PROGRAMACIÓN ORIENTADA A OBJETOS

CARLOS FONTELA Y NICOLÁS PAEZ

Tabla de contenido

Programación Orientada a Objetos

Este libro surgió a partir de la necesidad de contar con un material de apoyo para la materia Algoritmos y Programación 3 que dictamos en la Facultad de Ingeniería de la Universidad de Buenos Aires. Durante varios años utilizamos como apoyo diversos libros que había escrito Carlos. Sin embargo, a partir de 2009 comenzamos a incorporar algunas innovaciones en el dictado de la materia que fueron generando cierto desfasaje con el enfoque planteado en los mencionados libros.

Entre dichas innovaciones de enfoque destacamos principalmente 3:

- La presentación del paradigma de Programación Oriendada a Objetos (POO) comienza haciendo un importante foco en los objetos, dejando las clases como una posible forma de implementación.
- La separación inicial de los conceptos del paradigma y la posterior traslación de dichos conceptos a diversos lenguajes de programación.
- La inclusión de ejercicios prácticos para guiar al lector en el proceso de aprendizaje.

Audiencia

Si bien los alumnos de nuestra materia son la audiencia central de este libro, nos hemos propuesto ser más abarcativos y no limitarnos a esa audiencia. Creemos que el libro puede ser de utilidad en diversos contextos, aunque hay una premisa de la que partimos: el lector ya posee conocimientos de programación. Puede que sepa programar con algún lenguaje estructurado o funcional, o incluso puede que haya usado un lenguaje orientado a objetos. Pero en ninguno de esos casos ha profundizado conscientemente en los pilares del paradigma de la programación orientada a objetos.

Dinámica de escritura y publicación

Respecto de la dinámica de escritura del libro hemos decidido trabajar en un esquema iterativo incremental con publicaciones parciales. De esta manera, podremos utilizar el material de forma temprana, lo cual esperamos que también nos permita incorporar feedback antes de la publicación final y correspondiente impresión física.

Carlos Fontela y Nicolás Paez, Agosto 2015

Introducción

Una curiosidad: la gente y sus opiniones sobre POO

La gente que se dedica a la programación y al desarrollo de software tiene opiniones de lo más diversas sobre lo que caracteriza a la POO. A continuación, mostramos el resultado de una encuesta realizada en marzo y abril de 2015 sobre 250 respuestas obtenidas [1]. A modo de caracterización del universo consultado, informamos:

- El 100% trabaja en desarrollo de software, aunque el 5% tiene menos de 1 año de experiencia. Un 65% dice tener más de 5 años de experiencia y un 28% más de 10 años.
- En cuanto a edades, el grupo etario más numeroso (se dividió en grupos cada 10 años, con edades terminadas en 5) es el de 26 a 35 años, con el 67% del total. El grupo de 25 años o menos sumó un 19% y de 36 a 45, el 12%. De allí se deduce que muy poca gente de más de 46 años completó la encuesta [2].

Respecto de los lenguajes de programación que dicen utilizar, descartando los que suman menos del 10%, obtuvimos:

- SQL: 73%
- Java: 70%
- JavaScript: 55%
- C++: 30%
- C#: 30%
- PHP: 28%
- C: 27%
- Python: 24%
- Ruby: 10%

Uno de los lenguajes elegidos para este libro, Smalltalk, sólo es conocido por el 6% de los encuestados. Ahora bien, los resultados que nos interesa destacar de la encuesta son los conceptos que los encuestados consideraron “fundamentales” o “muy importantes” dentro de la programación orientada a objetos. De ellos, los 10 más elegidos, fueron, con sus porcentajes:

- Clase: 91%
- Objeto: 90%
- Encapsulamiento: 90%
- Polimorfismo: 88%
- Herencia: 87%
- Interfaces: 76%
- Clases abstractas: 71%
- Ocultamiento de implementación: 71%
- Métodos abstractos: 70%
- Sobrecarga de métodos: 65%

Para los autores, lo que caracteriza a la POO es que un programa es un conjunto de objetos enviándose mensajes y reaccionado ante los mismos, más la idea de que es responsabilidad de cada objeto saber cómo responder a esos mensajes, además del hecho de que cada objeto podría responder a los mensajes de manera distinta.

Por lo tanto, los conceptos centrales de POO para nosotros son: objetos, mensajes, encapsulamiento y polimorfismo. Luego hay conceptos muy interesantes, pero que no hacen a la orientación a objetos en sí misma, sino que son cuestiones de implementación o usos adquiridos de otros paradigmas. Curiosamente, el concepto de mensaje sólo es “fundamental” o “muy importante” para el 48% de los encuestados, mientras que el concepto más importante, con el 91% de los encuestados considerándolo “fundamental” o “muy importante” es el de clase, que es una cuestión de

implementación solamente [3]. Otra cosa que llama poderosamente la atención es la cantidad de conceptos que para nuestros encuestados son centrales en la orientación a objetos: hay casi dos tercios que creen que hay 10 conceptos centrales, lo cual parece mucho para cualquier paradigma.

Sólo estos resultados justifican la escritura de este libro…

Razones de un paradigma

Construcción reparando en la complejidad

Una de las preguntas que muchos nos hacemos al acercarnos a la POO es: ¿por qué necesito otro paradigma? La verdad es que casi cualquier programa se puede desarrollar en cualquier paradigma. Entonces, ¿para qué sirve el paradigma de objetos? Más allá de lo que se pretendió en sus orígenes, que está muy discutido, hoy el éxito del paradigma de objetos se debe a que permite un mejor manejo de la complejidad. En efecto, las aplicaciones que construimos son cada vez más complejas, y necesitamos tener herramientas – paradigmas – que nos permitan afrontar esa complejidad a un costo razonable. Ha habido muchos intentos de enfrentar la complejidad, casi siempre de la mano de la consigna “divide y vencerás”, que sugiere que un problema complejo se resuelve mejor atacándolo por partes.

El enfoque del paradigma orientado a objetos ha sido el de construir en base a componentes. Es decir, así como todas las industrias se dividen el trabajo y construyen en base a componentes ya fabricados (pensemos, por ejemplo, en la industria del automóvil, que para armar un vehículo le incorpora cubiertas, vidrios, equipos de audio, equipos de climatización, etc., fabricados por otras industrias), la industria del software ha descubierto que puede hacer lo mismo. De esa manera, cada uno hace lo que es propio del problema que está encarando, y simultáneamente le incorpora partes ya probadas y optimizadas, a costos y tiempos menores.

Por ejemplo, si necesitamos construir una aplicación de venta de productos en línea, podemos desarrollar una parte e incorporar y ensamblar otras, como el componente de autenticación del usuario, el carrito de compras, la validación del medio de pago, etc. Cada uno de estos componentes puede ser provisto por terceros, y nosotros simplemente incorporarlos en nuestro desarrollo. Los componentes de la POO son… sí, objetos.

Por supuesto, para construir usando componentes externos, debemos ponernos de acuerdo en la forma en que vamos a ensamblar dichos componentes. Eso requiere de dos condiciones fundamentales: encapsulamiento y contratos. Es decir, cada componente debe tener el comportamiento esperado, sin que nuestro desarrollo dependa de la manera en que está implementado (a eso llamamos encapsulamiento en POO), y además debemos conocer qué nos ofrece cada componente y cómo conectarnos con él (lo que definimos como contrato en el capítulo correspondiente). Esta conjunción de encapsulamiento y contratos es lo que llamaremos abstracción.

El caso del estudio: Sudoku

A lo largo del libro vamos a desarrollar un ejemplo basado en el juego de mesa de origen japonés llamado Sudoku. Este juego trabaja con un tablero de 9 filas por 9 columnas, conteniendo 9 “cajas” de 3 filas por 3 columnas. Como se muestra en la figura 0.1, al comienzo del juego hay algunos celdas del tablero ocupadas con números del 1 al 9. Notemos también en la figura la subdivisión del tablero en cajas, limitadas por líneas más gruesas que las que separan celdas.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

El objetivo del juego es ir llenando las celdas vacías con números del 1 al 9, de forma tal que en cada fila, en cada columna y en cada caja queden todos los números del 1 al 9. Como corolario, no puede haber ningún número que se repita en una misma fila, una misma columna o una misma caja.

La figura 1.2 muestra la solución al tablero de la figura 1.1

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Como ejemplo, vamos a ir desarrollando una aplicación que permita jugar al Sudoku, verificando que el usuario sólo pueda colocar números en los lugares permitidos, sin poner números en celdas ya ocupadas, y sin repetir un número en una misma fila, columna o caja.

Para entendernos mejor, vamos a mostrar cómo llamaremos a las distintas celdas, filas, columnas y cajas a lo largo del libro. Esto se muestra en las siguientes figuras:

1_1	1_2	1_3	1_4	1_5	1_6	1_7	1_8	1_9
2_1	2_2	2_3	2_4	2_5	2_6	2_7	2_8	2_9
3_1	3_2	3_3	3_4	3_5	3_6	3_7	3_8	3_9
4_1	4_2	4_3	4_4	4_5	4_6	4_7	4_8	4_9
5_1	5_2	5_3	5_4	5_5	5_6	5_7	5_8	5_9
6_1	6_2	6_3	6_4	6_5	6_6	6_7	6_8	6_9
7_1	7_2	7_3	7_4	7_5	7_6	7_7	7_8	7_9
8_1	8_2	8_3	8_4	8_5	8_6	8_7	8_8	8_9
9_1	9_2	9_3	9_4	9_5	9_6	9_7	9_8	9_9

fila 1
fila 2
fila 3
fila 4
fila 5
fila 6
fila 7
fila 8
fila 9

columna 1	columna 2	columna 3	columna 4	columna 5	columna 6	columna 7	columna 8	columna 9

caja superior izquierda (1)	caja superior media (2)	caja superior derecha (3)
caja media izquierda (4)	caja media media (5)	caja media derecha (6)
caja inferior izquierda (7)	caja inferior media (8)	caja inferior derecha (9)

[1] La hizo Carlos Fontela. No pretendemos que esta encuesta sea estadísticamente incuestionable. Fue publicitada por las redes sociales Twitter, Facebook y LinkedIn, además de enviarse por correo electrónico a grupos informáticos diversos, con la aclaración de que se podía difundir libremente. Por todo esto, puede estar sesgada por las opiniones de gente más cercana a los autores. Se tomaron las primeras 250 respuestas, recibidas en una semana aproximadamente.

[2] Siendo los autores profesores universitarios, es probable que los grupos de estudiantes expliquen este sesgo.

[3] En defensa de nuestros encuestados, digamos que casi todos los lenguajes orientados a objetos implementan el comportamiento de los objetos en base a clases. Lo curioso sigue siendo que haya un 55% de los encuestados que trabajan en JavaScript y que no hayan notado que JavaScript es orientado a objetos, pero no tiene clases. Algo parecido podría decirse de la relación de JavaScript con la herencia, que el 87% de los encuestados considera tan importante.

Sobre los lenguajes de programación

La gran mayoría de los libros de POO utilizan un único lenguaje de programación. Al mismo tiempo es común que los lenguajes que soportan POO ofrezcan características que exceden (y en algunos casos incluso contradicen) el paradigma OO. Creemos esta situación puede generar confusión en el lector llevando a que confunda particularidades de un lenguaje con elementos del paradigma de POO.

Esto por esto que en esta publicación hemos decido trabajar con más de un lenguaje. En concreto, trabajamos principal e indistintamente con dos lenguajes: Smalltalk y Java. Cada uno de estos lenguajes implementa el paradigma de POO de una forma distinta, lo cual creemos es beneficioso para ayudar a que el lector comprenda mejor los conceptos del paradigma y conozca más de una forma de implementación de los mismos. Adicionalmente para explicar algunas cuestiones puntuales utilizamos también Javascript.

En las siguientes secciones ofrecemos una breve introducción sobre las herramientas/ambientes de programación para estos lenguajes.

Java

Java fue creado en los años 90' y desde entonces su popularidad ha ido en ascenso.

Para programar con Java es necesario instalar el Java Development Kit (JDK). Existen diversas implementaciones del JDK siendo las más populares en la actualidad la de Oracle y la Open Source (Open JDK). Para los fines de este libro es indistinto en uso de cualquiera de las dos. El JDK incluye entre otros cosas el entorno de ejecución de Java (Java Runtime Environment, JRE) necesario para ejecutar aplicaciones Java y también un conjunto de herramientas para construir aplicaciones Java, entre las que se encuentra el compilador de Java (javac). Este libro está basado en la versión 1.7 de Java, conocida comúnmente como Java 7.

