Capturar las interfaces entre los subsistemas antes en el ciclo de vida del software. Esto ayuda cuando reflexionamos sobre la arquitectura y cuando utilizamos interfaces como elementos de sincronización entre diferentes equipos de desarrollo.
- Ser capaces de visualizar y reflexionar sobre el diseño utilizando una notación común.
- Crear una abstracción sin costuras de la implementación del sistema, en el sentido de que la implementación es un refinamiento directo del diseño que rellena lo existente sin cambiar la estructura. Esto permite la utilización de tecnologías como la generación de código y la ingeniería de ida y vuelta entre el diseño y la implementación.

En este capítulo y los siguientes presentaremos cómo conseguir estos objetivos. Enfocamos el flujo de trabajo del diseño de una forma muy similar a como hicimos en el flujo de trabajo del análisis (*véase* Figura 9.1).

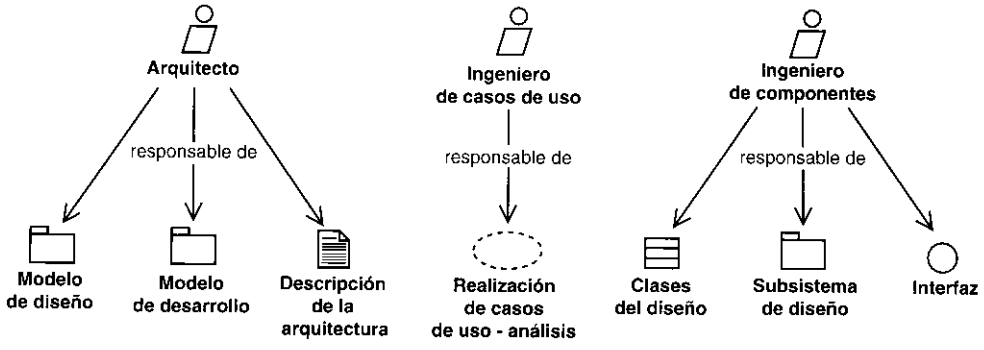


Figura 9.1. Trabajadores y artefactos involucrados en el diseño.

9.2. El papel del diseño en el ciclo de vida del software

El diseño es el centro de atención al final de la fase de elaboración y el comienzo de las iteraciones de construcción (*véase* Figura 9.2). Esto contribuye a una arquitectura estable y sólida y a crear un plano del modelo de implementación. Más tarde, durante la fase de construcción, cuando la arquitectura es estable y los requisitos están bien entendidos, el centro de atención se desplaza a la implementación.

No obstante, el modelo de diseño está muy cercano al de implementación, lo que es natural para guardar y mantener el modelo de diseño a través del ciclo de vida completo del software. Esto es especialmente cierto en la ingeniería de ida y vuelta, donde el modelo de diseño se puede utilizar para visualizar la implementación y para soportar las técnicas de programación gráfica.

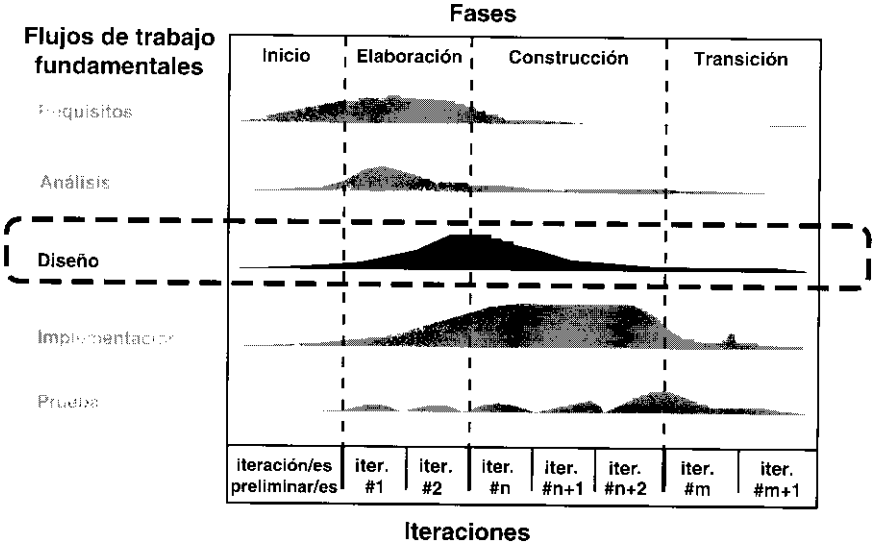


Figura 9.2. El diseño.

9.3. Artefactos

9.3.1. Artefacto: modelo de diseño

El modelo de diseño es un modelo de objetos que describe la realización física de los casos de uso centrándose en cómo los requisitos funcionales y no funcionales, junto con otras restricciones relacionadas con el entorno de implementación, tienen impacto en el sistema a considerar. Además, el modelo de diseño sirve de abstracción de la implementación del sistema y es, de ese modo, utilizada como una entrada fundamental de las actividades de implementación.

El modelo de diseño define la jerarquía que se ilustra en la Figura 9.3.

El modelo de diseño se representa por un sistema de diseño que denota el subsistema de nivel más alto del modelo. La utilización de otro subsistema es, entonces, una forma de organización del modelo de diseño en porciones más manejables.

Los subsistemas de diseño y clases del diseño representan **abstracciones** (Apéndice C) del subsistema y componentes de la implementación del sistema. Estas abstracciones son directas, y representan una sencilla correspondencia entre el diseño y la implementación.

En el modelo de diseño, los casos de uso son realizados por las clases de diseño y sus objetos. Esto se representa por colaboraciones en el modelo de diseño y denota *realización de caso de uso-diseño*. Adviértase que realización de caso de uso-diseño es diferente de la realización de casos de uso-análisis. Lo anterior describe cómo se realiza un caso de uso en términos de interacción entre objetos del diseño, mientras que lo último describe cómo se realiza un caso de uso en términos de interacción entre objetos del análisis.

Los artefactos del modelo de diseño se presentan en detalle en las siguientes secciones.

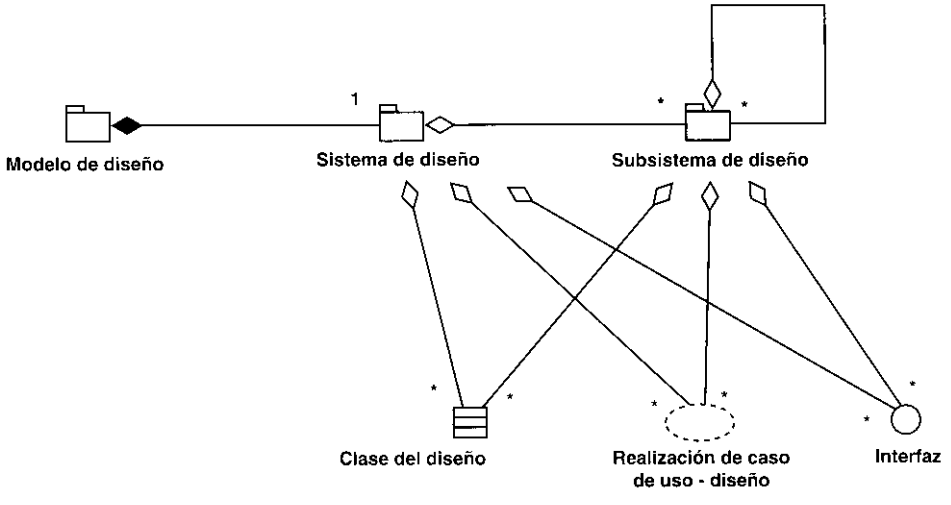


Figura 9.3. El modelo de diseño es un jerarquía de subsistemas de diseño que contienen clases del diseño, realizaciones de caso de uso-diseño e interfaces.

9.3.2. Artefacto: clase del diseño

Una clase de diseño es una abstracción sin costuras de una clase o construcción similar en la implementación del sistema (*véase* Figura 9.4). Esta abstracción es sin costuras en el siguiente sentido:

- El lenguaje utilizado para especificar una clase del diseño es lo mismo que el lenguaje de programación. Consecuentemente, la operaciones, parámetros, atributos, tipos y demás son especificados utilizando la sintaxis del lenguaje de programación elegido.
- La visibilidad de los atributos y las operaciones de una clase de diseño se especifica con frecuencia. Por ejemplo, las palabras clave *public, protected, private* son usadas muy a menudo en C++.
- Las relaciones de aquellas clases del diseño implicadas con otras clases, a menudo tienen un significado directo cuando la clase es implementada. Por ejemplo, la generalización o algún estereotipo de generalización tiene una semántica que se corresponde con el significado de generalización (o herencia) en el lenguaje de programación. Esto es, las asociaciones y agregaciones a menudo se corresponden con variables (atributos) de clases en la implementación para proporcionar referencias entre objetos.
- Los métodos (o lo que es lo mismo, las realizaciones de operaciones) de una clase del diseño tienen correspondencia directa con el correspondiente método en la implementación de las clases (esto es, en el código). Si los métodos se especifican en el diseño, se suelen especificar en lenguaje natural, o en pseudocódigo, y por eso pueden ser utilizados como comentarios en las implementaciones del método. Esto es una de las principales abstracciones entre diseño e implementación y es raramente necesario por lo que recomendamos que el mismo desarrollador diseñe e implemente una clase (*véase* Sección 9.4.3)
- Una clase de diseño puede postponer el manejo de algunos requisitos para las subsiguientes actividades de implementación, indicándolos como requisitos de implementación de la clase. Esto hace posible postponer decisiones que son inapropiadas de manejar en el modelo de diseño, como las que tienen que ver con el código de la clase.
- Una clase de diseño a menudo aparece como un estereotipo sin costuras que se corresponde con una construcción en el lenguaje de programación dado. Por ejemplo, una clase de diseño para una aplicación en Visual Basic podría estereoriparse como un *«class module»*, *«form»*, *«user control»*, etc.

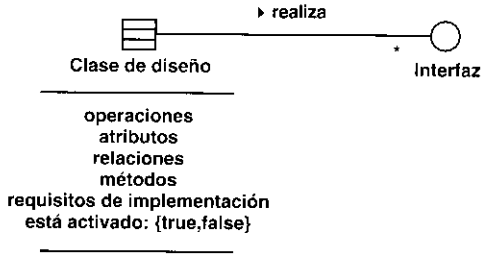


Figura 9.4. Los atributos clave y las asociaciones de una clase de diseño.

- Una clase de diseño puede realizar —y por tanto, proporcionar— interfaces si tiene sentido hacerlo en el lenguaje de programación. Por ejemplo, una clase de diseño que representa una clase Java puede proporcionar una interfaz.
- Una clase de diseño puede activarse, implicando que objetos de la clase mantengan su propio hilo de control y se ejecuten concurrentemente con otros objetos activos. No obstante, las clases del diseño no están normalmente activas, lo que implica que sus objetos se ejecuten en el espacio de direcciones y bajo el control de otros objetos activos. Por contra, la semántica detallada de esto es dependiente del lenguaje de programación y las tecnologías de distribución y concurrencia que se utilicen. Adviértase que hay diferencias significativas entre la semántica de las clases activas y las que no son activas. Por esto, las clases activas deberían, como alternativa, residir en su propio modelo de proceso en lugar de en el modelo del diseño. En concreto, esto puede ser apropiado cuando hay muchas clases activas cuyos objetos tienen interacciones complejas —como por ejemplo, en algunos sistemas en tiempo real.

Ejemplo **La clase de diseño Factura**

La Figura 9.5 ilustra la clase de diseño Factura tal y como se elabora en el diseño. El atributo Cuenta sugerido en el análisis ha sido convertido en una asociación con la clase Cuenta.

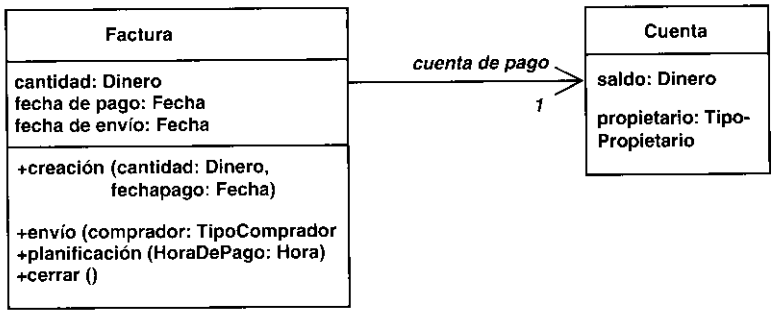


Figura 9.5. La clase de diseño Factura con sus atributos, operaciones y una asociación que asocia los objetos Factura con los objetos Cuenta.

9.3.3. Artefacto: realización de caso de uso-diseño

Una realización de caso de uso – diseño es una colaboración en el modelo de diseño que describe cómo se realiza un caso de uso específico, y cómo se ejecuta, en términos de clases de diseño y sus objetos. Una *realización de caso de uso-diseño* proporciona una traza directa a una realización de caso de uso-análisis en el modelo de análisis (*véase* Figura 9.6). Nótese que una realización de caso de uso-diseño de esta forma también puede tener una traza a un caso de uso en el modelo de casos de uso a través de una realización de caso de uso-análisis.

Cuando el modelo de análisis no va a mantenerse a lo largo del ciclo de vida del software pero en cambio se utiliza sólo para crear un buen diseño, no tendremos realización de caso de

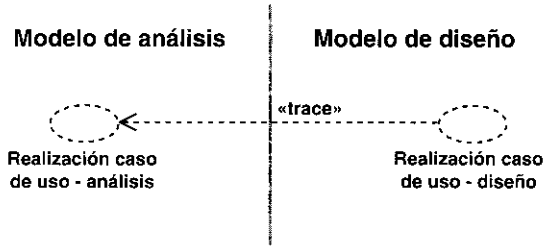


Figura 9.6. Trazas existentes entre realizaciones de caso de uso en diferentes modelos.

uso-análisis. La dependencia de traza de una realización de caso de uso-diseño irá en este caso directamente hasta el caso de uso en el modelo de casos de uso.

Una realización de caso de uso-diseño tiene una descripción de flujo de eventos textual, diagramas de clases que muestra sus clases de diseño participantes, y diagramas de interacción que muestran la realización de un flujo o escenario concreto de un caso de uso en términos de interacción entre objetos del diseño (*véase* Figura 9.7). Si fuera necesario, los diagramas pueden mostrar también los subsistemas e interfaces implicados en la realización de casos de uso (es decir, los subsistemas que contienen las clases participantes del diseño).

Una realización de caso de uso-diseño proporciona una realización física de la realización de caso de uso-análisis para la que es trazado, y también gestiona muchos requisitos no funcionales (es decir, requisitos especiales) capturados de la realización de caso de uso-análisis. Por consiguiente, una realización de caso de uso-diseño puede, como pueden las clases diseñadas, postponer el manejo de algunos requisitos hasta las subsiguientes activiadades de implementación anotándolas como requisitos de implementación en la realización.

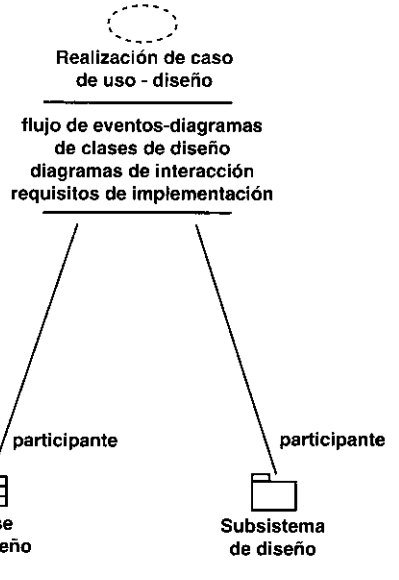


Figura 9.7. Los atributos clave y asociaciones de una realización de caso de uso-diseño.

9.3.3.1. Diagramas de clases

Una clase de diseño y sus objetos, y de ese modo también los subsistemas que contienen las clases de diseño, a menudo participan en varias realizaciones de casos de uso. También puede darse el caso de algunas operaciones, atributos y asociaciones sobre una clase específica que son relevantes para sólo una realización de caso de uso. Esto es importante para coordinar todos los requisitos que diferentes realizaciones de casos de uso imponen a una clase, a sus objetos y a los subsistemas que contiene. Para manejar todo esto, utilizamos diagramas de clases conectados a una realización de caso de uso, mostrando sus clases participantes, subsistemas y sus relaciones. De esta forma podemos guardar la pista de los elementos participantes en una realización del caso de uso.

Los ejemplos de diagramas de clases se proporcionan en la Sección 9.5.2.1 y 9.5.2.3.

9.3.3.2. Diagramas de interacción

La secuencia de acciones en un caso de uso comienza cuando un actor invoca el caso de uso mediante el envío de algún tipo de mensaje al sistema. Si consideramos el "interior" del sistema, tendremos algún objeto de diseño que recibe el mensaje del actor. Después el objeto de diseño llama a algún otro objeto, y de esta manera los objetos implicados interactúan para realizar y llevar a cabo el caso de uso. En el diseño, es preferible representar esto con diagramas de secuencia ya que nuestro centro de atención principal es el encontrar secuencias de interacciones detalladas y ordenadas en el tiempo.

En algunos casos incluimos subsistemas en los diagramas de secuencia para describir cuáles de ellos participan en una determinada realización de caso de uso, y quizás qué interfaces intervienen de entre los que proporcionan esos subsistemas. Gracias a ello, podemos diseñar los casos de uso a un nivel alto antes de que se hayan desarrollado los diseños internos de los subsistemas que intervienen. Esto es útil, por ejemplo, cuando tenemos que identificar las interfaces de los subsistemas en una fase temprana del ciclo de vida del software, antes de haber desarrollado el diseño interno.

En los diagramas de secuencia, mostramos las interacciones entre objetos mediante transferencia de mensajes entre objetos o subsistemas. Cuando decimos que un subsistema "recibe" un mensaje, queremos decir en realidad que es un objeto de una clase del subsistema el que recibe el mensaje. Cuando un subsistema "envía" un mensaje, realmente es un objeto de una clase del subsistema el que envía el mensaje. El nombre del mensaje debería indicar una operación del objeto que recibe la invocación o de una interfaz que el objeto proporciona.

En la Sección 9.5.2.2 damos ejemplos de estos diagramas de interacción.

9.3.3.3. Flujo de sucesos-diseño

Los diagramas de una realización de caso de uso, y especialmente los diagramas de interacción, son difíciles de interpretar por sí solos. Por esto, puede ser útil el artefacto flujo-de-sucesos-diseño, que es una descripción textual que explica y complementa a los diagramas y a sus etiquetas. El texto debería redactarse en términos de objetos que interactúan para llevar a cabo el caso de uso, o bien en términos de los subsistemas que participan en él. Sin embargo, la descripción no debería hacer mención de ninguno de los atributos, operaciones y asociaciones de los objetos, ya que ello haría que la descripción fuese difícil de mantener debido a que los atributos, operaciones y asociaciones de las clases de diseño cambian con frecuencia. Además, en la descripción no debería aparecer ninguna operación de interfaz, en caso de que utilicemos interfaces en los diagramas. Mediante esta técnica, minimizamos la necesidad de tener que actualizar las

descripciones flujo-de sucesos-diseño cuando se actualizan los diagramas descritos en ellas, especialmente cuando cambian las operaciones que se muestran en los mensajes de los diagramas de interacción.

El artefacto-flujo-de-sucesos-diseño es especialmente útil cuando tenemos realizaciones de caso de uso descritas por múltiples diagramas de interacción o cuando hay diagramas que representan flujos complejos. Obsérvese lo siguiente:

- El flujo-de-sucesos-diseño de una realización de caso de uso no es local a un determinado diagrama de secuencia. Por tanto, puede utilizarse para describir las relaciones entre varios diagramas.
- Las etiquetas (marcas de tiempo o descripciones de acciones durante una activación) de un diagrama de secuencia son locales al diagrama. Sin embargo, si ponemos muchas etiquetas, podemos oscurecer el diagrama. Si ocurre esto, puede omitirse el texto de algunas etiquetas, y podemos incluirlo en su lugar en el flujo de sucesos-diseño.

Cuando se utilicen tanto etiquetas como un flujo de sucesos-diseño, ambas representaciones deberían ser complementarias.

Ofrecemos ejemplos de descripciones flujo de sucesos-diseño en la Sección 9.5.2.2.

9.3.3.4. Requisitos de la implementación Los requisitos de la implementación son una descripción textual que recoge requisitos, tales como los requisitos no funcionales, sobre una realización de caso de uso. Nos referimos a requisitos que se capturan sólo en la fase de diseño, pero que es mejor tratar en la implementación. Algunos de estos requisitos pueden haber sido identificados en flujos de trabajo anteriores y por tanto sólo se cambian a una realización de caso de uso. Sin embargo, puede que algunos de ellos sean requisitos nuevos o derivados, que se identifican a medida que avanza el trabajo del diseño.

Ofrecemos ejemplos de requisitos de implementación sobre una realización de caso de uso-diseño en la Sección 9.5.2.5.

9.3.4. Artefacto: subsistema de diseño

Los subsistemas de diseño son una forma de organizar los artefactos del modelo de diseño en piezas más manejables (*véase* la Figura 9.8). Un subsistema puede constar de clases del diseño, realizaciones de caso de uso, interfaces y otros subsistemas (recursivamente). Por otro lado, un subsistema puede proporcionar interfaces que representan la funcionalidad que exportan en términos de operaciones.

Un subsistema debería ser cohesivo; es decir, sus contenidos deberían encontrarse fuertemente asociados. Además, los subsistemas deberían ser débilmente acoplados; esto es, sus dependencias entre unos y otros, o entre sus interfaces, deberían ser mínimas.

Los subsistemas de diseño también deberían poseer las siguientes características:

- Los subsistemas pueden representar una separación de aspectos del diseño. Por ejemplo, en un sistema grande, algunos subsistemas pueden desarrollarse por separado, y quizá de manera simultánea, por equipos de desarrollo diferentes con aptitudes de diseño distintas.

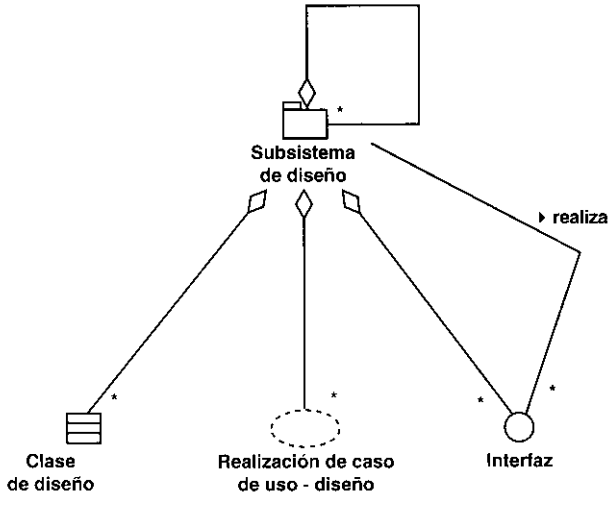


Figura 9.8. Los contenidos de un subsistema y sus asociaciones fundamentales.

- Las dos capas de aplicación de más alto nivel y sus subsistemas dentro del modelo de diseño suelen tener trazas directas hacia paquetes y/o clases del análisis.
- Los subsistemas pueden representar componentes de grano grueso en la implementación del sistema; es decir, componentes que proporcionan varios interfaces compuestos a partir de otros varios componentes de grano más fino, como los que especifican clases de implementación individuales, y que se convierten ellos mismos en ejecutables, ficheros binarios o entidades similares que pueden distribuirse en diferentes nodos.
- Los subsistemas pueden representar productos software reutilizados que han sido encapsulados en ellos. Por tanto, pueden utilizarse los subsistemas en el modelo de diseño para representar la integración de productos software reutilizados. Subsistemas de este tipo residen en las capas intermedias y de software del sistema.
- Los subsistemas pueden representar sistemas heredados (o parte de ellos) encapsulándolos. Por tanto, podemos utilizar subsistemas para incluir sistemas heredados en el modelo de diseño.

En la Sección 9.5.1.2, daremos más detalles sobre los subsistemas y sus aspectos prácticos, junto con un método para identificarlos a ellos y a sus interfaces.

9.3.4.1. Subsistemas de servicio Los subsistemas de servicio diseño se utilizan en un nivel inferior de la jerarquía de subsistemas de diseño por el mismo motivo por el cual utilizábamos los paquetes de servicio en el modelo de diseño, es decir, para prepararnos para los cambios en servicios individuales, aislando los cambios en los correspondientes subsistemas de servicio. Puede encontrarse una descripción de qué es un servicio en la Sección 8.4.4.1 del Capítulo 8.

La identificación de subsistemas de servicio se basa en los paquetes de servicio del modelo de análisis, y normalmente existe una traza uno-a-uno (isomórfica) entre ambos. En consecuencia, los subsistemas de servicio son más comunes en las dos capas superiores (la capa

específica de la aplicación y la capa general de la aplicación). Sin embargo, los subsistemas de servicio deben tratar más aspectos que sus correspondientes paquetes de servicio, por los siguientes motivos:

- Los subsistemas de servicio pueden tener que ofrecer sus servicios en términos de interfaces y de sus operaciones.
- Los subsistemas de servicio contienen clases del diseño en lugar de clases del análisis. Los subsistemas de servicio se enfrentan por tanto a muchos requisitos no funcionales y a muchas otras restricciones asociadas al entorno de la implementación. En consecuencia, los subsistemas de servicio tienden a contener más clases que sus correspondientes paquetes de servicio, y puede que requieran una descomposición adicional en subsistemas más pequeños sólo para reducir su tamaño.
- Un subsistema de servicio suele dar lugar a un componente ejecutable o binario en la implementación. Pero en algunos casos, debemos descomponer un subsistema de servicio, y cada una de sus partes se distribuirá en un nodo diferente, lo cual puede implicar que se necesita un componente binario o ejecutable para cada nodo. En estos casos, puede que tengamos que descomponer el subsistema de servicio en subsistemas más pequeños, y cada uno de ellos encapsulará la funcionalidad que debe distribuirse en cada nodo particular.

Ofrecemos ejemplos de subsistemas de servicio en la Sección 9.5.1.2.1.

Obsérvese que la manera general de organizar los artefactos del modelo de diseño sigue siendo la utilización de subsistemas de diseño normales como se trató en la sección anterior. Sin embargo, incluimos aquí un estereotipo «service subsystem» para poder diferenciar de manera explícita los subsistemas que representan servicios. Esto es especialmente importante en sistemas grandes (que contienen muchos subsistemas) para ser capaces de diferenciar de forma fácil los distintos tipos de subsistemas. Recuérdese que una motivación parecida justifica el estereotipo «service package» que presentamos en el Capítulo 8.

9.3.5. Artefacto: interfaz

Las interfaces se utilizan para especificar las operaciones que proporcionan las clases y los subsistemas del diseño (*véase* la Figura 9.9).

Una clase del diseño que proporcione una interfaz debe proporcionar también métodos que realicen las operaciones de la interfaz. Un subsistema que proporcione una interfaz debe

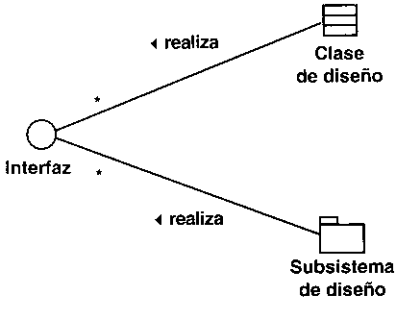


Figura 9.9. Las asociaciones fundamentales de una interfaz.

contener también clases del diseño u otros subsistemas (recursivamente) que proporcionen la interfaz.

Las interfaces constituyen una forma de separar la especificación de la funcionalidad en términos de operaciones de sus implementaciones en términos de métodos. Esta distinción hace independiente de la implementación de la interfaz a cualquier cliente que dependa de ella. Podemos sustituir una implementación concreta de una interfaz, como puede ser una clase o un subsistema del diseño, por otra implementación sin tener que cambiar los clientes.

La mayoría de las interfaces entre subsistemas se consideran relevantes para la arquitectura debido a que definen las interacciones permitidas entre los subsistemas. En algunos casos, también es útil diseñar interfaces estables pronto dentro del ciclo de vida del software, antes de implementar mediante subsistemas la funcionalidad que representan. De esta forma, los equipos de desarrollo encargados del diseño de los subsistemas pueden considerar estas interfaces como requisitos, y pueden utilizarse también como instrumentos de sincronización entre diferentes equipos que pueden estar trabajando de manera simultánea con diferentes subsistemas [2].

En la Sección 9.5.1.2.4, daremos más detalles sobre las interfaces y sus aspectos prácticos, junto con un método para identificarlas.

9.3.6. Artefacto: descripción de la arquitectura (vista del modelo de diseño)

La descripción de la arquitectura contiene una **vista de la arquitectura del modelo de diseño** (Apéndice C), que muestra sus artefactos relevantes para la arquitectura (Figura 9.10).

Suelen considerarse significativos para la arquitectura los siguientes artefactos del modelo de diseño:

- La descomposición del modelo de diseño en subsistemas, sus interfaces, y las dependencias entre ellos. Esta descomposición es muy significativa para la arquitectura en general, debido a que los subsistemas y sus interfaces constituyen la estructura fundamental del sistema.
- Clases del diseño fundamentales, como clases que poseen una traza con clases del análisis significativas, clases activas[1], y clases del diseño que sean generales y centrales, que

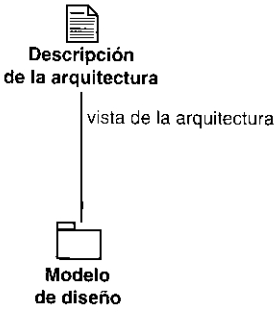


Figura 9.10. La descripción de la arquitectura contiene una vista arquitectónica del modelo de diseño.

[1] Si vamos a ubicar las clases activas en su propio modelo de procesos (como se trató anteriormente), deberían mostrarse en cambio en la vista de la arquitectura del modelo de procesos.

representen mecanismos de diseño genéricos, y que tengan muchas relaciones con otras clases del diseño. Normalmente basta con considerar significativas para la arquitectura a las clases abstractas, y no a sus subclases, a menos que las subclases representen algún comportamiento interesante y significativo para la arquitectura diferente al de la clase abstracta.

- Realizaciones de caso de uso-diseño que describan alguna funcionalidad importante y crítica que debe desarrollarse pronto dentro del ciclo de vida del software, que impliquen muchas clases del diseño y por tanto tengan una cobertura amplia, posiblemente a lo largo de varios subsistemas, o que impliquen clases del diseño fundamentales como las que mencionamos en el punto anterior. Normalmente sucede que los casos de uso correspondientes se encuentran en la vista arquitectónica del modelo de casos de uso, y que las correspondientes realizaciones de caso de uso-análisis se encuentran en la vista arquitectónica del modelo de análisis.

En las Secciones 9.5.1.2, 9.5.1.3 y 9.5.1.4 ofrecemos ejemplos de lo que podría incluirse en la vista arquitectónica del modelo de diseño.

9.3.7. Artefacto: modelo de despliegue

El modelo de despliegue es un modelo de objetos que describe la distribución física del sistema en términos de cómo se distribuye la funcionalidad entre los nodos de cómputo (*véase* la Figura 9.11). El modelo de despliegue se utiliza como entrada fundamental en las actividades de diseño e implementación debido a que la distribución del sistema tiene una influencia principal en su diseño.

Podemos observar lo siguiente sobre el modelo de despliegue:

- Cada nodo representa un recurso de cómputo, normalmente un procesador o un dispositivo hardware similar.
- Los nodos poseen relaciones que representan medios de comunicación entre ellos, tales como *Internet, intranet, bus*, y similares.
- El modelo de despliegue puede describir diferentes configuraciones de red, incluidas las configuraciones para pruebas y para simulación.

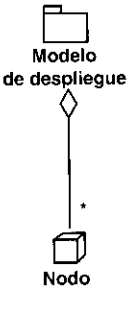


Figura 9.11. El modelo de despliegue contiene nodos.

- La funcionalidad (los procesos) de un nodo se definen por los componentes que se distribuyen sobre ese nodo.
- El modelo de despliegue en sí mismo representa una correspondencia entre la arquitectura *software* y la arquitectura *del sistema* (el hardware).

Daremos ejemplos del modelo de despliegue en la Sección 9.5.1.1.

9.3.8. Artefacto: descripción de la arquitectura (vista del modelo de despliegue)

La descripción de la arquitectura contiene una **vista de la arquitectura del modelo de despliegue** (Apéndice C), que muestra sus artefactos relevantes para la arquitectura (*véase* la Figura 9.12).

Debido a su importancia, deberían mostrarse todos los aspectos del modelo de despliegue en la vista arquitectónica, incluyendo la correspondencia de los componentes sobre los nodos tal como se identificó durante la implementación.

En la Sección 9.5.1.1, ofrecemos ejemplos de lo que podría incluirse en la vista arquitectónica del modelo de despliegue.

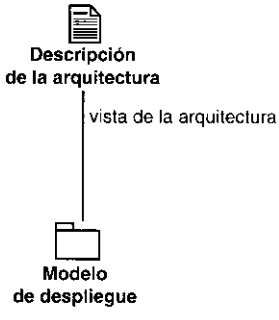


Figura 9.12. La descripción de la arquitectura contiene una vista arquitectónica del modelo de despliegue.

9.4. Trabajadores

9.4.1. Trabajador: arquitecto

En el diseño, el arquitecto es responsable de la integridad de los modelos de diseño y de despliegue, garantizando que los modelos son correctos, consistentes y legibles en su totalidad (*véase* la Figura 9.13). Al igual que en el modelo de análisis, puede incluirse, para sistemas grandes y complejos, un trabajador aparte para asumir las responsabilidades del subsistema de más alto nivel del modelo de diseño (el sistema de diseño).

Los modelos son correctos cuando realizan la funcionalidad, y sólo la funcionalidad, descrita en el modelo de casos de uso, en los requisitos adicionales, y en el modelo de análisis.

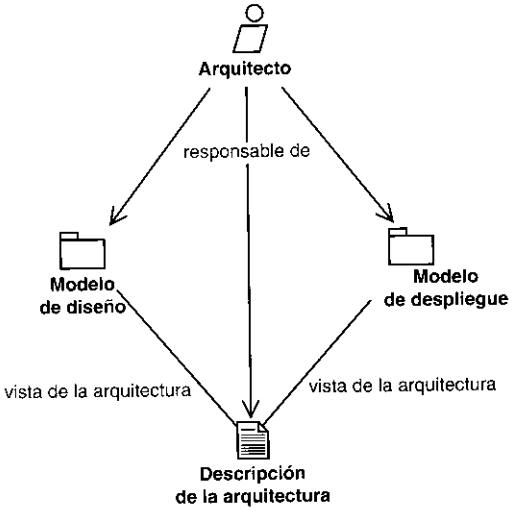


Figura 9.13. Las responsabilidades del arquitecto en el diseño.

El arquitecto también es responsable de la arquitectura de los modelo de diseño y despliegue, es decir, de la existencia de sus partes significativas para la arquitectura, como se muestran en las vistas arquitectónicas de esos modelos. Recuérdese que esas vistas son parte de la descripción de la arquitectura del sistema.

Obsérvese que el arquitecto no es responsable del desarrollo y mantenimiento continuos de los distintos artefactos del modelo de diseño. Estos se encuentran bajo la responsabilidad de los correspondientes ingenieros de casos de uso y de componentes (*véase* las Secciones 9.4.2 y 9.4.3).

9.4.2. Trabajador: ingeniero de casos de uso

El ingeniero de casos de uso es responsable de la integridad de una o más realizaciones de casos de uso-diseño, y debe garantizar que cumplen los requisitos que se esperan de ellos (*véase* la Figura 9.14). Una realización de caso de uso-diseño debe realizar correctamente el

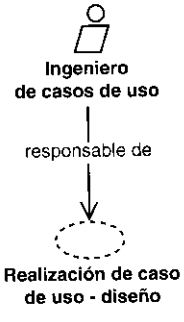


Figura 9.14. Las responsabilidades del ingeniero de casos de uso en el diseño.

comportamiento de su correspondiente realización de caso de uso-análisis del modelo de análisis, así como el comportamiento de su correspondiente caso de uso del modelo de casos de uso, y sólo esos comportamientos.

Esto incluye hacer legibles y adecuadas para sus propósitos todas las descripciones textuales y todos los diagramas que describen la realización del caso de uso.

Obsérvese que el ingeniero de casos de uso no es responsable de las clases, subsistemas, interfaces y relaciones de diseño que se utilizan en la realización del caso de uso. Éstas son responsabilidad del correspondiente ingeniero de componentes.

9.4.3. Trabajador: ingeniero de componentes

El ingeniero de componentes define y mantiene las operaciones, métodos, atributos, relaciones y requisitos de implementación de una o más clases del diseño, garantizando que cada clase del diseño cumple los requisitos que se esperan de ella según las realizaciones de caso de uso en las que participa (*véase* la Figura 9.15).

El ingeniero de componentes puede mantener también la integridad de uno o más subsistemas. Esto incluye garantizar que sus contenidos (clases y sus relaciones) son correctos, que las dependencias de otros subsistemas y/o interfaces son correctas y mínimas, y que realizan correctamente las interfaces que ofrecen.

Suele ser adecuado hacer que el ingeniero de componentes responsable de un subsistema sea también responsable de los elementos del modelo que éste último contiene. Además, para conseguir un desarrollo uniforme y sin discontinuidades, lo natural es que los artefactos del modelo de diseño (las clases y subsistemas del diseño) se conserven en el flujo de trabajo de implementación y que los implemente el mismo ingeniero de componentes.

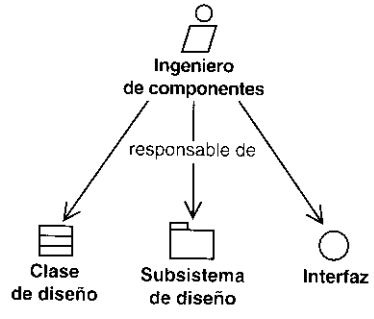


Figura 9.15. Las responsabilidades del ingeniero de componentes en el diseño.

9.5. Flujo de trabajo

Hasta ahora en este capítulo, hemos descrito el trabajo de diseño en términos estáticos. A continuación, vamos a utilizar un diagrama de actividad para razonar sobre su comportamiento dinámico; *véase* la Figura 9.16.

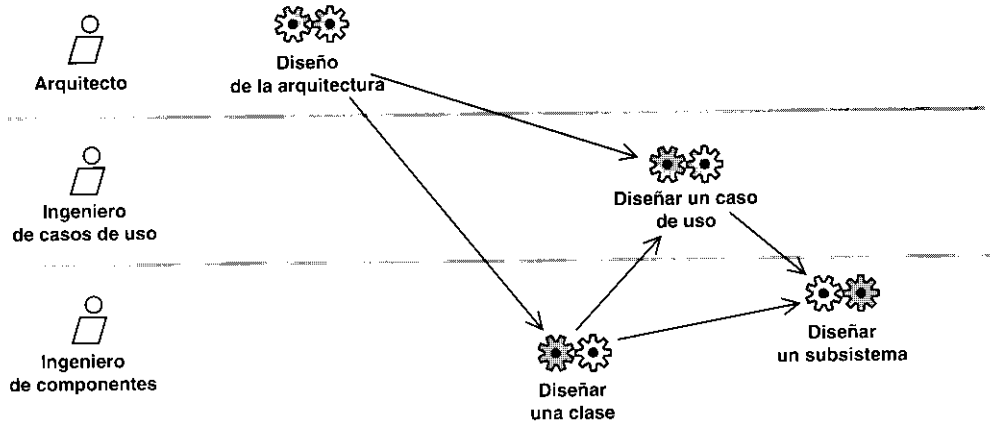


Figura 9.16. El flujo de trabajo en el diseño, incluyendo los trabajadores participantes y sus actividades.

Los arquitectos inician la creación de los modelos de diseño y de despliegue (tal y como se han definido anteriormente en este capítulo). Ellos esbozan los nodos del modelo de despliegue, los subsistemas principales y sus interfaces, las clases del diseño importantes como las activas, y los mecanismos genéricos de diseño del modelo de diseño. Después, los ingenieros de casos de uso realizan cada caso de uso en términos de clases y/o subsistemas del diseño participantes y sus interfaces. Las realizaciones de caso de uso resultantes establecen los requisitos de comportamiento para cada clase o subsistema que participe en alguna realización de caso de uso. Los ingenieros de componentes especifican a continuación los requisitos, y los integran dentro de cada clase, bien mediante la creación de operaciones, atributos y relaciones consistentes sobre cada clase, o bien mediante la creación de operaciones consistentes en cada interfaz que proporcione el subsistema. A lo largo del flujo de trabajo del diseño, los desarrolladores identificarán, a medida que evolucione el diseño, nuevos candidatos para ser subsistemas, interfaces, clases y mecanismos de diseño genéricos, y los ingenieros de componentes responsables de los subsistemas individuales deberán refinarlos y mantenerlos.

9.5.1. Actividad: diseño de la arquitectura

El objetivo del diseño de la arquitectura es esbozar los modelos de diseño y despliegue y su arquitectura mediante la identificación de los siguientes elementos (*véase* la Figura 9.17):

- Nodos y sus configuraciones de red.
- Subsistemas y sus interfaces.
- Clases del diseño significativas para la arquitectura, como las clases activas.
- Mecanismos de diseño genéricos que tratan requisitos comunes, como los requisitos especiales sobre persistencia, distribución, rendimiento y demás, tal y como se capturaron durante el análisis sobre las clases y las realizaciones de caso de uso-análisis.

A lo largo de esta actividad los arquitectos consideran distintas posibilidades de reutilización, como la reutilización de partes de sistemas parecidos, o de productos software generales. Los subsistemas, interfaces u otros elementos del diseño resultantes se añadirán posteriormente

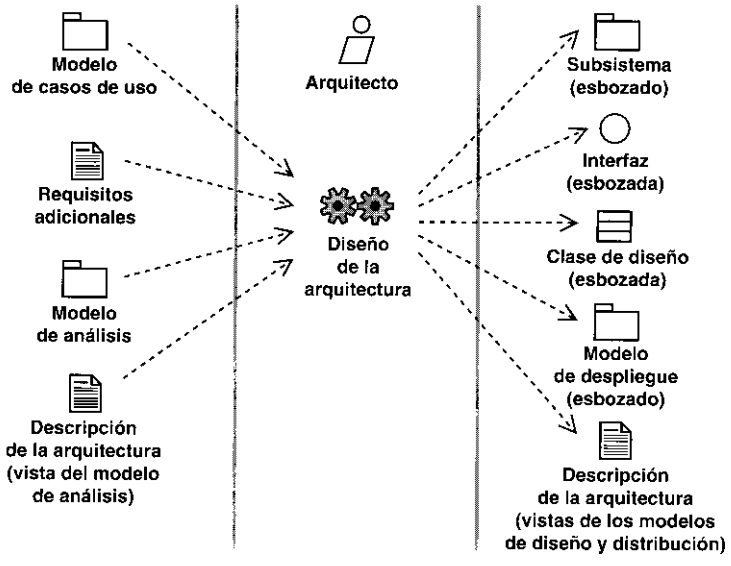


Figura 9.17. Las entradas y los resultados del diseño de la arquitectura.

al modelo de diseño. El arquitecto también mantiene, refina y actualiza la descripción de la arquitectura y sus vistas arquitectónicas de los modelos de diseño y despliegue.

9.5.1.1. Identificación de nodos y configuraciones de red

Las configuraciones físicas de red suelen tener una gran influencia sobre la arquitectura del software, incluyendo las clases activas que se necesitan y la distribución de la funcionalidad entre los nodos de la red. Las configuraciones de red habituales utilizan un patrón de tres capas en el cual los clientes (las interacciones de los usuarios) se dejan en una capa, la funcionalidad de base de datos en otra, y la lógica del negocio o de la aplicación en una tercera. El patrón cliente/servidor simple es un caso especial de este patrón de tres capas en el cual la lógica del negocio o de la aplicación se ubica en una de las otras capas (en la capa del cliente o en la de base de datos).

Entre los aspectos de configuraciones de red podemos resaltar:

- ¿Qué nodos se necesitan, y cuál debe ser su capacidad en términos de potencia de proceso y tamaño de memoria?
- ¿Qué tipo de conexiones debe haber entre los nodos, y qué protocolos de comunicaciones se deben utilizar sobre ellas?
- ¿Qué características deben tener las conexiones y los protocolos de comunicaciones, en aspectos tales como ancho de banda, disponibilidad y calidad?
- ¿Es necesario tener alguna capacidad de proceso redundante, modos de fallo, migración de procesos, mantenimiento de copias de seguridad de los datos, o aspectos similares?

Gracias al conocimiento de los límites y las posibilidades de los nodos y de sus conexiones, el arquitecto puede incorporar tecnologías como object request brokers y servicios de replicación de datos, que pueden hacer más fácil la realización de la distribución del sistema.

Ejemplo Configuración de red para el sistema Interbank

El sistema Interbank se ejecutará sobre tres nodos servidores y un cierto número de nodos cliente. En primer lugar, tenemos un nodo servidor para el comprador y uno para el vendedor, debido a que cada una de las organizaciones compradoras o vendedoras requiere un servidor central para sus objetos de negocio y su procesamiento. Los usuarios finales, como el Comprador, acceden al sistema mediante nodos cliente. Estos nodos se comunican mediante el protocolo TCP/IP de Internet e intranet; *véase* la Figura 9.18.

Tenemos también un tercer nodo servidor para el propio banco. En él se producen los verdaderos pagos de facturas (es decir, las transferencias entre cuentas).

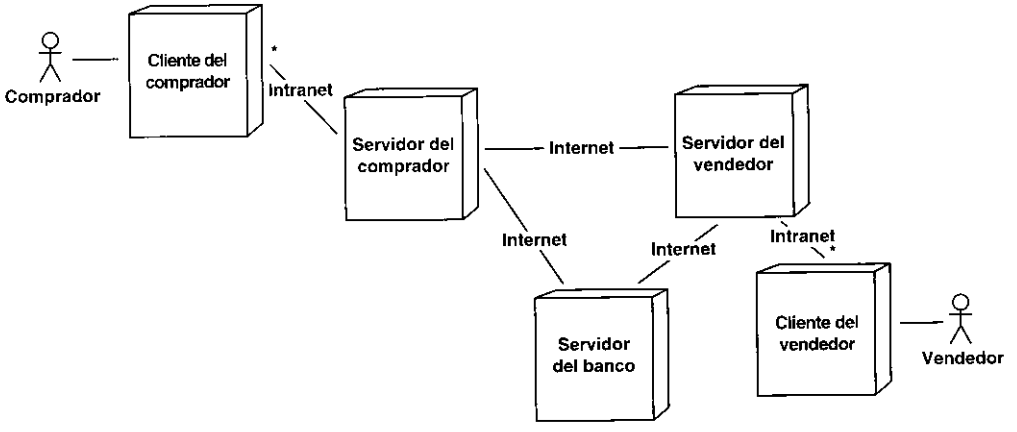


Figura 9.18. Diagrama de despliegue para el sistema Interbank.

Cada configuración de red, incluidas las configuraciones para pruebas y simulación, debería mostrarse en un diagrama de despliegue separado. Dadas esas configuraciones de red, podemos comenzar a analizar cómo puede distribuirse la funcionalidad entre ellas (*véase* la Sección 9.5.1.3.2).

9.5.1.2. Identificación de subsistemas y de sus interfaces

Los subsistemas constituyen un medio para organizar el modelo de diseño en piezas manejables. Pueden bien identificarse inicialmente como forma de dividir el trabajo de diseño, o bien pueden irse encontrando a medida que el modelo de diseño evoluciona y va "creciendo" hasta convertirse en una gran estructura que debe ser descompuesta.

Obsérvese también que no todos los subsistemas se desarrollan internamente en el proyecto en curso. En realidad, algunos subsistemas representan productos reutilizados y otros son recursos existentes en la empresa. La inclusión de subsistemas de esos tipos en el modelo de diseño permite analizar y evaluar las alternativas de reutilización.

9.5.1.2.1. Identificación de subsistemas de aplicación

En este paso identificamos los subsistemas de las capas específica de la aplicación y general de la aplicación (es decir, los subsistemas en las dos capas superiores). *Véase* la Figura 9.19.

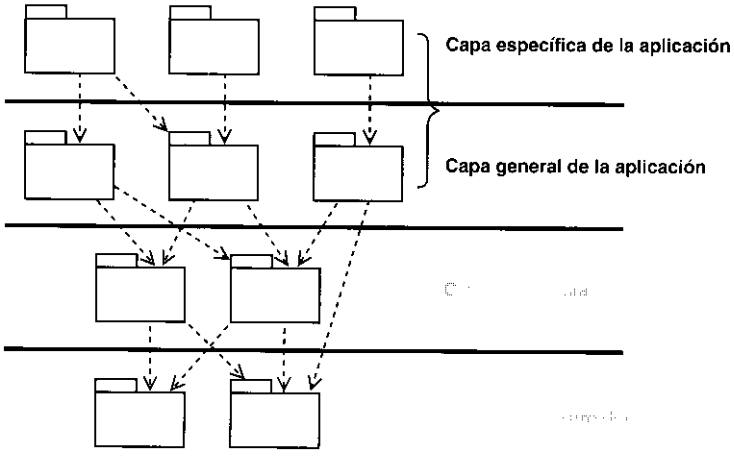


Figura 9.19. Subsistemas en las capas específica de la aplicación y general de la aplicación.

Si se hizo durante el análisis una descomposición adecuada en paquetes del análisis, podemos utilizar esos paquetes tanto como sea posible e identificar los correspondientes subsistemas dentro del modelo de diseño. Esto es especialmente importante en el caso de los paquetes de servicio, de forma que podamos identificar subsistemas de servicio correspondientes que no rompan con la estructuración del sistema de acuerdo a los servicios que ofrece. Sin embargo, podemos refinar ligeramente esta identificación de subsistemas inicial durante el diseño para tratar temas relativos al diseño, la implementación y la distribución del sistema, como vamos a exponer a continuación.

Ejemplo Identificación de subsistemas del diseño a partir de paquetes del análisis

Los paquetes Gestión de Facturas de Comprador y Gestión de Cuentas del modelo de análisis se utilizan para identificar los correspondientes subsistemas del modelo de diseño (*véase* la Figura 9.20).

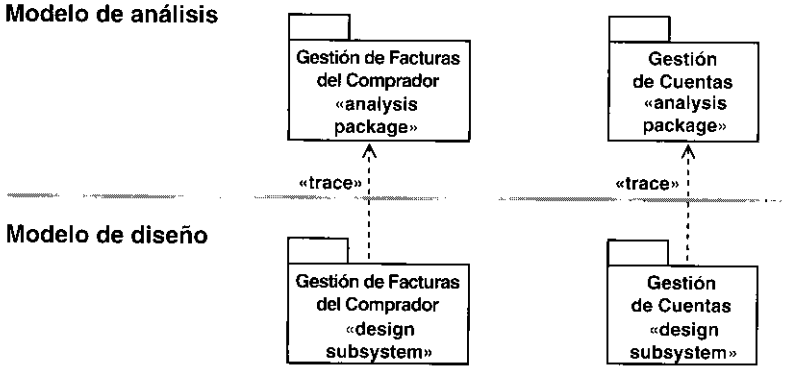


Figura 9.20. Identificación de subsistemas a partir de paquetes del análisis existentes.

Además, los paquetes de servicio Cuentas y Riesgos dentro del paquete de Gestión de Cuentas del modelo de análisis se utilizan para identificar los correspondientes subsistemas de servicio del modelo de diseño (*véase* la Figura 9.21).

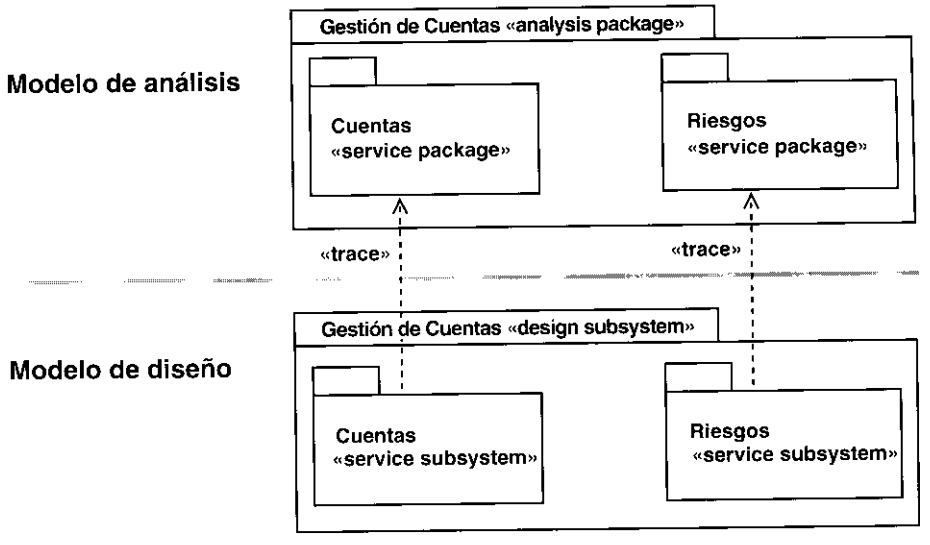


Figura 9.21. Identificación de subsistemas de servicio a partir de paquetes de servicio existentes.

Puede ser necesario un refinamiento de esta descomposición inicial en subsistemas, obtenida de los paquetes del análisis del modelo de análisis, en los siguientes casos:

- Una parte de un (anterior) paquete del análisis se corresponde con un subsistema por sí misma (es decir, si encontramos que esa parte puede ser compartida y utilizada por varios otros subsistemas).

Ejemplo Refinamiento de los subsistemas para tratar funcionalidades compartidas

Interbank Software consideró el implementar todos los paquetes de servicio para el pago de facturas en el subsistema Gestión de Facturas de Comprador, que es donde parecían encajar tras el análisis de los casos de uso relacionados con la gestión de facturas. En ese momento, los desarrolladores se dieron cuenta de que varios casos de uso futuros podrían beneficiarse de la existencia de un subsistema de servicio de pagos general. Por tanto, decidieron agrupar la funcionalidad de los pagos en un subsistema de servicio independiente, llamado Gestión de Planificación de Pagos. Esto implica que el subsistema Gestión de Facturas de Comprador utiliza la funcionalidad para planificar facturas de este nuevo subsistema de servicio. Más adelante, cuando el pago planificado debe hacerse efectivo, el subsistema Gestión de Planificación de Pagos utiliza el subsistema Gestión de Cuentas para la transferencia real de dinero de una cuanta a otra. *Véase* la Figura 9.22.

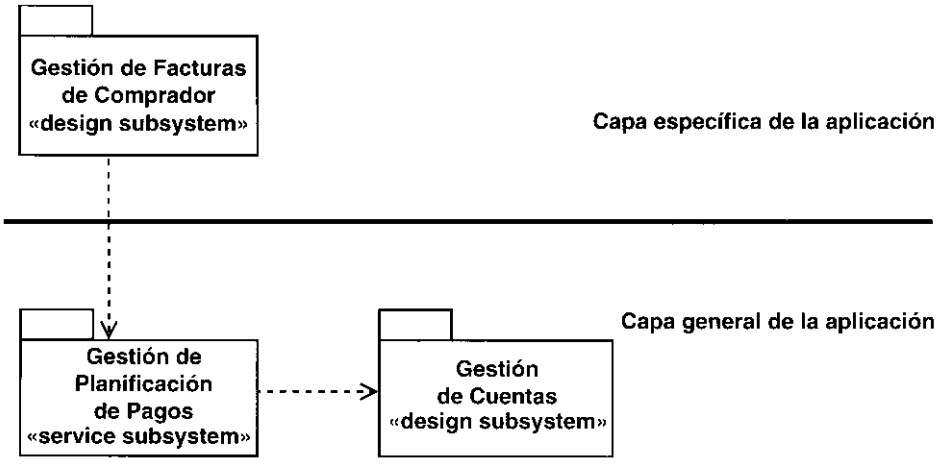


Figura 9.22. Durante el diseño, se identificó el subsistema de servicio Gestión de Planificación de Pagos para proporcionar un servicio general que puedan utilizar diferentes realizaciones de casos de uso.

- Algunas partes de un (anterior) paquete del análisis se realizan mediante productos software reutilizados. Estas funcionalidades podrían asignarse a capas intermedias o subsistemas de software del sistema (*véase* la Sección 9.5.1.2.2).
- Los (anteriores) paquetes del análisis no representan una división del trabajo adecuada.
- Los (anteriores) paquetes del análisis no representan la incorporación de un sistema heredado. Podemos encapsular un sistema heredado, o parte de él, mediante un subsistema de diseño independiente.
- Los (anteriores) paquetes del análisis no están preparados para una distribución directa sobre los nodos. Puede que la descomposición de subsistemas tenga que tratar en este caso los aspectos de distribución de ese tipo descomponiendo adicionalmente algunos subsistemas en subsistemas más pequeños, de forma que cada uno de ellos pueda asignarse a un nodo determinado. Después, debemos refinar esos subsistemas más pequeños para minimizar el tráfico de la red, y así sucesivamente.

Ejemplo Distribución de un subsistema entre los nodos

El subsistema Gestión de Facturas de Comprador debe distribuirse en varios nodos diferentes para su despliegue. Para poder hacerlo, descomponemos a su vez el subsistema en tres subsistemas: IU del Comprador, Gestión de Solicitudes de Pago y Gestión de Facturas (*véase* la Figura 9.23). Los componentes generados a partir de cada uno de esos tres subsistemas más pequeños pueden después distribuirse respectivamente en los nodos Cliente del Comprador, Servidor del Comprador y Servidor del Vendedor.

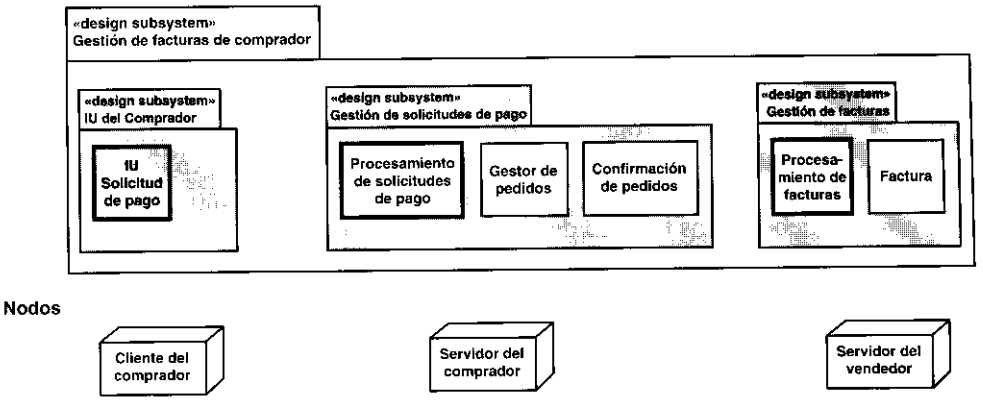


Figura 9.23. Un subsistema descompuesto recursivamente en tres nuevos subsistemas para tratar su distribución.

9.5.1.2.2. Identificación de subsistemas intermedios y de software del sistema

El software del sistema y la capa intermedia constituyen los cimientos de un sistema, ya que toda la funcionalidad descansa sobre software como sistemas operativos, sistemas de gestión de base de datos, software de comunicaciones, tecnologías de distribución de objetos, kits de diseño de IGU, y tecnologías de gestión de transacciones [3] (*véase* la Figura 9.24). La selección e integración de productos software que se compran o se construyen son dos de los objetivos fundamentales durante las fases de inicio y elaboración. El arquitecto verifica que los productos software elegidos encajan en la arquitectura y permiten una implementación económica del sistema.

Un aviso de precaución: cuando compramos middleware y software del sistema, tenemos un control limitado o nulo sobre su evolución. Por tanto es importante mantener una adecuada libertad de acción y evitar hacernos totalmente dependientes de un determinado producto o fa-

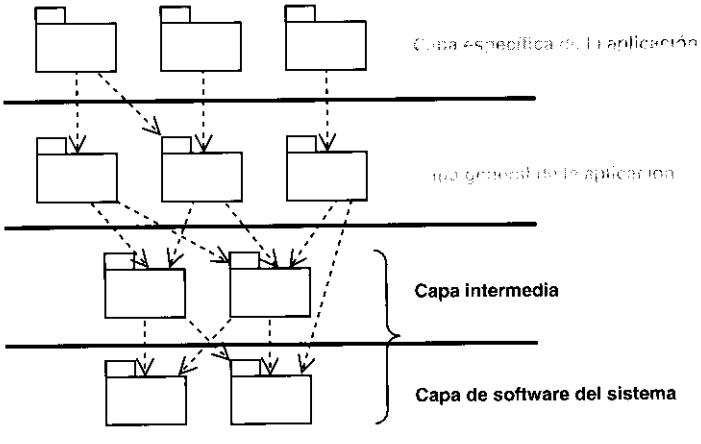


Figura 9.24. Los subsistemas de las capas intermedia y de software del sistema encapsulan productos software comprados.

bricante sobre el cual nuestro proyecto tiene poca influencia. Debemos intentar limitar bien las dependencias de productos comprados, limitando así el riesgo asociado con su uso, en caso de que cambien en el futuro, o bien ser capaces de cambiar de fabricante si es necesario.

Una forma de controlar las dependencias es considerar cada producto software comprado como si fuese un subsistema independiente con interfaces explícitos para el resto de los sistemas. Por ejemplo, si tenemos que implementar un mecanismo para distribución transparente de objetos, deberíamos hacerlo definiendo un interfaz estricto con un subsistema que se encargará del mecanismo. Esto mantiene la libertad de elegir entre diferentes productos, ya que queda delimitado el coste de actualizar el sistema.

Ejemplo Utilización de Java para construir la capa intermedia

Varias de las implementaciones de subsistemas de aplicación que desarrolla Interbank Software pueden tener que ejecutarse sobre diferentes tipos de máquinas, como computadores personales y estaciones de trabajo UNIX, y por tanto deben ser capaces de interoperar entre diferentes plataformas. Interbank Software decidió implementar esta interoperación mediante middleware, en este caso, mediante los paquetes Java AWT (*Abstract Windowing Toolkit*), RMI (*Remote Method Invocation*) y Applet. Estos paquetes de Java se representan como subsistemas en la Figura 9.25. Estos subsistemas se ejecutan sobre la Máquina Virtual Java, que es el intérprete de Java que debe instalarse para que una máquina sea capaz de ejecutar código Java. Utilizamos en nuestros ejemplos un navegador de Internet para cargar páginas Web que incorporan applets.

A bajo nivel, Interbank Software se construirá sobre software del sistema, como el protocolo TCP/IP para la comunicación en Internet (*véase* la Figura 9.25).

El subsistema Java Applet es un middleware que permite a Interbank la creación de *applets* de Java.

Cada ventana del interfaz de usuario se diseña mediante la clase Java Applet junto con otras clases de interfaz de usuario como List, que proporciona el subsistema AWT. El paquete Java Abstract Windowing Toolkit (java.awt) se construyó para permitir la creación de interfaces de usuario independientes de la plataforma. El paquete AWT incluye clases como Field, Scrollbar y CheckBox.

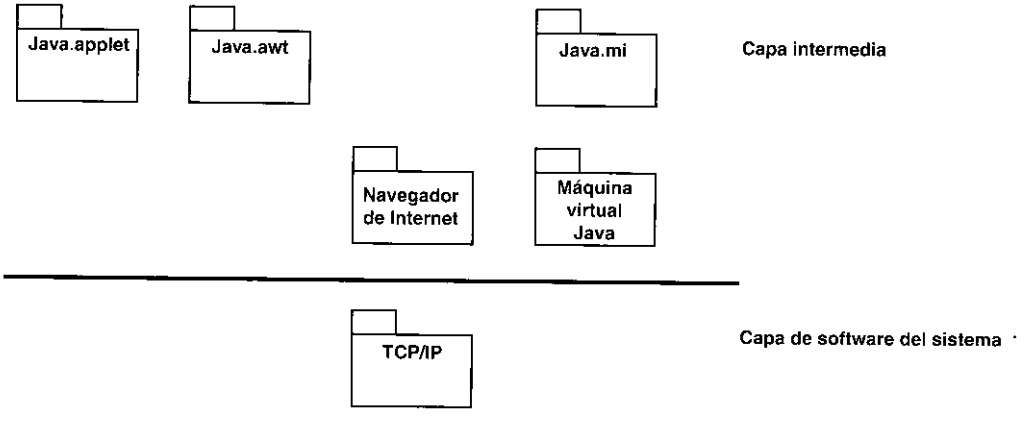


Figura 9.25. La capa intermedia de Java proporciona interfaces gráficos de usuario independientes de la plataforma (java.awt) y distribución de objetos (java.rmi). Esta figura no muestra ninguna dependencia entre subsistemas (las presentaremos en la Figura 9.26).

Java RMI es un mecanismo para la distribución de objetos integrado en las bibliotecas de clases de Java. Hemos elegido aquí RMI para la distribución de objetos para hacer más sencillos los ejemplos y para evitar mezclar diferentes técnicas y lenguajes. Podríamos haber utilizado también una solución CORBA o ActiveX/DCOM.

La Figura 9.25 ilustra cómo pueden organizarse en subsistemas los servicios Java en la capa de middleware. Cada subsistema contiene clases diseñadas para proporcionar ciertos servicios cuando se utilizan juntas. De igual manera, los componentes ActiveX, que comprenden hojas de cálculo, multimedia, procesamiento de imágenes, gestión de seguridad, persistencia, distribución, motores de inferencia, procesos y soporte de concurrencia, pueden representarse como subsistemas. También suele ser práctico el crear un subsistema independiente para los tipos de datos estándar o las clases fundamentales que otros subsistemas pueden importar y utilizar (por ejemplo, java.lang, que proporciona muchas de las clases fundamentales de Java, como Boolean, Integer y Float).

9.5.1.2.3. Definición de dependencias entre subsistemas Deberían definirse dependencias entre subsistemas si sus contenidos tienen relaciones unos con otros. La dirección de la dependencia debería ser la misma que la dirección (navegabilidad) de la relación. Si tenemos que esbozar las dependencias antes de saber los contenidos de los subsistemas, consideraremos las dependencias entre los paquetes del análisis que se corresponden con los subsistemas del diseño. Estas dependencias serán probablemente parecidas en el modelo de diseño. Además, si utilizamos interfaces entre subsistemas, las dependencias deberían ir hacia las interfaces, no hacia los propios subsistemas (*véase* la Sección 9.5.1.2.4).

Ejemplo Dependencias y capas

La Figura 9.26 muestra los subsistemas y algunas de las dependencias iniciales del sistema Interbank.

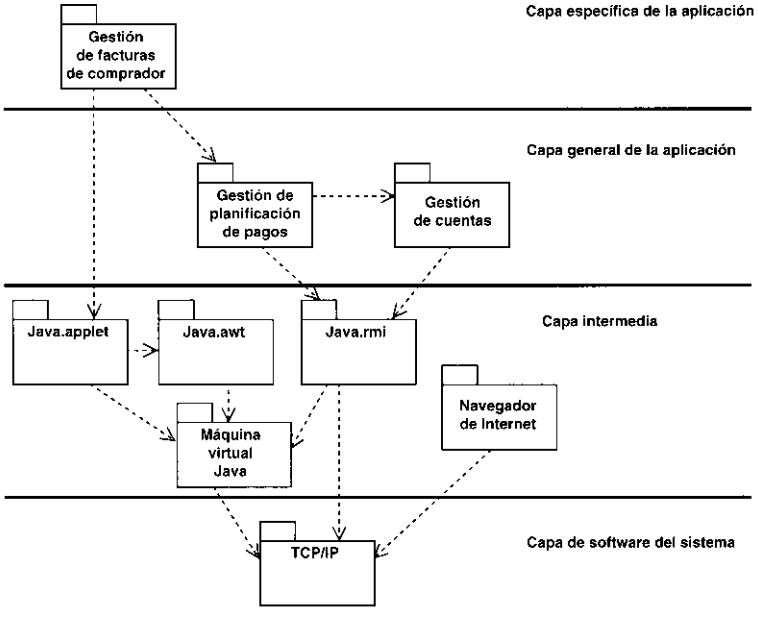


Figura 9.26. Dependencias y capas de algunos de los subsistemas del sistema Interbank.

Obsérvese que algunas de esas dependencias, especialmente las de las capas intermedias y de software del sistema, pueden ser más o menos implícitas en el entorno de implementación (Java en este caso). Pero cuando las dependencias tienen impacto en la arquitectura del sistema, especielamente cuando "atraviesan" capas, es útil hacerlas explícitas en el modelo de diseño y mostrarlas en diagramas de clases como el de la Figura 9.26. Así, los ingenieros de componentes y de casos de uso pueden hacer referencia a esos diagramas al diseñar casos de uso, clases y subsistemas.

9.5.1.2.4. Identificación de interfaces entre subsistemas Las interfaces proporcionadas por un subsistema definen operaciones que son accesibles "desde fuera" del subsistema. Estas interfaces las proporcionan o bien clases o bien otros subsistemas (recursivamente) dentro del subsistema.

Para esbozar inicialmente las interfaces, antes de que se conozcan los contenidos de los subsistemas, comenzaremos considerando las dependencias entre subsistemas que se identificaron en el paso anterior (Sección 9.5.1.2.3). Cuando un subsistema tiene una dependencia que apunta hacia él, es probable que deba proporcionar una interfaz. Además, si existe un paquete del análisis que podamos obtener mediante una traza desde el subsistema, entonces cualquier clase del análisis referenciada desde el exterior del paquete puede implicar una interfaz candidata para el subsistema (como se muestra en el siguiente ejemplo).

Ejemplo **Identificación de interfaces candidatas a partir del modelo de análisis**

El paquete de servicio Cuentas contiene una clase del análisis llamada Transferencias entre Cuentas a la cual se hace referencia desde fuera del paquete. Podemos por tanto identificar una interfaz inicial, llamada Transferencias, proporcionada por el correspondiente subsistema de servicio Cuentas del modelo de diseño (*véase* la Figura 9.27).

Mediante esta técnica, identificamos inicialmente las interfaces que se muestran en la Figura 9.28 en las dos capas superiores del modelo de diseño.

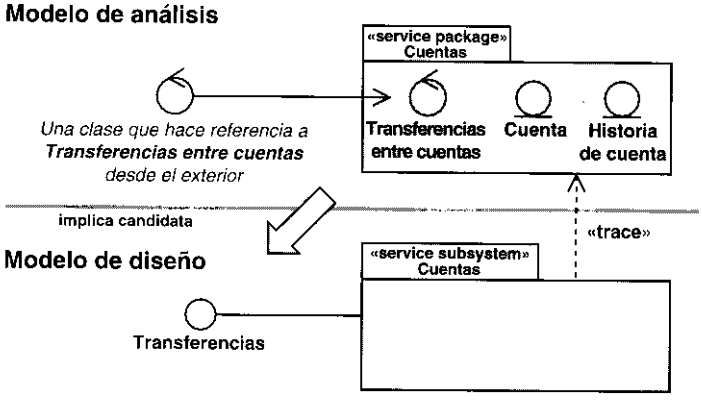


Figura 9.27. Una interfaz identificada inicialmente a partir del modelo de análisis.

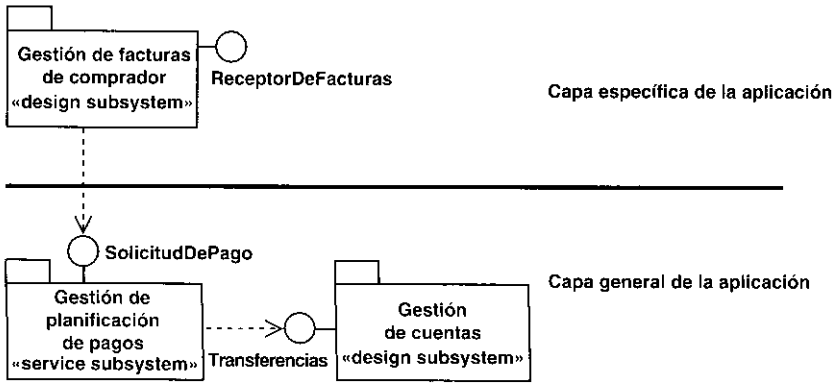


Figura 9.28. Las interfaces en las dos capas superiores del modelo de diseño.

El subsistema Gestión de Cuentas proporciona la interfaz Transferencias para tranferir dinero entre cuentas. Ésta es la misma interfaz que proporciona el subsistema de servicio Cuentas dentro de Gestión de Cuentas (*véase* el ejemplo anterior). El subsistema Gestión de Planificación de Pagos proporciona la interfaz SolicitudDePago que se utiliza para planificar los pagos. El subsistema Gestión de Facturas de Comprador proporciona la interfaz ReceptorDeFacturas para recibir nuevas facturas de un vendedor. Esta interfaz se utiliza en el caso de uso Enviar Factura al Comprador, en el cual se manda una nueva factura al comprador.

Obsérvese que las interfaces que hemos identificado aquí nos permiten refinar las dependencias entre los subsistemas, como se trató en la Sección 9.5.1.2.3, cambiando las dependencias para que sean entre interfaces y no entre subsistemas.

Respecto a las interfaces de las dos capas inferiores (las capas intermedias y de software del sistema), el problema de identificar interfaces es más sencillo, ya que los subsistemas de estas capas encapsulan productos software, y esos productos suelen tener algún tipo de interfaces predefinidas.

Sin embargo, no es suficiente con identificar las interfaces; también debemos identificar las operaciones que deben definirse sobre cada una de ellas. Esto se hace mediante el diseño de los casos de uso en términos de subsistemas y de sus interfaces, como se describe en las Secciones 9.5.2.3 y 9.5.2.4, en la siguiente actividad de diseño de casos de uso. Así se establecerán los requisitos sobre las operaciones que deban definir las interfaces. Los requisitos de las diversas realizaciones de caso de uso se tratan y se integran de esta forma para cada interfaz, como se describe en la Sección 9.5.4.2.

9.5.1.3. Identificación de clases del diseño relevantes para la arquitectura

Suele ser práctico identificar las clases del diseño relevantes para la arquitectura pronto dentro del ciclo de vida del software, para comenzar el trabajo de diseño. Sin embargo, la mayoría de las clases del diseño se identificarán al diseñar las clases dentro de la actividad de diseño de clases, y se refinarán de acuerdo a los resultados obtenidos en la actividad de diseño de casos de uso (*véase* las Secciones 9.5.2 y 9.5.3). Por este motivo, los desarrolladores deberían evitar el identificar demasiadas clases en esta etapa o el quedar atrapados en demasiados detalles. Sería suficiente con un esbozo inicial de las clases significativas para la arquitectura (*véase* la Sección 9.3.6). De otro modo, probablemente habrá que retocar gran

parte del trabajo cuando más adelante se utilicen los casos de uso para justificar las clases del diseño (las clases que participan en las realizaciones de caso de uso). Una clase del diseño que no participa en ninguna realización de caso de uso es innecesaria.

9.5.1.3.1. Identificación de clases del diseño a partir de clases del análisis

Podemos esbozar inicialmente algunas clases del diseño a partir de las clases del análisis significativas para la arquitectura que encontramos en el análisis. Además, podemos utilizar las relaciones entre esas clases del análisis para identificar un conjunto tentativo de relaciones entre las correspondientes clases del diseño.

Ejemplo **Esbozo de una clase de diseño a partir de una clase del análisis**

La clase del diseño Factura se esboza inicialmente a partir de la clase de entidad Factura del modelo de análisis (*véase* la Figura 9.29).

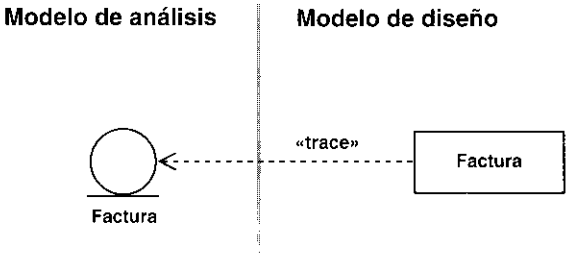


Figura 9.29. La clase del diseño Factura se esboza inicialmente a partir de la clase de entidad Factura.

9.5.1.3.2. Identificación de clases activas

El arquitecto también identifica las clases activas necesarias en el sistema considerando los requisitos de concurrencia del mismo. Por ejemplo:

- Los requisitos de rendimiento, tiempo de respuesta y disponibilidad que tienen los diferentes actores en su interacción con el sistema. Si, por ejemplo, un determinado actor tiene requisitos exigentes en cuanto a tiempo de respuesta, entonces quizá podríamos dedicar un objeto activo a la gestión de ese actor, que tome sus entradas y le envíe las respuestas —un objeto que no se pare debido a que otros objetos tengan una carga alta (suponiendo que tenemos suficientes recursos de capacidad de procesador y de memoria).
- La distribución del sistema sobre los nodos. Los objetos activos deben soportar la distribución sobre varios nodos diferentes, que pueden requerir, por ejemplo, al menos un objeto activo por nodo y objetos activos aparte para gestionar la intercomunicación entre los nodos.
- Otros requisitos como requisitos sobre el arranque y terminación del sistema, progresión, evitación del interbloqueo, evitación de la inanición, reconfiguración de nodos y la capacidad de los nodos.

Cada clase activa se esboza mediante la consideración del ciclo de vida de sus objetos activos y de cómo deberían comunicarse, sincronizarse y compartir información los objetos activos. Después, se asignan los objetos activos a los nodos del modelo de despliegue. Al hacer esto último, es necesario considerar la capacidad de los nodos, sus procesadores y sus tamaños de memoria, y las características de las conexiones, como el ancho de banda y la disponibilidad. Debemos tener en cuenta la regla básica de que el tráfico de la red suele tener un impacto importante en los recursos de cómputo (incluyendo tanto al hardware como al software) que el sistema necesita, y por tanto debe mantenerse bajo un control estricto. Esto puede condicionar de manera muy significativa el modelo de diseño.

Para esbozar las clases activas inicialmente podemos utilizar los resultados del análisis y el modelo de despliegue como entradas y después hacer corresponder los diseños de las respectivas clases del análisis (o de partes de ellas) con los nodos, mediante el uso de clases activas.

Ejemplo Utilización de clases del análisis para esbozar clases activas

Como ya hemos dicho, el sistema Interbank debe ser distribuido en nodos: cliente del Comprador, servidor del Comprador, servidor del Banco y demás. Por otro lado, hemos identificado clases del análisis tales como IU Solicitud de Pago, Gestor de Pedidos, Confirmación de Pedidos, Factura, etc. (*véase* la Figura 9.30). Ahora, nos encontramos con un comprador que está interesado en la funcionalidad que proporcionan las clases del análisis Confirmación de Pedido y Gestor de Pedidos, pero esta funcionalidad requiere más potencia de cálculo de la que tiene el nodo cliente del Comprador. Por tanto, las partes principales de esas dos clases del análisis deben ubicarse en el Servidor del Comprador, y deben ser gestionadas por una clase activa separada (Procesamiento de Solicitudes de Pago) en ese nodo.

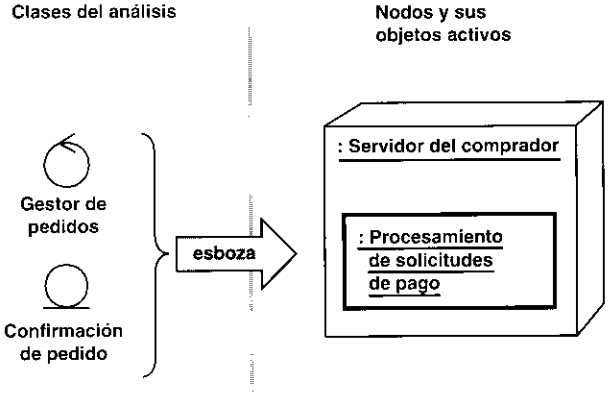


Figura 9.30. Una clase activa, procesamiento de solicitudes de pago, obtenida mediante el estudio de las clases del análisis Gestor de Pedidos y Confirmación de Pedido y del nodo Servidor del Comprador. Se decidió que las principales partes de las clases del diseño correspondientes se ubicarían en el nodo servidor del Comprador. Después se identificó la clase Procesamiento de Solicitudes de Pago para encapsular el hilo de control de esas clases del diseño dentro del nodo.

