

Texto de apoyo conceptual de
Algoritmos y Programación III

Facultad de Ingeniería de la Universidad de Buenos Aires

Versión Beta 0.7 del libro:

Programación Orientada a Objetos

Tercera edición

con Smalltalk, Java y UML

Carlos Fontela

2018

Prefacio

Empecé a escribir este libro apenas terminé con la segunda edición de mi libro de programación orientada a objetos [Fontela 2008]. Luego, al avanzar con la segunda edición de mi libro de Java y UML [Fontela 2010], quedó a un lado, pero al poco tiempo me di cuenta de que se necesitaba un enfoque totalmente distinto.

En efecto, hasta 2010 todos mis libros habían encarado la programación orientada a objetos con lenguajes compilados, de comprobación estática de tipos y basados en clases. Esto estaba ignorando de manera flagrante a cualquier otro lenguaje del estilo de Smalltalk, Python, Ruby o JavaScript, alguno de ellos histórico, otros más modernos y de un auge creciente. Creo que fue un error, tal vez motivado por mi llegada al mundo de los objetos desde C++ y Object Pascal, pero un error al fin: ese error debía corregirse.

La cuestión es que, después de más de dos décadas enseñando programación orientada a objetos, cambié totalmente mi enfoque. El resultado son mis cursos actuales de Algoritmos y Programación III en la Universidad de Buenos Aires, y este libro totalmente rediseñado.

Por eso también creo que este libro tiene un enfoque original, que hace que tenga sentido leerlo aun cuando hay cientos o miles de libros de orientación a objetos. Entiendo que puede mostrar la esencia de la orientación a objetos, a partir de pensar los problemas desde este paradigma, y no desde adaptaciones de otras formas de resolución de problemas, como la procedural, la funcional o la relacional.

La orientación a objetos es hoy un paradigma que busca facilitar el modelado, construcción y mantenimiento de software complejo, y lo logra con bastante éxito. Este libro pretende ayudar en la transición a este paradigma.

El libro supone que el lector conoce ya algo de programación. Esto es así porque en la mayor parte de las currículas universitarias de informática o computación se accede al paradigma de objetos cuando ya se han pasado uno o dos cursos previos de programación, y pensando en ese público escribí mi libro¹.

Todo el código del libro fue compilado y probado, así que presumo que debería estar libre de errores. Pero... como siempre releo mis libros antes de entregarlos al editor, las enmiendas que hago a mis programas, probados previamente, pueden hacer que éstos dejen de funcionar, sobre todo en los últimos retoques que se hacen sólo en papel y sobre la última versión para imprimir.

¿Cómo debe leerse el libro?

Si el lector es novato en orientación a objetos, le recomiendo leer toda la obra. Para él lo escribí, cuidando de presentar los temas en un orden lógico, así como los lenguajes, notaciones y ejemplos. De hecho, recomiendo fuertemente leer los temas repetidos en distintos lenguajes, pues con esa intención escribí: el uso de más de un lenguaje tiene que ver con poder analizar las distintas construcciones que presentan.

Ahora bien, si el lector considera saber mucho de algún tema en particular, puede saltar los

¹ Hay muchas prestigiosas universidades del mundo que han seguido el camino inverso, habitualmente llamado currículum invertido [Meyer 1993]. Eso se ha hecho incluso en una de las universidades en las que enseño otra materia, la UNTREF. Pero no es lo más habitual. De todas maneras, al estar formando profesionales para las próximas cuatro o cinco décadas, el paradigma con el que se empieza a programar no debería ser el más demandado por la industria, sino que el que permita adquirir conceptos y facilite “aprender a aprender” programación.

capítulos o ítems que domine.

Importante:

Este libro, a pesar del tiempo que me ha llevado, no está terminado. Probablemente pueda cerrarlo a mediados de 2018 y salga a la venta en papel a fines de ese año. Pero creo que, mientras tanto, puedo ir liberando parte del mismo, para que sirva a mis lectores de hoy, que no estarán dispuestos a esperar un año más, y también para recibir críticas que me ayuden a mejorarlo.

Por lo recién dicho, es claramente un libro incompleto: le faltan ítems y capítulos enteros, habrá mejorar la tipografía, uniformizar la tipografía y los gráficos, incorporar ayudas visuales, etc. Por eso solamente lo estoy lanzando como una versión beta, para quien le pueda servir.

Contenido

PREFACIO.....	2
CONTENIDO.....	4
1. INTRODUCCIÓN	12
Contexto	12
Una curiosidad: la gente y sus opiniones sobre POO	12
Razones de un paradigma	14
Construcción reparando en la complejidad.....	14
Diseño pensando en el cambio.....	14
Casos de estudio	15
Sudoku	15
Banco simplificado.....	18
Decisiones de lenguajes, herramientas y metodología	18
Los lenguajes del libro	18
Por qué UML.....	19
Los entornos de desarrollo	19
Prácticas ágiles.....	19
Recapitulación	20
2. UN MUNDO DE OBJETOS	21
Contexto	21
Nuestra primera aproximación a la solución a un problema	21
Paso 1: encontrar objetos.....	21
Paso 2: resolver cómo deben interactuar los objetos	22
Paso 3: implementar el comportamiento de los objetos	26
Conceptualizando.....	27
El comportamiento como aspecto central	27
Los objetos tienen identidad, estado y comportamiento	27
Encapsulamiento: no importa el cómo sino el qué	28
Polimorfismo: cada objeto responde a los mensajes a su manera	29
Algunas cuestiones de implementación	29

Tipología de los lenguajes de programación orientados a objetos	29
Implementación del comportamiento en los lenguajes basados en clases	30
Creación de objetos en los lenguajes basados en clases: instanciaión.....	31
Implementación de la creación de objetos y del comportamiento en los lenguajes basados en prototipado	31
Particularidades de los lenguajes de programación en cuanto a la comprobación de tipos.....	32
Smalltalk y la comprobación dinámica	32
Java y la comprobación estática	34
Ventajas e inconvenientes de cada enfoque	35
Un vistazo a la implementación	35
Implementando un objeto en JavaScript.....	35
Implementando una clase en Smalltalk.....	36
Implementando una clase en Java.....	38
Ejercicio adicional	38
Ejercicios propuestos	40
Recapitulación	40
3. PARÉNTESIS METODOLÓGICO: DISEÑO POR CONTRATO Y UN PROCEDIMIENTO CONSTRUCTIVO	41
Contexto	41
Pensando el comportamiento de una clase: contratos	41
Diseño por contrato.....	41
Firmas de métodos	41
Precondiciones	42
Postcondiciones	42
Invariantes	43
El procedimiento del diseño por contrato.....	43
Diseño por contrato aplicado... y un poco más	48
Pruebas unitarias... ¿pruebas?	54
Desarrollo empezando por las pruebas.....	54
Pruebas en código	55
TDD restringido.....	55
Pruebas como ejemplos de uso	56
Pruebas como control de calidad	56
Pruebas unitarias en lenguajes de comprobación dinámica	56
Algunas recomendaciones.....	57

Los métodos de consulta y modificación de propiedades: ¿hay que probarlos?	57
Invariantes y constructores	57
Ejercicio adicional	58
Ejercicios propuestos	62
Recapitulación	62
4. LOS OBJETOS COLABORAN.....	63
Contexto	63
Relaciones entre objetos.....	63
Los objetos interactúan	63
Yendo a lo práctico... o casi.....	63
¿Cómo implementamos esto?.....	67
Conceptualizando.....	69
Dependencia y asociación	69
Interludio metodológico: ¿en qué orden probar cuando tenemos objetos que deben construirse luego?	70
Relaciones entre clases.....	71
Colaboración por delegación.....	71
Programación por diferencia: herencia	72
Programación por diferencia: delegación de comportamiento	74
Redefinición	75
Clases abstractas	76
Métodos abstractos.....	77
Cuestiones de implementación	78
Delegación y herencia en Smalltalk y Java.....	78
Visibilidad	79
Métodos y clases abstractos.....	80
Constructores en situaciones de herencia y asociación	81
Herencia y compatibilidad en lenguajes con comprobación de tipos estática.....	82
Ejercicios adicionales.....	83
Construcción de objetos en cadena.....	83
Varios tipos de cuentas.....	90
Ejercicios propuestos	95
Recapitulación	95
5. HABLEMOS DE HERRAMIENTAS: FRAMEWORKS XUNIT Y SU USO.....	96

Contexto	96
La necesidad de herramientas.....	96
La necesidad de herramientas específicas	96
Primer intento: SUnit.....	96
Los herederos	99
Integración en los IDEs	100
La evolución: NUnit y JUnit 4.....	100
Herramientas de cobertura	100
Ejercicio adicional	102
Ejemplo con SUnit.....	102
Ejemplo con JUnit	104
Ejercicios propuestos	108
Recapitulación	108
6. POLIMORFISMO: LOS OBJETOS SE COMPORTAN A SU MANERA.....	109
Contexto	109
De regreso al polimorfismo	109
¿Qué es polimorfismo?.....	109
¿Por qué es importante el polimorfismo?.....	111
Casos que se resuelven con polimorfismo	112
Polimorfismo y vinculación tardía	113
Polimorfismo y herencia: ¿realmente deben ir juntos?	114
Métodos abstractos y comprobación estática	117
Polimorfismo sin herencia en lenguajes de comprobación estática: interfaces	117
Algunas propiedades interesantes de las interfaces de Java.....	119
Ejercicio adicional	120
Ejercicios propuestos	122
Recapitulación	122
7. PARÉNTESIS METODOLÓGICO: REFACTORIZACIÓN Y CIERRE DE TDD	123
Contexto	123
Cómo evitar la degradación del código.....	123
Mantenimiento y entropía	123
Refactorización al rescate.....	124
Condiciones de una refactorización segura.....	125