Al trabajar con Java, uno escribe su código en archivos de texto con extensión .java. Luego esos archivos con compilados (con el compilador javac) para generar un archivo binario de extensión .class. Dicho archivo es ejecutado por el entorno de ejecución de Java (JRE). Esta arquitectura permite que el mismo binario pueda ser portable entre distintos sistemas operativos, siempre que los mismos cuenten con un entorno de ejecución de Java.

En este sentido el trabajo con Java es muy similar al trabajo con C, en el sentido que existe código fuente, un compilador y código binario. La diferencia radica en que en el caso de C el binario no es potable entre sistemas operativos, mientras que en Java si lo es.

El siguiente fragmento de código muestra el tradicional "hola mundo".

```
// HolaMundo.java

class HolaMundo {

    public void saludar() {
        System.out.println("Hola mundo");
    }

    public static void main(String args[]) {
        HolaMundo unHolaMundo = new HolaMundo();
        unHolaMundo.saludar();
    }

}
```

Para poder ejecutar esta aplicación debemos compilarla primero, para ello ejecutamos:

```
javac HolaMundo.java
```

Como resultado, se genera como salida un archivo llamado *HolaMundo.class*, el cual podemos ejecutar de la siguiente manera:

```
java HolaMundo
```

Smalltalk

Smalltalk data de los años 70' y fue uno de los primeros lenguajes con soporte para POO. Desde aquella época han existido diversas implementaciones del lenguaje, siendo la gran mayoría compatibles con el dialecto conocido como Smalltalk-80.

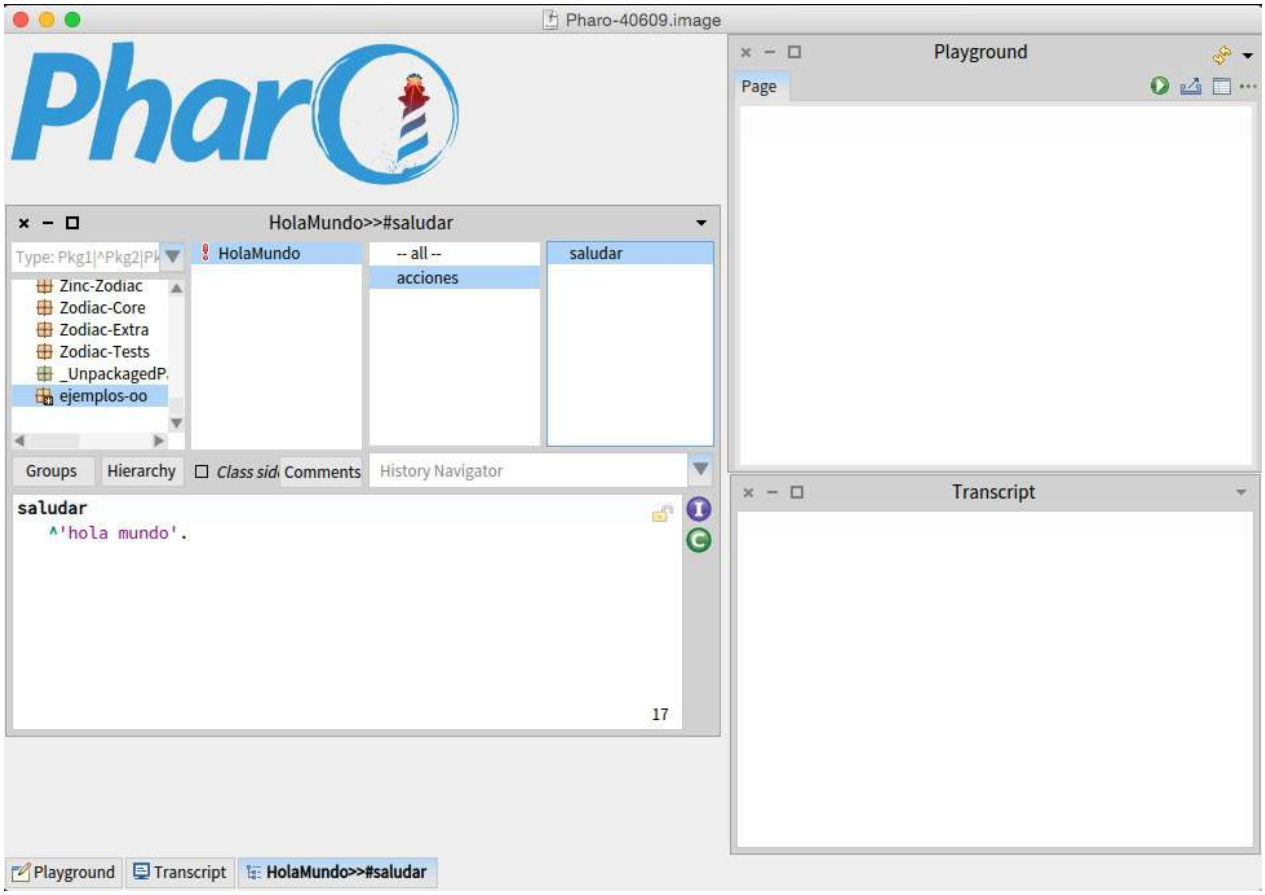
El trabajo con Smalltalk es radicalmente distinto al trabajo con Java, donde el programador trabaja sobre archivos de texto. En Smalltalk uno trabaja sobre una imagen de memoria que se almacena en disco y que es ejecutada por un máquina virtual. O sea que al trabajar en Smalltalk tendremos dos componentes: una imagen de memoria y una máquina virtual que tomará la imagen de un archivo del disco y la ejecutará para levantarla en memoria. Asimismo, dentro de la imagen el programador encontrará un conjunto de herramientas para crear sus aplicaciones.

En nuestro caso, trabajaremos con la implementación de Smalltalk llamada Pharo. La misma es de código abierto y está disponible en pharo.org.

Dentro de todo ambiente Smalltalk existen un conjunto de herramientas/objetos característicos:

- El Workspace: es un objeto donde uno uno puede escribir código Smalltalk y ejecutarlo. En la versión 4 de Pharo el worspace es llamado Playground
- El Transcript: es equivalente Smalltalk a la salida estándar. Es un objeto en el que podemos escribir mensajes.
- El Browser: es el código que nos permite navegar el sistema y escribir nuestro código. El browser de Pharo en la versión 4 se llama Nautilus.

La siguiente figura muestra los mencionados objetos en Pharo:



Algunos otros objetos comunes en los ambientes Smalltalk, y que veremos a lo largo del libro son: el Inspector, el Debugger y el Test Runner.

Dado que Pharo es un ambiente interactivo, describir en texto los pasos para hacer un aplicación podría resultar extremadamente tedioso. Por eso invitamos al lector a ver los videos del canal de Youtube de nuestra materia para entender mejor la forma de trabajar en el ambiente Pharo.

Compartimos a continuación el código del clásico "HolaMundo "hola mundo" en Smalltalk:

```
Object subclass: #HolaMundo
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'ejemplos-oo'!

!HolaMundo methodsFor: 'as acciones' !
saludar
    Transcript show: 'hola mundo'! !
```

Un mundo de objetos

Nuestra primera aproximación a la solución de un problema

La POO plantea que, para resolver un problema, en primer lugar debemos encontrar entidades del dominio del problema, que van a ser nuestros “objetos”. Como segundo paso, debemos hallar cómo interactúan esas entidades para resolver el problema: decimos que buscamos los “mensajes” que los objetos se envían, en qué orden y bajo qué condiciones lo hacen. En tercer lugar, deberíamos poder determinar cómo hacen esos objetos para responder a los mensajes: lo que llamamos “comportamiento” de los objetos.

Nota: si el lector conoce de programación estructurada por refinamientos sucesivos, notemos que el enfoque de la POO es diferente. Tanto la programación estructurada como la POO atacan el problema por partes. Pero mientras la programación estructurada busca las partes en la propia solución del problema (las partes – funciones, procedimientos o como se las llame – son acciones que el programa debe realizar), la POO busca las partes en las entidades que surgen del dominio del problema en sí mismo. De a poco iremos incorporando esta forma de trabajar, así que sigamos leyendo…

Por lo tanto, lo que debemos hacer son 3 pasos básicos:

- Encontrar objetos
- Determinar cómo deben interactuar los objetos
- Implementar el comportamiento de los objetos

El problema que vamos a encarar para resolver mediante estos tres pasos es el de ver si podemos colocar un número en una celda en un juego de Sudoku que debamos programar. Vamos a resolver el problema – al menos al comienzo – sin usar ningún lenguaje de programación, de modo tal de enfocarnos en el paradigma y no en cuestiones de implementación. Como herramienta de notación, vamos a usar diagramas UML.

Paso 1: encontrar objetos

Lo primero que deberíamos hacer es buscar entidades del dominio. Una buena idea para ello, al menos en problemas sencillos, es empezar por los sustantivos que surgen del propio enunciado del problema. En el caso del Sudoku, si lo que queremos es establecer si el número 7 se puede colocar en la celda ubicada en la fila 2 y la columna 3, los objetos candidatos a participar en este escenario son:

- El tablero del Sudoku
- El número 7
- La celda 2 3
- La fila 2
- La columna 3
- La caja que contiene la celda 2 3

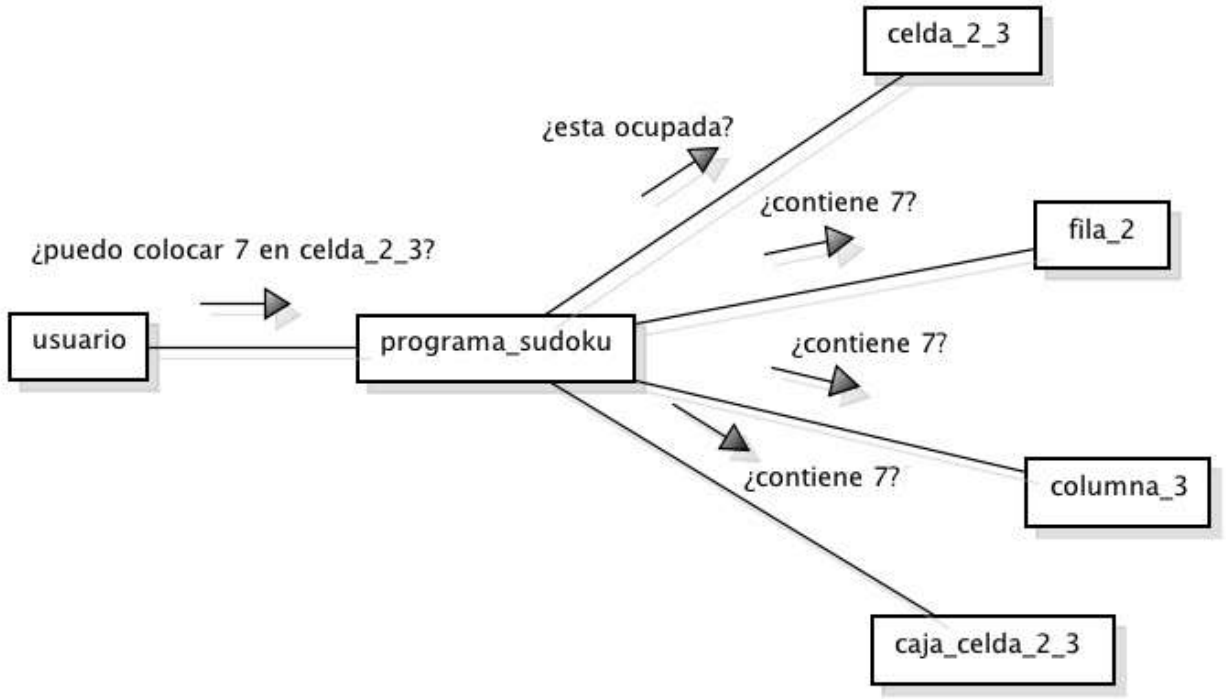
Bueno, con esto ya tenemos los objetos, o al menos los que descubrimos por el momento. Puede ser que más adelante descubramos que necesitamos más objetos, pero por el momento trabajemos con estos siete.

Paso 2: resolver cómo deben interactuar los objetos

Para ver si un número (por ejemplo, el 7) puede ser colocado en una celda (por ejemplo, la celda de fila 2 y columna 3), el procedimiento podría ser:

- Ver si la celda_2_3 si está ocupada
- Ver si la fila en la que está la celda en cuestión (fila_2) contiene el número (7 si este caso)
- Ver si la columna en la que está la celda en cuestión (columna_3) contiene el número (7 si este caso)
- Ver si la caja en la que está la celda en cuestión (caja_celda_2_3) contiene el número (7 si este caso)
- Si ni fila, ni la columna, ni la caja contienen el número en cuestión, entonces podemos colocarlo en la celda. Si no, no.