Otra posibilidad para esbozar clases activas es utilizar los subsistemas que hemos obtenido anteriormente y asignar esos subsistemas a nodos particulares, mediante la identificación de una clase activa dentro del subsistema (*véase* la Sección 9.5.1.2.1). Debemos recordar que puede que tengamos que refinar la descomposición en subsistemas para hacer esto.

Cualquier clase activa que represente un proceso pesado es candidata para la identificación de un componente ejecutable durante la implementación. Por tanto, la asignación de clases activas a los nodos en este paso es una entrada importante para la asignación de componentes (ejecutables) a los nodos durante la implementación. Por otro lado, cuando asignamos un subsistema entero a un nodo, todos los constituyentes del subsistema suelen conformar juntos un componente (ejecutable) en ese nodo.

9.5.1.4. Identificación de mecanismos genéricos de diseño

En este paso estudiamos requisitos comunes, como los requisitos especiales que se identificaron durante el análisis en las realizaciones de caso de uso-análisis y en las clases del análisis, y decidimos cómo tratarlos, teniendo en cuenta las tecnologías de diseño e implementación disponibles. El resultado es un conjunto de mecanismos genéricos de diseño que pueden manifestarse como clases, colaboraciones o incluso subsistemas, de forma parecida a lo descrito en [1].

Los requisitos que deben tratarse suelen estar relacionados con aspectos como:

- Persistencia.
- Distribución transparente de objetos.
- Características de seguridad.
- Detección y recuperación de errores.
- Gestión de transacciones.

En algunos casos, el mecanismo no puede identificarse *a priori*, y se descubre en cambio a medida que se exploran las realizaciones de caso de uso y las clases del diseño.

Ejemplo **Un mecanismo de diseño para la distribución transparente de objetos**

Algunos objetos, como los objetos Factura, deben ser accesibles desde varios nodos, por lo que deben diseñarse para un sistema distribuido. Interbank Software decide implementar esa distribución de objetos haciendo que cada clase distribuida sea subclase de la clase abstracta de Java java.rmi.UnicastRemoteObject, que soporta RMI; *véase* la Figura 9.31. RMI es la técnica que Java utiliza para obtener una distribución transparente de los objetos (es decir, objetos distribuidos de manera que el cliente no tiene conocimiento de dónde reside el objeto).

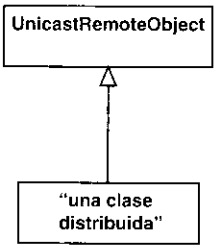


Figura 9.31. Cualquier clase que deba ser distribuida debería ser un subtipo de UnicastRemoteObject.

Ejemplo Mecanismos de diseño para la persistencia

Algunos objetos, como los objetos Pedido y Factura, deben ser persistentes. Para conseguirlo, Interbank Software puede utilizar un sistema de gestión de bases de datos orientado a objetos, o bien uno relacional, o incluso ficheros binarios planos. La mejor elección depende de cómo debamos acceder y actualizar los objetos, y de la facilidad de implementación y evolución futura de cada una de las opciones. Una base de datos relacional suele dar un mejor rendimiento para datos tabulares, mientras que una base de datos orientada a objetos lo hace para estructuras de objetos complejas. Los sistemas de gestión de base de datos relacionales son, por otro lado, una tecnología más madura que las bases de datos de objetos, pero la migración a un gestor de objetos de un sistema implementado anteriormente con una base de datos relacional puede ser muy costosa. Sea cual fuere la solución elegida, el arquitecto debe documentarla como un mecanismo de diseño genérico para el tratamiento de los aspectos de persistencia.

El último ejemplo muestra que cada mecanismo suele poder tratarse de varias formas diferentes, cada una de ellas con sus pros y sus contras. Si no es factible utilizar un mismo mecanismo para todas las situaciones, puede que sea necesario proporcionar más de uno y usar el más adecuado en cada situación.

El arquitecto también debería identificar colaboraciones genéricas que puedan servir como patrones utilizados por varias realizaciones de caso de uso dentro del modelo de diseño.

Ejemplo Una colaboración genérica utilizada en varias realizaciones de caso de uso

Mientras trabajan en los casos de uso y sus realizaciones, los arquitectos identifican un patrón en el cual un actor crea un Objeto de Comercio, como por ejemplo un Pedido o una Factura, y lo envía a otro actor:

- Cuando un Comprador decide pedir ciertos bienes o servicios a un vendedor, el comprador invoca el caso de uso Pedir Bienes o Servicios. El caso de uso permite al comprador el especificar y enviar electrónicamente un pedido al vendedor.
- Cuando un Vendedor decide enviar una factura a un comprador, invoca el caso de uso Enviar Factura al Comprador, el cual envía electrónicamente una factura a un comprador.

Éste es un tipo de comportamiento común que puede representarse mediante una colaboración genérica, como se muestra en el diagrama de colaboración de la Figura 9.32.

En primer lugar, IU Creación de Objeto de Comercio recibe información del actor Emisor, que utiliza como entrada para la creación de un Objeto de Comercio (1) (los números entre paréntesis hacen referencia a la Figura 9.32). Después, IU Creación de Objeto de Comercio solicita a Procesamiento del Emisor la creación de un objeto de comercio (2). Procesamiento del Emisor pide a la correspondiente clase Objeto de Comercio que cree una instancia (3). Procesamiento del Emisor envía la referencia del objeto de comercio a Procesamiento del Receptor (4). Procesamiento del Receptor puede consultar en ese momento al objeto de comercio la información que necesite (5). Procesamiento del Receptor envía después la referencia del objeto de comercio a IU Presentación de Objeto de Comercio (6), y éste último muestra el objeto de comercio al actor Receptor (7).

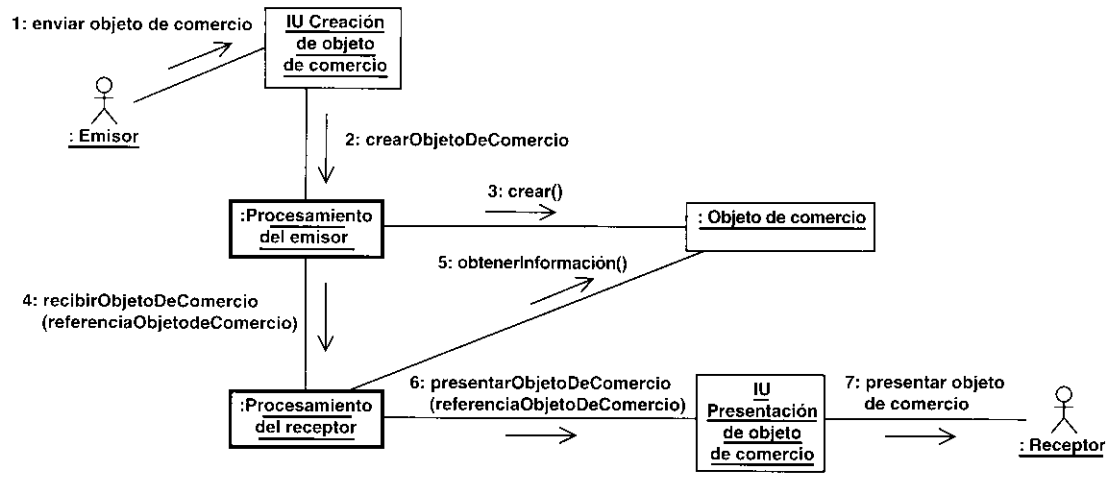


Figura 9.32. Diagrama de colaboración que ejemplifica una colaboración genérica para crear, enviar, recibir y presentar objetos de comercio.

Según este esquema, cuando se lleva a cabo el caso de uso Enviar Factura al Comprador, por ejemplo, podemos hacer que ciertos clasificadores concretos sean subtipos de cada uno de los clasificadores abstractos que participan en la colaboración genérica como se muestra en la Figura 9.33.

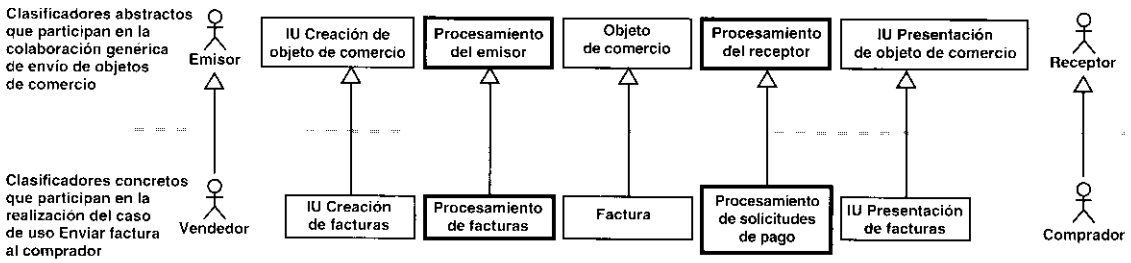


Figura 9.33. Clasificadores concretos participantes en una determinada realización de caso de uso que son subtipos de los clasificadores abstractos que participan en una colaboración genérica.

Según lo expuesto, la realización del caso de uso Enviar Factura al Comprador sólo debe hacer referencia a la colaboración genérica, pero no necesita duplicarla o especializarla, suponiendo que las operaciones (virtuales) de los clasificadores abstractos se realizan mediante métodos de los clasificadores concretos que son sus subtipos.

Podemos utilizar la misma técnica para realizar el caso de uso Pedir Bienes o Servicios.

Obsérvese que el uso de generalizaciones no es la única forma de emplear una colaboración genérica. Por ejemplo, los patrones que son colaboraciones parametrizadas (con clases parametrizadas) también son genéricos y pueden utilizarse asociando clases concretas con los parámetros.

La mayoría de los mecanismos genéricos deberían ser identificados y diseñados en la fase de elaboración. Si se hace así con cuidado, el arquitecto suele ser capaz de diseñar un conjunto de mecanismos que darán solución a los aspectos más difíciles del diseño, haciendo que la mayor parte de los casos de uso sean fácil y directamente realizables en la fase de construcción. Los mecanismos relacionados con productos software comprados son candidatos naturales para la capa intermedia. Los demás mecanismos encontrarán probablemente su sitio natural en la capa general de la aplicación.

9.5.2. Actividad: diseño de un caso de uso

Los objetivos del diseño de un caso de uso son:

- Identificar las clases del diseño y/o los subsistemas cuyas instancias son necesarias para llevar a cabo el flujo de sucesos del caso de uso.
- Distribuir el comportamiento del caso de uso entre los objetos del diseño que interactúan y/o entre los subsistemas participantes.
- Definir los requisitos sobre las operaciones de las clases del diseño y/o sobre los subsistemas y sus interfaces.
- Capturar los requisitos de implementación del caso de uso.

9.5.2.1. Identificación de clases del diseño participantes

En este paso identificaremos las clases del diseño que se necesitan para realizar el caso de uso (*véase* la Figura 9.34). Debemos hacer lo siguiente:

- Estudiar las clases del análisis que participan en la correspondiente realización de caso de uso-análisis. Identificar las clases del diseño que poseen una traza hacia esas clases del

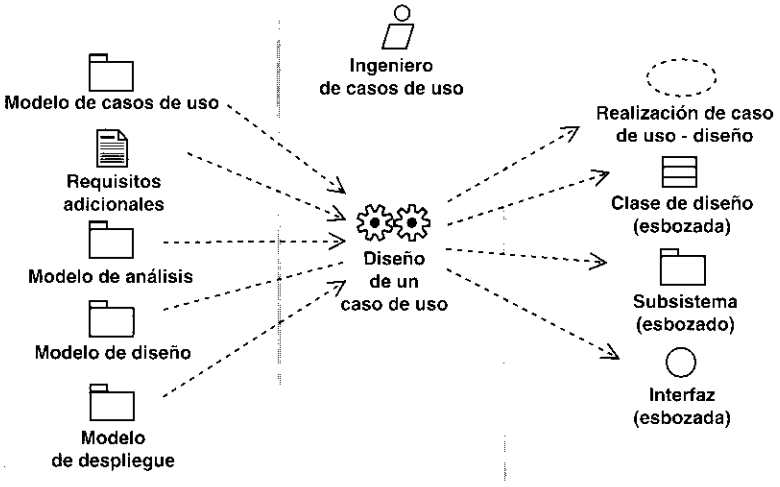


Figura 9.34. Las entradas y los resultados del diseño de un caso de uso. La correspondiente realización de caso de uso-análisis es una entrada fundamental para esta actividad.

análisis, creadas por el ingeniero de componentes en el diseño de clases o por el arquitecto en el diseño arquitectónico.

- Estudiar los requisitos especiales de la correspondiente realización de caso de uso-análisis. Identificar las clases del diseño que realizan esos requisitos especiales. Éstas últimas pueden haber sido identificadas bien por el arquitecto durante el diseño arquitectónico (en la forma de mecanismos genéricos) o bien por el ingeniero de componentes durante el diseño de clases.
- Como resultado, deberíamos identificar las clases necesarias, y deberíamos asignar su responsabilidad a algún ingeniero de componentes.
- Si aún faltase alguna clase del diseño para el caso de uso en particular, el ingeniero de casos de uso debería comunicárselo a los arquitectos o a los ingenieros de componentes. Debería identificarse la clase necesaria, y esta última debería asignarse a un ingeniero de componentes.

Debemos recoger las clases del diseño que participan en una realización de caso de uso en un diagrama de clases asociado con la realización. Utilizaremos este diagrama para mostrar las relaciones que se utilizan en la realización del caso de uso.

Ejemplo Clases participantes en la realización del caso de uso Pagar Factura

La Figura 9.35 muestra un diagrama de clases que contiene las clases que participan en la realización del caso de uso Pagar Factura, y sus asociaciones.

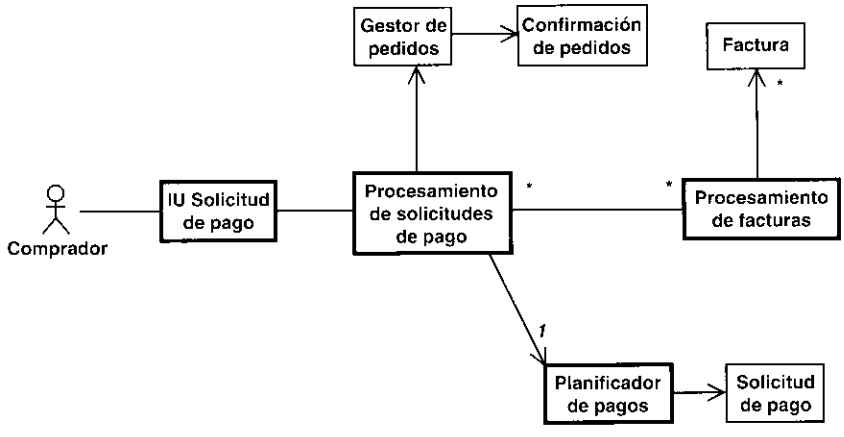


Figura 9.35. Las clases que participan en la realización del caso de uso Pagar Factura y sus asociaciones. En el diagrama, mostramos las clases activas con bordes más gruesos.

Algunas clases activas, como Procesamiento de Solicitudes de Pago y Procesamiento de Facturas, son las que principalmente soportan la ejecución del sistema Interbank. Lo hacen mediante la transferencia de objetos de comercio entre diferentes nodos desde un emisor a un receptor, por ejemplo, la transferencia de una factura del vendedor al comprador.

9.5.2.2. Descripción de las interacciones entre objetos del diseño

Cuando tenemos un esquema de las clases del diseño necesarias para realizar el caso de uso, debemos describir cómo interactúan sus correspondientes objetos del diseño. Esto se hace mediante diagramas de secuencia que contienen las instancias de los actores, los objetos del diseño, y las transmisiones de mensajes entre éstos, que participan en el caso de uso. Si los casos de uso tienen varios flujos o subflujos distintos, suele ser útil en crear un diagrama de secuencia para cada uno de ellos. Esto puede hacer más clara la realización del caso de uso, y también permite extraer diagramas de secuencia que representan interacciones generales y reutilizables.

Para comenzar este paso, debemos estudiar la correspondiente realización de caso de uso-análisis. Podemos utilizarla para obtener un esbozo de la secuencia necesaria de mensajes entre los objetos del diseño, aunque puede que se hayan añadido muchos objetos del diseño nuevos. En algunos casos, incluso puede merecer la pena transformar un diagrama de colaboración de la realización de caso de uso-análisis en un esbozo inicial del correspondiente diagrama de secuencia.

Para crear un diagrama de secuencia, debemos comenzar por el principio del flujo del caso de uso, y después seguir ese flujo paso a paso, decidiendo qué objetos del diseño y qué interacciones de instancias de actores son necesarias para realizar cada paso. En la mayoría de los casos, los objetos se ajustan de manera natural a la secuencia de interacciones de la realización de caso de uso. Debemos observar lo siguiente sobre estos diagramas de secuencia:

- El causante de la invocación del caso de uso es un mensaje de una instancia de un actor hacia un objeto del diseño.
- Cada clase del diseño identificada en el paso anterior debería tener al menos un objeto del diseño participante en el diagrama de secuencia.
- Los mensajes que realizan el caso de uso se envían entre **líneas de vida de los objetos** (Apéndice A). Un mensaje puede tener un nombre temporal que después pasará a ser el nombre de una operación tras haber sido identificado por el ingeniero de componentes responsable de la clase del objeto receptor.
- La secuencia en el diagrama debería ser la principal preocupación, ya que la realización de caso de uso-diseño es la entrada principal para la implementación del caso de uso. Es importante comprender el orden cronológico de las transferencias de mensajes entre objetos.
- Utilizaremos etiquetas y el flujo de sucesos-diseño para complementar los diagramas de secuencia.
- El diagrama de secuencia debería tratar todas las relaciones del caso de uso que realiza. Por ejemplo, si el caso de uso A es una especialización de un caso de uso B mediante una relación de generalización, el diagrama de secuencia que realice el caso de uso A puede tener que hacer referencia a la realización (es decir, al diagrama de secuencia) del caso de uso B. Obsérvese que también es posible que ese tipo de referencias esté presente en las correspondientes realizaciones de caso de uso-análisis.

Ejemplo **Diagrama de secuencia para la primera parte del caso de uso Pagar Factura**

La Figura 9.36 muestra el diagrama de secuencia de la primera parte del caso de uso Pagar Factura.

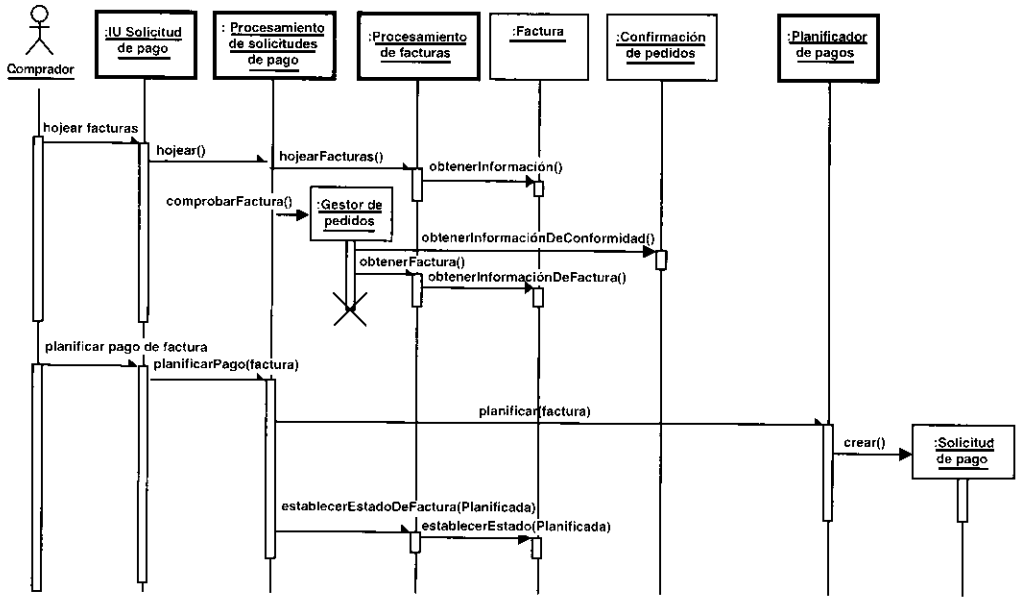


Figura 9.36. Diagrama de secuencia con los objetos de diseño que llevan a cabo parte de la realización del caso de uso Pagar Factura.

La descripción del flujo de sucesos-diseño que complementa a este diagrama de secuencia podría ser algo parecido a lo siguiente:

El comprador utiliza el sistema mediante el applet IU Solicitud de Pago y mediante la aplicación de Procesamiento de Solicitudes de Pago para hojear las solicitudes de pago recibidas. Procesamiento de Solicitudes de Pago utiliza al Gestor de Pedidos para comprobar las facturas con sus confirmaciones de pedido asociadas, antes de que el IU Solicitud de Pago muestre la lista de facturas al comprador.

El comprador selecciona una factura mediante el IU Solicitud de Pago y planifica su pago, y a su vez el IU Solicitud de Pago pasa esta solicitud a Procesamiento de Solicitudes de Pago. Procesamiento de Solicitudes de Pago solicita al Planificador de Pagos la planificación del pago de la factura, y éste crea una Solicitud de Pago. La aplicación de Procesamiento de Solicitudes de Pago solicita después a la aplicación de Procesamiento de Facturas que cambie el estado de la factura a "Planificada".

A medida que vamos detallando los diagramas de interacción, encontraremos muy probablemente nuevos caminos alternativos que puede tomar el caso de uso. Podemos describir estos caminos en las etiquetas de los diagramas o en diagramas de interacción independientes. Al añadir más información, el ingeniero de componentes descubrirá con frecuencia nuevas excepciones que no se tuvieron en cuenta durante la captura o el análisis de los requisitos. Estos tipos de excepciones incluyen:

- Gestión de temporizadores cuando los nodos o las conexiones dejan de funcionar.
- Entradas erróneas que pueden proporcionar los actores, sean éstos humanos o no.
- Mensajes de error generados por la capa intermedia, el software del sistema, o el hardware.

9.5.2.3. Identificación de subsistemas e interfaces participantes

Hasta aquí, hemos diseñado un caso de uso como una colaboración de clases y de sus objetos. Sin embargo, a veces es más apropiado diseñar un caso de uso en términos de los subsistemas y/o interfaces que participan en él. Por ejemplo, en el desarrollo descendente, puede que sea necesario capturar los requisitos sobre los subsistemas y sus interfaces antes de haber diseñado sus contenidos, o en algunos casos debería ser fácil sustituir un subsistema y su diseño interno particular por otro subsistema que tenga otro diseño distinto. En esos casos, podemos describir las realizaciones de caso de uso-diseño en varios niveles de la jerarquía de subsistemas (*véase* la Figura 9.37).

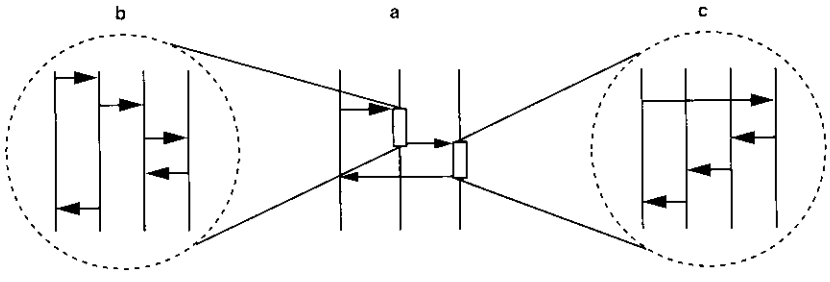


Figura 9.37. Las líneas de vida en el centro del diagrama (a) representan subsistemas, y muestran los mensajes que se envían y reciben entre subsistemas. Los otros diagramas (b, c) representan el diseño interno de los subsistemas, y muestran cómo esos mensajes (los que se muestran en a) los reciben y envían los elementos internos de los subsistemas. El diagrama (a) puede ser diseñado antes que los diagramas (b y c).

Para comenzar, es necesario identificar los subsistemas necesarios para realizar el caso de uso, mediante los siguientes pasos:

- Estudiar las clases del análisis que participan en las correspondientes realizaciones de caso de uso-análisis. Identificar los paquetes del análisis que contienen a esas clases del análisis, si existen. Después, identificar los subsistemas del diseño que poseen una traza hacia esos paquetes del análisis.
- Estudiar los requisitos especiales de las correspondientes realizaciones de caso de uso-análisis. Identificar las clases del diseño que realizan esos requisitos especiales, si existen. Después, identificar los subsistemas del diseño que contienen a esas clases.

Debemos recoger los subsistemas que participan en una realización de caso de uso en un diagrama de clases asociado a la realización. Utilizaremos este diagrama de clases para mostrar las dependencias entre esos subsistemas y cualquier interfaz que se utilice en la realización del caso de uso.

Ejemplo **Subsistemas e interfaces participantes en la realización del caso de uso Pagar Factura**

La Figura 9.38 muestra un diagrama de clases que incluye subsistemas, interfaces, y sus dependencias, para la primera parte del caso de uso Pagar Factura.

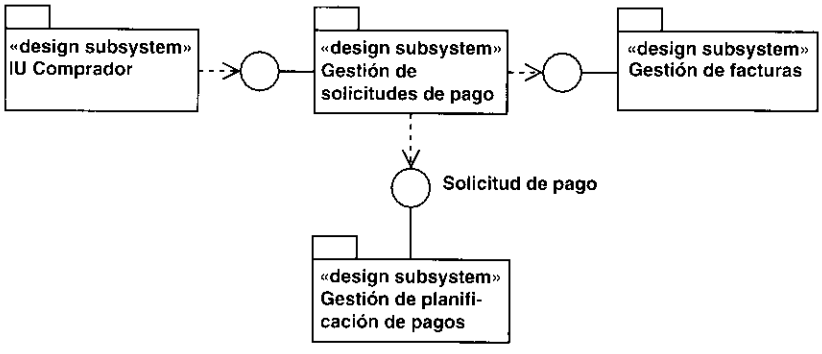


Figura 9.38. Diagrama de clases con subsistemas, interfaces, y sus dependencias.

9.5.2.4. Descripción de interacciones entre subsistemas Cuando tenemos un esbozo de los subsistemas necesarios para realizar el caso de uso, debemos describir cómo interactúan los objetos de las clases que contiene en el nivel de subsistema. Lo haremos mediante diagramas de secuencia que contengan las instancias de actores, subsistemas, y transmisiones de mensajes entre éstos que participan. Al hacerlo, utilizaremos una técnica similar a la descrita en la Sección 9.5.2.2, con las siguientes diferencias:

- Las **líneas de vida** (Apéndice A) en los diagramas de secuencia denotan subsistemas en lugar de objetos del diseño.
- Cada subsistema identificado en la Sección 9.5.2.3 debería tener al menos una línea de vida que lo denote en un diagrama de secuencia.
- Si asignamos un mensaje a una operación de una interfaz, puede resultar apropiado cualificar el mensaje con el interfaz que proporciona la operación. Esto es necesario cuando un subsistema proporciona varias interfaces, y debemos distinguir qué interfaz se utiliza en cada mensaje.

9.5.2.5. Captura de requisitos de implementación En este paso, incluimos en la realización del caso de uso todos los requisitos identificados durante el diseño que deberían tratarse en la implementación, como los requisitos no funcionales.

Ejemplo **Requisitos de implementación del caso de uso Pagar Factura**

El siguiente es un ejemplo de un requisito de implementación hallado en la realización del caso de uso Pagar Factura:

> *Un objeto de la clase activa Procesamiento de Solicitudes de Pago debería ser capaz de soportar 10 clientes de comprador diferentes sin un retraso perceptible para cada determinado comprador.*

9.5.3. Actividad: diseño de una clase

El propsósito de diseñar una clase es crear una clase del diseño que cumpla su papel en las realizaciones de los casos de uso y los requisitos no funcionales que se aplican a estos (*véase* Figura 9.39). Esto incluye el mantenimiento del diseño de clases en sí mismo y los siguientes aspectos de éste:

- Sus operaciones.
- Sus atributos.
- Las relaciones en las que participa.
- Sus métodos (que realizan sus operaciones).
- Los estados impuestos.
- Sus dependencias con cualquier mecanismo de diseño genérico.
- Los requisitos relevantes a su implementación.
- La correcta realización de cualquier interfaz requerida.

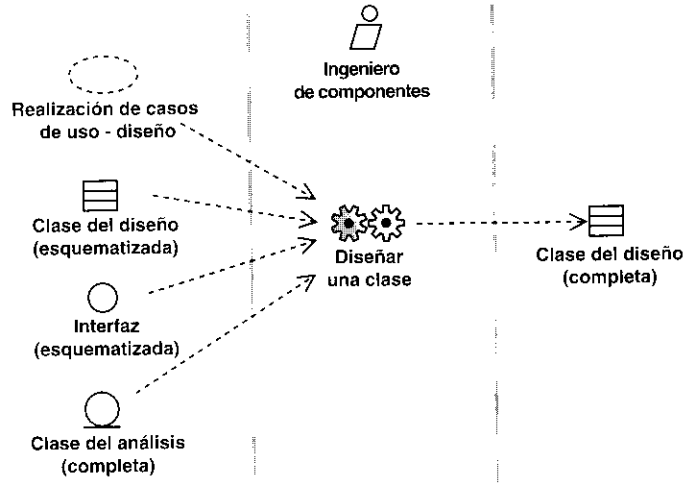


Figura 9.39. La entrada y los resultados del diseño de una clase.

9.5.3.1. Esbozar la clase del diseño Como un primer paso, necesitamos esbozar una o varias clases del diseño, dada la entrada en términos de clases de análisis y/o interfaces. Cuando tomamos una interfaz como entrada, suele ser simple y directo el asignar a una clase de diseño para que proporcione esa interfaz.

Cuando se dan como entrada una o varias clases del análisis, los métodos utilizados dependen del estereotipo de la clase de análisis:

- Diseñar clases de interfaz es dependiente de la tecnología de interfaz específica que se utilice. Por ejemplo, las clases de interfaz diseñadas en Visual Basic implicarían clases del diseño estereotipadas como «forms» junto con otras clases del diseño que representan «controls», posiblemente «controls» de ActiveX, en el interfaz de ususario. También hay que darse cuenta de que en algunas herramientas de diseño de interfaces de usuario modernas, los interfaces de usuario creados de forma visual directamente sobre la panta-

lla, haciendo así implita la creación de las correspondientes clases de diseño. Cualquier prototipo de interfaz de usuario existente es una entrada esencial en esta etapa.

- Diseñar clases de entidad que representen información persistente (o clases que tienen otros requisitos persistentes) a menudo implica el uso de tecnologías de bases de datos específicas. Por ejemplo, se puede incluir la creación de clases de diseño para hacer la correspondencia con tablas en un modelo de datos relacional. Este paso puede automatizarse parcialmente utilizando las herramientas de modelado disponibles actualmente, aunque puede ser bastante delicado y requerir la adopción de una estrategia persistente. Puede haber muchos aspectos del diseño involucrados en la adopción de una estrategia, especialmente a la hora de hacer corresponder el modelo de diseño orientado a objetos con el modelo de datos relacional. Estos aspectos pueden a su vez requerir trabajadores distintos (por ejemplo, diseñadores de bases de datos), actividades distintas (por ejemplo, diseño de bases de datos), y modelos distintos (por ejemplo, modelos de datos) en el proceso de desarrollo —estos casos no son objeto de este libro.
- Diseñar clases de control es una tarea delicada. Debido a que encapsulan secuencias, coordinación de otros obejetos o algunas veces pura lógica del negocio, es necesiario considerar los siguientes aspectos:
  - — Aspectos de distribución: si la secuencia necesita ser distribuida y manejada por diferentes los nodos de una red, se puede requerir separar las clases del diseño en diferentes nodos para realizar la clase de control.
  - — Aspectos de rendimiento: puede que no sea justificable tener clases del diseño separadas para realizar la clase de control. En cambio, la clase de control podría realizarse por las mismas clases del diseño que están realizando algunas clases de interfaz o clases de entidad relacionadas.
  - — Aspectos de transacción: las clases de control a menudo encapsulan transacciones. Sus correpondientes diseños deben incorporar cualquier tecnología de manejo de transacción existente que se esté utilizando.

Las clases de diseño identificadas en este paso deberán ser asignadas trazando dependencias a las correspondientes clases de análisis que son diseñadas. Es importante conservar en mente los "orígenes" de la clase de diseño cuando sea refinada en los pasos posteriores.

9.5.3.2. Identificar operaciones En esta etapa identificamos las operaciones que la clases de diseño van a necesitar y describimos esas operaciones utilizando la sintaxis de los lenguajes de programación. Esto incluye especificar la visibilidad de cada operación (por ejemplo, *public, protected, private* en C++). Algunas entradas importantes en esta etapa son:

- Las responsabilidades de cualquier clase del análisis que tenga una traza con la clase del diseño. Una responsabilidad a menudo implica una o varias operaciones. Es más, si las entradas y salidas se describen para las responsabilidades, pueden ser utilizadas como un primer bosquejo de parámetros formales y valores de retorno de las operaciones.
- Los requisitos especiales de cualquier clase de análisis que tenga una traza con la clase del diseño. Recuérdese que aquellos requisitos a menudo necesitan ser manejados por el modelo de diseño, posiblemente incorporando algún mecanismo o tecnología de diseño genérico o tecnología de base de datos.

- Las interfaces que la clase de diseño necesita proporcionar. Las operaciones de las interfaces también necesitan ser proporcionadas por la clase de diseño.
- Las realizaciones de caso de uso-diseño en las que la clase participa (*véase* Sección 9.5.2).

Las operaciones de la clase de diseño necesitan soportar todos los roles que las clases desempeñan en las diferentes realizaciones de casos de uso. Estos roles se encuentran yendo a través de las realizaciones de casos de uso y mirando si la clase y sus objetos están incluidos en los diagramas y en las descripciones de flujos de sucesos-diseño de las realizaciones.

Ejemplo Operaciones de la clase Factura

La clase Factura participa en varias realizaciones de casos de uso, como aquellas de Pagar Factura, Enviar Aviso y Enviar Factura al Comprador. Cada una de estas realizaciones leen o cambian el estado de los objetos Factura. El caso de uso Enviar Factura al Comprador crea y envía facturas. El caso de uso Pagar Factura planifica Facturas, y así todos. Cada una de estas realizaciones de casos de uso, por tanto, utiliza objetos Factura de forma diferente; en otras palabras, la clase Fatura y sus objetos desempeñan distintos papeles en estas realizaciones de casos de uso.

Primero, los ingenieros de componentes pensaron en implementar estos cambios de estado como una operación llamada setStatus, que tenía un parámetro que indicaba la acción deseada y el estado destino (por ejemplo, setStatus (scheduled)). Pero entonces decidieron que era preferible implementar operaciones explícitas para cada transición de estados. Es más, decidieron utilizar este enfoque no sólo para la clase Factura sino también para la clase Objeto de Comercio, de la cual es un subtipo la clase Factura (*véase* Figura 9.40). La clase Objeto de Comercio soporta de esta forma las siguientes operaciones (cada una de las cuales cambia el estado de Objeto de Comercio): Crear, Enviar, Planificar y Cerrar. No obstante, estas operaciones son solamente operaciones virtuales que definen una signatura. Las clases que son subtipos de la clase Objeto de Comercio tienen que definir cada una un método concreto que realice estas operaciones.

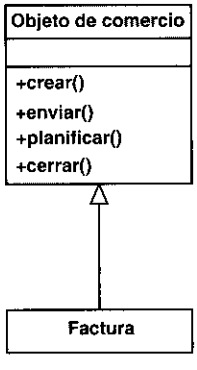


Figura 9.40. La clase Factura, su supertipo Objeto de Comercio, y las operaciones que soporta.

Para algunas clases el comportamiento de sus objetos depende fuertemente del estado del objeto. Estas clases se decriben mejor utilizando diagramas de estado. *Véase* Sección 9.5.3.7.

9.5.3.3. Identificar atributos En este paso, identificamos los atributos requeridos por la clase de diseño y los describimos utilizando la sintaxis del lenguaje de programación. Un atributo especifica una propiedad de una clase de diseño y está a menudo implicado y es requerido por las operaciones de la clase (como se consideró en el paso anterior). Las siguientes normas generales han de tenerse en cuenta cuando se identifiquen atributos:

- Considerar los atributos sobre cualquier clase de análisis que tenga una traza con la clase de diseño. Algunas veces estos atributos implican la necesidad de uno o más atributos de la clase de diseño.
- Los tipos de atributos disponibles están restringidos por el lenguaje de programación.
- Cuando se elige un tipo de atributo, intentar reutilizar uno que ya exista.
- Una simple instancia de atributo no puede ser compartida por varios objetos de diseño. Si esto fuera necesario, el atributo necesita ser definido en una clase separada.
- Si una clase de diseño llega a ser muy complicada de comprender por culpa de sus atributos, algunos de estos atributos pueden ser separados en clases independientes.
- Si hay muchos atributos o son complejos los atributos de una clase, se puede ilustrar ésta en un diagrama de clases separado que muestre solamente el apartado de atributos.

9.5.3.4. Identificar asociaciones y agregaciones Los objetos de diseño interactúan unos con otros en los diagramas de secuencia. Estas interacciones a menudo requieren asociaciones entre las clases correspondientes. Los ingenieros de componentes deben, por tanto, estudiar la transmisión de mensajes en los diagramas de secuencia para determinar qué asociaciones son necesarias. Las instancias de las asociaciones deben ser utilizadas para abarcar las referencias con otros objetos, y para agrupar objetos en agregaciones con el propósito de enviarles mensajes.

El número de relaciones entre clases debe estar minimizado. Estas no son en un principio relaciones del mundo real que deban ser modeladas como asociaciones o agregaciones, pero las relaciones deben existir en respuesta a las demandas de varias realizaciones de casos de uso. Nótese también que puesto que los aspectos de rendimiento necesitan ser manejados durante el diseño, debe requerirse el modelado de rutas de búsqueda óptimas a través de las asociaciones y agregaciones.

Deben tenerse en cuenta las siguientes directrices generales a la hora de definir o redefinir las asociaciones y agregaciones:

- Considerar las asociaciones y agregaciones involucrando la correspondiente clase de análisis (o clases). Algunas veces estas relaciones (en el modelo de análisis) implican la necesidad de una o varias relaciones correspondientes (en el modelo de diseño) que involucre la clase de diseño.
- Refinar la multiplicidad de las asociaciones, nombres de rol, clases de asociación, roles de ordenación, roles de cualificación y asociaciones *n*-arias de acuerdo con el soporte del lenguaje de programación utilizado. Por ejemplo, los nombres de los roles pueden llegar a ser atributos de la clase de diseño cuando se genere el código, por lo tanto se restringe la forma de los nombres de roles. O, una asociación de clases puede llegar a ser una clase nueva entre las dos clases originales, por lo que se requiere nuevas asociaciones con la multiplicidad apropiada entre la clase "asociación" y las otras dos clases.

- Refinar la navegabilidad de las asociaciones. Considerar los diagramas de interacción en los que se emplean asociaciones. La dirección de las transmisiones de los mensajes entre los objetos de diseño implica que se corresponda con la navegabilidad de las asociaciones entre sus clases.

9.5.3.5. Identificar las generalizaciones Las generalizaciones deben ser utilizadas con la misma semántica definida en el lenguaje de programación. Si el lenguaje de programación no admite generalización (o herencia), las asociaciones y/o agregaciones deben utilizarse en lugar de ésta para proporcionar delegación desde los objetos de clases más específicas a objetos de clases más generales (por ejemplo, enviar un mensaje como petición para realizar trabajo, obtener el trabajo realizado y un mensaje de confirmación).

Ejemplo **Generalizaciones en el modelo diseño**

Tanto Facturas como Pedidos cambian de estado de forma similar y soportan operaciones similares. Ambas son especializaciones de una clase Objeto de Comercio más general (*véase* Figura 9.41).

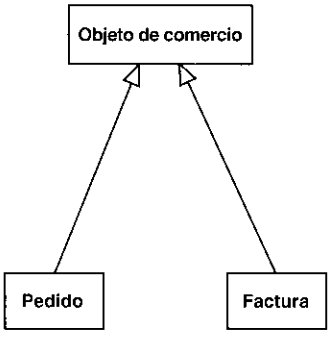


Figura 9.41. La clase Objeto de Comercio generaliza Facturas y Pedidos. Adviértase que estas generalizaciones también existen en el modelo de análisis entre las correspondientes clases de análisis.

9.5.3.6. Describir los métodos Los métodos pueden ser utilizados durante el diseño para especificar cómo se deben realizar las operaciones. Por ejemplo, un método puede especificar un algoritmo que sea utilizado para realizar una operación. El método puede ser especificado utilizando lenguaje natural o pseudocódigo si se considera más apropiado.

No obstante, la mayoría de las veces los métodos no son especificados durante el diseño. En cambio, pueden ser creados durante la implementación utilizando un lenguaje de programación directamente. Esto es porque el mismo ingeniero de componentes debe diseñar e implementar las clases, de forma que se elimine la necesidad de menejo de especificaciones como las de los métodos.

Nótese que si la herramienta de diseño soporta una generación de código integrada para los métodos de las clases de diseño, pueden especificarse los métodos directamente en la herramienta de diseño utilizando un lenguaje de programación, pero esto se considera una actividad

de implementación, y no se trata dentro del diseño. Les referimos al Capítulo 10 y al tema de la actividad de implementación de clases (Sección 10.5.4) para más detalles.

9.5.3.7. Describir estados Algunos objetos del diseño son estados controlados, lo que significa que sus estados determinan su comportamiento cuando reciben un mensaje. En estos casos, es significativa la utilización de diagramas de estado para describir las diferentes transiciones de estado de un objeto del diseño. Cada diagrama de estados es entonces una entrada de valor para la implementación de la correspondiente clase del diseño.

Ejemplo **Diagrama de estados para la clase Factura**

Los objetos Factura cambian de estado según sean creados, enviados, planificados o cerrados. Como todos los objetos de comercio, estos estados cambian siguiendo una estricta secuencia. Por ejemplo, una factura puede no ser planificada antes de haber sido enviada. Esta secuencia de estados puede ser definida utilizando un diagrama de estados, *véase* Figura 9.42.

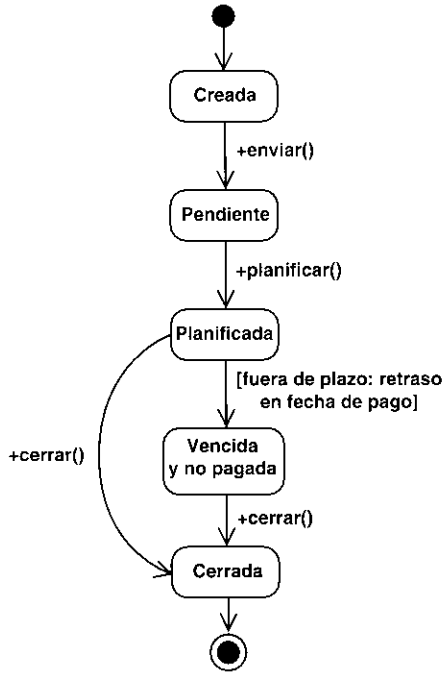


Figura 9.42. Un diagrama de estados para la clase Factura.

Una factura se crea cuando un vendedor quiere que un comprador pague un pedido. Entonces la factura es enviada al comprador, y el estado cambia a pendiente. Cuando el comprador decide pagar, el estado de la factura transita a planificada. Entonces, si el comprador no paga a tiempo, el estado de la factura transita a vencida y no pagada. Finalmente, como la factura se ha pagado, el estado transita a cerrada.

9.5.3.8. Tratar requisitos especiales En esta fase debe ser tratado cualquier requisito que no haya sido considerado en pasos anteriores. Cuando se hace esto, hay que estudiar los requisitos formulados en la realización de los casos de uso en los que la clase participa. Éstos pueden establecer requisitos (no funcionales) para la clase de diseño. También es necesario estudiar los requisitos especiales de cualquier clase de análisis que trace la clase de diseño. Estos requisitos especiales a menudo necesitan ser manejados por la clase de diseño.

Ejemplo **Emplear un mecanismo de diseño para manejar requisitos especiales**

Los objetos Factura necesitan ser accedidos desde diferentes nodos, desde el Servidor del Comprador y desde el Servidor del Vendedor. La Factura no es una clase activa, pero tiene que ser diseñada para un sistema distribuido. En nuestro ejemplo, implementamos esta distribución de objetos haciendo la clase Factura una subclase de la clase abstracta de Java java.rmi.UnicastRemoteObject, que soporta la Invocación de Mensajes Remotos (RMI), *véase* Figura 9.43. Adviértase que este mecanismo de diseño está identificado y descrito por el arquitecto en la actividad de diseño arquitectónico.

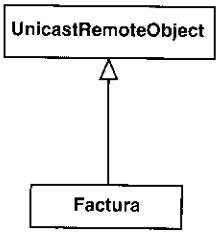


Figura 9.43. Los objetos de la clase Factura deben estar distribuidos. Esto se realiza convirtiéndolos en subtipos de UnicastRemoteObjects.

Al tratar estos requisitos, debemos utilizar siempre que sea posible los mecanismos de diseño que identificó el arquitecto.

Sin embargo, podría ser adecuado posponer hasta la implementación el tratamiento de algunos requisitos. Esos requisitos deberían anotarse como *requisitos de implementación* para la clase del diseño.

Ejemplo **Requisito de implementación para una clase Activa**

El siguiente es un ejemplo de un requisito de implementación para la clase Procesamiento de Solicitud de Pago.

- Un objeto de la clase Procesamiento de Solicitud de Pago debería ser capaz de gestionar 10 clientes de Comprador diferentes sin un retardo perceptible para los compradores.

9.5.4. Actividad: diseño de un subsistema

Los objetivos del diseño de un subsistema son (*véase* la Figura 9.44):

- Garantizar que el subsistema es tan independiente como sea posible de otros subsistemas y/o de sus interfaces.
- Garantizar que el subsistema proporciona las interfaces correctos.
- Garantizar que el subsistema cumple su propósito de ofrecer una realización correcta de las operaciones tal y como se definen en las interfaces que proporciona.

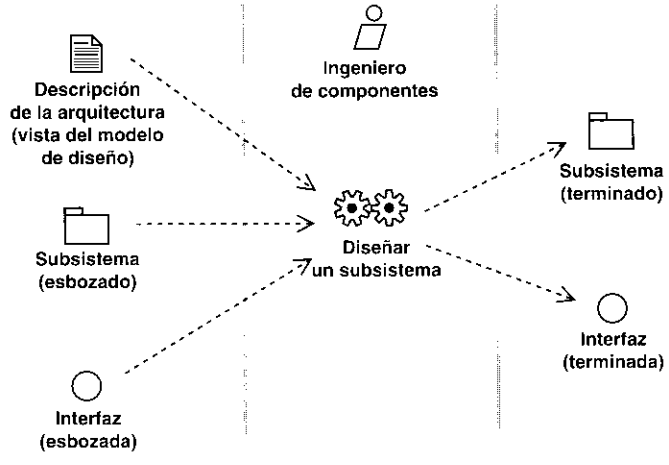


Figura 9.44. Las entradas y los resultados del diseño de un subsistema.

9.5.4.1. Mantenimiento de las dependencias entre subsistemas

Deberían definirse y mantenerse dependencias en un subsistema respecto a otros cuando los elementos contenidos en estos últimos estén asociados con elementos dentro de aquél. No obstante, si esos otros subsistemas proporcionan interfaces, las dependencias (de uso) deberían declararse hacia las interfaces en vez de hacia los propios subsistemas. Es mejor ser dependiente de un interfaz que serlo de un subsistema, ya que un subsistema podría ser sustituido por otro que posea un diseño interno distinto, mientras que en ese mismo caso, no estaríamos obligados a sustituir la interfaz.

Debemos intentar minimizar las dependencias entre subsistemas y/o interfaces, tratando de reubicar las clases contenidas en un subsistema que sean demasiado dependientes de otros subsistemas.

9.5.4.2. Mantenimiento de interfaces proporcionadas por el subsistema

Las operaciones definidas por las interfaces que proporciona un subsistema deben soportar todos los roles que cumple el subsistema en las diferentes realizaciones de caso de uso. Incluso si las interfaces fueron esbozadas por los arquitectos, puede ser necesario que el ingeniero de componentes las refine a medida que evoluciona el modelo de diseño y se van

diseñando los casos de uso. Recuérdese que los ingenieros de componentes pueden utilizar un mismo subsistema y sus interfaces dentro de varias realizaciones de caso de uso, proporcionando por ello más requisitos sobre las interfaces (*véase* la Sección 9.5.2.4).

La técnica para mantener interfaces y sus operaciones es similar a la técnica para mantener clases del diseño y sus operaciones, tal y como se describió en la Sección 9.5.3.2.

9.5.4.3. Mantenimiento de los contenidos de los subsistemas

Un subsistema cumple sus objetivos cuando ofrece una realización correcta de las operaciones tal y como están descritas por las interfaces que proporciona. Aún cuando haya sido el arquitecto quien esbozó los contenidos de los subsistemas, puede ser necesario que los ingenieros de componentes los refinen a medida que evoluciona el modelo de diseño. Algunos aspectos generales relacionados con esto son los siguientes:

- Por cada interfaz que proporcione el subsistema, debería haber clases del diseño u otros subsistemas dentro del subsistema que también proporcionen la interfaz.
- Para clarificar cómo el diseño interno de un subsistema realiza cualquiera de sus interfaces o casos de uso, podemos crear colaboraciones en términos de los elementos contenidos en el subsistema. Podemos hacer esto para justificar los elementos contenidos en el subsistema.

Ejemplo **Una clase del diseño que proporciona una interfaz dentro de un subsistema**

El subsistema Gestión de Facturas del Comprador proporciona la interfaz Factura. El ingeniero de componentes responsable del subsistema decide hacer que la clase Factura realice esa interfaz, como se muestra en la Figura 9.45. Una realización alternativa podría haber sido hacer que alguna otra clase del diseño realizase la interfaz, y que a su vez usase a la clase Factura.

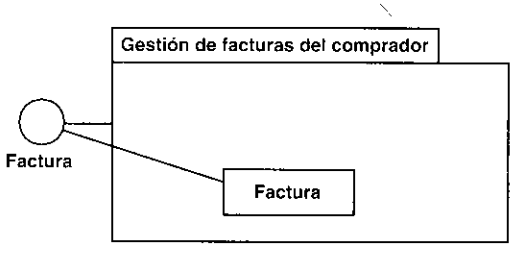


Figura 9.45. La clase Factura proporciona la interfaz Factura que proporciona el subsistema Gestión de Facturas del Comprador.

9.6. Resumen del diseño

El principal resultado del diseño es el modelo de diseño que se esfuerza en conservar la estructura del sistema impuesta por el modelo de análisis, y que sirve como esquema para la implementación. El modelo de diseño incluye los siguientes elementos:

- Subsistemas del diseño y subsistemas de servicio y sus dependencias, interfaces y contenidos. Los subsistemas del diseño de las dos capas superiores (las capas específica y general de la aplicación) se obtienen a partir de los paquetes del análisis (Sección 9.5.1.2.1). Algunas de las dependencias entre subsistemas del diseño se obtienen a partir de las correspondientes dependencias entre paquetes del análisis (Sección 9.5.1.2.3). Algunos de las interfaces se obtienen a partir de las clases del análisis (Sección 9.5.1.2.4).

- Clases del diseño, incluyendo las clases activas, y sus operaciones, atributos y requisitos de implementación. Algunas clases del diseño relevantes para la arquitectura se obtienen a partir de las clases del análisis relevantes para la arquitectura (Sección 9.5.1.3.1). Algunas clases activas se obtienen a partir de clases del análisis (Sección 9.5.1.3.2). En general, las clases del análisis se utilizan como especificaciones al obtener las clases del diseño (Sección 9.5.3.1).

- Realizaciones de caso de uso-diseño, que describen cómo se diseñan los casos de uso en términos de colaboraciones dentro del modelo de diseño. En general, al obtener las realizaciones de caso de uso-diseño utilizamos las realizaciones de caso de uso-análisis como especificaciones (Sección 9.5.2).

- La vista arquitectónica del modelo de diseño, incluyendo sus elementos significativos de cara a la arquitectura. Como ya se dijo, se utilizan como especificaciones los elementos significativos para la arquitectura del modelo de análisis cuando se identifican los elementos significativos para la arquitectura del modelo de diseño.

El diseño también obtiene como resultado un modelo de despliegue, que describe todas las configuraciones de red sobre las cuáles debería implantarse el sistema. El modelo de despliegue incluye:

- Nodos, sus características, y sus conexiones.

- Una correspondencia inicial de clases activas sobre los nodos.

- La vista arquitectónica del modelo de despliegue, que incluye a sus elementos relevantes para la arquitectura. --

Como presentaremos en los siguientes capítulos, los modelos de diseño y de despliegue se consideran la entrada principal para las subsiguientes actividades de implementación y prueba. Más en concreto:

- Los subsistemas del diseño y los subsistemas de servicio serán implementados mediante subsistemas de implementación que contienen los verdaderos componentes: ficheros de código fuente, *scripts*, binarios, ejecutables y similares. Estos subsistemas de implementación poseerán una traza uno-a-uno (isomórfica) hacia los subsistemas del diseño.

- Las clases del diseño se implementarán mediante componentes de fichero que contendrán el código fuente. Es frecuente implementar varias clases del diseño dentro de un mismo componente de fichero, aunque esto depende del lenguaje de programación que estemos utilizando. Por otro lado, las clases activas del diseño que representen procesos pesados se utilizarán como entrada cuando identifiquemos los componentes ejecutables.

- Las realizaciones de caso de uso-diseño se utilizarán al planificar y llevar a cabo el esfuerzo de implementación en pasos pequeños y manejables, que dan como resultado

"construcciones". Cada construcción implementará principalmente un conjunto de realizaciones de caso de uso o partes de ellas.

El modelo de despliegue y las configuraciones de red se utilizarán al distribuir el sistema instalando componentes ejecutables en los nodos.

9.7. Referencias

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley, 1994.

[2] Ivar Jacobson, Stefan Bylund, Patrik Jonsson, Staffan Ehnebom, "Using contracts and use cases to build plugable architectures", *Journal of Object-Oriented Programming*, June, 1995.

[3] Ivar Jacobson, Martin Griss, and Patrik Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*, Reading MA: Addison-Wesley, 1997.

Capítulo 10

Implementación

10.1. Introducción

En la implementación empezamos con el resultado del diseño e implementamos el sistema en términos de componentes, es decir, ficheros de código fuente, *scripts*, ficheros de código binario, ejecutables y similares.

Afortunadamente, la mayor parte de la arquitectura del sistema es capturada durante el diseño, siendo el propósito principal de la implementación el desarrollar la arquitectura y el sistema como un todo. De forma más específica, los propósitos de la implementación son:

- Planificar las integraciones de sistema necesarias en cada iteración. Seguimos para ello un enfoque incremental, lo que da lugar a un sistema que se implementa en una sucesión de pasos pequeños y manejables.
- Distribuir el sistema asignando componentes ejecutables a nodos en el diagrama de despliegue. Esto se basa fundamentalmente en las clases activas encontradas durante el diseño.
- Implementar las clases y subsistemas encontrados durante el diseño. En particular, las clases se implementan como componentes de fichero que contienen código fuente.
- Probar los componentes individualmente, y a continuación integrarlos compilándolos y enlazándolos en uno o más ejecutables, antes de ser enviados para ser integrados y llevar a cabo las comprobaciones de sistema.

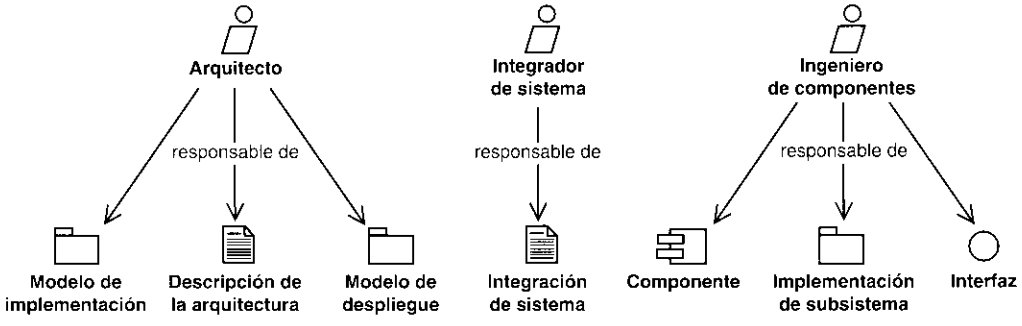


Figura 10.1. Los trabajadores y artefactos involucrados en la implementación.

En éste y en los capítulos siguientes presentaremos cómo se lleva a cabo la implementación y qué trabajadores y artefactos están involucrados (*véase* Figura 10.1). Seguimos la línea de trabajo que seguimos para el diseño.

10.2. El papel de la implementación en el ciclo de vida del software

La implementación es el centro durante las iteraciones de construcción, aunque también se lleva a cabo trabajo de implementación durante la fase de elaboración, para crear la línea base ejecutable de la arquitectura, y durante la fase de transición, para tratar defectos tardíos como los encontrados con distribuciones beta del sistema (como se indica en la Figura 10.2 con un pico en la columna de transición).

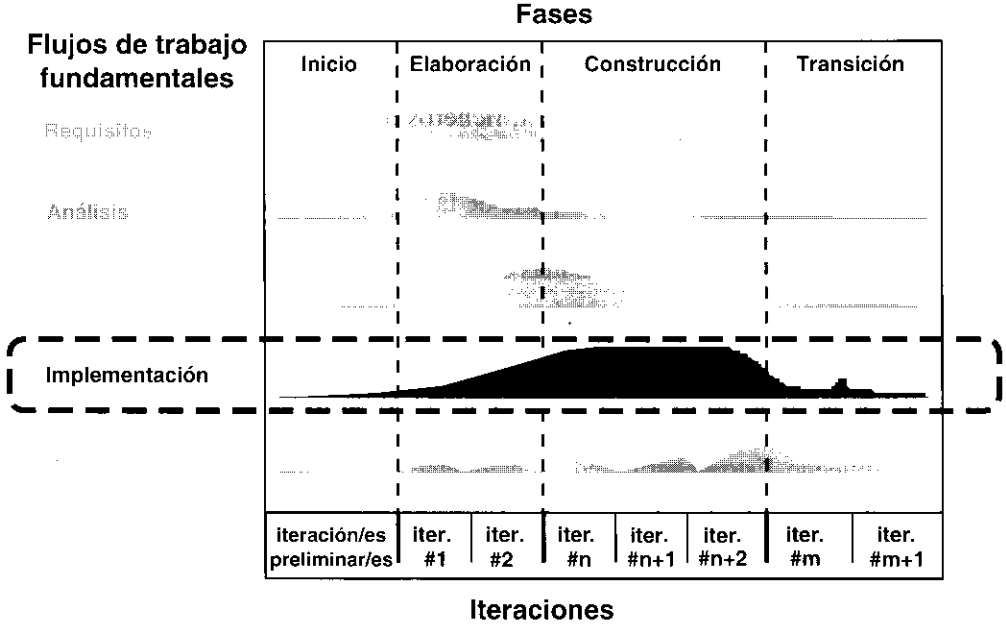


Figura 10.2. La implementación.

Ya que el modelo de implementación denota la implementación actual del sistema en términos de componentes y subsistemas de implementación, es natural mantener el modelo de implementación a lo largo de todo el ciclo de vida del software.

10.3. Artefactos

10.3.1. Artefacto: modelo de implementación

El modelo de implementación describe cómo los elementos del modelo de diseño, como las clases, se implementan en términos de componentes, como ficheros de código fuente, ejecutables, etc. El modelo de implementación describe también cómo se organizan los componentes de acuerdo con los mecanismos de estructuración y modularización disponibles en el entorno de implementación y en el lenguaje o lenguajes de programación utilizados, y cómo dependen los componentes unos de otros.

El modelo de implementación define una jerarquía, tal y como se ilustra en la Figura 10.3.

El modelo de implementación se representa con un sistema de implementación que denota el subsistema de nivel superior del modelo. El utilizar otros subsistemas es por tanto una forma de organizar el modelo de implementación en trozos más manejables.

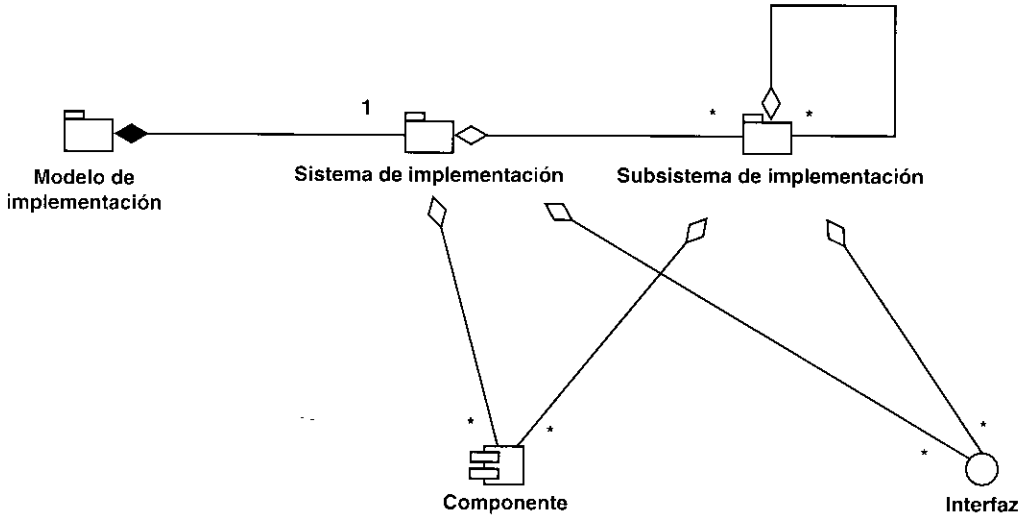


Figura 10.3. El modelo de implementación es una jerarquía de subsistemas de implementación que contiene componentes e interfaces.

10.3.2. Artefacto: componente

Un componente es el empaquetamiento físico de los elementos de un modelo, como son las clases en el modelo de diseño [5]. Algunos estereotipos estándar de componentes son los siguientes:

- «executable» es un programa que puede ser ejecutado en un nodo.
- «file» es un fichero que contiene código fuente o datos.
- «library» es una librería estática o dinámica.
- «table» es una tabla de base de datos.
- «document» es un documento.

Durante la creación de componentes en un entorno de implementación particular estos estereotipos pueden ser modificados para reflejar el significado real de estos componentes. Los componentes tienen las siguientes características:

- Los componentes tienen relaciones de traza con los elementos de modelo que implementan.
- Es normal que un componente implemente varios elementos, por ejemplo, varias clases; sin embargo, la forma exacta en que se crea esta traza depende de cómo van a ser estructurados y modularizados los ficheros de código fuente, dado el lenguaje de programación que se esté usando.

Ejemplo **Un componente que sigue la traza de una clase de diseño**

En el sistema Interbank, la clase de diseño Transferencias entre Cuentas se implementa en el componente de código fuente TransferenciasEntreCuentas.java. Esto es debido a la convención y uso común en Java de crear un fichero de código fuente ".java" para cada clase, aunque esto no es obligatorio. En cualquier caso, esto se modela con una dependencia de traza entre el modelo de diseño y el de implementación (*véase* Figura 10.4).

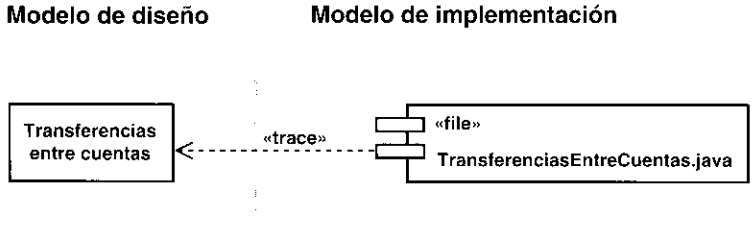


Figura 10.4. Dependencias de traza entre componentes y clases de diseño.

- Los componentes proporcionan las mismas interfaces que los elementos de modelo que implementan.

Ejemplo **Interfaces en el diseño y en la implementación**

En el sistema Interbank, la clase de diseño Transferencias entre Cuentas proporciona una interfaz Transferencias. Esta interfaz es también proporcionado por el componente TransferenciasEntreCuentas.java, el cual implementa la clase Transferencias entre Cuentas (*véase* Figura 10.5).

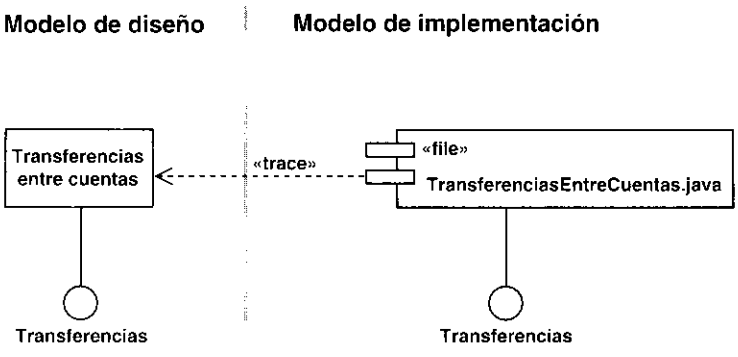


Figura 10.5. Los componentes proporcionan las mismas interfaces que las clases que implementan.

- Puede haber dependencias de compilación entre componentes, denotando qué componentes son necesarios para compilar un componente determinado.

Ejemplo Dependencias de compilación entre componentes

En el sistema Interbank, el componente (fichero) TransferenciasEntreCuentas.java se compila a una componente (ejecutable)[1] TransferenciasEntreCuentas.class (*véase* Figura 10.6).

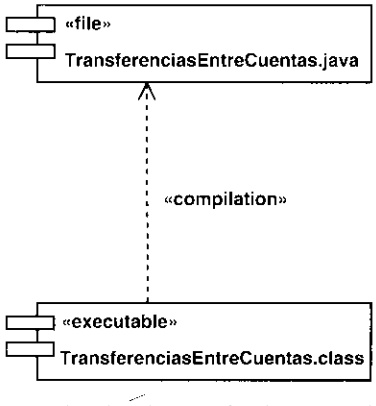


Figura 10.6. Dependencias de compilación entre dos componentes.

10.3.2.1. Stubs Un *stub*[2] es un componente con una implementación esquelética o de propósito especial que puede ser utilizada para desarrollar o probar otro componente que depende del *stub;* una definición similar aparece en [1]. Los *stubs* pueden ser utilizados para minimizar el número de componentes nuevos necesarios en cada nueva versión (intermedia) del sistema, simplificando así los problemas de integración y las pruebas de integración (*véase* Sección 10.5.2, "Actividad: integrar el sistema").

[1] Para ser más concretos, el componente es en realidad *interpretable* en la máquina virtual de Java.

[2] Por coherencia con las anteriores traducciones y con el estereotipo del mismo nombre hemos decidido mantener el original en inglés. *N. del RT.*

10.3.3. Artefacto: subsistema de implementación

Los subsistemas de implementación proporcionan una forma de organizar los artefactos del modelo de implementación en trozos más manejables (*véase* Figura 10.7). Un subsistema puede estar formado por componentes, interfaces y otros subsistemas (recursivamente). Además, un subsistema puede implementar —y así proporcionar— las interfaces que representan la funcionalidad que exportan en forma de operaciones.

Es importante entender que un subsistema de implementación se manifiesta a través de un "mecanismo de empaquetamiento" concreto en un entorno de implementación determinado, tales como:

- Un *paquete* en Java [2].
- Un *proyecto* en Visual Basic [3].
- Un *directorio* de ficheros en un proyecto de C++.
- Un *subsistema* en un entorno de desarrollo integrado como Rational Apex.
- Un *paquete* en una herramienta de modelado visual como Rational Rose.

Por tanto, la semántica de la noción de *subsistema de implementación* se refinará ligeramente cuando se manifieste en un entorno de implementación determinado. Sin embargo, en este capítulo discutimos la implementación a un nivel general, aplicable a la mayoría de los entornos de implementación.

Los subsistemas de implementación están muy relacionados con los subsistemas de diseño en el modelo de diseño (*véase* Capítulo 9, Sección 9.3.4, "Artefacto: subsistema de diseño"). De hecho, los subsistemas de implementación deberían seguir la traza uno a uno (isomórficamente) de sus subsistemas de diseño correspondientes. Recordemos que un subsistema de diseño ya define:

- Las dependencias sobre otros subsistemas o interfaces de otros subsistemas.
- Las interfaces que han de ser proporcionadas por el subsistema.
- Las clases de diseño o, de forma recursiva, subsistemas de diseño dentro del subsistema que deberían proporcionar las interfaces proporcionadas por el subsistema mismo.

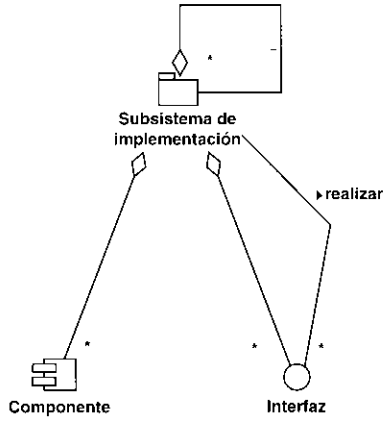


Figura 10.7. Contenido y asociaciones clave de un subsistema.

Estos aspectos son importantes para el subsistema de implementación correspondiente por las siguientes razones:

- El subsistema de implementación debería definir dependencias análogas hacia otros subsistemas de implementación o interfaces (correspondientes).
- El subsistema de implementación debería proporcionar las mismas interfaces.
- El sistema de implementación debería definir qué componentes o, recursivamente, qué otros subsistemas de implementación dentro del subsistema deberían proporcionar las interfaces proporcionadas por el subsistema. Además, estos componentes (o subsistemas de implementación) contenidos deberían seguir la traza de las clases (o subsistemas de diseño) correspondientes en el subsistema de diseño que implementan.

La Figura 10.8 aclara la relación entre los modelos de diseño e implementación.

Cualquier cambio en el modo en que los subsistemas proporcionan y usan interfaces, o cualquier cambio en las interfaces mismas, está descrito en el flujo de trabajo del diseño (*véase* Capítulo 9, Sección 9.5.4). Por tanto, estos cambios no son tratados en el flujo de trabajo de la implementación, aunque también afectan al modelo de implementación.

Obsérvese que la correspondencia descrita también se da para los subsistemas de servicio en el modelo de diseño. Por tanto, los subsistemas de implementación que siguen la traza de los subsistemas de servicio residirán en un nivel inferior en la jerarquía del modelo de implemen-

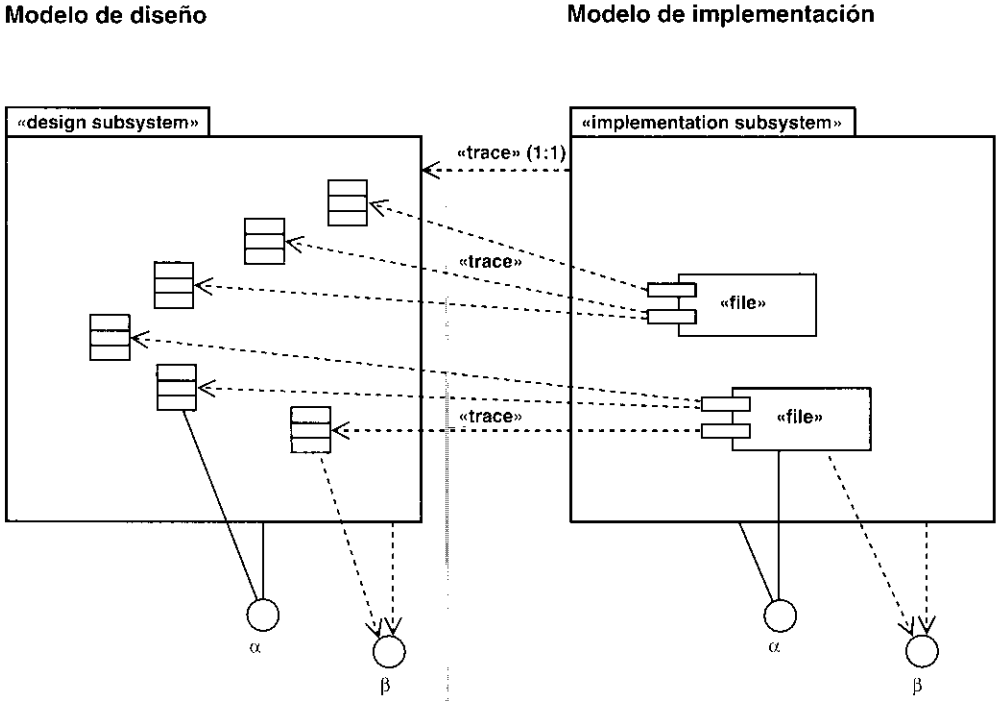


Figura 10.8. Un subsistema de implementación en el modelo de implementación que sigue la traza uno a uno a un subsistema de diseño en el modelo de diseño. Los subsistemas en los diferentes modelos proporcionan la misma interfaz (α) y dependen de la misma interfaz (β). Los componentes en el sistema de implementación implementan clases en el subsistema de diseño.

tación y cumplirán el mismo propósito que los subsistemas de servicio, es decir, estructurar el sistema de acuerdo con los servicios que proporciona. Por esta razón, los subsistemas de implementación que siguen el rastro de subsistemas de servicio en el modelo de diseño encapsularán muy probablemente componentes ejecutables que proporcionen los diversos servicios del sistema.

10.3.4. Artefacto: interfaz

Las interfaces han sido descritas en detalle en el Capítulo 9. Sin embargo, las mismas interfaces pueden ser utilizadas en el modelo de implementación para especificar las operaciones implementadas por componentes y subsistemas de implementación. Además, como se mencionó anteriormente, los componentes y los subsistemas de implementación pueden tener "dependencias de uso" sobre interfaces (Figura 10.9).

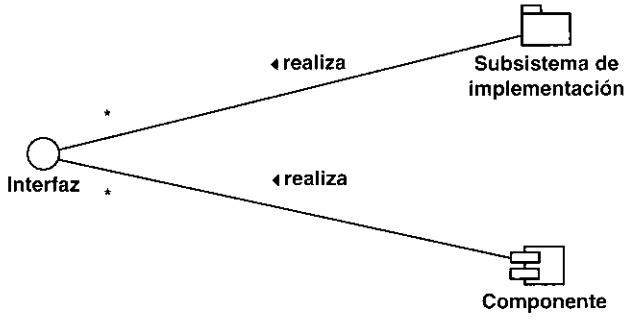


Figura 10.9. Las asociaciones fundamentales de una interfaz.

Un componente que implementa (y por tanto proporciona) una interfaz ha de implementar correctamente todas las operaciones definidas por la interfaz. Un subsistema de implementación que proporciona una interfaz tiene también que contener componentes que proporcionen la interfaz u otros subsistemas (recursivamente) que proporcionen la interfaz.

Ejemplo **Un subsistema de implementación que proporciona una interfaz**

El sistema Interbank tiene un subsistema de implementación llamado GestiónDeCuentas (un paquete Java), el cual proporciona la interfaz Transferencias (*véase* Figura 10.10).

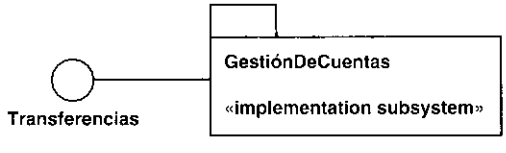


Figura 10.10. El subsistema GestiónDeCuentas proporciona la interfaz Transferencias.

La Figura 10.11 muestra el código Java de la interfaz Transferencias.

```
package GestiónDeCuentas;
// interfaces proporcionados:
public interface Transferencias {
        public Cuenta Crear (Cliente propietario, Dinero saldo,
                                        NúmeroCuenta id_cuenta);
        public void Depositar (Dinero cantidad, Cadena razón);
}
```

Figura 10.11. Código Java para la interfaz Transferencias proporcionado por el subsistema GestiónDeCuentas.

10.3.5. Artefacto: descripción de arquitectura (vista del modelo de implementación)

La descripción de arquitectura contiene una **visión de la arquitectura del modelo de implementación** (apéndice C), el cual representa sus artefactos significativos arquitectónicamente (*véase* Figura 10.12).

Los siguientes artefactos son usualmente considerados en el modelo de implementación significativos arquitectónicamente:

- La descomposición del modelo de implementación en subsistemas, sus interfaces y las dependencias entre ellos. En general, esta descomposición es muy significativa para la arquitectura. Sin embargo, ya que los subsistemas de implementación siguen la traza de los subsistemas de diseño uno a uno y los subsistemas de diseño serán muy probablemente representados en la vista de la arquitectura del modelo de diseño, usualmente es innecesario representar los subsistemas de implementación en la vista de la arquitectura del modelo de implementación.

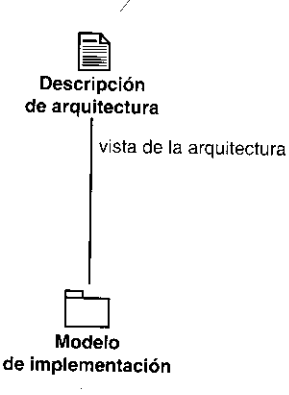


Figura 10.12. Vista de la arquitectura del modelo de implementación.

- Componentes clave, como los componentes que siguen la traza de las clases de diseño significativas arquitectónicamente, los componentes ejecutables y los componentes que son generales, centrales, que implementan mecanismos de diseño genéricos de los que dependen muchos otros componentes.

En la Sección 10.5.1.1, "Identificación de los componentes significativos arquitectónicamente", damos ejemplos de lo que podría incluirse en la vista de la arquitectura del modelo de implementación.

10.3.6. Artefacto: plan de integración de construcciones

Es importante construir el software incrementalmente en pasos manejables, de forma que cada paso dé lugar a pequeños problemas de integración o prueba. El resultado de cada paso es llamado "construcción", que es una versión ejecutable del sistema, usualmente una parte específica del sistema. Cada construcción es entonces sometida a pruebas de integración (como se describe en el Capítulo 11) antes de que se cree ninguna otra construcción. Para prepararse ante el fallo de una construcción (por ejemplo, si no pasa las pruebas de integración) se lleva un control de versiones de forma que se pueda volver atrás a una construcción anterior. Los beneficios de este enfoque incremental son los siguientes:

- Se puede crear una versión ejecutable del sistema muy pronto, en lugar de tener que esperar a una versión más completa. Las pruebas de integración comienzan pronto, y las versiones ejecutables pueden ser utilizadas para hacer demostraciones de las características del sistema tanto a los participantes directos en el proyecto como a otras personas involucradas en él.
- Es más fácil localizar defectos durante las pruebas de integración, porque sólo se añade o refina en una construcción existente una parte pequeña y manejable del sistema. Incluso mejor, los defectos están muy probablemente (aunque no siempre) relacionados con la parte nueva o refinada.
- Las pruebas de integración tienden a ser más minuciosas que las pruebas del sistema completo porque se centran en partes más pequeñas y más manejables.

La **integración incremental** (apéndice C) es para la integración del sistema lo que el desarrollo iterativo controlado es para el desarrollo de software en general. Ambos se centran en un incremento bastante pequeño y manejable de la funcionalidad.

Poniendo esto en un contexto de desarrollo iterativo, cada iteración resultará en al menos una construcción. Sin embargo, la funcionalidad a ser implementada en una iteración determinada es a menudo demasiado compleja para ser integrada en una sola construcción. En lugar de esto, puede que se cree una secuencia de construcciones dentro de una iteración, cada una de las cuales representará un paso manejable en el que se hace un pequeño incremento al sistema. Cada iteración resultará entonces en un incremento mayor del sistema, posiblemente acumulado a lo largo de varias construcciones (*véase* Capítulo 5).