A qué llamamos observable.....	126
Simplificando métodos largos	127
Extracción de código repetido	130
Catálogos	133
TDD completo	134
Refactorización como parte de TDD	134
TDD y refactorización: un matrimonio bien avenido.....	135
Refactorización y desarrollo incremental	135
Situaciones de refactorizaciones más complejas	136
Refactorizaciones que rompen las pruebas.....	136
Grandes refactorizaciones	137
Refactorización de código legacy.....	138
Refactorización de pruebas automáticas	138
Ejercicio adicional	139
Ejercicios propuestos	141
Recapitulación	141
8. HABLEMOS DE HERRAMIENTAS: UML Y SU USO	142
Contexto	142
UML como herramienta de comunicación.....	142
Discutiendo el diseño	142
Documentando lo realizado	143
¿UML como lenguaje de programación?.....	143
UML: un lenguaje de modelos.....	144
Modelos: representación simple de algo complejo.....	144
Modelos de software.....	145
Diagramas de UML.....	145
Todos los diagramas	145
Representación de objetos.....	146
Representación de comportamiento: diagramas de secuencia y de comunicación	147
La visión estática: diagrama de clases	151
La visión macro: paquetes	156
Diagramas de actividades	160
Diagramas de estados.....	162
Otros diagramas de UML.....	163

Ejercicio adicional	164
Ejercicios propuestos	164
Recapitulación	164
9. LO QUE FUE QUEDANDO DE LADO	165
Contexto	165
Excepciones.....	165
Errores y excepciones	165
Lanzamiento de excepciones.....	166
Captura de excepciones y manejadores.....	167
Restitución de recursos luego de la excepción.....	167
Excepciones derivadas.....	168
Excepciones en lenguajes sin verificación previa a la ejecución: el caso de Smalltalk	169
Excepciones en lenguajes de verificación estática y el caso extremo de Java	169
Atributos y métodos en excepciones creadas por el programador.....	172
Pasando al lado de la clase.....	173
Comportamiento y estado de las clases	173
Métodos de clase.....	174
Inicialización de objetos.....	174
Atributos de clase	176
Genericidad.....	177
Antes de empezar: colecciones estilo Smalltalk	177
Genericidad en lenguajes de verificación estática: el caso de Java.....	178
Implementación de clases y métodos genéricos	179
Genericidad más allá de las colecciones	181
Información de tipos en tiempo de ejecución	181
Cómo se obtiene la información de tipos.....	181
Por qué evitar la información de tipos en tiempo de ejecución.....	181
Modelo de objetos y clases en Smalltalk.....	182
Modelo de objetos y clases en Java.....	184
Reflexión y uso en Java	186
Genericidad y RTTI en Java	188
Ejercicios propuestos	189
Recapitulación	189
10. DISEÑO ORIENTADO A OBJETOS	190

Contexto	190
Definición y objetivos del diseño.....	190
Principios de diseño orientado a objetos	192
No repetir	192
Alta cohesión y bajo acoplamiento.....	192
Única responsabilidad	193
Encapsulamiento de lo que varía.....	194
Principio abierto-cerrado.....	194
Minimizar la herencia de implementación	194
Sustitución de Liskov	196
Inversión de dependencia	196
Segregación de la interfaz.....	197
Granularidad fina.....	198
Recomendaciones para el diseño de paquetes	198
Separación de incumbencias	199
Cuándo es malo un diseño.....	200
Patrones de diseño.....	201
¿Qué es un patrón de diseño?.....	201
Iterator.....	202
Template Method.....	202
Factory Method	204
Observer	205
Factory Class	208
Composite.....	209
Command simplificado	210
Strategy.....	211
Chain of Responsibility	212
Decorator.....	212
State.....	214
Double Dispatch y Visitor.....	216
Adapter	222
Facade.....	223
Proxy.....	225
MVC	226
Separación en capas	228
El polémico patrón Singleton.....	230
Resumen de patrones.....	232

Ejercicios propuestos	234
Recapitulación	234
11. ENTRE DOS PARADIGMAS.....	235
Contexto	235
A pesar de todo... código duplicado	235
El problema.....	235
Una solución posible.....	236
Expresiones lambda.....	238
La visión de Smalltalk: bloques de código que son objetos.....	239
¿Qué tiene que ver esto con la programación funcional?.....	240
Ejercicios propuestos	240
Recapitulación	240
12. CIERRE METODOLÓGICO: MIRANDO AL SOFTWARE COMO PRODUCTO.....	241
13. APÉNDICE A: ELEMENTOS BÁSICOS DE SMALLTALK.....	242
14. APÉNDICE B: ELEMENTOS BÁSICOS DE JAVA	243
15. APÉNDICE C: ALGUNAS OBJECIONES A LAS PRÁCTICAS Y HERRAMIENTAS DE ESTE LIBRO	244
UML	244
La crítica ágil	244
NoUML.....	244
UML en la pizarra.....	244
TDD	244
BIBLIOGRAFÍA	245

1. Introducción

Contexto

Este libro trata sobre programación orientada a objetos, en adelante POO. Sin embargo, es difícil decidir qué es la POO si le preguntamos a los profesionales de computación o informática, aun aquéllos que conocen el paradigma. Los resultados de una encuesta realizada por el autor se presentan en este capítulo para mostrar estas confusiones.

Otra cuestión que se aborda es el objetivo general del paradigma de objetos. Vale decir, cuándo y por qué resulta una buena manera de resolver problemas.

También vamos a definir el tono general del libro, la simbología del mismo, los lenguajes, técnicas y herramientas que vamos a utilizar, y presentaremos algunos casos de estudio con los que vamos a seguir trabajando en los capítulos sucesivos.

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, voy a mostrar el resultado de una encuesta realizada en marzo y abril de 2015 sobre 250 respuestas obtenidas².

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 encuesta³.

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%

² La hice yo mismo sin pretensión de que fuera 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 mí mismo. Se tomaron las primeras 250 respuestas, recibidas en una semana aproximadamente.

³ Siendo yo mismo profesor universitario, es probable que los grupos de estudiantes expliquen este sesgo.

- 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 POO. 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 el autor, lo que caracteriza a la POO es que, en este paradigma:

- Un programa es un conjunto de objetos enviándose mensajes y reaccionando ante los mismos.
- Es responsabilidad de cada objeto saber cómo responder a esos mensajes.
- Cada objeto podría responder a los mensajes de manera distinta.

Por lo tanto, los conceptos centrales de POO son **objeto** y **mensaje**, términos que aparecen claramente en las tres oraciones anteriores. Como derivación implícita, el autor agregaría **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 mencionado, con el 91% de los encuestados considerándolo fundamental o muy importante es el de clase, que es una cuestión de implementación solamente⁴. 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.

⁴ 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.

Como vemos, hay mucha confusión respecto de lo que es la POO: esto justifica claramente la escritura de este libro.

Razones de un paradigma

Construcción reparando en la complejidad

Una de las preguntas que muchos nos hicimos 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 conceptuales – 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 fabricados⁵, 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, 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).

Diseño pensando en el cambio

Otro aspecto en el cual se ven las bondades de la POO es la facilidad de cambio. Desde ya, esto es importante en una industria como la del software, cuyos productos más exitosos evolucionan constantemente, adaptándose a cambios de requerimientos, tecnológicos, metodológicos o de negocio.

¿Por qué decimos que la POO facilita el cambio del software? Por dos razones.

⁵ 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.

En primer lugar, el mismo hecho de escribir en base a componentes intercambiables, facilita quitar un componente y cambiarlo por otro (algo como desenchufar uno y enchufar otro). Si el sistema se había construido respetando los principios de encapsulamiento y contratos, el costo de esta operación será menor que en un sistema en el que las partes son muy interdependientes.

Pero también cuando lo que se quiere cambiar es sólo una parte de un componente, la POO nos ayuda, de nuevo gracias al encapsulamiento y los contratos, ya que ese cambio parcial, si es sólo interno y no de comportamiento observable, no debería afectar al resto del sistema.

Casos de estudio

A lo largo del libro vamos a ir desarrollando partes de unos pocos casos de estudio. Esperamos que al hacerlo de esta manera, no haya que explicar un problema nuevo cada vez y podamos concentrarnos en los aspectos centrales de la POO.

Los dos casos que van a guiar la mayor parte de nuestros ejercicios son: una aplicación para jugar al Sudoku y una aplicación que maneja de manera muy simplificada cuentas bancarias.

Sudoku

Sudoku es un juego de mesa de origen japonés. 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 1.1, al comienzo del juego hay algunas 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		

Figura 1.1 Tablero de Sudoku al principio del juego

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

Figura 1.2 Tablero de Sudoku resuelto

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 figura 1.3 a 1.6.

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

Figura 1.3 Nomenclatura de celdas

fila 1
fila 2
fila 3
fila 4
fila 5

fila 6
fila 7
fila 8
fila 9

Figura 1.4 Nomenclatura de filas

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

Figura 1.5 Nomenclatura de columnas

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)

Figura 1.6 Nomenclatura de cajas

Banco simplificado

Puede que algunos de los lectores prefieran un ejemplo más “profesional”, no un juego. Si bien no me gusta esta discriminación de software más o menos profesional, vamos a mostrar otro ejemplo que haga ver que la POO se aplica a mucho más que juegos.