Un diagrama que represente esto podría ser el de la figura 3.1:



Analicemos este diagrama. Allí se muestran 6 objetos, que se indican mediante rectángulos, y hemos llamado “usuario”, “programa_sudoku”, “celda_2_3”, “fila_2”, “columna_3” y “caja_celda_2_3”. El primero representa a un usuario de la aplicación, el segundo a la propia aplicación, el tercero a la celda en la cual quiero colocar el número, y los últimos tres, a la fila, la columna y la caja que corresponden a dicha celda. Ahora bien: en el diagrama figuran objetos que no habíamos encontrado antes. Uno de ellos es “usuario”. Efectivamente, para poder desarrollar un programa, debemos pensar en que habrá un usuario que interactuará con la aplicación. Así que, si bien no es una entidad del dominio del problema del Sudoku, sí lo es del programa que resuelve el Sudoku. Lo mismo pasa con el objeto “Programa Sudoku”, que necesitamos para ver cómo interactúa el usuario con el mismo. Por otro lado, ha desaparecido otro objeto: se trata del objeto “tablero”, que nos ocurrió que no encontramos que fuera necesario en este escenario. Otro objeto que no aparece en el escenario es el número 7. ¿O sí? Lo que ocurre con este objeto es que no está participando como entidad activa en el escenario, por lo cual no aparece en ningún rectángulo de los que representan objetos. Pero sigue siendo un objeto, que va como argumento de los distintos mensajes. ¿Qué hemos hecho hasta ahora? Los objetos que encontramos representan entidades del dominio del problema. Son estos objetos los que nos van a ayudar a resolver el problema. La resolución del problema la hacemos planteando que el objeto “usuario” le envíe un mensaje al programa. El mensaje “¿puedo colocar 7 en celda_2_3?”, pretende significar el envío de una consulta del usuario al programa, en el cual el primero pregunta al segundo si puede o no colocar el número 7 en la celda en cuestión. El objeto “Programa Sudoku”, para resolver el problema, envía cuatro mensajes a cuatro objetos diferentes, uno para ver si la celda está libre, y otros tres para saber si el número 7 ya se encuentra en la fila, columna o caja correspondiente. Una vez que tenga las respuestas a sus consultas, el objeto “programa_sudoku” puede responderle al objeto “usuario” si se puede o no colocar el número 7 allí.

Definición: objeto (preliminar) Un objeto es una entidad que existe en tiempo de ejecución y que puede enviar y recibir mensajes. Este mecanismo por el cual el objeto “programa_sudoku”, necesita de la colaboración de otros objetos para poder responder, se denomina delegación. Decimos que “programa_sudoku” delega, mediante el envío de mensajes, en

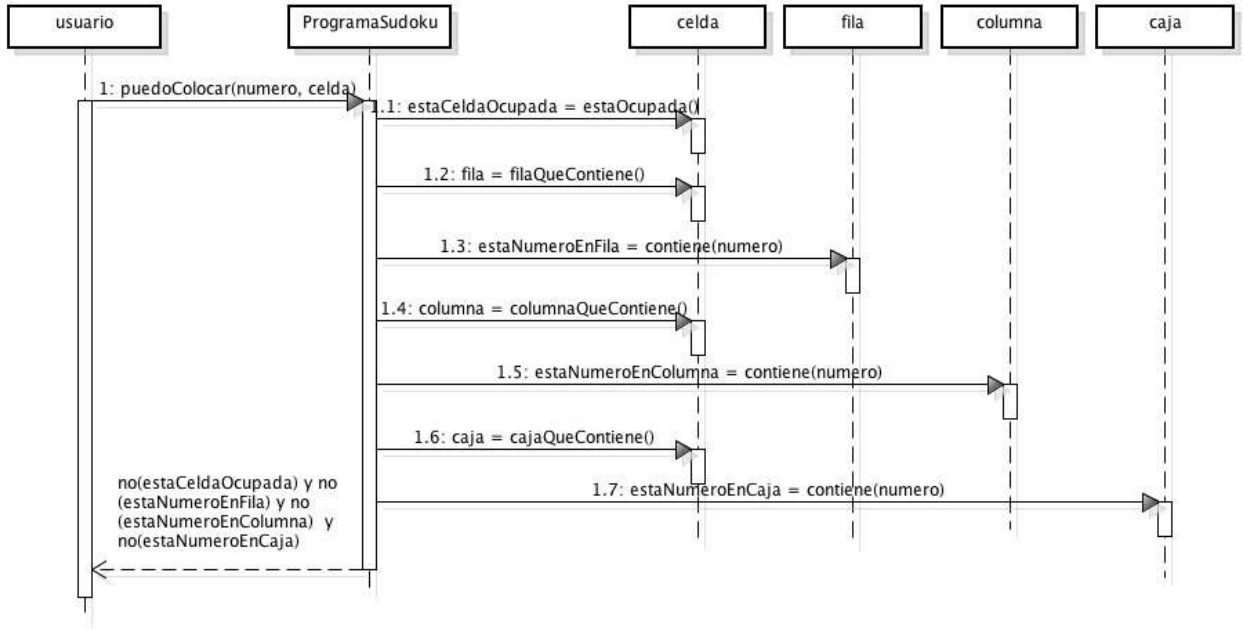
los objetos “celda_2_3”, “fila_2”, “columna_3” y “caja_celda_2_3”.

Definición: delegación Cuando un objeto, para responder un mensaje, envía mensajes a otros objetos, decimos que delega ese comportamiento en otros objetos.

Definición: comportamiento Las posibles respuestas a los mensajes recibidos por un objeto se denominan comportamiento.

Ahora bien, ¿cómo sabemos a priori cuáles son la columna, la fila y la caja a chequear? Podríamos hacer que sea la propia celda la que nos diga en cuál columna, fila y caja se encuentra. Vamos a introducir esto en nuestro diagrama. Pero antes, una observación. El diagrama anterior no dice nada del orden en que debo enviar los mensajes. Parece bastante obvio que el orden no importa: una vez que programa_sudoku recibe el mensaje del usuario, podría haber enviado los mensajes delegados en cualquier orden. Sin embargo, en la segunda versión, al agregar las preguntas a la celda, para saber en qué columna, fila y caja se encuentra, el orden de los mensajes sí importa, así que vamos a indicarlo explícitamente usando un diagrama que deje esto más en claro.

Eso se muestra en el diagrama de la figura 3.2, que se denomina un diagrama de secuencia. En él, el paso del tiempo se indica por el desplazamiento vertical (lo de más arriba ocurre antes, lo de abajo, después), y también por los números que indican precedencia y delegación entre mensajes.



En el diagrama, hemos aprovechado para generalizar la determinación de si se puede colocar un número (no necesariamente sólo el 7) en una celda cualquiera. También trabajamos con nombres más afines a la programación: no hay tildes, letras latinas, espacios en blanco, signos de pregunta, todos elementos que no suelen llevarse bien con los lenguajes de programación. El símbolo = implica asignación y el & la conjunción lógica. Veamos qué dice el diagrama.

Lo primero es que el usuario le envía el mensaje *puedoColocar* al ProgramaSudoku, enviándole como argumentos el número a chequear y la celda en la cual desea colocarlo. Al final de todo, la línea punteada del diagrama que vuelve, indica que el ProgramaSudoku le responde al usuario con la conjunción lógica de un conjunto de variables que obtuvo en los pasos anteriores.

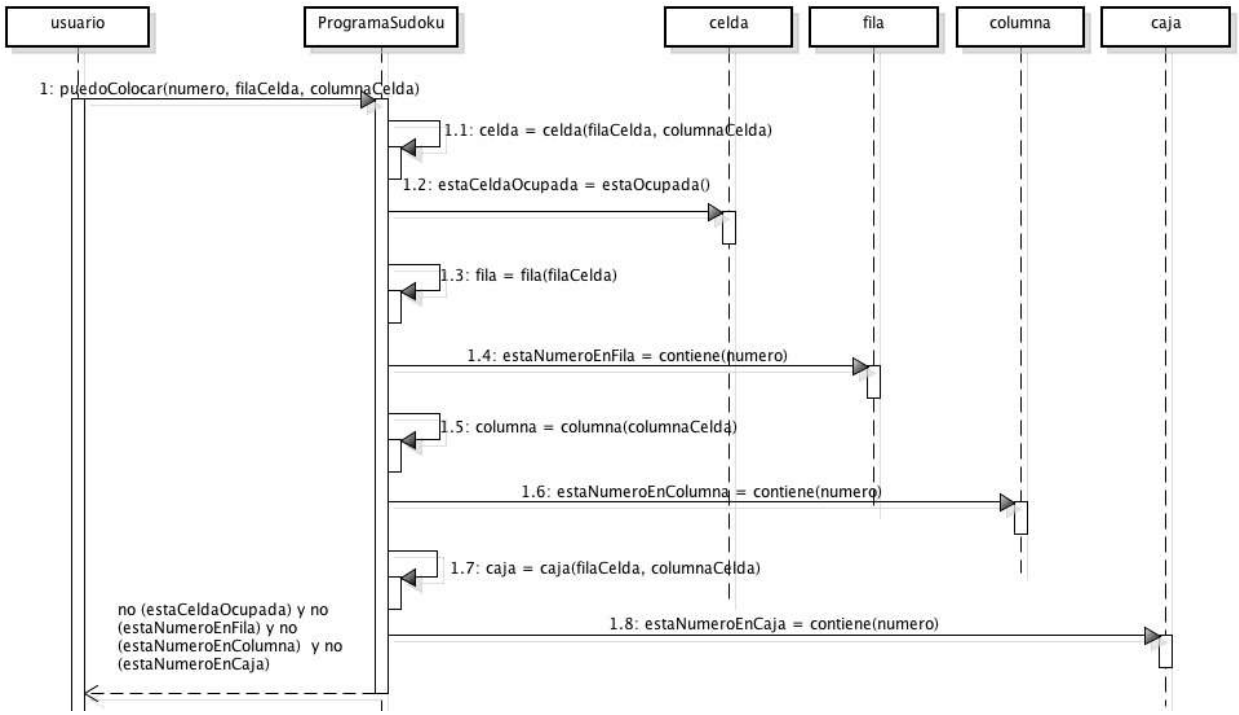
En el medio, ProgramaSudoku va enviando los siguientes mensajes, en forma secuencial:

- estaOcupada, enviado a la celda: esto sirve para que la celda le indique si está ocupada o no, y le devuelve el resultado almacenado en la variable *estaCeldaOcupada*.
- filaQueContiene, enviado a la celda: esto sirve para que la celda le indique en qué fila está, y le devuelve el resultado como una referencia a un objeto almacenado en la variable *fila*.
- contiene, con el número a verificar como argumento, enviado a la fila recientemente obtenida: esto sirve para que la fila chequee si el número en cuestión ya se encuentra en esa fila, devolviendo el resultado booleano en la variable *estaNumeroEnFila*.
- columnaQueContiene, enviado a la celda: esto sirve para que la celda le indique en qué columna está, y le devuelve el resultado como una referencia a un objeto almacenado en la variable *columna*.
- contiene, con el número a verificar como argumento, enviado a la columna recientemente obtenida: esto sirve para que la columna chequee si el número en cuestión ya se encuentra en esa columna, devolviendo el resultado en la variable *estaNumeroEnColumna*.
- cajaQueContiene, enviado a la celda: esto sirve para que la celda le indique en qué caja está, y le devuelve el resultado como una referencia a un objeto almacenado en la variable *caja*.
- contiene, con el número a verificar como argumento, enviado a la caja recientemente obtenida: esto sirve para que la caja chequee si el número en cuestión ya se encuentra en esa caja, devolviendo el resultado en la variable *estaNumeroEnCajaDisponible*.
- Finalmente, como dijimos, el resultado enviado al usuario es una conjunción lógica de lo que obtuvo en las variables *estaCeldaOcupada*, *estaNumeroEnFila*, *estaNumeroEnColumna* y *estaNumeroEnCaja*.

Tengamos en cuenta, no obstante, que no todas las precedencias que hemos introducido en el diagrama de secuencia son necesarias. Por ejemplo, el problema estaría bien resuelto también si primero pregunto a una celda en qué fila, columna y celda se encuentra, y recién después trabajo preguntándole a cada una si el número está o no.

Ahora bien, ¿tiene sentido que el usuario de la aplicación pase un objeto celda al programa? ¿De dónde obtiene el usuario el objeto celda? Para un usuario corriente, lo que intenta es colocar un número en el tablero, en una celda identificada por su posición. Entonces más que puedoColocar(numero, celda), la invocación debería ser puedoColocar(numero, filaCelda, columnaCelda), donde filaCelda y columnaCelda son dos simples números, como en el ejemplo: puedoColocar(7, 2, 3).

En el diagrama de la figura 3.3, introdujimos el cambio en el mensaje puedoColocar.