Un *plan de integración de construcciones* describe la secuencia de construcciones necesarias en una iteración. Más concretamente, un plan de este tipo describe lo siguiente para cada construcción:

- La funcionalidad que se espera que sea implementada en dicha construcción. Consiste en una lista de casos de uso o escenarios o parte de ellos, como se discutió en capítulos anteriores. Esta lista puede también referirse a otros requisitos adicionales.
- Las partes del modelo de implementación que están afectadas por la construcción. Esto es una lista de los subsistemas y componentes necesarios para implementar la funcionalidad supuesta por la construcción.

En la Sección 10.5.2, "Actividad: integrar el sistema", describimos un enfoque sistemático para crear planes de integración de construcciones.

10.4. Trabajadores

10.4.1. Trabajador: arquitecto

Durante la fase de implementación, el arquitecto es responsable de la integridad del modelo de implementación y asegura que el modelo como un todo es correcto, completo y legible. Como en el análisis y el diseño, para sistemas grandes y complejos puede introducirse un trabajador adicional para asumir la responsabilidad del subsistema de nivel más alto (es decir, el sistema de implementación) del modelo de implementación.

El modelo es correcto cuando implementa la funcionalidad descrita en el modelo de diseño y en los requisitos adicionales (relevantes), y sólo esta funcionalidad.

El arquitecto es responsable también de la arquitectura del modelo de implementación, es decir, de la existencia de sus partes significativas arquitectónicamente como se representó en la vista de la arquitectura del modelo de despliegue. Recordemos que esta vista es parte de la descripción de la arquitectura del sistema. *Véase* Figura 10.13.

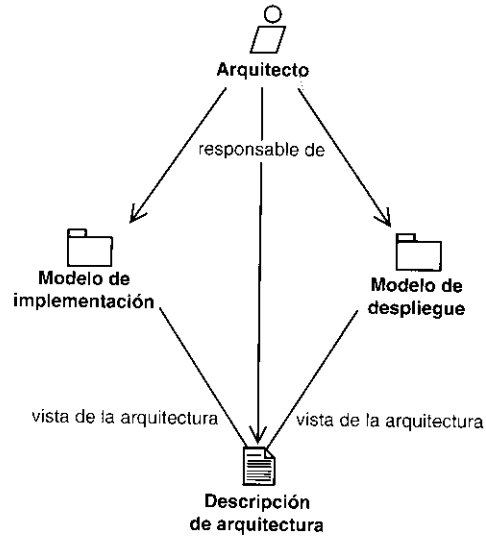


Figura 10.13. Las responsabilidades del arquitecto en la implementación.

Por último, un resultado importante de la implementación es la asignación de componentes ejecutables a nodos. El arquitecto es responsable de esta asignación, la cual se representa en la vista de la arquitectura del modelo de despliegue. *Véase* detalles en el Capítulo 9, Sección 9.3.7.

Obsérvese que el arquitecto no es responsable del desarrollo en marcha ni del mantenimiento de los diversos artefactos en el modelo de implementación; por el contrario, esto es responsabilidad del ingeniero de componentes correspondiente.

10.4.2. Trabajador: ingeniero de componentes

El ingeniero de componentes define y mantiene el código fuente de uno o varios componentes, garantizando que cada componente implementa la funcionalidad correcta (por ejemplo, como especifican las clases de diseño).

A menudo, el ingeniero de componentes también mantiene la integridad de uno o varios subsistemas de implementación. Ya que los subsistemas de implementación siguen la traza uno a uno a los subsistemas de diseño, la mayoría de los cambios en estos subsistemas tienen lugar durante el diseño. Sin embargo, el ingeniero de componentes necesita garantizar que los contenidos (por ejemplo, los componentes) de los subsistemas de implementación son correctos, que sus dependencias con otros subsistemas o interfaces son correctas y que implementan correctamente las interfaces que proporcionan. *Véase* Figura 10.14.

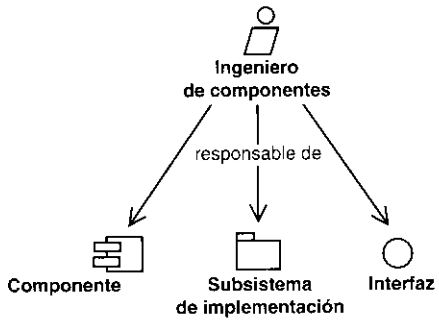


Figura 10.14. Las responsabilidades de un ingeniero de componentes en la implementación.

A menudo es apropiado también que el ingeniero de componentes que es responsable de un subsistema sea responsable de los elementos de modelo contenidos en él (por ejemplo, los componentes). Además, para alcanzar un desarrollo tranquilo y sin problemas, es natural que un ingeniero de componentes tenga la responsabilidad de un subsistema y de sus contenidos a lo largo de las etapas de diseño e implementación. Un ingeniero de componentes debería, por tanto, diseñar e implementar las clases bajo su responsabilidad.

10.4.3. Trabajador: integrador de sistemas

La integración del sistema está más allá del ámbito de cada ingeniero de componentes individual. Por el contrario, esta responsabilidad recae sobre el integrador de sistemas. Entre sus responsabilidades se incluye el planificar la secuencia de construcciones necesarias en cada iteración

y la integración de cada construcción cuando sus partes han sido implementadas. La planificación da lugar a un plan de integración de construcciones. *Véase* Figura 10.15.

Figura 10.15. Las responsabilidades de un integrador de sistemas.

10.5. Flujo de trabajo

En las secciones anteriores hemos descrito el trabajo de implementación de forma estática. A continuación utilizaremos un diagrama de actividades para razonar su comportamiento dinámico (*véase* Figura 10.16).

El objetivo principal de la implementación es implementar el sistema. Este proceso es iniciado por el arquitecto[3] esbozando los componentes clave en el modelo de implementación. A continuación, el integrador de sistemas planea las integraciones de sistema necesarias en la presente iteración como una secuencia de construcciones. Para cada construcción, el integrador de sistemas describe la funcionalidad que debería ser implementada y qué partes del modelo de implementación (es decir, subsistemas y componentes) se verán afectadas. Los requisitos sobre los subsistemas y componentes en la construcción son entonces implementados por ingenieros

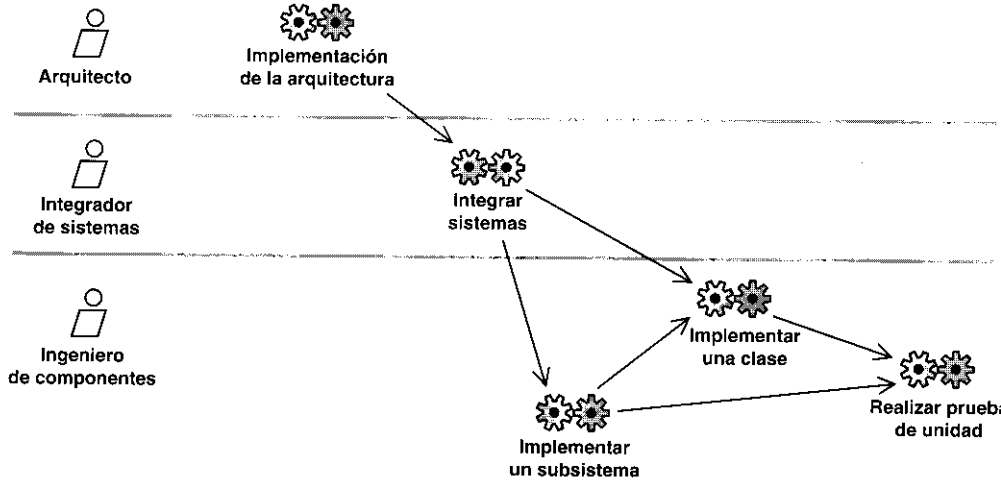


Figura 10.16. Flujo de trabajo en la etapa de implementación, incluyendo los trabajadores participantes y sus actividades.

[3] Es decir, por un número de individuos que actúan juntos como el trabajador arquitecto.

de componentes. Los componentes resultantes son probados y pasados al integrador de sistemas para su integración. Entonces, el integrador de sistemas integra los nuevos componentes en una construcción y la pasa a los ingenieros de pruebas de integración para llevar a cabo pruebas de integración (*véase* Capítulo 11). A continuación, los desarrolladores inician la implementación de la siguiente construcción, tomando en consideración los defectos de la construcción anterior.

10.5.1. Actividad: implementación de la arquitectura

El propósito de la implementación de la arquitectura es esbozar el modelo de implementación y su arquitectura mediante:

- La identificación de componentes significativos arquitectónicamente, tales como componentes ejecutables.
- La asignación de componentes a los nodos en las configuraciones de redes relevantes.

Recordemos que durante el diseño de la arquitectura (*véase* Capítulo 9, Sección 9.5.1) se esbozan los subsistemas de diseño, sus contenidos e interfaces. Durante la implementación utilizamos subsistemas de implementación que siguen la traza uno a uno a estos subsistemas de diseño y proporcionan las mismas interfaces. La identificación de los subsistemas de implementación y sus interfaces es entonces más o menos trivial y por tanto no la tratamos aquí. Por el contrario, el mayor reto durante la implementación es crear dentro de los subsistemas de implementación los componentes que implementen los subsistemas de diseño correspondientes.

Durante esta actividad, el arquitecto mantiene, refina y actualiza la descripción de la arquitectura y las vistas de la arquitectura de los modelos de implementación y despliegue.

10.5.1.1. Identificación de los componentes significativos arquitectónicamente

A menudo resulta práctico identificar pronto en el ciclo de vida del software los componentes significativos arquitectónicamente, para iniciar así el trabajo de implementación (*véase* Figura 10.17). Sin embargo, muchos componentes, en particular componentes

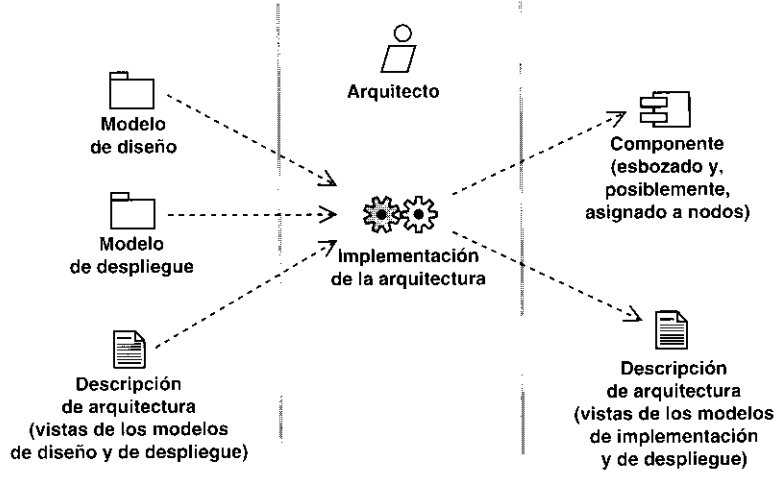


Figura 10.17. La entrada y el resultado de la implementación de la arquitectura.

de fichero, son casi triviales de crear cuando las clases son implementadas pues estos componentes no proporcionan más que una forma de empaquetar la implementación de las clases en ficheros de código fuente. Por esta razón, los desarrolladores deberían tener cuidado de no identificar demasiados componentes en esta etapa ni ahondar en demasiados detalles. De lo contrario, gran parte del trabajo habrá de volverse a hacer cuando se implementen las clases. Bastaría con un esbozo inicial de los componentes significativos arquitectónicamente [*véase* Sección 10.3.5, "Descripción de arquitectura (vista del modelo de implementación)"].

10.5.1.1.1. Identificación de componentes ejecutables y asignación de éstos a nodos Para identificar los componentes ejecutables que puedan ser desplegados sobre los nodos, consideramos las clases activas encontradas durante el diseño y asignamos un componente ejecutable por cada clase activa, denotando así un proceso pesado. Esto podría incluir la identificación de otros componentes de fichero o de código binario necesarios para crear los componentes ejecutables, aunque esto es secundario.

Ejemplo Identificación de componentes ejecutables

En el modelo de diseño hay una clase activa denominada Procesamiento de Solicitudes de Pago. En la implementación, identificamos el componente ejecutable correspondiente llamado ProcesamientoDeSolicitudesDePago (*véase* Figura 10.18).

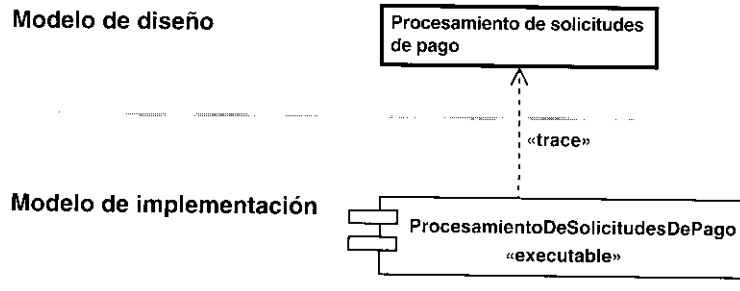


Figura 10.18. Las clases activas se usan para identificar componentes ejecutables.

Continuando con el análisis de los modelos de diseño y despliegue para identificar componentes ejecutables, podemos examinar si hay objetos activos asignados a nodos. Si se da el caso, los componentes que siguen la traza de las clases activas correspondientes deberían ser desplegados sobre los mismos nodos.

Ejemplo Despliegue de Componentes sobre nodos

Un objeto activo de la clase Procesamiento de Solicitudes de Pago está asignada al nodo Servidor del Comprador. Ya que el componente ProcesamientoDeSolicitudesDePago implementa la clase Procesamiento de Solicitudes de Pago, éste debería también ser desplegado en el nodo Servidor del Comprador (*véase* Figura 10.19).

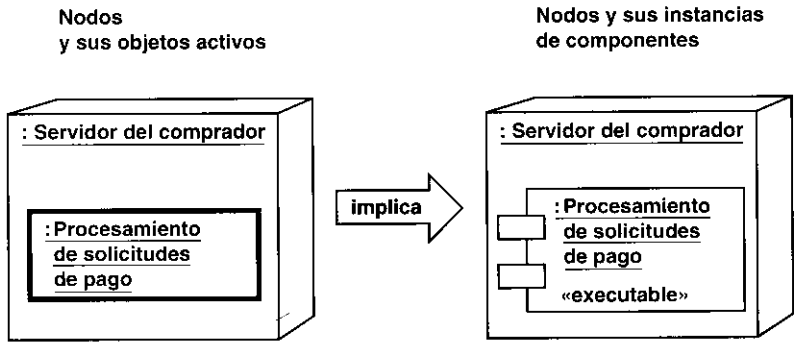


Figura 10.19. La asignación de objetos activos a nodos implica el correspondiente despliegue de componentes.

La asignación de componentes a nodos es muy importante para la arquitectura del sistema, y debería ser representada en una vista de la aruitectura del modelo de despliegue.

10.5.2. Actividad: integrar el sistema

Los objetivos de la integración del sistema son:

- Crear un plan de integración de construcciones que describa las construcciones necesarias en una iteración y los requisitos de cada construcción (*véase* Figura 10.20).
- Integrar cada construcción antes de que sea sometida a pruebas de integración.

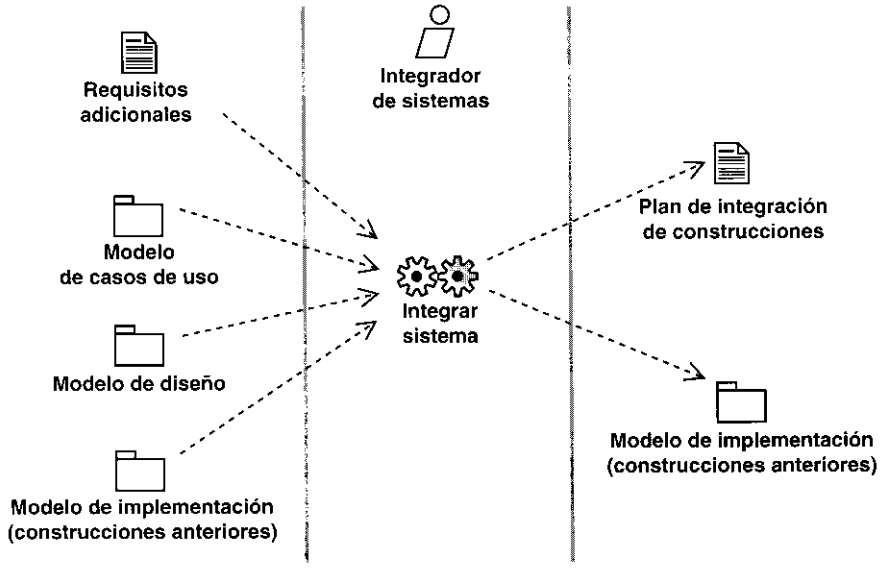


Figura 10.20. Entrada y resultado de la integración de sistemas. Las realizaciones de casos de uso-diseño en el modelo de diseño es una entrada esencial para esta actividad.

10.5.2.1. Planificación de una construcción En esta sección discutimos cómo planificar los contenidos de una construcción, independientemente de que partamos de una construcción previa o de que no partamos de ninguna. Suponemos que tenemos un número de casos de uso o escenarios (es decir, caminos a través de casos de uso) y otros requisitos que han de ser implementados en la iteración actual.

Entre los criterios para crear una construcción tenemos:

- Una construcción debería añadir funcionalidad a la construcción previa implementando casos de uso completos o escenarios de éstos. Las pruebas de integración de la construcción estarán basadas en estos casos de uso y escenarios; es más fácil probar casos de uso completos que fragmentos de éstos (*véase* Capítulo 11).
- Una construcción no debería incluir demasiados componentes nuevos o refinados. Si no es así, puede ser muy difícil integrar la construcción y llevar a cabo las pruebas de integración. Si fuera necesario algunos componentes pueden implementarse como *stubs* para minimizar el número de nuevos componentes introducidos en la construcción (*véase* Sección 10.3.2.1, "Stubs").
- Una construcción debería estar basada en la construcción anterior, y debería expandirse hacia arriba y hacia los lados en la jerarquía de subsistemas. Esto significa que la construcción inicial debería empezar en las capas inferiores (por ejemplo, en las capas intermedia y de software del sistema); las construcciones subsiguientes se expanden entonces hacia arriba a las capas general de aplicación y específica de aplicación. La razón fundamental de esto es simplemente que es difícil implementar componentes en las capas superiores antes de que estén colocados y funcionando adecuadamente los componentes necesarios en las capas inferiores.

Manteniendo estos criterios en mente, uno puede empezar a evaluar los requisitos, tales como los casos de uso (o escenarios de ellos), que han de ser implementados. Obsérvese que probablemente será necesario un compromiso para cumplir de forma apropiada estos criterios. Por ejemplo, la implementación de un caso de uso completo (punto uno) puede requerir muchos componentes nuevos (punto dos); puede hacerse sin embargo si implementar el caso de uso es importante para la construcción actual. En cualquier caso, es importante identificar los requisitos apropiados a implementar en una construcción y dejar el resto de los requisitos para construcciones futuras. Para cada caso de uso que pueda ser implementado hágase lo siguiente:

1. Considerar el diseño del caso de uso, identificando su realización de caso de uso-diseño correspondiente en el modelo de diseño. Recuérdese que puede seguirse la traza de la realización de caso de uso-diseño al caso de uso mismo (y por tanto, implícitamente, a cualquiera de sus escenarios) por medio de las dependencias de traza.
2. Identificar los subsistemas y clases de diseño que participan en la realización de caso de uso-diseño.
3. Identificar los subsistemas y componentes de implementación en el modelo de implementación que siguen la traza de los subsistemas y clases de diseño encontrados en el paso 2. Éstos son los subsistemas y componentes de implementación que son necesarios para implementar el caso de uso.
4. Considerar el impacto de implementar los requisitos de estos subsistemas de implementación y de los componentes sobre la construcción en cuestión. Obsérvese que estos requisitos se especifican como subsistemas y clases de diseño, como se hizo notar en

el paso 3. Evaluar si este impacto es aceptable de acuerdo con el criterio descrito anteriormente. Si lo es, planifíquese el implementar el caso de uso en la construcción. En otro caso, déjese para una construcción posterior.

Los resultados deberían estar recogidos en el plan de integración de la construcción y ser comunicados a los ingenieros de componentes responsables de los subsistemas y componentes de implementación afectados. Los ingenieros de componentes pueden entonces empezar a implementar los requisitos de los subsistemas y componentes de implementación en la construcción actual (como se describe en las Secciones 10.5.3, "Actividad: implementar un subsistema", y 10.5.4, "Actividad: implementar una clase") y llevar a cabo las pruebas individuales de cada unidad (como se describe en la Sección 10.5.5, "Actividad: realizar prueba de unidad"). A continuación, los subsistemas de implementación y los componentes se pasan al integrador de sistemas para su integración (*véase* Sección 10.5.2.2, "Integración de una construcción").

10.5.2.2. Integración de una construcción Si la construcción ha sido planificada cuidadosamente como se describe en la sección anterior debería ser fácil integrar la construcción. Esto se hace recopilando las versiones correctas de los subsistemas de implementación y de los componentes, compilándolos y enlazándolos para generar una construcción. Obsérvese que puede ser necesario que la compilación se realice de abajo a arriba en la jerarquía de capas, ya que pueden existir dependencias de compilación de unas capas a otras en un nivel inferior.

La construcción resultante se somete entonces a pruebas de integración, y a pruebas de sistema si pasa las pruebas de integración y es la última construcción dentro de una iteración (*véase* Capítulo 11).

10.5.3. Actividad: implementar un subsistema

El propósito de implementar un subsistema es el de asegurar que un subsistema cumple su papel en cada construcción, tal y como se especifica en el plan de integración de la construcción. Esto quiere decir que se asegura que los requisitos (por ejemplo, escenarios o casos de uso) implementados en la construcción y aquéllos que afectan al subsistema son implementados correctamente por componentes o por otros subsistemas (recursivamente) dentro del subsistema (*véase* Figura 10.21).

10.5.3.1. Mantenimiento de los contenidos de los subsistemas Un subsistema cumple su propósito cuando los requisitos a ser implementados en la construcción actual y aquéllos que afectan al subsistema están implementados correctamente por componentes dentro del subsistema.

Incluso si el contenido de un subsistema (por ejemplo, sus componentes) es esbozado por los arquitectos, puede que éste necesite ser refinado por el ingeniero de componentes conforme evoluciona el modelo de implementación. Este refinamiento puede incluir:

- Cada clase en el correspondiente subsistema de diseño que es necesaria en la construcción actual debería ser implementada mediante componentes en el subsistema de implementación (*véase* Figura 10.22).

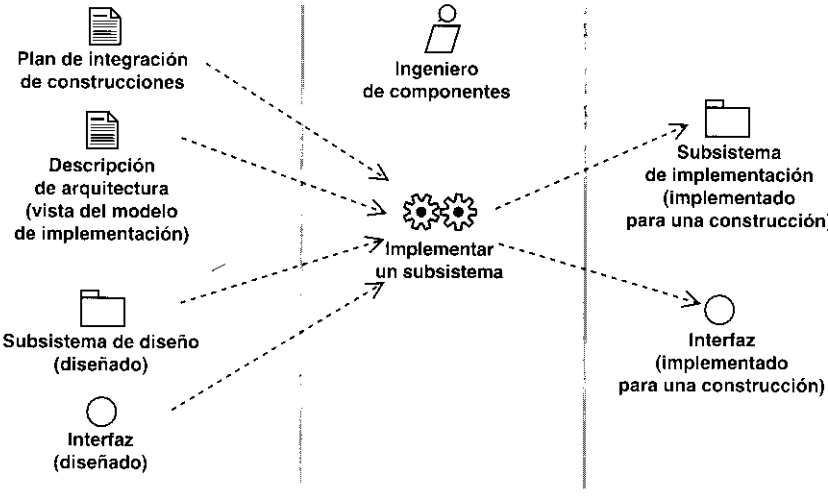


Figura 10.21. La entrada y el resultado de la implementación de subsistemas.

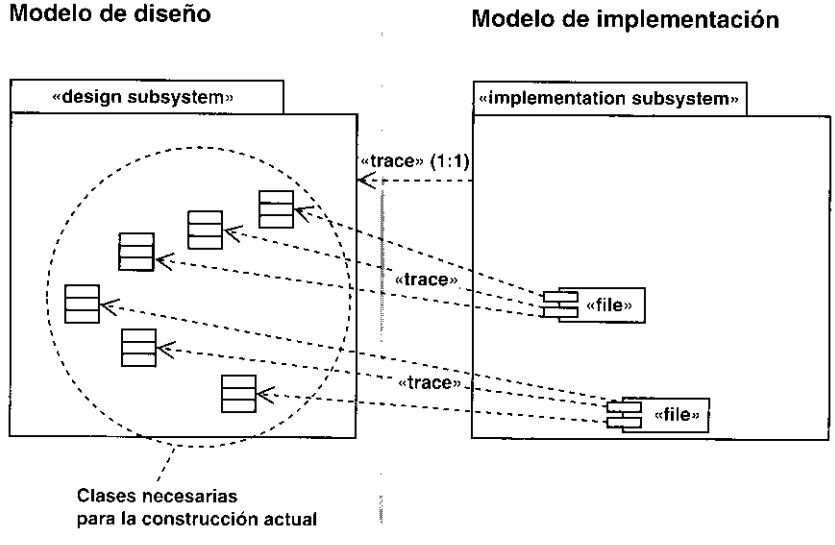


Figura 10.22. Las clases de diseño necesarias en una construcción se implementan con componentes.

Obsérvese que si el subsistema de diseño contiene otros subsistemas de diseño (recursivamente) requeridos por la construcción actual, el método es análogo: cada uno de estos subsistemas de diseño contenidos debería ser implementado por un subsistema de implementación correspondiente que es contenido a su vez por el subsistema de implementación bajo consideración.

- Cada interfaz proporcionada por el subsistema de diseño correspondiente requerido en la construcción actual debería también ser proporcionada por el subsistema de implementación. Por tanto, el subsistema de implementación ha de contener un componente o un subsistema de implementación (recursivamente) que proporcione la interfaz (*véase* Figura 10.23).

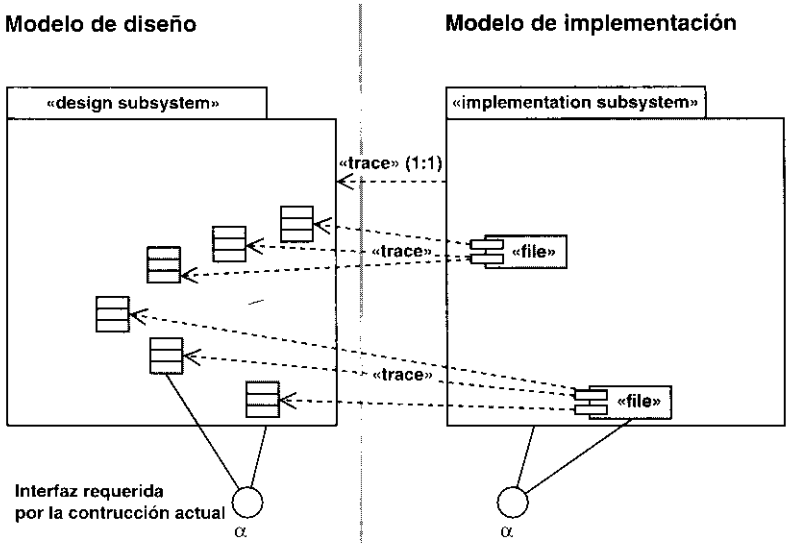


Figura 10.23. Una interfaz requerida en una construcción (α) debería ser proporcionada también por un subsistema de implementación.

Ejemplo Un subsistema que proporciona una interfaz

El subsistema Gestión de Facturas de Comprador necesita proporcionar la interfaz Factura en la construcción actual. El ingeniero de componentes responsable del subsistema decide entonces dejar que el componente Procesamiento de Facturas implemente dicha interfaz (*véase* Figura 10.24).

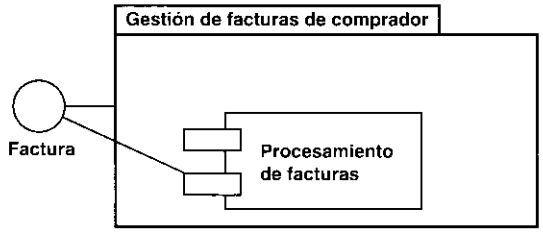


Figura 10.24. El componente Procesamiento de Facturas proporciona la interfaz Facturas.

Dada esta situación, los ingenieros de componentes pueden empezar a implementar lo que es requerido por los componentes dentro del subsistema (como se describe en la Sección 10.5.5, "Actividad: realizar prueba de unidad"). El sistema resultante se pasa entonces al integrador de sistemas para su integración (como se describe en la Sección 10.5.2, "Actividad: integrar el sistema").

10.5.4. Actividad: implementar una clase

El propósito de la implementación de una clase es implementar una clase de diseño en un componente fichero. Esto incluye lo siguiente (*véase* Figura 10.25):

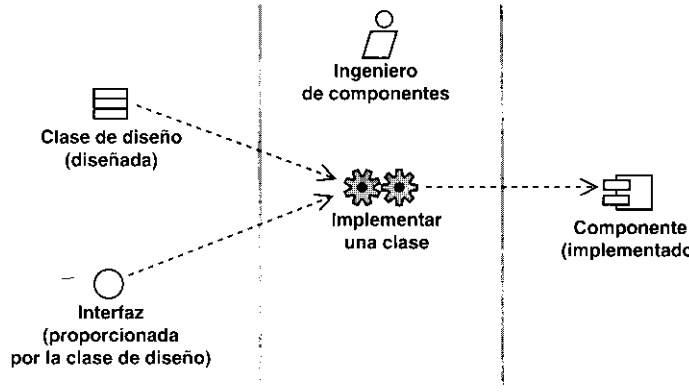


Figura 10.25. La entrada y el resultado de la implementación de una clase.

- Esbozo de un componente fichero que contendrá el código fuente.
- Generación de código fuente a partir de la clase de diseño y de las relaciones en que participa.
- Implementación de las operaciones de la clase de diseño en forma de métodos.
- Comprobación de que el componente proporciona las mismas interfaces que la clase de diseño.

Aunque no lo describimos aquí, esta actividad también incluye varios aspectos del mantenimiento de las clases implementadas, como la reparación de defectos detectados durante la prueba de la clase.

10.5.4.1. Esbozo de los componentes fichero El código fuente que implementa una clase de diseño reside en un componente fichero. Por tanto, hemos de esbozar el componente fichero y considerar su ámbito. Es normal implementar varias clases de diseño en un mismo componente fichero. Recordemos, sin embargo, que el tipo de modula-rización y las convenciones de los lenguajes de programación en uso restringirán la forma en que los componentes fichero son esbozados (*véase* Sección 10.3.2, "Artefacto: componente"). Por ejemplo, cuando usamos Java, creamos un componente fichero ".java" para cada clase de implementación. En general, los componentes fichero elegidos deberían facilitar la compilación, instalación y mantenimiento del sistema.

10.5.4.2. Generación de código a partir de una clase de diseño A lo largo del diseño, muchos de los detalles relacionados con la clase de diseño y con sus relaciones son descritos utilizando la sintaxis del lenguaje de programación elegido, lo que hace que la generación de partes del código fuente que implementan la clase sea muy fácil. En particular, esto se cumple para las operaciones y atributos de la clase, así como para las relaciones en las que la clase participa. Sin embargo, sólo se genera la signatura de las operaciones; las operaciones en sí han de ser implementadas aún (*véase* Sección 10.5.4.3, "Implementación de operaciones").

Obsérvese también que puede ser muy delicado generar código a partir de asociaciones y agregaciones, y la forma en que esto se haga depende en gran medida del lenguaje de programación utilizado. Por ejemplo, es normal que una asociación que se puede recorrer en una dirección sea implementada con una "referencia" entre ambos objetos; esta referencia sería representada como un atributo en el objeto que referencia, y el nombre del atributo sería el nombre del rol del ex-

tremo opuesto de la asociación. La multiplicidad del extremo opuesto de la asociación indicaría, a su vez, si el tipo del atributo (de la referencia) debería ser un puntero[4] simple (si la multiplicidad es menor o igual que uno) o una colección de punteros (si la multiplicidad es mayor que uno) [6].

10.5.4.3. Implementación de operaciones Cada operación definida por la clase de diseño ha de ser implementada, a no ser que sea "virtual" (o "abstracta") y ésta sea implementada por descendientes (como subtipos) de la clase. Utilizamos el término *métodos* para denotar la implementación de operaciones [5]. Ejemplos de métodos en componentes fichero reales son los *métodos* en Java [2], los *métodos* en Visual Basic [3] y las *funciones miembro* en C++ [4].

La implementación de una operación incluye la elección de un algoritmo y unas estructuras de datos apropiadas, y la codificación de las acciones requeridas por el algoritmo. Recordemos que el método puede haber sido especificado utilizando lenguaje natural o utilizando pseudocódigo durante el diseño de las clases de diseño (aunque esto es poco común y a menudo una pérdida de tiempo; *véase* Sección 9.5.3.6); por supuesto podría utilizarse cualquier "método de diseño" como entrada en esta etapa. Obsérvese también que los estados descritos para la clase de diseño pueden influenciar el modo en que son implementadas las operaciones, ya que sus estados determinan su comportamiento cuando ésta recibe un mensaje.

10.5.4.4. Los componentes han de proporcionar las interfaces apropiadas El componente resultante debería proporcionar las mismas interfaces que las clases de diseño que éste implementa. *Véase* ejemplo en la Sección 10.3.2, "Artefacto: componente".

10.5.5. Actividad: realizar prueba de unidad

El propósito de realizar la prueba de unidad es probar los componentes implementados como unidades individuales (*véase* Figura 10.26). Se llevan a cabo los siguientes tipos de pruebas de unidad:

- La *prueba de especificación,* o "prueba de caja negra", que verifica el comportamiento de la unidad observable externamente.
- La *prueba de estructura,* o "prueba de caja blanca", que verifica la implementación interna de la unidad.

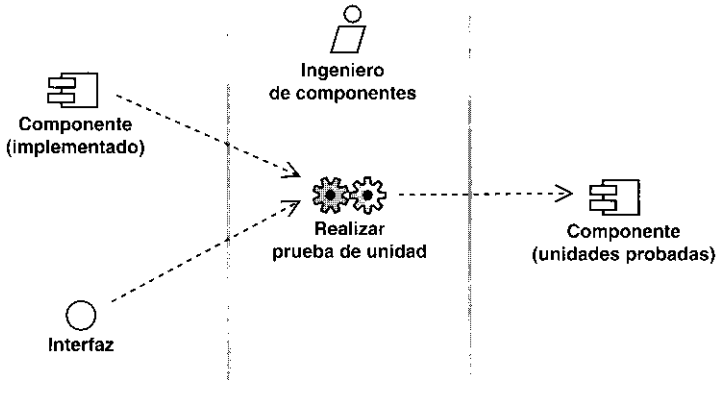


Figura 10.26. La entrada y el resultado de la prueba de una unidad.

[4] El término inglés "pointer" también se traduce frecuentemente por apuntador. *N. del RT.*

Obsérvese que puede haber también otras pruebas que realizar sobre las unidades, como pruebas de rendimiento, utilización de memoria, carga y capacidad. Además, han de ser realizadas las pruebas de integración y sistema para asegurar que los diversos componentes se comportan correctamente cuando se integran (*véase* Capítulo 11).

10.5.5.1. Realización de pruebas de especificación La prueba de especificación se realiza para verificar el comportamiento del componente sin tener en cuenta *cómo* se implementa dicho comportamiento en el componente. Las pruebas de especificación, por tanto, tienen en cuenta la salida que el componente devolverá cuando se le da una determinada entrada en un determinado estado. El número de combinaciones de posibles entradas, estados iniciales y salidas es a menudo considerable, lo que hace impracticable probar todas las combinaciones una a una. En su lugar, el número de entradas, salidas y estados se divide en *clases de equivalencia*. Una clase de equivalencia es un conjunto de valores de entrada, estado o salida para los que se supone que un objeto se comporta de forma similar. Probando un componente para cada combinación de las clases de equivalencia de entradas, salidas y estados iniciales es posible obtener casi la misma cobertura "efectiva" de prueba que si se prueban individualmente cada una de las combinaciones de valores, pero con un esfuerzo considerablemente menor.

Ejemplo **Clases de equivalencia**

El estado de una cuenta tiene tres clases de equivalencia: *vacía, saldo negativo* (quizás en descubierto) y *saldo positivo*. De forma similar, los argumentos de entrada pueden dividirse en dos clases de equivalencia: *cero* y *números positivos*. Por último, los valores de salida pueden caer dentro de dos clases de equivalencia: *cantidad positiva retirada* y *nada retirado*.

El ingeniero de componentes debería escoger valores para las pruebas utilizando heurísticas:

- Valores normales en el rango permitido para cada clase de equivalencia, como retirar 3, 3.14 o 5.923 euros de una cuenta.
- Valores que están en el límite de las clases de equivalencia, como retirar 0, el menor valor positivo posible (por ejemplo, 0,00000001) y el mayor valor positivo posible.
- Valores fuera de las clases de equivalencia permitidas, como retirar un valor mayor o menor que los permitidos en el rango.
- Valores ilegales, como retirar –14 y *A*.

Cuando se eligen los valores de prueba el ingeniero de componentes debería intentar cubrir todas las combinaciones de clases de equivalencia de entrada, estado y salida. Por ejemplo, retirar 14 euros de

- Una cuenta con –234,13 euros, lo que resulta en que nada es retirado.
- Una cuenta con 0 euros, lo que resulta en que nada es retirado.
- Una cuenta con 13,125 euros, lo que resulta en que nada es retirado.
- Una cuenta con 15 euros, lo que resulta en que se retiran 14 euros.

El resultado neto de estos cuatro casos de prueba es que todas las combinaciones posibles de clases de equivalencia de estados (*saldo positivo* y *saldo negativo*) y salidas (*cantidad positiva retirada* y *nada retirado*) son probadas para un valor en una clase de equivalencia. El ingeniero de componentes debería entonces elegir probar casos con el mismo valor para el estado (quizás –234,13, 0 o 15 euros) y la salida (0 y 14 euros), pero con otro valor de la misma clase de equivalencia de la entrada, como 3,14.

A continuación el ingeniero de componentes prepara rangos similares para probar los casos de otras clases de equivalencia para los valores de entrada. Por ejemplo, intentar retirar 0, 4, 3,14, 5.923, 0,00000001, 37.000.000.000.000.000.000.000 (si éste es el mayor número posible), 37.000. 000.000.000.000.000.001, –14 y *A* euros.

10.5.5.2. Realización de pruebas de estructura Las pruebas de estructura se realizan para verificar que un componente funciona internamente como se quería. El ingeniero de componentes debería asegurarse de probar todo el código durante las pruebas de estructura, lo que quiere decir que cada sentencia ha de ser ejecutada al menos una vez. El ingeniero de componentes debería también asegurarse de probar los caminos más importantes en el código. Entre estos caminos tenemos los que son seguidos más comúnmente, los caminos más críticos, los caminos menos conocidos de los algoritmos y otros caminos asociados con riesgos altos.

Ejemplo Código fuente Java para un método

La Figura 10.27 muestra una implementación (simplificada) del método *retirar* definido por la clase Cuenta:

```
public class Cuenta {
1    // En este ejemplo la clase Cuenta tiene sólo un saldo
2    private Dinero saldo = new Dinero (0);
3    public Dinero retirar(Dinero cantidad) {
4                    // En primer lugar tenemos que asegurar que el saldo
5                    // es al menos tan grande como la cantidad a retirar
6                    if saldo >= cantidad
7                    // Entonces comprobamos que no retiraremos una cantidad negativa
8                            then { if amount >= 0
9                                    then {
10                                           try {
11                                                   saldo = saldo – cantidad;
12                                                   return cantidad
13                                                   }
14                                           catch (Exception exc) {
15                                           // Trata fallos en la reducción del saldo
16                                           // ... ha de ser definida ...
17                                   }
18                           }
19                           else {return 0}
20                   }
21                   else {return 0}
22           }
  }
```

Figura 10.27. Código fuente Java para un método *retirar* simple definido por la clase Cuenta.

Al probar este código tenemos que asegurarnos de que todas las instrucciones *if* se evalúan a *true* y a *false* y que todo el código es ejecutado. Por ejemplo, podemos probar lo siguiente:

- Retirar 50 euros de una Cuenta con saldo de 100 euros, con lo que el sistema ejecutará las líneas 10-13.
- Retirar –50 euros de una Cuenta con saldo de 10 euros, con lo que el sistema ejecutará la línea 21.
- Retirar 50 euros de una Cuenta con saldo de 10 euros, con lo que el sistema ejecutará la línea 19.
- Disparar una excepción cuando el sistema ejecuta la sentencia saldo = saldo – cantidad, con lo que el sistema ejecutará las líneas 14-17.

10.6. Resumen de la implementación

El resultado principal de la implementación es el modelo de implementación, el cual incluye los siguientes elementos:

- Subsistemas de implementación y sus dependencias, interfaces y contenidos.
- Componentes, incluyendo componentes fichero y ejecutables, y las dependencias entre ellos. Los componentes son sometidos a pruebas de unidad.
- La vista de la arquitectura del modelo de implementación, incluyendo sus elementos arquitectónicamente significativos.

La implementación también produce como resultado un refinamiento de la vista de la arquitectura del modelo de despliegue, donde los componentes ejecutables son asignados a nodos.

Como presentaremos en el capítulo siguiente, el modelo de implementación es la entrada principal de las etapas de prueba que siguen a la implementación. Más concretamente, durante la etapa de prueba cada construcción generada durante la implementación es sometida a pruebas de integración, y posiblemente también a pruebas de sistema.

10.7. Referencias

[1] IEEE, Std 610.12-1990.

[2] Ken Arnold and James Gosling, *The Java™ Programming Language*, Reading, MA: Addison-Wesley, 1996.

[3] Anthony T. Mann, *Visual Basic5—Developer's Guide*, Indianapolis, IN: SAMS Publishing, 1997.

[4] Bjarne Stroustrup, *The C++ Programming Language*, Third Edition, Reading, MA: Addison-Wesley, 1997.

[5] The Unified Modeling Language for Object-Oriented Development, Documentation set, ver. 1.1, Rational Software Corp., September 1997.

[6] James Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.

Capítulo 11

Prueba

11.1. Introducción

En el flujo de trabajo de la prueba verificamos el resultado de la implementación probando cada construcción, incluyendo tanto construcciones internas como intermedias, así como las versiones finales del sistema a ser entregadas a terceros. En [2] puede encontrarse una buena visión general de la prueba. Más concretamente, los objetivos de la prueba son:

- Planificar las pruebas necesarias en cada iteración, incluyendo las pruebas de integración y las pruebas de sistema. Las pruebas de integración son necesarias para cada construcción dentro de la iteración, mientras que las pruebas de sistema son necesarias sólo al final de la iteración.
- Diseñar e implementar las pruebas creando los casos de prueba que especifican qué probar, creando los procedimientos de prueba que especifican cómo realizar las pruebas y creando, si es posible, componentes de prueba ejecutables para automatizar las pruebas.
- Realizar las diferentes pruebas y manejar los resultados de cada prueba sistemáticamente. Las construcciones en las que se detectan defectos son probadas de nuevo y posiblemente devueltas a otro flujo de trabajo, como diseño o implementación, de forma que los defectos importantes puedan ser arreglados.

En este capítulo presentaremos cómo se realizan las pruebas y qué trabajadores y artefactos están involucrados (*véase* Figura 11.1). El enfoque que utilizamos para este flujo de trabajo es similar al que utilizamos para la implementación.

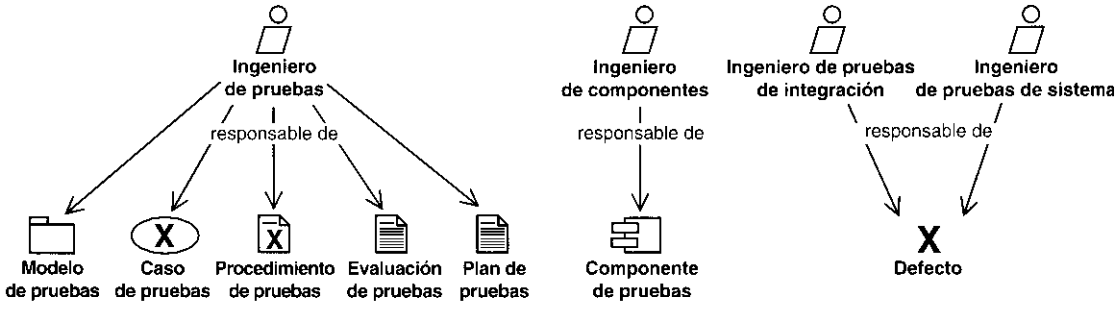


Figura 11.1. Los trabajadores y artefactos involucrados en las pruebas.

11.2. El papel de la prueba en el ciclo de vida del software

Durante la fase de inicio puede hacerse parte de la planificación inicial de las pruebas cuando se define el ámbito del sistema. Sin embargo, las pruebas se llevan a cabo sobre todo cuando una construcción (como un resultado de implementación) es sometida a pruebas de integración y de sistema. Esto quiere decir que la realización de pruebas se centra en las fases de elaboración, cuando se prueba la línea base ejecutable de la arquitectura, y de construcción, cuando el grueso del sistema está implementado. Durante la fase de transición el centro se desplaza hacia la corrección de defectos durante los primeros usos y a las pruebas de regresión. *Véase* Figura 11.2.

Debido a la naturaleza iterativa del esfuerzo de desarrollo, algunos de los casos de prueba que especifican cómo probar las primeras construcciones pueden ser utilizadas también como casos de prueba de regresión que especifican cómo llevar a cabo las pruebas de regresión sobre las construcciones siguientes. El número de pruebas de regresión necesarias crece por tanto de

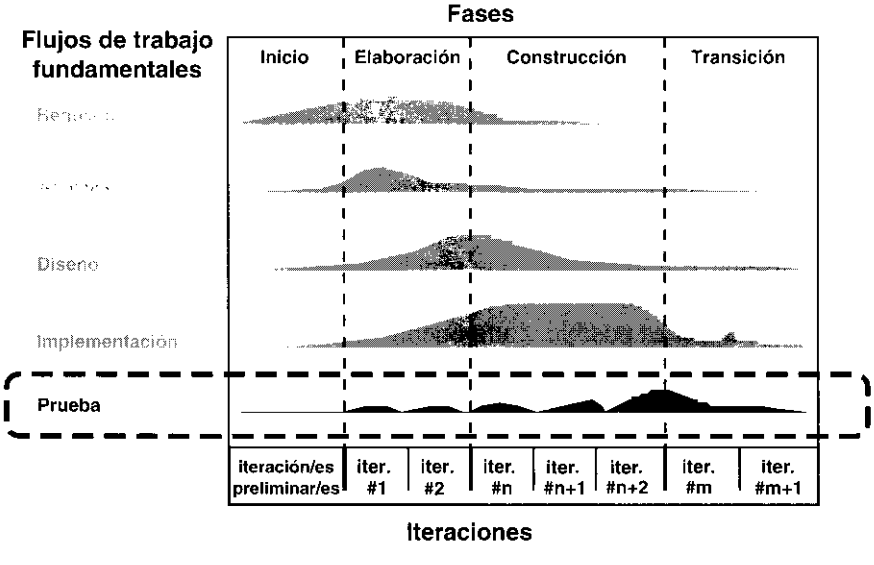


Figura 11.2. Las pruebas.

forma estable a lo largo de las iteraciones, lo que significa que las últimas iteraciones requerirán un gran esfuerzo en pruebas de regresión. Es natural, por tanto, mantener el modelo de pruebas a lo largo del ciclo de vida del software completo, aunque el modelo de pruebas cambia constantemente debido a:

- La eliminación de casos de prueba obsoletos (y los correspondientes procedimientos de prueba y componentes de prueba).
- El refinamiento de algunos casos de prueba en casos de prueba de regresión.
- La creación de nuevos casos de uso para cada nueva construcción.

11.3. Artefactos

11.3.1. Artefacto: modelo de pruebas

El modelo de pruebas describe principalmente cómo se prueban los componentes ejecutables (como las construcciones) en el modelo de implementación con pruebas de integración y de sistema. El modelo de pruebas puede describir también cómo han de ser probados aspectos específicos del sistema; por ejemplo, si la interfaz de usuario es utilizable y consistente o si el manual de usuario del sistema cumple con su cometido. Como se ilustra en la Figura 11.3, el modelo de pruebas es una colección de casos de prueba, procedimientos de prueba y componentes de prueba.

Las siguientes secciones presentan los artefactos en el modelo de pruebas en detalle. Obsérvese que si el modelo de pruebas es grande, es decir, si contiene una gran cantidad de casos de prueba, procedimientos de prueba y componentes de prueba, puede ser útil introducir paquetes en el modelo para manejar su tamaño. Ésta es una extensión más o menos trivial del modelo de pruebas y no lo tratamos en este capítulo.

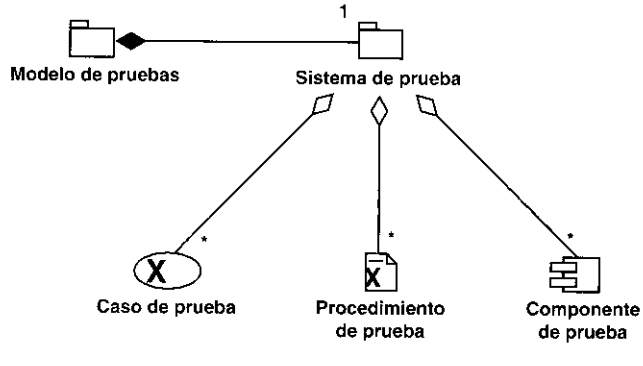


Figura 11.3. El modelo de pruebas.

11.3.2. Artefacto: caso de prueba

Un caso de prueba especifica una forma de probar el sistema, incluyendo la entrada o resultado con la que se ha de probar y las condiciones bajo las que ha de probarse (*véase* Figura 11.4).

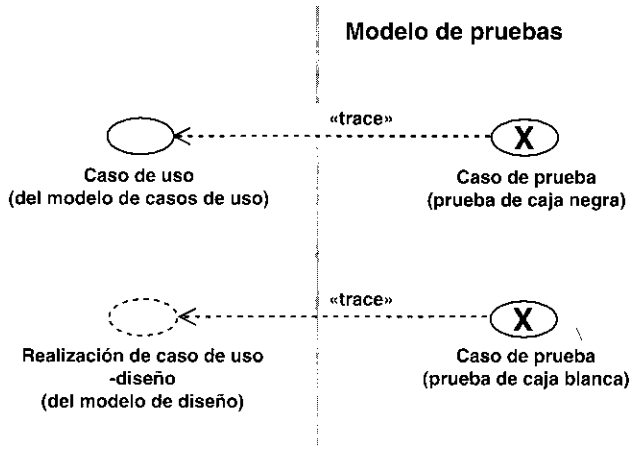


Figura 11.4. Un caso de prueba puede derivarse de, y por tanto puede seguir la traza de, un caso de uso en el modelo de casos de uso o de una realización de caso de uso en el modelo de diseño.

En la práctica, lo que se prueba puede venir dado por un requisito o colección de requisitos del sistema cuya implementación justifica una prueba que es posible realizar y que no es demasiado cara de realizar. Los siguientes son casos de prueba comunes:

- Un caso de prueba que especifica cómo probar un caso de uso o un escenario específico de un caso de uso. Un caso de prueba de este tipo incluye la verificación del resultado de la interacción entre los actores y el sistema, que se satisfacen las precondiciones y postcondiciones especificadas por el caso de uso y que se sigue la secuencia de acciones especificadas por el caso de uso. Obsérvese que un caso de prueba basado en un caso de uso especifica típicamente una prueba del sistema como "caja negra", es decir, una prueba del comportamiento observable externamente del sistema.
- Un caso de prueba que especifica cómo probar una realización de caso de uso-diseño o un escenario específico de la realización. Un caso de prueba de este tipo puede incluir la verificación de la interacción entre los componentes que implementan dicho caso de uso. Obsérvese que los casos de prueba basados en una realización de caso de uso típicamente especifican una prueba del sistema como "caja blanca", es decir, una prueba de la interacción interna entre los componentes del sistema.

Ejemplo Caso de prueba

Los ingenieros de pruebas sugieren un conjunto de casos de prueba para probar el caso de uso Pagar Factura en el que cada caso de prueba verificará un escenario del caso de uso. Uno de los casos de prueba propuestos es el pago de una factura de 300 euros por un pedido de una bicicleta de montaña. Los ingenieros denominan a este caso de uso Pagar 300-Bicicleta de Montaña.

Para ser completos, el caso de prueba ha de especificar la entrada, el resultado esperado y otras condiciones relevantes para la verificación del escenario del caso de uso.

Entrada

- Existe un pedido válido de una bicicleta de montaña y éste ha sido enviado a un vendedor, Crazy Mountaineer, Inc. El precio de la bicicleta es 300 euros, incluyendo gastos de envío.

- El comprador ha recibido una confirmación de pedido (ID 98765) de una bicicleta de montaña. El precio confirmado de la bicicleta es de 300 euros, incluyendo gastos de envío.
- El comprador ha recibido una factura (ID 12345), la cual se conforma con la confirmación del pedido de la bicicleta de montaña. Ésta es la única factura presente en el sistema. El total facturado sería por un total de 300 euros, y la factura debería estar en el estado Pendiente. La factura debería apuntar a una cuenta 22-222-2222, la cual debería recibir el dinero. Esta cuenta tiene un saldo de 963.456,00 euros, y su titular debería ser el vendedor.
- La cuenta 11-111-1111 del comprador tiene un saldo de 350 euros.

Resultado

- El estado de la factura debería ser puesto a Cerrada, indicando así que ha sido pagada.
- La cuenta 11-111-1111 del comprador debería tener un saldo de 50 euros.
- El saldo de la cuenta 22-222-2222 del vendedor debería haber aumentado hasta 963.756,00 euros.

Condiciones

- No se permite que otras instancias de casos de uso accedan a las cuentas durante el caso de prueba.

Obsérvese que algunos casos de uso pueden ser parecidos y diferenciarse únicamente en un solo valor de entrada o resultado. Esto se da a veces para casos de prueba que verifican diferentes escenarios del mismo caso de uso. En estos casos, puede ser apropiado especificar los casos de uso en forma de tabla, donde cada caso de prueba está representado por una fila y cada rango de valores de entrada y resultado se representa con una columna. Una tabla de este tipo puede proporcionar una buena visión general de casos de prueba similares y proporcionar una entrada utilizable en la creación de procedimientos de prueba y componentes de prueba (*véase* Secciones 11.3.3 y 11.3.4).

Se pueden especificar otros casos de prueba para probar el sistema como un todo. Por ejemplo:

- Las *pruebas de instalación* verifican que el sistema puede ser instalado en la plataforma del cliente y que el sistema funcionará correctamente cuando sea instalado.
- Las *pruebas de configuración* verifican que el sistema funciona correctamente en diferentes configuraciones; por ejemplo, en diferentes configuraciones de red.
- Las *pruebas negativas* intentan provocar que el sistema falle para poder así revelar sus debilidades. Los *ingenieros de pruebas* identifican los casos de prueba que intentan utilizar el sistema en formas para los que no ha sido diseñado, por ejemplo, utilizando configuraciones de red incorrectas, capacidad de hardware insuficiente o una carga de trabajo "imposible" [1].
- Las *pruebas de tensión o de estrés* identifican problemas con el sistema cuando hay recursos insuficientes o cuando hay competencia por los recursos.

---

[1] Los casos que presentan una carga de trabajo "imposible" se llaman también "casos de abuso".

Muchos de estos casos de prueba pueden encontrarse considerando los casos de uso del sistema. Damos más detalles en la Sección 11.5.2.

11.3.3. Artefacto: procedimiento de prueba

Un *procedimiento de prueba* especifica cómo realizar uno a varios casos de prueba o partes de éstos. Por ejemplo, un procedimiento de prueba puede ser una instrucción para un individuo sobre cómo ha de realizar un caso de prueba manualmente, o puede ser una especificación de cómo interaccionar manualmente con una herramienta de automatización de pruebas para crear componentes ejecutables de prueba (*véase* Sección 11.3.4).

El cómo llevar a cabo un caso de prueba puede ser especificado por un procedimiento de prueba, pero es a menudo útil reutilizar un procedimiento de prueba para varios casos de prueba y reutilizar varios procedimientos de prueba para un caso de prueba (*véase* Figura 11.5).

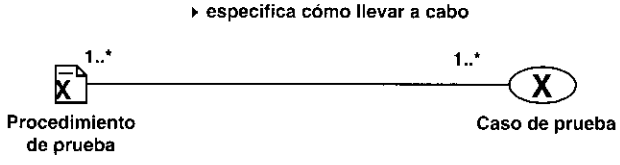


Figura 11.5. Hay asociaciones muchos-a-muchos entre los procedimientos de prueba y los casos de prueba.

Ejemplo **Procedimiento de prueba**

Se necesita un procedimiento de prueba para que un individuo lleve a cabo el caso de uso Pagar 300-Bicicleta de Montaña discutido en el ejemplo anterior (*véase* Sección 11.3.2). La primera parte del procedimiento de prueba se especifica como sigue (el texto entre corchetes no tiene que incluirse en la especificación ya que ya está especificado en el caso de uso):

Caso de uso: Pagar 300-Bicicleta de Montaña

1. Seleccione el menú Hojear Facturas de la ventana principal. Se abre la ventana de diálogo Consultar Facturas.
2. En el campo Estado de la Factura, seleccione **Pendiente** y pulse el botón **Consultar**. Aparece la ventana Resultados de la Consulta. Verifique que la factura especificada en el caso de uso [ID 12345] está en la lista en la ventana Resultados de la Consulta.
3. Seleccione la factura a pagar especificada pulsando el botón dos veces. Aparece la ventana Detalles de Factura para la factura seleccionada. Verifique los siguientes detalles:
   - el Estado es Pendiente;
   - la Fecha de Pago está vacía;
   - el Identificador de Confirmación de Pedido coincide con el identificador en el caso de uso [ID 98765];
   - la Cantidad de la Factura coincide con la especificada en el caso de prueba [300 euros]; y
   - el número de cuenta coincide con el especificado en el caso de prueba [22-222-2222].

4. Seleccione la opción Autorizar Pago para iniciar el pago de esta factura. Aparece la ventana de diálogo para el Pago de Factura.
5. Etc. (Se especifica cómo se lleva a cabo a través de la interfaz de usuario el camino completo del caso de uso Pagar Factura dando ciertas entradas al sistema, e indicando qué es necesario verificar en las salidas del sistema.)

Obsérvese que este procedimiento de prueba puede ser utilizado para otros casos de uso similares en los que se utilizan diferentes valores de entrada y resultado, es decir, los valores entre corchetes son diferentes. Obsérvese también que el procedimiento de prueba es similar a la descripción de flujo de eventos del caso de uso Pagar Factura (*véase* Sección 7.4.3.1), aunque el procedimiento de prueba incluye información adicional, como los valores de entrada del caso de uso a utilizar, la forma en la que estos valores han de ser introducidos en el interfaz de usuario y lo que hay que verificar.

Ejemplo **Procedimientos de prueba generales**

Cuando los diseñadores de pruebas sugieren procedimientos de prueba hacen notar que varios casos de uso (y los casos de prueba para verificarlos) empiezan validando objetos al comienzo del caso de uso, como cuando se compara la factura recibida con la confirmación de la orden.

Los diseñadores de pruebas sugieren entonces un procedimiento de prueba general denominado Validar Objetos de Negocio para probar dichas secuencias. El procedimiento de prueba especifica que los objetos a validar deberían en primer lugar ser creados, leyendo los valores que deberían tener de un fichero y creándolos entonces en una base de datos. A continuación el procedimiento de prueba especifica que el objeto activo que valida los objetos de negocio debería ser invocado; como Gestor de Pedidos en el caso de uso Pagar Factura. Por último, el procedimiento de prueba especifica que el resultado de la validación debería ser comparado con el resultado esperado, como se describe en el fichero mencionado.

Los diseñadores de pruebas sugieren también un procedimiento de prueba denominado Verificar Planificación, el cual especifica cómo probar la planificación de una factura y después verifica que la planificación de la factura dispara una transferencia bancaria.

11.3.4. Artefacto: componente de prueba

Un componente de prueba automatiza uno o varios procedimientos de prueba o partes de ellos (*véase* Sección 10.3.2). *Véase* Figura 11.6.

Los componentes de prueba pueden ser desarrollados utilizando un lenguaje de guiones o un lenguaje de programación, o pueden ser grabados con una herramienta de automatización de pruebas.

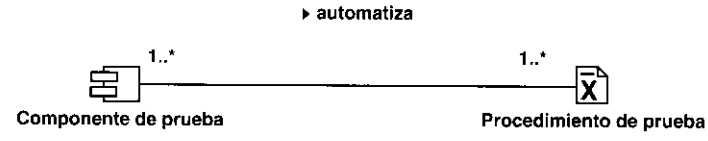


Figura 11.6. Hay asociaciones muchos-a-muchos entre los componentes de prueba y los procedimientos de prueba.

Los componentes de prueba se utilizan para probar los componentes en el modelo de implementación (*véase* Capítulo 10), proporcionando entradas de prueba, controlando y monitorizando la ejecución de los componentes a probar y, posiblemente, informando de los resultados de las pruebas. En inglés, los componentes de prueba son a veces llamados *test drivers, test harnesses* [1] y *test scripts.*

Obsérvese que los componentes de prueba pueden ser implementados usando tecnología de objetos. Si varios componentes de prueba tienen interacciones internas complejas o interacciones complejas con los componentes ordinarios en el modelo de implementación puede utilizarse un "modelo de diseño de pruebas" (similar al modelo de diseño; *véase* Sección 9.3.1) para modelar los componentes de prueba y representar vistas de alto nivel de ellos. Aunque puede ser útil, dicho modelo no es discutido en detalle en este libro.

11.3.5. Artefacto: plan de prueba

El plan de prueba describe las estrategias, recursos y planificación de la prueba. La estrategia de prueba incluye la definición del tipo de pruebas a realizar para cada iteración y sus objetivos, el nivel de cobertura de prueba y de código necesario y el porcentaje de pruebas que deberían ejecutarse con un resultado específico.

11.3.6. Artefacto: defecto

Un defecto es una anomalía del sistema, como por ejemplo un síntoma de un fallo software o un problema descubierto en una revisión. Un defecto puede ser utilizado para localizar cualquier cosa que los desarrolladores necesitan registrar como síntoma de un problema en el sistema que ellos necesitan controlar y resolver.

11.3.7. Artefacto: evaluación de prueba

Una evaluación de prueba es una evaluación de los resultados de los esfuerzos de prueba, tales como la cobertura del caso de prueba, la cobertura de código y el estado de los defectos.

11.4. Trabajadores

11.4.1. Trabajador: diseñador de pruebas

Un diseñador de pruebas es responsable de la integridad del modelo de pruebas, asegurando que el modelo cumple con su propósito. Los diseñadores de pruebas también planean las pruebas, lo que significa que deciden los objetivos de prueba apropiados y la planificación de las pruebas. Además, los diseñadores de pruebas seleccionan y describen los casos de prueba y los procedimientos de prueba correspondientes que se necesitan, y son responsables de la evaluación de las pruebas de integración y de sistema cuando éstas se ejecutan. *Véase* Figura 11.7.

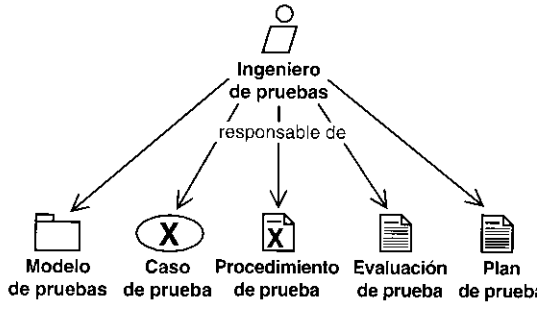


Figura 11.7. Las responsabilidades de un diseñador de pruebas en el flujo de trabajo de la prueba.

Obsérvese que los diseñadores de pruebas realmente no llevan a cabo las pruebas, sino que se dedican a la preparación y evaluación de las mismas. Otros dos tipos de trabajadores, los ingenieros de pruebas de integración y los ingenieros de pruebas de sistema, son los encargados de llevarlas a cabo.

11.4.2. Trabajador: ingeniero de componentes

Los ingenieros de componentes son responsables de los componentes de prueba que automatizan algunos de los procedimientos de prueba (no todos los procedimientos de prueba pueden ser automatizados). Esto es así porque la creación de dichos componentes puede necesitar de substanciales habilidades como programador (habilidades que también son precisas en el ingeniero de componentes en el flujo de trabajo de la implementación; *véase* Sección 10.4.2).

11.4.3. Trabajador: ingeniero de pruebas de integración

Los ingenieros de pruebas de integración son los responsables de realizar las pruebas de integración que se necesitan para cada construcción producida en el flujo de trabajo de la implementación (*véase* Sección 10.5.2). Las pruebas de integración se realizan para verificar que los componentes integrados en una construcción funcionan correctamente juntos. Por esto, las pruebas de integración se derivan a menudo de los casos de prueba que especifican cómo probar realizaciones de casos de uso-diseño (*véase* Sección 11.3.2).

El ingeniero de pruebas de integración se encarga de documentar los defectos en los resultados de las pruebas de integración.

Obsérvese que un ingeniero de pruebas de integración prueba el resultado, es decir, una construcción, creado por el integrador de sistemas en el flujo de trabajo de la implementación. Es por esto por lo que en algunos proyectos se decide que estos trabajadores sean el mismo individuo o grupo de individuos para minimizar el solapamiento en el conocimiento necesario.

11.4.4. Trabajador: ingeniero de pruebas de sistema

Un ingeniero de pruebas de sistema es responsable de realizar las pruebas de sistema necesarias sobre una construcción que muestra el resultado (ejecutable) de una iteración completa (*véase*

Sección 10.5.2). Las pruebas de sistema se llevan a cabo principalmente para verificar las interacciones entre los actores y el sistema. Por esto, las pruebas de sistema se derivan a menudo de los casos de prueba que especifican cómo probar los casos de uso, aunque también se aplican otros tipos de pruebas al sistema como un todo (*véase* Sección 11.3.2).

El ingeniero de pruebas de sistema se encarga de documentar los defectos en los resultados de las pruebas de sistema.

Debido a la naturaleza de las pruebas de sistema, los individuos que actúan como ingenieros de pruebas de sistema necesitan saber mucho sobre el funcionamiento interno del sistema. Por el contrario, éstos deberían tener familiaridad con el comportamiento observable externamente del sistema. Por tanto, algunas pruebas de sistema pueden ser realizadas por otros miembros del proyecto, como los especificadores de casos de uso, o incluso por personas externas al proyecto, como usuarios de versiones beta.

11.5. Flujo de trabajo

En las secciones anteriores describimos las pruebas estáticamente. Ahora utilizaremos un diagrama de actividad (Figura 11.8) para razonar su comportamiento dinámico:

El principal objetivo de la prueba es realizar y evaluar las pruebas como se describe en el modelo de pruebas. Los ingenieros de pruebas inician esta tarea planificando el esfuerzo de prueba en cada iteración, y describen entonces los casos de prueba necesarios y los procedimientos de prueba correspondientes para llevar a cabo las pruebas. Si es posible, los ingenieros de componentes crean a continuación los componentes de prueba para automatizar algunos de los procedimientos de prueba. Todo esto se hace para cada construcción entregada como resultado del flujo de trabajo de implementación.

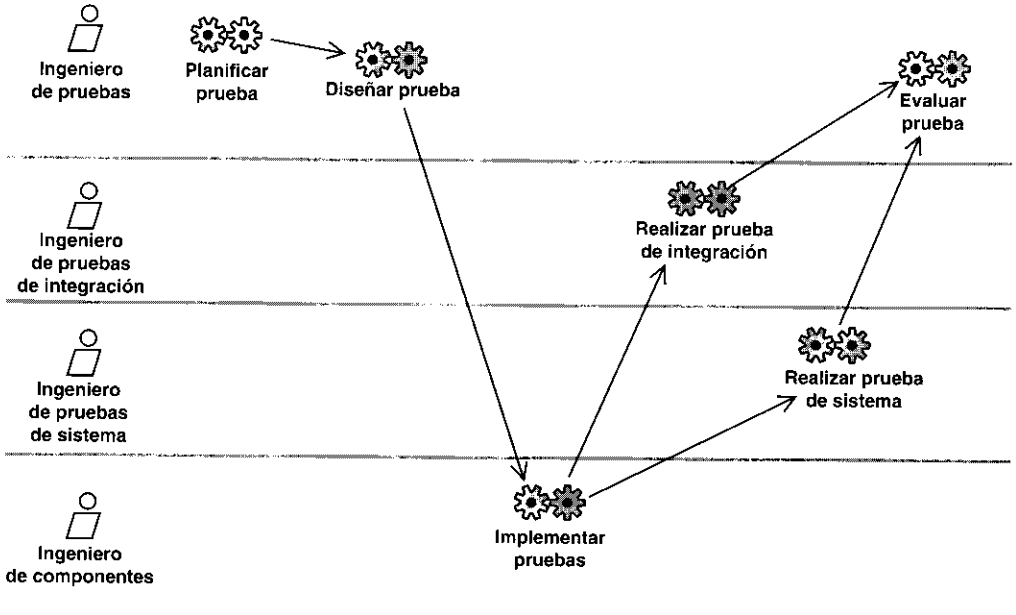


Figura 11.8. El flujo de trabajo durante la prueba, incluyendo los trabajadores participantes y sus actividades.

Con estos casos, procedimientos y componentes de prueba como entrada, los ingenieros de pruebas de integración y de sistema prueban cada construcción y detectan cualquier defecto que encuentren en ellos. Los defectos sirven como realimentación tanto para otros flujos de trabajo, como el de diseño y el de implementación, como para los ingenieros de pruebas para que lleven a cabo una evaluación sistemática de los resultados de las pruebas.

11.5.1. Actividad: planificar prueba

El propósito de la planificación de la prueba (*véase* Figura 11.9) es planificar los esfuerzos de prueba en una iteración llevando a cabo las siguientes tareas:

- Describiendo una estrategia de prueba.
- Estimando los requisitos para el esfuerzo de la prueba, por ejemplo, los recursos humanos y sistemas necesarios.
- Planificando el esfuerzo de la prueba.

Cuando prepara el plan de prueba los ingenieros de prueba se mueven sobre un rango de valores de entrada. El modelo de casos de uso y los requisitos adicionales les ayudan a decidirse por un tipo adecuado de pruebas y a estimar el esfuerzo necesario para llevar a cabo las pruebas. El diseñador de pruebas usará también otros artefactos como entrada, como por ejemplo el modelo de diseño.

Los diseñadores de pruebas desarrollan una estrategia de prueba para la iteración, es decir, deciden qué tipo de pruebas ejecutar, cómo ejecutar dichas pruebas, cuándo ejecutarlas y cómo determinar si el esfuerzo de prueba tiene éxito.

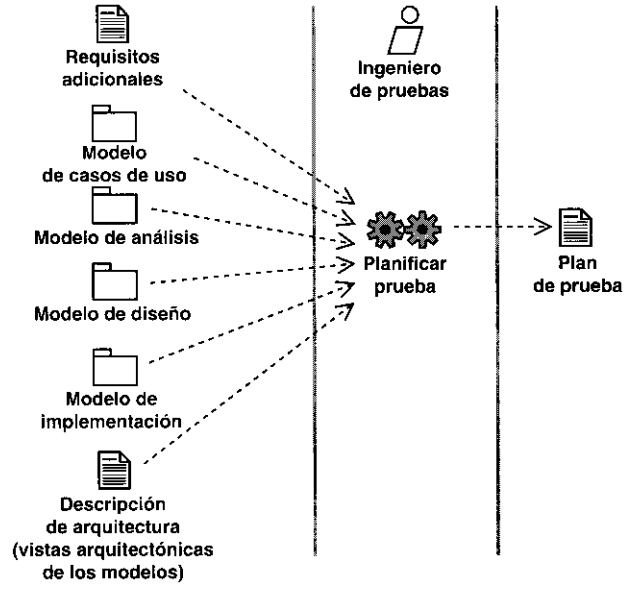


Figura 11.9. La entrada y el resultado de la planificación de la prueba.

Ejemplo **Estrategia de prueba de sistema para la última iteración en la fase de elaboración**

Al menos el 75 por ciento de las pruebas deberían estar automatizadas, y el resto debería ser manual. Cada caso de uso será probado para su flujo normal y para tres flujos alternativos.

Criterio de éxito: 90 por ciento de los casos de prueba pasados con éxito. No hay ningún defecto de prioridad media-alta sin resolver.

El desarrollo, realización y evaluación de cada caso de prueba, procedimiento de prueba y componente de prueba lleva algún tiempo y cuesta algún dinero. Sin embargo, ningún sistema puede ser probado completamente; por tanto, deberíamos identificar los casos, procedimientos y componentes de prueba con un mayor retorno a la inversión en términos de mejora de calidad [3]. El principio general consiste en desarrollar casos y procedimientos de prueba con un solapamiento mínimo para probar los casos de uso más importantes y probar los requisitos que están asociados a los riesgos más altos.

11.5.2. Actividad: diseñar prueba

Los propósitos de diseñar las pruebas (*véase* Figura 11.10) son:

- Identificar y describir los casos de prueba para cada construcción.
- Identificar y estructurar los procedimientos de prueba especificando cómo realizar los casos de prueba.

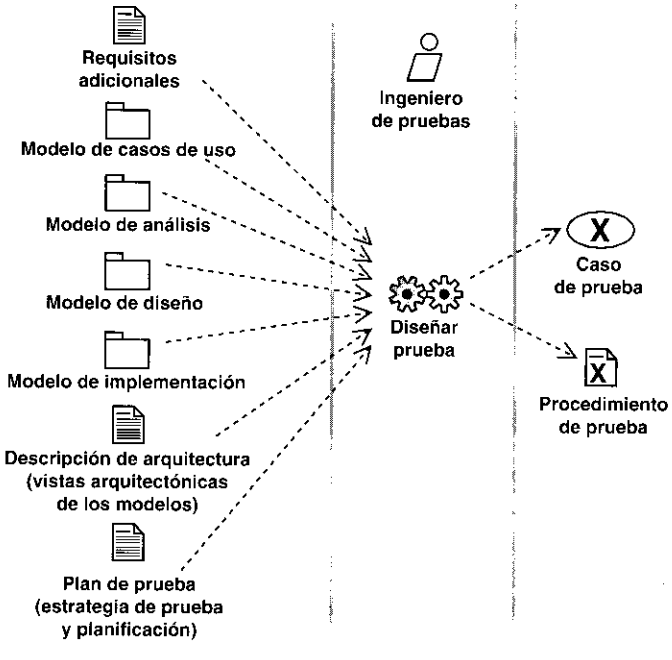


Figura 11.10. La entrada y el resultado del diseño de la prueba.

11.5.2.1. Diseño de los casos de prueba de integración Los casos de prueba de integración se utilizan para verificar que los componentes interaccionan entre sí de la forma apropiada después de haber sido integrados en una construcción (*véase* Sección 10.5.2). La mayoría de los casos de prueba de integración pueden ser derivados de las realizaciones de casos de uso-diseño, ya que las realizaciones de casos de uso describen cómo interaccionan las clases y los objetos, y por tanto cómo interaccionan los componentes.

Los ingenieros de pruebas deberían crear un conjunto de casos de prueba que hicieran posible alcanzar los objetivos establecidos en el plan de prueba con un esfuerzo mínimo. Para poder hacer esto, los diseñadores de pruebas intentan encontrar un conjunto de casos de prueba con un solapamiento mínimo, cada uno de los cuales prueba un camino o escenario interesante a través de una realización de casos de uso.

Cuando los diseñadores de pruebas crean los casos de prueba de integración consideran como entrada primeramente los diagramas de interacción de las realizaciones de casos de uso. Los diseñadores de pruebas buscan combinaciones de entrada, salida y estado inicial de sistema que den lugar a escenarios interesantes que empleen las clases (y por tanto los componentes) que participan en los diagramas.

Ejemplo Caso de prueba de integración

Los diseñadores de pruebas comienzan considerando un diagrama de secuencia que es parte de la realización de caso de uso-diseño para el caso de uso Pagar Factura. La Figura 11.11 muestra la primera parte de este diagrama (*véase* Sección 9.5.2.2).

Obsérvese que pueden haber varias "secuencias" diferentes en el diagrama de secuencias dependiendo, por ejemplo, del estado "inicial" del sistema y de la entrada del actor. Por ejemplo, si consideramos el diagrama de secuencia en la Figura 11.11 podemos observar que habría muchos mensajes que no serían enviados si no hubiera facturas en el sistema.

Un caso de pruebas que se deriva de un diagrama de secuencia como el mostrado en la Figura 11.11 debería describir cómo probar una secuencia interesante en el diagrama, tomando el estado inicial del sistema, la entrada del actor y demás circunstancias necesarias que hacen que ocurra la secuencia.

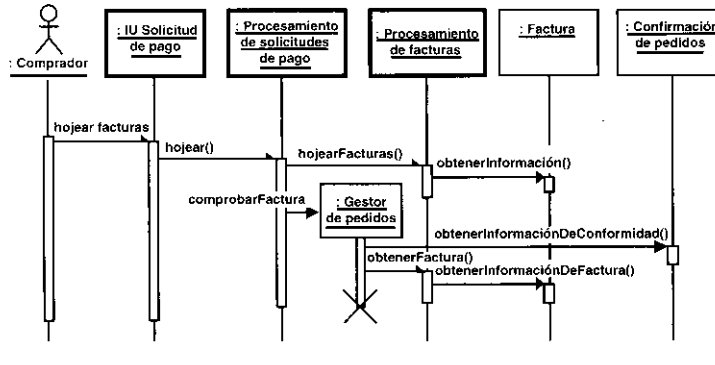


Figura 11.11. Primera parte de un diagrama de secuencia para la realización de caso de uso-diseño del caso de uso Pagar Factura.

Más tarde, cuando se realiza la prueba de integración correspondiente, tomamos las interacciones actuales de objetos en el sistema; por ejemplo, creando trazas de su ejecución o ejecutándola paso a paso. A continuación, comparamos las interacciones actuales con el diagrama de interacciones, las cuales deberían ser iguales; en otro caso se trata de un defecto.

11.5.2.2. Diseño de los casos de prueba de sistema Las pruebas de sistema se usan para probar que el sistema funciona correctamente como un todo. Cada prueba de sistema prueba principalmente combinaciones de casos de uso instanciados bajo condiciones diferentes. Estas condiciones incluyen diferentes configuraciones hardware (procesadores, memoria principal, discos duros, etc.), diferentes niveles de carga del sistema, diferentes números de actores y diferentes tamaños de la base de datos. Cuando se desarrollan los casos de prueba de sistema los diseñadores de pruebas deberían dar prioridad a las combinaciones de los casos de uso que:

- Se necesita que funcionen en paralelo.
- Posiblemente funcionan en paralelo.
- Posiblemente se influencian mutuamente si se ejecutan en paralelo.
- Involucran varios procesadores.
- Usan frecuentemente recursos del sistema, como procesos, procesadores, bases de datos y software de comunicaciones, quizás en formas complejas e impredecibles.

Pueden encontrarse, por tanto, muchos casos de prueba de sistema considerando los casos de uso, especialmente considerando sus flujos de eventos y sus requisitos especiales (como los requisitos de rendimiento).

11.5.2.3. Diseño de los casos de prueba de regresión Algunos casos de prueba de construcciones anteriores pueden ser usados para pruebas de regresión en construcciones subsiguientes, aunque no todos los casos de prueba son adecuados para pruebas de regresión. Para ser adecuados y contribuir a la calidad del sistema los casos de prueba han de ser suficientemente flexibles para ser resistentes a cambios en el software que está siendo probado. La flexibilidad requerida en un caso de prueba de regresión puede suponer un esfuerzo de desarrollo extra, lo que significa que debemos ser cuidadosos y convertirlos en casos de prueba de regresión únicamente cuando el esfuerzo merezca la pena.