El otro problema sobre el que vamos a ir trabajando es una aplicación que permita hacer ciertas operaciones bancarias: trabajar con cuentas, depositar y extraer dinero de las mismas, transferir dinero entre cuentas, manejar clientes, etc.

Vamos a suponer que una cuenta bancaria es simplemente un repositorio de dinero identificado para el banco por un número y para el conjunto del sistema bancario por una CBU⁶. También, siguiendo la nomenclatura del sistema bancario argentino, denominaremos cajas de ahorro a las cuentas simples, en las cuales sólo se puede extraer dinero previamente depositado, y cuentas corrientes a aquellas cuentas que permiten extraer un monto mayor al saldo, usando una línea de crédito del banco habitualmente denominada giro en descubierto.

Decisiones de lenguajes, herramientas y metodología

Los lenguajes del libro

Los lenguajes elegidos para el libro son Smalltalk y Java, más UML como lenguaje de modelado.

Por supuesto, lo primero que hay que aclarar es que se trata de un libro conceptual, no de lenguajes. Por lo tanto, los lenguajes serán usados como soporte y no se verá ninguno hasta sus más mínimos detalles. Pero si se quiere programar, algún lenguaje hay que usar.

Hemos elegido Smalltalk por varias razones. En primer lugar, como muchos sabrán, es el decano de la POO: en Smalltalk surgieron la mayoría de los conceptos que hoy consideramos fundamentales para la POO⁷. Pero más importante que eso es el hecho de que en Smalltalk sólo se puede programar con POO: **toda construcción de Smalltalk es un objeto o un mensaje, incluyendo las estructuras de control del lenguaje, los ambientes de desarrollo, las herramientas del programador y todo lo que se nos ocurra**. Eso hace que no podamos programar en Smalltalk sin saber objetos, y nos obliga a pensar permanentemente en términos de este paradigma.

La elección de Java es también bastante comprensible. Si bien algunos dirán que Java implementa el paradigma de POO de forma más laxa que Smalltalk, lo cierto es que es realmente difícil programar en Java sin objetos. Es cierto también que Java ha crecido en las últimas versiones, desbordando el propio paradigma de POO, pero en este libro no entraremos en esas características extrañas para evitar que se pierda el foco. La gran ventaja de Java es su enorme difusión, la gran cantidad de herramientas presentes y una comunidad de desarrollo que todavía hoy es la más extendida en el mundo de la POO.

En algunas ocasiones nos referiremos a JavaScript, y hasta mostraremos algo de código. Esto se debe a que JavaScript – un lenguaje también muy difundido y que soporta el paradigma – presenta algunas particularidades en su implementación de POO que lo hacen interesante para mostrar alguno que otro concepto: particularmente, el concepto de objeto en estado puro, sin necesidad de definir clases. Sin embargo, como JavaScript es un lenguaje multiparadigma, no

⁶ CBU es “clave bancaria uniforme”, un número de 22 dígitos que se usa en la Argentina para identificar cualquier cuenta del sistema bancario

⁷ Si bien ya existía Simula cuando se lanzó Smalltalk, el soporte de objetos de Simula era bastante primitivo.

vamos a ahondar mucho más que esto. Self hubiera podido ser otra opción para mostrar un lenguaje sin clases, pero ha salido muy poco del laboratorio en el que fue concebido.

También mostraremos algo de C# cuando algún concepto lo necesite. Este es otro lenguaje muy popular, parecido a Java en sintaxis, pero que tiene algunas diferencias interesantes de mostrar en ciertos casos.

Estamos dejando de lado a muchos otros lenguajes buenos y exitosos, pero la idea es hacer un libro ameno, no excesivamente largo, y eso hace que prefiramos mantener el foco.

Particularmente C++, el lenguaje que más propició la transición del paradigma estructurado a la POO, es cada vez más un lenguaje centrado en cuestiones de desempeño, portabilidad y buenas abstracciones, pero soporta varios paradigmas en forma simultánea y sería difícil de analizar en un libro centrado exclusivamente en POO.

Ruby es también un lenguaje con una imponente comunidad y con buen soporte de POO. Python, que cada vez es más usado como lenguaje para iniciar la programación, también soporta el paradigma. Pero, de nuevo: todo no se puede.

Por qué UML

La elección de UML como lenguaje de modelado tiene que ver con dos cuestiones: que es aceptablemente bueno para modelar objetos, interacciones, clases y otros conceptos del paradigma, a la vez que se ha convertido en un estándar de facto de la comunidad de desarrollo.

El único argumento en contra de UML podría ser su aumento de complejidad en sus últimas versiones. Pero en este libro vamos a usar UML solamente en las cuestiones que más interesan para explicar POO, evitando entrar en complejidades sin sentido para nosotros.

Los entornos de desarrollo

Al trabajar en Java hemos usado Eclipse como entorno integrado de desarrollo (IDE⁸). La verdad es que hay muchos y muy buenos IDE para Java. La razón para elegir Eclipse tiene que ver más que nada con la familiaridad del autor con el mismo, y con que, como varios más, provee herramientas integradas o adicionables para pruebas unitarias, refactorización, cobertura de código, control de versiones, y tantas más. Además, la comunidad que existe detrás de Eclipse es una garantía de pervivencia en el tiempo y de mantenimiento de sus mejores cualidades.

En el caso de Smalltalk, las distintas distribuciones incluyen todo: entorno de desarrollo, compilador, entorno de ejecución, máquina virtual y varias cosas más. La elección de Pharo, por lo tanto, no incluyó solamente las bondades del IDE, que son innegables, sino también lo que vimos como ventajas del ambiente Pharo en su totalidad. Pharo, entonces, fue elegido por su facilidad de uso para el principiante, por su comunidad de colaboradores y su aspecto moderno y cómodo.

Prácticas ágiles

Además de presentar la programación orientada a objetos, y algunos lenguajes de programación y modelado que la soportan, el libro se detiene en algunas prácticas metodológicas provenientes de los métodos ágiles: TDD, refactorización, integración continua y demás.

⁸ IDE es el acrónimo de Integrated Development Environment, que significa entorno integrado de desarrollo.

Las razones para incluir prácticas metodológicas en un libro de programación tienen que ver con la necesidad de detenerse no sólo en lo técnico, sino en enfoques que llevan a construir mejor programas orientados a objetos. La elección de prácticas ágiles por sobre otras tuvo que ver con que el autor es un creyente en la agilidad y con que la agilidad misma ha demostrado ya una ubicuidad y difusión que hacen difícil pensar hoy en día en mejores formas de desarrollar software.

Recapitulación

En este capítulo, hemos mostrado las opiniones de los profesionales de desarrollo de software sobre lo que es la POO y cuáles son sus objetivos. También hemos explicado las razones que llevan a utilizar el paradigma y presentamos algunos casos de estudio que vamos a utilizar a lo largo de todo el libro. También hemos mencionado los lenguajes, técnicas y herramientas sobre los cuales vamos a trabajar.

En el próximo capítulo nos vamos a centrar en la resolución de problemas según la óptica de POO.

2. Un mundo de objetos

Contexto

Como mostramos en el capítulo anterior, hay poco acuerdo entre los profesionales sobre lo que es la POO. Por eso, en este capítulo vamos a centrarnos en mostrar cómo resuelve los problemas la POO, y por qué eso es diferente del planteo de otros paradigmas. Mientras lo hacemos, también irán surgiendo algunos de los conceptos centrales del mismo.

Luego vamos a hablar de cuestiones de implementación del paradigma en distintos lenguajes, mostrando pequeños fragmentos de código para facilitar la comprensión.

Nuestra primera aproximación a la solución a 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 UML⁹.

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

⁹ UML es el acrónimo en inglés para “Lenguaje Unificado de Modelado”. Es un estándar de modelado de software que analizaremos más en detalle en capítulos posteriores. Ver <http://uml.org/>

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.

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:

- Fijarse si ya hay un número en la celda (si está ocupada)
- Fijarse si el número ya está contenido en la fila en la que está la celda en cuestión
- Fijarse si el número ya está contenido en la columna en la que está la celda en cuestión
- Fijarse si el número ya está contenido en la caja en la que está la celda en cuestión

Si no está en ninguno de las tres, podemos colocarlo en la celda. Si no, no.