Paso 3: implementar el comportamiento de los objetos

Si vamos a programar esto, deberíamos poder asegurar que los objetos que reciben los mensajes van a poder entenderlos y hacer algo con ellos. Por ejemplo, si queremos que una celda nos pueda responder si está libre, vamos a tener que implementar a más bajo nivel esa celda. En términos de POO decimos que necesitamos implementar un método para la celda cuyo nombre es estaLibre.

Definición: método. Llamamos método a la implementación de la respuesta de un objeto a un mensaje. En términos de implementación, se asemeja a funciones o procedimientos de programación en otros paradigmas.

Una posibilidad de implementación sería que la celda contenga un valor entero con el número que la ocupa, dejando el 0 para cuando la celda está vacía. En ese caso, la pregunta sobre si la celda está libre o no lo está, se respondería viendo si contiene o no un 0. Pero eso son cuestiones de implementación que, como el lector estará sospechando, no estamos urgidos de resolver todavía. Por el momento vamos a saltear esta parte de la solución, pero tengamos en cuenta que lo que nos falta es implementar los métodos (usamos la notación objeto >> método):

- programaSudoku >> celda
- celda >> estaOcupada
- programaSudoku >> fila
- fila >> contiene
- programaSudoku >> columna
- columna >> contiene
- programaSudoku >> caja
- caja >> contiene

Pero todo esto, depende del lenguaje y de la forma en que éste implemente la POO, lo dejaremos para más adelante.

Conceptualizando

El comportamiento como aspecto central

Luego de lo que vimos, podemos reformular nuestra definición de objeto:

Definición: objeto (preliminar) Un objeto es una entidad que existe en tiempo de ejecución y que tiene comportamiento.

Es que el comportamiento de los objetos es la diferencia más importante con la programación estructurada tradicional, en la que trabajamos con variables y tipos simples, más algunos datos estructurados. Un objeto es mucho más que un dato estructurado: es, ante todo, una entidad con comportamiento, que suele guardar su estado en variables internas, pero sólo como soporte a ese comportamiento.

El mismo hecho del comportamiento como aspecto central nos va a llevar, apenas un poco más adelante, a la noción necesaria de encapsulamiento.

Los objetos tienen identidad, estado y comportamiento

Todo objeto tiene tres características: identidad, estado y comportamiento.

La **identidad** es lo que distingue a un objeto de otro. Sin importar lo parecidas que sean en cuanto a comportamiento, la celda 2 3 y la celda 5 7 del Sudoku son dos objetos diferentes, y así debemos considerarlos nosotros. La identidad de un

objeto lo acompaña desde su creación hasta su muerte.

El **estado** tiene que ver con algo que cambia a través del tiempo. Por ejemplo, la celda 2 3 de nuestro programa de Sudoku se encuentra libre al inicio del juego. A medida que éste avanza puede cambiar esta situación, al punto que al final, como vimos en la figura 1.2, queda allí el número 2. Decimos que el objeto celda 2 3 ha cambiado de estado, pasando de no tener un valor a tener el valor 2. Habitualmente, para cambiar el estado de un objeto, tiene que ocurrir algo que provoque ese cambio. Ese “algo” suele ser un mensaje recibido por el objeto. Aquí nos encontramos, entonces, con el primer vínculo entre estado y comportamiento: uno de los comportamientos posibles de un objeto es el cambiar de estado al recibir un mensaje. Además, el estado no tiene necesariamente que ser conocido desde fuera del objeto. Hay ocasiones en que el estado en que está un objeto es relevante para otros objetos del sistema, y ocasiones en que no, en las que solamente sirve como soporte interno para brindar determinado comportamiento. Ya volveremos sobre esto al hablar de encapsulamiento. Pero lo que nos importa resaltar acá es otro vínculo importante entre comportamiento y estado: hay ocasiones en las que el objeto exhibe su estado a través del comportamiento. Por ejemplo, en nuestro programa de Sudoku muy probablemente necesitemos preguntarle a una celda si contiene o no un número. Si bien éste es un estado del objeto, recurriremos a un mensaje (comportamiento) para que el objeto nos dé información sobre el estado. Por todo lo dicho, lo habitual es que el estado de un objeto se considere como algo privado, que no deba ser conocido desde afuera, salvo que ése conocimiento sea necesario, en cuyo caso accederemos a él mediante mensajes (comportamiento).

Del **comportamiento** ya hemos hablado: es el conjunto de posibles respuestas de un objeto ante los mensajes que recibe. Ahora bien, luego de todo lo dicho, podemos extendernos un poco más. El comportamiento de un objeto está compuesto por las respuestas a los mensajes que recibe un objeto, que a su vez pueden provocar:

- Un cambio de estado en el objeto receptor del mensaje.
- La devolución del estado de un objeto, en su totalidad o parcialmente.
- El envío de un mensaje desde el objeto receptor a otro objeto (delegación).

Otra manera de definir esta interacción entre comportamiento y estado es decir que un objeto tiene “responsabilidades”. Llamamos responsabilidades de un objeto a la conjunción del comportamiento activo (actuar ante la llegada de mensajes) y el mantenimiento de un estado interno.

Encapsulamiento: no importa el cómo sino el qué

Venimos hablando de comportamiento y estado hace unas páginas. Es más, hemos definido que el comportamiento es el aspecto central de la POO, pero a la vez dijimos que los objetos suelen tener estado interno, que sirve para dar soporte a ese comportamiento, el cual a veces es accedido mediante mensajes (más comportamiento). En definitiva, como programadores clientes de un objeto, esperamos que éste exhiba cierto comportamiento, a modo de un servicio que nos brinda el propio objeto. También esperamos que ese servicio sea brindado, en ocasiones, recurriendo al estado, o incluso cambiando el estado. Pero lo que no es esperable es que quien solicita el servicio deba saber cómo lo hace el objeto.

Definición: encapsulamiento Cada objeto es responsable de responder a los mensajes que recibe, sin que quien le envía el mensaje tenga que saber cómo lo hace. Esto es lo que llamamos encapsulamiento.

Las razones de ser del encapsulamiento son varias, entre ellas:

- Puede haber implementaciones alternativas para una misma operación.
- En el futuro, podemos cambiar una implementación por otra, ambas correctas, sin afectar al cliente que utiliza el servicio.

Un corolario del encapsulamiento es el principio de diseño de software OO conocido como “Tell, don’t ask”, y que implica que los objetos deben manejar su propio comportamiento, sin que nosotros manipulemos su estado desde afuera.

Por ejemplo, si en nuestro programa de Sudoku implementásemos la celda haciendo que en una celda desocupada haya un 0, podríamos sentirnos tentados de preguntar si está libre, haciendo:

```
celda >> contiene (0)
```

Pero eso violaría el encapsulamiento, ya que el cliente debería conocer cuestiones de implementación interna del objeto. Lo más razonable sería hacer la misma consulta, así:

```
celda >> estaLibre
```

Y en ese caso, la implementación interna sería desconocida para el cliente, posibilitando elegir varias alternativas e incluso cambiarla en el futuro.

Polimorfismo: cada objeto responde a los mensajes a su manera

Relacionado con lo anterior, notemos que a tres objetos distintos le estamos pasando el mensaje esta: se trata de los objetos fila, columna y caja. Más allá de que se trata del mismo mensaje, no sabemos si necesariamente va a ser implementado de la misma manera para cada objeto. Esta posibilidad de que distintos objetos entiendan el mismo mensaje, pero la respuesta al mismo pueda variar según de qué objeto se trate, se llama polimorfismo. Es un tema que trataremos en detalle más adelante, por ser un concepto central de la POO, y le dedicaremos un capítulo completo. Por eso, por ahora lo dejamos y nos quedamos con esta definición preliminar:

Definición: polimorfismo (preliminar) El polimorfismo es la capacidad que tienen distintos objetos de responder de diferentes maneras a un mismo mensaje.

Algunas cuestiones de implementación

Tipología de los lenguajes de programación orientados a objetos

Hay muchos lenguajes de POO, y los mismos han elegido distintos caminos de implementación. También hay tantas clasificaciones como autores. Sin ánimo de introducir una nueva clasificación definitiva, nos interesa aquí destacar que, en cuanto a la manera de implementar los conceptos vistos hasta aquí, los autores distinguimos tres cuestiones:

- Respecto de cómo implementan el comportamiento: lenguajes basados en clases y lenguajes basados en prototipos o ejemplos.
- Respecto de la implementación de la creación de objetos: en tiempo de ejecución en el área de memoria dinámica o en tiempo de compilación en la pila.
- Respecto de la verificación de tipos: con verificación en tiempo de compilación o con verificación en tiempo de ejecución.

Vamos entonces a analizar los dos primeros aspectos (el tercero queda para más adelante), usando tres lenguajes para ello: Smalltalk, Java y JavaScript.

- Smalltalk es un lenguaje basado en clases, que crea los objetos en el área de memoria dinámica en tiempo de ejecución. Si bien es un lenguaje en el que los programas se compilan, al menos en la mayor parte de sus implementaciones, sólo hace chequeo de tipos en tiempo de ejecución.
- Java es también un lenguaje basado en clases, que crea los objetos en el área de memoria dinámica en tiempo de ejecución. En cuanto al chequeo de tipos, se hace casi en su totalidad en tiempo de compilación.

- Javascript es un lenguaje basado en prototipos, que crea los objetos en el área de memoria dinámica en tiempo de ejecución. Es un lenguaje interpretado y sin tipos, por lo que no hay ningún chequeo de tipos, ni en tiempo de compilación – que no existe – ni en tiempo de ejecución.

Para ver la sintaxis de estos lenguajes se recomienda recurrir a los apéndices.

Implementación del comportamiento en los lenguajes basados en clases

La mayor parte de los lenguajes agrupan los objetos en clases, siendo éstas conjuntos de objetos que, por lo menos, tienen el mismo comportamiento (entienden los mismos mensajes y responden a ellos de la misma manera). En nuestro caso de estudio podemos pensar en 6 clases, entre otras: Usuario, Programa, Celda, Columna, Fila y Caja. Todos los objetos de un programa son instancias de clases. Por ejemplo, el objeto celda 2 3 y el objeto celda 2 6, son ambos instancias de la clase Celda. Y dado que son instancias de la clase Celda, entienden los mismos mensajes, que por el momento es estaLibre. Además, como veremos, responden de la misma manera: si no fuese así, no diríamos que pertenecen a la misma clase. La clase es el tipo del objeto, en el mismo sentido en que, en el reino animal, la especie es el tipo de individuo. Entonces decimos que celda_2_3 es instancia de la clase Celda porque Celda es el tipo del objeto celda_2_3. De la misma manera, diríamos que Lassie es una instancia de la especie Perro porque Perro es el tipo de animal que es Lassie. Diríamos que Perro es un concepto, mientras que Lassie es un perro concreto, con existencia real. De la misma manera, celda_2_3 es una celda concreta con la que trabajamos en nuestro programa, mientras que la clase Celda es más bien un concepto que engloba a todas las celdas posibles. Lo interesante de la POO, cuando está implementada con clases, es que esas clases pueden ser definidas por el programador, y es en esa definición que el programador define el comportamiento posible de los objetos que son instancias de esas clases.

Por ejemplo, en Smalltalk, la implementación del método estaLibre será algo como:

```
estaLibre
    ^ ( contenido = 0 )
```

Y en Java podría ser:

```
public boolean estaLibre () {
    return contenido == 0;
}
```

Más allá de las cuestiones de implementación, lo que esto significa es que podemos definir cómo debe responder una celda cualquiera al mensaje estaLibre, y esa respuesta es única para cualquier objeto instancia de la clase Celda. Otro corolario interesante, que ya habíamos mencionado antes, es la interacción entre estado y comportamiento o, dicho de otra manera, la existencia del estado como soporte del comportamiento.

Como vemos del código anterior, hay una variable llamada contenido, que permite almacenar estado dentro de una instancia de Celda, y que a la vez nos permite implementar la condición de celda libre, de modo tal de brindar soporte a la respuesta del mensaje estaLibre. Esta variable, generalmente denominada atributo, variable de instancia o propiedad, es parte del estado interno del objeto. En los lenguajes basados en clases, estos atributos se definen en la clase y existen en cada una de las instancias de la misma.

Definición: atributo En POO, llamamos atributo a una variable interna del objeto que sirve para almacenar parte del estado del mismo.

Creación de objetos en los lenguajes basados en clases: instanciación

En los lenguajes basados en clases, las clases sirven también como molde de creación de objetos. Por ejemplo, en Smalltalk, la creación del objeto que representa la celda_2_3 se haría enviando el mensaje new a la clase Celda:

```
celda_2_3 := Celda new.
```

Y en Java se llama a un constructor que tiene el mismo nombre de la clase:

```
Celda celda_2_3 = new Celda ( );
```

En ambos casos significa lo mismo. Se crea un objeto, instancia de la clase Celda, y su referencia queda alojada en la variable celda_2_3. Hay pequeñas diferencias entre ambos lenguajes, pero por el momento no son importantes. Hay otros lenguajes basados en clases en los que lo que se guarda en la variable no es una referencia, pero se mantiene la noción de que la clase es el molde a partir del cual se crean los objetos.