11.5.2.4. Identificación y estructuración de los procedimientos de prueba Los diseñadores de pruebas pueden trabajar caso de prueba a caso de prueba y sugerir procedimientos de prueba para cada uno. Intentamos reutilizar procedimientos de prueba existentes tanto como sea posible, lo que significa que podemos necesitar modificarlos de forma que puedan usarse para especificar cómo realizar un caso de prueba nuevo o cambiado. Los diseñadores de pruebas también intentan crear procedimientos de prueba que puedan ser reutilizados en varios casos de prueba. Esto permite a los diseñadores de pruebas usar un conjunto reducido de procedimientos de prueba con rapidez y precisión para muchos casos de prueba.

Ya que los casos de prueba están basados en casos de uso o en realizaciones de casos de uso, la mayoría de los casos de prueba probarán varias clases de varios subsistemas de servicios, pero cada procedimiento de prueba debería, si fuera posible, especificar cómo probar clases de

un único subsistema de servicios; sin embargo, varios procedimientos de prueba pueden especificar cómo probar un subsistema de servicios. Cada caso de prueba precisará varios procedimientos de prueba, quizás uno por cada subsistema de servicios probado en el caso de pruebas. Relacionando de esta forma los procedimientos de prueba con los subsistemas de servicios los procedimientos de prueba serían más fáciles de mantener. Cuando se cambia un subsistema de servicios los efectos del cambio que tienen que ver con los procedimientos de prueba pueden entonces estar limitados a los procedimientos de prueba usados para verificar el subsistema de servicios, y no se vería afectado ningún otro procedimiento de prueba.

Ejemplo **Procedimiento de prueba**

El subsistema de servicios Cuentas proporciona funcionalidad para mover dinero entre cuentas. Esta funcionalidad está involucrada en varias realizaciones de caso de uso, como las de Pagar Factura y Transferencia entre Cuentas. Un procedimiento de prueba llamado Verificar Transferencia entre Cuentas especificará cómo probar el movimiento de dinero entre cuentas. El procedimiento de prueba especificado por Verificar Transferencia entre Cuentas toma como parámetros de entrada dos identidades de cuentas y una cantidad a transferir y valida la transferencia preguntando por el saldo de las dos cuentas involucradas antes de transferir y después de la transferencia.

Los diseñadores de pruebas crean 8 casos de prueba para el caso de uso Pagar Factura y 14 casos de uso para el caso de uso Transferencia entre Cuentas. El procedimiento de prueba Verificar Transferencia entre Cuentas especifica cómo se realizan (parte de) todos esos casos de prueba.

11.5.3. Actividad: implementar prueba

El propósito de la implementación de las pruebas es automatizar los procedimientos de prueba creando componentes de prueba si esto es posible, pues no todos los procedimientos de prueba pueden ser automatizados; *véase* Figura 11.12.

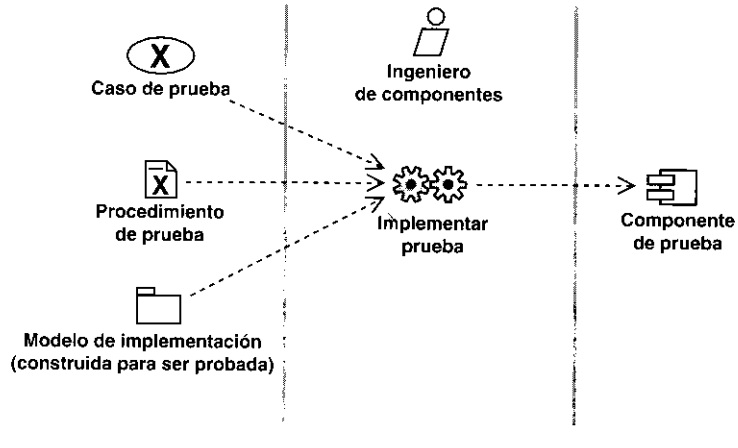


Figura 11.12. La entrada y el resultado de la implementación de pruebas.

Los componentes de prueba se crean usando los procedimientos de prueba como entrada:

- Cuando usamos una herramienta de automatización de pruebas realizamos o especificamos las acciones como se describe en los procedimientos de prueba. Estas acciones son entonces grabadas dando como salida un componente de prueba; por ejemplo, generando un *script* de prueba en Visual Basic.
- Cuando programamos los componentes de prueba explícitamente usamos los procedimientos de prueba como especificaciones iniciales. Obsérvese que dicho esfuerzo de programación podría requerir personal con buenas habilidades como programador.

Los componentes de prueba usan a menudo grandes cantidades de datos de entrada para ser probados y producen grandes cantidades de datos de salida como resultado de las pruebas. Es útil poder visualizar estos datos de forma clara e intuitiva de manera que puedan especificarse correctamente y los resultados de las pruebas puedan ser interpretados. Utilizamos para ello hojas de cálculo y bases de datos.

11.5.4. Actividad: realizar pruebas de integración

En esta actividad se realizan (*véase* Sección 10.5.2) las pruebas de integración (*véase* Sección 11.5.2.1) necesarias para cada una de las construcciones creadas en una iteración y se recopilan los resultados de las pruebas. *Véase* Figura 11.13.

Las pruebas de integración se llevan a cabo en los siguientes pasos:

1. Realizar las pruebas de integración relevantes a la construcción realizando los procedimientos de prueba manualmente para cada caso de prueba o ejecutando cualquier componente de prueba que automatice los procedimientos de prueba.
2. Comparar los resultados de las pruebas con los resultados esperados e investigar los resultados de las pruebas que no coinciden con los esperados.

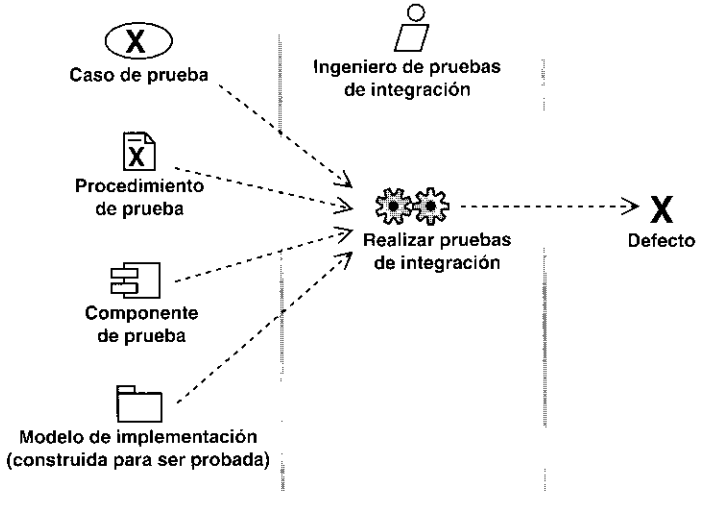


Figura 11.13. Entrada y resultado de las pruebas de integración.

3. Informar de los defectos a los ingenieros de componentes responsables de los componentes que se cree contienen los fallos.
4. Informar de los defectos a los diseñadores de pruebas, quienes usarán los defectos para evaluar los resultados del esfuerzo de prueba (como se describe en la Sección 11.5.6).

11.5.5. Actividad: realizar prueba de sistema

El propósito de la prueba de sistema es el realizar las pruebas de sistema (*véase* Sección 11.5.1.2) necesarias en cada iteración y el recopilar los resultados de las pruebas (*véase* Figura 11.14).

La prueba de sistema puede empezar cuando las pruebas de integración indican que el sistema satisface los objetivos de calidad de integración fijados en el plan de prueba de la iteración actual; por ejemplo, el 95 por ciento de los casos de prueba de integración se ejecutan con el resultado esperado.

La prueba de sistema se realiza de forma análoga a la forma en que se realiza la prueba de integración (*véase* Sección 11.5.4).

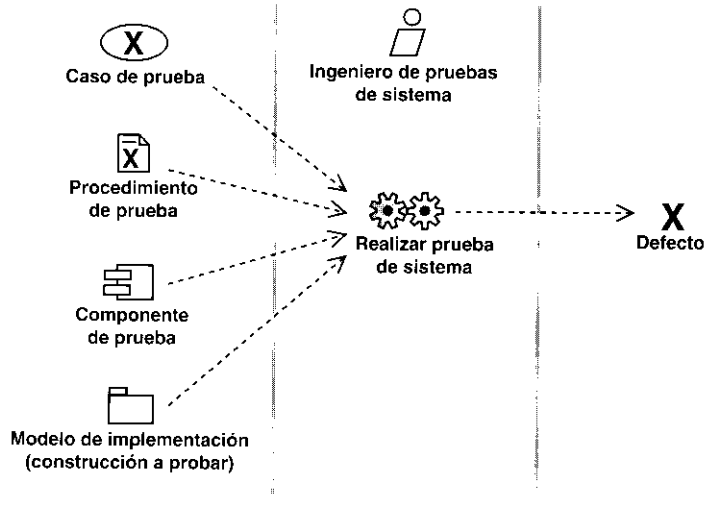


Figura 11.14. Entrada y resultado de la prueba de sistema.

11.5.6. Actividad: evaluar prueba

El propósito de la evaluación de la prueba es el de evaluar los esfuerzos de prueba en una iteración (*véase* Figura 11.15).

Los diseñadores de pruebas evalúan los resultados de la prueba comparando los resultados obtenidos con los objetivos esbozados en el plan de prueba. Éstos preparan métricas que les permiten determinar el nivel de calidad del software y qué cantidad de pruebas es necesario hacer.

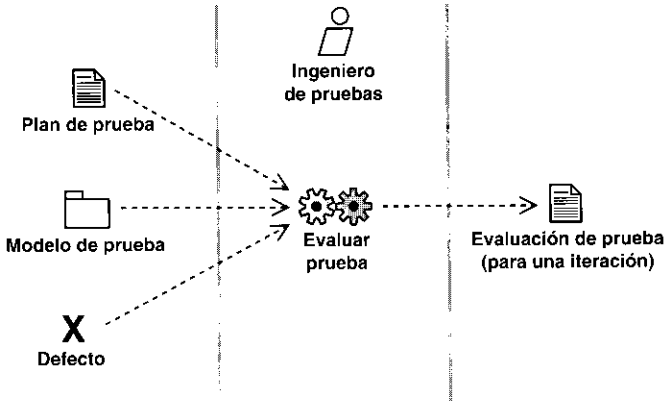


Figura 11.15. Entrada y resultado de la evaluación de prueba.

En concreto, el ingeniero de pruebas observa dos métricas:

- *Compleción de la prueba,* obtenida a partir de la cobertura de los casos de prueba y de la cobertura de los componentes probados. Esta métrica indica el porcentaje de casos de prueba que han sido ejecutados y el porcentaje de código que ha sido probado.
- *Fiabilidad,* la cual se basa en el análisis de las tendencias en los defectos detectados y en las tendencias en las pruebas que se ejecutan con el resultado esperado.

Para determinar la fiabilidad del sistema, los diseñadores de pruebas crean diagramas de las tendencias de los defectos, donde representan la distribución de tipos específicos de defectos (como defectos nuevos o fatales) a lo largo del tiempo. La prueba puede crear también diagramas de tendencias que representan el porcentaje de pruebas ejecutadas con éxito (es decir, ejecuciones de pruebas que generan los resultados esperados) a lo largo del tiempo.

Las tendencias de los defectos siguen a menudo patrones que se repiten en varios proyectos. Por ejemplo, normalmente el número de defectos nuevos generados cuando se prueba un sistema se incrementa bastante rápidamente tan pronto como comienza la prueba, después se mantiene estable durante un tiempo y finalmente empieza a caer lentamente. Por tanto, comparando la tendencia actual con tendencias similares en proyectos anteriores es posible predecir el esfuerzo necesario para alcanzar un nivel de calidad aceptable.

Basándose en el análisis de la tendencia de los defectos los diseñadores de pruebas pueden sugerir otras acciones, como por ejemplo:

- Realizar pruebas adicionales para localizar más defectos, si la fiabilidad medida sugiere que el sistema no está suficientemente maduro.
- Relajar el criterio para las pruebas, si los objetivos de calidad para la iteración actual se pusieron demasiado altos.
- Aislar las partes del sistema que parecen tener una calidad aceptable y entregarlas como resultado de la iteración actual. Las partes que no cumplieron los criterios de calidad han de ser revisados y probadas de nuevo.

Los diseñadores de pruebas documentan la compleción de la prueba, su fiabilidad y sugieren acciones en una descripción de la evaluación de la prueba.

11.6. Resumen de la prueba

El resultado principal de la prueba es el modelo de prueba, el cual describe cómo ha sido probado el sistema. El modelo de prueba incluye:

- Casos de prueba, que especifican qué probar en el sistema.
- Procedimientos de prueba, que especifican cómo realizar los casos de prueba.
- Componentes de prueba, que automatizan los procedimientos de prueba.

La prueba da también como resultado un plan de prueba, evaluaciones de las pruebas realizadas y los defectos que pueden ser pasados como entrada a flujos de trabajo anteriores, como el diseño y la implementación.

11.7. Referencias

[1] IEEE, Std 610.12-1990.

[2] Bill Hetzel, *The Complete Guide to Software Testing*, Second Edition, Wellesley, MA: QED Information Sciences, Inc., 1988.

[3] Robert V. Binder, "Developing a test budget", *Object Magazine*, 7(4), june 1997.

Parte III

Desarrollo iterativo e incremental

Un sistema software pasa por varios ciclos de desarrollo a lo largo de su tiempo de vida. Cada uno de estos ciclos da como resultado una nueva entrega del producto a clientes y usuarios, pudiendo ser la primera de estas entregas, muy probablemente, la más difícil. En esta primera entrega residen los cimientos, la arquitectura del sistema, y en ella se explora un área nueva que puede encerrar serios riesgos. Un ciclo de desarrollo tiene por tanto un contenido diferente dependiendo del lugar en que se encuentra el sistema en el ciclo de vida del software. Un cambio serio en la arquitectura puede suponer más trabajo en las fases iniciales en entregas posteriores; sin embargo, si la arquitectura original se puede extender, en la mayoría de las entregas posteriores el nuevo proyecto simplemente construiría sobre lo que ya estaba hecho, es decir, una entrega del producto se construirá sobre la entrega anterior.

Cada vez más gente adopta la idea de trabajar en los problemas lo más pronto posible dentro de cada ciclo de desarrollo. Aplican el término *iteración* para referirse a las secuencias de resoluciones de problemas en las fases de inicio y de elaboración, además de a cada serie de construcciones en la fase de construcción.

Los riesgos no vienen en un paquete estupendo con una tarjeta de identificación plegada bajo un lazo rosa. Han de ser identificados, delimitados, monitorizados y eliminados —y es importante atajar los riesgos más importantes primero—. De forma similar, ha de pensarse cuidadosamente el orden en el que tienen lugar las iteraciones, de forma que los problemas más serios se resuelvan primero. Abreviando: *haz el trabajo duro primero.*

En la Parte II describimos por separado cada flujo de trabajo. Por ejemplo, la Figura 6.2 describía dónde se centra el flujo de trabajo de los requisitos a lo largo de las diferentes fases; de

forma similar la Figura 8.2 hacía lo mismo para el análisis, la Figura 9.2 para el diseño, la Figura 10.2 para la implementación y la Figura 11.2 para la prueba.

En esta parte mostramos cómo se combinan de diversas formas los flujos de trabajo, dependiendo del lugar del ciclo de vida en el que estamos. Describimos primero, en el Capítulo 12, lo que es común a todas las fases, es decir, las cosas que aparecen en todas las fases, como la planificación de una iteración, el establecimiento de los criterios de evaluación para una iteración, el establecimiento de una lista de riesgos, la asignación de prioridades a los casos de uso y la evaluación de las iteraciones. Cada uno de los capítulos sucesivos se centra en cada una de las fases.

En la fase de inicio (Capítulo 13) la actividad se concentra en el primer flujo de trabajo, el de los requisitos, realizándose poco trabajo en los flujos de trabajo segundo y tercero (análisis y diseño). Esta fase raramente realiza ningún trabajo en los dos últimos flujos de trabajo, la implementación y la prueba.

En la fase de elaboración (Capítulo 14), mientras todavía hay una gran actividad dedicada a completar los requisitos, los flujos de trabajo segundo y tercero, análisis y diseño, reciben una gran parte de la actividad, ya que son la base de la creación de la arquitectura. Para alcanzar la línea base ejecutable de la arquitectura hay necesariamente alguna actividad en los flujos de trabajo finales, la implementación y la prueba.

En la fase de construcción (Capítulo 15) el flujo de trabajo de los requisitos disminuye, el análisis se aligera y los tres últimos flujos de trabajo representan el grueso de la actividad.

En la fase de transición (Capítulo 16) la actividad de los distintos flujos de trabajo depende de los resultados de las pruebas de aceptación y de las versiones beta del sistema. Por ejemplo, si las pruebas de las versiones beta descubren algún defecto en la implementación habrá una actividad considerable en los reanimados flujos de implementación y prueba.

El capítulo final, el Capítulo 17, vuelve al tema central del libro; en un único capítulo mostramos cómo los diversos apartados –flujos de trabajo, fases e iteraciones– interaccionan para dar lugar a un proceso bien diseñado para utilizar en el proceso de desarrollo de software de sistemas críticos. Este capítulo también dedica unos cuantos párrafos a explicar cómo deberían administrarse estas relaciones y a cómo puede ser realizada la transición hacia el Proceso Unificado en una organización que no lo utiliza.

Capítulo 12

El flujo de trabajo de iteración genérico

En este capítulo volvemos a la idea de la iteración genérica discutida en el Capítulo 5. La intención de este capítulo es destilar el patrón común que caracteriza a todas las iteraciones a partir de las distintas iteraciones que ocurren durante las cuatro fases.

Utilizamos este patrón genérico como base sobre la que construir las iteraciones concretas; el contenido de una iteración cambia para acomodarse a los objetivos particulares de cada fase (*véase* Figura 12.1).

El flujo de trabajo de iteración genérica incluye los cinco flujos de trabajo: requisitos, análisis, diseño, implementación y prueba. Éste también incluye la planificación, que precede a los flujos de trabajo, y la evaluación, que va detrás de ellos (los Capítulos 6-11 describen cada uno de los flujos de trabajo por separado). En este capítulo nos centraremos en la planificación, en la evaluación y en otras actividades comunes a todos los flujos de trabajo.

La planificación es necesaria a lo largo de todo el ciclo de desarrollo, pero antes de que podamos planear es necesario saber qué es lo que tenemos que hacer; los cinco flujos de trabajo fundamentales proporcionan un punto de partida. Otro aspecto clave de la planificación es la administración de riesgos, es decir, la identificación y mitigación de riesgos realizando el correspondiente conjunto de casos de uso. Por supuesto, un plan no estará completo sin la

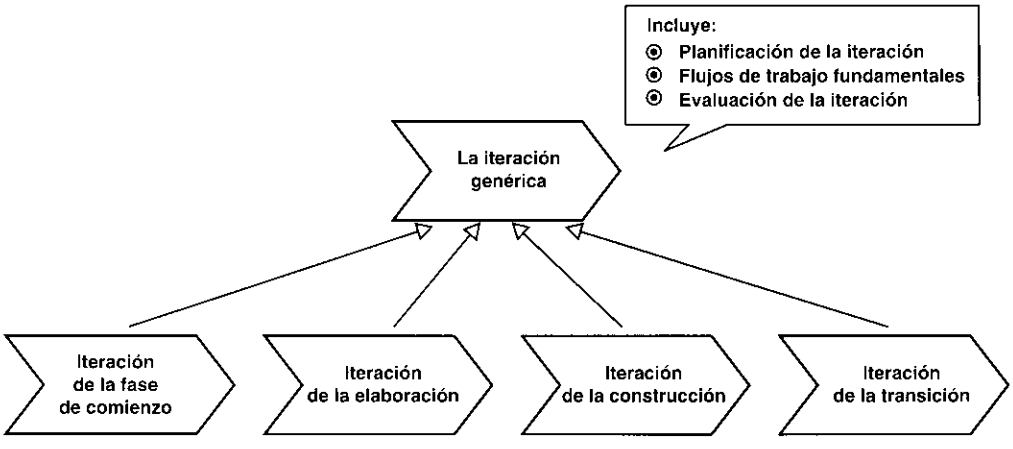


Figura 12.1. El flujo de trabajo de iteración genérica se emplea para describir los flujos de trabajo de iteración concretos para cada fase. (La intención de esta figura es ilustrar la forma en la que hemos estructurado la Parte III, con las generalidades sobre la iteración en el Capítulo 12 y las especializaciones para cada fase en los Capítulos 13-16).

estimación de recursos que éste requerirá y, por último, también llevarse a cabo la evaluación de la ejecución de cada iteración y fase.

12.1. La necesidad de equilibrio

En cada uno de los momentos en el ciclo de vida de un proyecto de desarrollo software están activas muchas secuencias de actividades diferentes: trabajamos en nuevas funciones, diseñamos la arquitectura, recibimos comentarios de los usuarios, mitigamos riesgos, planificamos el futuro, etc. Debemos equilibrar y sincronizar en todo momento estas secuencias de actividades diferentes para controlar así esta complejidad.

Los desarrolladores dividen el trabajo, que es abrumadoramente complejo como un todo, en partes más pequeñas y comprensibles. A lo largo del ciclo de vida de desarrollo dividen el trabajo en fases, y cada una de estas fases en iteraciones, como se esbozó en la Parte I.

Dentro de cada iteración el proyecto se esfuerza en alcanzar un equilibrio entre las secuencias de actividad que se ejecutan a lo largo de la iteración, lo que significa que deberíamos trabajar en las cosas apropiadas en cada iteración. Las cosas correctas en las que trabajar en cada momento dependen del punto del ciclo de vida en que nos encontramos; la tarea de un proyecto es seleccionar las cosas correctas en las que trabajar en cada secuencia de actividad. Determinar el equilibrio de las secuencias de actividades es también importante para asegurar que son de una importancia parecida, de forma que pueda asignárseles una prioridad y sincronizarlas con facilidad. Fallar en la consecución de este tipo de equilibrio y ejecución eficiente se traduce en general en el desastre de muchos ciclos de vida de desarrollo incrementales e iterativos.

En las primeras iteraciones trabajamos con riesgos críticos, casos de uso fundamentales, cuestiones arquitectónicas, la elección del entorno de desarrollo, todas las cuestiones orientadas

a la investigación, mientras que en las últimas iteraciones trabajamos en actividades orientadas al desarrollo, como la implementación y la prueba, problemas de evaluación del rendimiento y el despliegue del sistema. El relacionar todas estas actividades entre sí es una cuestión de equilibrio bastante delicada, lo que hace que el desarrollo de software sea una tarea extraordinariamente difícil.

Lo que hacemos en una iteración es entender estas secuencias de actividades diferentes y buscar el equilibrio entre ellas. En el Proceso Unificado algunas de estas secuencias de actividades han sido identificadas y descritas en los flujos de trabajo fundamentales. Hay otras secuencias que no hemos identificado formalmente, pero que perfectamente podrían ser tratadas de la misma forma que los flujos de trabajo. Por ejemplo:

- La interacción con clientes en requisitos nuevos.
- La preparación de una oferta para los clientes.
- La comprensión del contexto de un sistema creando un modelo de negocio.
- La planificación y administración de un proyecto.
- El establecimiento y administración de un entorno de desarrollo, es decir, el proceso y las herramientas.
- La administración de riesgos.
- La instalación de un producto en su lugar de destino.

12.2. Las fases son la primera división del trabajo

El primer paso hacia la división del proceso de desarrollo de software consiste en separar las partes en cuatro fases atendiendo al momento en que se realizan: inicio, elaboración, construcción y transición. Cada una de las fases se divide entonces en una o más iteraciones. Este capítulo esboza la naturaleza general de estas fases e iteraciones y los cuatro capítulos próximos consideran cada una de las fases en detalle.

12.2.1. La fase de inicio establece la viabilidad

El objetivo principal de esta fase es establecer el análisis de negocio —el análisis para decidir si se sigue adelante con el proyecto—, aunque este análisis se seguirá desarrollando en la fase de elaboración conforme se vaya disponiendo de más información. La fase de inicio no es un estudio completo del sistema propuesto, sino que en ella buscamos el porcentaje de casos de uso necesarios para fundamentar el análisis de negocio inicial. Para realizar este análisis seguimos cuatro pasos:

1. Delimitar el ámbito del sistema propuesto, es decir, definir los límites del sistema y empezar a identificar las interfaces con sistemas relacionados que están fuera de los límites.
2. Describir o esbozar una propuesta de la arquitectura del sistema, y en especial de aquellas partes del sistema que son nuevas, arriesgadas o difíciles. En este paso sólo

llegamos hasta una descripción de la arquitectura, raramente hasta un prototipo ejecutable; esta descripción de la arquitectura consiste en unas primeras versiones de los modelos. El principal objetivo es hacer creíble el que se pueda crear una arquitectura estable del sistema en la siguiente fase. Esta arquitectura no es construida en esta fase, simplemente hacemos creíble el que se pueda crear una. La construcción de esta arquitectura es el producto más importante de la fase de elaboración.

3. Identificar riesgos críticos, es decir, los que afectan a la capacidad de construir el sistema, y determinar si podemos encontrar una forma de mitigarlos, quizás en una etapa posterior. En esta fase consideramos sólo los riesgos que afectan la viabilidad, es decir, aquéllos que amenazan el desarrollo con éxito del sistema. Cualquier riesgo no crítico que se identifique es colocado en la lista de riesgos para su posterior consideración detallada en la fase siguiente.

4. Demostrar a usuarios o clientes potenciales que el sistema propuesto es capaz de solventar sus problemas o de mejorar sus objetivos de negocio construyendo un prototipo. En la fase de inicio podemos construir un prototipo para mostrar una solución al problema de los clientes o usuarios potenciales, el cual demuestra las ideas básicas del nuevo sistema haciendo énfasis en su uso —interfaces de usuario o algún algoritmo nuevo interesante—. Este prototipo tiende a ser exploratorio, es decir, que demuestra una posible solución pero que puede que no dé lugar al producto final, sino que acabe descartándose. Por el contrario, un prototipo arquitectónico, desarrollado en la fase de elaboración, suele ser capaz de evolucionar, es decir, un prototipo capaz de adaptarse sufriendo modificaciones en la etapa siguiente.

Continuamos con estos esfuerzos hasta el momento en que desarrollar el sistema parece ser rentable económicamente, es decir, hasta que se concluye que el sistema proporcionará ingresos u otros beneficios proporcionales a la inversión necesaria con un margen suficiente para construirlo. En otras palabras, hemos realizado una primera versión del análisis de negocio, el cual será refinado en la fase siguiente, la de elaboración.

La intención es minimizar los gastos en tiempo de planificación, esfuerzo y fondos en esta fase hasta que decidamos si el sistema es viable o no. En el caso de un sistema completamente nuevo en un dominio poco explorado esta consideración puede llevar un tiempo y un esfuerzo considerables y puede extenderse a varias iteraciones. En el caso de un sistema bien conocido en un dominio establecido o en el caso de tratarse de la extensión de un sistema a una nueva versión, los riesgos y casos desconocidos pueden ser mínimos, permitiendo que esta primera fase se complete en pocos días.

12.2.2. La fase de elaboración se centra en la factibilidad

El resultado principal de la fase de elaboración es una arquitectura estable para guiar el sistema a lo largo de su vida futura. Esta fase también lleva el estudio del sistema propuesto al punto de planificar la fase de construcción con gran precisión. Con estos dos grandes objetivos —la arquitectura y la estimación de costes con gran precisión— el equipo hace lo siguiente:

1. Crea una línea base para la arquitectura que cubre la funcionalidad del sistema significativa arquitectónicamente y las características importantes para las personas involu-

cradas como se describe en el Capítulo 4. Esta línea base consiste en los artefactos de los modelos, la descripción de la arquitectura y en una implementación ejecutable de ésta. Esta línea base no sólo demuestra que podemos construir una arquitectura estable, sino que encierra a la arquitectura.

2. Identifica los riesgos significativos, es decir, los riesgos que podrían perturbar los planes, costes y planificaciones de fases posteriores, y los reduce a actividades que pueden ser medidas y presupuestadas.
3. Especifica los niveles a alcanzar por los atributos de calidad, como la fiabilidad (porcentajes de defectos) y los tiempos de respuesta.
4. Recopila casos de uso para aproximadamente el 80 por ciento de los requisitos funcionales, suficiente para planificar la fase de construcción. (Clarificaremos lo que queremos decir con el 80 por ciento más tarde en este capítulo.)
5. Prepara una propuesta de la planificación cubierta, personal necesario y coste dentro de los límites establecidos por las prácticas de negocio.

Los requisitos y la arquitectura (en el análisis, el diseño y la implementación) representan el grueso del esfuerzo en las fases de inicio y de elaboración.

12.2.3. La fase de construcción construye el sistema

El objetivo general de la fase de construcción viene indicado por su tarea fundamental: la capacidad de operación inicial. Es decir, un producto listo para ser distribuido como versión beta y ser sometido a pruebas. Esta fase emplea más personal a lo largo de un periodo de tiempo más largo que ninguna otra fase, y es por esto por lo que es tan importante tener todos los detalles importantes bien preparados antes de empezar con la construcción. Generalmente esta fase requiere un mayor número de iteraciones que las fases anteriores.

En muchos casos el tener un trabajo de requisitos, análisis y diseño pobre parece implicar que la construcción lleve una cantidad de tiempo demasiado grande. En estos casos los desarrolladores tienen que construir el sistema como pueden, empleando más tiempo de lo necesario para que se satisfagan los requisitos y para eliminar los defectos *(bugs)*. Una de las grandes ventajas de construir software utilizando un enfoque que use múltiples fases y un desarrollo iterativo e incremental es que nos permite equilibrar la asignación de recursos y de tiempo a lo largo del ciclo de vida (*véase* Sección 12.1).

Entre las actividades de la fase de construcción tenemos:

1. La extensión de la identificación, descripción y realización de casos de uso a todos los casos de uso.
2. La finalización del análisis (posiblemente quedan todavía más de la mitad de los casos de uso por ser analizados cuando comienza la fase de construcción), del diseño, de la implementación y de la prueba (posiblemente queda un 90 por ciento).
3. El mantenimiento de la integridad de la arquitectura, modificándola cuando sea necesario.
4. La monitorización de los riesgos críticos y significativos arrastrados desde las dos primeras fases, y su mitigación si se materializan.

12.2.4. La fase de transición se mete dentro del entorno del usuario

La fase de transición comienza a menudo con la entrega de una versión beta del sistema, es decir, la organización distribuye un producto software capaz ya de un funcionamiento inicial a una muestra representativa de la comunidad de usuarios. El funcionamiento del producto en el entorno de los usuarios es frecuentemente una prueba del estado de desarrollo del producto más severa que el funcionamiento en el entorno del que lo desarrolla.

Entre las actividades de la transición se incluyen:

- Preparar las actividades, como la preparación del lugar.
- Aconsejar al cliente sobre la actualización del entorno (hardware, sistemas operativos, protocolos de comunicaciones, etc.) en los que se supone que el software va a funcionar.
- Preparar los manuales y otros documentos para la entrega del producto. En la fase de construcción se prepara una documentación preliminar para los usuarios de las versiones beta.
- Ajustar el software para que funcione con los parámetros actuales del entorno del usuario.
- Corregir los defectos encontrados a lo largo de las pruebas realizadas a la versión beta.
- Modificar el software al detectar problemas que no habían sido previstos.

La fase de transición termina con la entrega del producto final. Sin embargo, antes de que el equipo del proyecto abandone el proyecto, los líderes del equipo llevan a cabo un estudio del sistema con los siguientes objetivos:

- Encontrar, discutir, evaluar y registrar las "lecciones aprendidas" para referencias futuras.
- Registrar asuntos útiles para la entrega o versión siguiente.

12.3. La iteración genérica

Nosotros distinguimos entre los flujos de trabajo fundamentales y los flujos de trabajo iterativos. Los flujos de trabajo fundamentales —requisitos, análisis, diseño, implementación y prueba— se describen en los Capítulos 6-11. En el Proceso Unificado estos flujos de trabajo fundamentales no ocurren sólo una vez, como sucede teóricamente en el modelo en cascada, sino que se repiten más bien en cada iteración, una y otra vez, como flujos de trabajo iterativos. En cada repetición, sin embargo, se diferencian en los detalles, se enfrentan a los asuntos centrales de cada iteración.

12.3.1. Los flujos de trabajo fundamentales se repiten en cada iteración

La iteración genérica consiste en los cincos flujos de trabajo: requisitos, análisis, diseño, implementación y prueba, e incluye también planificación y evaluación. *Véase* Figura 12.2.

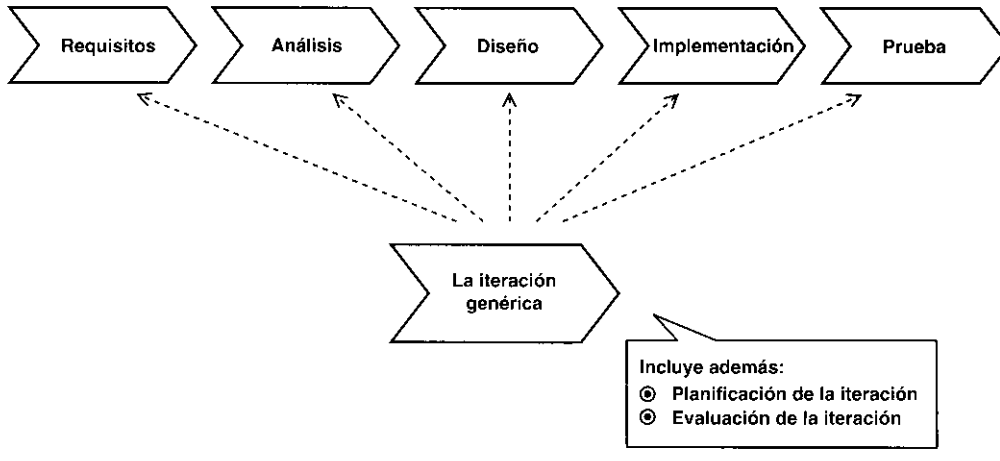


Figura 12.2. Los cinco flujos de trabajo fundamentales se repiten en cada iteración, precedidos por la planificación y seguidos por la evaluación.

En las Secciones 12.4-12.7 discutimos la planificación que precede a la iteración y en la Sección 12.8 la evaluación de cada iteración. A continuación, en los Capítulos 13-16, mostramos en detalle cómo se aplican los cinco flujos de trabajo en cada una de las fases.

12.3.2. Los trabajadores participan en los flujos de trabajo

A veces nos hemos referido al desarrollo de software como un proceso "complejo". La Figura 12.3, sin embargo, proporciona una visión general simplificada del proceso; aunque ésta dista mucho de ser simple y, como es bien sabido, la realidad es todavía mucho más complicada. En este capítulo no vamos a describir en detalle cómo produce cada trabajador los artefactos de los que él o ella es responsable, ni siquiera cuáles son estos artefactos. Las pequeñas ruedas dentadas en la figura representan ese trabajo y las flechas entre estas actividades representan las relaciones temporales. *Véase* los Capítulos 6-11 para obtener estos detalles.

A pesar de esto, la Figura 12.3 nos da una impresión de qué se realiza dentro de cada iteración. Por ejemplo, empezando en la esquina superior izquierda, un analista de sistemas identifica los casos de uso y los actores y los estructura dentro de un modelo de casos de uso. A continuación, un especificador de casos de uso detalla cada uno de los casos de uso y un diseñador de interfaces de usuario construye un prototipo de los interfaces de usuario. El arquitecto asigna prioridades a los casos de uso a ser desarrollados dentro de las iteraciones teniendo en cuenta los riesgos.

Puede verse que las actividades particulares realizadas dentro del "círculo" de los requisitos podrían variar dependiendo de la posición de la iteración dentro del proceso de desarrollo completo. En la fase de inicio, por ejemplo, los trabajadores se limitan a detallar y asignar prioridades a la pequeña proporción de casos de uso necesarios en esa fase. Los primeros cuatro flujos de trabajo se suceden a lo largo del tiempo, aunque pueda darse algún solapamiento. Los tres trabajadores involucrados en el análisis continúan con su trabajo en el flujo de diseño.

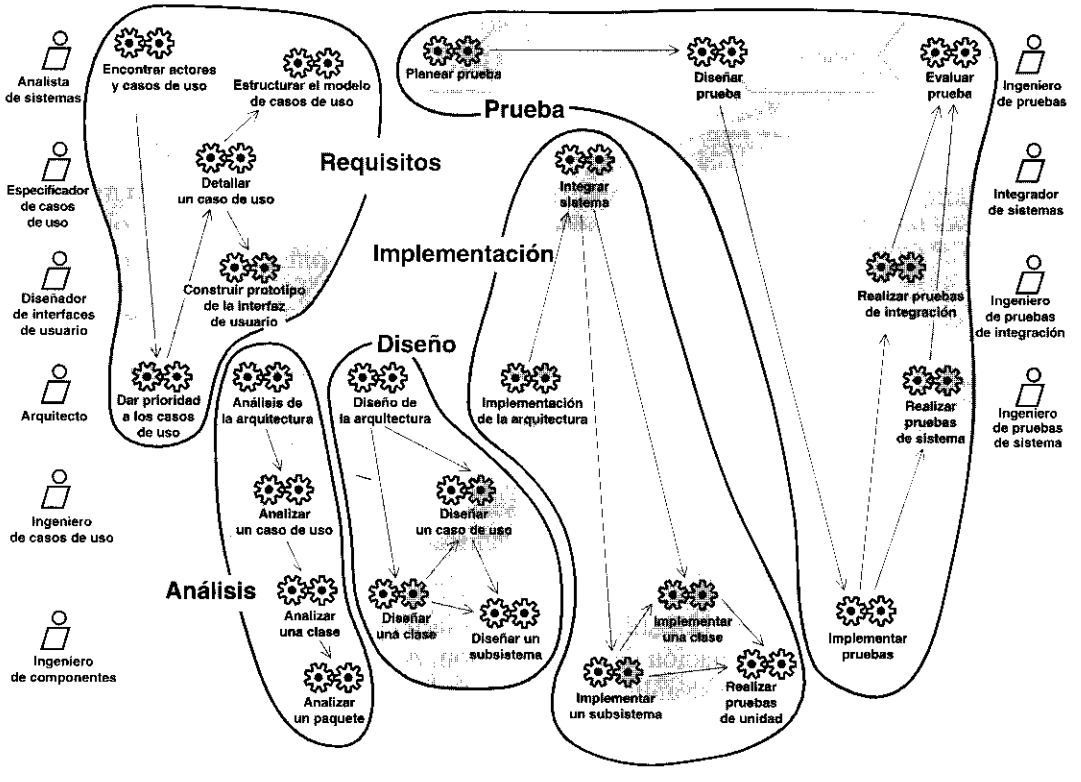


Figura 12.3. Los nombres de los trabajadores están colocados en los márgenes izquierdo y derecho, identificando las "calles". El tiempo avanza de izquierda a derecha. Los flujos de trabajo, incluyendo los trabajadores y las actividades que realizan, están representados por los "círculos" de trazo libre. El ingeniero de componentes, por ejemplo, diseña una clase y un subsistema en el flujo de diseño, implementa una clase y un subsistema y realiza una prueba de unidad en el flujo de implementación.

La prueba, sin embargo, comienza muy pronto, cuando el ingeniero de pruebas planifica lo que hay que hacer. Tan pronto como hay disponibles detalles suficientes en el flujo de implementación, el ingeniero de pruebas diseña las pruebas. Conforme son integrados los componentes que han pasado las pruebas de unidad, los ingenieros de pruebas de sistema y los de integración prueban los resultados de varios niveles de integración. El ingeniero de pruebas evalúa si la prueba que había solicitado ha sido adecuada o no.

En los capítulos siguientes añadiremos sobre la Figura 12.3 lo que necesitemos para representar los flujos de trabajo de iteración en cada fase. Todos estos flujos de trabajo siguen el patrón de esta figura genérica, aunque ya que el foco cambia en las diferentes fases los flujos de trabajo de iteración correspondientes tendrán un grado distinto.

12.4. El planificar precede al hacer

Cuando comenzamos la fase de inicio sabemos tres cosas:

- Vamos a llevar a cabo el proyecto en una serie de iteraciones en cuatro fases.
- Tenemos la información sobre el sistema propuesto que nuestros predecesores recopilaron (y que llevaron a comenzar un proyecto).

- Tenemos nuestra propia información básica sobre el dominio y sobre sistemas similares en los que trabajamos en el pasado.

Partiendo de esta información hemos de planear el proyecto (plan de proyecto) y cada iteración (plan de iteración). Al principio, con información limitada con la que trabajar, ambos planes contienen poco detalle, pero conforme trabajamos en las fases de inicio y de elaboración los planes llegan a estar más detallados.

En primer lugar describimos cómo planear las fases y las iteraciones y cómo evaluar cada iteración. En la Sección 12.5 discutimos cómo afectan los riesgos al plan y en la Sección 12.6 cómo mitigamos estos riesgos seleccionando los casos de uso correctos. Por último, discutimos la localización de recursos en la Sección 12.7.

12.4.1. Planear las cuatro fases

Gracias a las indicaciones del Proceso Unificado sabemos qué hay que hacer en cada una de las fases. Nuestra tarea, en el plan del proyecto, es reducir estas indicaciones a términos concretos:

- Asignaciones de tiempo. Decidimos cuánto tiempo asignar a cada fase y la fecha en la que cada fase ha de haber sido completada. Estos tiempos, aunque establecidos con precisión, pueden ser bastante inestables al principio de la fase de inicio, pero se irán fijando conforme vayamos aprendiendo más. Después de todo no es hasta el final de la fase de elaboración cuando hacemos una propuesta firme.
- Hitos principales. Una fase termina cuando ha sido alcanzado el criterio actual. Al principio estos criterios pueden ser poco más que creencias basadas en alguna información, información que puede venir de nuestra experiencia pasada en el dominio, de las similitudes del sistema propuesto con sistemas anteriores, de las especificaciones del rendimiento que se pretende que el nuevo sistema alcance y de la capacidad de nuestra organización de desarrollo de software.
- Iteraciones por fase. Dentro de cada fase el trabajo se lleva a cabo a lo largo de varias iteraciones. El carácter de cada iteración está contenido en el plan de proyecto.
- Plan de proyecto. El plan de proyecto esboza un "mapa de carreteras" global, que cubre la planificación, las fechas y criterios de los objetivos principales y la división de las fases en iteraciones.

Uno puede esperar que la primera iteración en la fase de inicio sea difícil, ya que además de la iteración misma uno ha de cubrir lo siguiente:

- Ajustar el Proceso Unificado al proyecto y seleccionar herramientas para automatizar el proceso.
- Empezar a reunir a gente con el talento necesario para el proyecto.
- Crear las relaciones que dan lugar a un equipo efectivo.
- Entender el dominio, que a menudo es desconocido para el equipo.
- Percibir la naturaleza del proyecto, que será más difícil si se trata de un desarrollo nuevo que si se trata de la extensión de un producto ya existente.
- Familiarizar al equipo con las herramientas apropiadas para el proceso y el proyecto.

12.4.2. Plan de iteraciones

Cada fase está formada por una o más iteraciones. La planificación de las iteraciones pasa por un conjunto de pasos comparables con los seguidos en la planificación de las fases:

- Planificación de iteración. Decidimos cuánto tiempo puede requerir cada iteración y su fecha de terminación; al principio a *grosso modo* y después cada vez con más precisión conforme aprendemos.
- Contenido de iteración. Mientras que el plan del proyecto esboza las iteraciones planeadas en términos generales, cuando la fecha de comienzo de una iteración se acerca, planeamos lo que ha de hacerse más en detalle. El contenido de una iteración se basa en lo siguiente:
  - — Los casos de uso que tienen que ser completados al menos parcialmente durante la iteración.
  - — Los riesgos técnicos que han de ser identificados, transformados en casos de uso y mitigados en ese momento.
  - — Los cambios que han sufrido los requisitos o los defectos que pueden haber sido corregidos.
  - — Los subsistemas que han de ser implementados parcial o completamente. Este punto varía dependiendo de la fase en que se considere. Por ejemplo, en la fase de elaboración identificamos la mayoría de los subsistemas y todas las clases significativas arquitectónicamente; en cambio, en la fase de construcción completamos los subsistemas con más comportamiento, resultando en componentes más completos.

El plan de la iteración actual está completamente detallado, y el de la siguiente va siendo más detallado conforme vamos aprendiendo. Los detalles de iteraciones posteriores pueden estar limitados por el conocimiento disponible en ese momento:

- **Hitos secundarios** (Apéndice C). El alcanzar los criterios preestablecidos (preferiblemente en la fecha planeada) marca la compleción de la iteración actual.
- **Plan de iteración** (Apéndice C). Las actividades de cada iteración son recopiladas en un detallado plan de iteración. Al comienzo de cada iteración identificamos los individuos disponibles para actuar como trabajadores.

El número de iteraciones planeado para cada fase depende, básicamente, de la complejidad del sistema propuesto. Un proyecto muy simple podría ser realizado con una sola iteración por fase, mientras que un proyecto más complicado podría requerir más iteraciones. Por ejemplo:

- **Fase de inicio** (Apéndice C): una iteración, principalmente dedicada a definir el ámbito del sistema.
- **Fase de elaboración** (Apéndice C): dos iteraciones, la primera para esbozar la arquitectura y la segunda para completar la línea base de la arquitectura.
- **Fase de construcción** (Apéndice C): dos iteraciones, para asegurar que los incrementos resultantes funcionan satisfactoriamente.
- **Fase de transición** (Apéndice C): una iteración.

Conforme el proyecto bajo consideración va siendo mayor y más complejo y requiere más consideraciones nuevas podemos esperar que el tamaño de la organización del desarrollo

vaya creciendo. Habrá más iteraciones y sus duraciones variarán, dependiendo del tamaño del sistema, entre una semana y tres meses por cada iteración.

12.4.3. Pensar a largo plazo

Durante el largo periodo de tiempo que el sistema puede durar, éste puede presenciar grandes cambios, como nuevas tecnologías, nuevas interfaces con el entorno del sistema o plataformas avanzadas. Además, los planificadores deberían examinar si el negocio necesitaría ser adaptado a otras organizaciones, como filiales o socios resultantes de fusiones. Sin embargo, no se debe llegar al absurdo especulando con el futuro. Uno no quiere construir una arquitectura rígida para el sistema si es posible prever que ésta cambiará en el futuro, pero tampoco queremos introducir una flexibilidad innecesaria en el sistema, es decir, una flexibilidad que no será utilizada nunca.

12.4.4. Planear los criterios de evaluación

Las iteraciones son cortas comparadas con los proyectos tradicionales. Para que no queden indefinidas los líderes del proyecto, además de establecer una planificación, han de esbozar los objetivos clave de cada iteración. Para completar los objetivos, éstos establecen criterios que indican la compleción de la iteración. Estos criterios, como el conjunto de características mínimo en un momento determinado, nos permiten centrarnos en una iteración y nos ayudan a fijarnos en lo temporalmente cercano. Los siguientes son ejemplos de criterios de este tipo:

- Requisitos funcionales expresados en forma de casos de uso.
- Requisitos no funcionales que acompañan a los casos de uso a los que se refieren.
- Requisitos no funcionales que, como se detalla en la Sección 6.7, no son específicos de un caso de uso concreto, se incluyen en una lista de requisitos adicionales.

Por ejemplo, uno de los objetivos en una de las primeras iteraciones podría ser resolver las ambigüedades en la expresión actual de los requisitos del cliente. Los criterios especifican lo que los planificadores intentan que se llegue a conseguir en una iteración en términos que puedan ser medidos, como rendimiento, u observados, como una característica deseable.

El jefe del proyecto establece criterios de evaluación para cada iteración y para la fase misma por adelantado. Cada una necesita un punto de terminación claro que permita a los desarrolladores darse cuenta de que han terminado. Además, estos puntos de terminación proporcionan los productos intermedios que los administradores pueden utilizar para determinar el progreso del trabajo.

A grandes rasgos, los criterios de evaluación pueden clasificarse en dos categorías: requisitos verificables y criterios más generales.

Los ingenieros de pruebas han estado participando en el desarrollo desde la fase de inicio, como se explica en el Capítulo 11, y son ellos los que identifican qué características de los casos de uso pueden ser confirmadas haciendo pruebas. Ellos planean los casos de prueba que definen las pruebas de integración y de regresión, así como las pruebas de sistema. En el desarrollo iterativo el ciclo de pruebas también es iterativo. Cada una de las construcciones

creadas en una iteración es sometida a prueba. Los ingenieros de pruebas aumentan y refinan las pruebas que se ejecutan sobre cada construcción, acumulándose así una cantidad de pruebas que serán utilizadas como pruebas de regresión en etapas posteriores. Las primeras iteraciones introducen más funciones nuevas y más pruebas que las últimas iteraciones. Conforme avanza la integración de construcciones disminuye el número de nuevas pruebas y se ejecuta un número creciente de pruebas de regresión para validar la implementación del sistema realizada hasta ese momento. Por tanto, las primeras construcciones e iteraciones requieren más planificación y diseño de pruebas, mientras que en las últimas se realiza más ejecución y evaluación de pruebas.

Los criterios generales no se pueden reducir a caminos dentro del código que los ingenieros de pruebas pueden probar; puede que éstos hayan de ser percibidos en prototipos primero y en las series de construcciones en funcionamiento y de iteraciones más tarde. Los usuarios, clientes y desarrolladores pueden ver pantallas e interfaces gráficas de usuario con mayor perspicacia de la que pueden emplear en estudiar la información estática contenida en los artefactos de los modelos.

Los criterios de evaluación dicen cómo verificar que los requisitos para una iteración han sido desarrollados correctamente. Especifican en términos que pueden ser observados o verificados hasta dónde intenta el jefe del proyecto que llegue la iteración. Al principio estos criterios son un tanto vagos, pero van siendo más detallados conforme los casos de uso, escenarios de los casos de uso, requisitos de rendimiento y casos de prueba concretan cómo han de ser los incrementos sucesivos.

12.5. Los riesgos influyen en la planificación del proyecto

El modo en que se planifica el desarrollo de un nuevo sistema está influenciado en gran medida por los riesgos que se perciben. Por tanto, uno de los primeros pasos, al principio de la fase de inicio, es crear una lista de riesgos. Al principio, esto puede ser difícil por la falta de información, pero probablemente tenemos alguna intuición de cuáles pueden ser los riesgos —aquéllos que determinarán si seremos capaces de construir el sistema—. Conforme se va realizando el trabajo inicial se va apreciando cuáles serán los riesgos críticos —aquéllos que han de ser mitigados para poder ofrecer una planificación y un coste y para determinar un objetivo de calidad.

12.5.1. Administrar la lista de riesgos

Una cosa es saber de una forma vaga que el desarrollo de software implica riesgos y otra distinta ponerlos donde todo el mundo pueda verlos, ser guiados por ellos y hacer algo con ellos. Ése es el propósito de la lista de riesgos. No se trata de algo que se guarde en un cajón o en una carpeta en el computador, todo lo que es preciso saber sobre un riesgo para poder trabajar con él, incluyendo su identificador único, debería estar en la lista. Esta lista incluye:

- Descripción: comienza con una breve descripción y se van añadiendo detalles conforme se va aprendiendo.

- Prioridad: se le asigna una prioridad al riesgo; al principio el riesgo se clasificará como crítico, significativo o rutinario, aunque conforme la lista se va desarrollando es posible que se quieran añadir más categorías.
- Impacto: indica qué partes del proyecto o del sistema se verán afectadas por el riesgo.
- Monitor: indica quién es responsable del seguimiento de un riesgo persistente.
- Responsabilidad: indica qué individuo o unidad de la organización es responsable de eliminar el riesgo.
- Contingencia: indica lo que ha de hacerse en caso de que el riesgo se materialice.

En un proyecto de un cierto tamaño pueden encontrarse cientos de riesgos; probablemente, en proyectos grandes los riesgos deberían ser almacenados en una base de datos de forma que puedan ser ordenados y se puedan hacer búsquedas sobre ellos de forma eficiente. Una de las razones por las que se sigue un desarrollo iterativo es que el equipo no puede centrarse en todo al mismo tiempo; los riesgos se ordenan por nivel de seriedad o por su influencia en el desarrollo y son resueltos en orden. Como hemos dicho muchas veces, los riesgos a tratar en primer lugar son aquéllos que podrían provocar que el proyecto fallara. Algunos riesgos no tienen una solución fácil y permanecen en la lista de riesgos por algún tiempo. Algunas organizaciones han encontrado útil destacar los "diez principales" de la lista como una forma de centrar la atención.

La lista de riesgos no es un instrumento estático, la lista crece conforme se descubren nuevos riesgos. Conforme son eliminados los riesgos o cuando pasamos un punto del desarrollo en que un riesgo particular no puede materializarse son quitados de la lista. El jefe del proyecto preside reuniones periódicas, a menudo coincidiendo con la terminación de las iteraciones, para revisar el estado de los riesgos más importantes. Otros líderes presiden sesiones para discutir riesgos menos importantes.

12.5.2. Los riesgos influyen en el plan de iteración

Durante la fase de inicio se identifican los riesgos críticos y el equipo intenta mitigarlos. Exploran su naturaleza hasta el punto de preparar un plan de iteración. Para saber lo suficiente para hacer un plan, por ejemplo, se ha de desarrollar el pequeño conjunto de casos de uso relacionados con el riesgo e implementarlo en un prototipo. A continuación, con ciertas entradas, se dan cuenta de que el prototipo genera una salida inaceptable —por ejemplo, un despido prematuro—, es decir, un riesgo crítico. Estas entradas han de estar dentro del rango de entradas especificado, quizás en sus límites, y aún así causar salidas inaceptables.

Además de las influencias que pueden tener los riesgos más serios sobre el éxito de un proyecto, todos los riesgos tienen algún impacto en la planificación, el coste o la calidad. Algunos de estos riesgos pueden ser lo suficientemente serios como para prolongar la planificación o incrementar el esfuerzo más allá del esfuerzo planeado —a no ser que sean mitigados antes de que sus efectos indeseables se materialicen—. En casi todos los casos un impacto en la planificación también afecta al esfuerzo y al coste. En algunos casos, puede que un riesgo tenga poco impacto sobre la planificación o el coste pero afecte negativamente a otros factores, como la calidad o el rendimiento.

12.5.3. Planificar la acción sobre los riesgos

El principio general es tener un plan de acción a seguir sobre los riesgos. Las fases y las iteraciones dentro de las fases proporcionan el medio de planificar las acciones sobre los riesgos. Por ejemplo, podemos planear tratar los riesgos que afectan la capacidad de construir el sistema en la iteración o iteraciones de la fase de inicio. Es decir, se eliminan si es posible o al menos se da un plan de contingencia.

En nuestra experiencia el no tener una planificación de riesgos no ha funcionado muy bien. Cuando no existe un esfuerzo consciente para actuar pronto sobre los riesgos, éstos se manifiestan usualmente al final de la planificación, mientras se realizan las pruebas de integración y de sistema. Resolver a esa altura cualquier problema serio, que puede requerir amplias modificaciones del sistema, puede retrasar la entrega durante varias semanas o incluso más. En el enfoque iterativo la construcción de prototipos, construcciones y artefactos descubre desde la primera fase los riesgos mientras hay aún tiempo para tratarlos.

Sabemos que es difícil identificar y describir algunos tipos de riesgos. Por muchas razones algunos riesgos pueden ser "oscuros" —usualmente porque la gente no los busca lo suficiente—. Otra razón por la que los riesgos pueden pasar desapercibidos es que algunas de las personas involucradas se confíen exageradamente en lo que puede ser realizado con el precio marcado y en el tiempo deseado. Si hay riesgos que no pueden ser identificados, entonces el proyecto no puede planificarlos en iteraciones ni mitigarlos en ningún orden.

Sea cual sea la razón, en algunos proyectos se pasarán por alto algunos riesgos hasta bastante avanzada la planificación, especialmente cuando el equipo del proyecto tenga poca experiencia administrando riesgos. Con práctica y experiencia los equipos mejorarán su capacidad para ordenar los riesgos de forma que se permita que el proyecto proceda por un camino lógico. Por ejemplo, en la fase de construcción los riesgos que podrían hacer que la segunda iteración no siguiera su planificación deberían ser mitigados no más tarde de la primera iteración de esa fase. El objetivo es hacer que cada iteración en la fase de construcción proceda libre de eventos y de acuerdo con el plan. Es improbable que esto ocurra si el proyecto se encuentra con riesgos inesperados que no puedan ser resueltos con rapidez.

12.6. Asignación de prioridades a los casos de uso

En esta sección discutimos la selección de casos de uso como guías dentro de una iteración. Recordemos que cada iteración en el Proceso Unificado está guiada por un conjunto de casos de uso, o, más exactamente, por un conjunto de escenarios a través de los casos de uso. Es más exacto porque en las iteraciones iniciales no es necesario considerar casos de uso completos, sino sólo los escenarios o caminos a través de los casos de uso más apropiados para la tarea que tenemos entre manos. A veces, cuando decimos que seleccionamos casos de uso, queremos decir que estamos seleccionando los escenarios más apropiados para la iteración.

La tarea que tiene como resultado esta selección se denomina *dar prioridades a los casos de uso* (*véase* Sección 7.4.2). Las prioridades son asignadas a los casos de uso —o a escenarios de ellos— según deberían ser éstos considerados en las iteraciones. Éstos son clasificados a lo largo de varias iteraciones; en las primeras iteraciones se clasifican algunos casos de uso (o escenarios de ellos), pero muchos no han sido identificados en ella y por tanto no han sido cla-

sificados todavía. Todos los casos de uso que se identifican son también clasificados. El resultado de la clasificación es una lista ordenada de casos de uso.

El control de esta clasificación es difícil. Ordenamos los casos de uso de acuerdo con el riesgo que conllevan. Obsérvese que aquí usamos el término *riesgo* en su sentido amplio. Por ejemplo, tener que cambiar la arquitectura del sistema en las últimas fases es un riesgo que queremos evitar. No construir el sistema correcto es otro riesgo que queremos mitigar pronto, encontrando los requisitos apropiados. El proceso de selección está por tanto guiado por los riesgos. Colocamos los riesgos que identificamos en una lista de riesgos, como se discutió en la Sección 12.5.1, y transformamos cada uno de estos riesgos en un caso de uso que mitigará el riesgo cuando sea implementado. Ese caso de uso será introducido entonces en la posición que le corresponde dentro de la clasificación de casos de uso de acuerdo con su nivel de riesgo.

En las primeras iteraciones dedicamos los casos de uso con mayor prioridad a los riesgos relacionados con el ámbito del sistema y con la arquitectura. En las últimas iteraciones seleccionamos nuevos casos de uso para completar la arquitectura ya seleccionada con más funcionalidad. Ponemos más músculos sobre el esqueleto. Los últimos casos de uso son añadidos en algún orden lógico, que se corresponde con el orden utilizado para ordenar la lista de casos de uso. Por ejemplo, los casos de uso que necesitan otros casos de uso para funcionar estarán más abajo en la lista y por tanto serán desarrollados después que los otros. *Véase* en la Sección 5.3 una discusión sobre el enfoque iterativo como un esfuerzo guiado por los riesgos. En las siguientes tres secciones tratamos las tres categorías de riesgos: riesgos específicos, riesgos arquitectónicos y riesgos de requisitos.

12.6.1. Riesgos específicos de un producto particular

Éste es el tipo de riesgo —los riesgos técnicos— que discutimos en la Sección 5.3.1. Cada uno de estos riesgos es transformado en un caso de uso que, una vez implementado correctamente, mitiga el riesgo. Tenemos que identificar estos riesgos uno por uno, pues el tratar con ellos no es formalmente parte del proceso. Con "formalmente parte del proceso" queremos decir que el proceso proporciona un lugar específico en el que tratar con cierto tipo de riesgo. Por ejemplo, como se discute en la Sección 12.6.2, ciertos riesgos arquitectónicos son considerados en la fase de inicio y otros en la fase de elaboración. Los riesgos que no son formalmente parte del proceso requieren ser tratados uno por uno y mitigados antes de que su presencia afecte al proceso de desarrollo.

12.6.2. Riesgo de no conseguir la arquitectura correcta

Uno de los riesgos más serios es el de no construir un sistema que pueda evolucionar suavemente por las fases siguientes o durante su tiempo de vida, es decir, el no establecer una arquitectura flexible. Este riesgo es considerado explícitamente durante las fases de inicio y de elaboración, cuando nos aseguramos de tener la arquitectura correcta y podemos entonces fijarla (excepto cambios menores en la fase de construcción). Esto es lo que queremos decir en el párrafo anterior cuando decíamos que estaba prefijado dentro del Proceso Unificado el encontrar y tratar determinado tipo de riesgos. En este caso, por ejemplo, las fases de inicio y de elaboración tratan con la arquitectura explícitamente.

¿Cómo determinamos cuáles son los casos de uso más importantes para conseguir la arquitectura correcta? ¿Cómo mitigamos el riesgo de no conseguir una arquitectura estable? Buscamos los casos de uso significativos arquitectónicamente, que son los que cubren las principales tareas o funciones que el sistema ha de realizar. Uno se pregunta a sí mismo: ¿por qué construimos este sistema?

La respuesta se encuentra en los casos de uso *críticos* —aquéllos que son más importantes para los usuarios del sistema—. Además, los casos de uso que tienen requisitos no funcionales importantes, como de rendimiento, tiempos de respuesta, etc., entran en esta categoría. Normalmente estos casos de uso ayudan a encontrar el esqueleto del sistema sobre el que añadir el resto de las funciones requeridas (*véase* Sección 7.2.4).

Otras categorías de casos de uso son:

- *Secundarios.* Estos casos de uso sirven de apoyo a los casos de uso críticos. Involucran funciones secundarias, como las de supervisión y compilación de estadísticas operativas. En su mayor parte esta categoría de casos de uso tiene sólo un impacto modesto sobre la arquitectura, aunque pueden todavía necesitar ser desarrollados pronto si, por ejemplo, un cliente tiene un especial interés en ver algún dato de salida, como el coste de la transacción descrito en el ejemplo de la Sección 12.6.3. Éste sería, por tanto, colocado más arriba en la lista porque queremos mitigar el riesgo de no conseguir los requisitos correctos.

- *Auxiliares.* Estos casos de uso no son claves para la arquitectura o para los riesgos críticos. Este nivel de casos de uso raramente entra en juego durante las iteraciones en las fases de inicio y de elaboración, y si lo hace es sólo para completar los casos de uso críticos o importantes.

- *Opcionales.* Pocos casos de uso pueden ser críticos o importantes, puede incluso que éstos no estén siempre presentes. Puede que necesitemos trabajar con ellos porque afecten a la arquitectura cuando están presentes.

Además, queremos estar seguros de que hemos considerado todos los casos de uso que podrían tener algún impacto sobre la arquitectura. No queremos dejar ninguna funcionalidad en la sombra para descubrir demasiado tarde que no tenemos una arquitectura estable. Necesitamos tener una alta cobertura de los casos de uso que puedan afectar la arquitectura. Tener esta alta cobertura es importante, no sólo para encontrar la arquitectura sino para asegurar que podemos predecir con precisión los costes de desarrollo del producto en el primer ciclo. Debemos evitar el riesgo de darnos cuenta demasiado tarde de que no podemos dar cabida a una funcionalidad descubierta recientemente.

Ésta es la razón por la que necesitamos cubrir alrededor del 80 por ciento de los casos de uso en la fase de elaboración. Con "cubrir" queremos decir que entendemos los casos de uso y el impacto que pueden tener sobre el sistema. En la práctica, como promedio, identificamos alrededor del 80 por ciento de los casos de uso y los incluimos en el modelo de casos de uso, pero usualmente no encontramos necesario describirlos todos en detalle. En un proyecto típico podemos encontrar necesario describir sólo partes de los casos de uso. Algunas de estas descripciones pueden ser breves, de sólo unas pocas líneas, si eso es suficiente para aclarar lo que necesitamos saber en esa fase. En proyectos relativamente simples en los que trabajamos sobre los requisitos podemos detallar una parte menor de los casos de uso. En proyectos más grandes y con grandes riesgos podemos encontrar recomendable describir en detalle el 80 por ciento o más de los casos de uso.

12.6.3. Riesgo de no conseguir los requisitos correctos

Otro serio riesgo es no conseguir un sistema que haga lo que los usuarios quieren realmente que haga. Las formas de tratar este riesgo son también parte del proceso. Al final de la fase de elaboración queremos estar seguros de que estamos construyendo el sistema correcto. Este descubrimiento no puede demorarse porque en la fase de construcción el dinero empieza a fluir en grandes cantidades. ¿Cuáles son los casos de uso necesarios para asegurar que estamos desarrollando el sistema correcto para los usuarios? ¿Cuáles son los casos de uso que aseguran que el sistema puede evolucionar en las otras fases de forma que podremos añadir todos los requisitos necesarios para la primera entrega? No podemos dejar ninguna funcionalidad en la sombra. Necesitamos saber que lo que construimos puede crecer.

Por supuesto, la primera parte de la respuesta es hacer el flujo de trabajo de los requisitos bien. Podríamos crear un modelo de negocio (o un modelo de dominio más limitado en algunos casos). La segunda parte de la respuesta es, basándose en las iteraciones iniciales y en la construcción de prototipos, construir el sistema que quieren los usuarios y obtener información sobre él tan pronto como sea posible. Sólo podemos estar seguros de que hemos construido el sistema correcto a partir del uso real.

Ejemplo **Sistema de facturación y pago**

En el sistema de facturación y pago podríamos asumir que el banco decidiera que es muy importante para ellos ganar dinero con sus servicios. Quizás quisiera cobrar una pequeña cantidad de dinero por cada transacción. Incorporar este coste sería una adición de nuevas funciones a los casos de uso fundamentales Solicitar Bienes o Servicios, Confirmar Pedido, Enviar Factura al Comprador y Pagar Factura. Sin embargo, desde el punto de vista del arquitecto la función de cobro puede no ser muy importante para hacer que la arquitectura sea correcta. El cobro puede ser tratado como una extensión de otros casos de uso, por ejemplo, puede combinarse con unos cuantos casos de uso que se encarguen de los cobros. Hasta donde concierne a los desarrolladores, la función de cobro es bastante rutinaria; ya lo han hecho antes. Sin embargo, desde el punto de vista del usuario es extremadamente importante que los casos de uso de los cobros sean implementados correctamente antes de la entrega. Por esta razón son clasificados como de alto riesgo, de forma que sean considerados importantes.

Como consecuencia de esto, cuando considera el orden de las iteraciones, el jefe del proyecto tiene que calibrar la importancia de la función de cobro. Por un lado, si descubre que el cobro en el caso en cuestión es una función simple que no presenta retos importantes a los desarrolladores podría decidir que los desarrolladores no necesitan tratarlo durante la fase de elaboración, y podría retrasarlo hasta la fase de construcción de forma segura. Por otro lado, si detecta que el cobro presenta una serie de problemas internos complejos (separado de otros casos de uso), debería planear encargarse del cobro como parte de una iteración durante la fase de elaboración. Uno de estos "problemas complejos" podría ser la necesidad del cliente de ver el cobro resuelto pronto.

12.7. Recursos necesitados

Uno puede pensar que el plan iterativo del desarrollo de software basado en fases tiene un mérito considerable, pero varias cuestiones pueden importunarnos:

- ¿Cuánto van a costar las fases de inicio y de elaboración, tanto en términos de esfuerzo como en términos de cualificación del personal necesario?
- ¿De dónde va a venir el dinero necesario para pagar estas fases?
- ¿Cuánto tiempo van a durar estas dos fases?
- ¿Por cuántos meses van a retrasar las primeras fases lo que mucha gente considera como el negocio real del desarrollo de software, es decir, la construcción?

12.7.1. Los proyectos difieren enormemente

Por supuesto, no es un secreto que los sistemas de software difieren enormemente en su preparación para empezar el desarrollo. Veamos cuatro ejemplos:

1. Un producto completamente nuevo o sin precedentes es un dominio inexplorado —una tierra virgen—. Nadie sabe mucho sobre lo que va a hacerse o incluso si es posible hacerlo. Hay poca experiencia en la que basarse y tendremos que depender de gente experimentada para hacer conjeturas basadas en su experiencia. Bajo estas circunstancias el que quiere el sistema es responsable de financiar las fases de inicio y de elaboración. Las fases han de ser financiadas casi como si se tratara de investigación, es decir, como si se tratara de un valor añadido. Ni siquiera se sabe lo suficiente como para fijar en las fases de inicio o de elaboración un presupuesto o una planificación. En este tipo de situaciones definir el ámbito, encontrar una arquitectura candidata, identificar los riesgos críticos y hacer el análisis de negocio son tareas de la fase de inicio que consumen mucho tiempo. De forma similar, alcanzar los objetivos de la fase de elaboración, es decir, llevar el proyecto hasta el punto en que se pueda planear la fase de construcción, lleva más tiempo.

2. Un producto de un tipo que ha sido realizado anteriormente en un campo en el cual productos anteriores proporcionan ejemplos pero no componentes reutilizables. Estos productos previos proporcionan una guía para la arquitectura candidata, pero puede llevar unos días asegurarse de que la arquitectura anterior encaja realmente. Bajo estas circunstancias, la fase de inicio será probablemente breve (pocas semanas). Puede requerir sólo un par de personas experimentadas a tiempo completo, pero posiblemente se basará en el conocimiento de otras personas experimentadas que el pequeño equipo del proyecto necesita consultar. Ya que este tipo de producto ha sido realizado antes, es improbable tener grandes riesgos, pero puede ser necesario dedicar algunos días a asegurarse de esto.

3. Hay un producto ya existente, pero éste ha de ser actualizado, por ejemplo pasando de funcionar en un *mainframe* a funcionar en una arquitectura cliente/servidor. Hasta cierto punto, partes del código existente pueden ser encapsuladas y utilizadas en el nuevo sistema. El equipo inicial ha de encontrar una arquitectura candidata, y ya que otras organizaciones han reutilizado productos existentes, el equipo sabe que puede hacerse. Sin embargo, si la organización que propone hacer el trabajo no lo ha hecho antes puede que no se conozca la información relacionada con el coste y con la planificación. Habrá que concebir una línea base para la arquitectura y se tendrá que identificar una interfaz entre el sistema nuevo y el viejo empezando con los casos de

uso y con encontrar los subsistemas —siendo uno de los subsistemas una encapsulación de las partes del sistema existente que no necesitan ser cambiadas.

4. Existen los componentes, ya sea en el mercado o en la misma empresa. La organización de software espera que un porcentaje considerable del nuevo sistema, quizás entre un 50 y un 90 por ciento, pueda construirse a partir de estos componentes, pero puede ser necesario hacer encajar los componentes, lo que requeriría código nuevo. El proyecto necesitará identificar y especificar las interfaces entre los componentes reutilizados y los nuevos y entre los sistemas externos y los usuarios. Desarrollar a partir de componentes lleva tiempo y esfuerzo, y el equipo puede tener riesgos, sin embargo, el desarrollo basado en componentes es más rápido y más barato que cuando se empieza desde cero.

Con estos ejemplos no intentamos definir diferentes categorías, sino que representan más bien posiciones solapadas. Tenemos que pensar, por tanto, en términos de estados de comienzo. ¿Cuánta experiencia tiene nuestra organización en el área de aplicación? ¿Cómo es de grande la base de componentes reusables en la que podemos apoyarnos? ¿Es ésta propuesta algo más que una nueva entrega de un producto ya existente? ¿Mejorará el estado del arte? ¿Es un sistema distribuido (donde sólo hemos realizado trabajos no distribuidos)?

12.7.2. Un proyecto típico tiene este aspecto

A pesar de las incertidumbres que introducen diferentes estados de comienzo, el ciclo de desarrollo inicial de un proyecto de tamaño medio podría tener, aproximadamente, una distribución de esfuerzo y planificación como el mostrado en la Figura 12.4. En general, el desarrollo basado en fases desplaza el trabajo para preparar la arquitectura y mitigar los riesgos al principio del ciclo, en contraposición con el desarrollo basado en el modelo en cascada.

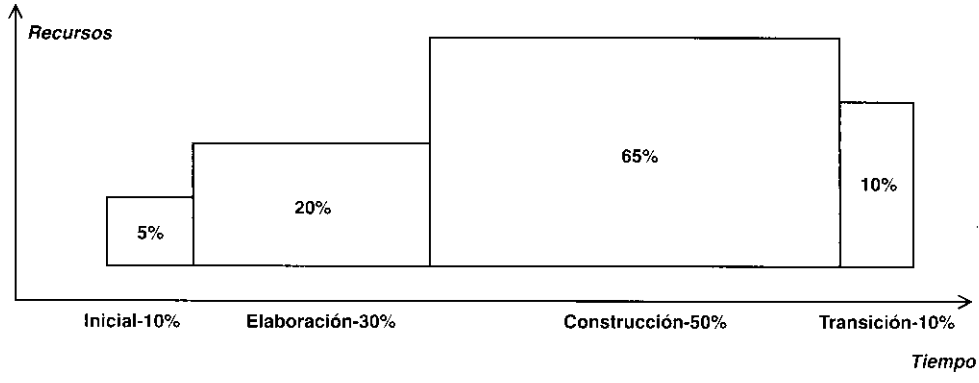


Figura 12.4. El bloque de esfuerzo y de tiempo de planificación más grande queda en la fase de construcción, pero las otras tres fases requieren cantidades significativas de tiempo y de esfuerzo.

12.7.3. Los proyectos más grandes tienen mayores necesidades

¿Cuál es el resultado de suponer un proyecto más grande y más complejo —uno con, por ejemplo, funcionalidad nueva, arquitectura distribuida u operaciones en tiempo real— que usa

también nueva tecnología? Probablemente, tendremos que realizar un mayor número de iteraciones y tendremos que poner más tiempo y esfuerzo en las fases de inicio y de elaboración. Por tanto, estas dos fases pueden crecer como se muestra en la Figura 12.5.

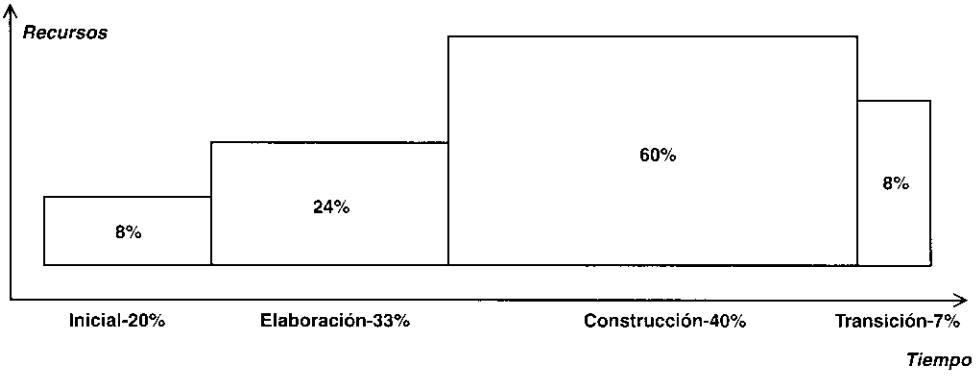


Figura 12.5. Bajo las condiciones impuestas en un proyecto más difícil se realiza más trabajo en las primeras fases, en las que se gasta más tiempo.

Nos apresuramos a enfatizar que los porcentajes que aparecen en estas figuras son hipotéticos, por lo que no deberían ser aplicados arbitrariamente a proyectos reales. Con estos porcentajes intentamos mostrar que cuantas más incógnitas presente un proyecto, más tiempo y esfuerzo tendrá que emplearse en las fases de inicio y de elaboración. Podemos ver cómo en el ejemplo dedicamos más tiempo a las fases de inicio y de elaboración que a las otras dos fases. Sin embargo, con el esfuerzo no ocurre lo mismo, de forma que no necesitamos incrementar los recursos en la misma proporción, aunque también aumenten. Al mismo tiempo, el esfuerzo y el tiempo para el proyecto aumentan como un todo.

La utilización de marcos de trabajo acorta la construcción significativamente pero tiene menos efecto sobre las primeras fases. Las fases de inicio y de elaboración serán más largas si consideramos reutilizar, pero las funciones que están ya disponibles como marcos de trabajo no necesitan ser analizadas ni diseñadas, por lo que globalmente el proyecto necesita menos tiempo y esfuerzo.

12.7.4. Una nueva línea de productos requiere experiencia

En la mayoría de los casos, y con seguridad para sistemas nuevos o difíciles, el equipo ha de adquirir más información de la que posee. La fuente natural de esta información son las personas informadas en el campo del sistema propuesto. Incluso cuando se dispone de requisitos detallados, el equipo necesita estas entrevistas para encontrar la arquitectura y centrarse en los riesgos. Sin embargo, las personas informadas son la mitad de la batalla. Los usuarios normales pueden conocer tan sólo su parte del proceso completo; o puede que éstos no sepan lo que los sistemas de computación son capaces de hacer.

Un fallo común cuando se comienza el trabajo en una nueva línea de productos es intentar hacerlo sin *reutilizar el conocimiento*. Como la mayor parte del conocimiento sobre la forma en que funciona la compañía reside en las mentes de las personas más que en los documentos (los cuales no se leen de todos modos), reutilizar el conocimiento básicamente se reduce a "reutilizar" a la gente experimentada. Este fallo se descubre a sí mismo cuando se asigna personal

nuevo al equipo inicial del proyecto en lugar de asignar personal experimentado en las formas de hacer las cosas de la compañía, cuando no en la nueva línea de productos misma.

Cuando una compañía planea desarrollar una nueva línea de productos ésta sabe en una parte de su mente colectiva que este trabajo requiere su gente más competente y experimentada. Al mismo tiempo, sin embargo, en otra parte de su mente colectiva está pensando que este tipo de personas es muy importante para mantener el negocio actual funcionando. Tienen que mantener satisfechos a sus clientes, ya que la compañía ha de mantener fluyendo la fuente de dinero existente.

No hay una respuesta fácil a este dilema. Los ejecutivos para quienes esta gente experimentada está trabajando se encuentran entre estos dos objetivos —obviamente, tanto la actual línea de productos como la nueva línea de productos son importantes—. En la práctica, muchos de los altos ejecutivos han estado ligados a la línea actual durante muchos años, y están por tanto psicológicamente casados con ella. A menudo, muchos de ellos son reacios a arriesgar el negocio actual dejando marchar al nuevo proyecto a su gente clave. Como consecuencia de esto, el nuevo equipo tendrá poca gente con experiencia, y puede que éstos no sean los líderes más fuertes —administrativa o técnicamente hablando—; puede que ésta sea la gente que los ejecutivos estaban deseando dejar marchar. Estos nuevos líderes tendrán entonces que completar sus equipos con una gran cantidad de gente nueva, a menudo directamente de la universidad.

Además de su inexperiencia, la gente nueva traerá consigo una particular actitud cultural. Éstos tienden a ver viejo, como pasado de moda, todo lo que la compañía ha estado haciendo. Para ellos sólo es bueno lo nuevo. Los recién llegados no aciertan a apreciar la importancia de las materias que no aprendieron en la universidad, como los métodos de producción y la administración del ciclo de vida.

Hemos observado que las compañías a menudo fallan al elegir el personal para el desarrollo de una nueva línea de productos; generalmente este personal no tiene el talento necesario para desarrollarla de forma que ésta esté lista para el mercado en el momento apropiado. Normalmente, las deficiencias no se corrigen hasta la segunda generación del producto, de forma que, en esencia, la primera versión se convierte en una prueba para preparar el terreno, aunque puede que ésta no haya sido la intención original de la compañía.