Un diagrama que represente esto podría ser el de la Figura 2.1

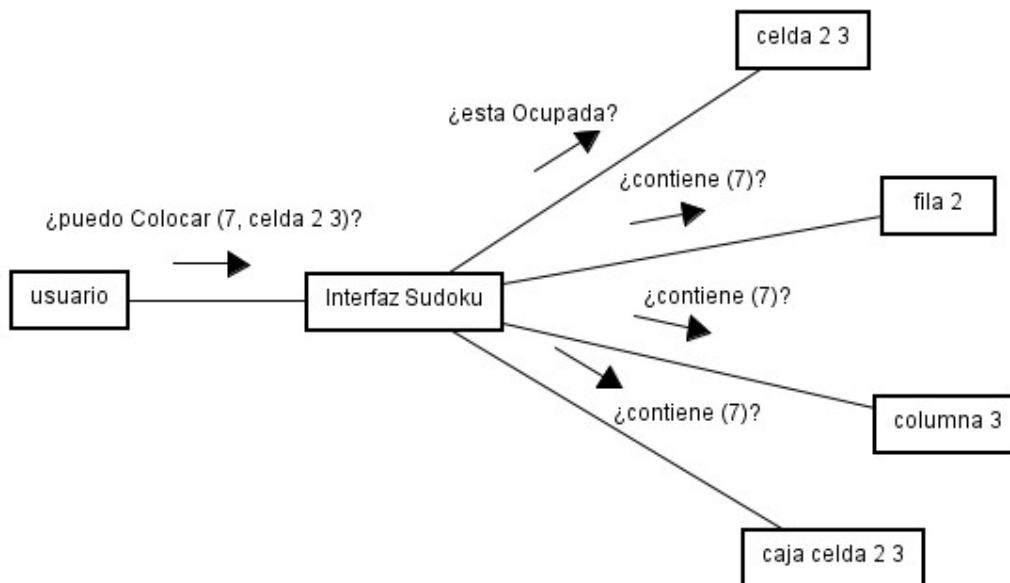


Figura 2.1 Escenario para ver si se puede colocar un número en una celda

Analicemos este diagrama. Allí se muestran 6 objetos, que se indican mediante rectángulos, y

hemos llamado “usuario”, “Interfaz 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 (o, si se quiere, a su interfaz con el usuario), 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 “Interfaz 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, celda23)?”, 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 “Interfaz 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 “Interfaz Sudoku” puede responderle al objeto “usuario” si se puede o no colocar el número 7 allí.

Definición preliminar: objeto

Un objeto es una entidad que puede recibir mensajes, responder a los mismos y enviar mensajes a otros objetos.

Definiciones: mensaje, cliente y receptor

Un mensaje es la interacción entre un objeto que pide un servicio y otro que lo brinda.

El objeto que envía el mensaje se llama objeto cliente y quien recibe el mensaje se llama objeto receptor.

Este mecanismo por el cual el objeto “Interfaz Sudoku”, necesita de la colaboración de otros objetos para poder responder, se denomina **delegación**. Decimos que “Interfaz 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 Interfaz 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 (antes de preguntarles a una fila, columna y caja si contienen el número debo saber a cuál fila, columna y caja preguntarles), así que vamos a indicarlo explícitamente usando un diagrama que deje esto más en claro.

Eso se muestra en el diagrama de la figura 2.2, 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.

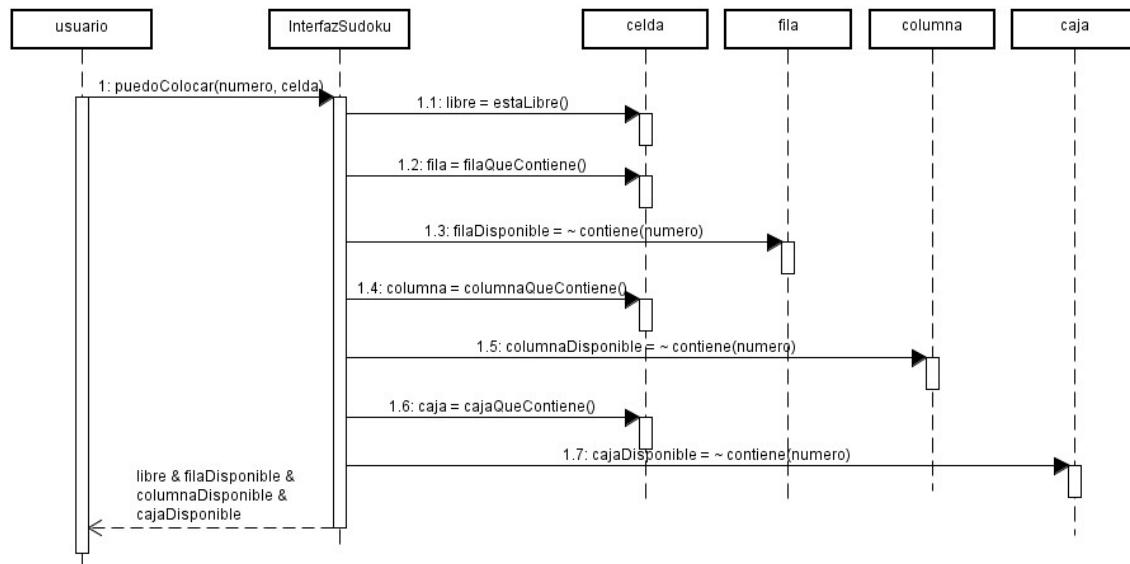


Figura 2.2 Escenario mejorado de la figura 2.1, con indicación de posibles precedencias

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 objeto *InterfazSudoku*, 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 *InterfazSudoku* le responde al *usuario* con la conjunción lógica de *libre*, *filaDisponible*, *columnaDisponible* y *celdaDisponible*, que obtuvo en los pasos anteriores.

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

- *estaLibre*, enviado a la celda: esto sirve para que la celda le indique si está libre o no, y le devuelve el resultado almacenado en la variable *libre*.
- *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 *filaDisponible*. Usamos el símbolo \sim para la negación lógica.
- *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 *columnaDisponible*.
- *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 *cajaDisponible*.
- Finalmente, como dijimos, el resultado enviado al usuario es la conjunción lógica de lo que obtuvo en las variables *libre*, *filaDisponible*, *columnaDisponible* y *celdaDisponible*.

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)*.

Esto nos lleva al diagrama de la figura 2.3.

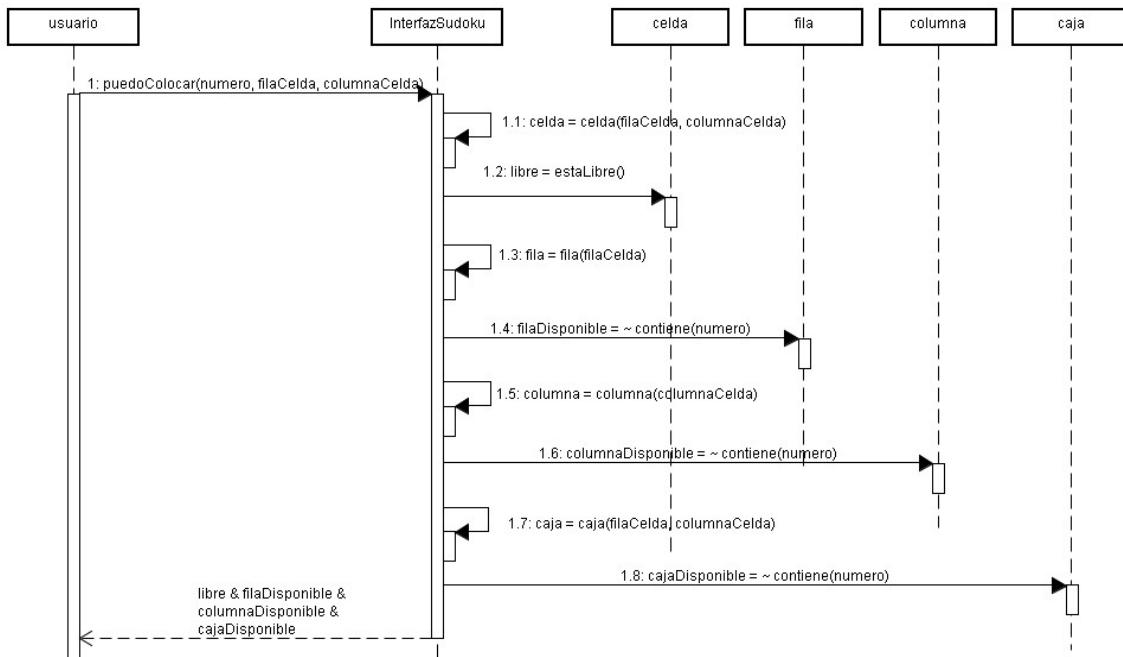


Figura 2.3 Escenario definitivo de la figura 2.2

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 saltar esta parte de la solución, pero tengamos en cuenta que lo que nos falta es implementar los métodos (usamos la notación objeto >> método):

- InterfazSudoku >> puedoColocar (numero, filaCelda, columnaCelda)
- InterfazSudoku >> celda (filaCelda, columnaCelda)
- celda >> estaLibre
- InterfazSudoku >> fila (filaCelda)
- fila >> contiene (numero)
- InterfazSudoku >> columna (columnaCelda)

- columna >> contiene (numero)
- InterfazSudoku >> caja (filaCelda, columnaCelda)
- caja >> contiene (numero)

Pero todo esto, dependiente del lenguaje y de la forma en que éste implemente la POO, lo dejaremos para más adelante: es el tema de los próximos capítulos, aunque algo veremos en este.

Conceptualizando

El comportamiento como aspecto central

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

Definición ¿definitiva?: objeto

Un objeto es una entidad 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.

Definición: identidad

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.

Definición: estado

El estado es la situación en que un objeto se encuentra.

Corolario:

Un objeto puede cambiar su estado a través del tiempo.

Por ejemplo, la *celda 2 3* de nuestro programa de Sudoku se encuentra libre al inicio del juego. Decimos que su estado es estar libre. 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 sí mismo.

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 ese 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, que serían algo como la suma del comportamiento más el manejo de su 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 hace el objeto para brindar ese servicio.