Implementación de la creación de objetos y del comportamiento en los lenguajes basados en prototipado

Hay lenguajes que definen objetos sin necesidad de clases. Por ejemplo, en JavaScript, un objeto que represente a la celda_2_3 se puede definir así:

```
var celda_2_3 = {
    fila: undefined,
    columna: undefined,
    numero: undefined,
    estaLibre: function() {
        return this.numero === undefined;
    }
};
```

Nótese que celda_2_3 no es más que una referencia a un objeto, y no necesitamos clases para definirlo.

Pero si queremos entender bien la idea de los lenguajes basados en prototipado, veamos cómo definir un nuevo objeto, referenciado por celda_5_7, como simple copia del objeto referenciado por celda_2_3:

```
var celda_5_7 = object.create(celda_2_3);
```

Es decir, primero creamos un objeto, referenciado por la variable celda_2_3, y luego creamos una copia del mismo, que queda referenciada por la variable celda57. El método estaLibre queda definido para cualquier otro objeto que use el mismo prototipo.

Como celda_5_7 está definida como una copia del prototipo, su comportamiento es el mismo, y cada método que agreguemos al prototipo se agrega a celda_5_7. Por supuesto, esto no se cumple a la inversa: celda_5_7 podría tener comportamiento adicional al del prototipo (cosa que en este caso no nos es útil, y que nos va a llevar a detenernos en esto en un capítulo posterior).

Particularidades de los lenguajes de programación en cuanto a la comprobación de tipos

Hemos visto algunas particularidades de implementación, y tenemos también un apéndice para ver la sintaxis de los distintos lenguajes que usamos en el libro. Sin embargo, hay una cuestión de implementación de la POO que no es tan

central como para haberla tratado hasta ahora ni tan secundaria como para relegarla a un apéndice. Nos referimos al momento en que cada lenguaje elige hacer comprobaciones de tipos. Hay lenguajes que comprueban todo lo que pueden en el momento de la compilación, como Java, y otros que difieren cualquier comprobación de tipos hasta la ejecución, como Smalltalk. También los hay que no se compilan, como JavaScript, y en este caso no hay comprobación posible hasta la ejecución. Vamos a ver, por lo tanto, cómo tratan estas cuestiones Smalltalk y Java. Para hacerlo, usaremos el mismo ejemplo en ambos lenguajes. Supongamos que tenemos una clase Cuenta, cuyas instancias entienden el mensaje depositar, una clase Celda, cuyas instancias entienden el mensaje filaQueContiene, y una clase Caja.

Smalltalk y la comprobación dinámica

Smalltalk, como decíamos, hace comprobación de tipos en tiempo de ejecución. Por eso, el siguiente fragmento de código no provoca ningún problema cuando uno lo compila:

```
caja := Caja new.
fila := caja filaQueContiene.
cuenta := Cuenta new.
cuenta := caja.
cuenta depositar: 200.
```

Hay muchos errores en este fragmento de código. Veamos de seguirlo con los números de línea. La primera línea crea una instancia de la clase Caja y se guarda la referencia en la variable caja. Hasta allí, todo bien. La línea 2, envía el mensaje filaQueContiene al objeto referenciado por caja. Pero lo más probable es que esto esté mal: no hemos definido que los objetos instancias de la clase Caja puedan comprender ese mensaje: el mismo lo habíamos pensado para instancias de Celda. Sin embargo, el compilador de Smalltalk no va a advertir el problema. Recién cuando intentemos ejecutar el código nos vamos a encontrar con un error. El entorno de Pharo, que es la versión de Smalltalk que usamos en el libro, abre una ventana con el texto:

```
MessageNotUnderstood: Caja>>filaQueContiene
```

Por ahora vamos a dejar de lado lo que nos muestra Pharo y asumamos que cometimos un error. Ahora, enmarquemos la línea 2 en un comentario y volvamos a compilar. La línea 3 tampoco tiene problemas: crea un objeto instancia de la clase Cuenta y se guarda la referencia en la variable cuenta. En la línea 4 parecería que se produce una confusión: estamos guardando la referencia a una Caja en una variable que hasta ahora tenía una Cuenta. Bueno, en realidad, en tiempo de compilación Smalltalk no tiene problema con esta asignación. Y en tiempo de ejecución tampoco: simplemente va a desalojar la referencia a la instancia de Cuenta y la va a reemplazar por la referencia a la instancia de Caja. Esto puede estar mal, y probablemente lo esté, pero Smalltalk no hace ese chequeo porque las variables en este lenguaje no tienen tipo: sólo los objetos son tipados, no las variables. La línea 5 va a ser la que evidencie el error cometido en la línea 4. En efecto, en tiempo de compilación, como ya sospechamos, Smalltalk no nos presenta problema. Pero cuando vayamos a ejecutar esa línea, en la que pretendemos que una instancia de Caja (que es lo que estamos referenciando ahora desde la variable celda) entienda el mensaje depositar, Pharo nos enviará un nuevo mensaje de error:

```
MessageNotUnderstood: Caja>>depositar
```

Como corolarios de lo anterior:

- En Smalltalk las variables no tienen tipo, los objetos sí.
- Smalltalk compila los programas, pero no presenta errores si operamos con variables que contienen referencias a objetos de tipos incompatibles. Al fin y al cabo, el compilador no tiene manera de darse cuenta de esto, ya que la variable no es tipada, mientras que el objeto, que sí es tipado, sólo existe en tiempo de ejecución.
- Smalltalk compila los programas, pero no presenta errores si enviamos mensajes a objetos que no los van a comprender. Al fin y al cabo, el compilador no tiene manera de darse cuenta de esto, ya que, como la variable no es

tipada, no tiene asociados métodos, mientras que el objeto, que sí conoce su clase y los métodos que ésta entiende, sólo existe en tiempo de ejecución.

Java y la comprobación estática

En Java, las variables tienen tipo. Por lo tanto, un intento de escribir el código equivalente al que hicimos en Smalltalk sería:

```
Caja caja;
Fila fila;
Cuenta cuenta;
caja = new Caja();
fila = caja.filaQueContiene();
cuenta = new Cuenta();
cuenta = caja;
cuenta.depositar(200);
```

Las tres primeras líneas definen las variables que vamos a utilizar, indicando claramente el tipo de cada una. La línea 4 hace lo que hacía la línea 1 de código Smalltalk: crea una instancia de Caja y guarda su referencia en la variable caja. En Java, esto obliga al compilador a chequear que el tipo de la variable sea compatible con el tipo del objeto que se está creando. Como se trata del mismo tipo, Caja, no presenta problema. Obviamente, tampoco hay problemas en tiempo de ejecución. La línea 5 pretende que se envíe el mensaje filaQueContiene a una instancia de Caja. Pero Java detecta el error de manera diferente a Smalltalk. Es el propio compilador que descubre que le estamos enviando un mensaje a un objeto referenciado desde la clase Caja que la clase Caja no tiene implementado. Por lo tanto, esta línea de código no puede compilar. Vamos a tener que encerrarla en un comentario para seguir. La línea 6, al igual que la 4, compila sin problemas, ya que crea una instancia de Cuenta y aloja su referencia en una variable cuyo tipo también es Cuenta. La línea 7, en cambio, no va a compilar. En efecto, estamos intentando guardar una referencia a una instancia de Caja en una variable cuyo tipo es Cuenta. La línea 8, contra lo que ocurría en Smalltalk, no presenta problema. En realidad, el compilador encuentra que la línea 7 está mal, y por lo tanto no la tiene en cuenta. En efecto, si eliminamos la línea 7, como en la variable cuenta hay una referencia a una instancia de Cuenta (en ningún momento pudimos poner otra cosa), es válido que enviemos el mensaje depositar, que entienden los objetos de la clase Cuenta.

Como corolarios de lo anterior:

- En Java, tanto los objetos como las variables tienen tipo. Los objetos son instancias de clases (sus tipos) y las variables tienen un tipo que es también una clase.
- Java compila los programas y, al hacerlo, verifica que no mezclemos variables de tipos incompatibles. Esto nos previene de errores con los objetos en tiempo de ejecución.
- Java compila los programas y, al hacerlo, verifica que no si enviemos mensajes a objetos que no los van a comprender. Esto hace que no haya mensajes no comprendidos en tiempo de ejecución.

Ventajas e inconvenientes de cada enfoque

Vimos que el enfoque de Java – que es propio de cualquier lenguaje con comprobación estática – lleva a que una gran cantidad de errores en el código surjan en tiempo de compilación. En el caso de Smalltalk – que es el de los lenguajes que sólo hacen comprobación dinámica – el compilador no se queja, y los errores se producen al ejecutar el programa. Desde ya, los lenguajes interpretados, como JavaScript o Python, al no tener compilación, caen también en la categoría de la comprobación dinámica. ¿Qué es mejor? En principio, suele ser considerado más seguro el enfoque de la comprobación estática. Esto se debe a que los errores surgen de entrada y no se propagan al programa en ejecución. Por otro lado, el compilador puede detectar errores sutiles que a un programador leyendo el código se le pueden pasar. Además, el chequeo estático, si está bien diseñado, es completo, mientras que la ejecución del código puede no recorrer algunas ramas del mismo, y ciertos errores pueden pasar inadvertidos por un tiempo. Entonces pareciera que la comprobación estática es mejor. Bueno, si consideramos que más seguro es sinónimo de mejor, sí. Pero los lenguajes de comprobación dinámica, incluidos los interpretados, por el mismo hecho de no exigir definir cada cosa con su tipo, permite ensayar

implementaciones parciales sin escribir tanto código y sin molestar con menudencias. Por lo tanto, existen defensores de ambos enfoques. Lo importante es tener en cuenta que, cuando trabajamos con lenguajes que sólo hacen comprobación dinámica, debemos ser más exhaustivos al probar nuestro código. Ya volveremos sobre esto último.

Pensando el comportamiento de una clase: contratos

Diseño por contrato

En el capítulo anterior mostramos cómo encara la POO la resolución de problemas. Vimos también la solución a un problema simple, encontrando objetos, su comportamiento esperado y hasta cómo implementar el comportamiento en algunos lenguajes. Sin embargo, no hemos explicado cómo se logra pasar del segundo al tercer paso: es decir, nos falta un enfoque metodológico.

Hace un cuarto de siglo ya que Bertrand Meyer propuso un método para derivar la implementación de clases a partir de la idea de que un objeto brinda servicios a sus clientes cumpliendo un contrato. A esto lo llamó diseño por contrato. (Bertrand Meyer, “Object-Oriented Software Construction”, Prentice Hall, 1988)

La idea primigenia del diseño por contrato es, entonces, que un objeto servidor brinda servicios a objetos clientes sobre la base de un contrato que ambos se comprometen a cumplir.

¿Y cómo se materializa un contrato? Meyer propone cuatro elementos fundamentales: firmas de métodos, precondiciones, postcondiciones e invariantes.

Firmas de métodos

Las firmas de los métodos son las que determinan cómo hacer para pedirles servicios a los objetos. En nuestro ejemplo de Sudoku, definimos las firmas de los métodos cuando escribimos:

- sudoku >> puedoColocarNumero:enCelda:
- celda >> estaOcupada
- sudoku >> fila:
- fila >> contieneNumero:
- sudoku >> columna:
- columna >> contieneNumero:
- sudoku >> cajaEnFila:yColumna
- caja >> contieneNumero:

Al conjunto de las firmas de métodos se lo suele llamar interfaz o protocolo del objeto, porque es lo que permite saber qué servicios expone y cómo dialogar con él. Nosotros no usaremos demasiado estos términos, porque hay lenguajes que tienen significados especiales para ellos.

Si el objeto cliente no conociera las firmas de los métodos, no sabría cómo comunicarse con el objeto servidor para solicitarle servicios. Ahora bien, las firmas de los métodos no nos dicen qué debe hacer un objeto al recibir un mensaje bien formado.

Precondiciones

Las precondiciones expresan en qué estado debe estar el medio ambiente antes de que un objeto cliente le envíe un mensaje a un receptor. En general, el medio ambiente está compuesto por el objeto receptor, el objeto cliente y los parámetros del mensaje, pero hay ocasiones en que hay que tener en cuenta el estado de otros objetos. Por ejemplo, antes de que el tablero le envíe el mensaje *estaLibre* al objeto referenciado por celda, debe cumplirse, como precondición, que *celda* referencie un objeto existente. Si una precondición no se cumple, el que no está cumpliendo el contrato es el cliente. Por lo tanto, el objeto receptor del mensaje, lo único que puede hacer es avisarle de ese incumplimiento al cliente y no seguir con la ejecución del método. Habitualmente, los lenguajes de POO tienen un mecanismo llamado excepciones para estos casos, que consiste en devolver un objeto especial avisando que algo anduvo mal.