12.7.5. El pago del coste de los recursos utilizados

Mantener el equipo que trabaja en las dos primeras fases, aunque sea pequeño, cuesta dinero. De hecho, la dificultad de conseguir fondos para costear estas dos primeras fases ha sido una de las causas que han llevado a las organizaciones fabricantes de software a acometer la fase de construcción prematuramente, dando lugar a veces a fallos bien conocidos. ¿De dónde han de venir estos fondos?:

- En el caso de una organización fabricante de software que produce un producto para vender, los fondos son parte de los "gastos generales" y, como tales, están bajo el control de la dirección. A largo plazo, sin embargo, estos fondos proceden de los clientes, es decir, estas organizaciones han de establecer los precios actuales lo suficientemente altos como para cubrir el coste de desarrollar futuros productos.
- En el caso de una organización fabricante de software que produce un producto para un cliente dentro de la misma empresa, el coste de las primeras dos fases es parte de los "gastos generales" de la compañía, de los fondos transferidos a ella por el cliente o de los

fondos asignados a ella por la alta dirección de la empresa. Lo que indican las dos últimas formas de financiación es que la dirección fuera de la organización fabricante de software ha de entender el valor de las primeras fases y asignar fondos para financiarlas.

- En el caso de una organización fabricante de software que produce un producto para una corporación distinta, el coste de las dos primeras fases puede venir de sus gastos generales propios. Sin embargo, obtener este dinero de los gastos generales sólo es posible si ya se ha invertido en desarrollos parecidos para proyectos anteriores. Si el proyecto encaja dentro del negocio normal de la organización este tipo de fondos puede ser suficiente, pero si el proyecto propuesto es más arriesgado de lo normal entonces el cliente debería contribuir a la financiación de las dos primeras fases.

La realidad es que en las primeras dos fases se lleva a cabo un trabajo importante, y que realizarlo lleva tiempo y cuesta dinero. Además de los fondos el llevarlas a cabo requiere la colaboración del cliente. Esta cooperación —en la que se proporciona gente de la que la compañía de software obtiene la información que necesita— también cuesta dinero (aunque puede que ésta no sea incluida formalmente en las cuentas).

12.8. Evaluar las iteraciones y las fases

Si se quieren conocer completamente los beneficios de la forma de trabajo iterativo, el proyecto debe evaluar la parte del trabajo que se ha completado al final de cada iteración o fase. El jefe de proyecto es responsable de esta evaluación. Este análisis se hace no sólo para evaluar la iteración en sí, sino para impulsar estos otros dos objetivos:

- Reconsiderar el plan de la iteración siguiente a la luz de lo que el equipo ha aprendido realizando ésta y realizar los cambios necesarios en éste.
- Modificar el proceso, adaptar las herramientas, prolongar la preparación y realizar otras acciones como sugiere la experiencia de la iteración evaluada.

El primer objetivo de una evaluación es examinar lo que ha sido realizado en términos del criterio de evaluación actual. El segundo objetivo es el de comparar el progreso realizado con el plan de la iteración o del proyecto:

- ¿Avanza el proyecto dentro del presupuesto y siguiendo la planificación?
- ¿Está alcanzando los requisitos de calidad, de acuerdo con los resultados de las pruebas o la observación de los prototipos, componentes y construcciones?

Idealmente, el proyecto cumplirá los criterios. El jefe del proyecto distribuye los resultados de la evaluación a las personas relacionadas con el proyecto y archiva el documento correspondiente. Este documento no se actualizará, cuando se informe de la siguiente evaluación se creará un nuevo documento.

12.8.1. Criterios no alcanzados

Las evaluaciones raramente van tan bien. Frecuentemente alguna iteración no alcanza los criterios satisfactoriamente. El proyecto puede tener que prolongar este trabajo durante la siguiente iteración (o hasta la iteración posterior apropiada). Este trabajo puede implicar:

- Modificar o extender el modelo de casos de uso.
- Modificar o extender la arquitectura.
- Modificar o extender los subsistemas desarrollados hasta entonces.
- Buscar otros riesgos.
- Incorporar ciertas habilidades al equipo.

También es posible que, simplemente, se necesite más tiempo para poder llevar a cabo el plan existente. Si éste es el caso, el proyecto podría extender la planificación de la primera iteración, aunque si esto ocurre entonces se debería especificar una fecha límite firme.

12.8.2. Los criterios mismos

También es importante tener en cuenta los criterios de evaluación mismos. El equipo puede haber establecido los criterios en un momento en que todavía no tenía disponible toda la información relevante. A lo largo de la iteración podría haber descubierto necesidades adicionales o que las necesidades que enumeró inicialmente se han mostrado innecesarias. Por tanto, los evaluadores podrían tener que cambiar los criterios, y no sólo comprobar si se alcanzan estos criterios.

12.8.3. La siguiente iteración

Un **hito principal** (Apéndice C) marca la terminación de una fase, el punto en el que no sólo el equipo del proyecto sino todas las personas involucradas en él, en particular las autoridades que proporcionan los fondos y los representantes de los usuarios, coinciden en que el proyecto ha alcanzado el criterio del hito y que está justificado el paso a la siguiente fase.

Sobre la base de la evaluación, el jefe de proyecto (asistido por algunas de las personas que trabajaron en la iteración o en la fase y por algunas de las personas que participarán en la siguiente fase) hace lo siguiente:

- Determina que el trabajo está listo para pasar a la siguiente iteración.
- Si se necesita rehacer algún trabajo asigna en cuáles de las siguientes iteraciones debería realizarse.
- Planea en detalle la siguiente iteración.
- Actualiza el plan, en menos detalle, de las iteraciones posteriores a la siguiente.
- Actualiza la lista de riesgos y el plan del proyecto.
- Compara el coste y la planificación de la iteración con los planeados.

Observamos que, en el desarrollo basado en componentes, el número de líneas de código escritas no es un indicador fiable del progreso. Puesto que un desarrollador puede reutilizar bloques (subsistemas, clases y componentes) ya diseñados, se puede avanzar mucho sin escribir mucho código.

12.8.4. Evolución del conjunto de modelos

Una característica clave del desarrollo iterativo en fases es la evolución del conjunto de modelos. Esta evolución contrasta con lo que ocurre en el modelo en cascada, donde imaginamos que los requisitos se completan primero, a continuación el análisis y así sucesivamente. En el desarrollo iterativo, los modelos crecen juntos a través de las fases. En las primeras iteraciones unos modelos van por delante de otros, por ejemplo, el modelo de casos de uso va por delante del modelo de implementación. En lugar de que un modelo evolucione casi independientemente del siguiente modelo pensamos en términos de un estado del sistema entero que evoluciona a un estado más avanzado del sistema entero, como representamos en el diagrama de la Figura 5.7. Cada iteración —quizás cada construcción dentro de una iteración— representa un avance en el estado del sistema completo, el cual se refleja en el movimiento gradual hacia la compleción del conjunto de modelos. Considerar el grado en que esta evolución ha progresado en cada evaluación será un indicador importante para el grupo evaluador.

En el siguiente capítulo volvemos a los comienzos del proyecto y consideramos la fase de inicio por sí sola.

Capítulo 13

La fase de inicio pone en marcha el proyecto

El objetivo global de la fase de inicio es poner en marcha el proyecto. Antes de esta fase, puede que tenga simplemente una idea interesante rondándole la cabeza. La idea puede ser sugestiva, si se trata de algo totalmente nuevo y en un campo nuevo. O puede ser sosegada, si consiste en una nueva versión de un producto ya existente. La fase de inicio puede tratarse de algo tan sencillo como una persona que recopile una declaración de objetivos, esboce una arquitectura usando diversos diagramas, y desarrolle un análisis de negocio razonable. O puede ser tan compleja como un proyecto de investigación completo. Lo importante es que no consideremos la fase de inicio como algo que ha de desarrollarse de una única forma. Después del inicio, incluso si el sistema es nuevo, se habrá delimitado el problema que quiere resolver, y se habrá hecho lo necesario para ver, con cierto grado de confianza, que es a la vez posible y deseable desarrollar un sistema.

Por supuesto, esta confianza no es sólo suya. El trabajo realizado durante la fase de inicio, da al cliente (o a sus representantes), a la organización de desarrollo, y al resto de personas involucradas en el proyecto, la seguridad de que el arquitecto y los encargados del desarrollo serán capaces de mitigar los riesgos críticos, formular una propuesta de arquitectura —una arquitectura candidata— y hacer el análisis inicial de negocio.

13.1. La fase de inicio en pocas palabras

El objetivo de la fase de inicio es desarrollar el análisis de negocio hasta el punto necesario para justificar la puesta en marcha del proyecto. Para desarrollar este análisis de negocio, primero te-

nemos que delimitar el alcance —el ámbito— del sistema propuesto. Necesitamos hacer esto para discernir qué es lo que debemos cubrir con el proyecto de desarrollo. Necesitamos saber cuál es el ámbito para comprender qué debe cubrir la arquitectura. Lo necesitamos para definir los límites dentro de los cuales debemos buscar riesgos críticos. Lo necesitamos para delimitar las estimaciones de coste, agenda y recuperación de la inversión —los ingredientes del análisis de negocio.

Lo que intentamos es hacernos una idea de la arquitectura, para asegurarnos de que existe una arquitectura que puede soportar el ámbito del sistema. Eso es lo que entendemos por "arquitectura candidata".

También intentamos anticiparnos a un fiasco. Numerosos proyectos difíciles quedaron empantanados porque se tropezaron con riesgos críticos, a lo mejor en fases tan avanzadas como en la integración del sistema y las pruebas, y no fueron capaces de mitigar estos riesgos con el presupuesto y el tiempo que aún quedaba disponible. La incertidumbre, o el riesgo, existe, nos guste o no. Nuestro único recurso es mitigar los riesgos pronto, reducir el ámbito del proyecto, o aumentar el tiempo y los recursos (es decir, los costes) para evitar riesgos que no se puedan mitigar.

Intentamos desarrollar el análisis inicial de negocio —para considerar el sistema previsto en términos económicos, es decir, estimaciones iniciales de costes, agenda y recuperación de la inversión. Nos preguntaremos:

- ¿Compensarán las ganancias procedentes del uso o la venta del producto software, los costes de su desarrollo?
- ¿Llegará el producto al mercado (o a su aplicación interna) a tiempo de obtener esas ganancias?

Intentamos dar apoyo al equipo del proyecto con un entorno de desarrollo —proceso y herramientas. Por supuesto, el entorno de desarrollo de una organización software es producto de años de esfuerzo, la mayor parte del cual se realiza antes de que se inicie un proyecto determinado. A pesar de esto, adaptaremos el Proceso Unificado al tipo de sistema en desarrollo y al nivel de competencia de la organización de desarrollo que va a realizar el trabajo. Éste es el tipo de cuestiones que la fase de inicio intenta abordar.

13.2. Al comienzo de la fase de inicio

Comience a planificar, extienda la descripción del sistema previsto, y empiece a reunir sus criterios de evaluación.

13.2.1. Antes de comenzar la fase de inicio

Incluso antes de empezar la fase de inicio, ya se tiene alguna noción de lo que se va a hacer. Alguien —una organización cliente, la gente de ventas, o incluso alguien de la propia organización de desarrollo— tuvo una idea y la justificó lo suficiente como para poner algo en marcha. La cantidad de trabajo realizada antes del comienzo de la fase de inicio varía notablemente, recorriendo todo un espectro en el que podemos distinguir tres puntos:

- Organizaciones software que desarrollan productos para la venta general. La cantidad de información de que se dispone para empezar puede ser muy sustanciosa. Probablemente, la gente de ventas y de dirección han dedicado un considerable esfuerzo a pensar en el pró-

ximo producto, y han encargado a alguien de desarrollo que realice estudios exploratorios. En otras palabras, ya han realizado parte del trabajo de la fase de inicio.

- Organizaciones software que producen sistemas para otras unidades de la misma empresa, es decir, la típica organización de desarrollo interno. Nos encontramos, a *grosso modo*, con dos situaciones. En la primera, un departamento establece la necesidad de un sistema software, y encarga a la organización software que lo desarrolle. El departamento solicitante facilita una descripción de lo que tiene en mente, pero tiene poco conocimiento de las ramificaciones software de su petición. En otras palabras, la organización software dispone de poca información cuando la fase de inicio se pone en marcha. En la segunda, la dirección ha detectado la necesidad de un sistema cuyo ámbito es toda la empresa, y ha puesto a trabajar a un grupo de ingeniería de negocios para determinar en qué debería consistir. En este caso, la fase de inicio puede comenzar con un buen conocimiento de los requisitos.
- Organizaciones software que producen sistemas para un cliente. A menudo, la petición inicial de propuestas contiene considerables detalles, quizás muchas páginas de requisitos. En otros casos, si la relación con el cliente es más informal, éste puede haber facilitado sólo una sucinta descripción de sus necesidades.

Si se trata de hacer una nueva versión de un producto, gran parte del trabajo de planificación de la primera iteración del nuevo producto habría sido hecha en la última iteración del ciclo anterior. Si, por el contrario, se trata de un producto fundamentalmente nuevo, quienes lo originaron pueden querer ver su idea desarrollada más profundamente. Pueden haber hecho una estimación de cuánto trabajo sería necesario, obtenido fondos para cubrir las fase de inicio, y establecido una agenda.

13.2.2. Planificación de la fase de inicio

Al comienzo de un proyecto, es posible que se encuentre en un dilema. La gente bienintencionada le aconseja que planifique, pero no se tiene mucha información sobre la que basar un plan. Sólo se reconoce lo que tiene que ser planificado si sabe lo que es necesario hacer. De esto nos ocuparemos en las Secciones 13.3 y 13.4. Mientras tanto, podemos empezar por llevar a cabo los siguientes pasos:

- Reunir la información recogida antes de que el proyecto comenzase.
- Organizarla de forma que pueda ser utilizada.
- Reunir a un pequeño grupo de gente que sepa cómo utilizarla.
- Descubrir lo que falta, no en términos de las cuatro fases, sino en términos de los objetivos altamente limitados de la fase de inicio.

En otras palabras, limite el esfuerzo a lo que sea necesario para cumplir los objetivos clave de esta fase, que están resumidos en la Sección 13.1. Entonces, desarrolle un plan provisional para clarificar los requisitos que conciernen a esos objetivos iniciales y para detallarlos en los correspondientes casos de uso. Planifique la creación de una arquitectura candidata. Proyecte desarrollar esta arquitectura sólo hasta el punto de establecer que el proyecto es factible, normalmente basta con esbozar las vistas de la arquitectura. Tan pronto como pueda, trate de constatar que va a necesitar una iteración, y en algunos casos, dos.

Tenga en cuenta que el plan inicial es *provisional*. Según vaya recopilando más información, modifique el plan para acomodar lo que vaya descubriendo. Trate, por supuesto, de completar la

fase de inicio dentro de los límites de agenda y coste que la dirección (o el cliente) establecieron inicialmente. Debido a que estas cifras tuvieron forzosamente que fijarse en base a un conocimiento limitado, mantenga a la dirección (y a todos los involucrados en el proyecto) informados del progreso durante este fase. Puede ser necesario cambiar estas cifras.

13.2.3. Ampliación de la descripción del sistema

Al comienzo del ciclo de vida, el equipo puede disponer de poco más de una página de descripción del sistema previsto. Esta descripción contendrá una lista de características (*véase* la Sección 6.3), algo de información acerca del rendimiento, apenas conocimiento sobre los riesgos con que los encargados del desarrollo pueden encontrarse, y quizás una vaga referencia a una posible arquitectura (puede ser simplemente la frase "cliente/servidor" o algo así), y unas cifras redondas estimando los aspectos económicos (como 10 millones de euros y dos años).

Este equipo inicial puede incluir al jefe de proyecto, el arquitecto, uno o dos desarrolladores con experiencia, un ingeniero de pruebas (en particular si los problemas de pruebas se presentan de gran importancia), y probablemente representantes del cliente o los usuarios. El primer paso es ampliar la descripción del sistema hasta el grado necesario para guiar la fase de inicio.

Esto es fácil de decir, pero difícil de llevar a cabo. Por esto en el equipo tendría que haber representantes de los intereses involucrados. Por esto tendría que haber gente con experiencia. Por esto estas personas deberían ser capaces de integrar diferentes puntos de vista, no sólo de ponerse de acuerdo en un punto intermedio. Por esto lleva algo de tiempo pensar en el desarrollo de esta difícil tarea. Tenga en cuenta que no estamos buscando el consenso; estamos buscando la mejor respuesta. Las respuestas correctas vienen de un líder, de la persona responsable de gestionar el dinero invertido en esta tarea, pero también de un líder asesorado por especialistas bien informados.

En el caso de un producto similar a otro hecho con anterioridad, puede bastar simplemente con determinar esta similitud, y hacer esto puede llevar sólo unos pocos días. Algunas fases de inicio tienen una duración de un día (por ejemplo, un segundo ciclo para un producto sencillo ya existente). Para un proyecto novedoso puede llevar varios meses.

En el caso de un producto altamente original, la fase de inicio puede ser larga y costosa.

13.2.4. Establecimiento de los criterios de evaluación

En el momento en que el jefe de proyecto dispone de suficiente información para presentar el plan detallado de la primera iteración, también presenta los criterios de evaluación que indican cuándo la iteración ha alcanzado sus objetivos. El plan "detallado" de la primera iteración puede ser bastante vago, debido a la escasez de información. Si éste es el caso, los criterios de evaluación pueden ser bastante generales. Según avanza la iteración, y se saben más cosas, el jefe de proyecto será capaz de refinar los criterios. Sin embargo, ofrecemos algunos criterios generales de evaluación para los cuatro objetivos de la fase de inicio:

Decidir el ámbito del sistema No cabe duda de que la descripción inicial proporciona alguna noción sobre el ámbito del sistema, pero lo que no hace es definirlo con precisión. En la fase de inicio, el proyecto traza una línea precisa alrededor de lo que ha de estar dentro del sis-

tema propuesto y de lo que está fuera. Se identifican entonces los actores externos con los que va a interactuar el sistema, que pueden ser otros sistemas o personas, y se especifica a alto nivel la naturaleza de esta interacción. Los criterios incluyen:

- ¿Está claro lo que va a formar parte del sistema?
- ¿Se han identificado todos los actores?
- ¿Se ha expuesto la naturaleza general de las interfaces con estos actores (interfaces de usuario y protocolos de comunicación)?
- ¿Puede, lo que está incluido en el ámbito, constituir por sí mismo un sistema que funcione?

Resolver ambigüedades en los requisitos necesarios en esta fase Los "requisitos" al comienzo de la fase de inicio pueden variar desde una visión general a muchas páginas de descripción textual. Sin embargo, estos requisitos iniciales suelen contener ambigüedades. Los aspectos para la evaluación son:

- ¿Se han identificado y detallado los requisitos (tanto funcionales como no funcionales) del limitado número de casos de uso necesarios para alcanzar los objetivos de esta fase?
- ¿Se han identificado y detallado los requisitos adicionales (*véase* la Sección 6.7)?

Determinar una arquitectura candidata Por una cuestión de experiencia, quienes participan en la fase de inicio son capaces de centrarse bastante rápidamente en las funciones que son nuevas o que requieren un rendimiento novedoso. De entre ellas, deben seleccionar las pocas que puedan poner en peligro el desarrollo de todo el sistema. Para estas pocas funciones, deben desarrollar como mínimo una arquitectura factible. Los criterios de evaluación son:

- ¿Satisface esta arquitectura las necesidades de los usuarios?
- ¿Es verosímil que funcione? (Considere este criterio en función del punto de desarrollo al que se ha llegado con la arquitectura candidata. Ya que no se ha preparado un prototipo, por ejemplo la descripción de la arquitectura candidata es juzgada en función de su "promesa" de funcionar.)

Para responder a esta pregunta, tenemos que considerar varias cuestiones. ¿Puede utilizar de forma apropiada la tecnología (bases de datos, redes de comunicación, etc.) sobre la que va a ser construida? ¿Puede ser eficiente? ¿Puede explotar los recursos existentes? ¿Puede ser fiable y tolerante a fallos? ¿Será robusta y flexible al cambio? ¿Evolucionará fácilmente si se añaden requisitos?

Mitigar los riesgos críticos Riesgos críticos son aquéllos que, si no son mitigados, pondrían en peligro el éxito del proyecto. Los aspectos a evaluar incluyen:

- ¿Se han identificado todos los riesgos críticos?
- ¿Se han mitigado los riesgos identificados o existe un plan para mitigarlos?

"Mitigado" no quiere decir que un riesgo crítico haya sido enteramente eliminado en esta fase. Podría significar que el cliente está dispuesto a modificar los requisitos relacionados con el riesgo, en vez de afrontar la posibilidad de que el proyecto fracase. Podría significar que el equipo del proyecto vislumbra un camino que evita el riesgo, aunque no perfilará los detalles hasta una fase posterior. En otros casos, podría significar que el equipo del proyecto vislumbra una forma para reducir la probabilidad del riesgo, o para minimizar su gravedad si ocurre. Podría significar redactar un plan de emergencia para tratar con su ocurrencia. Ninguna de estas cuestio-

nes lleva consigo un discernimiento simple y mecánico, al menos en situaciones extremas. El objetivo principal de la fase de inicio es alcanzar un punto en el proceso de desarrollo de software en el cual el jefe de proyecto, los ejecutivos y los responsables financieros (incluidos los clientes) puedan evaluar estos criterios.

Juzgando el valor del caso de negocios inicial El asunto de las evaluaciones ¿es el caso de negocios inicial lo suficientemente bueno como para justificar que sigamos hacia adelante con el proyecto?

El caso de negocios inicial abarca otra área del proyecto que exige un criterio riguroso por parte de los responsables. En un área conocida, probablemente se tendrá un buen número de casos de negocios hasta el final de la fase de inicio. Pero en un área nueva, es difícil y puede que contemos con un rango no demasiado amplio de posibilidades en las que basar nuestro criterio.

13.3. Flujo de trabajo arquetípico de una iteración en la fase de inicio

En la fase de inicio se desarrollan tres conjuntos de actividades. Una es planificar las iteraciones, como se describe en las Secciones 12.4 a 12.7 y en la Sección 13.2.3; la segunda son los cinco flujos de trabajo fundamentales, sobre los que se discute brevemente en la Sección 13.3.1 y de forma extensa en la Sección 13.4; y la tercera es la selección del entorno de desarrollo adecuado para el proyecto, como se describe en la Sección 13.3.2. Además, la Sección 13.5 describe el análisis inicial de negocio, y la Sección 13.6 incluye la evaluación de esta fase.

13.3.1. Introducción a los cinco flujos de trabajo fundamentales

El principal objetivo de la fase de inicio es realizar el análisis de negocio —el decidir, desde un punto de vista de negocio, si seguir adelante con el proyecto—. Para lograr este objetivo, necesitamos determinar el ámbito del sistema propuesto, esbozar una arquitectura, identificar los riesgos críticos para el éxito del proyecto y perfilar un plan para mitigarlos. Si estamos proponiendo un nuevo tipo de sistema, será necesario hacer una demostración del sistema y su uso construyendo un prototipo, normalmente uno que desecharemos posteriormente. También pueden utilizarse prototipos de forma selectiva para gestionar y mitigar el riesgo: identificar áreas de alto riesgo y construir un prototipo de partes clave del sistema de funcionalidad difícil, o de rendimiento conocido, o que presenten problemas de fiabilidad. Por ejemplo, en un sistema de intercambio financiero tolerante a fallos, se puede querer construir un prototipo del mecanismo de recuperación de fallos ya en un momento muy temprano del proceso de desarrollo.

La mayor parte del trabajo de la fase de inicio se lleva a cabo en el primer flujo de trabajo, el de requisitos, como muestra la Figura 13.1, seguido de algo de trabajo en los flujos de análisis y diseño. Hay poco trabajo que realizar en los flujos finales, los de implementación y pruebas. En esta fase, nuestro interés reside en mostrar nuevos conceptos, no en asegurar que los prototipos exploratorios, si existen, funcionan correctamente hasta en el mínimo detalle.

El analista de sistemas identifica los casos de uso y actores que definen el ámbito del sistema. El arquitecto establece prioridades entre estos casos de uso, seleccionando aquéllos que son relevantes para la arquitectura candidata, y prepara una descripción inicial de dicha arquitectura. Los

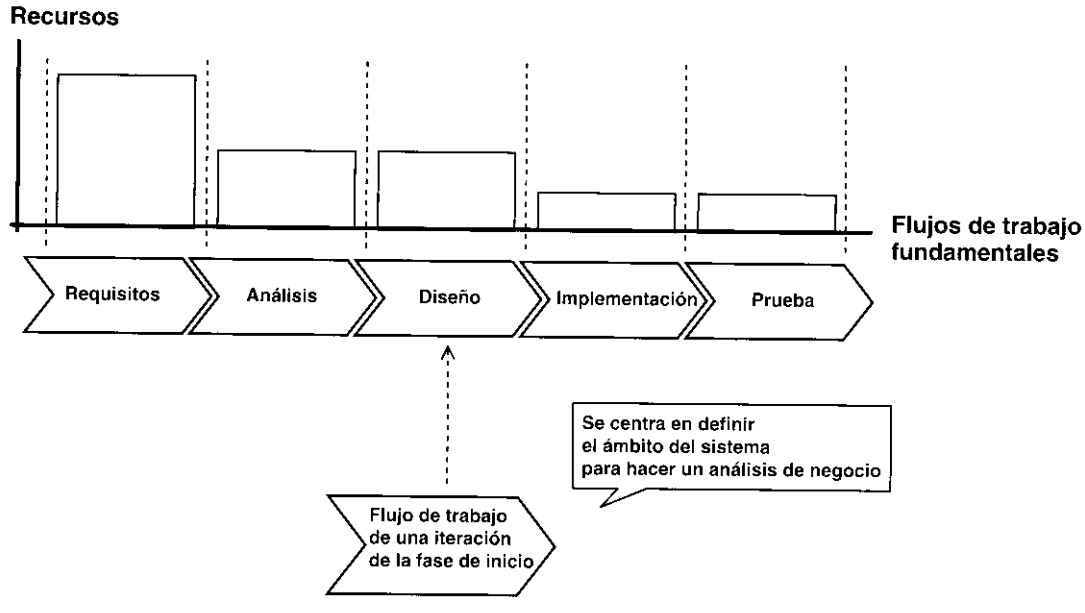


Figura 13.1. El trabajo de una iteración en la fase de inicio transcurre a través de los cinco flujos de trabajo fundamentales. Los tamaños relativos de los rectángulos sólo indican qué flujos de trabajo requieren más atención.

encargados de especificar los casos de uso detallan algunos caminos a través de los casos de uso identificados. Los casos de uso que se describen detalladamente son aquéllos que son relevantes para la comprensión del ámbito del sistema y la arquitectura candidata, y para entender los riesgos críticos, es decir, aquéllos relevantes para realizar el análisis inicial de negocio. Un resultado de esta tarea es el primer modelo de casos de uso, si el proyecto es novedoso, o una nueva versión del modelo de casos de uso, si está siendo modificado un sistema ya existente. También podemos preparar una lista clasificando los casos de uso, como se discutió en el Capítulo 12.

En el flujo de trabajo de análisis, creamos un modelo de análisis inicial para la mayoría de los casos de uso y escenarios de casos de uso con los que tratamos durante la fase de inicio. El modelo de análisis es importante para comprender claramente los casos de uso. Es también importante comprender qué es lo que subyace bajo nuestra primera aproximación a la arquitectura.

Si el sistema es nuevo o novedoso, el equipo de la fase de inicio puede preparar, de forma rápida, un prototipo exploratorio para mostrar las ideas clave. La intención de tal prototipo es la de mostrar los conceptos involucrados, no la de evolucionar hacia la implementación final. En otras palabras, probablemente se trate de un prototipo desechable. En algunos casos, determinar que existe un algoritmo para llevar a cabo una nueva computación puede ser suficiente, suponiendo que el equipo vea que los ingenieros de componentes no tendrán problemas a la hora de implementar el algoritmo mediante una serie de componentes capaces de lograr el rendimiento necesario.

La iteración puede acabarse con la "descripción de la arquitectura candidata", incluyendo bocetos de las vistas de los modelos, tan pronto como el jefe de proyecto determine que la arquitectura candidata se muestre capaz de funcionar, y que el riesgo es menos que crítico o mitigable. Éste sería el caso en el que la arquitectura está cimentada en la experiencia pasada del proyecto y de las personas involucradas en él.

13.3.2. Ajuste del proyecto al entorno de desarrollo

El entorno de desarrollo consiste en un proceso, las herramientas para llevarlo a cabo, y una serie de servicios para los proyectos. Incluye la configuración y mejora del proceso, la selección y adquisición de herramientas, los servicios técnicos, formación y asesoría.

Las herramientas incluyen las que sirven de apoyo a los flujos de trabajo fundamentales: requisitos, análisis, diseño, implementación y pruebas, así como las herramientas administrativas para la gestión de proyectos (planificación, estimación, seguimiento), gestión de la configuración y los cambios, generación de documentación, y herramientas *on-line* relacionadas con el propio proceso. Además de estas herramientas, que a menudo son de propósito general y se obtienen de proveedores externos, un proyecto puede desarrollar herramientas especiales, ya sea dentro del proyecto o de la empresa, para servir de apoyo a sus necesidades específicas. Los servicios se refieren a la administración del sistema, copia de seguridad y telecomunicaciones.

En la fase de inicio, comienza la tarea de hacer que el entorno de desarrollo de la organización software sirva de apoyo al proyecto. Este trabajo continúa en la fase de elaboración, cuando el proyecto se encuentra con una mayor necesidad de apoyo por parte de herramientas y otros servicios. La organización del proyecto gestiona su entorno de desarrollo en paralelo con el resto del trabajo a realizar en esta fase.

13.3.3. Identificación de los riesgos críticos

En el Capítulo 12, dedicamos las Secciones 12.5 y 12.6 a identificar y mitigar los riesgos que afectan al desarrollo. En la fase de inicio, son los riesgos *críticos* los que tenemos que identificar y mitigar o planificar cómo mitigar. Un riesgo crítico es uno que haría el proyecto inviable. En la Sección 13.2.4 discutimos lo que significa mitigar un riesgo crítico. Es de la máxima importancia encontrar los riesgos de esta magnitud durante la fase de inicio. Si encontramos un riesgo de este tipo y no podemos hallar una forma de mitigarlo o un plan de emergencia para contenerlo, debemos de considerar el abandono del proyecto.

13.4. Ejecución de los flujos de trabajo fundamentales, de requisitos a pruebas

En esta sección del capítulo, describimos con más detalle lo que necesitamos hacer en esta fase. Esta descripción cubre el trabajo de la fase de inicio para un proyecto novedoso, es decir, un producto nuevo, partiendo de un papel en blanco. Continuar con el desarrollo de un producto existente será normalmente mucho más sencillo.

Cada iteración sigue la estructura de un pequeño ciclo de vida en cascada, pasando a través de los cinco flujos de trabajo fundamentales, desde el de requisitos hasta el de pruebas. En la Parte II, dedicamos un capítulo a cada uno de estos flujos de trabajo fundamentales —con dos capítulos dedicados al primero de ellos, requisitos—. En esta sección vamos a dedicar una subsección a cada uno de los cinco flujos de trabajo, pero considerándolos en el contexto de la fase de inicio. Para mostrar dónde reside el énfasis, hemos delimitado dos grupos de actividades en la Figura 13.2. Uno de estos grupos incluye las actividades relativas a la determinación del ámbito del sistema, y el otro, a la comprensión de la arquitectura candidata.

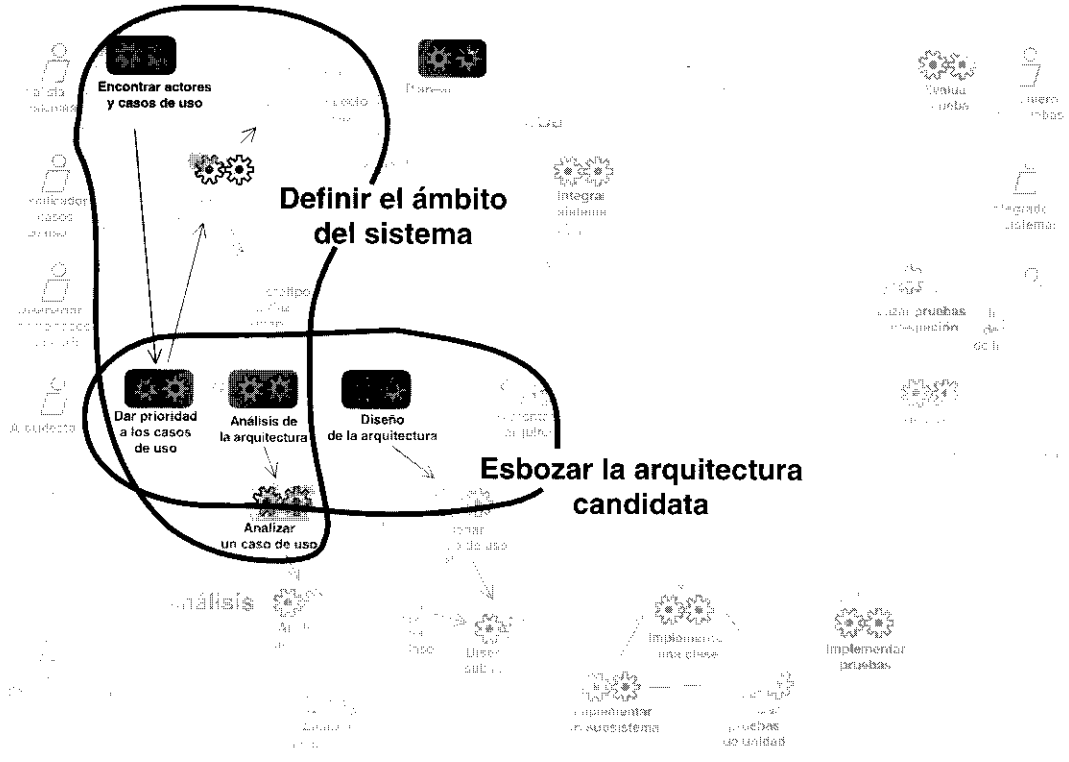


Figura 13.2. En esta figura se resaltan las actividades principales de la fase de inicio: definir el ámbito del sistema y esbozar la arquitectura candidata. A pesar del aparente detalle, se muestra sólo un esquema de los trabajadores y artefactos descritos en los Capítulos 6 a 11.

Los trabajadores que se muestran en la figura representan los papeles que juegan las personas durante el desarrollo. No obstante, en esta fase el equipo de trabajo es pequeño, por lo que sus miembros desempeñan varios papeles distintos.

13.4.1. Recopilación de requisitos

En la fase de inicio, el énfasis reside principalmente en el primer flujo de trabajo, el de requisitos. Este flujo de trabajo incluye identificar y detallar los casos de uso pertinentes en esta fase. Esto incluye los siguientes aspectos, descritos en el Capítulo 6:

1. Enumerar los requisitos candidatos a figurar en la lista de características del sistema.
2. Comprender el contexto del sistema.
3. Representar los requisitos funcionales pertinentes como casos de uso.
4. Recoger los requisitos no funcionales relacionados.

Enumerar los requisitos candidatos La lista de características surge a menudo de la experiencia que los clientes o usuarios tienen con sistemas predecesores o similares. En el caso de productos de gran distribución, las nuevas características surgen por demanda del mercado. Adicionalmente, muchas de las ideas de posibles características surgen de los trabajadores de la propia organización de desarrollo. Algunas características surgen de la interacción entre la organi-

zación software y los usuarios; otras se originan en el departamento de ventas. Las características provenientes de todas esas fuentes se convierten en requisitos candidatos para la lista de características, descrita en el Capítulo 6.

Comprender el contexto del sistema Si el cliente tiene un modelo de negocio (o un modelo del dominio), el equipo de personas de la fase de inicio pueden trabajar sobre él. Si el cliente no dispone en la actualidad de tal modelo, deberíamos alentarle a desarrollarlo, incluso aunque el hacerlo requiera por lo general un tiempo y recursos que exceden en mucho los que están a disposición de un único proyecto. Sea enérgico.

Recopilar los requisitos funcionales Los requisitos funcionales se representan como casos de uso, tal como se indica en el Capítulo 7 y en la Sección 13.4.1.1.

Recopilar los requisitos no funcionales Los requisitos funcionales específicos de un caso de uso, deben ser adjuntados al caso de uso al que se aplican (Secciones 6.3 y 7.2.3.2). Aquéllos que sean específicos de un objeto del modelo de negocio o del dominio deben reflejarse en el glosario que se adjunta al modelo de casos de uso (Sección 6.3); aquéllos que sean más generales (que serán mucho menores en número), se recogen como requisitos adicionales (Sección 6.7). Algunos de estos últimos son muy importantes a la hora de elegir la plataforma, así como la arquitectura de las capas del sistema e intermedia, y serán muy importantes en esa fase (*véase* la Sección 4.3, "Casos de uso y arquitectura").

13.4.1. Representar los requisitos como casos de uso

Volvamos ahora a la Figura 13.2 para discutir las actividades representadas en ella. En esta sección relatamos cómo representar los requisitos como casos de uso en términos de las actividades que muestra la Figura 13.2.

Encontrar actores y casos de uso (*véase* la Sección 7.4.1) Una relación "completa" de los requisitos, lo mismo que un modelo de casos de uso completo, está más allá del alcance de los recursos de la fase de inicio. Los participantes en esta fase deben, en primer lugar, clasificar el subconjunto de casos de uso necesarios para llevar a cabo el trabajo de esta fase, y en segundo lugar, detallarlos, como se establece en el Capítulo 7. Cómo encontrar estos casos de uso está esbozado en la Sección 12.6.

De nuevo, el problema de la fase de inicio está en restringir el volumen de trabajo, fundamentalmente para detallar aquellos casos de uso que afecten a los objetivos de la fase. Ignore las alternativas o caminos dentro de un caso de uso, e ignore los casos de uso que no sean importantes para el ámbito del sistema o la arquitectura candidata, los que no planteen riesgos críticos, o los que tengan poco efecto en el análisis inicial de negocio.

Para clasificar los casos de uso con los que trabajar en la fase de inicio, el jefe de proyecto debe colaborar con el analista de sistemas y el arquitecto. El analista de sistemas se ocupará de establecer el ámbito del sistema. El arquitecto deberá prestar atención a todos los casos de uso, con objeto de decidir cuáles no precisan ser tenidos en cuenta por el momento. El arquitecto debe ocuparse de los riesgos críticos y significativos, e identificar aquellos casos de uso necesarios para planificar el trabajo en la arquitectura, trabajo que es realizado fundamentalmente en la fase siguiente. El esfuerzo necesario podría ser muy pequeño en aquellas partes del sistema en las que el arquitecto ha trabajado en sistemas anteriores, en las que sabe que no hay nada importante para los objetivos de esta fase. El arquitecto debe atender a todos los casos de uso, al menos para decidir cuáles no son dignos de consideración en este momento.

Determinar la prioridad de los casos de uso (*véase* la Sección 7.4.2) A continuación, el arquitecto examina el modelo de casos de uso y proporciona información al jefe de proyecto para crear el plan de proyecto o plan de la iteración. Esta tarea va en paralelo con las actividades de los flujos de trabajo fundamentales en cada fase; es decir, es posible planificar futuras iteraciones al mismo tiempo que se detallan los casos de uso encontrados ya. El plan de la iteración es el resultado final que muestra las prioridades de los casos de uso asociados a los objetivos de esta fase.

Detallar un caso de uso (*véase* la Sección 7.4.3) Al mismo tiempo que aceptamos la necesidad de esta restricción de trabajo, reiteramos no obstante la importancia de completar todas las alternativas pertinentes en los casos de uso que tengan que ver con la fase de inicio, es decir, los casos de uso necesarios para determinar el ámbito del sistema, planificar cómo mitigar los riesgos críticos y planificar el trabajo de la línea base de la arquitectura En un gran número de casos, se cree que se han comprendido los requisitos necesarios, pero en realidad se han pasado por alto algunos que son clave. Existe una tendencia a creer que los beneficios de detallar los casos de uso necesarios no compensan el coste de hacerlo. Por el contrario, sí que merece la pena, siempre suponiendo que detallarlos sea realmente necesario.

El objetivo es saber hacia dónde nos dirigimos. Esto no significa, no obstante, que se deba trabajar sobre un gran número de casos de uso. Por lo general, ya que no es necesario construir un prototipo de la arquitectura en esta fase, no son necesarias labores de implementación y pruebas. Sin embargo, si se quiere mostrar las ideas fundamentales con un prototipo desechable, se tendrá que tener en cuenta un pequeño porcentaje del conjunto de casos de uso (Apéndice C), lo que se explicará a continuación, para este prototipo. Se puede tener que detallar algo así como el 10 por ciento del conjunto de casos de uso del modelo de casos de uso.

Lo que esta afirmación quiere expresar es que, aunque se toman en consideración muchos casos de uso, sólo se detalla una fracción de ellos. Se entra en detalles en aquéllos que son pertinentes. Por ejemplo, si seleccionamos como potencialmente pertinentes para esta fase el 50 por ciento de los casos de uso totales, y a continuación nos damos cuenta de que sólo es necesario detallar el 20 por ciento, en promedio, de los escenarios de cada uno de ellos, habremos trabajado en detalle sólo en el 10 por ciento del conjunto total de casos de uso. La cuestión está en que intentemos contener el coste y agenda de la fase de inicio. Cuáles son los porcentajes exactos para cada proyecto concreto depende de lo difícil o inusual que sea.

Todos los requisitos funcionales necesarios en esta fase se representan como casos de uso, como se discute en el Capítulo 7.

Construir un prototipo de la interfaz de usuario (*véase* la Sección 7.4.4) No es de interés en esta fase.

Estructurar el modelo de casos de uso (*véase* la Sección 7.4.5) No es de interés en esta fase.

13.4.2. Análisis

Los objetivos generales del flujo de trabajo de análisis son analizar los requisitos, refinarlos y estructurarlos en un modelo de objetos que sirva como primera impresión del modelo de diseño. En esta fase, el resultado es un modelo inicial de análisis. Utilizaremos este modelo de análisis para definir con precisión los casos de uso, y como ayuda para guiarnos en el establecimiento de

la arquitectura candidata —el desarrollo completo de la línea base de la arquitectura es asunto de la fase de elaboración. Por supuesto, esto significa que sólo una parte muy pequeña del modelo de análisis se completa en la fase de inicio (digamos un 5 por ciento). De hecho, podríamos considerar el modelo de análisis de la fase de inicio como una primera impresión; se trata sólo del primer paso para la vista de la arquitectura del modelo de análisis.

Análisis de la arquitectura (*véase* la Sección 8.6.1) La tarea a realizar en la fase de inicio consiste en clasificar los casos de uso o escenarios que necesitamos examinar cuidadosamente para el propósito de esta fase —principalmente, comprenderlos y refinarlos. Dado este conjunto inicial de casos de uso y escenarios, el arquitecto construye una primera versión del modelo de análisis para estas partes del sistema. No es necesario que sea exhaustivo, y no es necesario que sea perfecto. No se necesita para empezar a construir directamente el proyecto sobre él en la siguiente fase. Se puede, de hecho, descartarlo, excepto por su valor como guía.

Analizar un caso de uso (*véase* la Sección 8.6.2) En algunas situaciones, considerar los casos de uso uno a uno no es suficiente. El modelo de casos de uso muestra sólo un caso de uso de cada vez. En realidad, algunos casos de uso comparten recursos en el sistema, tales como bases de datos, recursos computacionales, etc. El modelo de análisis revela esos recursos compartidos. Por tanto, a menudo necesitamos realizar el análisis hasta el punto de resolver estos conflictos.

En el flujo de trabajo de esta iteración, se pueden analizar, y a continuación refinar, algunos de los casos de uso (el 10 por ciento del conjunto de casos de uso) que se detallaron en la Sección 13.4.1.1. Se puede detallar la mitad de ellos, es decir, alrededor del 5 por ciento del conjunto de casos de uso.

Analizar una clase y analizar un paquete Si se realiza en esta fase, hágalo mínimamente.

13.4.3. Diseño

En esta fase, el objetivo principal del flujo de trabajo de diseño es esbozar un modelo de diseño de la arquitectura candidata, con objeto de incluirlo en la descripción de la arquitectura preliminar.

Si necesitamos desarrollar un prototipo de demostración, lo haremos utilizando módulos prefabricados, lenguajes de cuarta generación (4GLs), o cualquier técnica de desarrollo rápido que simplemente muestre la idea. La demostración de interfaces de usuario y algoritmos no habituales hace creíble a todos los involucrados en el proyecto que merece la pena seguir adelante. Mostraremos este prototipo a usuarios representativos, para asegurarnos que satisface sus necesidades, y si hace falta, para hacer cambios para satisfacer mejor dichas necesidades.

Diseño de la arquitectura (*véase* la Sección 9.4.1) La intención, en el flujo de trabajo de diseño, es desarrollar un esbozo inicial del modelo de diseño, un primer paso para la vista de la arquitectura del modelo de diseño que realice los casos de uso (que fueron identificados anteriormente en el flujo de trabajo de requisitos) como colaboraciones entre subsistemas o clases. Debemos ser perspicaces al identificar interfaces (y como mucho definir algunas de sus operaciones) entre los subsistemas o clases, incluso si se trata simplemente de un esbozo del diseño. Estas interfaces son importantes porque forman el núcleo de la arquitectura. Más aún, necesitamos identificar los mecanismos genéricos de diseño que necesitaremos en las capas subyacentes de los subsistemas que van a llevar a cabo los casos de uso que hemos identificado. Elegiremos el software del sistema y los marcos de trabajo que se utilizarán en la capa intermedia (*véase* la Sección 9.5.1). Deberíamos hacer este modelo inicial de diseño incluso aunque

sea un esbozo, ya que en esta fase vamos a profundizar sólo hasta el punto de la descripción de una arquitectura candidata.

El modelo de diseño lleva a cabo no sólo los requisitos funcionales representados por los casos de uso designados, sino también los requisitos no funcionales, como el rendimiento, que puedan ser un riesgo.

Si el sistema propuesto va a ser distribuido sobre varios nodos, el arquitecto diseñará una versión a pequeña escala del modelo de despliegue, limitado, por ejemplo, a los nodos cuyo rendimiento es puesto en cuestión o a conexiones entre nodos. En este punto, el jefe de proyecto podría pedir a un ingeniero de casos de uso que modele las partes de dos nodos y de la interacción entre ellos en donde resida la amenaza.

Diseñar un caso de uso (*véase* la Sección 9.4.2) En la fase de inicio, el trabajo de diseñar casos de uso es mínimo.

Diseñar una clase y diseñar un subsistema De nuevo, si se realiza en esta fase, hágalo mínimamente.

13.4.4. Implementación

La actividad en este flujo de trabajo depende de decisiones que el jefe de proyecto haya tomado anteriormente. ¿Debe finalizar la fase de inicio con la descripción de una arquitectura candidata?

Por un lado, hay quien sostiene que no se puede estar seguro de que la arquitectura candidata sirva hasta que no se vea un prototipo funcionando. No se puede estar seguro de que se haya eliminado un riesgo hasta que la parte del prototipo referente a ese riesgo funcione realmente. Por otro lado, al mantener al mínimo la plantilla y el tiempo dedicado a la fase de inicio, se deben parar los flujos de trabajo tan pronto como se tenga la descripción de una arquitectura que parezca que pueda funcionar. Decidir el punto exacto hasta el que desarrollar la arquitectura candidata es responsabilidad del jefe de proyecto.

En situaciones normales, éste finalizará la fase con la descripción de la arquitectura candidata, en cuyo caso, seguir con el flujo de trabajo de implementación no es necesario.

No obstante, puede ser necesario un prototipo de demostración (un prototipo desechable). Si esto es así, el proyecto entrará en un flujo de trabajo de implementación, si bien éste puede ser pequeño.

13.4.5. Pruebas

Como muestra la Figura 13.2, en paralelo con las actividades de análisis, diseño e implementación, los ingenieros de pruebas se van poniendo al corriente de la naturaleza general del sistema propuesto, van considerando qué pruebas requerirá, y van desarrollando algunos planes provisionales de pruebas. Sin embargo, no se realiza un trabajo significativo de pruebas durante la fase de inicio, como indican las Figuras 11.2 y 13.1, ya que el prototipo exploratorio de demostración tiene, por lo general, carácter ilustrativo, más que operativo. En cualquier caso, el jefe de proyecto puede considerar útil el dedicar un pequeño esfuerzo a pruebas.

13.5. Realización del análisis inicial de negocio

Una vez que hemos establecido, al final de la fase de inicio, que el proyecto tiene una arquitectura candidata y que puede superar los riesgos críticos, es hora de traducir la idea del sistema a términos económicos, considerando las exigencias de recursos del proyecto, los costes de la inversión, las estimaciones de beneficios y la aceptación del mercado (o interna). Una parte del análisis de negocio es la apuesta económica; la otra cara de la moneda, las ganancias que se deriven del eventual uso del producto.

13.5.1. Esbozar la apuesta económica

Las fórmulas de estimación que subyacen a la apuesta económica, dependen normalmente del "tamaño" del producto final. No obstante, al acabar la fase de inicio, este tamaño puede diferir del tamaño final en un porcentaje sustancial, digamos un 50 por ciento (recuerde que sólo se ha detallado un pequeño porcentaje del conjunto de casos de uso). Según esto, las estimaciones en este punto pueden diferir del coste real al final del proyecto en el mismo 50 por ciento.

Por ejemplo, tomemos los casos de uso como medida del tamaño, con objeto de realizar las estimaciones. La cantidad de horas-persona necesarias para diseñar, implementar y probar un caso de uso varía desde un centenar hasta varios miles de horas. Que un caso de uso dado caiga en este rango depende de varios factores:

- *Estilo.* Si los encargados del desarrollo incluyen más características dentro de los límites de un único caso de uso, éste será más potente que un caso de uso promedio, pero puede costar más horas-persona.
- *Complejidad del sistema propuesto.* Cuanto más complejo sea el sistema, más costoso resultará (para un tamaño dado). Averigüe si puede simplificar el sistema reduciendo su funcionalidad.
- *Tamaño.* Por regla general, un caso de uso de un sistema pequeño es más fácil de implementar que un caso de uso de un sistema grande. El factor *tamaño del sistema* puede tener un impacto tan grande como 10 veces sobre las horas-persona por caso de uso.
- *Tipo de aplicación.* Algunos sistemas de tiempo real sobre entornos distribuidos, como por ejemplo los sistemas tolerantes a fallos o de alta disponibilidad, tienen un significativo impacto sobre el coste por caso de uso, con un factor de entre 3 y 5, comparados a otras aplicaciones.

Esta lista no es exhaustiva. Las organizaciones de desarrollo y los clientes del proyecto pueden ajustar las variables de estimación según su propia experiencia. En ausencia de experiencia, le sugerimos que haga las estimaciones tal y como haya venido haciéndolo tradicionalmente. A continuación, añada estimaciones del coste y tiempo necesarios para aprender un enfoque nuevo, usar herramientas nuevas, y adoptar el resto de nuevas características del Proceso Unificado. Después de una o dos experiencias con el Proceso Unificado, las organizaciones notan que sus costes han caído drásticamente, que los plazos de entrega han mejorado enormemente y que la calidad y fiabilidad de los sistemas ha aumentado.

Los equipos de proyecto no están normalmente en disposición de poder hacer el análisis de negocio definitivo al acabar la fase de inicio. Por este motivo, el Proceso Unificado no exige una "apuesta firme" hasta el final de la fase de elaboración. Es por esto que estamos llamando al aná-

lisis de negocio en la fase de inicio el análisis *inicial* de negocio. Este análisis inicial de negocio sólo necesita ser lo suficientemente preciso —en la práctica muy impreciso— como para justificar el entrar en la fase de elaboración. Por ejemplo, en la situación actual del mercado de computadores personales, no es necesario realizar una fase de inicio para averiguar que no hay oportunidad de negocio para lanzar al mercado otro procesador de textos típico. Sin embargo, en el caso de los proyectos que contemplan realmente las empresas, el análisis de negocio es siempre una cuestión que precisa de una gran cantidad de investigación.

Los propios requisitos de plantilla y agenda de las primeras iteraciones precisan de respaldo financiero. La necesidad que tienen los jefes de proyecto de las primeras iteraciones de datos sobre los cuales basar el esfuerzo y agenda de futuros proyectos es, con frecuencia, poco valorada. Por tanto, las organizaciones deben guardar registros de métricas adecuadas para la fase de inicio. Estas cifras les proporcionarían una base para estimar cuáles podrían ser las cifras para las iteraciones en la fase de inicio del próximo proyecto. Las cifras de proyectos anteriores pueden modificarse de acuerdo con la adecuada valoración de si el próximo proyecto va a ser más o menos complejo que los anteriores.

13.5.2. Estimar la recuperación de la inversión

Esta estimación de negocio proporciona una parte del análisis de negocio. Para la otra cara de la moneda, no existe una fórmula clara de calcular las ganancias que proporcionará el software. En el caso de un software que vaya a ser puesto en el mercado, el número de unidades vendidas, el precio al que se venderá el producto, y el periodo de tiempo en que se prolongarán las ventas, son aspectos que deben considerar los expertos en ventas y que deberán juzgar los ejecutivos. En el caso de software para uso interno, solicite a los departamentos afectados que estimen el ahorro que esperan conseguir. Normalmente, el margen de error es grande, pero al menos, la estimación proporciona una base para ponderar las ganancias frente a los costes.

Para un producto software comercial, el análisis de negocio debería incluir un conjunto de suposiciones sobre el proyecto y el orden de magnitud de la recuperación de la inversión si esas suposiciones son ciertas. Para asegurarse de que el análisis de negocio es lo razonablemente exacto como para merecer la pena, la dirección a menudo determina que ha de superarse un elevado margen de recuperación de la inversión.

Llegados a este punto, se puede realizar un análisis de negocio en general; es decir, si parece rentable continuar con el proyecto. Lo que necesitamos para concluir la fase de inicio no son cifras exactas, sino el conocimiento de que el sistema está —económicamente— al alcance de la organización de desarrollo y de sus clientes. En este punto, aún no se dispone de forma detallada de la información necesaria para completar el análisis de negocio en términos financieros. Para hacer esto, es necesario llegar a la situación de tener una apuesta económica y una agenda firmes y estables. Estas suposiciones se comprueban de nuevo al final de la fase de elaboración, una vez que se ha definido el proyecto de forma más precisa.

13.6. Evaluación de la iteración o iteraciones de la fase de inicio

Cercano al comienzo de la fase de inicio, tan pronto como la información adecuada estuvo disponible, el jefe de proyecto estableció los criterios con los cuales evaluar la conclusión de la pri-

mera iteración y de toda la fase, tal como se detalla en la Sección 13.2.4. Al llegar la fase de inicio a su fin, el jefe de proyecto designa un grupo de personas (que pueden ser tan sólo un par de ellas) para evaluarla. Normalmente, el grupo de evaluación incluye representantes del cliente o los usuarios. Para un proyecto de cierta envergadura, puede ser necesario incluir representantes de todos las personas involucradas. Algunos de los criterios pueden mostrarse como no alcanzables dentro del plan original. Ejemplos de estos criterios incluyen:

- Ampliación del modelo de casos de uso hasta el extremo necesario en esta fase.
- Desarrollo de un prototipo exploratorio hasta el punto de la demostración.
- Sospecha de que no se hayan encontrado todos los riesgos críticos.
- El hecho de que no todos los riesgos críticos considerados hayan sido mitigados o suficientemente cubiertos por un plan de emergencia.

El jefe de proyecto traslada los criterios aún no satisfechos a posteriores iteraciones, y modifica en consecuencia los planes y agendas de las iteraciones afectadas. Por ejemplo, el que sea necesario que trabajadores adicionales, con ciertas habilidades o antecedentes, se unan al equipo en la siguiente iteración.

Un resultado crucial de la evaluación de la fase de inicio es la importante decisión de si seguir adelante o abandonar el proyecto. Examine los objetivos de esta fase —ámbito, riesgos críticos, arquitectura candidata— y decida bien continuar o bien abandonar el proyecto. Puede ser necesario esperar a este hito principal antes de decidir seguir adelante. Se debería abandonar tan pronto como se conozcan los hechos que justifiquen hacerlo —no tiene sentido invertir esfuerzos adicionales. Sin embargo, esta decisión no es arbitraria. Seguir adelante o parar requiere la conformidad de las personas involucradas, en particular, de los inversores y de los representantes de los usuarios. Después de todo, en el caso de una posible cancelación, el cliente puede discurrir una forma de evitar el obstáculo.

13.7. Planificación de la fase de elaboración

Hacia el fin de la fase de inicio, a medida que vamos siendo conscientes de los costes y agenda de la fase de elaboración, comenzaremos a planificarla. En ella, intentaremos determinar alrededor del 80 por ciento de los requisitos; trataremos de no pasar por alto nada que sea importante para la arquitectura. Necesitamos hacer ambas cosas tanto para ser capaces de realizar una apuesta económica más exacta que la que permitieron los limitados datos disponibles en la fase de inicio, como para ser capaces de usarlos para elegir nuestra arquitectura. El 80 por ciento del conjunto de casos de uso es la proporción aproximada que se necesita habitualmente para fijar la apuesta económica. De este 80 por ciento, puede resultar necesario analizar el 50 por ciento para comprender correctamente los requisitos.

Para obtener la línea base de la arquitectura, podemos necesitar hasta un 80 por ciento para asegurarnos de que no hemos pasado por alto nada importante. De este 80 por ciento, seleccionaremos la parte significativa del conjunto total de casos de uso sobre la que basaremos el diseño de la línea base de la arquitectura. Estos casos de uso significativos serán aún un porcentaje más pequeño del conjunto total de casos de uso que ese 80 por ciento que hemos examinado, digamos, el 40 por ciento de los casos de uso y el 20 por ciento (en promedio) de cada uno de ellos. El producto de estos dos porcentajes es un conjunto de casos de uso de sólo el 8 por cien-

to. En otras palabras, por lo general, menos del 10 por ciento del conjunto de casos de uso alumbra todo lo que necesitamos saber en este momento sobre los casos de uso significativos. Usaremos esta fracción para dirigir el trabajo en la línea base de la arquitectura, lo que incluye una descripción de la arquitectura y versiones de todos los modelos.

De esta forma encaminaremos nuestros pasos a las iteraciones necesarias en la fase de elaboración —si se necesita más de una. Podemos asumir una iteración, pero puede haber más en casos complejos. Decidiremos lo que es conveniente hacer en cada iteración, qué requisitos implementar y probar y, de este modo, qué riesgos mitigar.

La experiencia indica que gran parte del diseño y la implementación desarrolladas en la fase de inicio, como los prototipos exploratorios y los usados para mostrar conceptos, no serán adecuados para utilizarlos en la fase siguiente.

De momento, puede ser que esté teniendo dificultades para seguir la pista de todos estos porcentajes. Por consiguiente, los hemos reunido en la Tabla 13.1, que sirve de referencia rápida.

Tabla 13.1 Trabajando con los caso de uso

	Modelo de negocio completado	**Casos de uso identificados**	**Conjunto de casos de uso descritos**	**Conjunto de casos de uso analizados**	**Conjunto de casos de uso diseñados, implementados y probados**
Fase de inicio	50-70%	50%	10%	5%	Un pequeño porcentaje para el prototipo
Fase de elaboración	Casi el 100%	80% o más	40%-80%	20%-40%	Menos del 10%
Fase de construcción	100%	100%	100%	100% si se mantienen	100%
Fase de transición					

Nota: Las cifras de la tabla son meramente indicativas. Distinguimos entre identificar y decir unas pocas palabras acerca de un caso de uso y describirlo más extensamente, lo que se hace en la actividad de detallar un caso de uso (*véase* la Sección 7.4.3) El análisis de los casos de uso se realiza en la actividad de analizar un caso de uso (Sección 8.4.2). La columna de la derecha indica qué parte del conjunto de casos de uso figura en la línea base al final de cada una de las fases.

13.8. Productos de la fase de inicio

La fase de inicio produce los siguientes productos:

- Una lista de características.
- Una primera versión del modelo de negocio (o del dominio) que describe el contexto del sistema.
- Un esbozo de los modelos que representan una primera versión del modelo de casos de uso, el modelo de análisis y el modelo de diseño. Respecto al modelo de implementación y el modelo de pruebas, puede existir algo rudimentario. También se genera una primera versión de los requisitos adicionales.

- Un primer esquema de la descripción de una arquitectura candidata, que perfila las vistas de los modelos de casos de uso, análisis, diseño e implementación.
- Posiblemente, un prototipo exploratorio que muestra el uso del nuevo sistema.
- Una lista inicial de riesgos y una clasificación de casos de uso.
- Los rudimentos de un plan para el proyecto en su totalidad, incluyendo el plan general de las fases.
- Un primer borrador del análisis de negocio, que incluye: contexto del negocio y criterios de éxito (estimaciones de beneficios, reconocimiento del mercado, estimaciones del proyecto).

Todas las personas involucradas en el proyecto deberían en este momento tener una comprensión bastante buena de la idea del proyecto y de que sea factible llevarlo a término. Se ha establecido un orden de prioridad entre los casos de uso. La información está ahora a mano, lo que permite al jefe de proyecto planificar detalladamente la siguiente fase. Los resultados alcanzados en esta fase se refinan en la fase de elaboración, hacia la que nos volvemos en el Capítulo 14.

Capítulo 14

La fase de elaboración construye la línea base de la arquitectura

Según entramos en la fase de elaboración, dejamos atrás un hito principal que señala tres logros:

- Hemos formulado la arquitectura inicial —la arquitectura candidata—, lo que significa que sabemos cómo construir una arquitectura que abarque las partes difíciles y novedosas del sistema propuesto.
- Hemos identificado los riesgos más serios —los riesgos críticos— y los hemos explorado hasta el punto de que confiamos en que sea factible construir el sistema.
- Hemos realizado el análisis inicial de negocio con el suficiente grado de detalle como para entrar en esta segunda fase, y hemos obtenido la aprobación de todos los implicados en el proyecto, en particular de aquéllos que financian la aventura.

4.1. La fase de elaboración en pocas palabras

Nuestros principales objetivos son:

- Recopilar la mayor parte de los requisitos que aún queden pendientes, formulando los requisitos funcionales como casos de uso.
- Establecer una base de la arquitectura sólida —la línea base de la arquitectura— para guiar el trabajo durante las fases de construcción y transición, así como en las posteriores generaciones del sistema.

- Continuar la observación y control de los riesgos críticos que aún queden, e identificar los riesgos significativos hasta el punto de que podamos estimar su impacto en el análisis de negocio, y en particular en la apuesta económica.
- Completar los detalles del plan del proyecto.

Para lograr estos objetivos, adoptaremos un punto de vista general del sistema, En algunos casos, en los que los riesgos técnicos predominen, o sean los más significativos, podemos necesitar profundizar para establecer una arquitectura sólida. En un proyecto de gran tamaño, podemos, por tanto, necesitar adoptar un punto de vista enfocado sobre los puntos clave del sistema. Los arquitectos del sistema deben identificar las partes más peliagudas del mismo tan pronto como sea posible, e iniciar experiencias piloto o prototípicas para identificar y gestionar el riesgo.

Tomaremos decisiones de la arquitectura basándonos en la comprensión del sistema *en su totalidad*: su ámbito y sus requisitos funcionales y no funcionales, como el rendimiento. Más aún, al tomar estas decisiones, tenemos que llegar a un equilibro entre los requisitos, representados en los casos de uso y el modelo de casos de uso, y la arquitectura. Ambos se desarrollan en conjunción y se influencian entre sí (*véase* la Sección 4.3).

El objetivo principal de la fase de elaboración es formular la línea base de la arquitectura. Esto implica desarrollar alrededor del 80 por ciento de los casos de uso y abordar los riesgos que interfieran en la consecución de este objetivo. En esta fase, acrecentaremos el entorno de desarrollo, no sólo para llevar a cabo las actividades de esta fase, sino para estar preparados para la fase de construcción. Hacia el final de esta fase, habremos acumulado la información necesaria para planificar la fase de construcción. También, en ese momento, tendremos información suficiente para realizar un análisis de negocio fiable, trabajo que comenzamos durante la fase de inicio.

14.2. Al comienzo de la fase de elaboración

Al comienzo de la fase de elaboración, recibimos de la fase de inicio un plan para la fase de elaboración, un modelo de casos de uso parcialmente completo y una descripción de la arquitectura candidata. Podemos tener también los rudimentos de un modelo de análisis y un modelo de diseño. Sin embargo, no podemos contar con reutilizar estos modelos, aunque puedan servirnos de guía. De hecho, una de las tareas de la fase de elaboración es completar estos modelos, tampoco esta vez en su totalidad, sino hasta el punto necesario para alcanzar la línea base de la arquitectura.

También podemos disponer de un prototipo que muestre el funcionamiento del sistema. Sin embargo, no podemos pretender construir sobre ese prototipo. Por lo general, ha sido preparado de la forma más rápida posible con objeto de determinar que el sistema es factible, no para servir de base sobre la que construir en la etapa siguiente.

14.2.1. Planificación de la fase de elaboración

La planificación de esta fase, realizada al final de la fase de inicio, puede no ser completa. A menudo, los recursos disponibles durante la fase de elaboración no se conocen con exactitud hasta que la fase ha comenzado. En algunos casos, hay un lapso de tiempo entre el fin de la fase de inicio y el comienzo de la fase de elaboración. Con un conocimiento más actualizado de los

recursos, la agenda y las personas disponibles, el jefe de proyecto modifica los planes de iteración y de fase previos.

14.2.2. Formación del equipo

No todo lo descubierto por el equipo de la fase de inicio queda registrado. Por consiguiente, el jefe de proyecto debe mantener en la fase de elaboración tantos miembros del equipo como sea necesario. Servirán, en parte, como "memoria del equipo". Además, durante la elaboración se pondrán de manifiesto necesidades adicionales. Por ejemplo, harán falta personas que tengan un conocimiento práctico de los bloques de construcción que podamos reutilizar en el proyecto. Según esto, el equipo de elaboración es, por lo general, algo mayor que el equipo de la fase de inicio. Subirán nuevas personas a bordo. El jefe de proyecto deberá seleccionar a algunas de esas personas para mantenerlas durante la fase de construcción, quizás para que lideren los equipos de diseño.

14.2.3. Modificación del entorno de desarrollo

A la luz de los resultados de la fase de inicio y de las necesidades de la fase de elaboración, el jefe de proyecto continúa haciendo cambios en el entorno de desarrollo, descrito por primera vez en la Sección 13.3.2.

14.2.4. Establecimiento de los criterios de evaluación

Los criterios específicos a alcanzar en una iteración, o en la fase de elaboración en su conjunto, son propios de cada proyecto, pero podemos considerar los resultados en función de los objetivos de esta fase.

14.2.4.1. Extender los requisitos

Los criterios de evaluación son:

- ¿Se han identificado los requisitos, actores y casos de uso necesarios para diseñar la línea base de la arquitectura? ¿Se han identificado los riesgos significativos?
- ¿Se han detallado lo suficiente como para lograr los objetivos de esta fase?

14.2.4.2. Definir la línea base de la arquitectura

Los criterios de evaluación son:

- ¿Satisface la línea base de la arquitectura no sólo los requisitos recopilados formalmente hasta ahora, sino también las necesidades de todos los usuarios?
- ¿Parece la línea base de la arquitectura lo suficientemente robusta como para resistir la fase de construcción y la adición de características que puedan ser necesarios en posteriores versiones del sistema?

14.2.4.3. Mitigar los riesgos significativos

Los criterios de evaluación son:

- ¿Se han mitigado de forma adecuada los riesgos críticos, ya sea eliminándolos o preparando un plan de emergencia?

- ¿Se han identificado todos los riesgos significativos? (*Véase* las Secciones 12.2.2 y 12.5)
- ¿Son los riesgos que aún permanecen en la lista de riesgos susceptibles de ser eliminados de forma rutinaria en la fase de construcción?

14.2.4.4. Juzgar la valía del análisis de negocio

Los criterios de evaluación son:

- ¿Está el plan del proyecto lo suficientemente definido como para apostar por un precio, agenda y calidad?
- ¿Parece verosímil que el análisis de negocio logre la recuperación de la inversión o alcance el margen de beneficios que el negocio impone?
- En resumen, ¿estamos preparados para redactar un contrato por un precio fijo (o el equivalente en el caso de un desarrollo interno)?

14.3. Flujo de trabajo arquetípico de una iteración en la fase de elaboración

La iteración arquetípica está formada por los cinco flujos de trabajo fundamentales, ilustrados en la Figura 14.1. Los flujos de trabajo están descritos en la Parte II, pero en este capítulo nos centramos sólo en el papel que desempeñan durante la elaboración. De nuevo, desarrollamos

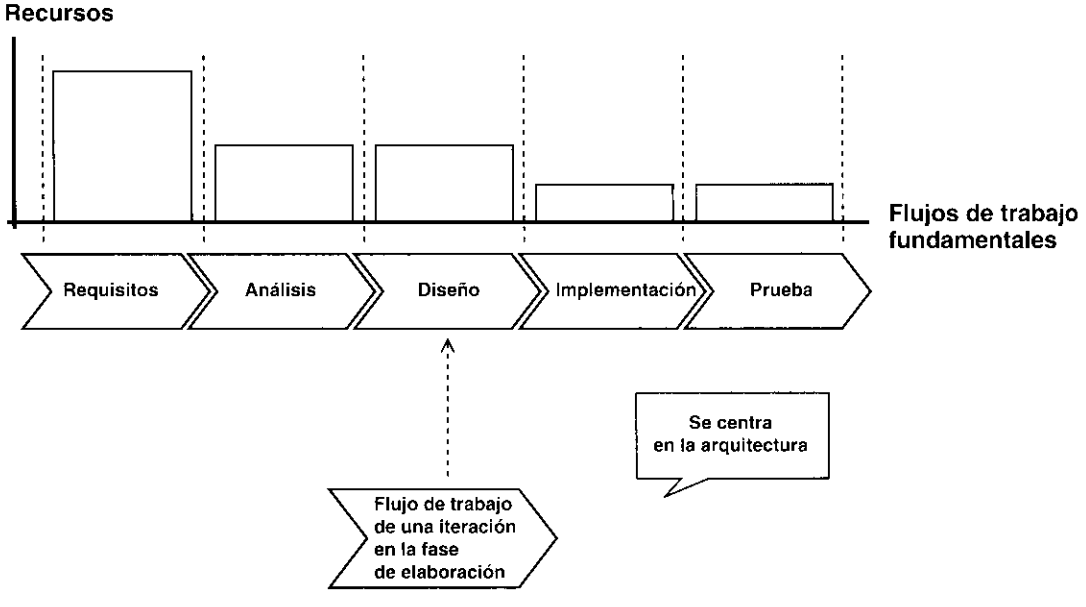


Figura 14.1. El trabajo de una iteración durante la fase de elaboración discurre a lo largo de los cinco flujos de trabajo fundamentales. La mayor parte del esfuerzo se realiza durante la recopilación de requisitos, análisis y diseño; es necesario comprender la mayoría de los requisitos y diseñar el sistema. En comparación, implementación y pruebas precisan menos recursos.

cuatro grupos de actividades en paralelo. Uno de ellos son los flujos de trabajo fundamentales; el segundo es planificar las iteraciones, como se describe en las Secciones 12.4 a 12.7 y en la Sección 14.2.1; el tercero es la evaluación descrita en las Secciones 12.8 y 14.6; y el cuarto es la preparación más detallada del entorno de desarrollo, descrita por primera vez en la Sección 13.5. En esta sección damos sólo una visión general de los flujos de trabajo fundamentales y su papel durante la fase de elaboración. En la Sección 14.4 entraremos en más detalles.

Durante el "diseño de la arquitectura", recopilaremos, analizaremos, diseñaremos, implementaremos y probaremos solamente los requisitos relevantes desde el punto de vista de la arquitectura. Apenas prestaremos atención a los detalles que no sean significativos, postergándolos hasta la fase de construcción. La línea base de la arquitectura resultante de estos esfuerzos será simplemente un esqueleto del sistema. No servirá de mucho por sí sola, exceptuando las partes que hemos tenido que implementar con detalle para comprobar que la arquitectura en su conjunto funciona. Desarrollaremos la arquitectura en una, dos, o en casos extremos varias iteraciones, dependiendo del ámbito del sistema, los riesgos, el grado de novedad, la complejidad de la solución técnica y la experiencia de los desarrolladores.

El objetivo de la investigación de riesgos en esta fase no es eliminar por completo los riesgos, sino reducirlos a un nivel aceptable para la fase de construcción. Otra forma de verlo, es decir que la fase de elaboración aborda los riesgos técnicos que tienen carácter de la arquitectura ¡mediante la implementación de la arquitectura! ¿Qué significa "nivel de riesgo aceptable para la fase de construcción"? Queremos decir que el riesgo ha sido explorado hasta el punto de que podemos prever la forma en que puede ser mitigado, y que podemos estimar el esfuerzo y el tiempo que llevará mitigarlo. En la práctica, el riesgo no será eliminado hasta que los casos de uso relacionados con él estén implementados, algunas veces en la fase de elaboración, normalmente en la fase de construcción, y en ocasiones, no antes de la fase de transición.

14.3.1. Recopilación y refinamiento de la mayor parte de los requisitos

¿Qué significa aquí recopilar "la mayor parte de los requisitos"? Esta cuestión empezó a abordarse en la Sección 13.7, al empezar a planificar la fase de elaboración. Allí dijimos que nuestra aspiración debería ser identificar alrededor del 80 por ciento de los casos de uso. Aquí podemos describir detalladamente entre el 40 y el 80 por ciento de todos los casos de uso. No es necesario identificar todos los casos de uso, y no es necesario describir en detalle todos los que identificamos, puesto que sabemos por experiencias anteriores que algunos sistemas pueden ser diseñados (arquitectónicamente) en seguida, no contener riesgos inesperados y pueden ser tasados de forma exacta. *Veáse* también los Capítulos 6 y 7.

Del conjunto de casos de uso que describiremos en detalle, seleccionaremos quizás la mitad para examinarlos muy cuidadosamente durante el análisis. De esta mitad, puede ser necesario considerar sólo una fracción de sus escenarios durante el diseño, implementación y pruebas, con objeto de obtener la arquitectura y mitigar los riesgos. *Véase* la Tabla 13.1. El objetivo es recopilar los requisitos hasta el punto de lograr los objetivos de esta fase.

14.3.2. Desarrollo de la línea base de la arquitectura

El arquitecto establece la prioridad de los casos de uso y realiza actividades de análisis, diseño e implementación —a nivel de la arquitectura, como se detalla en los Capítulos 8, 9 y 10. Otros

trabajadores llevan a cabo actividades de diseño, como se describe en los Capítulos 8 y 9; es decir, analizan las clases y paquetes (*véase* el Capítulo 8) y diseñan las clases y subsistemas (*véase* el Capítulo 9).

Los ingenieros de pruebas se centran en construir el entorno de pruebas y en probar los componentes y la línea base completa que implementa los casos de uso significativos desde un punto de vista de la arquitectura.

14.3.3. Iterando mientras el equipo es pequeño

Mientras el equipo es aún pequeño, como sucede durante la elaboración, tenemos la oportunidad de iterar y ensayar diferentes soluciones (tecnologías, marcos de trabajo, estructuras, etc.). Si el proyecto es todo un desafío, pueden necesitarse tres o cuatro iteraciones hasta conseguir una arquitectura estable. Más tarde, durante la fase de construcción, cuando pueda haber decenas de personas en el equipo y centenares de miles de líneas de código a las que seguir la pista, será necesario partir de una arquitectura estable y hacer crecer el sistema de forma incremental.

Una única iteración puede ser suficiente si el sistema es pequeño y sencillo, pero puede tratarse sólo del primer paso si el sistema es grande y complicado. La existencia de iteraciones adicionales dependen de aspectos como la complejidad del sistema y de la línea base de la arquitectura necesaria para guiar su desarrollo, o de la seriedad de los riesgos.

Las iteraciones continúan hasta que la arquitectura alcance un nivel estable, es decir, hasta que represente el sistema de forma afectable y haya llegado a un punto en el que sean previsibles pocos cambios.

14.4. Ejecución de los flujos de trabajo fundamentales, de requisitos a pruebas

En la fase de elaboración construiremos sobre el trabajo de la fase de inicio. Sin embargo, durante el inicio, sólo tuvimos que hacer creíble que podríamos, en una fase posterior, construir una arquitectura. Ahora, durante la elaboración, será cuando la hagamos real. Revisaremos lo hecho anteriormente, pero en la práctica, probablemente encontremos poco que podamos reutilizar. Ahora no estamos buscando simplemente casos de uso que representen riesgos críticos, sino los casos de uso que son significativos desde un punto de vista de la arquitectura. En segundo lugar, será necesaria una cobertura mucho mayor de los casos de uso con objeto de fijar una apuesta económica precisa. Más aún, nos enfrentamos a la tarea de concluir la fase con una línea base de la arquitectura ejecutable, una línea base estable sobre la que podamos añadir cosas en la fase de construcción. Por tanto, debemos construir prestando más atención a la calidad y la extensibilidad de lo que fue necesario en la fase de inicio.

Al comienzo de esta iteración, revisaremos los riesgos e identificaremos los casos de uso, como se describe en el Capítulo 12. Necesitamos cubrir alrededor del 80 por ciento de los requisitos para encontrar los que son significativos desde un punto de vista de la arquitectura, y también para recopilar suficiente información como para realizar una apuesta económica. En general, el encontrar el 10 por ciento que son relevantes para desarrollar la línea base de la arquitectura lleva aproximadamente esta proporción de los casos de uso.

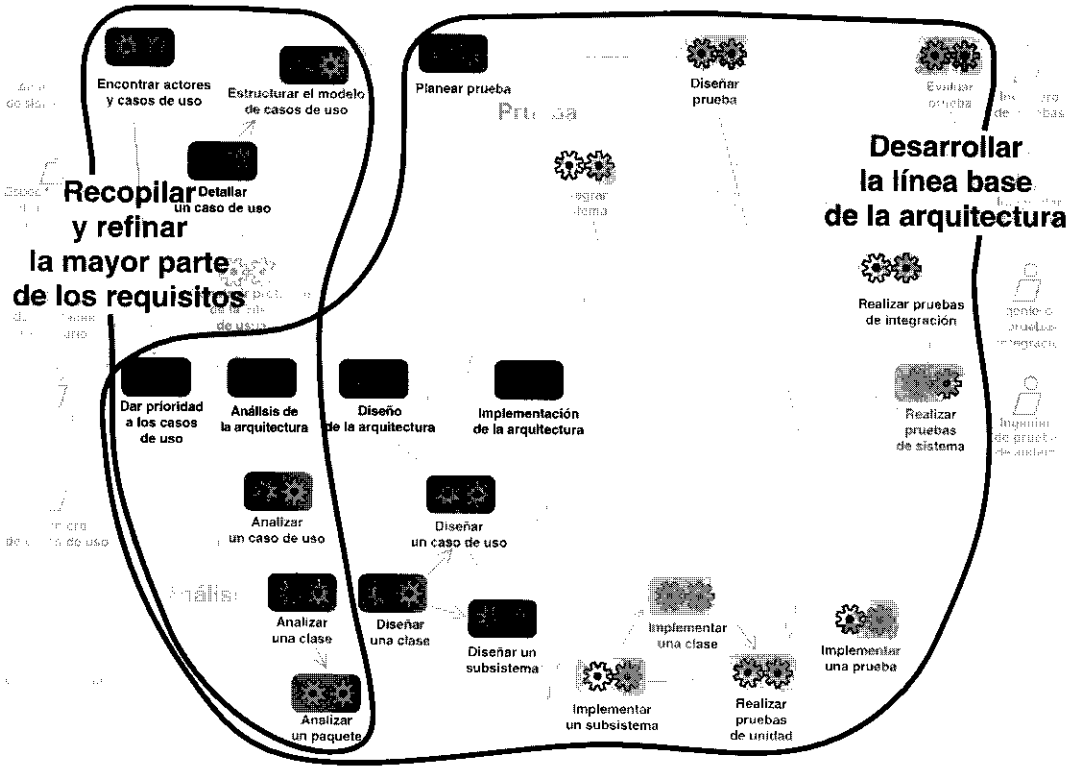


Figura 14.2. En esta figura se resaltan las actividades principales de la fase de elaboración.