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”¹⁰, 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 que quien implemente la celda pueda elegir varias alternativas e incluso cambiar la implementación 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 *contiene*: 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: polimorfismo

El polimorfismo es la capacidad de respuesta que tienen distintos objetos de responder de maneras diferentes 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, podemos distinguir tres cuestiones:

- Respeto de cómo implementan el comportamiento: lenguajes basados en clases y lenguajes basados en prototipos o ejemplos.
- Respeto 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.
- Respeto de la verificación de tipos: con verificación en tiempo de compilación o con verificación en tiempo de ejecución.

Vamos entonces a analizar estos aspectos, usando tres lenguajes para ello: Smalltalk, Java y

¹⁰ En inglés, significa algo como “Pida, no me ordene”. Como la traducción no resulta clara, deberíamos enunciarlo más bien como “Dígame qué hacer, no cómo hacerlo”.

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 compilán, 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 ejecución¹¹.

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 solamente el mensaje *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 Biología, 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, en Biología dirímos que *Lassie* es una instancia de la especie *Perro* porque *Perro* es el tipo de animal que es *Lassie*. Dirímos 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 como¹²:

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

Y en java podría ser¹³:

¹¹ Esta caracterización de JavaScript puede ser polémica para algunos. No obstante, vamos a volver sobre ella más adelante. Por el momento, es una definición suficientemente exacta.

¹² Ver sintaxis de Smalltalk en el apéndice A.

¹³ Ver sintaxis de Java en el apéndice A.

```
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 propiedad¹⁴, 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 (o variable de instancia)

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: instantiación

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*:

```
celda23 := Celda new.
```

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

```
Celda celda23 = 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 *celda23*. Hay pequeñas diferencias entre ambos lenguajes, pero por el momento no son importantes.

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 así¹⁵:

```
var celda23 = {  
    fila: undefined,  
    columna: undefined,  
    numero: undefined,
```

¹⁴ En este libro vamos a evitar equívocos, llamándolas siempre atributos.

¹⁵ Ver sintaxis de JavaScript en el apéndice A.

```
estaLibre: function() {
    return this.numero === undefined;
}
};
```

Nótese que no necesitamos tener clases para definir objetos, sino que en el mismo objeto se definen atributos y métodos.

Eso es lo central de este modelo. Pero, ¿qu'pasaría si necesitamos más objetos con el mismo comportamiento? En ese caso, hay una solución, pero cuidado con esto: no siempre necesitamos más de un objeto con el mismo comportamiento: los casos del tablero del Sudoku o la misma interfaz del programa son objetos únicos en su tipo.

Ahora bien, es cierto que es bastante común que haya objetos con comportamientos similares. En este caso, estos lenguajes usan la noción de **prototipo**.

Por ejemplo, si quisiésemos otro objeto similar a *celda23*, podemos hacerlo así:

```
celda57 = object.create(celda23);
```

Es decir, primero creamos un objeto, referenciado por la variable *celda23*, y luego creamos una copia del mismo, que queda referenciada por la variable *celda57*. Decimos que *celda23* funciona como prototipo para *celda57*. El método *estaLibre* queda definido para cualquier otro objeto que use el mismo prototipo.

Como *celda57* está definida como una copia del prototipo, su comportamiento es el mismo, y cada método que agreguemos al prototipo se agrega a *celda57*. Por supuesto, esto no se cumple a la inversa: *celda57* podría tener comportamiento adicional al del prototipo (cosa que en este caso no nos es útil).

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:

```
1 caja := Caja new.
```

```

2 fila := caja filaQueContiene.
3 celda := Celda new.
4 celda := caja.
5 libre := celda estaLibre.

```

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 *Celda* y se guarda la referencia en la variable *celda*.

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 *Celda*. 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 *Celda* 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 mensajes 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 sería¹⁶:

```
1 Caja caja;
2 Fila fila;
3 Celda celda;
4 caja = new Caja();
5 fila = caja.filaQueContiene();
6 celda = new Celda();
7 celda = caja;
8 celda.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 el que descubre que le estamos enviando un mensaje a un objeto referenciado desde la clase *Caja* que la clase *Caja* no comprende. 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 *Celda* y aloja su referencia en una variable cuyo tipo también es *Celda*.

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 *Celda*.

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 *celda* hay una referencia a una instancia de *Celda* (en ningún momento pudimos poner otra cosa), es válido que envíemos el mensaje *depositar*, que entienden los objetos de la clase *Celda*.

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 envíemos mensajes a objetos que no los van a comprender. Esto hace que no haya mensajes no comprendidos en tiempo de ejecución.

¹⁶ Por favor, si intentas probar este código, ignora las advertencias (warnings) del compilador y concéntrate sólo en los errores.

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.

Un vistazo a la implementación

Éste es un libro de programación. Sin embargo, hace rato que venimos abrumando al lector con conceptos y definiciones. Por eso, en este apartado, vamos a tratar de mostrar un poco de código para ver cómo se traducen estos conceptos en la implementación del comportamiento.

Vamos a analizar la implementación de algún método de nuestro Sudoku. Dejamos toda sofisticación para más adelante, así como el resto de la implementación, ya que necesitamos detenernos en cuestiones metodológicas en el próximo capítulo. Sólo estamos intentando mostrar un poco de código para entender cómo se ve en cada caso y cómo se implementan los objetos en distintos lenguajes.

Lo haremos en tres lenguajes: JavaScript, Smalltalk y Java.

Implementando un objeto en JavaScript

El código JavaScript para implementar la respuesta al mensaje *puedoColocar* para el objeto *InterfazSudoku* sería:

```
var InterfazSudoku = {
    puedoColocar: function(numero, filaCelda, columnaCelda) {
        var celda = this.celdaTablero(filaCelda, columnaCelda);
        var libre = celda.estaLibre();
        var fila = this.filaTablero(filaCelda);
        var filaDisponible = !fila.contiene(numero);
        var columna = this.columnaTablero(columnaCelda);
        var columnaDisponible = !columna.contiene(numero);
    }
}
```