Definición: excepción

Una excepción es un objeto que un objeto receptor de un mensaje envía a su cliente como aviso de que el propio cliente no está cumpliendo con alguna precondición de ese mensaje.

Por ejemplo, por cada una de las precondiciones, podemos definir una excepción. En el caso recién planteado podría ser el objeto de excepción *CeldaInexistente*.

Postcondiciones

En términos estrictos, el conjunto de postcondiciones expresa el estado en que debe quedar el medio como consecuencia de la ejecución de un método. En términos operativos, es la respuesta ante la recepción del mensaje. Por ejemplo, cuando el tablero le envía el mensaje *estaLibre* al objeto referenciado por celda, se espera que el objeto referenciado por celda actúe haciendo cumplir las siguientes postcondiciones:

- Si celda no referencia un objeto existente, lanzar una excepción.
- Si se cumple la precondición y la celda está libre, devuelve verdadero.
- Si se cumple la precondición y la celda está ocupada, devuelve falso.

El cumplimiento de las postcondiciones es responsabilidad del receptor. Si una postcondición no se cumple se debe a que el método está mal programado por quien deba implementar el objeto receptor. Por lo tanto, el cumplimiento de una postcondición – y, por lo tanto, lo correcto del método programado – se debe chequear con alguna prueba que verifique los resultados esperados del método: lo que se denomina una prueba unitaria.

Definición: prueba unitaria Una prueba unitaria es aquélla prueba que comprueba la corrección de una única responsabilidad de un método.

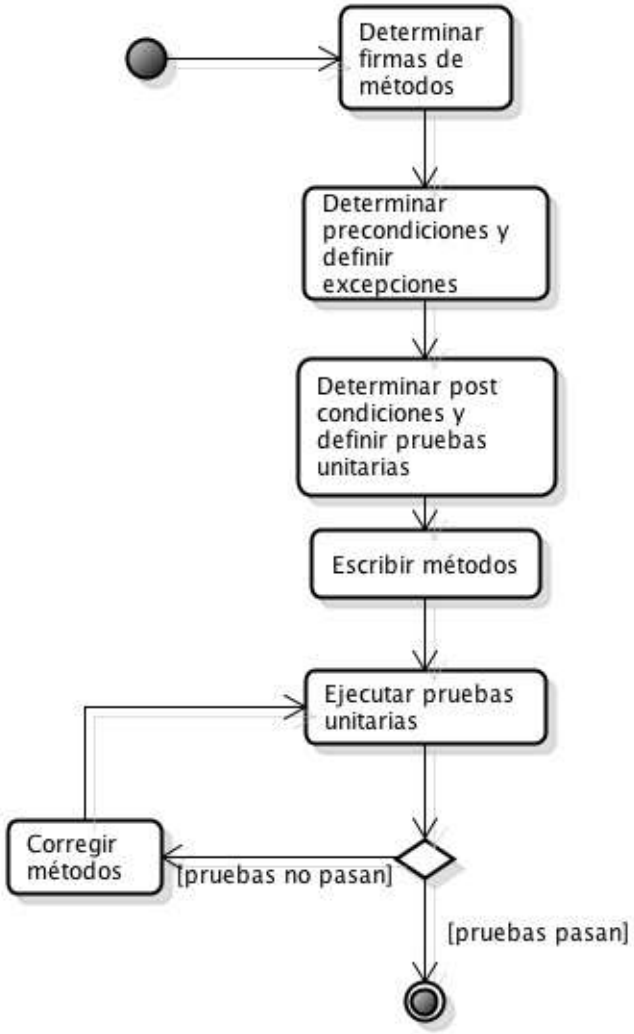
Corolario: Deberíamos tener al menos una prueba unitaria por cada postcondición.

Invariantes

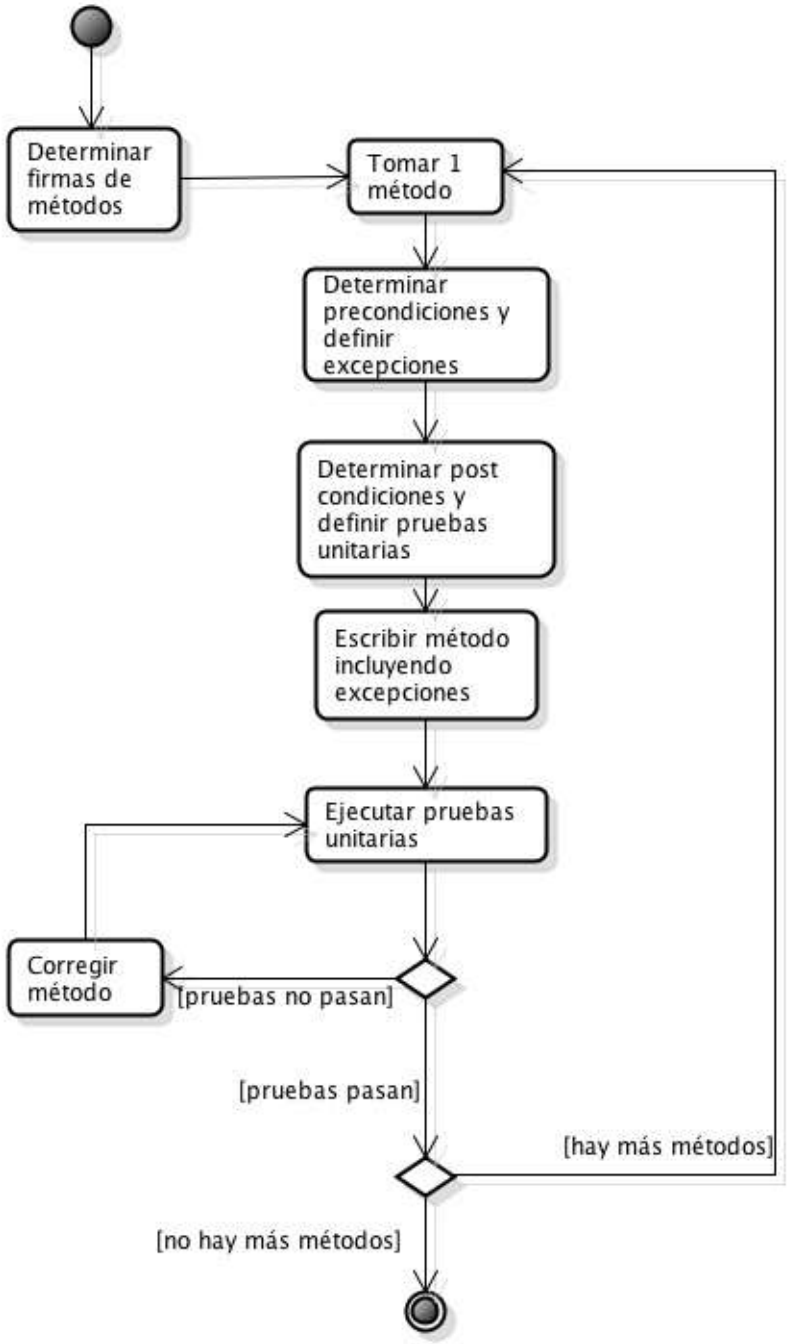
Los invariantes son condiciones que debe cumplir un objeto durante toda su existencia. Por ejemplo, un invariante que debe cumplir cualquier objeto celda del Sudoku es que, o bien está libre, o el número que contiene es un valor entero entre 1 y 9. El cumplimiento de los invariantes es responsabilidad de todos los métodos de un objeto, desde su creación. De alguna manera, pueden considerarse precondiciones y postcondiciones de todos los métodos. En general, suelen expresarse en forma de precondiciones o postcondiciones. Por ejemplo, el invariante recién mencionado de que el número a colocar en una celda debe estar entre 1 y 9 nos va a determinar una precondición a algún método que escribamos más adelante para colocar números en celdas. Si bien los invariantes suelen estar presentes a través de precondiciones o postcondiciones, no está nada mal analizarlos, y hasta verificar su cumplimiento en forma implícita o explícita.

El procedimiento del diseño por contrato

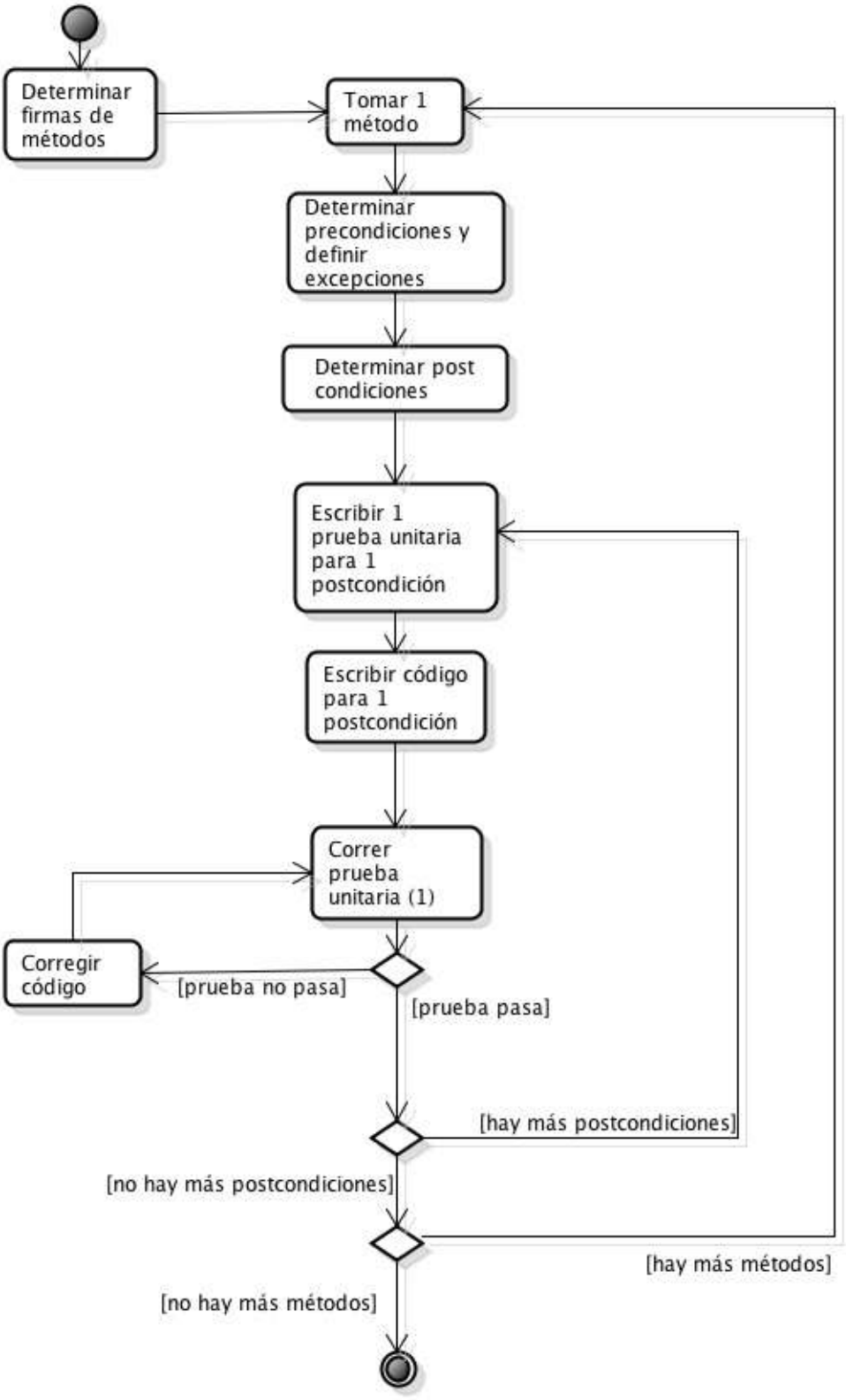
El procedimiento del diseño por contrato se podría entonces definir mediante el diagrama de actividades de la figura 4.1, aplicándolo para cada escenario que vayamos encontrando en nuestro problema:



Ahora bien, el problema de este procedimiento así escrito es que cuando fallan las pruebas no sabemos bien en qué momento introdujimos el error. Conviene ir desarrollando de a pasos menores, en forma iterativa e incremental. La figura 4.2 refina el procedimiento, definiendo de a un método por vez.