En el proyecto hipotético que consideremos de aquí en adelante, supondremos un sistema moderadamente complejo para el que podamos obtener la línea base de la arquitectura en una iteración. Supondremos que el proyecto es novedoso. Como ya hemos indicado, el jefe de proyecto dispone de un plan del proyecto rudimentario y un plan de iteración algo más detallado para esta primera iteración, desarrollados en la fase de inicio. El primer paso es añadir detalles al plan de iteración en colaboración con el arquitecto y con los desarrolladores experimentados.

La relación siguiente está estructurada en términos de los cinco flujos de trabajo. Esta sucesión de flujos de trabajo podría inducir a pensar que se desarrollan secuencialmente, pero el trabajo puede realizarse de forma concurrente entre los distintos flujos, como muestra la Figura 14.2.

14.4.1. Recopilar los requisitos

En esta sección encontraremos, estableceremos la prioridad y estructuraremos los casos de uso (*véase* los Capítulos 6 y 7 para una discusión detallada sobre los requisitos).

14.4.1.1. Encontrar casos de uso y actores El analista de sistemas identifica casos de uso y actores adicionales a aquéllos identificados en la fase de inicio (*véase* la Sección 7.4.1). Aunque es necesario *comprender* alrededor del 80 por ciento de los casos de uso para alcanzar los objetivos de esta fase, no es necesario *detallar* toda esa cantidad. Podemos

identificar casi todos (el 80 por ciento), describir solamente una fracción de ellos, y analizar sólo partes de aquéllos que describimos. Por "comprender" queremos decir "aprehender lo que es significativo desde un punto de vista de la arquitectura", y estar seguros de que no hemos pasado por alto nada que pueda tener un impacto en la arquitectura o la apuesta económica.

Cuánto debemos recopilar también depende de lo exactos que necesitemos ser. Si vamos a establecer un precio fijo, por ejemplo, puede ser necesario detallar más casos de uso, es posible que hasta cerca del 80 por ciento de ellos. Para algunos sistemas complejos, puede ser necesario identificar casi todos los casos de uso y detallar hasta un 80 por ciento de ellos. Si somos nosotros mismos quienes financiamos el proyecto, podemos parar en un porcentaje menor. Por supuesto, parar ahí incrementaría el riesgo, por lo que sería un compromiso de la dirección el aceptar un riesgo mayor a cambio de menos tiempo y esfuerzo en la fase de elaboración. Sería absurdo afrontar, quizá sin saberlo, un riesgo posterior substancial para obtener una pequeña ganancia inmediata.

14.4.1.2. Desarrollar prototipos de las interfaces de usuario

Otra actividad que tiene lugar durante la recopilación de requisitos es la identificación de las interfaces de usuario (*véase* la Sección 7.4.4).

Durante la elaboración, sólo nos preocuparemos de las interfaces de usuario si son interesantes desde un punto de vista de la arquitectura. Sin embargo, éste es rara vez el caso —sólo en unos pocos casos son las interfaces de usuario únicas en algún sentido. Si lo son, podemos tener que crear nuestro propio marco de trabajo de interfaces de usuario. Un ejemplo es cuando el mismo sistema que estamos desarrollando es un marco de trabajo de interfaces de usuario. Otro ejemplo es un sistema con protocolos de comunicación exclusivos que son importantes para la arquitectura en términos de rendimiento o de tiempo de respuesta.

Hay una razón adicional para hacer una interfaz de usuario incluso si no es significativa desde un punto de vista de la arquitectura, y es para averiguar si funciona, utilizando para ello a los usuarios reales. Sin embargo, deberíamos llegar a este extremo sólo si no hemos sido capaces de demostrar la valía del sistema durante la fase de inicio. Por norma general, no es necesario desarrollar prototipos de las interfaces de usuario durante la elaboración.

14.4.1.3. Determinar la prioridad de los casos de uso

Al construir sobre el modelo parcial de casos de uso preparado en la fase de inicio, perseguimos dos objetivos: completar los casos de uso y trabajar en la línea base de la arquitectura (*véase* la Sección 7.4.2). Al principio, invertiremos tiempo en encontrar más casos de uso, para a continuación dirigir nuestra atención sobre la arquitectura. Sin embargo, debemos coordinar estos dos objetivos. Nuestras decisiones están influenciadas por las prioridades asociadas a los riesgos percibidos, y por el orden en que decidimos seguir el desarrollo. (*Véase* el Capítulo 7, Sección 7.4.2 y el Capítulo 12, Sección 12.6). A partir del modelo de casos de uso, el arquitecto genera una vista que se incluye en la descripción de la arquitectura.

14.4.1.4. Detallar un caso de uso

Los encargados de especificar los casos de uso completarán los detalles que sean necesarios para entender completamente los requisitos y para crear la línea base de la arquitectura (*véase* el Capítulo 7, Sección 7.4.3). En esta fase, limitaremos nuestros esfuerzos a realizar descripciones preliminares de casos de uso completos y arquitectónicamente significativos. Por lo general, no detallamos en su totalidad los casos de uso seleccionados, sino que limitamos el detalle a los escenarios que necesitamos para esta fase.

Evitaremos describir más de lo que sea necesario. Sin embargo, como dijimos antes, en algunos casos complejos puede ser necesario detallar casi todos los escenarios y casos de uso, es decir, cerca del 100 por cien del total de casos de uso.

14.4.1.5. Estructurar el modelo de casos de uso El analista de sistemas revisa lo que ha hecho y busca similitudes, simplificaciones y oportunidades para mejorar la estructura del modelo de casos de uso. El analista emplea mecanismos como la extensión o la generalización (*véase* la Sección 7.4.5) para lograr un modelo mejor estructurado y más fácil de entender. Podemos lograr que el modelo sea más fácil de modificar, ampliar y mantener, si, por ejemplo, reducimos la redundancia. No obstante, a veces el analista no logra descubrir en este momento la mejor estructura. Puede necesitar esperar hasta más adelante en la iteración, en ocasiones, no antes de que los casos de uso hayan sido sujeto de los flujos de trabajo de análisis y diseño.

Ejemplo **Estructuración del modelo de casos de uso**

A medida que los desarrolladores trabajen en el modelo de casos de uso, descubrirán que varios casos de uso tienen realizaciones similares. Por ejemplo, los casos de uso Pedir Bienes o Servicios, Confirmar Pedido, Enviar Factura al Comprador y Enviar Aviso, implican el envío de objetos de comercio entre Compradores y Vendedores. Un caso de uso reutilizable para este comportamiento común es Enviar Objetos de Comercio, que se añade al reestructurar el modelo de casos de uso. Cuando los desarrolladores procedan a realizar los casos de uso, reutilizarán la realización de Enviar Objetos de Comercio. Los objetos de comercio incluidos en este conjunto, tales como Factura, Pedido y Confirmación de Pedido, cambian su estado de forma similar, admiten operaciones similares, y son intercambiados entre Compradores y Vendedores. El hecho de que existan estas similitudes indica que todos ellos pueden ser construidos a partir de una clase abstracta: Objeto de Comercio.

14.4.2. Análisis

Durante la fase de inicio, comenzamos a hacer un borrador del modelo de análisis (*véase* el Capítulo 8 para una discusión detallada sobre análisis). Ahora, construiremos sobre este borrador, pero podemos descubrir que es necesario desechar partes substanciales de él. En la fase de elaboración, necesitamos trabajar con los casos de uso que son significativos desde un punto de vista de la arquitectura, y con aquellos casos de uso complejos que necesitemos refinar para comprender mejor los detalles de la apuesta económica.

En esta sección, abordamos las actividades de análisis de la arquitectura, analizar un caso de uso, analizar una clase y analizar un paquete. En el análisis, necesitamos ocuparnos de los casos de uso significativos desde un punto de vista de la arquitectura. Por lo general, esta proporción es menos del 10 por ciento del total. También analizaremos los casos de uso para entenderlos de forma más precisa y para discernir la interferencia de unos con otros. En total, podemos necesitar considerar el 50 por ciento de los casos de uso que hemos descrito en detalle.

Adaptar los requisitos a la arquitectura

Según vamos recolectando más requisitos —al ir completando casos de uso adicionales— utilizaremos nuestro progresivo conocimiento de la arquitectura que está siendo desarrollada para hacer esta tarea de forma hábil (*véase* la Sección 4.3). Al evaluar el valor y el coste de cada nuevo requisito o caso de uso, lo haremos a la luz de la línea base de la arquitectura que ya tenemos. La arquitectura nos dirá que algunos requisitos serán fáciles de implementar, mientras que otros serán difíciles.

Después de estudiar la situación que crearía un nuevo requisito, podríamos encontrar, por ejemplo, que un cambio en los requisitos —un cambio que tuviese poco o nulo impacto semántico— podría hacer la implementación más simple. Sería más simple debido a que el cambio en los requisitos llevase a un diseño más compatible con la arquitectura existente. Negociaremos este cambio en los requisitos con el cliente.

Según procedemos a analizar, diseñar, implementar y probar el sistema, necesitamos ajustar cualquier cambio en el diseño con la arquitectura ya existente en el modelo de diseño. El efectuar este ajuste significa que debemos tener en cuenta los subsistemas, componentes, interfaces, realizaciones de casos de uso, clases activas, etc. que ya existan. De esta forma, podemos crear un nuevo diseño de forma efectiva a partir del ya existente.

Ejemplo **Renegociando los requisitos**

Imaginemos que somos una empresa que vende un paquete software llamado PortfolioPlus, que analiza carteras de acciones, a clientes particulares. Hace tres años, los clientes estaban de acuerdo con introducir los cambios en los precios de las acciones de forma manual. Con la explosión de la World Wide Web, los clientes se han vuelto mucho más exigentes, al poder conseguir las cotizaciones sin esfuerzo, de manera gratuita y casi en tiempo real. Para seguir siendo competitivos, necesitamos hacer que PortfolioPlus pueda recibir las cotizaciones a este ritmo.

Debido a la arquitectura de PortfolioPlus, cambiarlo para que pueda "leer las tiras de cotizaciones" directamente llevaría un montón de trabajo. Una alternativa menos costosa podría usar la API que hemos desarrollado con anterioridad para recibir datos de una hoja de cálculo Excel. Una solución simple y efectiva consistiría en implementar una macro en PortfolioPlus que no sólo cargue una hoja de cálculo Excel, sino que solicite una nueva versión de dicha hoja a nuestro servidor de Web, con las cotizaciones que el usuario necesite. Nuestro trabajo se limita entonces a tres tareas:

- Generar la hoja de cálculo en nuestro servidor de Web cuando los clientes de PortfolioPlus la soliciten.
- Habilitar el servidor de Web para acomodar el volumen esperado de usuarios de PortfolioPlus.
- Escribir la macro que traiga la hoja de cálculo.

Ejemplo **El mundo real**

Este ejemplo demuestra cómo reutilizar en sentido real, y muestra el impacto que las negociaciones pueden tener en un proyecto.

(continúa)

Por medio de la negociación de los requisitos con el cliente a la luz de la arquitectura disponible, las empresas de software han sido capaces de construir sistemas con menor coste y mayor calidad. Un ejemplo típico relativo a una empresa de telecomunicaciones ilustra esta mejora.

El cliente había preparado una completa lista de requisitos en un formato similar a los casos de uso. Al estimar el coste de desarrollar este sistema de forma tradicional, se descubrió que llevaría alrededor de 25 personas-año. El proveedor de software mostró a la empresa de telecomunicaciones que, modificando los requisitos para ajustarlos a una arquitectura ya existente, se podría conseguir algo similar, aunque no exactamente lo mismo. Al realizar el diseño según esta arquitectura fue posible recortar los costes de desarrollo ¡en un 90 por ciento!

La empresa de telecomunicaciones decidió adoptar la arquitectura propuesta, y consiguió un sistema que era un producto estándar, ligeramente modificado para acomodar sus necesidades específicas. Así ahorró más de 20 personas-año de esfuerzo de desarrollo. Además de esto, no se enfrentó con los costes adicionales de mantener hardware y software desarrollados a medida, sino que bastó con el mantenimiento, mucho más barato, del producto estándar.

La diferencia entre lo que el cliente quería inicialmente y lo que acordó comprar al final fue el resultado de cómo el vendedor ajustó los casos de uso a la arquitectura. Con ligeras variaciones en las interfaces de usuario, en la forma de supervisar los procesos principales, en la de medir y presentar el flujo de tráfico, etc. se logró ese espectacular 90 por ciento de reducción de costes. Además, el cliente consiguió una funcionalidad mayor que la solicitada. Recibió a un precio más bajo, lo que clientes anteriores habían pagado mucho más caro. El vendedor de software pudo mantener bajo el precio de estas funciones adicionales, debido a que ya había implementado y probado el sistema.

14.4.2.1. Análisis de la arquitectura En la fase de inicio, desarrollamos el análisis de la arquitectura sólo hasta el extremo de determinar que había una arquitectura factible (*véase* la Sección 8.6.1). Normalmente, esto no es ir demasiado lejos. Ahora, en la fase de elaboración, tenemos que extender el análisis de la arquitectura hasta el extremo de que pueda servir de base a una línea base de la arquitectura ejecutable.

Con este propósito, el arquitecto realiza una partición inicial (de alto nivel) del sistema en paquetes de análisis, trabajando sobre la vista de la arquitectura del modelo de casos de uso, los requisitos relacionados con ellos, el glosario, y el conocimiento del dominio, disponible en forma de modelo de negocio (o de un modelo simplificado del dominio). Para ello, puede emplear una arquitectura en capas, identificando los paquetes específicos de la aplicación y los paquetes generales; éstos son los paquetes más importantes desde la perspectiva de la aplicación. Al observar los casos de uso "directores" de la vista de la arquitectura del modelo de casos de uso, el arquitecto puede identificar paquetes de servicio y clases de análisis que sean obvios y arquitectónicamente significativos.

Además, al trabajar con las necesidades colectivas de los casos de uso, el arquitecto buscará los mecanismos subyacentes necesarios para la implementación de los casos de uso, e identificará los mecanismos genéricos del análisis que se necesiten (*véase* la Sección 8.6.1.3). Estos mecanismos, incluyen tanto colaboraciones genéricas (*véase* el Capítulo 3) como paquetes genéricos. La colaboraciones genéricas incluyen características tales como recuperación de erro-

res o procesamiento de transacciones. Los paquetes genéricos se refieren a características tales como persistencia, interfaces de usuario gráficas o distribución de objetos.

El arquitecto estará ahora en disposición de mejorar la vista del modelo de análisis.

14.4.2.2. Analizar un caso de uso Muchos casos de uso no son claramente comprensibles tal y como están descritos en el modelo de casos de uso (*véase* la Sección 8.6.2) Los casos de uso deben ser refinados en función de las clases del análisis que existen en el ámbito de los requisitos pero que no se implementan necesariamente de forma directa. Esta necesidad de refinamiento es particularmente aguda para los casos de uso complejos, y para aquéllos que tienen impacto unos en otros, es decir, para los casos de uso que son dependientes unos de otros. Por ejemplo, para que un caso de uso sea capaz de acceder a determinada información, algunos otros casos de uso deben haber proporcionado dicha información.

Por tanto, los casos de uso interesantes desde un punto de vista de la arquitectura, junto con los casos de uso cuya comprensión es importante, deben ser refinados en función de estas clases del análisis. Los otros casos de uso, los que nos son interesantes desde la perspectiva de la arquitectura o de la comprensión de los requisitos, no se refinan ni analizan. Para estos casos de uso, los ingenieros de casos de uso sólo necesitan una comprensión de lo que son, y del hecho de que no tendrán impacto. Sabrán cómo tratar con ellos cuando sea la hora de implementarlos —durante la fase de construcción.

No es necesario describir con mucho detalle los casos de uso significativos o complejos, sólo hasta el punto de que los analistas comprendan la tarea que los casos de uso están perfilando, es decir, la línea base de la arquitectura y el análisis de negocio. Si considerásemos el 80 por ciento de los casos de uso con el objeto de comprender su papel en el sistema, y describiésemos menos del 40 por ciento del total de los casos de uso, entonces podríamos ocuparnos por lo general de algunos menos en el análisis, ya que algunos de estos casos de uso no tendrán ningún impacto en el análisis de negocio (*véase* la Tabla 13.1, referente a estos porcentajes).

A continuación, los ingenieros de casos de uso empezarán a buscar clases del análisis que realizan el caso de uso. Utilizarán como entrada las clases arquitectónicamente significativas que fueron identificadas por el arquitecto, y asignarán responsabilidades a dichas clases. Gran parte del trabajo al analizar los casos de uso consiste en trabajar sobre cada caso de uso del modelo de casos de uso y especificarlo de forma más detallada en función de las clases y sus responsabilidades. También mostrarán sus relaciones (entre clases) y sus atributos (de las clases).

Ejemplo **Responsabilidades de una clase**

Durante la primera iteración, mientras los desarrolladores trabajan sobre el caso de uso Pagar Factura, sugieren una clase para planificar y efectuar pagos —el Planificador de Pagos— con las siguientes responsabilidades:

- Crear una solicitud de pago.
- Iniciar la transferencia de dinero en la fecha indicada.

En una iteración posterior, los desarrolladores pueden encontrarse con que son necesarias más responsabilidades. Añadirlas no debería implicar reestructurar las clases. En un buen modelo de análisis, deberíamos ser capaces de añadir nuevas responsabilidades sin desechar lo que ya habíamos hecho o, lo que sería aún peor, sin tener que reestructurar las clases que ya habíamos encontrado.

Más adelante, cuando los desarrolladores amplíen el ámbito de su trabajo, digamos durante una segunda iteración en la fase de elaboración, puede descubrirse que la clase debe tener más responsabilidades, que deberían ser capaces de acomodar sin reestructurar. Estas responsabilidades podrían incluir:

- Seguir el rastro de los pagos que han sido planificados.
- Notificar a una Factura una vez que haya sido planificado su pago (Planificada) y cuando haya sido pagada (Cerrada).

Tomando como base esta labor de análisis sobre los casos de uso, el arquitecto selecciona las clases que son arquitectónicamente significativas. Estas clases se convierten en la base de la vista de la arquitectura del modelo de análisis.

14.4.2.3. Analizar una clase Los ingenieros de componentes deberán refinar las clases identificadas en los pasos anteriores, mezclando las responsabilidades que han sido asignadas a estas clases desde diferentes casos de uso. También identificarán los mecanismos de análisis disponibles y averiguarán cómo son utilizados por cada clase (*véase* la Sección 8.6.3).

14.4.2.4. Analizar un paquete Como hemos mencionado anteriormente en el análisis de la arquitectura, el arquitecto meditará sobre los servicios del sistema y sobre el agrupamiento de clases en paquetes de servicio. Esto se hará en la actividad de análisis de la arquitectura; dada esta agrupación en paquetes de servicio, los ingenieros de componentes asumirán la responsabilidad de los paquetes, su refinamiento y mantenimiento (*véase* la Sección 8.6.4).

14.4.3. Diseño

Por lo general, en esta fase diseñaremos e implementaremos menos del 10 por ciento de los casos de uso. Este pequeño porcentaje es sólo una fracción del total de casos de uso identificado durante esta fase. En la fase de elaboración diseñaremos desde un punto de vista de la arquitectura. Esto quiere decir que diseñaremos los casos de uso, clases y subsistemas que sean arquitectónicamente significativos. Los paquetes, durante el análisis, y los subsistemas, durante el diseño, son críticos para definir las vistas de la arquitectura. Mientras algunos clasificadores pueden ser significativos o no, desde un punto de vista de la arquitectura, por lo general, los paquetes y los subsistemas sí lo serán. (*Véase* el Capítulo 9.)

14.4.3.1. Diseño de la arquitectura El arquitecto es responsable del diseño de los aspectos arquitectónicamente significativos del sistema, tal como están descritos en la vista de la arquitectura del modelo de diseño (*véase* la Sección 9.3.6). La vista de la arquitectura del modelo de diseño incluye subsistemas, clases, interfaces y realizaciones de casos de uso arquitectónicamente significativos, incluidos éstos en la vista del modelo de casos de uso. Otros aspectos del diseño caen del lado del ingeniero de casos de uso y del ingeniero de componentes.

El arquitecto identifica la arquitectura en capas (incluyendo mecanismos de diseño genéricos), los subsistemas y sus interfaces, las clases de diseño arquitectónicamente significativas y las configuraciones de nodos de las que se trata a continuación:

Identificar la arquitectura en capas. El arquitecto continúa el trabajo comenzado en la fase de inicio y diseña la arquitectura en capas. Vuelve a considerar la capa de software del sistema y la capa intermedia tratadas en la Sección 9.5.1.2.2 y selecciona los productos que se van a utilizar finalmente. El arquitecto puede incorporar sistemas legados desarrollados por su propia organización, en cuyo caso debe identificar qué partes pueden ser reutilizadas y las interfaces de dichas partes. El arquitecto seleccionará los productos de las capas inferiores como implementaciones de los mecanismos de diseño que se corresponden con los mecanismos de análisis hallados en pasos anteriores (*véase* la Sección 14.5.2.1). Recuerde que por "mecanismos de diseño", nos referimos a mecanismos del sistema operativo sobre el que el sistema propuesto debe operar, lenguajes de programación, sistemas de base de datos, *object request brokers* (ORB), etc. El entorno de implementación limita los mecanismos de diseño que pueden ser utilizados por el producto, los cuales se obtienen ya sea desarrollándolos o comprando productos que los implementen, y serán a menudo los subsistemas de las capas intermedia y del software del sistema de una arquitectura en capas. Estos productos pueden ser construidos o desarrollados en paralelo con el flujo de trabajo de análisis. Los ingenieros de componentes realizarán sus diseños en función de ellos.

Ejemplo **Distribución de objetos mediante Java RMI**

Se usará Java RMI para la distribución de objetos, es decir, el paquete java.rmi será utilizado en la iteración de elaboración para implementar el caso de uso Enviar Objeto de Comercio.

Identificar los subsistemas y sus interfaces A continuación, el arquitecto trabajará en los niveles más altos de la arquitectura, cercanos a los niveles de aplicación. Por tanto, basándose en los paquetes del modelo de análisis, identificará los subsistemas correspondientes, que deben incluirse en el modelo de diseño. Normalmente, intentará hacer de cada paquete de servicio del modelo de análisis un subsistema de servicio en el diseño; los paquetes del análisis de más alto nivel se convertirán en subsistemas en el modelo de diseño.

Este enfoque resulta bien en algunos casos, pero en otros, entra en juego la impedancia entre análisis y diseño. En algunas situaciones, un paquete del análisis no se traduce en un sistema del diseño, sino en un sistema legado (o parte de él). Tampoco la equivalencia tiene por qué ser uno a uno. Más bien, el sistema legado puede realizar varios paquetes de análisis o partes de ellos, o un paquete de análisis puede traducirse en varios sistemas legados diferentes, es decir, se trata de una relación muchos a muchos.

En otras situaciones, el arquitecto puede elegir utilizar bloques de construcción reutilizables tales como marcos de trabajo, ya sean de desarrollo propio o fabricados por proveedores externos. Puede ser que estos bloques no se correspondan exactamente con la estructura del paquete que propone el modelo de análisis, así que el arquitecto puede tener que elegir una estructura de subsistema para el diseño de la arquitectura que sea de alguna manera diferente a la escogida durante al análisis de la arquitectura.

Identificar las clases de diseño significativas para la arquitectura El arquitecto "traducirá" las clases del análisis significativas para la arquitectura en clases de diseño. Según se van identificando nuevas clases del diseño, seleccionará aquéllas que sean interesantes desde un

punto de vista de la arquitectura, por ejemplo clases activas, y las describirá en la descripción de la arquitectura.

Si se trata de un sistema distribuido, identificar los nodos y las configuraciones de red El arquitecto reflexionará sobre la concurrencia y distribución requerida por el sistema, estudiando los hilos y procesos necesarios y la red física de procesadores y otros dispositivos. Los casos de uso ya diseñados, en particular tal y como se muestran en los diagramas de interacción, son de gran ayuda para esta tarea. El arquitecto alojará los objetos usados en los diagramas de interacción en clases activas, y éstas, a su vez, son asignadas a procesadores y otros dispositivos. Este paso distribuye la funcionalidad tanto en un sentido lógico como físico.

El arquitecto preparará una nueva versión de la vista de la arquitectura del modelo de diseño y una nueva versión de la vista del modelo de despliegue, incluyendo ambas en la descripción de la arquitectura.

14.4.3.2. Diseñar un caso de uso

Los casos de uso arquitectónicamente significativos son diseñados ahora en términos de subsistemas del diseño, subsistemas de servicio y clases del diseño (*véase* la Sección 9.5.2). (Los otros casos de uso que fueron identificados, detallados y analizados no han sido objeto de diseño durante esta fase.) Esta actividad es similar a la que hicimos en el análisis (analizar un caso de uso) con unas pocas diferencias importantes. Durante el análisis, queríamos analizar y refinar los casos de uso para conseguir una especificación que fuese robusta, fácil de adaptar a futuros cambios y reutilizable. Esta especificación funcionaría como un primer boceto del diseño. También tratamos de determinar las responsabilidades de las clases del análisis identificadas.

En el diseño, entraremos en muchos más detalles. Al moverse del análisis al diseño, los ingenieros de componentes tienen que adaptar el modelo de análisis para conseguir un modelo de diseño que pueda funcionar, puesto que éste último está limitado por los mecanismos de diseño. Sin embargo, los paquetes y clases del análisis nos proporcionan una forma directa de encontrar subsistemas y clases del diseño. Una vez encontrados, describiremos no sólo las responsabilidades que estos elementos del diseño deben asumir, sino también el detalle de las interacciones entre ellos.

En el análisis, describimos cómo el foco de atención se mueve de un elemento al siguiente al realizar un caso de uso, y utilizamos diferentes clases de diagramas de interacción para mostrar este movimiento. En el diseño, especificaremos además las operaciones usadas para la comunicación. En el diseño, necesitamos tener en cuenta qué subsistemas, marcos de trabajo o sistemas legados se van a reutilizar, y a continuación qué operaciones proporcionan. Si éstos elementos son difíciles de entender, el diseño será difícil de entender.

El resultado de esta actividad es un conjunto de realizaciones de caso de uso —diseño, uno de tales artefactos por cada caso de uso arquitectónicamente significativo.

14.4.3.3. Diseñar una clase

Diseñaremos las clases que participaron en las realizaciones de caso de uso del paso anterior. Obsérvese que, por lo general, las clases no están aún completas; participarán en más realizaciones de caso de uso que se desarrollarán en iteraciones posteriores. Los ingenieros de componentes integran los diferentes roles de cada clase en una clase consistente, tal como se describe en la Sección 10.3.

14.4.3.4. Diseñar un subsistema Los ingenieros de componentes diseñan los subsistemas resultantes del diseño de la arquitectura. Durante esta fase, el arquitecto actualizará, si es necesario, la vista de la arquitectura del modelo de diseño.

14.4.4. Implementación

Este flujo de trabajo implementa y prueba los componentes arquitectónicamente significativos a partir de los elementos de diseño arquitectónicamente significativos. El resultado es la línea base de la arquitectura, implementada normalmente a partir de menos del 10 por ciento de los casos de uso. En esta sección abordamos las actividades de implementación de la arquitectura, implementación de una clase e implementación de un subsistema, así como la integración del sistema. (*Véase* el Capítulo 10).

14.4.4.1. Implementación de la arquitectura Basándose en la vista de la arquitectura del modelo de diseño y la vista de la arquitectura del modelo de despliegue, se identifican los componentes necesarios para implementar los subsistemas de servicio. Los componentes ejecutables se asignan a los nodos de la red de computadores en la que van a ejecutarse. A continuación, el arquitecto ilustrará esto en la vista de la arquitectura del modelo de implementación (*véase* la Sección 10.5.1).

14.4.4.2. Implementación de una clase e implementación de un subsistema En el flujo de trabajo de diseño, los ingenieros de componentes diseñaron una serie de clases que eran relevantes para la creación de la línea base de la arquitectura. Esta línea base va a ser una versión ejecutable preliminar del sistema que vamos a construir. En este flujo de trabajo, los ingenieros de componentes implementarán estas clases en términos de componentes (por lo general, existen uno o más componentes que implementan un subsistema de servicio del modelo de diseño). La actividad de realizar pruebas de unidad asegura que cada componente funciona como una unidad. (*Véase* las Secciones 10.5.3. y 10.5.4.)

14.4.4.3. Integrar el sistema Sobre la base del pequeño porcentaje de casos de uso que se han de implementar en esta iteración, el responsable de integrar el sistema establece la secuencia de integración en un plan de integración y a continuación integra los subsistemas y los componentes correspondientes en una línea base de la arquitectura ejecutable. (*Véase* la Sección 10.5.2.)

Ejemplo **Tres construcciones**

El responsable de integrar el sistema sugiere tres construcciones iniciales. *Véase* la Figura 14.3.

El ejemplo que se muestra en la Figura 14.3 está compuesto de tres construcciones, cada una de las cuales es sujeto de la actividad de realizar pruebas de integración:

1. Las clases del subsistema de Gestión de Cuentas que engloba el sistema bancario legado.
2. Las clases del Paquete de Facturas de Comprador que están involucradas en el caso de uso Pagar Factura, junto con la primera construcción.

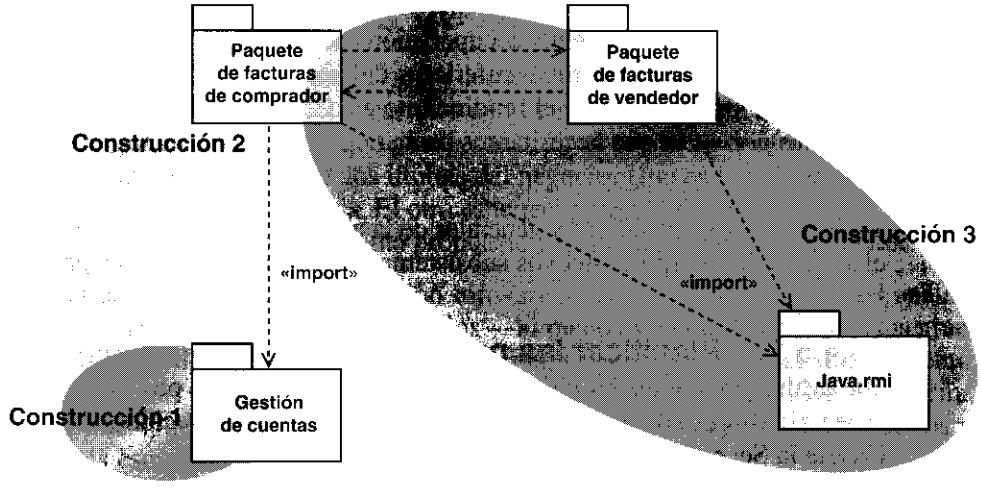


Figura 14.3. La primera iteración se compone de tres construcciones. Obsérvese que la Construcción 2 incluye también los resultados de la Construcción 1.

3. Las clases del Paquete de Facturas de Vendedor y del Paquete de Facturas de Comprador que están involucradas en el caso de uso Enviar Factura al Comprador. Estos subsistemas abrigan inicialmente las clases genéricas para el caso de uso abstracto Enviar Objeto de Comercio. Más adelante estas clases genéricas serán trasladadas a un sistema que podrá ser reutilizado en varios otros subsistemas. Esta construcción tiene que integrarse con el paquete Java RMI.

Aunque con algún esfuerzo, podríamos arreglarnos para trabajar sin herramientas durante la fase de inicio, pero claramente en la fase de elaboración no resulta práctico trabajar sin herramientas. Por ejemplo, es ahora cuando empezamos a manejar versiones, así que se necesitarán herramientas de control de la configuración. Es posible trabajar con lápiz y papel hasta llegar a la elaboración, pero no deberíamos acabar esta fase sin tener la línea base de la arquitectura bajo el control de alguna herramienta.

De esta forma, los trabajadores habrán sido capaces de utilizar el lenguaje de "producción", digamos Java, y las herramientas de desarrollo correspondientes, y este uso proporcionará la ventaja de probar los diferentes entornos de desarrollo y familiarizarse con herramientas y métodos nuevos. En este caso, es razonable que el primer incremento use la infraestructura del sistema final, es decir, el software del sistema y de la capa intermedia, y será más verosímil que el prototipo evolucione hacia el sistema final.

14.4.5. Pruebas

Aquí el objetivo es asegurarse de que los subsistemas de todos los niveles (subsistemas de servicio y subsistemas del diseño) y de todas las capas (desde la capa del sistema hasta las capas específicas de la aplicación) funcionen; *véase* el Capítulo 11. Por supuesto, sólo podemos probar los componentes ejecutables. Si funcionan, tendremos cierta seguridad de que otras cosas (en otros modelos) también funcionarán.

Empezar por las capas más bajas de la arquitectura significa probar los mecanismos de distribución, almacenamiento, recuperación (persistencia) y concurrencia de objetos, así como otros mecanismos de las capas inferiores del sistema. Esto implica no sólo comprobar la funcionalidad, sino también si el rendimiento es aceptable. Muchas de las capas no necesitan ser probadas *per se*; lo que necesitamos probar es cómo las capas superiores hacen uso de las inferiores.

En las capas específicas y generales de la aplicación, las pruebas miden la facilidad de escalar el sistema al incorporar nuevos subsistemas que usan interfaces ya previstas.

14.4.5.1. Planificar las pruebas El ingeniero de pruebas seleccionará los objetivos que evaluarán la línea base de la arquitectura. Un objetivo, por ejemplo, podría ser ejecutar un escenario de caso de uso con tiempo de respuesta preestablecido dado un cierto nivel de carga. (*Véase* la Sección 11.5.1).

14.4.5.2. Diseñar las pruebas Tomando como base estos objetivos, el ingeniero de pruebas identificará los casos de pruebas necesarios, y preparará procedimientos de pruebas para comprobar la sucesiva integración de subsistemas hasta acabar con la línea base completa. (*Véase* la Sección 11.5.2).

14.4.5.3. Realizar pruebas de integración Según se van comprobando los componentes, quedarán listos para las pruebas de integración. Los encargados de realizar estas pruebas comprobarán cada construcción. (*Véase* la Sección 11.5.4).

14.4.5.4. Realizar pruebas del sistema Una vez que el sistema, tal y como queda definido por los casos de uso arquitectónicamente significativos, ha sido integrado, el ingeniero de pruebas del sistema lo prueba. Este sistema (que es una versión del sistema final) es la línea base de la arquitectura. El ingeniero de pruebas notificará los defectos a los ingenieros de componentes o al arquitecto para su corrección. (*Véase* la Sección 11.5.5).

Los ingenieros de pruebas revisarán los resultados de las pruebas del sistema para verificar que se cumplen los objetivos originales o para decidir cómo se deben modificar los casos de prueba con objeto de alcanzar estos objetivos.

Ejemplo **Las transferencias inconsistentes suponen un riesgo grave**

Las pruebas del sistema determinan que la mayor parte de la funcionalidad parece satisfacer los objetivos de calidad esperados, con una excepción: cuando los encargados de probar el sistema ejecutan el caso de uso Pagar Factura, algunos de los resultados son incorrectos. La envolvente (*wrapper*) del subsistema legado de Gestión de Cuentas no produce el resultado previsto para algunas transferencias de cantidades con decimales, como por ejemplo 134.65 euros. Otras transferencias, como 124.54 euros, funcionan correctamente. Los encargados de las pruebas llaman la atención sobre este problema y el arquitecto lo califica de riesgo grave. El jefe de proyecto designa un grupo de trabajo para abordarlo inmediatamente.

4.5. Desarrollo del análisis de negocio

La razón subyacente para mitigar los riesgos y desarrollar la línea base de la arquitectura es el llevar el proyecto hasta un punto del desarrollo en el cual el equipo pueda empezar la fase de construcción con plena confianza de que puede construir el producto dentro de los límites del negocio. Fundamentalmente, existen dos límites del negocio. Uno es la planificación, esfuerzo y coste estimados para una calidad dada. El otro es la recuperación de la inversión (o alguna métrica comparable) indicando que el sistema propuesto será económicamente un éxito.

Al final de la fase de inicio, la organización software podía juzgar las virtudes del análisis de negocio con unos márgenes de error muy amplios —al menos, en el caso de proyectos grandes, difíciles o novedosos. Al final de la fase de elaboración, se ha avanzado en el conocimiento del proyecto hasta el punto de haber estrechado notablemente estos márgenes. De esta forma, se puede preparar una apuesta económica y desarrollar el análisis de negocio dentro de los márgenes mucho más estrechos de la práctica empresarial.

14.5.1. Preparar la apuesta económica

Se supone que el equipo de desarrollo desarrollará la fase de construcción desde una perspectiva de negocio, independientemente de que el acuerdo económico sea un contrato con un cliente externo, un acuerdo con otro departamento de la misma empresa o el desarrollo de un producto para vender a muchos clientes.

Hemos observado que las estimaciones del software se basan a menudo en el tamaño del proyecto y el productividad de la organización de desarrollo. La productividad de la organización se deriva de la experiencia en proyectos anteriores, pero el tamaño estimado se basa en lo que se ha aprendido en la fase de elaboración. Para hacer una estimación realista, habrá que perseverar en la fase de elaboración hasta el punto en que tengamos una buena apreciación del trabajo a realizar en la fase de construcción. Eso lo proporciona la línea base de la arquitectura. En segundo lugar, si el proyecto presenta riesgos de cierta magnitud, habrá que investigarlos hasta el extremo de estimar cuánto tiempo y esfuerzo llevará superarlos.

14.5.2. Actualizar la recuperación de la inversión

El análisis de negocio, reducido a lo esencial es éste: ¿Superarán las ganancias resultantes del uso o la venta del producto software el coste de su desarrollo? ¿Llegará al mercado (o a su uso interno) a tiempo de obtener esas ganancias?

La organización software dispone ahora de una estimación del coste de las fases de construcción y transición en forma de apuesta económica. Esta estimación constituye una parte del análisis de negocio. Para la otra cara de la moneda, no existe una fórmula clara para calcular las ganancias que proporcionará el software.

En el caso de un software que vaya a ser puesto en el mercado, el número de unidades vendidas, el precio al que se venderá el producto, y el periodo de tiempo en que se prolongarán las ventas son aspectos que deben considerar los expertos en ventas y que deberán juzgar los ejecutivos. En el caso de software para uso interno, solicite a los departamentos afectados que estimen el ahorro que esperan conseguir. Normalmente, el margen de error es grande, pero al menos, la estimación proporciona una base para ponderar las ganancias frente a los costes.

14.6. Evaluación de las iteraciones de la fase de elaboración

Al concluir una iteración, debe ser evaluada según a los criterios establecidos en el plan de iteración preparado antes de su comienzo, como se esbozó en la Sección 14.2.5. El equipo de evaluación revisa los resultados de cada iteración para comprobar que, efectivamente, la línea base representa una arquitectura capaz de llevar a cabo los objetivos iniciales y mitigar los riesgos.

Si va a haber varias iteraciones, el resultado de la primera de ellas puede ser sólo una primera aproximación a la arquitectura. Esta aproximación puede consistir perfectamente en una buena descripción de las vistas de la arquitectura de los diferentes modelos: casos de uso, análisis, diseño, despliegue e implementación. El resultado de la segunda iteración es la segunda aproximación a la arquitectura. La iteración final proporciona la línea base de la arquitectura. Al concluir cada iteración, el jefe de proyecto, normalmente de forma conjunta con un grupo de evaluación, evalúa lo que se ha logrado frente a los criterios establecidos de antemano. Si el proyecto no ha alcanzado estos criterios, el jefe de proyecto lo reorienta, como se esboza en el Capítulo 12. Durante la fase de elaboración, esto puede significar traspasar actividades no acabadas a la iteración siguiente.

Dado que el jefe de proyecto ha involucrado al cliente y otras personas implicadas en el proyecto en el logro de cada hito secundario, por lo general éstos encontrarán el hito principal (fin de la fase) menos traumático. El equipo de proyecto habrá desarrollado una relación más estrecha con el cliente de la que a menudo se obtiene al seguir el modelo de ciclo de vida en cascada. El cliente habrá dispuesto de oportunidades para sugerir mejoras a los modelos de desarrollo a lo largo de este camino.

En este momento, al final de la fase de elaboración, la evaluación debe convencer a los implicados en el proyecto que la fase de elaboración ha mitigado los riesgos graves y ha construido una línea base de la arquitectura estable. Les convence de que el sistema puede ser construido de acuerdo al plan del proyecto y a la apuesta económica para la fase de construcción.

14.7. Planificación de la fase de construcción

Hacia el final de la fase de elaboración, el jefe de proyecto comienza a planificar de forma detallada la primera iteración de la fase de construcción, y a esbozar en términos más generales las iteraciones restantes. El número de iteraciones necesarias depende del tamaño y complejidad del proyecto. Normalmente, los jefes de proyecto planifican dos o tres, algunas veces cuatro o más, en el caso de empresas grandes y complejas. En cada iteración, esbozan una serie de construcciones, cada una de ellas añadirá una pieza relativamente pequeña a lo que haya sido hecho hasta entonces.

La lista de riesgos contendrá aún muchos riesgos. El jefe de proyecto planifica el orden en que se investigarán los riesgos remanentes con objeto de mitigar cada uno de ellos antes de que aparezca en la secuencia de construcciones o iteraciones. Los principios operativos siguen siendo los mismos: reducir los riesgos antes de que interrumpan la secuencia de construcción.

La fase de elaboración habrá completado sólo porcentajes de cada uno de los modelos. El jefe de proyecto reflexionará sobre el orden en que trabajar en los casos de uso y escenarios restantes, y en que completar los modelos. Estas consideraciones llevan al secuenciamiento de las construcciones e iteraciones. En proyectos grandes, para reducir la planificación temporal total utilizando más personas, el jefe de proyecto organizará el trabajo que los trabajadores pueden

llevar a cabo en paralelo. El desarrollo de sistemas grandes, de uso industrial, implica que el equipo de proyecto debe encontrar rutas paralelas a través del proyecto, ya que estos proyectos se enfrentan con restricciones temporales. El equipo busca la forma de desplegar un gran número —a menudo decenas— de personas.

Este despliegue se basa en los subsistemas establecidos en la línea base de la arquitectura. En el flujo de trabajo de diseño (inspirado por los paquetes de análisis), nos encontraremos con subsistemas de diferentes niveles. Estos subsistemas tendrán interfaces, y un objetivo principal era la identificación y definición de estas interfaces. Las interfaces son el núcleo de la arquitectura. Una vez identificados y especificados los subsistemas y sus interfaces, estamos preparados para trabajar en paralelo.

El responsable de un subsistema de servicio de un subsistema del diseño será un individuo. Un grupo será responsable de un subsistema del diseño.

Si los individuos o pequeños grupos van a trabajar con un grado de independencia razonable, las interfaces que delimitan sus áreas de actividad deben ser sólidas. Para enfatizar la importancia de estas especificaciones de interfaz, se las llama a veces *contratos*. Un contrato compromete a los desarrolladores actuales, y a aquéllos de los ciclos subsiguientes, con una interfaz. El hecho de que una interfaz esté firmemente definida es lo que hace posible una arquitectura conectable. Más adelante, los desarrolladores pueden sustituir un subsistema por otro, siempre que no rompan el contrato de la interfaz. La construcción de subsistemas que se interconectan a través de contratos de interfaz (o algo equivalente) es en principio lo mismo que la construcción de sistemas de sistemas, de lo que trataba el recuadro de la Sección 7.2.3, "Modelado de grandes sistemas".

14.8. Productos clave

Los productos son:

- Preferiblemente un modelo completo de negocio (o del dominio) que describe el contexto del sistema.
- Una nueva versión de todos los modelos: casos de uso, análisis, diseño, despliegue e implementación. (Al final de la fase de elaboración estos modelos estarán completos en menos de un 10 por ciento, exceptuando los modelos de casos de uso y de análisis que pueden incluir más (en algunos casos hasta el 80 por ciento) casos de uso para tener la certeza de que se han comprendido los requisitos. La mayoría de los casos de uso han sido comprendidos para asegurarnos de que no han dejado de lado ningún caso de uso arquitectónicamente importante, y que podemos estimar los costes de introducirlos.)
- Una línea base de la arquitectura.
- Una descripción de la arquitectura, incluyendo vistas de los modelos de casos de uso, análisis, diseño, despliegue e implementación.
- Una lista de riesgos actualizada.
- El plan del proyecto para las fases de construcción y transición.
- Un manual de usuario preliminar (opcional).
- El análisis de negocio completo, incluida la apuesta económica.

Capítulo 15

La construcción lleva a la capacidad operativa inicial

El propósito primordial de esta fase es dejar listo un producto software en su versión operativa inicial, a veces llamada "versión beta". El producto debería tener la calidad adecuada para su aplicación y asegurarse de cumplir los requisitos. La construcción debería tener lugar dentro de los límites del plan de negocio.

Al final de la fase de elaboración, hemos llevado el producto software al estado de una línea base de la arquitectura ejecutable. Las fases previas han reducido los riesgos críticos y significativos a niveles rutinarios que pueden ser gestionados durante el plan de construcción. Durante la fase de elaboración, el equipo del proyecto estableció los fundamentos de los elementos arquitectónicamente significativos de los modelos de diseño y despliegue. Estos fundamentos incluyen los subsistemas, las clases activas, y los componentes y sus interfaces, así como las realizaciones de los casos de uso significativos. Esto se llevó a cabo detallando tan sólo un 10 por ciento de los casos de uso. Recuerde que para cumplir los objetivos de la fase de elaboración, se han recopilado casi todos los requisitos (normalmente alrededor del 80 por ciento), pero aún no han sido detallados completamente. Esto es lo que vamos a hacer en la fase de construcción.

15.1. La fase de construcción en pocas palabras

El equipo que trabaja en la fase de construcción, a partir de una línea base de la arquitectura ejecutable, y trabajando a través de una serie de iteraciones e incrementos, desarrolla un producto software listo para su operación inicial en el entorno del usuario, a menudo llamado versión beta. Para ello, detalla los casos de uso y escenarios restantes, modifica si es necesario la descripción de la arquitectura, y continúa los flujos de trabajo a través de iteraciones adicionales, dejando ce-

rrados los modelos de análisis, diseño e implementación. Además, integra los subsistemas y los prueba, e integra todo el sistema y lo prueba.

A medida que el proyecto pasa de la fase de elaboración a la de construcción, se produce un cambio de enfoque. Mientras que las fases de inicio y elaboración podrían ser consideradas como investigación, la fase de construcción es análoga al desarrollo. El énfasis se traslada de la acumulación del conocimiento básico necesario para construir el proyecto a la construcción propiamente dicha de un sistema o producto dentro de unos parámetros de coste, esfuerzo y agenda.

Durante la fase de construcción, el jefe de proyecto, el arquitecto y los desarrolladores se aseguran de establecer la prioridad de los casos de uso, que son agrupados en construcciones e iteraciones y desarrollados según un orden que evita las vueltas atrás.

Además, mantendrán actualizada la lista de riesgos, mediante su continuo refinamiento, de forma que refleje los riesgos reales del proyecto. Su objetivo es acabar esta fase con todos los riesgos mitigados (exceptuando aquéllos que se derivan de la operación, que son tratados en la fase de transición). El arquitecto vigila que la construcción se ajuste a la arquitectura, y si es necesario, modifica dicha arquitectura para incorporar los cambios que surgen durante el proceso de construcción.

15.2. Al comienzo de la fase de construcción

Al final de la fase de elaboración, el jefe de proyecto planificó la fase de construcción. Una vez que recibe autorización de los responsables financieros para seguir adelante, puede tener que modificar el plan de la fase de construcción de acuerdo con las circunstancias. En particular, podemos citar dos posibilidades que se dan a menudo.

Una es el posible lapso de tiempo entre las fases de elaboración y construcción. Los responsables financieros pueden autorizar inmediatamente la fase de construcción, habilitando al jefe de proyecto y su equipo a continuar sin interrupción, aprovechando su detallado conocimiento del proyecto. Desgraciadamente, puede producirse un lapso de meses entre la presentación de la propuesta y la apuesta económica y la firma real del contrato o la autorización para continuar. Mientras tanto, el jefe de proyecto y el equipo se dedican a otras tareas y pueden no estar reunidos de nuevo cuando se recibe la autorización de continuar. En el peor de los casos, la responsabilidad de la fase de construcción recae sobre un nuevo jefe de proyecto y un equipo en su mayoría nuevo.

La segunda posibilidad es que los fondos y la agenda sean menores de lo planificado para el proyecto durante la fase de elaboración. El ámbito del sistema puede haberse reducido para ajustarse a los fondos y agenda, o puede que no. La situación a la que estamos llegando es que, al principio de la fase de construcción, las circunstancias pueden diferir —poco o mucho— de aquéllas sobre las que se basaba la planificación al final de la fase de elaboración. Nos guste o no —y por lo general no nos gustará—, el jefe de proyecto tendrá que rehacer en cierta medida la planificación. En casi todos los casos, tendrá que adaptar el plan de proyecto de la fase de elaboración para ajustarse a los recursos de personal disponibles y a la agenda que han fijado los inversores.

15.2.1. Asignación de personal para la fase

En la fase de construcción, el trabajo se extiende más allá de la arquitectura (es decir, de los elementos del modelo arquitectónicamente significativos). Los subsistemas de servicio identifica-

dos por el arquitecto en la fase de elaboración no estaban completos —solamente se implementaron los casos de usos fundamentales y sus principales escenarios. El jefe de proyecto debe asignar personal adicional para esta tarea.

La línea base de la arquitectura con su representación de subsistemas e interfaces es la base de la que parte el jefe de proyecto para dividir el trabajo. Cada subsistema se convierte en responsabilidad de un desarrollador, que actuará como ingeniero de componentes. Normalmente, como dijimos en el Capítulo 9, el desarrollador responsable de un subsistema es también responsable de las clases de dicho subsistema. No es habitual que un desarrollador sea responsable de una sola clase; ésa sería una división demasiado detallada del trabajo.

La fase de elaboración da empleo a los siguientes puestos de trabajo para las tareas principales de la fase: ingeniero de casos de uso, ingeniero de componentes, ingeniero de pruebas, responsable de la integración del sistema, encargado de las pruebas de integración y encargado de las pruebas del sistema. Comparado con la fase elaboración, el número de trabajadores aumenta notablemente, aproximadamente al doble, como se muestra en la Sección 12.7. La Tabla 13.1 sugiere el volumen aproximado de trabajo que se deja para la fase de construcción por cada flujo de trabajo —un 60 por ciento en análisis y un 90 por ciento en diseño, implementación y pruebas.

Además de los trabajadores citados anteriormente, el arquitecto debe continuar disponible, aunque el tiempo que dedica a esta fase es, por lo general, menor. Por otra parte, ya que quedan por recopilar alrededor del 20 por ciento de los requisitos, el analista de sistemas y el encargado de especificar los casos de uso continúan su trabajo.

15.2.2. Establecimiento de los criterios de evaluación

Los criterios específicos que deben lograrse en una iteración o en la fase de construcción en su conjunto son exclusivos de cada proyecto, y han sido establecidos, en el curso del desarrollo de los casos de uso. Como hemos mencionado en capítulos anteriores, los casos de uso corresponden por sí mismos a requisitos funcionales, aunque también llevan asociados requisitos no funcionales, tales como rendimiento, y se utilizan para mitigar los riesgos. Cada construcción o iteración implementa un conjunto de casos de uso. Los criterios de evaluación para este conjunto de casos de uso se basan, por tanto, en los requisitos funcionales y no funcionales relacionados con dichos casos de uso.

Estos criterios de evaluación, relativos a los casos de uso, permiten a los propios desarrolladores ver claramente cuándo han completado una determinada construcción o iteración.

Además, durante la fase de construcción se prepara material adicional que también requiere criterios de evaluación. Por ejemplo:

Material de usuario En la fase de construcción se prepara una primera versión de los materiales escritos de ayuda a los usuarios finales, tales como guías de usuario, textos de ayuda, notas de versión, manuales de usuario, etc. Los criterios de evaluación, ¿son suficientes para dar soporte a los usuarios en la fase de transición?

Material de cursos Una versión preliminar de los materiales para cursos que den soporte a los usuarios finales, tales como diapositivas, notas, ejemplos y tutoriales, se preparan también durante esta fase. ¿Son suficientes para dar soporte a los usuarios en la fase de transición?

Los criterios de evaluación durante la fase de construcción, deberán tener presente si la capacidad operativa está suficientemente madura y estable para permitir la entrega de versiones

beta o preliminares a los usuarios, sin que las organizaciones de software y las propias comunidades de usuarios sufran riesgos inaceptables o de difícil control.

15.3. Flujo de trabajo arquetípico de una iteración en la fase de construcción

La iteración arquetípica está formada por los cinco flujos de trabajo, ilustrados en la Figura 15.1 Los flujos de trabajo están descritos detalladamente en la Parte II, pero en este capítulo nos centramos sólo en el papel que desempeñan durante la construcción. De nuevo, desarrollamos cuatro grupos de actividades en paralelo. El primero son los cinco flujos de trabajo fundamentales; el segundo es planificar las iteraciones, tal y como se describe en las Secciones 12.4 a 12.7 y en la Sección 15.2; el tercero es el análisis de negocio, descrito en las Secciones 13.5 y 15.5; y el cuarto es la evaluación, descrita en las Secciones 12.8 y 15.6. En esta sección damos sólo una visión general de los cinco flujos de trabajo fundamentales. En la sección siguiente entraremos en más detalles.

En las primeras iteraciones de la fase de construcción, los flujos de trabajo iniciales reciben mayor énfasis; los últimos, un énfasis menor. Este énfasis se desplaza a lo largo de las iteraciones de construcción. Por ejemplo, si trazásemos una curva de campana para ilustrar las sucesivas cargas de trabajo, el pico de la curva se desplazaría de izquierda a derecha, según cada iteración pusiese mayor énfasis en el flujo de trabajo de implementación.

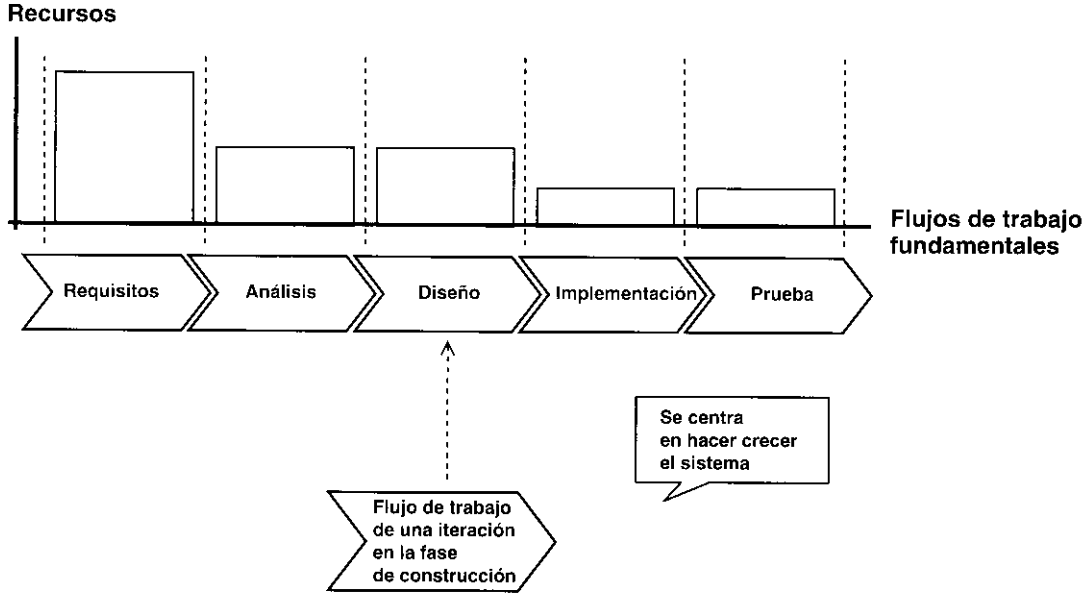


Figura 15.1. El trabajo de una iteración durante la fase de construcción discurre a lo largo de los cinco flujos de trabajo —requisitos, análisis, diseño, implementación y pruebas. Aquí se detallan y analizan los requisitos restantes, pero la carga de trabajo en estos dos flujos es relativamente leve. La mayor parte de este trabajo fue realizada en las dos fases anteriores. El diseño juega un papel importante, y es en esta fase en la que tiene lugar la mayor parte del trabajo de los flujos de implementación y pruebas. (El tamaño de las barras tiene un carácter meramente ilustrativo, y será distinto en diferentes proyectos.)

Construcción del sistema En este momento, los requisitos y la arquitectura son estables. El énfasis se pone a completar las realizaciones de casos de uso para cada uno de ellos, diseñando los subsistemas y clases necesarios, implementándolos como componentes y probándolos tanto de forma individual como en construcciones. En cada construcción, los desarrolladores tomarán un conjunto de casos de uso, como los definieron el jefe de proyecto y el responsable de integrar el sistema, y los realizarán.

Un desarrollo iterativo, guiado por los casos de uso y centrado en la arquitectura, construye el software mediante pequeños incrementos, y añade cada incremento a la acumulación previa de incrementos de tal forma que siempre se tenga una construcción ejecutable. La secuencia de incrementos se ordena de forma progresiva, de modo que los constructores nunca tienen que volver atrás varios incrementos y rehacerlos.

Construir el software en incrementos relativamente pequeños hace que un proyecto sea más manejable. Reduce el ámbito de los flujos de análisis, diseño, implementación y pruebas al número de aspectos y problemas comparativamente menor encontrados dentro de un incremento individual. Aísla en gran parte riesgos y defectos dentro del ámbito de una construcción individual, haciendo que sean más fáciles de encontrar y tratar.

Las fases anteriores han investigado y mitigado los riesgos críticos y significativos, pero habrá aún muchos riesgos en la lista de riesgos. Además, pueden aparecer nuevos riesgos a medida que los desarrolladores realicen construcciones e iteraciones y que los usuarios prueben nuevos incrementos. Por ejemplo, la mayoría de los lenguajes de programación han venido usándose desde hace tiempo, y es perfectamente comprensible que en un proyecto se asuma que el compilador que se planea utilizar funcione de manera satisfactoria. Sin embargo, los lenguajes se amplían y los compiladores se reeditan, y una nueva edición de un compilador puede contener defectos. En un proyecto con 80.000 líneas de código, el jefe de proyecto descubrió finalmente que los repetidos fallos de pruebas sucesivas eran causados por el compilador. A esto siguió un retraso adicional hasta que el fabricante del compilador encontró el sutil defecto y lo corrigió.

15.4. Ejecución de los flujos de trabajo fundamentales, de requisitos a pruebas

En la sección anterior esbozamos el objetivo general de la fase de construcción. En esta sección, presentamos estas actividades con más detalle, utilizando como guía la Figura 15.2. Como en capítulos anteriores, esta sección se estructura en función de los cinco flujos de trabajo. Del mismo modo, aunque la presentación es secuencial, la labor de los flujos de trabajo se desarrolla de forma concurrente, como muestra la figura.

En el curso de este trabajo, los trabajadores participan en los flujos de trabajo tal y como se esbozó en la Parte II, construyendo sobre los artefactos de la línea base de la arquitectura o sobre iteraciones posteriores.

En la fase de elaboración, puede que el proyecto haya diseñado e implementado menos del 10 por ciento de los casos de uso —sólo lo necesario para producir la línea base de la arquitectura. Ahora, en la fase de construcción, el proyecto se enfrenta a la tarea de añadir masa muscular a este esqueleto de la arquitectura. En realidad, se van a completar los modelos ilustrados en los Capítulos 4 y 5. En particular, *véase* la Figura 5.7, que muestra los modelos que se completan en el curso de las cuatro fases. Cada construcción y cada iteración añade más realizaciones de casos de uso, clases, subsistemas y componentes a la estructura evolutiva del modelo.

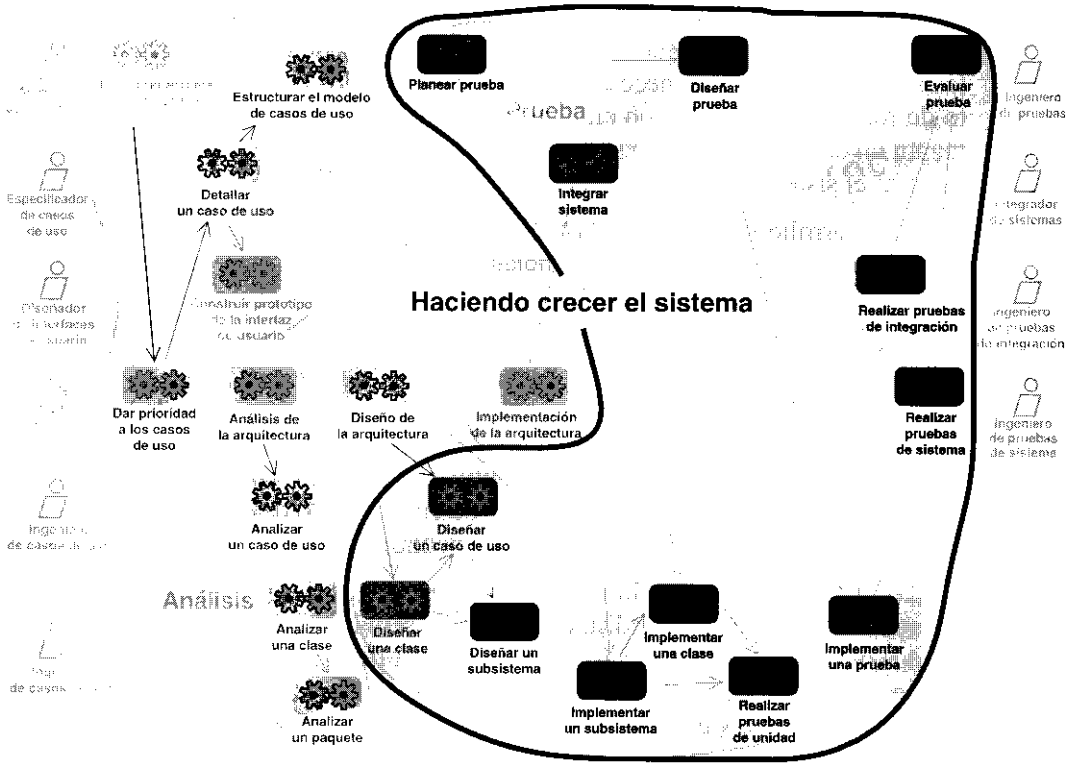


Figura 15.2. En esta figura se resaltan las actividades principales de la fase de construcción.

En las subsecciones siguientes describimos de forma secuencial los flujos de trabajo, como indica la Figura 15.2, pero, en la práctica, los trabajadores no realizan su labor en esta secuencia. Por ejemplo, la planificación de las pruebas se realiza al comienzo de cada iteración, para poner en marcha el trabajo de pruebas necesario en la iteración. Esta planificación de las pruebas puede realizarse antes de detallar, analizar, diseñar e implementar los casos de uso. Este trabajo en paralelo no queda reflejado en el texto a continuación. Merece la pena repetir que los cinco flujos de trabajo se repiten en cada iteración.

15.4.1. Requisitos

Tras esta introducción a la fase de construcción, nos dirigiremos ahora a encontrar casos de uso y actores, construir prototipos de las interfaces de usuario, y detallar y estructurar los casos de uso. En la fase de elaboración, identificamos todos los caso de uso y actores; *comprendimos* alrededor del 80 por ciento, pero sólo describimos en detalle del orden del 20 por ciento —de este número, fuimos capaces de seleccionar el pequeño número (menos del 10 por ciento, lo que necesitamos implementar en aquel momento) porque sólo necesitamos un grado de detalle que bastase para establecer la línea base de la arquitectura. Por supuesto, en la fase de construcción vamos a recorrer todo el camino hasta lograr el sistema inicial con capacidad operativa, así que tenemos que realizar la recopilación completa de requisitos (es decir, identificar y detallar el 100 por cien).

Encontrar actores y casos de uso Normalmente, sólo resta identificar una pequeña fracción de los casos de uso y actores durante la fase de construcción (menos del 20 por ciento). Si es necesario, el analista de sistemas actualizará los casos de uso y actores en el modelo de casos de uso.

Desarrollar un prototipo de la interfaz de usuario Por regla general, en las fases de inicio y elaboración no desarrollamos prototipos de las interfaces de usuario, sólo si teníamos un nuevo tipo de interfaz o necesitábamos un prototipo con objeto de realizar una demostración. Sin embargo, ahora debemos diseñar las interfaces de usuario. Lo que tenemos que hacer aquí depende del tipo de sistema que estemos construyendo.

Para algunos sistemas —especialmente aquéllos en los que algunos casos de uso requieren una interfaz de usuario muy complicada—, es difícil comprender la interfaz de usuario sin construir un prototipo. Así que construiremos un prototipo (o varios si es necesario) y haremos que los usuarios lo prueben. Basándonos en sus respuestas, amoldaremos el prototipo hasta que satisfaga las necesidades de los usuarios. El diseño de la interfaz de usuario es parte del flujo de trabajo de requisitos y no del de diseño (incluso aunque el nombre sugiera lo contrario) y necesita hacerse antes de que podamos trasladarnos a los flujos de trabajo siguientes. El prototipo se convierte entonces en la especificación de la interfaz de usuario del sistema (*véase* el Capítulo 7 para más detalle).

Para sistemas de los que esperemos un gran número de ventas, construiremos un prototipo de la interfaz de usuario incluso si no es muy compleja. El coste de reemplazar una interfaz poco satisfactoria una vez distribuido el producto sería muy grande.

Determinar la prioridad de los casos de uso En la fase de elaboración, clasificamos los casos de uso necesarios para el desarrollo de la línea base de la arquitectura. En esta fase, a medida que identificamos casos de uso, los añadimos a la clasificación, con objeto de establecer su prioridad (*véase* las Secciones 12.6 y 7.4.2).

Detallar un caso de uso Los encargados de especificar los casos de uso detallarán completamente todos los casos de uso y escenarios de caso de uso que queden, trabajando en ellos según su importancia.

Estructurar el modelo de casos de uso El analista de sistemas puede querer mejorar la estructura del modelo de casos de uso. No obstante, debido a que en esta fase el sistema ya tiene una arquitectura estable, cualquier cambio debe referirse fundamentalmente a casos de uso que aún no hayan sido desarrollados. Cada caso de uso modificado, implica la correspondiente realización de caso de uso en los modelos de análisis y diseño.

15.4.2. Análisis

En esta subsección consideraremos de nuevo las actividades de análisis de la arquitectura, analizar un caso de uso, analizar una clase y analizar un paquete, iniciadas en la fase de elaboración. En esa fase, sólo debimos prestar atención a aquellos casos de uso que eran significativos desde un punto de vista de la arquitectura o que eran necesarios para apoyar la apuesta económica. Para dar al lector una idea de dónde nos hallamos en este momento, digamos que podemos haber analizado el 40 por ciento de los casos de uso en la fase de elaboración. Esto es alrededor de la mitad del total de los casos de uso, un 80 por ciento, que normalmente se necesita para realizar el análisis económico. Por favor, no tome los porcentajes de forma literal. Reiteramos que pueden variar de alguna forma dependiendo de las circunstancias de cada proyecto. Ahora, en la fase de construcción, nos interesan todos los casos de uso, pero no necesariamente debemos extender el modelo de análisis con ellos.

Tal como se discutió en el Capítulo 8 (*véase* la Sección 8.3), existen casos en los que el modelo de análisis no se mantendrá después de la primera fase de elaboración de un producto nue-

vo. Sin embargo, en otros casos el jefe de proyecto puede continuar utilizando el modelo de análisis en la fase de construcción y, de hecho, hasta el final del proyecto o a lo largo de la vida del producto. Ya que esta última situación es más compleja, será la que asumamos en lo sucesivo. La diferencia fundamental entre la fase de elaboración y la de construcción estriba en que en la construcción completaremos el modelo de análisis. El modelo de análisis que teníamos al final de la fase de elaboración era la vista de la arquitectura —se refería sobre todo a la arquitectura—. Ahora, la vista de la arquitectura del modelo de análisis será sólo parte del modelo de análisis completo. Al final de la construcción, tenderemos el modelo de análisis completo. La vista de la arquitectura será sólo un pequeño subconjunto de él.

Análisis de la arquitectura Al final de la fase de elaboración, el arquitecto habrá preparado la vista de la arquitectura del modelo de análisis. Por consiguiente, tendrá poco que hacer en el conjunto de actividades de la fase de construcción, aparte de las actualizaciones necesarias debidas a los cambios que afecten a la arquitectura.

Analizar un caso de uso En la fase de elaboración, el arquitecto utilizó solamente aquellos casos de uso que eran arquitectónicamente significativos, y los aplicó a la vista de la arquitectura del modelo de análisis. En cada iteración de la fase de construcción, ampliaremos el modelo de análisis con los casos de uso que sean incluidos en la iteración.

Analizar una clase Los ingenieros de componentes continúan el trabajo que comenzaron en la fase de elaboración.

Analizar un paquete El arquitecto identificó los paquetes durante la fase de elaboración, y los refinó en la fase de construcción, según fuera necesario para acomodar los nuevos casos de uso. El ingeniero de componentes mantendrá los paquetes durante la construcción.

15.4.3. Diseño

Normalmente, en esta fase se diseña e implementa el 90 por ciento restante de los casos de uso —aquéllos que no fueron utilizados para desarrollar la línea base de la arquitectura—. Al considerar el flujo de trabajo de diseño, hacemos énfasis de nuevo en que éste y los otros flujos de trabajo se repiten en cada nueva iteración.

Diseño de la arquitectura Por lo general, en la fase de construcción el arquitecto no añadirá subsistemas del diseño ni subsistemas de servicio. Estos elementos ya existen, en forma de esqueleto, para ser exactos, en la línea base de la arquitectura.

El arquitecto puede añadir subsistemas si son similares, puede ser que alternativos, a los que ya están en la arquitectura. Por ejemplo, si hay un subsistema para un protocolo de comunicación y se añade otro protocolo de comunicación que no necesita nuevas interfaces, entonces es aceptable añadir un nuevo subsistema para ese protocolo de comunicación.

El comportamiento de un subsistema de servicio puede derivarse normalmente de partes de un único caso de uso o de un conjunto de casos de uso relacionados. Otra forma de decir esto es que un subsistema de servicio desempeña un papel dominante para ayudar a realizar un caso de uso. Entre el 40 y el 60 por ciento de las responsabilidades de un subsistema de servicio vienen de un caso de uso de este estilo. Si el porcentaje es más alto, digamos un 80 por ciento, el jefe de proyecto evaluará si el proyecto debe completar el 20 por ciento restante del servicio en la misma construcción o dejarlo para una construcción posterior. Puede ser una buena idea autorizar el

desarrollo completo del servicio en este momento, incluso si otras partes del subsistema provienen de casos de uso de menor rango que el caso de uso dominante. Incluso aunque estos casos de uso restantes tengan una prioridad más baja y puedan ser postergados, puede ser más eficiente completar el paquete entero de una sola vez.

Por un lado, trabajar en los casos de uso de menor prioridad en este momento permite al ingeniero de componentes desarrollar el subsistema entero de una sola vez. Parece más probable que se consiga desarrollar correctamente su estructura en virtud de haber examinado el subsistema entero al mismo tiempo. Si el ingeniero de componentes, o alguna otra persona, debe volver sobre el subsistema en una iteración posterior para trabajar en los casos de uso restantes, de menor prioridad, puede encontrarse con discrepancias que precisen de una modificación por completo del diseño. Se tendría que reescribir código porque no funcionase bien junto con el código añadido que se deriva de los casos de uso de prioridad más baja. Por tanto, es bueno incluir partes basadas en casos de uso que deban ser diseñados posteriormente si —y esto es importante— el impacto que tendrán está claro. La razón es que es mejor hacer una pieza de trabajo completa de una sola vez que repartir sus pedazos a lo largo de varias iteraciones.

Por otro lado, tratar los casos de uso de baja prioridad en una de las primeras construcciones contradice nuestra intención general de examinar primero los casos de uso de prioridad alta. La regla general es que si es conveniente tomar unos cuantos casos de uso de baja prioridad junto a los de alta prioridad y no lleva un tiempo excesivo, entonces, hágase así. Pero, incluso en ese caso, la intención es obtener el diseño correcto, no implementar y probar todos los caso de uso de baja prioridad que hemos tomado. El jefe de proyecto debería postergar estas tareas (diseño, implementación y pruebas de los casos de uso de prioridad baja) hasta una iteración posterior, la que corresponda a su prioridad.

El arquitecto introducirá mejoras en la vista de la arquitectura de los modelos de diseño y despliegue para reflejar la experiencia de la fase de construcción. No obstante, y en general, habrá completado la arquitectura al final de la fase de elaboración, así que lo que hará en este momento es actualizarla. Para una descripción del resto de actividades de este flujo de trabajo (diseño de casos de uso, clases y subsistemas), nos referimos al Capítulo 9. Aquí, basta con decir que el diseño es el principal objetivo de la fase de construcción (y la implementación aún más), tal como indica la Figura 15.1. Su resultado es el modelo de diseño y el modelo de despliegue. Ambos se actualizan a lo largo del ciclo de desarrollo y se transportan a ciclos futuros. El modelo de diseño es el "esquema" del modelo de implementación y de la propia implementación.

15.4.4. Implementación

Este flujo de trabajo implementa y lleva a cabo las pruebas de unidad de todos los componentes, trabajando principalmente a partir del modelo de diseño. El resultado, después de varias iteraciones y de la integración y pruebas del sistema, es la versión operativa inicial, que representa el 100 por cien de los casos de uso. En esta subsección tratamos las actividades de implementación de la arquitectura, implementar una clase e implementar un subsistema, realizar pruebas de unidad, e integrar el sistema.

Es en este flujo de trabajo en el que el proyecto lleva a cabo la mayor parte del trabajo de la fase de construcción, construyendo los componentes, como se describe en el Capítulo 10. El proyecto rellena cada componente con más y más código, construcción tras construcción, iteración tras iteración, hasta que al final de la fase de construcción, todos los componentes están "llenos".

Implementación de la arquitectura En este momento la arquitectura está firmemente asentada. El papel del arquitecto, en lugar de realizar una vigilancia continua, será sólo el de actualizarla si es necesario.

Implementar una clase e implementar un subsistema Los ingenieros de componentes implementarán las clases y subsistemas del modelo de implementación. Implementarán clases *stub* cuando sea necesario para armar una construcción.

Realizar pruebas de unidad El ingeniero de componentes será el responsable de realizar las pruebas de unidad del componente que fabrique. Si es necesario, corregirá el diseño y la implementación del componente.

Integrar el sistema El responsable de la integración del sistema creará un plan de integración de construcciones que perfile la secuencia de construcciones. Este plan mostrará los casos de uso o escenarios de un caso de uso que va a implementar la construcción. Estos casos de uso y escenarios nos conducirán a los subsistemas y componentes.

Los responsables de la integración de sistemas normalmente encuentran aconsejable empezar por construir las capas inferiores de la jerarquía de la arquitectura en capas, como la capa del software del sistema o la capa intermedia (ilustradas en la Figura 4.5). Las construcciones siguientes se expandirán hacia arriba, hacia las capas general y específica de la aplicación. Es difícil montar una construcción sin tener las funciones de apoyo que proporcionan las capas inferiores.

Por ejemplo, la organización software de una determinada empresa química está pensada para realizar un incremento, en promedio, cada dos semanas. Se dice que los proyectos en Microsoft Corporation desarrollan una construcción cada día —una vez que el proyecto llega al punto de tener una construcción de código. Cada construcción es sometida a pruebas —pruebas de las nuevas adiciones y pruebas de regresión del código añadido con el ya existente— para asegurar que el código en desarrollo es capaz de funcionar. Este proceso diario de construcción comprueba cada día que las unidades de código comprobadas el día anterior son compatibles. Esta práctica presiona a los desarrolladores para que no "rompan la construcción". Al mismo tiempo, sin embargo, reduce la presión a largo plazo sobre la organización software, ya que los problemas de integración son descubiertos durante las pruebas, normalmente cada noche, y resueltos poco después. El hacer una construcción cada día puede parecer que impone una considerable presión temporal sobre un proyecto, pero no necesariamente. Los desarrolladores comprobarán su código cuando consideren que está listo. No tiene demasiado sentido comprobar código que no esté listo, posiblemente eso rompería la construcción. Sin embargo, los desarrolladores individuales estarán bajo la presión de comprobar su código a tiempo de cumplir con el plan de integración de construcciones.

La duración de cada periodo de construcción es ciertamente un asunto que cada organización debe considerar. El principio director es construir con una frecuencia suficiente como para obtener las ventajas que las construcciones frecuentes traen consigo.

Para limitar el tamaño de cada construcción, a menudo el responsable de la integración del sistema encargará un *stub* o un *driver* para representar un componente que no haya sido construido aún. Un *stub* es un elemento muy sencillo que simplemente proporciona una respuesta a un estímulo —o a todos los estímulos que el componente puede recibir de otros componentes (aún no completados) de la construcción. De forma similar, un *driver* proporciona un estímulo a los otros componentes que son parte de la construcción que se está probando. Debido a que son sencillos, no es probable que *stubs* y *drivers* introduzcan complicaciones adicionales.

Por tanto, el responsable de la integración del sistema, tendrá en cuenta el orden en el que ensamblar los componentes, con objeto de formar una configuración estable y que pueda ser probada. Documentará sus hallazgos en el plan de integración de construcciones, que mostrará en qué momento es necesaria cada construcción para cumplir con el plan de integración y pruebas. El responsable de la integración del sistema reunirá clases completas y *stubs* en una construcción ejecutable, de acuerdo con el plan de construcción, y hará construcciones cada vez mayores, mientras los encargados de las pruebas de integración y regresión las comprueban. En el paso final de cada iteración, y por último de la propia fase, el responsable de la integración del sistema conectará el sistema en su conjunto, y los responsables de probar el sistema y de realizar las pruebas de regresión lo comprobarán.

Esta planificación establecerá una secuencia que ordenará las construcciones de cada iteración y las iteraciones de la fase de construcción. Debido a que a menudo existen dependencias de compilación de las capas superiores de la arquitectura respecto a las capas inferiores, los responsables de la integración del sistema podrán tener que planificar la compilación de abajo a arriba.

15.4.5. Pruebas

Los esfuerzos de los ingenieros de pruebas para descubrir lo que puede ser comprobado de forma efectiva y para desarrollar casos de pruebas y procedimientos de pruebas para hacerlo, descritos en el Capítulo 11, tendrán su fruto en la fase de construcción. Ésta es una actividad fundamental de esta fase, como muestra la Figura 15.1. Debido a que realizar pruebas de unidad es responsabilidad de los ingenieros de componentes, los ingenieros de pruebas deberán estar disponibles para proporcionar asistencia técnica. No obstante, los ingenieros de componentes y sus colaboradores, el encargado de las pruebas de integración y el encargado de las pruebas del sistema, son los responsables de la comprobación de las construcciones, es decir, los incrementos al final de las iteraciones, y en última instancia, de la construcción final, que constituye la versión completa del sistema.

Planificar las pruebas Los ingenieros de pruebas seleccionarán los objetivos que comprueben las sucesivas construcciones y, por último, el propio sistema.

Diseñar las pruebas Los ingenieros de pruebas, determinarán cómo probar los requisitos en el conjunto de construcciones, con objeto de verificar los requisitos que puedan ser comprobados. Prepararán casos y procedimientos de prueba con este propósito. De los casos y procedimientos de prueba de construcciones precedentes, seleccionarán aquéllos que aún sean pertinentes y los modificarán para utilizarlos en las pruebas de regresión de las construcciones siguientes. Los ingenieros de pruebas verificarán los componentes que deberán ser comprobados de forma conjunta, tal y como se estableció originalmente en el plan de pruebas. El propósito de estas pruebas de integración es verificar las interfaces entre los componentes que estén siendo probados y comprobar que los componentes pueden trabajar conjuntamente.