```

        var caja = this.cajaTablero(filaCelda, columnaCelda);
        var cajaDisponible = caja.contiene (numero);
        return libre && filaDisponible && columnaDisponible &&
cajaDisponible;
    }
};


```

Por supuesto, en el código anterior falta definir mucho del objeto *InterfazSudoku*: sólo mostramos la implementación de *puedoColocar*.

Implementando una clase en Smalltalk

En Smalltalk¹⁷, dependiendo de la implementación particular que usemos, el código se irá escribiendo de maneras distintas. Habitualmente el código se va insertando en el entorno de desarrollo.

En Pharo, que es la implementación de Smalltalk que vamos a usar en este libro, hay que abrir un *System Browser*, que en la versión que estamos usando se llama *Nautilus*, el cual tiene la apariencia como el de la figura 2.4.

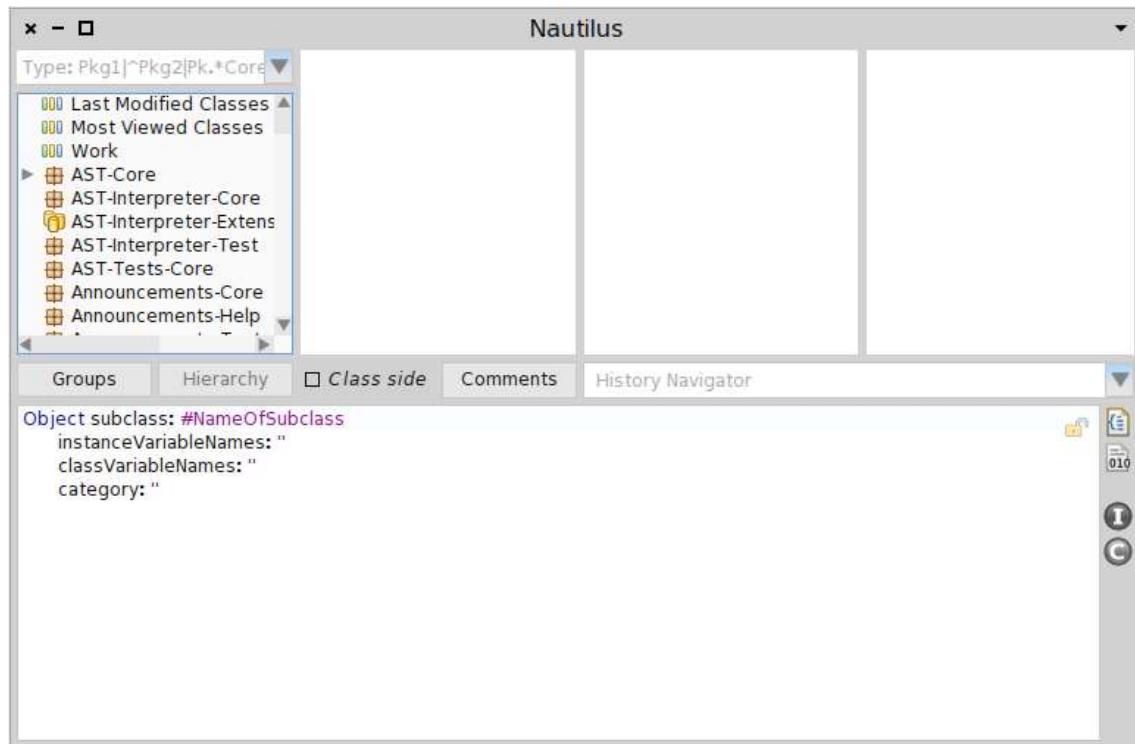


Figura 2.4 System Browser de Pharo (Nautilus)

En el recuadro de más abajo, en el que dice “Object subclass: #NameOfSubclass”, debemos reemplazar “NameOfSubclass” por “Celda”, que es el nombre de la clase que deseamos crear. Más abajo, a la derecha de donde dice “instanceVariableNames” y figura un ‘,’ pondremos ‘contenido’, que es el atributo que queremos definir.

A continuación, grabamos la clase *Celda* y nos encontraremos con la pantalla de la figura 2.5.

¹⁷ Ver sintaxis de Smalltalk en el apéndice A.

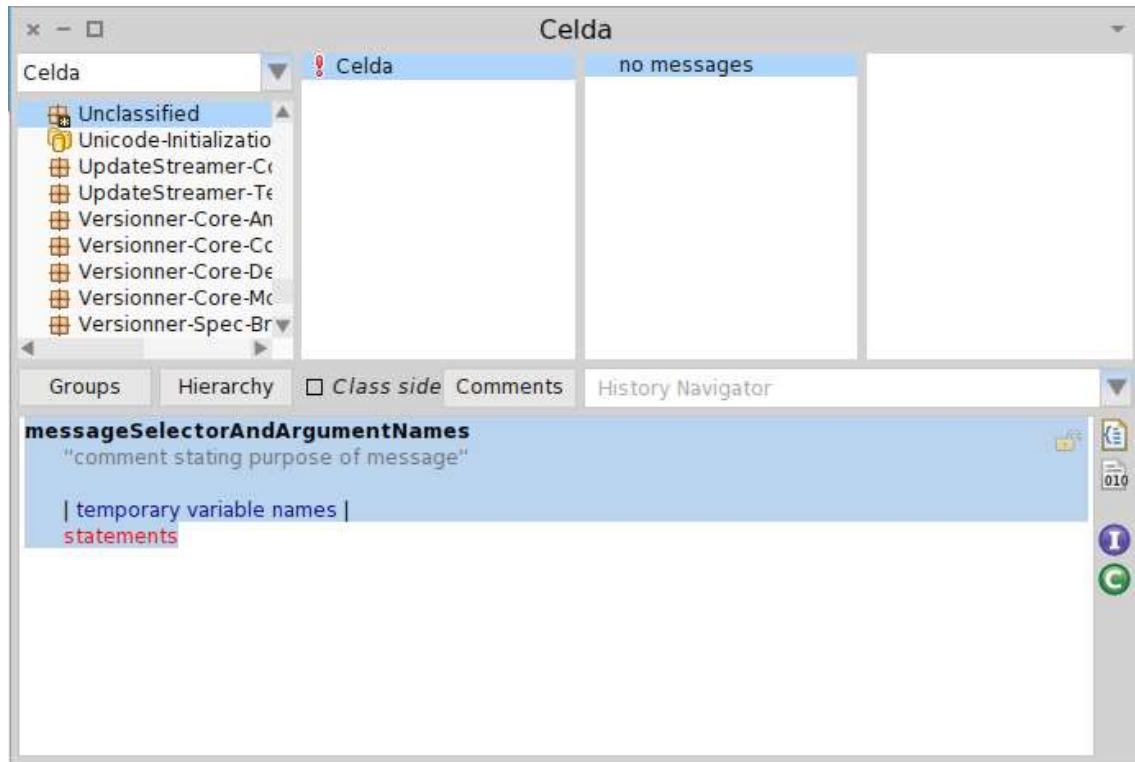


Figura 2.5: Pharo listo para que escribamos un método

Allí vemos el tercer recuadro de arriba en blanco, con la leyenda “no messages”, y si lo seleccionamos, en el recuadro inferior se nos abre una plantilla de implementación de mensajes. En la misma, debemos cambiar “messageSelectorAndArgumentNames” por el nombre de nuestro método, y escribir el código debajo.

Vamos a hacer el procedimiento 3 veces, con los tres métodos que siguen:

```
initialize
    self contenido := 0.
```

```
inicializarConValor: numero
    self contenido := numero.
```

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

De todas maneras, lo que hicimos recién no es la clase completa, sino una primera aproximación a la implementación del comportamiento de respuesta al mensaje *estaLibre*.

También puede resultar de interés ver cómo implementar la respuesta al mensaje *puedoColocar* en la clase *Tablero*. Para ello, debemos crear la clase y luego escribir el método:

```
puedoColocar: numero enFila: filaCelda enColumna: columnaCelda
    | celda libre fila columna caja filaDisponible columnaDisponible
    cajaDisponible |
        celda := self celdaFila: filaCelda celdaColumna: columnaCelda.
        libre := celda estaLibre.
        fila := self filaTablero: filaCelda.
```

```

filaDisponible := (fila contiene: numero) not.
columna := self columnaTablero: columnaCelda.
columnaDisponible := (columna contiene: numero) not.
caja := self cajaTableroFila: filaCelda columna: columnaCelda.
cajaDisponible := (caja contiene: numero) not.
^ ( libre & filaDisponible & columnaDisponible & cajaDisponible).

```

Implementando una clase en Java

En Java¹⁸, la clase *Celda* se define en un único archivo de texto de código fuente, cuyo nombre es *Celda.java*. Como ya hemos dicho, lo que sigue no es la clase completa, sino una primera aproximación a la implementación del comportamiento de respuesta al mensaje *estaLibre*.

```

public class Celda {
    private int contenido;
    public Celda () {
        contenido = 0;
    }
    public Celda (int numero) {
        contenido = numero;
    }
    public boolean estaLibre () {
        return contenido == 0;
    }
}

```

El mensaje *puedoColocar(numero, celda)* de la clase *Tablero* podría implementarse así:

```

public class Tablero {
    public boolean puedoColocar (int numero, int filaCelda, int
columnaCelda) {
        Celda celda = celda (filaCelda, columnaCelda);
        if ( ! celda.estaLibre ( ) ) return false;
        Fila fila = fila (filaCelda);
        boolean filaDisponible = !fila.contiene(numero);
        Columna columna = columna (columnaCelda);
        boolean columnaDisponible = !columna.contiene(numero);
        Caja caja = caja (filaCelda, columnaCelda);
        boolean cajaDisponible = !caja.contiene(numero);
        return ( filaDisponible && columnaDisponible && cajaDisponible)
    }
}

```

Ejercicio adicional

Para ver un caso más de resolución de problemas con objetos, analicemos cómo sería la resolución del problema apertura de una cuenta bancaria, con la eventual creación de un cliente nuevo si éste no fuera ya cliente del banco. Estamos trabajando con nuestro segundo caso de

¹⁸ Ver sintaxis de Java en el apéndice A.

estudio, tal como se definió en el capítulo introductorio.

Los objetos que surgen en este escenario son:

- La persona que desea abrir la cuenta.
- La nueva cuenta.
- El cliente a crearse a partir de la persona (es decir, la persona que no es cliente se convertiría en cliente en este escenario).
- La base de datos de clientes (para verificar si la persona ya es cliente).

Un diagrama de secuencia de la creación de la cuenta podría ser el de la figura 2.6.

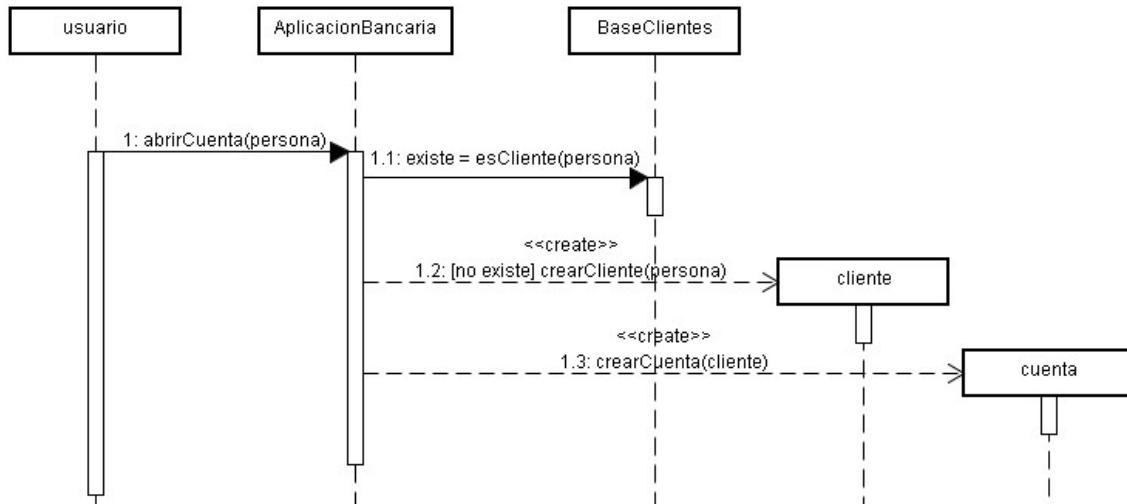


Figura 2.6 Escenario de creación de una cuenta bancaria

Un escenario adicional podría ser la transferencia de dinero entre dos cuentas del mismo banco. En este caso, los objetos que intervienen son:

- La base de datos de cuentas.
- La cuenta a debitar.
- La cuenta a acreditar.

El diagrama de secuencia del escenario podría ser el de la figura 2.7.

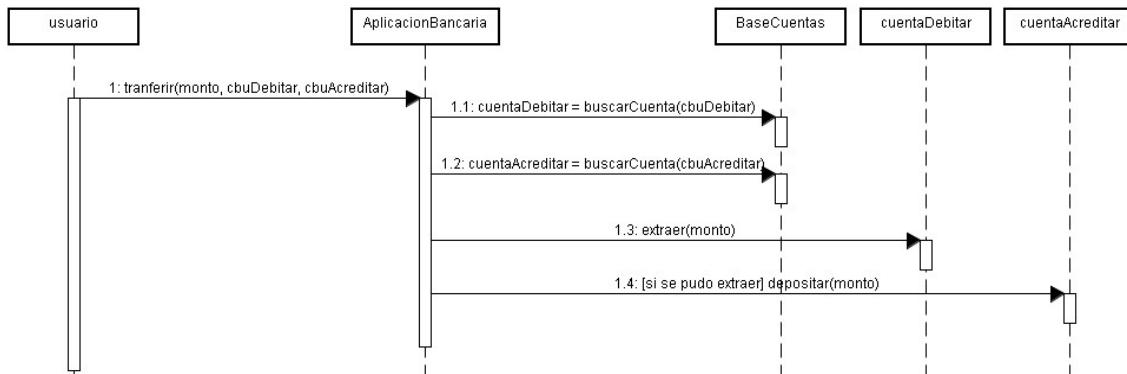


Figura 2.7 Escenario de transferencia entre cuentas

Ejercicios propuestos

- 1) En una sucursal de una empresa que cobra servicios públicos, se quieren establecer las siguientes reglas para la atención:
 - Si hay algún mostrador libre, el cliente que llega es atendido por cualquier mostrador que esté en esta situación.
 - Si hay gente esperando y se libera un mostrador, los clientes especiales (discapacitados, embarazadas y ancianos) tienen prioridad, en el orden de llegada.
 - Si no, se atienden en el orden de llegada.

Hacer el diagrama de secuencia, que modele es escenario para determinar cuál es el cliente a ser atendido en un momento dado, manejando una cola común y otra para los clientes especiales.

Contenido incompleto en esta versión del libro.

Recapitulación

En este capítulo hemos visto la manera en que se resuelven los problemas usando POO, con objetos y mensajes. Hemos introducido también algunos conceptos centrales. También hemos visto que los diferentes lenguajes tienen aproximaciones diferentes en cuanto a la implementación del paradigma, y hemos visto un poco de código.

En el próximo capítulo nos vamos a centrar en cómo llevar a cabo la implementación del comportamiento de los objetos, con un enfoque consistente y completo.

3. Paréntesis metodológico: diseño por contrato y un procedimiento constructivo

Contexto

Hasta aquí hemos mostrado rudimentos de la POO. Pero no podemos avanzar hacia la implementación sin antes tener un enfoque metodológico que nos permita hacerlo. Por eso, este capítulo va a centrarse en un método de implementación del comportamiento, basado en la idea de contratos y de desarrollar en base a pruebas automatizadas. Veremos también algunas secuelas virtuosas de trabajar de esta manera.

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 escribimos¹⁹:

- interfazSudoku >> puedoColocar (numero, filaCelda, columnaCelda)
- interfazSudoku >> celda (filaCelda, columnaCelda)
- celda >> estaLibre

¹⁹ Esta es una forma de definir las firmas que pretende ser independiente del lenguaje. En Smalltalk los métodos y los parámetros no tienen tipo, mientras que en Java deberían agregarse el tipo devuelto y los tipos de los parámetros, cosa que en Smalltalk no tendría sentido.

- interfazSudoku >> fila (filaCelda)
- fila >> contiene (numero)
- interfazSudoku >> columna (columnaCelda)
- columna >> contiene (numero)
- interfazSudoku >> caja (filaCelda, columnaCelda)
- caja >> contiene (numero)

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 la interfaz del programa le envíe el mensaje *estaLibre* al objeto referenciado por *celda*, debe cumplirse, como precondición, que *celda* referencia 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 el 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 la interfaz del programa le envía el mensaje *estaLibre* al objeto referenciado por *celda*, se espera que el objeto referenciado por *celda* actúe haciendo cumplir

las siguientes postcondiciones²⁰:

- 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* 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 3.1²¹, aplicándolo para cada escenario que vayamos encontrando en nuestro problema:

²⁰ Elegimos empezar a escribir las postcondiciones por los casos excepcionales, dejando para el final el camino feliz. Esto es una costumbre propia, que pongo el foco primero en lo que puede fallar, de modo tal de no olvidarme después. Probablemente haya lectores que prefieran hacerlo al revés. Pero también es casi seguro que, al llevar a código en algún lenguaje, primero se chequen las precondiciones y luego se siga con el camino feliz.

²¹ Este es un diagrama de actividades de UML.

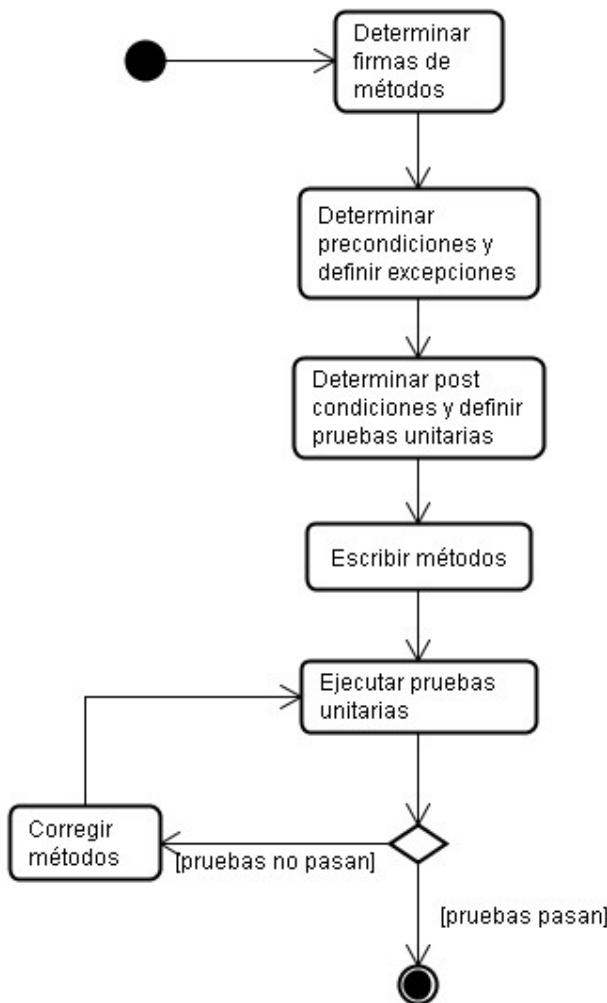


Figura 3.1 Diseño por contrato

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 3.2 refina el procedimiento, definiendo de a un método por vez.

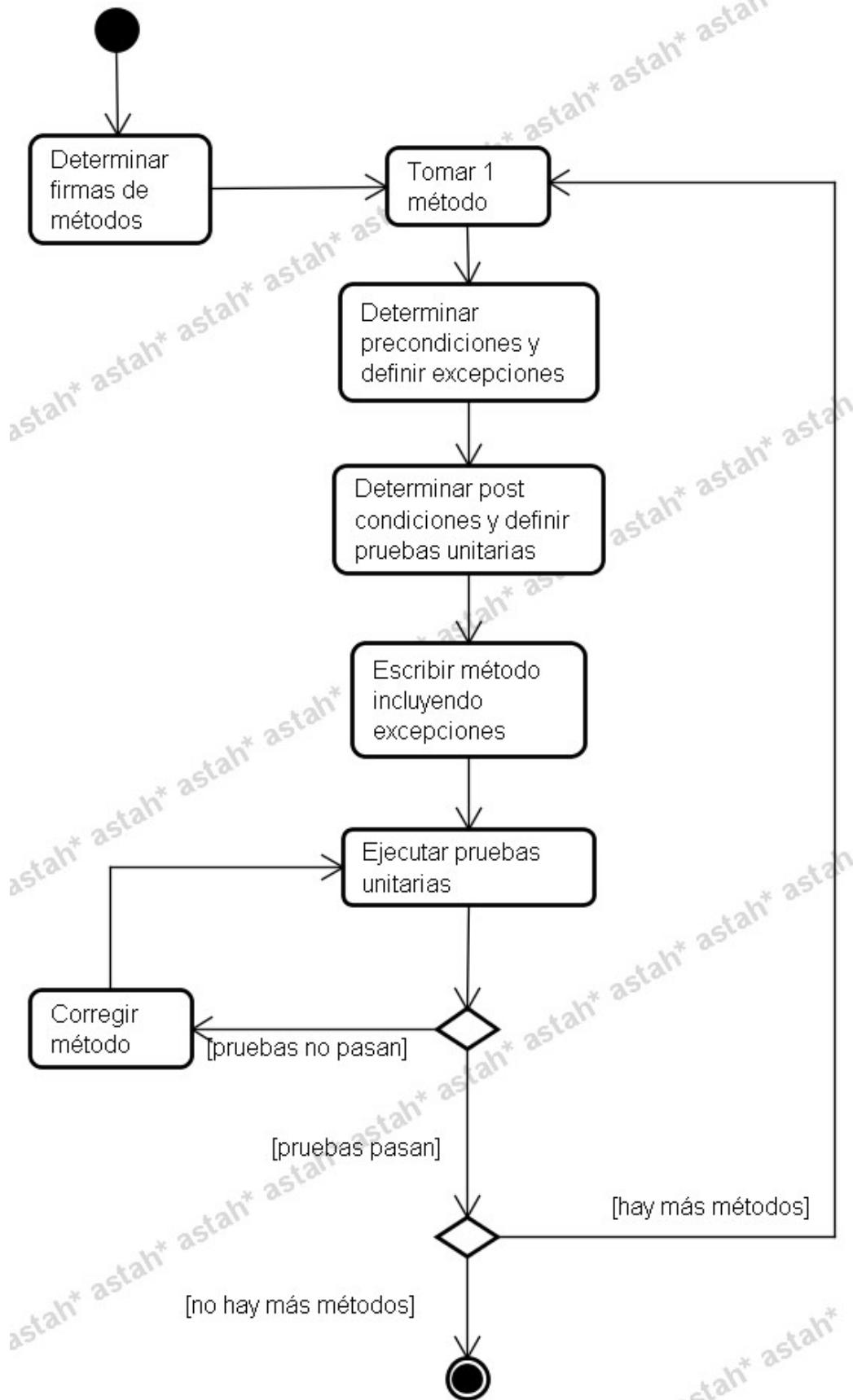


Figura 3.2 Diseño por contrato de a un método a la vez

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 3.3.

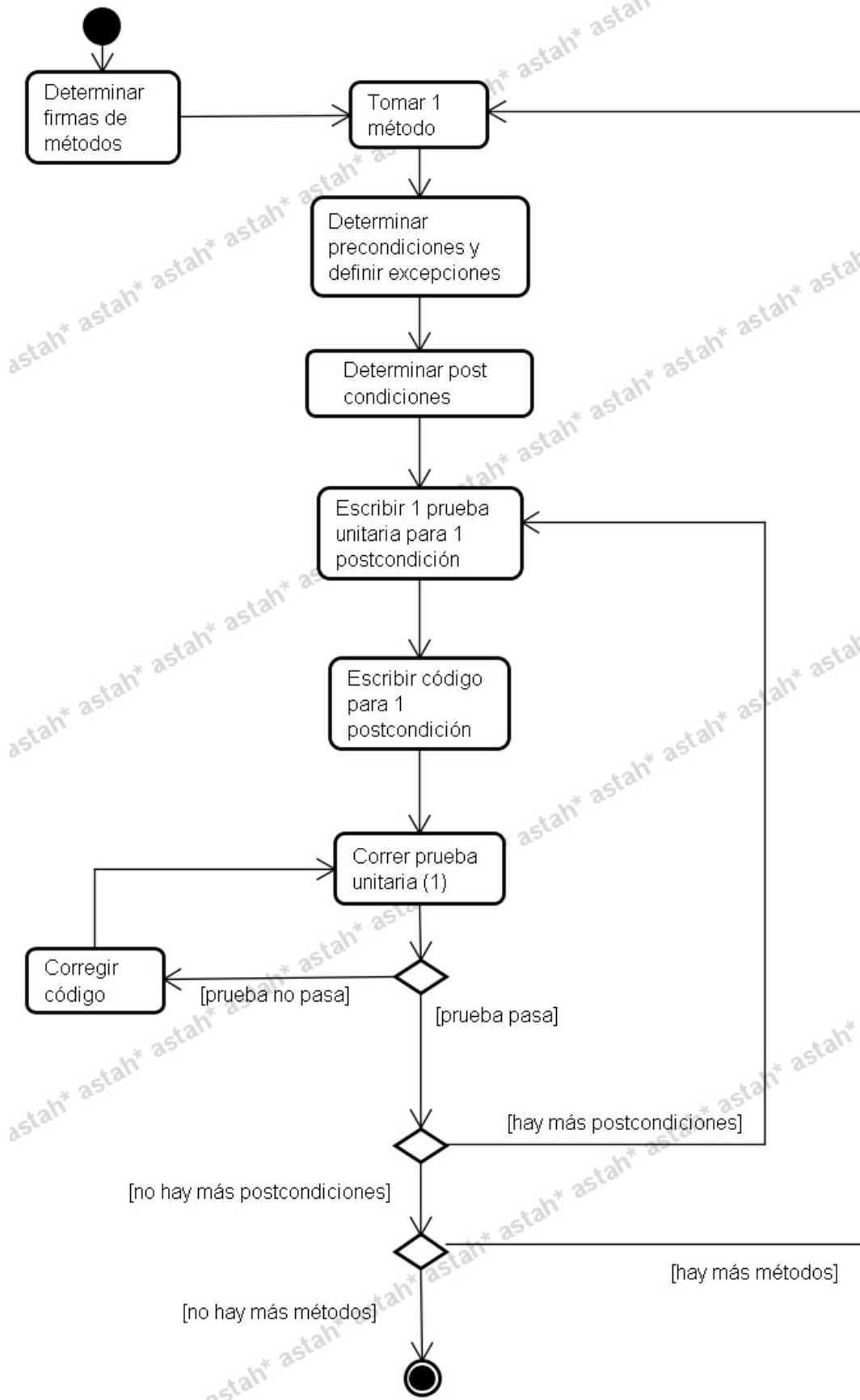


Figura 3.3 Diseño por contrato de a una postcondición a la vez

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 un método, 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. Y lo haremos primero en Smalltalk.

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* referencia 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.

Atención: 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*: 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?

Atención: asegúrese de que entiende lo que le dice el programa y por qué lo dice.

Atención: 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.

Stop. 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. Así no vamos a poder compilar (Pharo no conoce a `libre`), por lo que antes de seguir agregamos a `libre` como 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 colocarNúmero: 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 *colocarNúmero*. 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 *colocarNúmero* 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 colocarNúmero: (-2)]
        on: ValorInvalido
        do: [:e | Transcript show:'La prueba pasó: se pasó un valor negativo y colocarNúmero lanzó ValorInvalido'; cr. ^nil].
    Transcript show:'La prueba NO pasó: se pasó un valor negativo y colocarNúmero NO lanzó ValorInvalido'; cr
```