Si bien el procedimiento utilizado es incremental, ya que está definiendo de a un método por vez, conviene hacerlo de a pasos aún más pequeños, por ejemplo, tomando una postcondición por vez y haciendo pasarlas de a una, de modo tal de saber cuál de las postcondiciones es la que está fallando. El diagrama más detallado se muestra en la figura 4.3.



En definitiva, hemos desarrollado un procedimiento iterativo e incremental en pequeños pasos, que va derivando la implementación de un objeto y sus métodos en base a los elementos del diseño por contrato.

Ahora vamos a aplicarlo a nuestro problema del Sudoku.

Diseño por contrato aplicado… y un poco más

En el capítulo anterior mostramos algo de código. Como lo hicimos sin utilizar metodología alguna, no podemos estar muy seguros de haber hecho las cosas bien. Por eso, ahora vamos a tratar de ir construyendo un objeto celda de la aplicación de Sudoku con las ideas que nos plantea el diseño por contrato. Haremos solamente lo que surge del escenario de evaluar si podemos colocar un número en una celda.

Como vimos en el capítulo previo, en el escenario en cuestión, deberíamos implementar el método que tiene la firma siguiente:

```
celda >> estaLibre
```

Como la implementación la haremos en Smalltalk, lo que necesitamos es crear una clase *Celda*, con el método *estaLibre*.

Para ansiosos: el ejercicio adicional, al final del capítulo, está hecho en Java.

La única precondición de *estaLibre* es que celda referencie un objeto existente.

Las postcondiciones de estaLibre son:

- Si celda no referencia un objeto existente, lanzar una excepción.
- Si se cumple la precondición y la celda está libre, devuelve verdadero.
- Si se cumple la precondición y la celda está ocupada, devuelve falso.

Por cada postcondición escribimos una prueba.

En Smalltalk, la primera postcondición la podemos obviar, porque el propio entorno de ejecución chequea esta precondición por nosotros, y lanza una excepción de tipo *MessageNotUnderstood*. Podríamos forzar el cumplimiento de nuestra postcondición haciendo que lance una excepción definida por nosotros, pero nos parece que no tiene sentido, al menos en este lenguaje.

Pasemos a la segunda postcondición: si la celda está libre, devuelve verdadero. Bueno, tenemos que escribir nuestra primera prueba. Para trabajar de manera ordenada, escribiremos una clase de pruebas, que llamaremos *PruebasCuenta*, y un método *ejecutarPruebas*, desde el cual invocaremos las pruebas que vayamos creando. La primera prueba a escribir la llamaremos *estaLibreDevuelveTrueCuandoCeldaLibre*, así que nuestro método ejecutarPruebas empezará teniendo una única invocación a ese único método de pruebas:

```
ejecutarPruebas
    PruebasCelda new estaLibreDevuelveTrueCuandoCeldaLibre.
```

Estamos llamando un método de pruebas que aún no hemos escrito. Por supuesto, al intentar grabar el método en la imagen, Pharo nos advierte que el método *estaLibreDevuelveTrueCuandoCeldaLibre* no existe, así que pasamos a definirlo así:

```
estaLibreDevuelveTrueCuandoCeldaLibre
    | celda |
    celda := Celda new.
    (celda estaLibre)
        ifTrue: [Transcript show: 'La prueba pasó: la celda está libre y estaLibre devuelve true'; cr]
        ifFalse: [Transcript show: 'La prueba NO pasó: la celda está libre y estaLibre devuelve false'; cr]
```

La prueba parece estar bien si suponemos que una celda se crea libre.

Tip: asegúrese de entender el código recién escrito y por qué decimos que esto prueba lo que esperamos que pruebe.

Sin embargo, al intentar grabar, Pharo nos dice que no conoce la clase *Celda*. Una vez que definimos la clase, Pharo sigue quejándose: no conoce el método *estaLibre*. Entonces, lo definimos así:

```
estaLibre
    ^false
```

Ahora ya no hay errores de compilación. Pero, ¿qué hemos hecho? Lo único que hicimos fue lograr que nuestra prueba compile, pero todavía no hicimos nada productivo en el método *estaLibre* ya que devuelve siempre falso. Por lo tanto, ¿qué debería ocurrir si corremos la prueba? Sí: la prueba debe fallar.

Veamos. Ejecutamos el método ejecutarPruebas y vemos en el Transcript:

```
La prueba NO pasó: la celda está libre y estaLibre devuelve false
```

¿Por qué pasó esto? ¿Se entiende?

Tip: Asegúrese de que entiende lo que le dice el programa y por qué lo dice.

Tip: Nótese que deseamos que la prueba falle. Si no fuera así, ¿qué significaría el hecho de que pase una vez escrito el método correctamente?

Corolario: apenas escrita una prueba, hay que asegurarse de que la misma falle.

Entonces, ahora sí, debemos poner manos a la obra e implementar el método *estaLibre*.

Para poder implementar el método, debemos antes tomar una decisión de implementación: ¿cómo hacemos internamente para indicar si una celda está libre u ocupada? Tomemos la siguiente determinación bastante simple y directa: usemos un valor lógico para ello, y hagamos que comience en falso al crear el objeto. ¡Alto!. Esto nos lleva a tener que definir el constructor, y para definirlo debemos antes establecer su contrato.

¿Dónde estamos entonces? Estamos tratando de escribir el método *estaLibre*, pero no podemos hacerlo sin antes escribir el método constructor del objeto. Por lo tanto, esto se torna recursivo: debemos empezar por escribir otro método, lo cual nos va a llevar a seguir toda la técnica del diseño por contrato, y luego volveremos a tratar de terminar con el método *estaLibre*. En fin: son cosas que ocurren…

El constructor no tiene precondiciones. Como postcondición, pongamos la siguiente:

- Una celda siempre se crea libre.

Esto nos lleva a escribir la siguiente prueba, que también debemos invocar desde ejecutarPruebas:

```
unaCeldaSeCreaLibre
    | celda |
    celda := Celda new.
    (celda estaLibre)
        ifTrue: [Transcript show: 'La prueba pasó: la celda recién creada está libre'; cr]
        ifFalse: [Transcript show: 'La prueba NO pasó: la celda recién creada está ocupada'; cr]
```

El método es prácticamente idéntico al *estaLibreDevuelveTrueCuandoCeldaLibre*. Lo mantenemos de todas maneras porque su intención es diferente. Si la ejecutamos, como debe resultar obvio, no pasa. Ahora bien, esta prueba nos obliga también a usar *estaLibre*, que por el momento está fallando, así que no vamos a tener más remedio que implementar el constructor y *estaLibre* al mismo tiempo.

El constructor es un método que se invoca automáticamente al crear un objeto. En Smalltalk todas las clases pueden tener un constructor cuyo nombre es initialize. Nuestro initialize será:

```
initialize
    libre := true
```

Donde *libre* es un atributo o – como la llama Pharo – una variable de instancia. La implementación de estaLibre puede ser, entonces:

```
estaLibre
    ^libre
```

Si ejecutamos ahora las pruebas, ambas pasan. Vemos en el Transcript:

```
La prueba pasó: la celda recién creada está libre
La prueba pasó: la celda está libre y estaLibre devuelve true
```

Entonces ya tenemos dos métodos en la clase *Celda* y dos pruebas que corren. ¿Ya terminamos? A ver… parece que no. Habíamos dicho que *estaLibre* tenía dos postcondiciones, y sólo probamos una. Nos falta la siguiente: si la celda está ocupada, devuelve falso. Bueno, escribamos entonces una prueba que nos sirva para comprobar la postcondición, cuya invocación colocamos en el método *ejecutarPruebas*, como siempre:

```
estaLibreDevuelveFalseCuandoCeldaOcupada
    | celda |
    celda := Celda new.
    celda colocarNumero: 5.
    (celda estaLibre)
        ifTrue: [Transcript show: 'La prueba pasó: la celda está ocupada y estaLibre devuelve false'; cr]
        ifFalse: [Transcript show: 'La prueba NO pasó: la celda está ocupada y estaLibre devuelve true'; cr]
```

Nuevamente encontramos un problema: al intentar grabar, Pharo nos advierte que no existe el método *colocarNumero*. A comenzar de nuevo, entonces: la implementación de estaLibre nos obliga a escribir otro método, para el cual debemos aplicar en forma recursiva el diseño por contrato. La única precondición del mensaje colocarNumero es:

- El número que se pasa como argumento debe ser un valor entre 1 y 9.

Y las postcondiciones:

- Si se pasa como argumento un número que no está en el rango entre 1 y 9, lanzar la excepción *ValorInvalido*.
- Si se cumple la precondición, la celda debe quedar ocupada.
- Si se cumple la precondición, el número que quede en la celda debe ser el que se pasó como argumento.

Escribamos una prueba para la primera postcondición:

```
siNumeroMenorQue1DebeLanzarExcepcion
    | celda |
    celda := Celda new.
    [celda colocarNumero:(-2)]
        on: ValorInvalido
        do: [:e | Transcript show:'La prueba pasó: se pasó un valor negativo y colocarNumero lanzó ValorInvalido'; cr.
    Transcript show:'La prueba NO pasó: se pasó un valor negativo y colocarNumero NO lanzó ValorInvalido'; cr
```

Es decir, el método empieza creando una celda y enviando el mensaje colocarNumero con argumento -2, por lo que se espera que el método lance la excepción. De allí que la comprobación para que la prueba pase sea el lanzamiento de la excepción y el incorrecto el no lanzamiento.

Tip: por favor, asegúrese de entender por qué el comportamiento correcto (o exitoso, si se quiere) es el lanzamiento de la excepción.

Si intentamos grabar, Pharo nos va a decir que no existen el método *colocarNumero* ni la clase *ValorInvalido*. Para que desaparezca el primer error de compilación, creemos un método colocarNumero vacío:

```
colocarNumero: valor
```

Para que que desaparezca el segundo, creemos una clase de excepción.

Tip: en POO las excepciones son objetos, como ya dijimos. En Smalltalk, en particular, los objetos de excepción deben ser instancias de una clase que derive de la clase Error.

Por lo tanto, vamos a crear una clase ValorInvalido:

```
Error subclass: #ValorInvalido
```

Ahora sí, nuestro código compila. Si ejecutamos la prueba, la misma falla, como deberíamos esperar a esta altura, con el mensaje:

```
La prueba NO pasó: se pasó un valor negativo y colocarNumero NO lanzó ValorInvalido
```

Ahora bien: ¿alcanza con esta prueba? No: en realidad, probamos con un valor inválido (un número negativo) para ver si se producía la excepción, pero sería bueno probar algunos valores de borde, así como números por encima y por debajo del rango permitido.

Tip: conviene siempre hacer pruebas con valores en los bordes de los rangos permitidos, con cadenas de caracteres vacías, con referencias en nil, etc., ya que son fuentes frecuentes de errores sutiles.

Proponemos las siguientes pruebas

```
siNumeroCeroDebeLanzarExcepcion
    | celda |
    celda := Celda new.
    [celda colocarNumero:0]
        on: ValorInvalido
        do: [:e | Transcript show:'La prueba pasó: se pasó un cero y colocarNumero lanzó ValorInvalido'; cr. ^nil].
    Transcript show:'La prueba NO pasó: se pasó un cero y colocarNumero NO lanzó ValorInvalido'; cr

siNumeroMayorQue9DebeLanzarExcepcion
    | celda |
    celda := Celda new.
    [celda colocarNumero:10]
        on: ValorInvalido
```

```
        do: [:e | Transcript show:'La prueba pasó: se pasó un 10 y colocarNumero lanzó ValorInvalido'; cr. ^nil].
    Transcript show:'La prueba NO pasó: se pasó un 10 y colocarNumero NO lanzó ValorInvalido'; cr
```

Ahora sí tenemos pruebas suficientes que chequean la primera postcondición. Por supuesto, si ejecutamos las pruebas, las últimas tres que escribimos van a fallar, ya que colocarNumero no hace nada. Escribamos el mínimo código posible para que las pruebas pasen:

```
colocarNumero:valor
    ( (valor < 1) | (valor > 9) )
        ifTrue: [ ValorInvalido new signal ].
```

Efectivamente, si ahora ejecutamos las tres últimas pruebas, pasan. Pasemos entonces a la segunda postcondición del método colocarNumero: la celda debe quedar ocupada. La prueba que puede ser útil para comprobar esa postcondición podría ser la misma que usamos para la segunda postcondición de estaLibre: estaLibreDevuelveFalseCuandoCeldaOcupada. Como la intención de lo que se quiere probar es exactamente la misma, la dejamos así. Si ejecutamos la prueba, veremos que falla, que es lo que ya a esta altura estamos acostumbrados a que pase. Agregamos código a colocarNumero para hacer funcionar la prueba:

```
colocarNumero:valor
    ( (valor < 1) | (valor > 9) )
        ifTrue: [ ValorInvalido new signal ].
    libre := false
```