Realizar pruebas de integración Los encargados de las pruebas de integración ejecutarán los casos de prueba, siguiendo los procedimientos de prueba. Si la construcción supera las pruebas, el responsable de la integración del sistema añadirá construcciones adicionales, según vayan estando disponibles, y el encargado de las pruebas de integración seguirá comprobándolas. Si las pruebas de integración detectan fallos, los encargados de las pruebas los registrarán y notificarán al jefe de proyecto. El jefe de proyecto, o la persona que éste designe, determinará cuál será el siguiente paso a dar. Esta persona puede ser el responsable de la integración del sistema,

el cual posee el conocimiento técnico pertinente. El siguiente paso podría ser, por ejemplo, prolongar el trabajo en la misma construcción, diferir la corrección del defecto hasta la construcción siguiente o, en particular en el caso de un fallo especialmente grave, asignar una serie de personas especialmente cualificadas para investigarlo.

Realizar pruebas del sistema En el momento en que las sucesivas construcciones alcancen el final de una iteración, habrán alcanzado el status de versión parcial del sistema, y entrarán en la jurisdicción del encargado de las pruebas del sistema. Éste, ejecutará los casos de prueba del sistema, siguiendo los procedimientos de prueba del sistema. Si estas comprobaciones detectan fallos, el encargado de las pruebas del sistema los registrará y notificará al jefe de proyecto o la persona que éste designe para su resolución.

Al final de la última iteración de la fase de construcción, el encargado de las pruebas del sistema comprobará la versión operativa inicial. De nuevo, notificará los fallos al jefe de proyecto para que encargue al ingeniero de componentes responsable de su corrección.

Evaluar las pruebas A medida que transcurren las pruebas de integración y del sistema, los ingenieros de pruebas revisarán los resultados de las pruebas al final de cada construcción a la luz de los objetivos originalmente fijados en el plan de pruebas (posiblemente modificado durante las sucesivas iteraciones). El propósito de evaluar las pruebas es asegurarse de que éstas alcancen sus objetivos. Si una prueba no alcanza sus objetivos, los casos y procedimientos de prueba deberán ser modificados para lograrlos (*véase* la Sección 11.5.6).

15.5. Control del análisis de negocio

Uno de los propósitos de la apuesta de negocio, desarrollada al final de la fase de elaboración, es el de servir de guía al jefe de proyecto y los inversores para ejecutar la fase de construcción. Con este objetivo, el jefe de proyecto comparará el progreso real al final de cada iteración con la agenda, esfuerzo y costes planificados. Revisará los datos de productividad del proyecto, cantidad de código desarrollado, tamaño de la base de datos y otras métricas.

Rara vez el número de líneas de código completadas es una buena medida del progreso en un desarrollo basado en componentes. Debido a que uno de sus objetivos es la reutilización de clases y componentes, un ingeniero de componentes u otros trabajadores pueden realizar un buen progreso con las construcciones e iteraciones y sin embargo escribir poco código nuevo. Una medida más pertinente del trabajo desarrollado, en estas circunstancias, es la finalización de las construcciones e iteraciones de acuerdo con el plan.

Discrepancias mayores de un pequeño porcentaje, especialmente en sentido negativo, llevarán al jefe de proyecto a tomar medidas. De forma similar, discrepancias de mayor tamaño llevarán a reuniones de revisión con los inversores.

A medida que el jefe de proyecto adquiere un mayor conocimiento sobre los costes y capacidades del producto durante la fase de construcción, puede encontrar necesario actualizar el análisis de negocio y comunicar el nuevo análisis a los inversores.

15.6. Evaluación de las iteraciones y la fase de construcción

Sobre la base de una revisión de los resultados de las pruebas y otros criterios de evaluación, incluyendo los esbozados en la Sección 15.2.2, el jefe de proyecto y el grupo de evaluación:

- Revisarán lo logrado en una iteración contrastándolo con lo que había sido planificado.
- Planificarán en cuál de las iteraciones siguientes se habrá de llevar a cabo el trabajo no completado.
- Determinarán que la construcción está lista para entrar en la siguiente iteración.
- Actualizarán la lista de riesgos.
- Completarán el plan de la iteración siguiente.
- Al final de la última iteración de esta fase, determinarán que el producto ha superado las pruebas del sistema y que ha alcanzado la capacidad operativa inicial.
- Autorizarán la entrada en la fase de transición.
- Actualizarán el plan de proyecto.

15.7. Planificación de la fase de transición

El equipo de proyecto no puede esperar planificar de antemano la fase de transición con tanto detalle como hizo para las fases anteriores. Sus miembros saben, por supuesto, que van a producir versiones beta (o su equivalente) para que las evalúen usuarios seleccionados. Esta parte de la fase de transición: seleccionar a los encargados de probar las versiones beta, reproducir el código de operación, preparar instrucciones de pruebas, etc. es la que se puede planificar con cierto detalle.

La respuesta recibida —riesgos, problemas, fallos, sugerencias— difícilmente puede preverse de antemano. Si el equipo ha tenido experiencia en pruebas beta, podrá hacerse una idea de lo que cabe esperar. Será capaz de estimar la cantidad aproximada de personal experimentado necesario para abordar los problemas que los usuarios de la beta descubran.

15.8. Productos clave

Los productos son:

- El plan de proyecto para la fase de transición.
- El sistema software ejecutable —la versión con capacidad operativa inicial—. Ésta es la construcción final de la fase.
- Todos los artefactos, incluyendo los modelos del sistema.
- La descripción de la arquitectura, mínimamente modificada y actualizada.
- Una versión preliminar del manual de usuario, lo suficientemente detallado como para guiar a los usuarios de la beta.
- El análisis de negocio, que refleja la situación al final de la fase.

El objetivo es que los productos etiquetados como "completos" lo sean realmente. La operación del software en la comunidad de usuarios durante la fase de transición puede desvelar que algunos de ellos no son realmente correctos. En ese caso, se modificarán para que lo sean.

Capítulo 16

La transición completa la versión del producto

Una vez que el proyecto entra en la fase de transición, el sistema ha alcanzado la capacidad operativa inicial.

El jefe de proyecto ha considerado que el sistema ofrece la confianza suficiente como para operar en el entorno del usuario, aunque no sea necesariamente perfecto. Por ejemplo, algunos problemas, riesgos y defectos que no se han puesto en evidencia durante las pruebas del sistema, al final de la fase de construcción, pueden ponerse de manifiesto en el entorno del usuario. El usuario puede descubrir con retraso la necesidad de determinadas características. Si son muy importantes y casan bien con el producto existente, el jefe de proyecto puede aceptar añadirlas. Sin embargo, los cambios deben ser lo suficientemente pequeños como para que puedan ser introducidos sin afectar seriamente el plan de proyecto. Si una característica propuesta afecta a la planificación, su necesidad debe ser aguda. En la mayoría de los casos, consideramos que es mejor añadirla a la lista de características y dejarla para el siguiente ciclo de desarrollo, es decir, para el desarrollo de la siguiente versión del sistema.

Los objetivos básicos de esta fase son:

- Cumplir los requisitos, establecidos en las fases anteriores, hasta la *satisfacción* de todos los usuarios.
- Gestionar todos los aspectos relativos a la operación en el entorno del usuario, incluyendo la corrección de los defectos remitidos por los usuarios de la versión beta o por los encargados de las pruebas de aceptación.

Las pruebas de aceptación son responsabilidad del cliente, aunque algunos pueden contratarlas a una organización especializada en pruebas de aceptación.

16.1. La fase de transición en pocas palabras

Esta fase se centra en implantar el producto en su entorno de operación. La forma en que el proyecto lleva a cabo este objetivo varía con la naturaleza de la relación del producto con su mercado. Por ejemplo, si un producto va a salir al mercado, el equipo de proyecto distribuye una versión beta a usuarios típicos pertenecientes a organizaciones "clientes beta" que sean representativas. Si un producto va a distribuirse a un único cliente (o quizás a una serie de usuarios en una organización grande), el equipo instala el producto en un solo sitio.[1]

El proyecto recibirá información de los usuarios para:

- Determinar si el sistema hace lo que demandan sus usuarios y el negocio.
- Descubrir riesgos inesperados.
- Anotar problemas no resueltos.
- Encontrar fallos.
- Eliminar ambigüedades y lagunas en la documentación de usuario.
- Centrarse en áreas en las que los usuarios muestren deficiencias y necesiten información o formación.

Partiendo de una respuesta de este tipo, el equipo del proyecto modifica el sistema o los artefactos relacionados. El equipo prepara la producción del producto, el empaquetado, la distribución y lanzamiento al mercado en general.

En esta fase no se busca reformular el producto. El cliente y el equipo del proyecto deberían haber incorporado cambios significativos en los requisitos durante las fases anteriores. Por el contrario, el equipo busca pequeñas deficiencias que pasaron desapercibidas durante la fase de construcción y que pueden ser corregidas en el marco de la línea base de la arquitectura existente.

En su relación con el cliente, el equipo puede también proporcionar ayuda para crear un entorno apropiado para el producto, y en la formación de la organización del cliente para que utilice el producto de forma efectiva. Puede ayudar a los usuarios a llevar en paralelo la operación del nuevo sistema con el sistema al que reemplaza. Puede ayudar a la conversión de bases de datos a la nueva configuración.

En el caso de productos para muchos usuarios (para el "mercado del software enlatado") estos servicios se construyen en el programa de instalación que es parte del producto, complementado con el servicio de asistencia técnica.

La fase de transición finaliza con el lanzamiento del producto.

---

[1] Debemos mencionar también otras dos posibilidades: pruebas alfa y validación por un tercero. Las pruebas alfa son similares a las pruebas beta, excepto en que se realizan dentro de la empresa que desarrolla el software, aunque fuera de la propia organización de desarrollo. Las personas que realizan las pruebas alfa son usuarios reales. En la validación por un tercero, una empresa especializada en pruebas realiza pruebas de aceptación por encargo del cliente.

16.2. Al comienzo de la fase de transición

El software se desarrolla bajo distintos tipos de acuerdos comerciales. Sin entrar en demasiado detalle, podemos agrupar esos acuerdos en dos patrones:

Producción por parte de un fabricante de software para la venta en un determinado mercado a muchos clientes A veces este mercado es muy grande, como el mercado de los computadores personales. Otras, es más pequeño, como es el caso de la producción de componentes reutilizables para fabricantes de software, o de programas prefabricados que puedan ser adaptados a cada instalación. En estos mercados, aunque el tamaño pueda variar, se da una relación *uno a muchos* (un vendedor, muchos clientes) que influye en las tareas de la fase de transición.

Producción por parte de una casa de software bajo contrato con un único cliente Este patrón tiene diversas variaciones:

- La organización software trabaja para un único cliente con una única sede.
- La organización trabaja para un cliente con muchas suborganizaciones y sedes.
- La organización software, como EDS, Computer Science o Andersen Consulting, que a veces desarrolla software inicialmente para un único cliente o sede, pero que después lo adapta para otras sedes o clientes.
- La organización software desarrolla software para otros departamentos de la misma empresa bajo acuerdos contables específicos.

La relación entre la organización de la fase de transición y el usuario o cliente varía, dependiendo de los distintos patrones. Según estos patrones, la fase de transición comienza con una versión operativa inicial que ha pasado por pruebas internas del sistema y por la evaluación del hito principal del final de la fase de construcción. No obstante, el equipo del proyecto en la fase de transición prepara artefactos adicionales, como programas de instalación, programas de conversión o de migración de datos, o modifica los desarrollados durante la fase de construcción para preparar el programa ejecutable para su distribución más allá de sus propios límites.

16.2.1. Planificación de la fase de transición

No debemos pretender planificar de forma detallada y por adelantado la fase de transición de un proyecto, como se hizo con la fase de construcción. Por un lado, el jefe de proyecto sabe que esta fase va a partir de versiones beta (o su equivalente), desarrolladas en la fase de construcción, para que las prueben los usuarios. Este conocimiento es la base para hacer una planificación inicial de la fase de transición. La cantidad de trabajo relativa a la producción de la versión beta, la preparación de la documentación de las pruebas, la selección de los usuarios, etc., es conocida. Por otro lado, habrá una cantidad de trabajo desconocida, en función de los resultados de las pruebas beta. El jefe de proyecto querrá tener a algunas personas a la espera. Puede asignarlos para trabajar en otros proyectos, pero teniéndolos disponibles para trabajar a destajo en los problemas de la transición.

Al planificar la fase de transición, el jefe de proyecto presupone que la versión operativa inicial de la fase de transición requerirá poca reelaboración como resultado de las pruebas beta. De

hecho, éste debería ser el caso, si el proceso de desarrollo ha sido realizado de forma iterativa. Este proceso de desarrollo permite a los desarrolladores experimentar durante las iteraciones iniciales, y descubrir sus errores de concepto en las pruebas de los incrementos iniciales y en la observación de su operación. De forma similar, los errores deberían ser detectados y corregidos, construcción a construcción, según se desarrolla el trabajo. En resumen, la reelaboración temprana es *buena*. En la fase de transición, como consecuencia del desarrollo iterativo, la reelaboración debería ser menor del 5 por ciento. Sin embargo, los jefes de proyecto deberían considerar que será mayor que cero. Al menos un pequeño número de omisiones y errores superarán todos los controles. La tendencia de un proyecto a despreciar la posibilidad de reelaboración puede ser el resultado de:

- Demasiada presión de la agenda, que conduce a cometer errores debidos a las prisas.
- Ausencia de pruebas del sistema y evaluación adecuadas al final de la fase de construcción.
- Error al evaluar el considerable trabajo que aún queda en la fase de transición.
- El sentimiento de que incluso considerar la necesidad de reelaboración hará que esa necesidad se materialice.
- La disposición de considerar la reelaboración como "mala", como una admisión de la incompetencia del proyecto.

El problema del software "lo bastante bueno" puede aparecer en la planificación de la fase de transición. Es un hecho que ningún producto software es perfecto. Por ejemplo, los productos se distribuyen con un algún porcentaje de defectos remanentes, con algunos requisitos postergados a versiones posteriores o con algunas necesidades descubiertas por los usuarios de la beta que la fase de transición carece de recursos para satisfacer. Hay tres respuestas para el problema del "lo bastante bueno".

En primer lugar, las fases e iteraciones del Proceso Unificado están diseñadas para identificar los riesgos, recoger de forma precisa los requisitos, y planificar el proyecto de acuerdo a esto. En el Proceso Unificado, el equipo del proyecto lleva a cabo estos esfuerzos en conjunción con los usuarios y clientes. Por tanto, la versión operativa inicial, o versión beta, debería estar próxima a lo que tanto el proyecto como los clientes esperan.

En segundo lugar, debido a que ni el proyecto ni los clientes pretenden que la versión operativa inicial esté libre de errores, se han reservado algunos recursos para la fase de transición.

En tercer lugar, debido a que el proyecto ha alcanzado las dos primeras "respuestas" en cooperación con los clientes, el resultado puede ser la ampliación de la fase o la postergación del trabajo inesperado hasta el siguiente ciclo de desarrollo.

16.2.2. Asignación de personal para la fase

Ya que el objetivo de esta fase consiste en proporcionar una versión para los usuarios, primero para las pruebas beta o de aceptación, y después para ser utilizada, las necesidades de personal son similares a las de la fase de construcción, aunque el énfasis puede ser algo diferente.

El análisis del resultado de las pruebas beta o de aceptación y la respuesta a este resultado puede necesitar de personas que estén más orientadas al servicio que al desarrollo. El considerar la bondad de incluso una pequeña mejora puede requerir de expertos no sólo en grandes par-

tes del sistema, sino en la naturaleza de la aplicación a la cual está siendo aplicado. Además, mientras las pruebas simplemente encuentran el defecto, realizar su traza hasta el origen requiere a menudo una visión profunda de todo el sistema, o al menos de la parte en la que aparece haberse originado el fallo. Más aún, el material de usuario y cursos, aunque iniciado en la fase de construcción, debe ser reescrito de forma técnica antes de que el producto sea distribuido a los clientes normales. Durante esta fase, el arquitecto ha de estar disponible, pero principalmente para asegurar y mantener la integridad de la arquitectura, y a veces (aunque pocas) para considerar pequeños cambios de la arquitectura.

16.2.3. Establecimiento de los criterios de evaluación

Al final de la fase de construcción, después de las pruebas del sistema, se consideró que el producto alcanzaba la capacidad operativa inicial, o en otras palabras, cumplía con los términos de la especificación de requisitos. Para la fase de transición, es necesario evaluar sólo los problemas que surjan en esta fase. Básicamente, hay cinco aspectos:

1. ¿Han cubierto los usuarios beta las funciones clave, por ejemplo las representadas por todos los casos de uso, relativas a la operación con éxito del producto en el campo de aplicación?
2. De forma similar, ¿ha superado el producto las pruebas de aceptación realizadas por el cliente? Los criterios de prueba vienen fijados por el contrato, que ha sido acordado tanto por la organización de desarrollo como por el cliente. Además, y de forma general, las pruebas de aceptación hacen funcionar el sistema en modo de producción durante un periodo de tiempo previamente acordado.
3. ¿Tiene el material de usuario una calidad aceptable?
4. ¿Está listo el material de cursos necesario (incluyendo la guía del profesor, en su caso)?
5. Por último, ¿parecen los usuarios y clientes satisfechos con el producto? Profundizaremos sobre este aspecto en la Sección 16.4.6.

16.3. Los flujos de trabajo fundamentales desempeñan un papel pequeño en esta fase

En esta fase, la actividad es baja en los cinco flujos de trabajo, como muestra la Figura 16.1. Como casi todo el trabajo se realizó en la fase de construcción, el nivel de actividad en esta fase es bajo, justo lo necesario para corregir los problemas encontrados durante las pruebas en el entorno del usuario. Sin embargo, esto no quiere decir que no haya una buena cantidad de trabajo en esta fase (*véase* la Sección 16.4 para más detalles). No debería haber apenas trabajo en los flujos de trabajo de requisitos, análisis y diseño. Normalmente, las actividades de diseño disminuyen durante la fase de transición, y en cualquier caso, consisten por lo general en poco más que hacer pequeñas mejoras de diseño destinadas a corregir problemas o defectos o para efectuar mejoras de última hora (por lo general, menores). La atención se desplaza a la corrección de defectos para eliminar los fallos que ocurran durante el uso inicial, a asegurarse de que las correcciones en sí están bien, y a hacer pruebas de regresión para asegurar que estas modificaciones no introduzcan nuevos defectos.

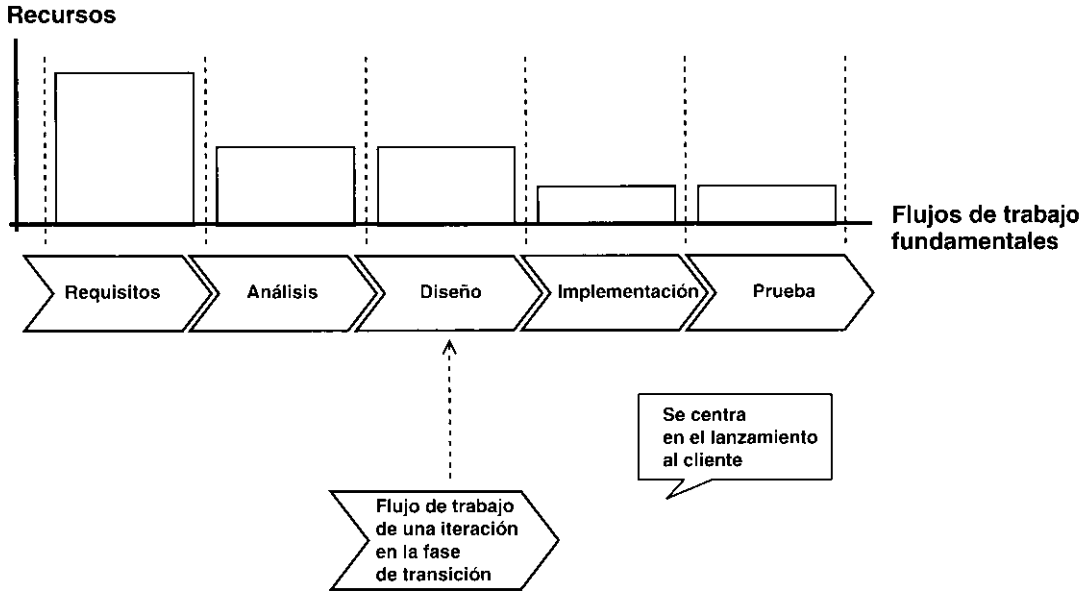


Figura 16.1. El trabajo de una iteración durante la fase de transición discurre a lo largo de los mismos cinco flujos de trabajo que las fases anteriores. (El tamaño de las barras tiene un carácter meramente ilustrativo, y será distinto en diferentes proyectos.) Los flujos de trabajo de implementación y pruebas son los que reciben mayor atención en esta fase. Es aquí donde los defectos encontrados durante las pruebas beta o las de aceptación se identifican, corrigen y prueban de nuevo. En unos pocos casos será necesario rediseñar, para corregir los defectos, lo que llevaría a un pequeño aumento en el flujo de trabajo de diseño. Los requisitos no se ven muy afectados.

La iteración arquetípica está formada por los cinco flujos de trabajo ya discutidos en capítulos anteriores. En este capítulo nos centramos sólo en el papel que juegan durante la transición. Más ampliamente, realizaremos cuatro grupos de actividades en paralelo. Una son los cinco flujos de trabajo en la iteración o iteraciones; la segunda es la planificación de las iteraciones, tal y como se describe en las Secciones 12.4 a 12.7; la tercera es un análisis más profundo del análisis de negocio, como se discute en la Sección 16.5, y la cuarta es la evaluación, descrita en las Secciones 12.8 y 16.6. En esta sección proporcionamos solamente una visión general de la fase de transición, que detallamos en la sección siguiente.

16.4. Lo que se hace en la fase de transición

El proyecto lleva a cabo las actividades de transición según las siguientes líneas:

- Preparar la versión beta (o de pruebas de aceptación) a partir de la versión con capacidad operativa inicial producida durante la fase de transición.
- Instalar (o preparar la instalación de) esta versión en los lugares elegidos, junto con las actividades relacionadas en dichos lugares, tales como la migración de datos desde el sistema anterior.
- Actuar a partir de la información recogida en las instalaciones de pruebas.
- Adaptar el producto corregido a las circunstancias de los usuarios.

- Completar los artefactos del proyecto.
- Determinar cuándo se acaba el proyecto.

Los detalles de esta secuencia de actividades variarán dependiendo de si el proyecto consiste en la producción de software de cara al mercado o para un cliente concreto. En el primer caso, habrá muchos usuarios potenciales, y la selección y dirección de los usuarios beta será una labor considerable. Además, como usuarios reales, no seguirán un plan de pruebas preestablecido, sino que usarán el producto como quieran e informarán de lo que encuentren.

En el segundo caso, probablemente el cliente seleccionará un lugar de pruebas para la instalación inicial, y las pruebas de aceptación seguirán un procedimiento formal, sistemático y acordado de antemano. Los informes resultantes se referirán fundamentalmente a desviaciones de las especificaciones formales. Si surgen problemas que estén más allá del ámbito del acuerdo contractual existente, las partes acordarán una ampliación del contrato.

También variará el esquema de las actividades, dependiendo de si el producto software es novedoso o si es una mejora de un producto ya existente o similar. De nuevo, ésta es una ocasión para que las organizaciones software especialicen el proceso a las circunstancias que afrontan, como se discute en el Capítulo 2.

16.4.1. Preparación de la versión beta

La mayor parte del conjunto inicial de usuarios para las pruebas beta (o de aceptación) serán usuarios experimentados. La organización de transición pretenderá que trabajen a partir de documentación relativamente preliminar, pero también les proporcionará instrucciones específicas de cómo informar de los hallazgos de sus pruebas y observaciones. Al principio de la fase de transición, el equipo del proyecto reunirá la documentación, preparada con anterioridad, que necesiten los usuarios beta o los encargados de las pruebas de aceptación. Esta información estará complementada con instrucciones específicas para las pruebas beta. Se seleccionará a los usuarios de la beta y se les distribuirá la versión beta y el material de acompañamiento.

16.4.2. Instalación de la versión beta

Las actividades en los lugares de prueba varían entre las pruebas beta y las de aceptación. Normalmente, habrá un gran número de lugares donde se realicen pruebas beta, y el personal de transición no estará presente en ellos, por lo que deben darse instrucciones específicas sobre cómo instalar el software de pruebas, cómo operar con él, en qué aspectos y problemas deben centrarse los clientes y usuarios de la beta, y cómo informar de los fallos y problemas encontrados. Si la versión es una actualización o sustitución de un software existente, el personal de transición debe proporcionar instrucciones a los usuarios beta sobre cómo deben migrar los datos o convertir las bases de datos a la nueva versión. Es posible que estas instrucciones deban indicar la operación en paralelo de la versión beta con el software existente durante un cierto periodo de tiempo.

Por el contrario, en las pruebas de aceptación, por lo general estarán presentes miembros del personal del proyecto. Habrá un documento formal de pruebas de aceptación, aunque será complementado mediante comunicaciones informales. Si es posible, los fallos y problemas se corregirán sobre el terreno, o se remitirán a los miembros del proyecto que tengan la cualificación adecuada, en caso de que sea necesario.

16.4.3. Reacción a los resultados de las pruebas

El proyecto recopila y analiza los resultados de las pruebas con el objeto de llevar a cabo acciones. Por lo general, los resultados entrarán en dos categorías: fallos de codificación relativamente menores, que simplemente tienen que ser localizados y corregidos (aunque esto puede ser difícil en algunos casos) y problemas más importantes que pueden tener ramificaciones más extensas, incluso hasta la posibilidad de un cambio en la arquitectura.

Fallos En primer lugar, un fallo resulta de un defecto en un componente, pero ese defecto puede tener que ser rastreado hasta una deficiencia en los modelos de diseño, análisis, e incluso anteriores. Si la traza es difícil, el personal de transición puede tener que buscar ayuda del ingeniero de componentes o del resto del personal que realizó el trabajo original en el área. De todos modos, una persona competente corregirá el defecto y el personal de pruebas lo probará y realizará las pruebas de regresión. Además, el proyecto se planteará cuestiones como:

- ¿Parece verosímil que el defecto esté relacionado con otros aún no descubiertos?
- ¿Puede ser corregido sin que afecte a la arquitectura o el diseño?
- ¿Ha sido corregido sin introducir nuevos defectos?

Problemas más amplios Algunas de las dificultades encontradas durante las pruebas beta pueden precisar una consideración más extensa que "corregir un fallo". Por ejemplo, pueden ampliarse hasta el grado de requerir una iteración de pruebas adicional. Pueden sugerir cambios, mejoras o características que deban ser tratadas formalmente, por ejemplo, a través de un Comité de Control de Cambios. A este respecto, a medida que se implementan los cambios, queremos hacer hincapié en la importancia de realizar el control de configuraciones. Como hemos dicho anteriormente, los cambios sustanciales que excediesen los recursos, retrasasen la distribución o modificasen la arquitectura, deberían postergarse, si es posible, hasta el siguiente ciclo de desarrollo.

Es importante mantener la integridad de la arquitectura del sistema al corregir problemas y defectos. Con este objetivo, el arquitecto supervisará el progreso del trabajo de transición. Trabajará en "modo de seguimiento" para asegurarse de que la corrección de problemas y defectos respeta la arquitectura. El arquitecto (o el equipo de arquitectura) se asegurará de que los problemas no se corrijan, por cuestión de conveniencia, de forma que se dañe la arquitectura. Lograr este objetivo requiere a veces un pequeño ajuste fino de la arquitectura. Si, finalmente, la actividad en esta fase afecta a la arquitectura, el arquitecto actualizará su descripción.

Por supuesto, pocas organizaciones software lanzan productos perfectos. A este respecto, el jefe de proyecto tiene que medir los costes y retrasos de elementos concretos que se deban reelaborar como aspectos del análisis de negocio.

16.4.4. Adaptación del producto a entornos de usuario variados

Como indicamos anteriormente, las organizaciones software elaboran productos de acuerdo a dos grandes formas de relación mercantil: el mercado de productos (uno a muchos) y el cliente individual (uno a uno). En cada relación, puede haber tareas adicionales de migración de datos o conversión de una base de datos de un sistema viejo a uno nuevo.

Relación de mercado El mercado puede consistir en un conjunto altamente diversificado de destinatarios, para el que el equipo tiene que preparar diferentes versiones del programa ejecutable. Algunas de las variantes incluyen el país, el idioma, la moneda y otras unidades de medida. Si el producto va a funcionar sobre nodos diferentes de una red, puede tener que ser modificado para cada nodo.

Probablemente, los usuarios de la primera versión comercial sean menos experimentados que los usuarios beta, así que el equipo ampliará la documentación preliminar de las pruebas beta para ajustarse a las necesidades de usuarios normales.

Por lo general, los productos de mercado son instalados por el usuario o por el administrador de sistemas de una empresa. En otros caso, un vendedor local realiza la instalación. El equipo preparará procedimientos para ser seguidos por el instalador y un guión para el servicio de asistencia telefónica. Estos procedimientos pueden ser muy complejos, por ejemplo, en casos en los que el producto va a instalarse sobre una serie de ordenadores personales o estaciones de trabajo conectados a través de una red privada. Los procedimientos serán aún más complejos cuando se instalen partes del producto en nodos diferentes, y estos nodos deban ser configurados de un modo determinado. Diferentes tipos de nodos pueden precisar diferentes procedimientos de instalación.

Relación de cliente individual La transición de un sistema a un único cliente sujeto a un acuerdo contractual sigue en gran parte el patrón que acabamos de enumerar. Sin embargo, existen unas pocas diferencias:

- Probablemente, representantes del cliente han participado en las fases anteriores, proporcionando informes sobre los incrementos.
- Han observado las pruebas finales del sistema, de acuerdo a las premisas del contratista software.
- Han participado en algunas de las sesiones de evaluación que marcan la superación del hito de construcción.
- Probablemente, la organización software ha ayudado a instalar el sistema en la sede inicial del cliente. En el caso de sistemas grandes y complejos, puede haber realizado el grueso del trabajo de instalación.
- Los representantes del contratista han observado las pruebas de aceptación y pueden haber realizado correcciones sobre el terreno cuando esto fuese posible. En caso de problemas más serios, han remitido los detalles a su propia organización para conseguir la ayuda de expertos para realizar los cambios y correcciones.
- Las pruebas de aceptación concluyen cuando el cliente y el proveedor determinan que el sistema cumple con los requisitos previamente acordados entre ambos. Por supuesto, pueden haberse detectado necesidades adicionales o cambios en las necesidades que lleven a una ampliación del contrato.

Migración y conversión de datos Por lo general habrá un sistema ya existente que va a ser reemplazado, lo que lleva a la necesidad de procedimientos de migración de datos o de conversión de bases de datos. Los datos antiguos pueden estar en un producto desarrollado por la misma casa de software, y puede ser muy sencillo transferirlos al nuevo producto. Sin embargo, si están en un producto desarrollado por otro proveedor, posiblemente un competidor, puede ser más difícil transferir los datos. Las responsabilidades pueden incluir:

- La sustitución del sistema antiguo por el nuevo, ya sea con una asunción completa de las tareas existentes por parte del nuevo sistema o a veces con la operación paralela de ambos sistemas hasta que el usuario quede satisfecho de que el nuevo sistema está funcionando correctamente.
- La transferencia de datos del sistema antiguo al nuevo, en ocasiones con un cambio de formato en los datos.
- Además de proporcionar instrucciones para estas transferencias, la documentación puede contener pruebas para permitir a los usuarios verificar que la instalación ha sido hecha correctamente.
- El equipo puede generar información explicativa adicional para el servicio de asistencia telefónica, especialmente información necesaria para ayudar a los usuarios en caso de que la instalación salga mal.

16.4.5. Finalización de los artefactos

La fase de transición no se cierra hasta que el proyecto haya completado todos los artefactos, incluyendo los modelos y la descripción de la arquitectura. En particular, hacemos hincapié en que el conjunto de modelos debería estar completo al final de la construcción, no rellenado de repente hacia el final de la fase de transición. Más bien, este conjunto de modelos evoluciona a lo largo de todas las iteraciones y todas las fases, como describimos en la Sección 12.8.4, y se corrige, si es necesario, en la fase de transición. Al final de la fase de transición, habremos verificado, a través del uso real, que todos los artefactos son consistentes unos con otros.

16.4.6. ¿Cuándo acaba el proyecto?

La fase de transición no acaba cuando se completan todas las tareas y artefactos, sino cuando el cliente queda "satisfecho". ¿Cuándo están los usuarios satisfechos? La determinación de este momento depende de la relación mercantil.

En el caso de productos de cara al mercado, el jefe de proyecto concluirá que la amplia masa de clientes quedará satisfecha, una vez que el proyecto haya reaccionado ante los resultados de las pruebas beta. En este punto, se lanzará al mercado una versión comercial. En la mayoría de los casos, por supuesto, tanto el entorno como el producto continuarán evolucionando, pero la organización software responderá a esta evolución en versiones posteriores, o en caso de una modificación grande, con un nuevo ciclo de desarrollo. Debido a que, tanto los productos de éxito como los sistemas a medida se amplían a través de revisiones menores y luego mayores, en sentido estricto, el "proyecto" nunca acaba. La fase de transición terminará cuando el proyecto transfiera la responsabilidad del mantenimiento continuado a una organización de apoyo.

En el caso de productos contratados por un cliente, el jefe de proyecto concluirá que el cliente está satisfecho una vez que el sistema pase las pruebas de aceptación. Por supuesto, este punto depende de la interpretación de los requisitos detallados en el contrato original (firmado al final de la fase de elaboración), y modificados a través de adiciones durante las fases posteriores. El cliente puede acordar determinadas modalidades de contrato de mantenimiento al proveedor software. De forma alternativa, el cliente puede llevar a cabo su propio mantenimiento o contratarlo a un tercero. Los detalles pueden diferir, pero el caso es que la fase de transición, como tal, habrá acabado.

16.5. Finalización del análisis de negocio

Los costes y agenda de la fase de transición fueron cubiertos en la apuesta de negocio al final de la fase de elaboración. Al final de la fase de transición, tendremos disponible más información sobre la que juzgar la bondad del análisis de negocio.

16.5.1. Control del progreso

El jefe de proyecto comparará el progreso real en la fase, frente a la agenda, esfuerzo y coste planificado para la fase.

Al final de la fase de transición, que es también el final del proyecto en términos presupuestarios, el jefe de proyecto convocará un grupo para revisar el tiempo, personas-hora, coste, porcentajes de defectos y otras métricas que la empresa pueda utilizar en relación con las cifras planificadas para el proyecto en su conjunto, con el objeto de:

- Ver si el proyecto ha alcanzado los objetivos planeados.
- Determinar las razones por las que no lo ha hecho (si éste es el caso).
- Añadir las métricas del proyecto a la base de datos de métricas de la empresa para su uso futuro.

16.5.2. Revisión del plan de negocio

Este plan prevé si el proyecto tendrá éxito económicamente hablando. De nuevo, existen dos patrones: producción bajo contrato y producción de cara al mercado. En el primer caso, el éxito es fácil de definir: ¿Ha cubierto el precio contratado los costes del proyecto? Por supuesto, este problema relativamente sencillo puede complicarse con consideraciones relacionadas, como si la empresa ha abierto con éxito una nueva área de negocio, o si alguno de los componentes o subsistemas fabricados bajo este contrato pueden llevar a una reducción de costes en contratos futuros.

En el segundo caso, el plan de negocio es más complicado. El éxito se mide de acuerdo a si el producto alcanzará objetivos tales como el margen de beneficios obtenido sobre el capital invertido en el desarrollo. Al final de la fase de transición y del proyecto, no se dispone aún de todos los datos, por ejemplo, el nivel de ventas futuras, pero en este punto, la empresa sabe lo que ha gastado y tiene un mejor conocimiento de las previsiones futuras que la que tenía cuando comenzó el proyecto. El ejecutivo responsable reunirá todos los datos disponibles y convocará un grupo para revisar el plan.

16.6. Evaluación de la fase de transición

El jefe de proyecto reunirá un pequeño grupo para evaluar la fase de transición (*véase* también la Sección 12.8) y para realizar una "autopsia" del ciclo de desarrollo en su conjunto. Esta evaluación difiere de las realizadas al final de las fases anteriores por dos motivos: primero, se trata de la última fase; no hay una fase siguiente a la que transferir el trabajo no concluido aún. No obstante, en un sistema de cierto tamaño, habrá un grupo de producto al cual se puedan remitir ideas útiles. Segundo, aunque ésta sea la última fase del ciclo de desarrollo actual, previsible-

mente existirán futuros ciclos de desarrollo. El grupo de evaluación debería registrar los hallazgos que sean de utilidad para ellos.

Estos hallazgos entrarán en dos categorías, como se describe en las subsecciones siguientes.

16.6.1. Evaluación de las iteraciones y de la fase

Por un lado, si la organización del proyecto ha llevado a cabo las tres primeras fases de forma efectiva, la fase de transición debería desarrollarse sin sobresaltos y completarse de acuerdo con la agenda y presupuesto asignados. Las pruebas beta detectarán sólo fallos rutinarios que el equipo corregirá rápidamente. El grupo de evaluación encontrará pocas cosas dignas de mención que sean útiles para el grupo de producto o para el proyecto que aborde el siguiente ciclo de vida.

Por otro lado, si la organización del proyecto fracasó al identificar todos los riesgos importantes, si fracasó al desarrollar una arquitectura que cumpla con los requisitos, o si fracasó al implementar un diseño que proporcione el sistema requerido, este tipo de deficiencias se mostrarán dolorosamente evidentes en la fase de transición. Como resultado de estas deficiencias, el jefe de proyecto puede ampliar la fase de transición justo hasta alcanzar un sistema mínimamente satisfactorio. Podrá tener que poner un producto "lo bastante bueno" en manos de la comunidad de usuarios. Podrá tener que postergar características originalmente especificadas en los requisitos para una versión posterior.

Por supuesto, estas deficiencias serán materia de trabajo del grupo de evaluación final, quienes, en efecto, evaluarán el proyecto completo —las cuatro fases— y se encargarán de registrar cualquier experiencia insatisfactoria con el proyecto. Desde el punto de vista del proceso, el objetivo de la evaluación es permitir que el proyecto que trabaje en la nueva versión lo haga mejor. En particular, la evaluación, como tal, no debería ser base de represalias sobre el personal. Estas acciones deberían basarse en otras evidencias. Si el grupo de evaluación considera que los hallazgos pueden ser utilizados de manera incorrecta, pueden decidir morderse la lengua. El completo establecimiento de los hechos es importante para el éxito futuro de la organización.

16.6.2. Autopsia del proyecto

A diferencia de la evaluación, la autopsia se dedica fundamentalmente a analizar lo que la organización del proyecto ha hecho bien y lo que ha hecho mal. El objetivo es proporcionar un registro que permitirá organizar futuros proyectos de forma más efectiva y llevar a cabo el proceso de desarrollo con mayor éxito, por ejemplo:

- Los aspectos pertinentes al sistema que ha sido desarrollado deberían ser registrados para su utilización por parte de los encargados del mantenimiento y por el equipo de la siguiente versión. Por ejemplo, mientras que se suelen registrar las razones para elegir el diseño que se ha utilizado, las razones para rechazar otros diseños posibles pueden ser también de utilidad para equipos posteriores.
- Los aspectos pertinentes al proceso de desarrollo en sí deberían ser considerados detalladamente. Por ejemplo:
  - ¿Se necesita más formación general?

— ¿Qué áreas requieren formación especializada?

— ¿Deberían proseguir las actividades de asesoramiento?

— ¿La experiencia de este proyecto con aspectos especializados del Proceso Unificado (*véase* el Capítulo 2) ha permitido desarrollar intuiciones que beneficiarán a futuros proyectos?

16.7. Planificación de la próxima versión o generación

La experiencia de décadas de la industria del software demuestra que pocos productos software resisten sin cambios durante largo tiempo. El sistema operativo y el hardware sobre el que se ejecutan sufren posteriores desarrollos. El entorno de negocios o gubernamental cambia. El software se aplica a sectores cada vez mayores de los ámbitos comercial, industrial, gubernamental y personal. Casi todos los sistemas software entran en un nuevo ciclo de desarrollo de forma casi inmediata. El nuevo ciclo repite las fases de inicio, elaboración, construcción y transición.

16.8. Productos clave

Los productos de esta fase son muy similares a los indicados para la fase de construcción (*véase* el Capítulo 15), pero han sido corregidos y ahora están completos:

- El propio software ejecutable, incluyendo el software de instalación.
- Documentos legales, como contratos, licencias, renuncias de derechos y garantías.
- La versión completa y corregida de línea base de la versión del producto, incluyendo todos los modelos del sistema.
- La descripción completa y actualizada de la arquitectura.
- Manuales y material de formación del usuario final, del operador y del administrador del sistema.
- Referencias (incluyendo referencias de la *Web*) para la ayuda del cliente, acerca de dónde encontrar más información, cómo informar de defectos o dónde encontrar información sobre defectos y actualizaciones.

Capítulo 17

Cómo hacer que el Proceso Unificado funcione

El desarrollo de software para sistemas críticos sigue siendo excesivamente complejo, como han dejado claro los capítulos precedentes de este libro. En este capítulo final, una vez más y como repaso, reunimos todos los elementos, con objeto de atar cabos. Nuestra intención es mostrar, de forma más breve que en los capítulos anteriores, cómo se relacionan y cooperan estos elementos unos con otros.

Además, ampliaremos este capítulo a dos nuevas áreas. Primero, si el proceso software es complejo por sí mismo, de esto se deriva que la gestión de este proceso también es compleja. Segundo, la transición entre no tener proceso o seguir un proceso *ad hoc*, y el Proceso Unificado no va a ser un asunto sencillo.

17.1. El Proceso Unificado ayuda a manejar la complejidad

En primer lugar, tenemos las cuatro fases: inicio, elaboración, construcción y transición. Más allá de las fases del ciclo inicial, y debido a que los grandes sistemas software están en continuo desarrollo, llegan versiones posteriores y, tras revisiones generales, nuevas generaciones. Dentro de estas fases se entremezclan los flujos de trabajo, la arquitectura, la gestión de riesgos, las iteraciones e incrementos, como:

- El desarrollo de software guiado por los casos de uso a través de los flujos de trabajo: requisitos, análisis, diseño, implementación y pruebas.

- El desarrollo guiado por la arquitectura —el esqueleto de los elementos estructurales y de comportamiento que permite que el sistema evolucione sin sobresaltos.
- El desarrollo a partir del uso de bloques de construcción y componentes, facilitando la reutilización.
- El desarrollo llevado a cabo a través de iteraciones y construcciones dentro de las iteraciones, lo que resulta de un crecimiento incremental del producto.
- El desarrollo con riesgos controlados.
- El desarrollo visualizado y registrado usando el Lenguaje Unificado de Modelado (UML).
- El desarrollo evaluado en los hitos.

Cuatro hitos controlan el proceso: los objetivos del ciclo de vida, la arquitectura del ciclo de vida, la capacidad operativa inicial y el lanzamiento del producto [1].

17.1.1. Los objetivos del ciclo de vida

El primer hito clarifica los objetivos del ciclo de vida del producto al plantear cuestiones como:

- ¿Se ha determinado con claridad el ámbito del sistema? ¿Se ha determinado lo que va a estar dentro del sistema y lo que va a estar fuera de él?
- ¿Se ha llegado a un acuerdo con todas las personas involucradas sobre los requisitos fundamentales del sistema?
- ¿Se vislumbra una arquitectura que implemente estas características?
- ¿Se han identificado los riesgos críticos para el desarrollo exitoso del proyecto? ¿Se prevé una forma de mitigarlos?
- ¿El uso del producto generará beneficios que justifiquen lo invertido en su construcción?
- ¿Es factible para su organización llevar adelante el proyecto?
- ¿Están los inversores de acuerdo con los objetivos?

Contestar a estas preguntas es asunto de la *fase de inicio*.

17.1.2. La arquitectura del ciclo de vida

El segundo hito clarifica la arquitectura para el ciclo de vida del producto a través de cuestiones como:

- ¿Se ha creado una línea base de la arquitectura? ¿Es adaptable y robusta? ¿Puede evolucionar a lo largo de la vida del producto?
- ¿Se han identificado y mitigado los riesgos graves hasta el punto de asegurar que no trastornarán el plan de proyecto?
- ¿Se ha desarrollado un plan de proyecto hasta el nivel necesario para respaldar una agenda, costes y calidad realistas y que cubran la apuesta?

- ¿Proporcionará el proyecto, tal y como está planificado y presupuestado en este momento, una recuperación adecuada de la inversión?
- ¿Se ha obtenido la aprobación de los inversores?

Contestar a estas preguntas es asunto de la *fase de elaboración.*

17.1.3. Capacidad operativa inicial

El tercer hito establece que el proyecto ha alcanzado la capacidad operativa inicial: ¿El producto ha alcanzado un nivel de capacidad adecuada para su operación inicial en el entorno del usuario, en particular para realizar las pruebas beta?

Ésta es la pregunta clave del tercer hito. Al comenzar a acercarnos a este hito, tenemos una línea base de la arquitectura, hemos investigado los riesgos, tenemos un plan de proyecto y tenemos recursos. De modo que construimos el producto. Si nos hemos decidido por los objetivos y la arquitectura del ciclo de vida, la construcción se desarrolla sin sobresaltos. Una labor importante en esta fase es el secuenciamiento de construcciones e iteraciones. Una buena secuencia indica que los prerrequisitos de cada iteración son los correctos. Una buena secuencia evita tener que dar marcha atrás y rehacer una iteración previa debido a algo aprendido más tarde.

No obstante, a pesar de nuestros esfuerzos para evitar problemas, algunos seguirán apareciendo. Lo que hay que hacer aquí es superar los problemas del día a día. La construcción del sistema es el objetivo de la *fase de construcción.*

17.1.4. Lanzamiento del producto

El cuarto hito determina que el producto está listo para su lanzamiento sin restricciones a la comunidad de usuarios: ¿Es el producto capaz de operar con éxito en entornos de usuario típicos?

Aún hay trabajo que hacer una vez alcanzada la capacidad operativa inicial; en concreto, las pruebas beta, las pruebas de aceptación, y la corrección de los problemas y defectos que surjan de la operación en el entorno de trabajo. Cuando tengamos algo con lo que los usuarios se sientan satisfechos, distribuiremos el producto.

Estas tareas son asunto de la *fase de transición.*

17.2. Los temas importantes

Estos temas son el recorrido a través de los cuatro hitos principales y la vinculación de unos con otros, pero también los requisitos, la arquitectura, el desarrollo basado en componentes, el Lenguaje Unificado de Modelado, las iteraciones o la gestión de riesgos. (El orden en que mencionamos los temas no refleja su importancia. Todos son importantes. Todos tienen que combinarse unos con otros.)

Obtenga correctamente los requisitos Obténgalos correctamente a través del modelado de casos de uso, el análisis, etc. El mejor comienzo del Proceso Unificado está guiado por los casos de uso.

Obtenga correctamente la arquitectura Cualquier proyecto de tamaño considerable tiene que estar centrado en la arquitectura. La arquitectura permite la partición del sistema y el que estas particiones colaboren entre sí. La arquitectura solidifica las interfaces entre las particiones, permitiendo que haya equipos trabajando de forma independiente a cada lado de la interfaz, y *manteniéndola correcta*. La vigilancia de la arquitectura controla el proyecto desde un punto de vista técnico.

Use componentes Las firmes interfaces de la arquitectura (así como las interfaces estándar de grupos estándar), son uno de los elementos que hacen posible el desarrollo basado en componentes. Los bloques de construcción reutilizables reducen los costes de desarrollo, ponen los productos en el mercado de forma más rápida y mejoran la calidad.

Piense y comuníquese en UML El desarrollo de software es algo más que escribir código. UML (junto con otras características del Proceso Unificado) convierte el desarrollo de software en una disciplina de ingeniería. Es un lenguaje gráfico con el que la gente del software puede pensar, visualizar, analizar, comunicar y registrar.

Sin un medio de comunicación estándar como UML, habría gran dificultad en comprender lo que otras personas y equipos están haciendo. Habría dificultades para transmitir la información a través de las fases y a las versiones y generaciones posteriores.

Itere Las iteraciones y construcciones proporcionan ventajas: tareas pequeñas, grupos de trabajo pequeños, una ligazón con la gestión de riesgos, controles frecuentes, y frecuentes realimentaciones.

Gestione los riesgos Identifique los riesgos —críticos, significativos, rutinarios—, póngalos en una lista de riesgos, y mitíguelos antes de que se manifiesten en el proceso de desarrollo.

17.3. La dirección lidera la conversión al Proceso Unificado

Más allá de la tarea de gestionar y controlar un proceso en curso, está la de llevar una empresa, pública o privada, de los viejos métodos a la nueva senda. El ámbito empresarial y gubernamental tiene, en la actualidad, muchas organizaciones que han avanzado poco en la senda del proceso software. Por supuesto que desarrollan software, ya que lo hacen de alguna forma, pero no lo están haciendo de forma organizada y repetible. Así mismo, hay muchas otras organizaciones que disponen de alguna clase de proceso software, pero que están, quizá de una forma vaga, insatisfechas con él. También éstas desarrollan software, a veces con un éxito considerable, pero sienten que debe haber algún camino mejor. Por supuesto que lo hay. ¿Cómo poner en marcha estas organizaciones por el buen camino?

Un solo profesional que ocupe un puesto bajo en el escalafón no está en condiciones de llevar a cabo esta transición. Puede ser el que actúe de líder de la nueva senda, una vez que la organización haya decidido adoptarla. No, ésta es claramente una responsabilidad del ejecutivo que esté en la cima de la empresa de desarrollo de software. Es él quien debe liderar, porque esto afecta a toda su organización; el que tiene que entender la idea de que existe un camino mejor y que es bueno transitarlo; el que tiene que sentir que la competencia convierte en imperativo hacer algo pronto. Quizás reciba el germen de esta idea de un colega cuya organización esté ya en marcha en el área del proceso software.

17.3.1. La necesidad de actuar

Debido a que el nuevo camino afectará a otras partes de la organización de desarrollo de software y a toda la empresa, el primer ejecutivo debe buscar el apoyo de otros ejecutivos. Puede hacer esto a través de una declaración de necesidad de actuar. Básicamente, esta declaración establece que la forma en la que se están haciendo las cosas ya no es válida. Que cuesta mucho. Que la calidad es baja y, lo que probablemente es más importante, que el tiempo de entrega es demasiado largo. Incluso si estamos utilizando nuestro software internamente, no estamos logrando los beneficios competitivos de la mejora del software a un ritmo adecuado. Ganancias año a año de un cinco por ciento ya no son lo bastante buenas. Ahora son posibles grandes ganancias en estas tres áreas —costes, agenda y calidad—, porque otras empresas lo han demostrado.

La declaración de la necesidad de actuar lleva a lo que debe ser esta acción. Por lo general, las organizaciones de desarrollo de software no están en el negocio del desarrollo de métodos, bloques de construcción ni componentes. Hay otras organizaciones haciendo esto. Por ejemplo, existen organizaciones que producen y venden componentes. Hay organizaciones, como SAP, que producen sistemas prefabricados que pueden ser adaptados a su negocio. (Un sistema prefabricado es un gran marco de trabajo en el cual el cliente, con la ayuda del proveedor, realiza pequeñas adaptaciones para satisfacer las necesidades específicas de su negocio.) Hay organizaciones, como Rational Software Corporation que respaldan el Proceso Unificado. La labor del ejecutivo del software es descubrir dónde puede conseguir la ayuda que su organización necesita.

Dado que las opciones no son muchas, para la mayoría de las organizaciones sólo hay un camino: el desarrollo de software basado en componentes. Decidir si tratamos de hacer mucho por nuestra propia cuenta, como en la banca, seguros, defensa, telecomunicaciones, etc., o si tratamos de encontrar un sistema prefabricado, de construir sobre bloques de construcción reutilizables, es un problema que debe resolver el ejecutivo del software en función de su propia situación. En la mayoría de los casos habrá aún algún —o mucho— desarrollo de software que realizar internamente. Alguien tendrá que adquirir los bloques de construcción; alguien tendrá que ajustarlos a las necesidades específicas de la empresa. Esto requiere un proceso software. En muchos casos, los ejecutivos del software descubrirán que el Proceso Unificado es la respuesta a esa parte de sus necesidades.

17.3.2. La directriz de reingeniería

El ejecutivo inicial explicará cuidadosa y extensamente, y sin exagerar en la directriz de reingeniería, por qué la empresa está cambiando a un proceso software mejorado. La directriz cubre:

- La situación actual del negocio y el hecho de que está cambiando.
- Lo que esperan los clientes en la actualidad.
- La competición a la que se enfrenta la empresa.
- Los retos que afronta la empresa.
- El diagnóstico al que llevan estos retos.
- Los riesgos que traerá el no realizar cambios.
- Lo que la empresa debe hacer sobre el proceso software en particular [2]. [1]

[1] En [2] se describe el contenido de la directriz de reingeniería, tal y como se aplica a la reutilización. La psicología subyacente a esta directriz es igualmente aplicable al proceso software.

Garantía de apoyo Los jefes de proyecto tienen que tener confianza en obtener apoyo financiero continuado. Entre otras cosas, esta garantía cubre la formación inicial, el asesoramiento mientras el jefe de proyecto vigila las aplicaciones iniciales del nuevo proceso, y la formación continuada a medida que cambian las necesidades. No intente realizar la transición al nuevo proceso sin tener a alguien que lo haya hecho antes. Los asesores saben cómo hacerlo. Pueden ser externos o internos. El comenzar un nuevo proyecto con un nuevo proceso depende de la plena integración de cuatro apoyos: proceso, herramientas, formación y asesoría.

Continuidad de los proyectos existentes En una empresa de cierto tamaño, con muchos proyectos en curso, los proyectos actuales y la mayoría de los que aparezcan en un futuro inmediato tendrán que continuar con el proceso actual. No se puede formar a todo el mundo a la vez. No todo proyecto puede permitirse el posible trastorno de realizar cambios considerables.

Al mismo tiempo, la directriz de reingeniería debería dejar claro que el personal y los jefes de proyecto que continúen con los proyectos existentes no van a ser dejados de lado. Cuando les toque el turno, serán formados y asignados a proyectos que se realicen bajo el Proceso Unificado. La transición lleva tiempo. Necesariamente, la empresa debe continuar con el negocio existente como parte del precio para estar ahí en el futuro. Se formará a estas otras personas de manera que puedan participar en las discusiones sobre lo que va a venir. No deben sentirse como proscritos. Si lo hacen, pueden dañar seriamente el esfuerzo de transición.

Confianza en el propio futuro Las personas implicadas en esta transición son profesionales del software. Este campo ha cambiado de forma drástica a lo largo de su vida, corta según los estándares históricos. Algunos han tenido dificultades para mantenerse al día. Los mandos intermedios, preocupados como están por tareas administrativas, sienten especialmente esta carga. Si se sienten razonablemente confiados en su propio futuro, esto ayudará a la transición. Las empresas que tradicionalmente han cuidado de sí mismas tienen ventaja en esto, pero todas tienen que ser conscientes de su importancia.

17.3.3. Implementación de la transición

El ejecutivo del software se enfrenta con los problemas de implementación.

El líder El primer problema es tener una organización a través de la cual implementar la transición. Ya que el propio ejecutivo del software tendrá probablemente todo su tiempo ocupado, necesita un ingeniero técnicamente cualificado para liderar el cambio día a día, es decir, para supervisar la transición. Este líder debe comprender el Proceso Unificado y, para lograr tal comprensión, debe realizar alguna formación y conseguir asesoría personalizada.

Este líder será, probablemente, un jefe de proyecto que sea técnicamente competente. Podría ser un arquitecto con capacidades de dirección de proyectos. Habrá estudiado el Proceso Unificado, estará convencido de que puede ayudar a la empresa y estará deseoso de defender su punto de vista. Sin embargo, y al mismo tiempo, tendrá la confianza tanto de los ejecutivos que le subvencionan, como de las personas que participan en el proyecto. Sabrá cómo trabajar tanto con los gestores como con el personal técnico. Será una persona que sea buena convenciendo a otros. Normalmente, estará interesado en métodos y proceso, pero al mismo tiempo será lo suficientemente maduro como para no comenzar desde cero con su propio y exclusivo método. Al contrario, adaptará el Proceso Unificado a las necesidades particulares del primer proyecto.

No es necesario que el líder haya estado antes en un proyecto que siguiese el Proceso Unificado. De hecho, no es probable que haya tenido esta oportunidad, ya que habrá estado traba-

jando para la organización desde hace algún tiempo. Sólo es necesario que comprenda lo que significa hacerlo. Basta con que esté deseoso de ponerse a hacerlo —con ayuda, por supuesto, del jefe de proyecto y el asesor, de los que hablamos a continuación.

El jefe del primer proyecto Además de este líder (por cierto, si no es capaz de encontrar este hombre o mujer excepcional que hemos descrito, elija al mejor de los que tenga y déle toda la ayuda que necesite), el ejecutivo del software responsable también necesitará un jefe de proyecto excepcionalmente capaz. Necesitará uno que sienta hasta la médula que es importante tanto adoptar el nuevo proceso como llevarlo a cabo. El proyecto aún tiene que desarrollarse con éxito ante las caras de perplejidad que pueden estar presenciando la adopción del nuevo proceso.

El asesor La formación no lo es todo. En particular, no proporciona experiencia directa en recorrer la nueva senda. Por tanto, el líder y el jefe de proyecto necesitarán estar respaldados por un asesor (interno o externo) que haya estado en proyectos que siguiesen el Proceso Unificado. El asesor no necesita tener experiencia de gestión, aunque no es malo que la tenga. Necesitará dos talentos especiales. Uno es la habilidad para anticipar problemas en el proyecto. Esta habilidad, por supuesto, estará basada en haber estado anteriormente en un proyecto que adoptase el Proceso Unificado. Segundo, tiene que ser capaz de colaborar con la variada gente con la que se encontrará, en particular el líder, pero también el jefe de proyecto, el personal del proyecto y el ejecutivo que lo financia.

Por dónde empezar El primer proyecto debería ser uno real cuyos resultados fuesen importantes. Nuestra experiencia con proyectos piloto artificiales no ha sido alentadora. Tienden a estancarse. Sus resultados, cualquiera que sean, no son relevantes. El primer proyecto debiera ser crítico, pero su agenda no debería ser demasiado crítica. Ajá, dirá alguien, todos los proyectos importantes tienen agendas apretadas.

Lo sabemos. En realidad, el trabajo bajo el Proceso Unificado discurre de forma más rápida que bajo métodos más antiguos. Se buscan antes los riesgos, se desarrolla antes una arquitectura, de modo que, en realidad, la construcción discurre más rápidamente. Además, el primer proyecto estará guiado por asesores. Son asesores porque habrán participado anteriormente en el proceso. Saben hacerlo funcionar.

En la práctica, aplicar la propuesta completa en un proyecto real es seguro, aunque esto no significa hacer demasiadas cosas al mismo tiempo. El proceso, sí. Las herramientas que van con el proceso, sí —van juntos, si las herramientas están bien integradas—. Pero, un nuevo sistema operativo, una nueva tecnología de base de datos, un nuevo lenguaje de programación, una nueva plataforma distribuida, probablemente no —no parta de demasiadas cosas nuevas al mismo tiempo, en particular de una tecnología que no esté estrechamente integrada con otra tecnología—. Dependiendo de las circunstancias, podría ser capaz de usar una tecnología nueva más. No en un sistema tan grande como el primero.

Consideraciones adicionales Nuestra experiencia nos dicta unas pocas cosas más:

- El enfoque que presentamos aquí no es siempre la forma en que se lleva a cabo, pero es la forma sistemática de adoptar el Proceso Unificado.
- Un enfoque más gradual, más paso a paso puede caer en el error contrario. Debido a que el progreso es casi invisible, el apoyo desaparece gradualmente y la transición fracasa.
- A veces, es cierto, se puede tener éxito al reestructurar el proceso paso a paso, pero no se puede contar con ello, lleva mucho tiempo, y a menudo falla.

- En cualquier caso, es mejor tener la transición bajo control efectivo de los gestores y llevarla a cabo. Los errores resultantes de una falta de control son difíciles de arreglar.

17.4. Especialización del Proceso Unificado

El Proceso Unificado, tal y como se presenta en este libro, no es lo único que se puede decir a este respecto. De hecho, hay otras dos cosas importantes que decir sobre él. Primero, es un marco de trabajo. Tiene que ser adaptado en una serie de variables: el tamaño del sistema en curso, el dominio en el que ese sistema ha de trabajar, la complejidad del sistema y la experiencia, pericia y nivel de proceso de la organización del proyecto y sus miembros. Segundo, este libro, aunque parezca detallado, da sólo una visión general del proceso de trabajo. En realidad, para aplicarlo, se necesita una considerable información adicional. Trataremos de estas dos especializaciones del Proceso Unificado en las próximas dos subsecciones.

17.4.1. Adaptación del proceso

Los sistemas y productos software y las organizaciones que los construyen siguen siendo tan diversos como siempre. Consiguientemente, aunque hay ciertas constantes en el Proceso Unificado, como las cuatro fases y los cinco flujos de trabajo, existen también numerosos factores variables. Por ejemplo, ¿cuál debería ser la longitud relativa de las fases? ¿Cuántas iteraciones son las adecuadas para cada fase bajo diferentes circunstancias? ¿En qué momento está la arquitectura candidata (o la mitigación de riesgos críticos, la línea base de la arquitectura, el análisis de negocio, etc.) lo suficientemente definida?

La respuesta a preguntas como éstas depende del tamaño del sistema, de la naturaleza de la aplicación, de la experiencia de la organización del proyecto en el dominio, de la complejidad del sistema, de la experiencia del equipo del proyecto, del grado de capacidad de los gestores, incluso del grado en el que son capaces de colaborar de forma efectiva todos los implicados en el proyecto.

Por poner un ejemplo —si el sistema es relativamente pequeño, y el equipo del proyecto tiene experiencia en el dominio (si ha desarrollado versiones anteriores o productos similares)— la fase de inicio puede ser muy breve. El equipo (y probablemente también el resto de los implicados en el proyecto) sabrán que no hay riesgos críticos en el curso del desarrollo con éxito. Sabrán que puede utilizarse de nuevo una arquitectura empleada con anterioridad. El resultado será que unos pocos días, para confirmar el ámbito del sistema y para asegurarse de que no ha surgido riesgos en este ámbito, pueden ser suficientes para completar la fase de inicio.

Por poner otro ejemplo —si el sistema es grande, complejo y novedoso para el equipo del proyecto, podemos suponer que la fase de inicio, así como la fase de elaboración, serán mucho más largas y discurrirán a través de más iteraciones.

El número de variaciones sobre estos temas es tan grande como el número de sistemas a construir. Aquí es donde la experiencia desempeña su papel. Aquí es donde el jefe de proyecto debe tener clara la conveniencia de incorporar la experiencia de otras gentes del software, así como el conocimiento de los usuarios. Se puede adaptar el proceso para ajustarlo a sus circunstancias [3].

17.4.2. Completando el marco de trabajo del proceso

El Proceso Unificado presentado en este libro es sólo una visión general del proceso que se necesita para gestionar un equipo de personas trabajando al unísono. El libro pretende proporcionar, a los que lo practican y a los gestores, una comprensión del proceso. No proporciona a los miembros del equipo las indicaciones detalladas que necesitan para guiar su trabajo diario. Sólo enumera algunos de los artefactos que el proceso crea y usa; existen otros. No proporciona plantillas de documentos. No identifica todos los diferentes tipos de trabajadores; existen más. Se refiere de vez en cuando al hecho de que las herramientas pueden ser de gran ayuda para llevar a cabo el proceso. De hecho, gran parte del trabajo rutinario relativo a implementar el proceso puede ser realizado por herramientas, y realizado de forma más rápida y precisa que por personas que no dispongan de estas ayudas. El libro, sin embargo, no detalla lo que son estas herramientas, cuáles se usan para qué, ni cómo usarlas. En esencia, el libro cubre las ideas básicas. Describe algunos flujos de trabajo, algunos artefactos, algunos trabajadores y algunas de sus actividades, así como los usos del Lenguaje Unificado de Modelado.

Hay mucho más. Para esta versión altamente desarrollada, se necesita el *Rational Unified Process*. Este producto de proceso es una base de conocimiento en la que pueden hacerse búsquedas *on-line* sobre un total aproximado de 1.800 páginas de material. Aumenta la productividad del equipo al proporcionar, a cada miembro del equipo, indicaciones para aumentar la productividad, plantillas y la ayuda de herramientas para actividades críticas del desarrollo. El proceso está integrado con las herramientas de *Rational*. De hecho, las herramientas y el proceso fueron desarrollados de forma conjunta. Los contenidos son actualizados de forma continua. Están respaldados por un exhaustivo conjunto de cursos. Ésta es la base a partir de la que los asesores ofrecen indicaciones personalizadas.

El *Rational Unified Process* es un desarrollo adicional del *Rational Objectory Process*, existente desde hace tiempo. Si se ha estado utilizando éste último, puede adaptarse fácilmente al nuevo *Rational Unified Process*.

17.5. Relación con comunidades más amplias

Además de dotar de un entorno de trabajo más efectivo al proyecto, y de relaciones más efectivas con los usuarios e inversores, el Proceso Unificado proporciona una interfaz más satisfactoria a comunidades aún más amplias:

- *La comunidad educativa* Los educadores y los encargados de la formación pueden centrar sus cursos en lo que los estudiantes necesitan saber para trabajar con el Proceso Unificado y para pensar y visualizar usando el Lenguaje Unificado de Modelado.
- *La comunidad software* Los arquitectos, desarrolladores, etc. pueden cambiar de proyecto, e incluso de empresa, sin un largo periodo de adaptación a prácticas exclusivas de la nueva empresa.
- *La comunidad de reutilización* Los proyectos pueden reutilizar subsistemas y componentes de forma más fácil, puesto que están representados en UML.
- *La comunidad de herramientas* Los proyectos estarán respaldados por herramientas más efectivas, ya que el amplio mercado del Proceso Unificado permite mejor soporte financiero para el desarrollo de herramientas.

17.6. Obtenga los beneficios del Proceso Unificado

Reafirme pronto los requisitos haciendo evolucionar los casos de uso conjuntamente con los usuarios reales y la gente de ingeniería y de proceso; muévase de forma más efectiva de los requisitos a los flujos de trabajo posteriores (análisis, diseño, etc.) —debido a que el Proceso Unificado está **dirigido por los casos de uso** (Apéndice C).

Consolide lo que el proyecto ha de hacer; dirija el proyecto al hacerlo; dote de una guía a las futuras generaciones del producto. Todo esto a través de una arquitectura del ciclo de vida que es comprensible, adaptable, robusta y evolutiva para los años venideros —debido a que el Proceso Unificado está *centrado en la arquitectura.*

Aprenda de la experiencia de construcciones e iteraciones sucesivas, tanto los desarrolladores como el resto de los implicados en el proyecto —debido a que el Proceso Unificado es **iterativo** (Apéndice C) e *incremental.*

Minimice la posibilidad de que riesgos críticos pongan en peligro el éxito del proyecto, o que comprometan su presupuesto o agenda riesgos significativos —debido a que el Proceso Unificado está **guiado por los riesgos.**

Mejore la velocidad del proceso de desarrollo, reduzca sus costes y aumente la calidad del producto a través de la reutilización de bloques de construcción —debido a que el Proceso Unificado está *basado en componentes.*

Logre que los miembros del equipo del proyecto, y el resto de los implicados en él, trabajen de forma conjunta —debido a que el Proceso Unificado es más que una guía para desarrolladores individuales, es un proceso de ingeniería.

El disponer de este marco de trabajo de fases, iteraciones, hitos y evaluaciones dota de una serie de puntos de apoyo a los cuales los usuarios pueden agarrarse durante el desarrollo y descubrir qué es lo que está pasando. Sabiendo lo que ocurre y estando informados por su propio conocimiento de la aplicación, los usuarios pueden realizar sugerencias para la mejora del sistema. Y lo que es más importante, debido a que las fases iniciales tienen en cuenta aspectos clave como son la arquitectura y los riesgos, pueden ofrecer sus ideas cuando el proyecto está aún a tiempo de hacer buen uso de ellas.

Basado en 30 años de trabajo en la práctica, el Proceso Unificado reúne la experiencia de varios líderes de pensamiento y organizaciones experimentadas. Esta unificado a partir de muchas aplicaciones y técnicas, tales como:

- "*Visualizable*" Los modelos visuales y artefactos empleados en el Proceso Unificado se expresan en el Lenguaje Unificado de Modelado, lo que nos conduce a sus muchas ventajas, tales como una gran reutilización del software y a esquemas del software.
- "*Mecanizable*" Un proceso unificado y un lenguaje estándar dotan del soporte financiero para herramientas más completas, lo que, a su vez, hace el proceso más efectivo.
- "*Adaptable*" Es un marco de trabajo de proceso, no un proceso rígido. Es especializable a diferentes campos de aplicación y necesidades organizativas.
- "*Extensible*" El Proceso Unificado no limita a sus usuarios a una única forma de llevar a cabo una actividad, por ejemplo, el análisis de riesgos. Los usuarios pueden dirigirse a otras fuentes para guiarse. Lo que hace el Proceso Unificado es determinar puntos lógicos

en el proceso en los cuales la actividad, en este caso el análisis de riesgos, va a tener lugar, fundamentalmente "temprano", antes que "tarde".

El Proceso Unificado capacita a las organizaciones de muchas maneras. La más importante es que proporciona la forma en la que el equipo de proyecto puede trabajar conjuntamente. Además, proporciona la forma en la que el equipo de proyecto puede trabajar con los usuarios y con el resto de los implicados.

17.7. Referencias

[1] Barry Boehm, "Anchoring the software process", *IEEE Software,* July 1996, pp. 73–82.

[2] Ivar Jacobson, Martin Griss, Patrik Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success,* Reading, MA: Addison-Wesley, 1997.

[3] Walker Royce, *Software Project Management: A Unified Framework,* Reading, MA: Addison-Wesley, 1998.

Apéndice A

Visión general de UML

A.1. Introducción

Las ramas más antiguas de la ingeniería han encontrado útil desde hace mucho tiempo representar los diseños mediante dibujos. Desde los inicios del software, los programadores han encapsulado sus conceptos en diversos tipos de dibujos o, más ampliamente, de modelos. La comunidad del software precisa de una forma de comunicar sus modelos, no sólo entre los miembros de un proyecto, sino a todas las personas involucradas en él, y, con el paso del tiempo, a los desarrolladores de futuras generaciones. Necesita un lenguaje no sólo para comunicarse con otros, sino para proporcionar un marco en el que desarrolladores individuales puedan pensar y analizar. Además, éstos no pueden retener todo esto en sus cabezas durante meses o años. Tienen que registrarlo sobre papel o electrónicamente. El Lenguaje Unificado de Modelado (UML) es un lenguaje estándar de modelado para software —un lenguaje para la visualización, especificación, construcción y documentación de los artefactos de sistemas en los que el software juega un papel importante. Básicamente, UML permite a los desarrolladores visualizar los resultados de su trabajo en esquemas o diagramas estandarizados. Por ejemplo, los símbolos o iconos característicos utilizados para capturar los requisitos son una elipse para representar un caso de uso y un monigote para representar un usuario que utiliza el caso de uso. De forma similar, el icono principal utilizado en diseño es un rectángulo para representar una clase. Estos iconos no son más que una notación gráfica, es decir, una sintaxis. En la Sección A.2 damos una visión general de la notación gráfica de UML. *Véase* [2] para una referencia más detallada y [3] para una guía de usuario. Sin embargo, tras esta notación gráfica, UML especifica un significado, es decir, una semántica. Damos en la Sección A.3 una breve visión general de esta semántica, proporcionando definiciones escuetas de los términos centrales de UML. *Véase* [2]

y [3] para un tratamiento más profundo de la semántica de UML. Otra referencia que trata tanto la notación como la semántica de UML es su documentación pública [1], aunque ésta es de una naturaleza más formal. Otras referencias generales sobre UML son [4] y [5].

A.1.1. Vocabulario

UML proporciona a los desarrolladores un vocabulario que incluye tres categorías: elementos, relaciones y diagramas. Las mencionaremos aquí sólo para darle una idea de la estructura básica del lenguaje.

Hay cuatro tipos de elementos: estructurales, de comportamiento, de agrupación y de anotación. Hay siete tipos principales de *elementos estructurales*: casos de uso, clases, clases activas, interfaces, componentes, colaboraciones y nodos. Hay dos tipos de *elementos de comportamiento*: interacciones y máquinas de estados. Hay cuatro tipos de *agrupaciones*: paquetes, modelos, subsistemas y marcos de trabajo. Y hay un tipo principal de *elementos de anotación*: notas.

Dentro de la segunda categoría, la de relaciones, encontramos de tres tipos: de dependencia, de asociación y de generalización.

Y en la tercera categoría, la de diagramas, UML proporciona nueve tipos: diagramas de caso de uso, de clases, de objetos, de secuencia, de colaboración, de estados, de actividad, de componentes y de despliegue.

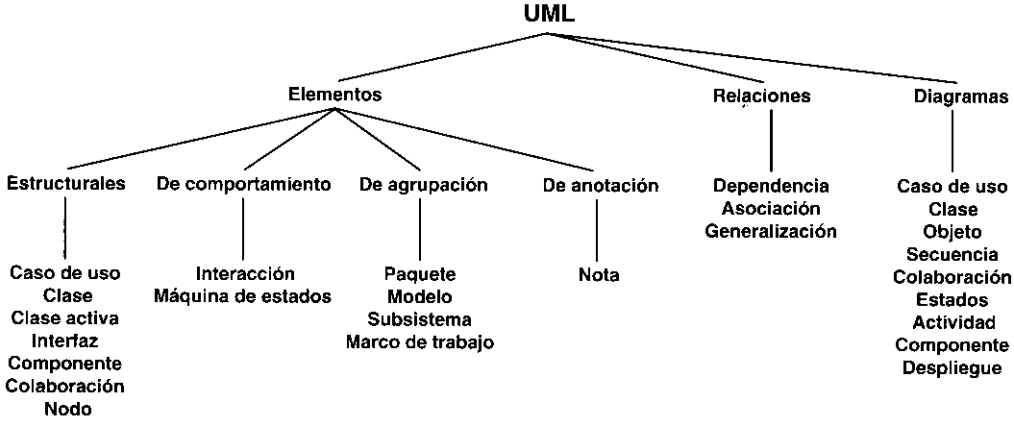


Figura A.1. El vocabulario de UML.

A.1.2. Mecanismos de extensibilidad

UML proporciona mecanismos de extensibilidad, los cuales permiten a sus usuarios refinar su sintaxis y su semántica. UML puede, por tanto, ajustarse a un sistema, proyecto o proceso de desarrollo específico si es necesario. *Véase* el Apéndice B para un ejemplo de dicho ajuste.

Los mecanismos de extensibilidad incluyen estereotipos, valores etiquetados y restricciones. Los estereotipos proporcionan una forma de definir nuevos elementos extendiendo y refinando la semántica de elementos como elementos y relaciones (*véase* Sección A.1.1) ya existentes. Los

valores etiquetados proporcionan una forma de definir nuevas propiedades de elementos ya existentes. Finalmente, las restricciones proporcionan una forma de imponer reglas (como reglas de consistencia o reglas de negocio) sobre elementos y sus propiedades.

A.2. Notación gráfica

El resto de las figuras en este apéndice están tomados de *El Lenguaje Unificado de Modelado* de Grady Booch, James Rumbaugh e Ivar Jacobson [3].

A.2.1. Cosas estructurales

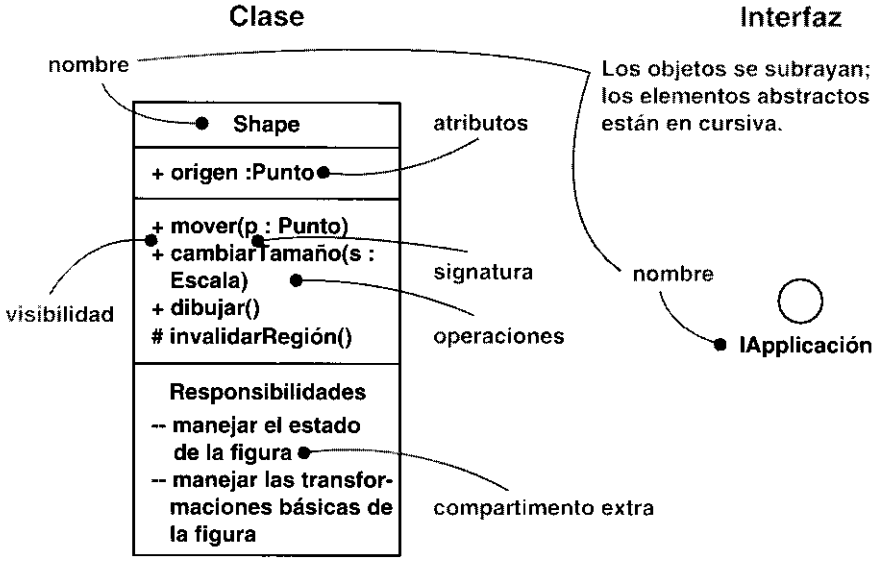


Figura A.2. Clases e interfaces.

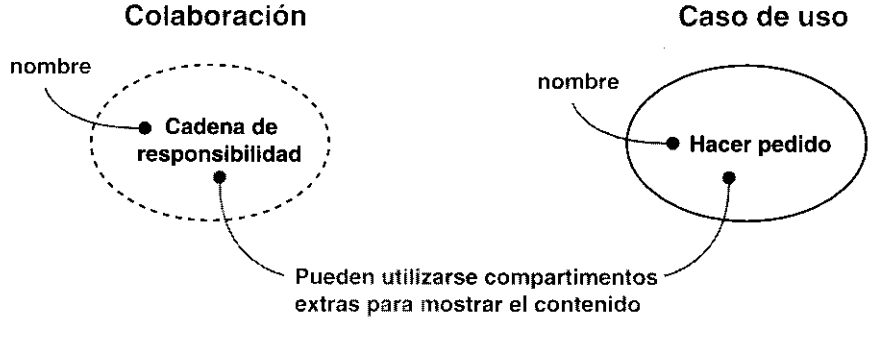


Figura A.3. Casos de uso y colaboraciones. (Obsérvese que los nombres de, por ejemplo, los casos de uso pueden ponerse fuera del símbolo si es conveniente.)

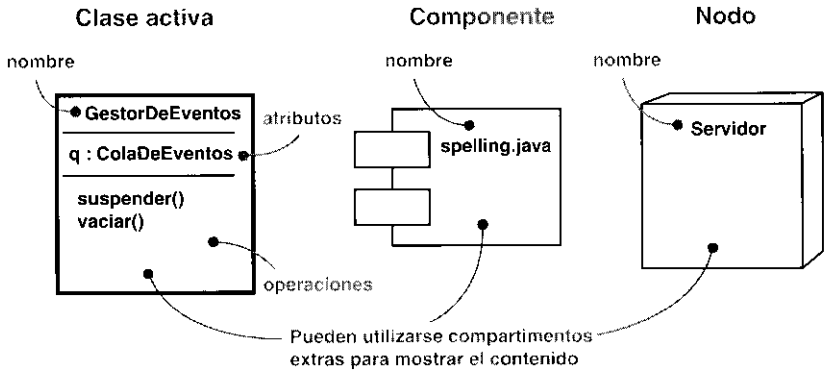


Figura A.4. Clases activas, componentes y nodos.

A.2.2. Elementos de comportamiento

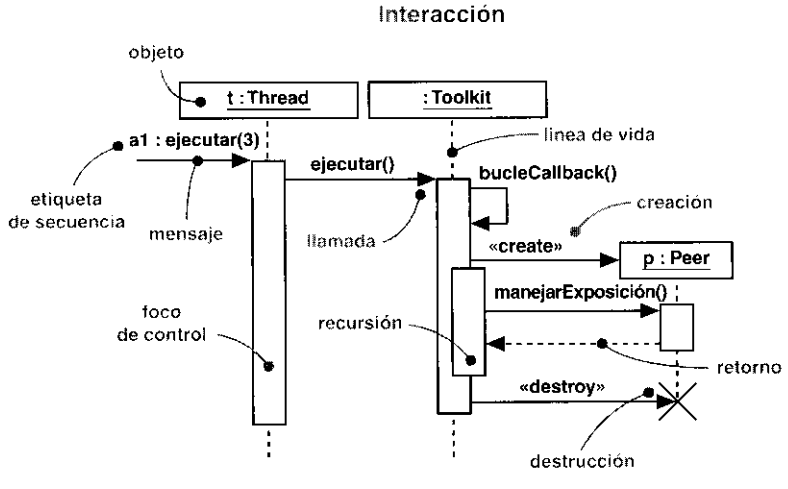


Figura A.5. Interacciones.