A continuación, como siempre, ejecutamos el conjunto de las pruebas, y las mismas pasan. ¡Terminamos! Hemos escrito pruebas para comprobar las postcondiciones de *estaLibre*, y están todas ejecutándose correctamente. Por lo tanto, podemos asegurar de que el método *estaLibre* ya está listo. Al fin y al cabo, hemos aplicado exitosamente los principios del diseño por contrato y lo hemos ido verificando con pruebas, así que podemos sentirnos satisfechos. ¿O no? A ver… Es cierto que *estaLibre* está completo. Pero quedó una postcondición de *colocarNumero* sin verificar: el número que quede en la celda debe ser el que se pasó como argumento. ¿Qué debemos hacer entonces? Nuestro objetivo era implementar el método *estaLibre*, y lo hemos conseguido. Luego, al ir escribiendo pruebas, nos dimos con que necesitábamos un método *colocarNumero*, que hemos logrado que al menos haga lo que necesitábamos para *estaLibre*. ¿Debemos terminar de implementarlo o podemos dejarlo así? Estamos ante una cuestión en la que no todos pensamos igual. Tal vez *colocarNumero* no sea un método útil en la clase *Celda*, y seguir con una implementación de ese método, que en principio sólo sirve para escribir pruebas, sea un sobrediseño de la clase *Celda*. De hecho, el chequeo de que *colocarNumero* lance una excepción cuando le pasamos un argumento inválido tampoco era algo que necesitásemos para el método *estaLibre*. Hay tres posibilidades:

- Nos detenemos acá, dejando *colocarNumero* a medio implementar, y sólo lo retomamos si luego lo necesitamos.
- Seguimos adelante con la implementación de *colocarNumero*, incluyendo la verificación de la tercera postcondición.
- Dejamos un método *colocarNumero* con lo mínimo indispensable para lograr que las pruebas de *estaLibre* pasen, lo cual implicaría también eliminar la prueba que chequea la excepción y el propio lanzamiento de la excepción en *colocarNumero*.

La primera opción es la más cómoda, ya que dejaríamos todo como está y listo. Pero es también la opción más inconsistente. El método *colocarNumero* está a medio hacer: tiene comportamientos que exceden lo que necesitamos para *estaLibre* y a la vez no está completo. A los autores no nos parece una opción seria.

La segunda es la que parece más profesional. Ya que estamos, sigamos adelante e implementemos *colocarNumero* en forma completa. Tiene como inconveniente que no sabemos si realmente vamos a necesitar ese método (aunque pareciera que sí…) y además puede que nos veamos en la necesidad de seguir recursivamente agregando métodos a la clase (invitamos al lector ansioso a que intente seguir adelante, y enseguida verá cómo se necesitan nuevos métodos). Siguiendo los principios de XP, vamos también a dejarla de lado. Nos quedamos con la tercera opción. Para hacer nuestra metodología más consistente, borramos la prueba que comprobaba el lanzamiento de la excepción y dejamos al método

colocarNumero sólo con lo que necesitamos. También, como una decisión de consistencia de nomenclatura, vamos a poner un nombre distinto a *colocarNumero*, que indique que no es un mensaje que vayamos a enviarle a las celdas por el momento:

```
paraPruebas_ocupar
    libre := false
```

Es decir, estamos diciendo que es un método que se utiliza sólo para pruebas, y que sólo sirve para ocupar celdas. Y de hecho, eso es lo que hace. Por supuesto, debemos cambiar nuestro método *estaLibreDevuelveFalseCuandoCeldaOcupada* para que llame a paraPruebas_ocupar en vez de colocarNumero.

Revisando lo que quedó construido:

- Una clase PruebasCelda que contiene las pruebas que fuimos escribiendo.
- Un método ejecutarPruebas, que contiene las invocaciones a cada uno de los métodos de prueba, y que podemos seguir haciendo crecer en el futuro.
- 3 métodos de prueba necesarios para comprobar el cumplimiento de las postcondiciones de estaLibre y del constructor.
- Una clase Celda con un inicializador (initialize) y un método estaLibre.

Tip: es conveniente volver a ejecutar el conjunto de todas las pruebas del sistema a ciertos intervalos de tiempo, de modo tal de asegurarse de que los últimos cambios no introdujeron nuevos errores. A esta práctica se la denomina pruebas de regresión, y es conveniente hacerlas con la mayor frecuencia que soporte el tiempo que las mismas demoren.

Una pizca de TDD

Desarrollo empezando por las pruebas

En las páginas previas, al plantear un procedimiento para aplicar diseño por contrato, hemos venido escribiendo las pruebas unitarias apenas conocíamos las postcondiciones, lo cual siempre implicó escribirlas antes del código que dichas pruebas chequeaban.

Esto acarrea algunas ventajas, tanto de orden técnico como metodológico. Por ejemplo:

- Minimiza el condicionamiento del autor por lo ya construido. Esto es, dado que la prueba se escribe antes, no se prueba solamente el camino feliz, la respuesta típica ante el envío de un mensaje, sino también los casos excepcionales o no esperados.
- Al escribir antes las pruebas, las mismas se convierten en especificaciones de lo que se espera como respuesta del objeto ante el envío de un determinado mensaje. Por lo tanto, no se debería codificar nada que no tenga definida su prueba de antemano, de modo tal de no implementar lo que no se necesita realmente.
- Permite especificar el comportamiento sin restringirse a una única implementación. Una vez especificado el comportamiento esperado mediante pruebas y realizada la implementación posterior, ésta podría variar más adelante, en la medida que la prueba continúe funcionando bien. De esta manera se refuerza el encapsulamiento.
- Al ir definiendo de a una prueba cada vez, tenemos un desarrollo incremental genuino, definido por pequeños incrementos que se corresponden con funcionalidades muy acotadas.
- Si bien se trata de una técnica de especificación del comportamiento de los objetos, deja un conjunto nada despreciable de pruebas de automatizadas que puede ir creciendo a medida que crece el programa que estamos desarrollando.

No son pocas ventajas, ¿no? Pero esto no es todo.

Pruebas en código

Otra cosa que venimos haciendo es que las pruebas queden expresadas en código y que se puedan ejecutar con una mínima intervención humana, además de que el reporte de pruebas nos da información bastante certera de qué fue lo que anduvo mal cuando una prueba falló. Esta condición también tiene sus ventajas. Entre ellas destacan:

- Permite independizarse del factor humano, con su carga de subjetividad y variabilidad en el tiempo. Una prueba automátizada (bien escrita) da el mismo resultado un viernes que un miércoles, un día a las 6 de la tarde o a las 11 de la mañana, al principio de un proyecto o minutos antes de una entrega.
- Facilita la repetición de las mismas pruebas, con un costo ínfimo comparado con las pruebas realizadas manualmente por una persona. Esto es aplicable a regresiones, debugging y errores provenientes del sistema ya en producción.
- El hecho de tener pruebas en código y de poder ejecutarlas con un mínimo costo, hace que el programador se resista menos a ejecutarlas con una alta frecuencia y regularidad. Por lo tanto, permite que el desarrollador trabaje con más confianza, basado en el hecho de que el código que escribe siempre funciona.

TDD restringido

Esta forma de trabajar – pruebas automatizadas y escritas antes de la implementación – no se nos ocurrió a los autores. Fue propuesta formalmente por primera vez por Kent Beck dentro del marco de una práctica metodológica llamada TDD (desarrollo guiado por las pruebas o, en inglés, Test-Driven Development). (Kent Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, 1999).

Según Beck, TDD incluye tres prácticas:

- Test-First: escribir las pruebas antes que el código, como ya venimos haciendo.
- Automatización: las pruebas deben correr con una mínima intervención humana, como hemos hecho, pero haremos mejor más adelante.
- Refactorización: mejorar la calidad de nuestro código en cada pequeño ciclo de codificación, como veremos en capítulos posteriores.

Por lo tanto, lo que venimos haciendo no es todavía TDD, ya que no incluimos la práctica de refactorización. A lo largo de los capítulos metodológicos vamos a ir incorporando más elementos de TDD para escribir cada vez mejor código.

Pruebas como ejemplo de uso

Al escribir las postcondiciones como pruebas en código, como vimos, estamos usándolas como herramientas de diseño. Pero hay algunos subproductos que nos quedan sin costo: entre ellos, ejemplos de uso que sirven de documentación. En efecto, una prueba escrita en código muestra a las claras la firma que tiene un método, sus posibles excepciones y los resultados que se esperan de su ejecución, más que cualquier documento escrito en prosa. Además, como ventaja sobre los documentos externos, las pruebas se suelen mantener junto con el código, por lo que resulta más fácil mantenerlas actualizadas.

Tip: mantener siempre las pruebas además del código “productivo”.

Pruebas como control de calidad

La palabra “prueba” suena en los oídos de los profesionales de desarrollo de software como sinónimo de control de calidad. En efecto, durante años hemos usado, y seguimos usando, pruebas como herramientas de control de calidad. Y no está mal, aunque haya otras herramientas. En este capítulo – sin embargo – hemos usado pruebas unitarias automatizadas como una herramienta de diseño de software. ¿Quiere decir entonces que estas pruebas no nos sirven como herramientas de control de calidad? No, no estamos diciendo eso. De hecho, otro subproducto de usar pruebas automatizadas para el desarrollo es que las mismas sirven a lo largo del tiempo, no sólo como documentación del uso de

los objetos, sino también como herramientas para los controles de regresión.

De todas maneras, tengamos en cuenta que las pruebas unitarias nunca pueden ser suficientes para controlar la calidad de un producto. Siempre necesitaremos pruebas más abarcativas, de integración (en el sentido de que prueben escenarios con varios objetos), de aceptación de usuarios y exploratorias. Ya volveremos sobre esto en un capítulo posterior.

Pruebas unitarias automatizadas en lenguajes de comprobación dinámica

Los lenguajes de comprobación estática, al hacer verificaciones de tipos en tiempo de compilación, nos evitan algunos errores sutiles antes de la ejecución de las pruebas. Pero esta ventaja no está presente en los lenguajes de comprobación dinámica.

NicoPaez: personalmente creo que la compilación en los lenguajes de comprobación estática suelen promover una falsa idea de calidad. Digo falsa porque el hecho de que el código compile nada dice sobre el correcto funcionamiento de ese código. Quedarse "tranquilo" con el hecho de que algo compila es engañarse a uno mismo.

Esto hace que las pruebas unitarias automatizadas sean aún más importantes en estos lenguajes, incluso como herramientas de control de calidad. Hay quienes dicen, precisamente, que en los lenguajes de comprobación estática, el compilador hace un primer control de calidad de alto nivel. Algunos programadores incluso escriben pruebas unitarias que chequean los tipos de los argumentos. Nosotros no lo hemos hecho, por dos razones: creemos que es una sobreutilización del método; y para no abrumar al lector con tanto código.

Invariantes y constructores

Al definir los invariantes, más arriba, dijimos que son condiciones que debe cumplir un objeto, durante toda su existencia. También dijimos que pueden considerarse precondiciones y postcondiciones de todos los métodos. Por lo tanto, si vemos a los inicializadores como métodos especiales que se invocan al crear un objeto – eso es lo que son en definitiva – también los inicializadores deberían tener que cumplir, al menos como postcondiciones, los invariantes.

Por ejemplo, en el caso de la clase Celda que hemos implementado parcialmente en este capítulo, si decimos que un invariante que debe cumplir cualquier celda es que, o bien esté libre o el contenido de la misma sea un número entre 1 y 9, ya el inicializador debe cumplirlo. Nosotros lo hicimos, en el *initialize* de Celda, al partir de la premisa de que una celda se crea libre.

Pero cuidado: no siempre vamos a poder hacer esto en cualquier lenguaje. Supongamos que definimos que el invariante para una fila es que se cree con una lista de celdas que le informamos en el momento de su creación. Eso nos obligaría a agregarle un parámetro al inicializador. Si bien en Java – como en otros lenguajes – esto no representa ningún problema, en Smalltalk la respuesta es más decepcionante: el método initialize de Smalltalk, que se llama inmediatamente después de crear cada objeto, no admite el pasaje de parámetros.

Por eso es que los programadores Smalltalk suelen enviar mensajes de inicialización al crear los objetos, como en el fragmento que sigue:

```
fila := Fila new inicializarConCeldas: coleccionCeldas.
```

Para ello, hay que definir un método inicializador denominado *inicializarConCeldas*:

```
Fila >> inicializarConCeldas : coleccionCeldas
     self celdas := coleccionCeldas
```

Sin embargo, notemos que esto no termina de solucionar el problema, pues un programador poco cuidadoso puede seguir creando instancias de Fila con la simple invocación del método new, en cuyo caso la fila sería creada sin que se cumpla el invariante.