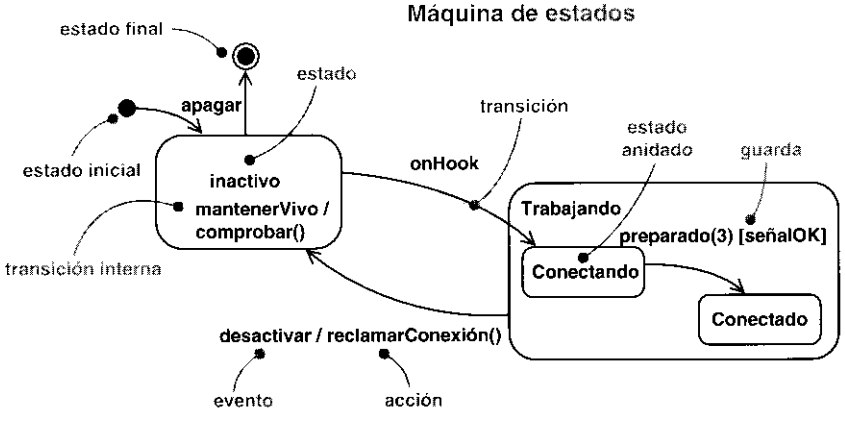


Figura A.6. Máquinas de estados.

A.2.3. Elementos de agrupación

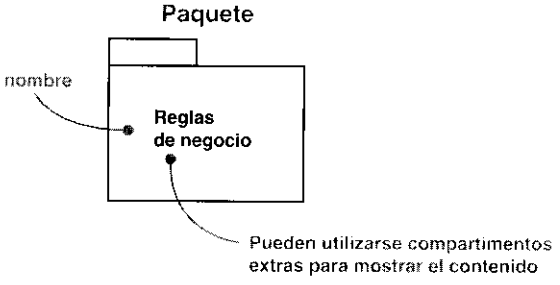


Figura A.7. Paquetes.

A.2.4. Elementos de anotación

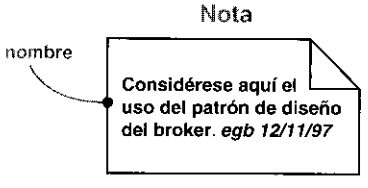


Figura A.8. Notas.

A.2.5. Relaciones de dependencia

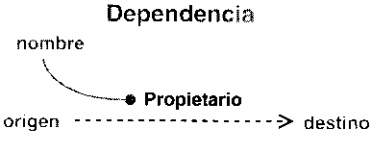


Figura A.9. Relaciones de dependencia.

A.2.6. Relaciones de asociación

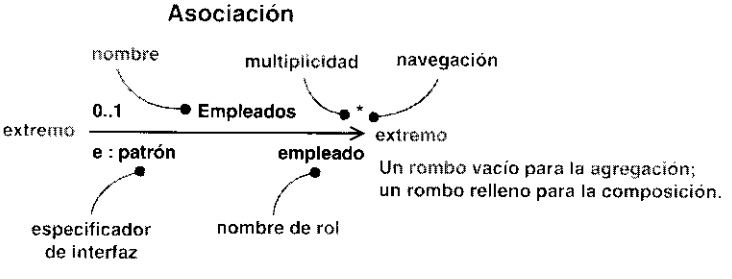


Figura A.10. Relaciones de asociación.

A.2.7. Relaciones de generalización

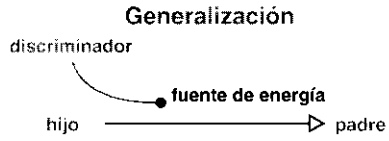


Figura A.11. Relaciones de generalización.

A.2.8. Mecanismos de extensibilidad

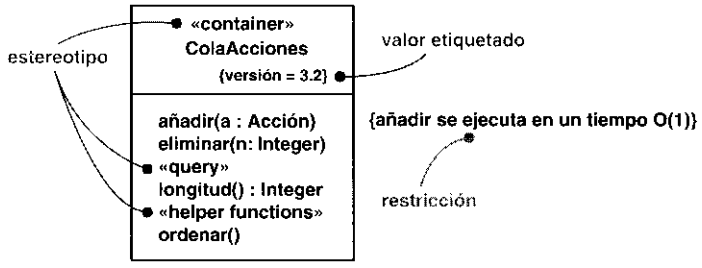


Figura A.12. Mecanismos de extensibilidad.

A.3. Glosario de términos

acción La especificación de una operación ejecutable que constituye la abstracción de un procedimiento computacional. Una acción da lugar a un cambio de estado y se lleva a cabo enviando un mensaje a un objeto o modificando un valor en un atributo.

acción asíncrona Una petición donde el objeto que la envía no se detiene para esperar los resultados.

acción síncrona Una petición donde el objeto emisor se detiene a esperar los resultados.

activación La ejecución de una acción.

actividad El estado en que se exhibe algún comportamiento.

actor Un conjunto coherente de *roles* que los usuarios de casos de uso desempeñan cuando interaccionan con estos casos de uso.

adorno Detalle de la especificación de un elemento añadido a su notación gráfica básica.

agregación Una forma especial de asociación que especifica una relación todo-parte entre el agregado (el todo) y una parte componente (la parte).

agregado Una clase que representa el "todo" en una relación de agregación.

ámbito El contexto que da un significado específico a un nombre.

asociación Una relación estructural que describe un conjunto de enlaces, donde un enlace es una conexión entre objetos; la relación semántica entre dos o más clasificadores que implican las conexiones entre sus instancias.

asociación binaria Una asociación entre dos clases.

asociación n-aria Una asociación entre *n* clases. Cuando *n* es igual a dos la asociación es binaria. Véase *asociación binaria.*

atributo Una propiedad con nombre de un clasificador que describe el rango de valores que las instancias de una propiedad pueden tomar.

calle Una partición de un diagrama de actividad para organizar responsabilidades de acciones.

cardinalidad El número de elementos en un conjunto.

caso de uso Una descripción de un conjunto de secuencias de acciones, incluyendo variaciones, que un sistema lleva a cabo y que conduce a un resultado observable de interés para un actor determinado.

clase Una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica.

clase abstracta Una clase que no puede ser instanciada directamente.

clase activa Una clase cuyas instancias son objetos activos. Véase *proceso, tarea, hilo.*

clase asociación Un elemento de modelado que tiene a la vez propiedades de asociación y de clase. Una clase asociación puede verse como una asociación que tiene además propiedades de clase, o una clase que tiene además propiedades de asociación.

clase concreta Una clase que puede ser instanciada directamente.

clasificación múltiple Una variación semántica de generalización en la cual un objeto puede pertenecer directamente a más de una clase.

clasificador Un mecanismo que describe características estructurales y de comportamiento. Los clasificadores incluyen interfaces, clases, tipos de datos, componentes y nodos.

cliente Un clasificador que solicita servicio de otro clasificador.

colaboración Una sociedad de clases, interfaces y otros elementos que trabajan juntos para proporcionar algún comportamiento cooperativo que es mayor que la suma de todos los elementos; la especificación de cómo un elemento, como un caso de uso o una operación, es llevada a cabo por un conjunto de clasificadores y asociaciones desempeñando roles específicos utilizados de una forma específica.

comentario Una anotación que se adjunta a un elemento o colección de elementos.

componente Una parte física y reemplazable de un sistema que se ajusta a, y proporciona la realización de, un conjunto de interfaces.

composición Una forma de agregación con un fuerte sentido de pertenencia y coincidencia en el tiempo de vida como parte del todo; las partes con multiplicidad variable pueden ser creadas después del compuesto mismo, pero una vez creadas viven con él; dichas partes pueden también ser destruidas explícitamente antes de la muerte del compuesto.

compuesto Una clase que está relacionada con una o más clases mediante una relación de composición.

concurrencia La ocurrencia de dos o más actividades durante el mismo intervalo de tiempo. La concurrencia puede conseguirse mediante el entrelazado o por la ejecución simultánea de dos o más hilos.

contenedor Un objeto que existe para contener otros objetos, y que proporciona operaciones para acceder o iterar sobre su contenido.

contexto Un conjunto de elementos relacionados para un propósito particular, como la especificación de una operación.

delegación La habilidad de un objeto de emitir un mensaje a otro objeto como respuesta a un mensaje.

dependencia Una relación semántica entre dos elementos, en la cual un cambio en un elemento (la cosa independiente) puede afectar la semántica del otro elemento (la cosa dependiente).

diagrama La presentación gráfica de un conjunto de elementos, usualmente representado como un grafo conectado de vértices (elementos) y arcos (relaciones).

diagrama de actividad Un diagrama que muestra el flujo de actividad a actividad; los diagramas de actividad tratan la vista dinámica de un sistema. Un caso especial de diagrama de estados en el cual todos o casi todos los estados son estados de acción y en el cual todas o casi todas las transiciones son disparadas por la terminación de las acciones en los estados origen.

diagrama de caso de uso Un diagrama que muestra un conjunto de casos de uso y de actores y sus relaciones; los diagramas de casos de uso muestran los casos de uso de un sistema desde un punto de vista estático.

diagrama de clases Un diagrama que muestra un conjunto de clases, interfaces y colaboraciones y las relaciones entre éstos; los diagramas de clases muestran el diseño de un sistema desde un punto de vista estático; un diagrama que muestra una colección de elementos (estáticos) declarativos.

diagrama de colaboración Un diagrama de interacción que enfatiza la organización estructural de los objetos que envían y reciben mensajes; un diagrama que muestra las interacciones organizadas alrededor de instancias y de los enlaces entre ellas.

diagrama de componentes Un diagrama que muestra un conjunto de componentes y sus relaciones; los diagramas de componentes muestran los componentes de un sistema desde un punto de vista estático.

diagrama de despliegue Un diagrama que muestra un conjunto de nodos y sus relaciones; un diagrama de despliegue muestra el despliegue de un sistema desde un punto de vista estático.

diagrama de estados Un diagrama que muestra una máquina de estados; los diagramas de estados tratan la vista dinámica de un sistema.

diagrama de interacción Un diagrama que muestra una interacción, consistente en un conjunto de objetos y sus relaciones, incluyendo los mensajes que pueden ser enviados entre ellos; los diagramas de interacción tratan la vista dinámica de un sistema; un término genérico que se aplica a varios tipos de diagramas que enfatizan las interacciones de objetos, incluyendo diagramas de colaboración, diagramas de secuencia y diagramas de actividad.

diagrama de objetos Un diagrama que muestra un conjunto de objetos y sus relaciones en un momento determinado; los diagramas de objetos muestran el diseño o los procesos de un sistema desde un punto de vista estático.

diagrama de secuencia Un diagrama de interacción que hace énfasis en la ordenación temporal de los mensajes.

disparar Ejecutar una transición de estado.

ejecutable Un programa que puede ser ejecutado en un nodo.

elemento Un constituyente atómico de un modelo.

emisor El objeto que pasa una instancia de un mensaje a un objeto receptor.

enlace Una conexión semántica entre objetos; una instancia de una asociación.

envío El paso de una instancia de un mensaje desde un objeto emisor a un objeto receptor.

escenario Una secuencia específica de acciones que ilustran un comportamiento.

espacio de nombres Una parte del modelo en el cual pueden ser definidos y usados los nombres; dentro de un espacio de nombres, un nombre tiene un significado único.

especificación Una manifestación textual de la sintaxis y semántica de un bloque de construcción específico; una declaración declarativa de lo que algo es o hace.

estado Una condición o situación durante la vida de un objeto durante la cual éste satisface alguna condición, lleva a cabo alguna actividad o espera algún evento.

estado de acción Un estado que representa la ejecución de una acción atómica, típicamente la invocación de una operación.

estereotipo Una extensión del vocabulario de UML, que permite la creación de nuevos tipos de bloques de construcción que se derivan de otros existentes pero que son específicos a un problema particular.

estímulo Una operación o una señal.

evento La especificación de una ocurrencia significativa que tiene una ubicación en el tiempo y en el espacio; en el contexto de máquinas de estados, un evento es una ocurrencia de un estímulo que puede disparar una transición de estados.

exportar En el contexto de los paquetes, hacer un elemento visible fuera del espacio de nombres en que se encuentra.

extremo de enlace Una instancia de extremo de asociación.

extremo de asociación El final de una asociación, que conecta dicha asociación a un clasificador.

fachada Una fachada es un paquete estereotipado que no contiene más que referencias a elementos de modelos que pertenecen a otros paquetes. Es utilizado para proporcionar una vista "pública" de alguno de los contenidos de un paquete.

foco de control Un símbolo en un diagrama de secuencia que muestra el periodo de tiempo durante el cual un objeto está llevando a cabo una acción, tanto si es directamente como si es a través de una operación subordinada.

generalización Una relación de especialización/generalización en la que objetos del elemento especializado (el subtipo) son sustituibles por objetos del elemento generalizado (el supertipo).

guarda Una condición que ha de ser satisfecha para activar el disparo de una transición asociada.

hilo Un flujo de control ligero que puede ejecutarse concurrentemente con otros hilos en el mismo proceso.

herencia El mecanismo mediante el cual elementos más específicos incorporan la estructura y el comportamiento de elementos más generales.

herencia de interfaz La herencia del interfaz de un elemento más específico; no incluye la herencia de la implementación.

herencia múltiple Una variación semántica de generalización en la cual un tipo puede tener más de un supertipo.

herencia simple Una variación semántica de generalización en la cual un tipo puede tener únicamente un supertipo.

importación En el contexto de los paquetes, una dependencia que muestra el paquete cuyas clases pueden ser referenciadas dentro de unos paquetes dados (incluyendo paquetes recursivamente importados por él).

interfaz Una colección de operaciones que son utilizadas para especificar un servicio de una clase o de un componente.

instancia Una manifestación concreta de una abstracción; una entidad sobre la que pueden aplicarse un conjunto de operaciones y que tiene un estado que almacena los efectos de las operaciones; un sinónimo de objeto.

interacción Un comportamiento que consta de un conjunto de mensajes intercambiados por un conjunto de objetos dentro de un contexto particular para llevar a cabo un propósito específico.

jerarquía de contención Una jerarquía en el espacio de nombres que está formada por elementos y las relaciones de contención que existen entre ellos.

lenguaje de restricción de objetos (OCL) Un lenguaje formal utilizado para expresar restricciones libres de efectos laterales.

ligadura o enlazado (binding) La creación de un elemento a partir de una plantilla suministrando los argumentos para los parámetros de la plantilla.

línea de vida Véase *línea de vida de un objeto.*

línea de vida de un objeto Una línea en un diagrama de secuencia que representa la existencia de un objeto a lo largo de un periodo de tiempo.

localización La colocación de un componente sobre un nodo.

máquina de estados Un comportamiento que especifica las secuencias de estados por los que pasa un objeto durante su tiempo de vida en respuesta a eventos, junto con sus respuestas a dichos eventos.

marco de trabajo Un patrón de la arquitectura que proporciona una plantilla extensible para aplicaciones dentro de un domino específico.

mecanismo de extensibilidad Uno de los tres mecanismos (estereotipos, valores etiquetados y restricciones) que pueden ser utilizados para extender UML de forma controlada.

mensaje Una especificación de una comunicación entre objetos que lleva información con la expectativa de que de ella se seguirá alguna actividad; la recepción de una instancia de mensaje es normalmente considerada una instancia de un evento.

metaclase Una clase cuyas instancias son clases.

método La implementación de una operación.

modelo Una abstracción de un sistema cerrada semánticamente.

multiplicidad Una especificación del rango de cardinalidades permitidas que un conjunto puede tener.

nodo Un elemento físico que existe en tiempo de ejecución y que representa un recurso computacional, que en general tiene al menos alguna memoria y a menudo capacidad de procesamiento.

nombre Como se llama a una cosa, relación o diagrama; una cadena de caracteres utilizada para identificar un elemento.

nota Un comentario asociado a un elemento o a un conjunto de elementos.

objeto Véase *instancia*.

objeto activo Un objeto que posee un proceso o hilo y puede iniciar actividad de control.

objeto persistente Un objeto que existe después de que el proceso o hilo que lo creó ha dejado de existir.

objeto transitorio Un objeto que existe sólo durante la ejecución de un hilo o proceso que lo creó.

operación La implementación de un servicio que puede ser solicitado por cualquier objeto de la clase para afectar su comportamiento.

paquete Un mecanismo de propósito general para organizar elementos en grupos.

parámetro La especificación de una variable que puede ser cambiada, pasada o devuelta.

plantilla Un elemento parametrizado.

postcondición Una restricción que ha de ser cierta al completarse una operación.

precondición Una restricción que ha de ser cierta cuando una operación es invocada.

proceso Un flujo de control pesado que puede ejecutarse concurrentemente con otros procesos.

propiedad Un valor con nombre que denota una característica de un elemento.

proveedor Un tipo, clase o componente que proporciona servicios que pueden ser invocados por otros.

realización Una relación semántica entre clasificadores, en la que un clasificador especifica un contrato que otro clasificador garantiza llevar a cabo.

recepción El manejo de una instancia de un mensaje enviada por un objeto emisor.

receptor El objeto que maneja una instancia de un mensaje enviado por un objeto emisor.

relación Una conexión semántica entre elementos.

responsabilidad Un contrato u obligación de un tipo o clase.

restricción Una extensión de la semántica de un elemento de UML que permite añadir nuevas reglas o modificar las existentes.

rol El comportamiento específico de una entidad que participa en un contexto particular.

señal La especificación de un estímulo asíncrono comunicado entre instancias.

signatura El nombre y los parámetros de una característica de comportamiento.

sistema Una colección de subsistemas organizados para llevar a cabo un propósito específico y descritos por un conjunto de modelos, posiblemente desde distintos puntos de vista.

subsistema Una agrupación de elementos, de los que algunos constituyen una especificación del comportamiento ofrecido por los otros elementos contenidos.

subtipo En una relación de generalización, la especialización de otro tipo, el supertipo o supratipo.

supratipo o supertipo[1] En una relación de generalización, la generalización de otro tipo, el subtipo.

tarea Un camino simple de ejecución a través de un programa, un modelo dinámico o alguna otra representación de un flujo de control; un hilo o proceso.

tipo Un estereotipo de clase utilizado para especificar un dominio de objetos junto con las operaciones (pero no métodos) aplicables a los objetos.

tipo de datos Un tipo cuyos valores no tienen identidad. Los tipos de datos incluyen los tipos primitivos predefinidos (como números y cadenas) y los tipos enumerados (como los booleanos).

tipo primitivo Un tipo básico predefinido, como un entero o una cadena de caracteres.

transición Una relación entre dos estados indicando que un objeto en el primer estado llevará a cabo ciertas operaciones especificadas y entrará en el segundo estado cuando un evento especificado ocurra y se satisfagan las condiciones especificadas.

traza Una dependencia que indica una relación histórica o de proceso entre dos elementos que representan el mismo concepto, sin reglas específicas para derivar una de la otra.

unidad de despliegue Un conjunto de objetos o componentes que están asignados a una tarea o a un procesador como un grupo.

uso Una dependencia en la que un elemento (el cliente) requiere la presencia de otro elemento (el proveedor) para su correcto funcionamiento o implementación.

valor etiquetado Una extensión de las propiedades de un elemento UML, lo que permite la creación de nueva información en la especificación de ese elemento.

visibilidad Cómo un nombre puede ser visto y usado por otros.

vista Una proyección de un modelo, la cual es vista desde una perspectiva determinada o punto estratégico y que omite las entidades que no son relevantes para esta perspectiva.

A.4. Referencias

[1] OMG Unified Modeling Language Specification. Object Management Group, Framingham, MA, 1998. Internet:.

[2] James Rumbaugh, Ivar Jacobson y Grady Booch, *The Unified Modeling Language Reference Manual*, Reading, MA: Addison-Wesley, 1998.

[3] Grady Booch, James Rumbaugh e Ivar Jacobson, *The Unified Modeling Language User Guide*, Reading, MA: Addison-Wesley, 1998.

[4] Martin Fowler, *UML Distilled*, Reading, MA: Addison-Wesley, 1997.

[5] Hans-Erik Eriksson y Magnus Penker, *UML Toolkit*, New York: John Wiley & Sons, 1998.

---

[1] A lo largo de los 3 libros que describen UML y el Proceso Unificado, hemos traducido el término inglés "supertype" indistintamente como supertipo o supratipo. *N. del RT.*

Apéndice B

Extensiones de UML específicas del Proceso Unificado

B.1. Introducción

Este apéndice describe las extensiones de UML que precisa el Proceso Unificado. Estas extensiones se describen en términos de estereotipos y valores etiquetados, es decir, en términos de los mecanismos de extensión proporcionados por UML, así como con la notación gráfica usada para representar algunos de los estereotipos. Los estereotipos que no son parte o que difieren de las extensiones estándar de UML [1] y [2] están indicados con un asterisco (*).

Para una visión de conjunto de UML, consulte el Apéndice A.

B.2. Estereotipos

Estereotipo	Se aplica a	Descripción resumida
use case model	modelo	Modelo formado por actores, casos de uso y relaciones entre ambos; modelo que describe lo que el sistema debería hacer por sus usuarios y bajo qué restricciones.

Estereotipo	Se aplica a	Descripción resumida
use case system	paquete de más alto nivel	Paquete de más alto nivel del modelo de casos de uso. ("Sistema de casos de uso" es un subtipo de "paquete de más alto nivel".)
analysis model	modelo	Modelo de objetos cuyos propósitos son (1) describir los requisitos de forma precisa; (2) estructurarlos de manera que facilite su comprensión; y (3) servir de punto de partida para dar forma al sistema durante su diseño e implementación —incluyendo su arquitectura.
analysis system	paquete de más alto nivel	Paquete de más alto nivel del modelo de análisis. ("Sistema del análisis" es un subtipo de "paquete de más alto nivel".)
control class	clase	Clase del modelo de análisis que representa la coordinación, secuenciación y control de otros objetos, y que se usa a menudo para encapsular control referido a un determinado caso de uso.
entity class	clase	Clase del modelo de análisis usada para modelar información de larga duración y a menudo persistente.
boundery class	clase	Clase del modelo de análisis usada para modelar la interacción entre el sistema y sus actores, esto es, usuarios y sistemas externos.
use case realization-analysis*	colaboración	Colaboración del modelo de análisis que describe cómo se realiza un determinado caso de uso, en términos de clases de análisis (clases de control, entidad y entorno) y sus objetos de análisis correspondientes.
analysis package*	paquete	Paquete que proporciona los medios para organizar los artefactos del modelo de análisis en piezas manejables. Un paquete del análisis puede consistir en clases del análisis (clases de control, entidad y entorno), realizaciones de casos de uso-análisis, y otros paquetes del análisis (recursivamente).

Estereotipo	Se aplica a	Descripción resumida
service package*	paquete	Variante de un paquete del análisis que es usado en un nivel inferior de la jerarquía de paquetes del análisis (en el modelo de análisis) para estructurar el sistema de acuerdo a los servicios que proporciona.
design model	modelo	Modelo de objetos que describe la realización física de los casos de uso y se centra en cómo los requisitos funcionales y no funcionales, junto con otras restricciones referidas al entorno de implementación, afectan al sistema en consideración.
design system	subsistema de más alto nivel	Subsistema de más alto nivel del modelo de diseño. ("Sistema del diseño" es un es un subtipo de "paquete de más alto nivel".)
design class*	clase	Una clase del diseño representa una abstracción de una clase o construcción similar en la implementación del sistema.
use case realization-design*	colaboración	Colaboración del modelo de diseño que describe cómo se realiza un determinado caso de uso, en términos de subsistemas del diseño y clases del diseño y sus objetos correspondientes.
design subsystem	subsistema	Paquete que proporciona los medios para organizar los artefactos del modelo de diseño en piezas manejables. Un subsistema del diseño puede consistir en clases del diseño, realizaciones de casos de uso-diseño, interfaces, y otros subsistemas del diseño (recursivamente).
service subsystem*	subsistema	Variante de un subsistema del diseño que es usado en un nivel inferior de la jerarquía de subsistemas del diseño (en el modelo de diseño) para estructurar el sistema de acuerdo a los servicios que proporciona.
deployment model*	modelo	Modelo de objetos que describe la distribución física del sistema, en términos de cómo se distribuye la funcionalidad entre nodos computacionales.

Estereotipo	Se aplica a	Descripción resumida
implementation model	modelo	Modelo que describe cómo se implementan los elementos del modelo de diseño, por ejemplo, las clases de diseño, en términos de componentes como archivos de código fuente y ejecutables.
implementation system	subsistema de más alto nivel	Subsistema de más alto nivel del modelo de implementación. ("Sistema de implementación" es un subtipo de "paquete de más alto nivel".)
implementation subsystem	subsistema	Subsistema que proporciona los medios para organizar los artefactos del modelo de implementación en piezas manejables. Un subsistema de implementación puede consistir en componentes, interfaces y otros subsistemas de implementación (recursivamente).
test model*	modelo	Modelo que describe fundamentalmente cómo los componentes ejecutables (como las construcciones) del modelo de implementación son probados mediante las pruebas de integración y del sistema.
test system*	paquete de más alto nivel	Paquete de más alto nivel del modelo de pruebas. ("Sistema de pruebas" es un subtipo de "paquete de más alto nivel".)
test component*	componente	Componente que automatiza uno o varios procedimientos de pruebas o partes de ellos.

B.3. Valores etiquetados

Valor etiquetado	Se aplica a	Descripción resumida
survey description (descripción de inspección)	modelo de casos de uso	Descripción textual cuya intención es explicar el modelo de casos de uso en su conjunto.

Valor etiquetado	Se aplica a	Descripción resumida
how oferents (flujo de eventos)	caso de uso	Descripción textual de la secuencia de acciones del caso de uso.
special requirements (requisitos especiales)	caso de uso	Descripción textual que recopila todos los requisitos de un caso de uso que no se ven reflejados en su flujo de eventos (por ejemplo, los requisitos no funcionales).
special requirements (requisitos especiales)	clase del análisis (de control, entidad y de contorno)	Descripción textual que recopila requisitos no funcionales de una clase del análisis. Se trata de requisitos que se especifican en el análisis, pero que se manejan mejor en el diseño y la implementación.
how oferats-analysis (flujo de eventos-análisis)	realización de caso de uso--análisis	Descripción textual que explica y complementa los diagramas (y sus leyendas) que definen la realización del caso de uso.
special requirements (requisitos especiales)	realización de caso de uso--análisis	Descripción textual que recopila requisitos, como, por ejemplo, requisitos no funcionales, de una realización de caso de uso. Se trata de requisitos que se especifican en el análisis, pero que se manejan mejor en el diseño y la implementación.
implementation requirements (requisitos de implementación)	clase del diseño	Descripción textual que recopila requisitos, como, por ejemplo, requisitos no funcionales, de una clase del diseño. Se trata de requisitos que se especifican en el diseño, pero que se manejan mejor en la implementación.
how of events-design (flujo de eventos-diseño)	realización de caso de uso-diseño	Descripción textual que explica y complementa los diagramas (y sus leyendas) que definen la realización del caso de uso.
implementation requirements (requisitos de implementación)	realización de caso de uso-diseño	Descripción textual que recopila requisitos, como los requisitos no funcionales, de una realización de caso de uso. Se trata de requisitos que se especifican en el análisis, pero que se manejan mejor en el diseño y la implementación.

B.4. Notación gráfica

La mayoría de los estereotipos presentados en la Sección B.2 no imponen por sí mismos ningún símbolo gráfico, sino que pueden dibujarse mostrando las palabras clave del estereotipo entre comillas dobles (<< y >>) en el símbolo al que se aplica el estereotipo.

Sin embargo, las clases de control, entidad y contorno imponen nuevos símbolos que pueden dibujarse como se muestra en la Figura B.1.

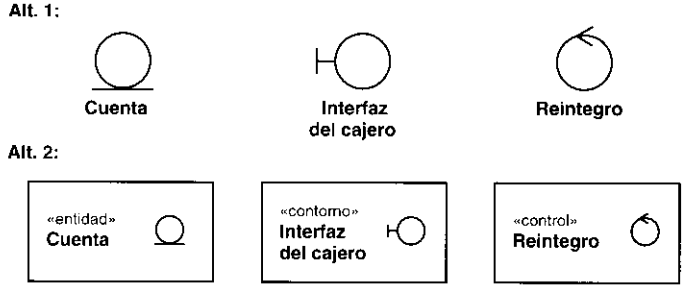


Figura B.1. Los tres estereotipos de clase estándar usados en el análisis.

B.5. Referencias

[1] OMG Unified Modeling Language Specification. Object Management Group, Framingham, MA, 1998. Internet: www.omg.org.

[2] James Rumbaugh, Ivar Jacobson, and Grady Booch, *Unified Modeling Language Reference Manual,* Reading, MA: Addison-Wesley, 1998.

Apéndice C

Glosario general

C.1. Introducción

Este apéndice recopila y define los términos generales usados para describir el Proceso Unificado, exceptuando los términos relacionados con UML y los términos relacionados con extensiones de UML específicas para el Proceso Unificado. Para una descripción resumida de estos términos, *véase* el Apéndice A "Visión general de UML", y el Apéndice B "Extensiones de UML específicas del Proceso Unificado".

C.2. Términos

abstracción Características esenciales de una entidad que la distinguen de cualquier otra clase de entidades. Una abstracción define un contorno relativo a la perspectiva del observador.

actividad Unidad tangible de trabajo realizada por un trabajador en un flujo de trabajo, de forma que (1) implica una responsabilidad bien definida para el trabajador, (2) produce un resultado bien definido (un conjunto de artefactos) basado en una entraba bien definida (otro conjunto de artefactos), y (3) representa una unidad de trabajo con límites bien definidos a la que, probablemente se refiera el plan de proyecto al asignar tareas a los individuos. También puede verse como la ejecución de una operación por un trabajador. Véase *artefacto*, *trabajador*.

object request broker (ORB) Mecanismo para la organización y reenvío de mensajes a objetos distribuidos en entornos heterogéneos. Véase *distribución*.

análisis (flujo de trabajo) Flujo de trabajo fundamental cuyo propósito principal es analizar los requisitos descritos en la captura de requisitos, mediante su refinamiento y estructuración. El objetivo de esto es (1) lograr una comprensión más precisa de los requisitos, y (2) obtener una descripción de los requisitos que sea fácil de mantener y que nos ayude a dar estructura al sistema en su conjunto —incluyendo su arquitectura.

aplicación (sistema) Sistema que ofrece a un usuario final un conjunto coherente de casos de uso.

arquitectura Conjunto de decisiones significativas acerca de la organización de un sistema software, la selección de los elementos estructurales a partir de los cuales se compone el sistema, y las interfaces entre ellos, junto con su comportamiento, tal y como se especifica en las colaboraciones entre esos elementos, la composición de estos elementos estructurales y de comportamiento en subsistemas progresivamente mayores, y el estilo arquitectónico que guía esta organización: estos elementos y sus interfaces, sus colaboraciones y su composición. La arquitectura del software se interesa no sólo por la estructura y el comportamiento, sino también por las restricciones y compromisos de uso, funcionalidad, funcionamiento, flexibilidad al cambio, reutilización, comprensión, economía y tecnología, así como por aspectos estéticos.

artefacto Pieza de información tangible que (1) es creada, modificada y usada por los trabajadores al realizar actividades; (2) representa un área de responsabilidad, y (3) es candidata a ser tenida en cuenta para el control de la configuración. Un artefacto puede ser un modelo, un elemento de un modelo, o un documento. Véase *trabajador, actividad.*

artefacto de gestión Artefacto que no es de ingeniería, por ejemplo un plan de proyecto creado por el jefe de proyecto. Véase *artefacto de ingeniería.*

artefacto de ingeniería Artefacto creado en los flujos de trabajo fundamentales. Véase *flujo de trabajo fundamental.*

capa Parte bien definida de un sistema, definida a partir de paquetes o subsistemas. Véase *capa general de aplicación, capa específica de aplicación.*

capa del software del sistema Capa que contiene el software para la infraestructura de computación y comunicación, por ejemplo, sistemas operativos, sistemas de gestión de bases de datos, interfaces para determinados componentes hardware, etc. Se trata de la capa inferior de la jerarquía de capas. Véase *capa intermedia.*

capa específica de aplicación La parte (paquetes o subsistemas) de un sistema que es específica de la aplicación y no es compartida por otras partes (subsistemas). Esta capa utiliza la capa general de aplicación. Véase *capa general de aplicación.*

capa general de aplicación La parte (paquetes o subsistemas) de un sistema que puede ser reutilizada dentro de un negocio o dominio. Esta capa es utilizada por la capa específica de aplicación. Véase *capa específica de aplicación.*

capa intermedia (middleware) Capa que ofrece bloques de construcción reutilizables (paquetes o subsistemas) a marcos de trabajo y servicios independientes de la plataforma, para cosas como computación con objetos distribuidos, o interoperabilidad en entornos heterogéneos. Ejemplos son los object request brokers (ORB), los marcos de trabajo independientes de la plataforma para crear interfaces de usuario gráficas o, en general, cualquier producto que llevan a cabo mecanismos de diseño genéricos. Véase *capa de software del sistema, object request brokers, mecanismo de diseño.*

caso de prueba Especificación de un caso para probar el sistema, incluyendo qué probar, con qué entradas y resultados y bajo qué condiciones.

centrado en la arquitectura En el contexto del ciclo de vida del software, significa que la arquitectura de un sistema se usa como un artefacto primordial para la conceptualización, construcción, gestión y evolución del sistema en desarrollo.

ciclo de vida del software Ciclo que cubre cuatro fases en el siguiente orden: inicio, elaboración, construcción y transición. Véase *inicio*, *elaboración*, *construcción* y *transición*.

cliente Persona, organización o grupo de personas que encarga la construcción de un sistema, ya sea empezando desde cero, o mediante el refinamiento de versiones sucesivas.

cohesivo Capacidad de una entidad (por ejemplo, un sistema, subsistema o paquete) de mantener juntas sus partes.

concurrencia Ocurre cuando varios trabajos (flujos de control, procesos) más o menos independientes comparten un único dispositivo hardware (procesador) al mismo tiempo.

construcción Versión ejecutable del sistema, por lo general, de una parte específica del mismo. El desarrollo transcurre a través de una sucesión de construcciones.

defecto Anomalía del sistema, por ejemplo un síntoma de un error en el software descubierto durante las pruebas, o un problema descubierto durante una reunión de revisión. Véase *pruebas*.

desarrollador Trabajador participante en un flujo de trabajo fundamental. Por ejemplo, un ingeniero de casos de uso, un ingeniero de componentes, etc. Véase *flujo de trabajo fundamental*.

desarrollo basado en componentes (DBC) Creación y despliegue de sistemas con gran cantidad de software mediante su ensamblado a partir de componentes, así como el desarrollo y recopilación de estos componentes.

descripción arquitectónica Descripción de la arquitectura del sistema, incluyendo las vistas arquitectónicas de los modelos. Véase *vista arquitectónica*, *vista arquitectónica del modelo de casos de uso*, *vista arquitectónica del modelo de análisis*, *vista arquitectónica del modelo de diseño*, *vista arquitectónica del modelo de despliegue*, *vista arquitectónica del modelo de implementación*.

diseño (flujo de trabajo) Flujo de trabajo fundamental cuyo propósito principal es el de formular modelos que se centran en los requisitos no funcionales y el dominio de la solución, y que prepara para la implementación y pruebas del sistema.

despliegue Ocurre cuando varios trabajos más o menos independientes (flujos de control, procesos) se distribuyen entre diferentes dispositivos hardware (procesadores).

dominio Área de conocimiento o actividad caracterizada por un conjunto de conceptos y terminología comprendidos por los practicantes de ese dominio.

dominio de la solución Dominio en el que se define una solución (para un problema) —por lo general, una solución que pone de manifiesto el diseño y la implementación de un sistema. El dominio de la solución es comprendido, por lo general, por los desarrolladores del sistema. Véase *dominio*, *desarrollador*.

dominio del problema dominio sobre el que se define un problema —generalmente un problema que debe ser 'resuelto' por un sistema. El dominio del problema es comprendido, por lo general, por el cliente del sistema. Véase *dominio*, *cliente*.

enfoque en cascada Enfoque para el desarrollo de un sistema en el cual el desarrollo se organiza en una secuencia lineal de trabajos, por ejemplo, en el orden siguiente: captura de requisitos, análisis, diseño, implementación y pruebas. Véase *requisitos*, *análisis*, *diseño*, *implementación*, *pruebas*.

estilo arquitectónico Los sistemas que comparten una estructura de alto nivel y mecanismos clave similares se dice que tienen un estilo arquitectónico similar.

evaluación de las pruebas Evaluación de los resultados del esfuerzo de prueba, como cobertura de casos de prueba, cobertura de código y estado de los defectos. Véase *prueba*, *caso de prueba*, *defecto*.

fase Periodo de tiempo entre dos hitos principales de un proceso de desarrollo. Véase *hito principal*, *inicio*, *elaboración*, *construcción*, *transición*.

fase de construcción Tercera fase del ciclo de vida del software, en la que el software es desarrollado a partir de una línea base de la arquitectura ejecutable, hasta el punto en el que está listo para ser transmitido a la comunidad de usuarios.

fase de elaboración Segunda fase del ciclo de vida, en la que se define la arquitectura.

fase de inicio Primera fase del ciclo de vida del software, en la que la idea inicial para el desarrollo es refinada hasta el punto de quedar lo suficientemente bien establecida como para garantizar la entrada en la fase de elaboración.

fase de transición Cuarta fase del ciclo de vida del software, en la que el software es puesto en manos de la comunidad de usuarios.

fiabilidad Habilidad de un sistema para comportarse correctamente sobre su entorno de ejecución real. Puede medirse, por ejemplo, en función de la disponibilidad del sistema, su exactitud, el tiempo medio entre fallos, los defectos por cada 1.000 líneas de código o los defectos por clase.

flujo de trabajo Realización de un caso de uso de negocio o parte de él. Puede describirse en términos de diagramas de actividad, que incluyen a los trabajadores participantes, las actividades que realizan y los artefactos que producen. Véase *flujo de trabajo fundamental*, *flujo de trabajo de una iteración*.

flujo de trabajo de una iteración Flujo que trabajo que representa una integración de los flujos de trabajo fundamentales: captura de requisitos, análisis, diseño, implementación y pruebas. Descripción de una iteración que incluye los trabajadores participantes, las actividades que éstos realizan y los artefactos que producen. Véase *flujo de trabajo*.

flujo de trabajo fundamental Cada uno de los flujos de trabajo de requisitos, análisis, diseño, implementación o pruebas. Véase *flujo de trabajo*, *requisitos*, *análisis*, *diseño*, *implementación*, *pruebas*.

framework Véase *marco de trabajo*.

gestión de la configuración Tarea de definir y mantener las configuraciones y versiones de los artefactos. Esto incluye la definición de líneas base, el control de versiones, el control de estado y el control del almacenamiento de los artefactos. Véase *artefacto*, *línea base*.

dirigido por los casos de uso En el contexto del ciclo de vida del software, indica que los casos de uso se utilizan como artefacto principal para definir el comportamiento deseado para el sistema, y para comunicar este comportamiento entre las personas involucradas en el sistema. También indica que los casos de uso son la entrada principal para el análisis, diseño, implementación y pruebas del sistema, incluyendo la creación, verificación y validación de la arquitectura del sistema. Véase *análisis*, *diseño*, *implementación*, *pruebas*, *arquitectura*.

dirigido por los riesgos En el contexto del ciclo de vida del software, indica que cada nueva versión se centra en atacar y reducir los riesgos más significativos para el éxito del proyecto.

dirigido por modelos En el contexto del ciclo de vida del software, significa que el sistema en desarrollo está organizado en función de diferentes modelos con propósitos específicos, y cuyos elementos están relacionados unos con otros.

hito principal Punto en el que han de tomarse importantes decisiones de negocio. Cada fase acaba en un hito principal en el cual los gestores han de tomar decisiones cruciales de continuar o no el proyecto, y decidir sobre la planificación, presupuesto y requisitos del mismo. Consideramos a los hitos principales como puntos de sincronización en los que coinciden una serie de objetivos bien definidos, se completan artefactos, se toman decisiones de pasar o no a la fase siguiente, y en los que las esferas técnica y de gestión entran en conjunción.

hito secundario hito intermedio entre dos hitos principales. Puede existir, por ejemplo, al acabar una iteración, o cuando se finaliza una construcción en una iteración. Véase *hito principal*, *iteración*, *construcción*.

implementación (flujo de trabajo) Flujo de trabajo fundamental cuyo propósito esencial es implementar el sistema en términos de componentes, es decir, código fuente, *guiones*, ficheros binarios, ejecutables, etc.

incremento Parte pequeña y manejable del sistema, normalmente la diferencia entre dos construcciones sucesivas. Cada iteración resultará, al menos, en una nueva construcción, de forma que se añada un incremento al sistema. Sin embargo, se puede crear una secuencia de construcciones en una sola iteración, cada una de ellas añadiendo al sistema un pequeño incremento. Por tanto, una iteración añade al sistema un incremento mayor, posiblemente acumulado a lo largo de varias construcciones. Véase *construcción*, *iteración*.

ingeniería directa En el contexto del desarrollo del software, la transformación de un modelo en código a través de su traducción a un determinado lenguaje de implementación. Véase *ingeniería inversa*.

ingeniería inversa En el contexto del desarrollo de software, transformación del código en un modelo a través de su traducción desde un determinado lenguaje de implementación. Véase *ingeniería directa*.

integración del sistema Compilación y ensamblado de parte de los componentes de un sistema en uno o más ejecutables (que también son componentes).

integración incremental En el contexto del ciclo de vida del software, proceso que implica la integración continua de la arquitectura del sistema para producir versiones, de forma que cada nueva versión incluya mejoras incrementales sobre las anteriores.

interfaz de usuario Interfaz a través de la cual un usuario interactúa con un sistema.

iteración Conjunto de actividades llevadas a cabo de acuerdo a un plan (de iteración) y unos criterios de evaluación, que lleva a producir una versión, ya sea interna o externa. Véase *plan de iteración, versión, versión interna, versión externa.*

iterativo En el contexto del ciclo de vida del software, proceso que implica la gestión de una serie de versiones ejecutables.

juego de aplicaciones Conjunto de diferentes aplicaciones (sistemas) cuyo propósito es trabajar conjuntamente para proporcionar valor añadido a algunos actores. Véase *aplicación (sistema).*

Lenguaje Unificado de Modelado (UML) Lenguaje estándar para el modelado de software —lenguaje para visualizar, especificar, construir y documentar los artefactos de un sistema con gran cantidad de software. Lenguaje usado por el Proceso Unificado. Lenguaje que permite a los desarrolladores visualizar el producto de su trabajo (artefactos) en esquemas o diagramas estandarizados. Véase *artefacto, Proceso Unificado, desarrollador.*

línea base Conjunto de artefactos revisados y aprobados que (1) representa un punto de acuerdo para la posterior evolución y desarrollo, y (2) solamente puede ser modificado a través de un procedimiento formal, como la gestión de cambios y configuraciones. Véase *línea base de la arquitectura, gestión de la configuración.*

línea base de la arquitectura Línea base resultado de la fase de elaboración, centrada en la arquitectura del sistema. Véase *elaboración, arquitectura, línea base.*

marco de trabajo Microarquitectura que proporciona una plantilla incompleta para sistemas de un determinado dominio. Puede tratarse, por ejemplo, un subsistema construido para ser ampliado y reutilizado.

total de casos de uso Conjunto completo de acciones en los casos de uso de un modelo de casos de uso.

mecanismo Solución común a un problema o requisito común. Ejemplos son los mecanismos de diseño que proporcionan facilidades de persistencia o distribución en el modelo de diseño.

mecanismo de diseño Conjunto de clases del diseño, colaboraciones e incluso subsistemas del modelo de diseño que llevan a cabo requisitos comunes, como requisitos de persistencia, distribución y funcionamiento.

modelado visual Visualización de productos (artefactos) en esquemas o diagramas estandarizados.

patrón Solución común a un problema común de un determinado contexto.

patrón arquitectónico Patrón que define una cierta estructura o comportamiento, por lo general para la vista arquitectónica de un determinado modelo. Ejemplos son los patrones Capa, Cliente-Servidor, 3 Niveles y EntrePares, Layer, Client-Server, Three-tier y Peer-to-peer, cada uno de los cuales define una cierta estructura para el modelo de despliegue y sugiere también cómo deben ser asignados a sus nodos los componentes (la funcionalidad). Véase *patrón, vista de la arquitectura.*

plan de emergencia Plan que describe cómo actuar si determinados riesgos se materializan. Véase *riesgo.*

plan de iteración Plan detallado para una iteración. Plan que determina, para una iteración, los costes previstos, en términos de dinero y recursos, y los resultados previstos, en términos de artefactos. Plan que establece quién debe hacer qué en la iteración y en qué orden. Esto se lleva a cabo asignando trabajadores y describiendo un flujo de trabajo detallado para la iteración. Véase *iteración*, *artefacto*, *trabajador*, *flujo de trabajo de una iteración.*

plan de proyecto Plan que esboza el 'mapa de carreteras' global de un proyecto, incluyendo la agenda, fechas y criterios de los hitos principales, y la descomposición de las fases en iteraciones. Véase *proyecto*, *hito principal.*

plan de pruebas Plan que describe las estrategias, recursos y programación de las pruebas.

portabilidad Grado en el que un sistema, en funcionamiento en un determinado entorno de ejecución, puede ser convertido fácilmente en un sistema funcionando en otro entorno de ejecución.

procedimiento de pruebas Especificación de cómo llevar a cabo uno o varios casos de prueba o partes de ellos. Véase *caso de prueba.*

proceso de desarrollo de software Proceso de negocio, o caso de uso de negocio, de un negocio de desarrollo de software. Conjunto total de actividades necesarias para transformar los requisitos de un cliente en un conjunto consistente de artefactos que representan un producto software y —en un punto posterior en el tiempo— para transformar cambios en dichos requisitos en nuevas versiones del producto software. Véase *proceso de negocio*, *Proceso Unificado.*

proceso de negocio Conjunto total de actividades necesarias para producir un resultado de valor percibido y medible para un cliente individual de un negocio.

Proceso Unificado Proceso de desarrollo de software basado en el Lenguaje Unificado de Modelado, y que es iterativo, centrado en la arquitectura y dirigido por los caso de uso y los riesgos. Proceso que se organiza en cuatro fases: inicio, elaboración, construcción y transición, y que se estructura en torno a cinco flujos de trabajo fundamentales: recopilación de requisitos, análisis, diseño, implementación y pruebas. Proceso que se describe en términos de un modelo de negocio, el cual está a su vez estructurado en función de tres bloques de construcción primordiales: trabajadores, actividades y artefactos. Véase *proceso de desarrollo de software*, *Lenguaje Unificado de Modelado*, *iterativo*, *centrado en la arquitectura*, *dirigido por los casos de uso*, *dirigido por los riesgos*, *fase*, *inicio*, *elaboración*, *construcción*, *transición*, *flujo de trabajo fundamental*, *requisitos*, *análisis*, *diseño*, *implementación*, *pruebas*, *trabajador*, *actividad* y *artefacto.*

prototipo arquitectónico Fundamentalmente, prototipo ejecutable que se centra en la vista arquitectónica del modelo de implementación y en los componentes que ponen de manifiesto el prototipo. Si un prototipo arquitectónico es evolutivo, probablemente estará fundado sobre una línea base, y su estructura se pondrá de manifiesto mediante una descripción arquitectónica más completa, aunque también prototípica (o esbozada), incluyendo todas sus vistas arquitectónicas. Véase *prototipo evolutivo*, *línea base*, *descripción arquitectónica*, *vista arquitectónica.*

prototipo de interfaz de usuario Fundamentalmente, un prototipo ejecutable de una interfaz de usuario, pero que puede, en los momentos iniciales del desarrollo, consistir únicamente en dibujos en papel, diseños de pantallas, etc. Véase *interfaz de usuario.*

prototipo evolutivo Prototipo que evoluciona y es refinado para convertirse finalmente en parte de un sistema en desarrollo. Prototipo que es candidato a ser sujeto de la gestión de configuraciones. Véase *gestión de configuraciones.*

prototipo exploratorio Prototipo usado únicamente con motivos exploratorios y que es desechado una vez que se han cumplido estos propósitos. Prototipo que no es candidato a ser sujeto de la gestión de configuraciones. Véase *gestión de configuraciones.*

proyecto Esfuerzo de desarrollo para llevar un sistema a lo largo de un ciclo de vida. Véase *ciclo de vida del software.*

proyecto novedoso Un proyecto que no tiene precedentes. Véase *proyecto.*

prueba de regresión Repetición de las pruebas de una construcción (o parte de ella) que ya había sido probada con anterioridad. Las pruebas de regresión se realizan fundamentalmente para comprobar que "la antigua funcionalidad" de "construcciones antiguas" aún sigue comportándose de manera correcta cuando se añade "funcionalidad nueva" en una "construcción nueva". Véase *prueba, construcción.*

pruebas (flujo de trabajo) Flujo de trabajo fundamental cuyo propósito esencial es comprobar el resultado de la implementación mediante las pruebas de cada construcción, incluyendo tanto construcciones internas como intermedias, así como las versiones finales del sistema que van a ser entregadas a terceras partes. Véase *implementación, construcción, versión interna, versión externa.*

requisito Condición o capacidad que debe cumplir un sistema.

requisitos (flujo de trabajo) Flujo de trabajo fundamental cuyo propósito esencial es orientar el desarrollo hacia el sistema correcto. Esto se lleva a cabo mediante la descripción de los requisitos del sistema de forma tal que se pueda llegar a un acuerdo entre el cliente (incluyendo los usuarios) y los desarrolladores del sistema, acerca de lo que el sistema debe hacer y lo que no. Véase *requisito, cliente, desarrollador.*

requisito adicional requisito de carácter general que no puede ser asociado a un caso de uso en concreto o a una clase concreta del mundo real, tal como una clase entidad del dominio o del negocio. Véase *requisito.*

requisito de rendimiento Requisito que impone condiciones de comportamiento sobre un requisito funcional, como velocidad, rendimiento, tiempo de respuesta y uso de memoria. Véase *requisito funcional, requisito.*

requisito funcional Requisito que especifica una acción que debe ser capaz de realizar el sistema, sin considerar restricciones físicas; requisito que especifica comportamiento de entrada/salida de un sistema. Véase *requisito.*

requisito no funcional Requisito que especifica propiedades del sistema, como restricciones del entorno o de implementación, rendimiento, dependencias de la plataforma, mantenibilidad, extensibilidad o fiabilidad. Requisito que especifica restricciones físicas sobre un requisito funcional. Véase *requisito, requisito de funcionamiento, fiabilidad, requisito funcional.*

riesgo Variable de un proyecto que pone en peligro o impide su éxito. Los riesgos pueden consistir en que un proyecto experimente sucesos no deseados, como retrasos en la programa-

ción, desviaciones de costes o una cancelación definitiva. Véase *riesgo técnico*, *riesgo no técnico*.

riesgo no técnico Riesgo relacionado con artefactos de gestión y con aspectos como los recursos (personas) disponibles, sus competencias, o con las fechas de entrega. Véase *artefacto de gestión*, *riesgo*, *riesgo técnico*.

riesgo técnico Riesgo relacionado con los artefactos de ingeniería y también con aspectos como las tecnologías de implementación, la arquitectura o el rendimiento. Véase *artefacto de ingeniería*, *arquitectura*, *requisito de rendimiento*, *riesgo no técnico*.

robustez capacidad de una entidad (por lo general, un sistema) para adaptarse al cambio.

sistema legado Un sistema existente, 'heredado' por un proyecto. Normalmente, un sistema viejo que fue creado usando tecnologías de implementación más o menos obsoletas, pero que, en cualquier caso, debe ser incorporado o reutilizado —ya sea en su totalidad o en parte— al construir un nuevo sistema durante el proyecto. Véase *proyecto*.

trabajador Puesto que puede ser asignado a una persona o equipo, y que requiere responsabilidades y habilidades como realizar determinadas actividades o desarrollar determinados artefactos. Véase *actividad*, *artefacto*.

usuario Humano que interactúa con un sistema.

versión Conjunto de artefactos relativamente completo y consistente —que incluye posiblemente una construcción— entregado a un usuario interno o externo; entrega de tal conjunto. Véase *artefacto*, *construcción*.

versión externa Versión expuesta a los clientes y usuarios, externos al proyecto y sus miembros. Véase *versión*.

versión interna Una versión no expuesta a los clientes y usuarios, sino sólo de forma interna, al proyecto y sus miembros. Véase *versión*.

vista Proyección de un modelo, que es visto desde un perspectiva dada o un lugar estratégico, y que omite las entidades que no son relevantes para esta perspectiva.

vista arquitectónica del modelo de análisis Vista de la arquitectura de un sistema, abarcando las clases, paquetes y realizaciones de casos de uso del análisis; vista que fundamentalmente aborda el refinamiento y estructuración de los requisitos del sistema. La estructura de esta vista se preserva en la medida de lo posible cuando se diseña e implementa la arquitectura del sistema. Véase *vista arquitectónica del modelo de diseño*, *vista arquitectónica del modelo de implementación*.

vista arquitectónica del modelo de casos de uso Vista de la arquitectura de un sistema abarcando los casos de uso significativos desde un punto de vista arquitectónico.

vista arquitectónica del modelo de despliegue Vista de la arquitectura de un sistema abarcando los nodos que forman la topología hardware sobre la que se ejecuta el sistema; vista que aborda la distribución, entrega e instalación de las partes que constituyen el sistema físico.

vista arquitectónica del modelo de diseño Vista de la arquitectura de un sistema, abarcando las clases, subsistemas, interfaces y realizaciones de casos de uso del diseño que forman el

vocabulario del dominio de la solución del sistema; vista que abarca también los hilos y procesos que establecen la concurrencia y mecanismos de sincronización del sistema; vista que aborda los requisitos no funcionales, incluyendo los requisitos de rendimiento y capacidad de crecimiento de un sistema.

vista arquitectónica del modelo de implementación Vista de la arquitectura de un sistema, abarcando los componentes usados para el ensamblado y lanzamiento del sistema físico; vista que aborda la gestión de la configuración de las versiones del sistema, constituida por componentes independientes que pueden ser ensamblados de varias formas para producir un sistema ejecutable.

Índice

A

acción, 5
actividad, 126, 136-138
actor, 20-21, 33, 39, 115-116, 120, 140-142, 154-155, 336, 351-353
  particular, 141-142
agregación, 199-200, 246
agrupamiento, 408, 411
alcance (sistema), 330
Alexander, Christopher, 67
analista de sistemas, 134-135, 138
anotación, 408, 411
apuesta económica, 340-350
arquitecto, 71-72
arquitectura, 55-79
  candidata, 329-331
artefacto, 18
  de gestión, 19
  de ingeniería, 19
aseguramiento de la calidad, 28
asignación de tiempo, 311, 312
asociación, 199-200
atributo, 173, 198-199, 245-246, 275-276
autocontenido, 20

B

Boehm, Barry, 82, 86

C

"calle", 23
campeón, 400
característica, 108-109, 335
caso de
  negocio, 81
  prueba, 52-53, 283-285
    candidato, 35
  uso, 5, 31-53
    abstracto, 159, 161
    candidato, 39-40
    concreto, 159, 161
    «real», 159, 161
ciclo
  de vida, 8-12, 17-18
    del software, 28
clase
  activa, 76-77, 210, 269, 410, 232-234, 249
  de análisis, 173-178
  de control, 176-177, 179, 195, 244
  de diseño, 47, 50, 209-210, 243-249
  de diagrama, 42, 43, 47
  de dominio, 112-115, 118-119, 189-190
  de entidad, 175, 179, 184, 192, 194, 244
  de equivalencia, 277
  de estereotipo, 42

de interfaz, 42, 174-175, 179
de negocio, 175
de responsabilidades, 34, 45, 356
frontera, 174-175, 179, 195-196
notación gráfica, 409, 424
rol, 41, 45, 114, 198
clasificador, 33
cliente, 382, 389-392
código, 18
código fuente, 18, 274-275
código máquina, 18
colaboración, 21, 34
comillas («»), 9, 42
componente, 50-52, 257-259
de dependencia, 259
de fichero, 274-276
de interfaz, 262
de prueba, 287, 294
ejecutable, 269-270
configuración de red, 222-224, 359
Constantine, Larry, 156
construcción, 62, 87-89
contrato, interfaz, 364-365
coste, 323-324

D

defecto, 288, 298, 388
dependencia
de compilación, 259
de traza, 10, 21-22, 37, 166, 258
entre componentes, 259
entre paquetes del análisis, 192
entre subsistemas, 228-231, 250
notación gráfica, 411
descripción
de caso de uso, 142-152
de flujo de eventos, 45
de la arquitectura, 57-58, 69-73
diagrama
de actividad, 23, 129, 151
de colaboración, 44-45, 129
de estados, 33, 129
de interacción, 178-180, 212-213
de secuencia, 47-48, 129
directiva de reingeniería, 399-400
director de prueba, 288, 376
directriz de arquitectura, 64-67, 69
diseñador de interfaz de usuario, 135-137
Drucker, Peter F., 85

E

ejecutivo, 398-400
elemento (UML), 407-408
elemento de comportamiento, 408, 410
elemento estructural, 408-410
entidad de negocio, 116, 118
entidad, clase, 42, 175, 179
entorno de desarrollo, 334
especificación funcional, 5
especificador de casos de uso, 135, 137, 147
estandarización, 59
estereotipo, 23, 419-422
estilo arquitectónico, 57
estructura de equipo, 14-15, 90, 347
evaluación, 313-314

F

fase, 8, 10-12
de construcción, 12, 82, 101, 367-379
de elaboración, 11, 82, 101, 345-365
de evaluación, 324-326
de inicio, 11, 81, 101, 327-344
de iteración, 95, 100-101
de planificación, 311
de transición, 12, 82, 101, 381-393
fiabilidad, 110
flujo de trabajo, 10, 22-24, 126

G

generalización, 159, 200, 236, 247, 412
gestión, 398-401
de requisitos, 28
de riesgo, 14
gestor de proyecto, 313, 325
glosario (artefacto), 132-133
guión, prueba, 287

H

Harness (prueba de), 302
herramienta, 13, 25-29
de programación, 28
hito, 10, 81, 98

I

incremento, 7, 8, 97
véase también iteración
ingeniero
de casos de uso, 185-186, 219

de componentes, 186, 220, 266, 278-279, 374-375, 378
de pruebas de integración, 288
de pruebas del sistema, 289
de pruebas, 289, 313
instalación (producto), 389, 390
instancia
de caso de uso, 129-130
de clase, 37
de proceso, 22
integrador del sistema, 266, 376-377
interacción, 410
interesado, 18-19
interfaz, 409
de usuario, 110, 133, 135, 152-158, 373
inversión (retorno de la), 341, 363
iteración, 6-8, 10, 82-102
controlada, 7
de flujo de trabajo, 94-95
genérica, 94-97, 303-304, 308-310
iteraciones de organización, 101

J

Java, 228-229, 278, 358
Jones, Capers, 87

L

Layer, 68
lenguaje de programación, 209, 275
lenguaje unificado de modelado (UML), 4, 398, 407-424
línea base de la arquitectura, 65-66
línea de vida, 242

M

manual de usuario, 38, 369
máquina de estados, 410
matriz de prueba, 285
mecanismo, 70
de diseño genérico, 234-238
de extensibilidad, 408-409, 412, 419-424
mensaje, 39, 211-213
mentor, 401
middleware, 62, 227-229
migración de datos, 389
miniproyecto, 7, 83
modelo, 19-22
(artefacto), 99
de análisis, 9-10, 166-204
de casos de uso, 5, 10, 21, 127-128
de despliegue, 10, 76-77
de diseño, 10, 21, 205-253
de implementación, 10, 32, 256-279
de negocio, 10, 115-121
del dominio, 10, 112-116, 118-119
en cascada, 86, 90, 95-96, 326
modularidad de funciones, 60

N

nodo, 76
nota, 411
notación "en forma de pez", 24
núcleo (flujo de trabajo), 94-95, 100, 308-310

O

objeto, 25, 177, 179, 195-196, 212, 239-241
activo, 76-77, 232, 270
véase también clase
organización (tipos de), 328, 329
organizaciones de software, 328-329

P

paquete, 411, 181-183, 189-192, 215
de análisis, 181-184, 201-202
de servicio, 182-183
patrón
Broker, 68, 69
Client/Server, 68
de arquitectura, 67-69
de diseño Proxy, 67
de diseño, 67
Layers, 68
organización, 18
persistencia, 235, 244
personas, 13-17
plan de construcción de integración, 264
planificación, 310-314
plantilla, 67
procedimiento de prueba, 52, 286-287, 294-295
proceso
automatizado, 25-27
común, 25
de desarrollo de software, 4, 13
de desarrollo, 4, 13, 22-29
de especialización, 24-25
genérico, 24
unificado, 3-12
Proceso Objectory de Rational, 403

Proceso Unificado de Rational, 403
producto, 9, 13
  carrera, 369
  de mercado, 388-390
prototipo, 133, 135, 152-158, 306, 332-333, 352, 373
  de interfaz de usuario, 133, 135, 152-158
proyecto, 13, 22
  green-field, 17
prueba, 281-299
  alfa, 382
  de aceptación, 384, 386-387
  de caja blanca, 35, 284
  de caja negra, 34, 283
  de configuración, 285
  de especificación, 277
  de estrés, 300
  de instalación, 285
  de integración, 264-265
  de regresión, 95-96, 282-283, 294
  de unidad, 276-279
  del sistema, 294, 297, 362
  negativa, 285

R

ranking (caso de uso), 316-317, 367
realización de casos de uso, 44-48, 117-118, 177-181, 210-212, 242, 284
recurso, 17
relación, 21-22, 33-34
  de cliente individual, 389
  de extensión, 160, 161
  de inclusión, 161-162
  de uso, 159
requisito
  de funcionalidad, 35-36
  de interfaz, 121
  de rendimiento, 110, 150
  especial, 132, 150, 180-181, 193, 197, 201, 249
  físico, 121
  no funcional, 40, 110
  suplementario, 111, 121-122
requisitos, 22, 105-123
resguardo (programa de prueba), 259, 376
restricción
  de diseño, 121
  de implementación, 122
resultado de valor, 141-142
retroalimentación, 15
reutilización, 59-60
riesgo de rendimiento, 92
rol, 16-17, 41, 45, 114, 120, 128-129, 140, 198
ruta (caso de uso), 40, 130, 240
  básica, 147-149

S

sentencia
  de caso de acción, 399
  de visión, 330
sistema
  AXE de Ericsson, 60-61
  de sistemas, 131-132
  heredado, 61
  prefabricado, 399
  software, 227-229
software, 18-22
subsistema, 49-50
  de más alto nivel, 21
  de servicio, 49-51, 60-61

T

trabajador, 16-17, 17-20

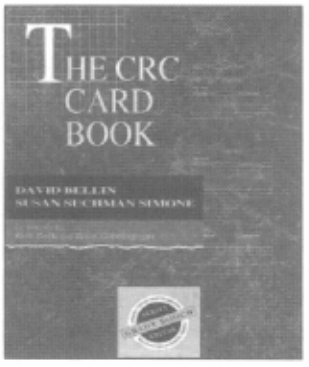
U

unidad de trabajo, 116
usuario, 5

V

validación de tercera parte, 382
valor etiquetado, 422-423
versión, 8, 17
  beta, 13
  externa, 84
  interna, 65, 84
viabilidad, 14, 81, 305, 334
vista, 20, 57-58

W



The CRC Card Book

David Bellin y Susan Suchman Simone
Prólogos de Kent Beck y Ward Cunningham
Addison-Wesley Object Technology Series

Las tarjetas CRC ayudan a los equipos de proyectos a "representar" las diferentes partes del dominio de un problema. El desarrollador de aplicaciones puede utilizar esas tarjetas para definir las Clases, las Relaciones entre clases y la Colaboración entre esas clases (CRC) antes de comenzar el diseño orientado a objetos del programa de aplicación. Los estudios de casos en este libro se presentan enlazados en forma de novela para mostrar cómo entran en juego las personalidades y la cultura de la organización cuando se utiliza la técnica CRC. Expertos en C++, Java y Smalltalk proporcionan ejemplos de implementación en cada lenguaje.

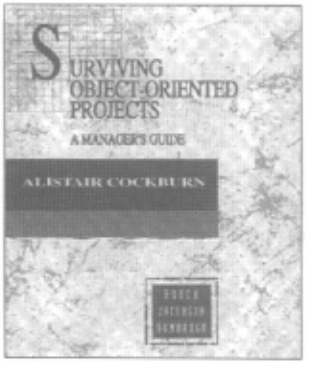
0-201-89535-8 • En rústica • 320 páginas • © 1997

Object Solutions

Managing the Object-Oriented Project
Grady Booch
Addison-Wesley Object Technology Series

Object Solutions es una consecuencia directa de la experiencia de Grady Booch con proyectos orientados a objetos en desarrollos en todo el mundo. Este libro se centra en el proceso de desarrollo, y es el recurso perfecto para los desarrolladores y los administradores que quieren implantar tecnologías de objetos por primera vez o afinar su práctica actual de desarrollo orientado a objetos. A partir de su conocimiento de las estrategias utilizadas tanto en proyectos con éxito como en otros fallidos, el autor ofrece sugerencias prácticas para aplicar tecnologías de objetos y controlar los proyectos de forma efectiva.

0-8053-0594-7 • En rústica • 336 páginas • © 1996

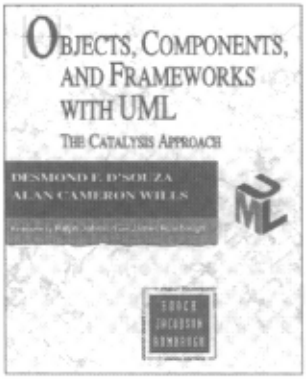


Surviving Object-Oriented Projects

A Manager's Guide
Alistair Cockburn
Addison-Wesley Object Technology Series

Este libro permite sobrevivir y, en última instancia, tener éxito con un proyecto orientado a objetos. Alistair Cockburn profundiza en su experiencia personal y su amplio conocimiento para proporcionar la información que los administradores necesitan para enfrentarse a los imprevistos retos que les esperan durante la implantación del proyecto. *Surviving Object-Oriented Projects* da soporte a sus ideas claves a través de pequeños casos de estudio extraídos de proyectos orientados a objetos reales. Un apéndice recoge esas guías y soluciones en forma de breves tablas, ideales para tener referencias a mano.

0-201-49834-0 • En rústica • 272 páginas • © 1998



Objects, Components, and Frameworks with UML

The Catalysis Approach

Desmond F. D'Souza and Alan Cameron Wills

Addison-Wesley Object Technology Series

Catalysis es un método basado en UML que está tomando auge rápidamente para el desarrollo de componentes y el desarrollo basado en *frameworks* con objetos. Los autores describen un enfoque único basado en UML para la especificación precisa de interfaces de componentes utilizando un modelo de tipos, permitiendo una descripción externa precisa del comportamiento sin restringir las implementaciones. Este enfoque proporciona a los desarrolladores de aplicaciones y a los arquitectos de sitemas unas técnicas bien definidas reutilizables que les ayudan a construir sistemas de objetos distribuidos y abiertos a partir de componentes y *frameworks*.

0-201-31012-0 • En rústica • 800 páginas • © 1999

Real-Time UML

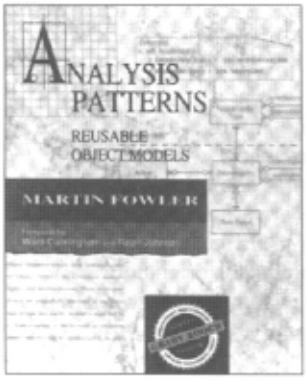
Developing Efficient Objects for Embedded Systems

Bruce Powel Douglass

Addison-Wesley Object Technology Series

El Lenguaje Unificado de Modelado está especialmente indicado para modelar sistemas de tiempo real y sistemas embebidos. *Real-Time UML* es la introducción que necesitan los desarrolladores de sistemas de tiempo real para hacer la transición hacia el análisis y el diseño orientados a objetos con UML. El libro cubre las características importantes de UML, y muestra cómo utilizar de forma efectiva esas características para modelar sistemas de tiempo real. También se incluyen discusiones especiales sobre las máquinas de estados finitos, estrategias de identificación de objetos y patrones de diseño de tiempo real, para ayudar a los desarrolladores principiantes y expertos.

0-201-32579-9 • En rústica • 400 páginas • © 1998



Analysis Patterns

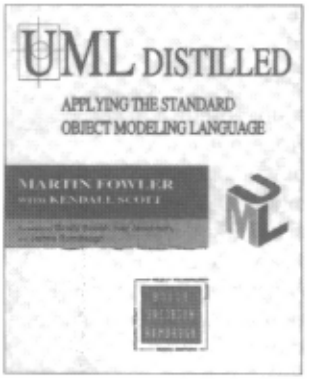
Reusable Object Models

Martin Fowler

Addison-Wesley Object Technology Series

Martin Fowler comparte la riqueza de su experiencia de modelado y su aguda vista para resolver problemas repetitivos y transformar las soluciones en modelos reutilizables. *Analysis Patterns* suministra un catálogo de patrones que han emergido en un amplio rango de dominios, incluyendo el comercio, las medidas, la contabilidad y las relaciones organizativas.

0-201-89542-0 • Pastas rígidas • 672 páginas • © 1997

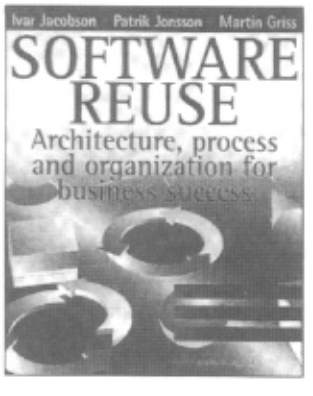


UML Distilled

Applying the Standard Object Modeling Language
Martin Fowler con Kendall Scott
Prólogo de Grady Booch, Ivar Jacobson y James Rumbaugh
Addison-Wesley Object Technology Series

Receptor del Productivity Award de 1997 de la revista Software Development, esta concisa visión global introduce al lector en el Lenguaje Unificado de Modelado, destacando los elementos claves de su notación, su semántica y sus procesos. Se incluye una breve explicación de la historia, el desarrollo y los fundamentos de UML, así como discusiones sobre cómo puede integrarse UML en el proceso de desarrollo orientado a objetos. Este libro también perfila varias técnicas de modelado asociadas a UML (casos de uso, tarjetas CRC, diseño por contrato, clasificación dinámica, interfaces y clases abstractas).

0-201-32563-2 • En rústica • 208 páginas • © 1997

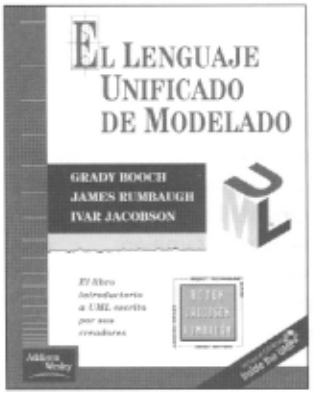


Software Reuse

Architecture, Process, and Organization for Business Success
Ivar Jacobson, Martin Griss y Patrik Jonsson
Addison-Wesley Object Technology Series

Este libro da un paso de gigante para acercar a los ingenieros de software, a los diseñadores, a los programadores y a sus jefes a un futuro en el que la norma es la ingeniería del software orientado a objetos basada en componentes. Jacobson, Griss y Jonsson desarrollan un modelo coherente y un conjunto de guías para asegurar el éxito con la reutilización orientada a objetos, sistemática y a gran escala. Su marco, denominado "Negocio de Ingeniería del Software Dirigida por la Reutilización" (Negocio de Reutilización) trata de forma sistemática las cuestiones claves de proceso de negocio, arquitectura y organización, que dificultan el éxito con la reutilización.

0-201-92476-5 • Pastas rígidas • 560 páginas • © 1997



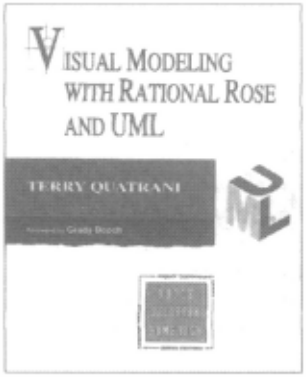
El Lenguaje Unificado de Modelado

Grady Booch, James Rumbaugh e Ivar Jacobson
Rational Software Corporation

Esta obra es la guía de usuario por excelencia del revolucionario Lenguaje Unificado de Modelado, UML, escrito por los creadores del lenguaje. Este libro presenta una descripción exhaustiva del lenguaje. Comienza desarrollando el modelo conceptual de UML y progresivamente, va aplicando UML a una serie de problemas cada vez más complejos y de diversos ámbitos. Este enfoque basado en los ejemplos ayuda al lector a entender y aplicar de forma rápida UML.

Se incluye el CD-ROM multimedia Inside the UML que proporciona una completa introducción al modelado visual con UML.

84-7829-028-1 • 448 páginas • © 1999



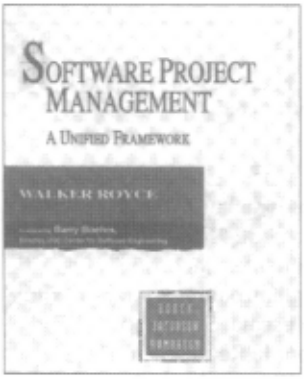
Visual Modeling with Rational Rose and UML

Terry Quatrani

Addison-Wesley Object Technology Series

Terry Quatrani, el apóstol de Rose en Rational Rose Corporation, enseña a hacer modelado visual con UML, facilitando la aplicación de un proceso iterativo e incremental para el análisis y el diseño. Con la orientación práctica que se ofrece en este libro, uno será capaz de especificar, visualizar, documentar y crear soluciones software. Los puntos más relevantes de este libro incluyen un examen del comportamiento de un sistema desde un enfoque basado en casos de uso; una discusión de los conceptos y las notaciones utilizadas para encontrar objetos y clases; una introducción a la notación necesaria para crear y documentar la arquitectura de un sistema; y una revisión del proceso de planificación de iteraciones.

0-201-31016-3 • En rústica • 240 páginas • © 1998



Software Project Management

A Unified Framework

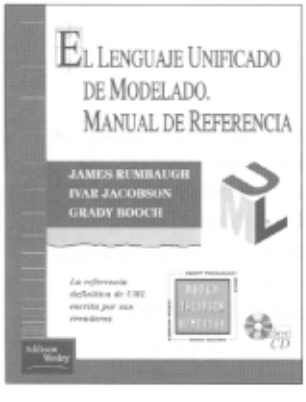
Walker Royce

Prólogo de Barry Boehm

Addison-Wesley Object Technology Series

Este libro presenta un nuevo marco de gestión, adaptado de forma única a las complejidades del desarrollo de software moderno. La perspectiva práctica de Walker Royce muestra los defectos de muchas prioridades de gestión bien aceptadas y proporciona a los profesionales del software un conocimiento actualizado derivado de sus veinte años de experiencia de gestión con éxito en el mundo real. En resumen, el libro proporciona a la industria del software unos puntos de referencia probados en la realidad para tomar decisiones tácticas y elecciones estratégicas que aumentarán las probabilidades de éxito de una organización.

0-201-30958-0 • Pastas rígidas • 448 páginas • © 1998



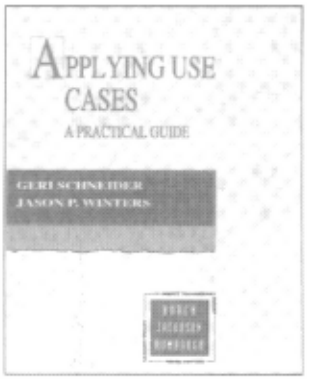
El Lenguaje Unificado de Modelado. Manual de referencia

James Rumbaugh, Ivar Jacobson y Grady Booch

Rational Software Corporation

James Rumbaugh, Ivar Jacobson y Grady Booch han creado la referencia definitiva de UML. Este libro a dos colores cubre todos los aspectos y detalles de UML y presenta el lenguaje de modelado en un formato de referencia útil que los arquitectos de software y los programadores profesionales deberían tener en su estantería. El libro se organiza por temas y ha sido diseñado para facilitar un acceso rápido. Los autores también proporcionan la información necesaria para facilitar la transición a UML de los usuarios existentes de las notaciones OMT, Booch y OOSE. El libro proporciona una visión global del fundamento semántico de UML a través de un conciso apéndice.

84-7829-037-0 • CD-ROM • 424 páginas • © 2000



Applying Use Cases

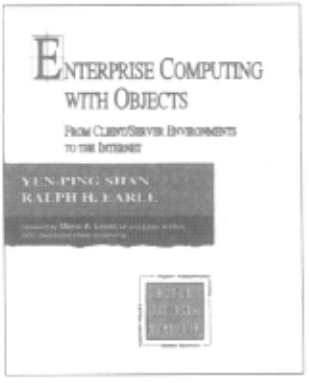
A Practical Guide

Geri Schneider y Jason P. Winters

Addison-Wesley Object Technology Series

Applying Use Cases proporciona una introducción clara y práctica al desarrollo de casos de uso, mostrando su uso a través de un continuo estudio de casos. Utilizando el Proceso Unificado de Desarrollo de Software como marco y el Lenguaje Unificado de Modelado como notación, los autores guían al lector a través de la aplicación de los casos de uso en diferentes fases del proceso, centrándose en dónde y cómo se aplican mejor los casos de uso. El libro también ofrece una comprensión de los errores y trampas más comunes que pueden aparecer en un proyecto orientado a objetos.

0-201-30981-5 • En rústica • 208 páginas • © 1998



Enterprise Computing with Objects

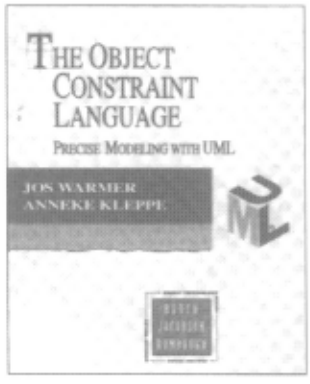
From Client/Server Environments to the Internet

Yen-Ping Shan y Ralph H. Earle

Addison-Wesley Object Technology Series

Este libro ayuda a situar las tecnologías rápidamente cambiantes (tales como Internet, la World Wide Web, la computación distribuida, la tecnología de objetos y los sistemas cliente/servidor) en sus contextos aproximados cuando uno se está preparando para el desarrollo, despliegue y mantenimiento de sistemas de información. Los autores distinguen lo esencial de lo circunstancial, mientras que ofrece una clara comprensión de cómo encajan juntas las tecnologías subyacentes. El libro examina temas esenciales, incluyendo persistencia de datos, seguridad, rendimiento, capacidad de crecimiento y herramientas de desarrollo.

0-201-32566-7 • En rústica • 448 páginas • © 1998



The Object Constraint Language

Precise Modeling with UML

Jos Warmer y Anneke Kleppe

Addison-Wesley Object Technology Series

El Lenguaje de Restricciones de Objetos (OCL) es un lenguaje notacional nuevo, un subconjunto del Lenguaje Unificado de Modelado, que permite a los desarrolladores de software expresar un conjunto de reglas que gobiernan aspectos muy específicos de un objeto en aplicaciones orientadas a objetos. Con OCL, los desarrolladores serán capaces de expresar más fácilmente limitaciones únicas y de escribir los detalles de bajo nivel que a menudo son necesarios en los diseños software complejos. El enfoque práctico del autor y el uso ilustrativo de ejemplos ayudará a los desarrolladores de aplicaciones a adquirir rápidamente presteza.

0-201-37940-6 • En rústica • 144 páginas • © 1999