Es decir, el método empieza creando una celda y enviando el mensaje *colocarNúmero* 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.

Atención:

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 *colocarNúmero* ni la clase *ValorInvalido*.

Para que desaparezca el primer error de compilación, creamos un método *colocarNúmero* vacío:

```
colocarNúmero: valor
```

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

Importante:

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.

Recomendación:

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. No 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 XP²², vamos también a dejarla de lado.

Nos quedamos con la tercera opción. Para hacer nuestra metodología más consistente, borramos

²² XP o Extreme Programming (la traducción más habitual en castellano es “Programación extrema”) es un conjunto de prácticas metodológicas de desarrollo de software. Entre las recomendaciones de XP está la de no agregar nada que no se necesite por el momento. En un capítulo posterior veremos más sobre XP.

la prueba que comprobaba el lanzamiento de la excepción y dejamos al método *colocarNúmero* sólo con lo que necesitamos. También, como una decisión de consistencia de nomenclatura, vamos a poner un nombre distinto a *colocarNúmero*, 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 *colocarNúmero*.

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 constructor (*initialize*) y un método *estaLibre*.

Recomendación:

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.

Definición: pruebas de regresión

Una prueba de regresión es cualquier prueba que se ejecuta luego de un cambio para comprobar que dicho cambio no haya afectado partes del programa que debieran seguir funcionando bien.

Pruebas unitarias... ¿pruebas?

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 escribir las 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 regresión 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 automática 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 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 en código y escritas antes de la implementación – no es una ocurrencia mía como autor del libro. Fue propuesta formalmente por primera vez²³ por Kent Beck dentro del marco de una práctica metodológica llamada TDD (desarrollo guiado por las pruebas o, en inglés, Test-Driven Development) [Beck 2000].

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

²³ Esto está discutido, y hay quienes dicen que Beck “redescubrió” una práctica existente desde 1957 (en el libro “Digital Computer Programming”, de D.D. McCracken).

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 ejemplos 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.

Recomendación: 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 en código 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 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 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.

Esto hace que las pruebas unitarias 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.

Algunas recomendaciones

Los métodos de consulta y modificación de propiedades: ¿hay que probarlos?

Como ya vimos, los objetos suelen tener estado, y cuando necesitamos acceder al estado, solemos usar métodos consulta o de modificación del estado. Es lo que Pharo llama “accesores” y en los ambientes de Java se suelen llamar “getters” y “setters”²⁴.

Es allí que surge la pregunta: estos métodos, ¿deberían tener pruebas unitarias o no?

Piense un poco antes de seguir leyendo...

Bueno, ocurre que la pregunta es engañosa. Si estamos siguiendo el procedimiento descripto en este capítulo, deberíamos escribir pruebas que evidencien comportamiento, no estado. Por lo tanto, la respuesta rápida es un rotundo no.

Sin embargo, a veces la consulta del estado, o de parte del mismo, o su modificación, pueden ser parte del comportamiento del objeto. En ese caso, sí escribiríamos dichas pruebas.

Por lo tanto, la regla de oro es: escribimos pruebas unitarias sobre el comportamiento de los objetos. Si la consulta del estado es parte de su comportamiento, tendrá pruebas que lo evidencien. De la misma manera, si la modificación del estado es parte del comportamiento, también habrá pruebas para ese comportamiento.

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 constructores como métodos especiales que se invocan al crear un objeto – eso es lo que son en definitiva – también los constructores 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 constructor 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 constructor. 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: colecciónCeldas.
```

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

²⁴ Estos nombres provienen de que los métodos en cuestión suelen empezar con las palabras “get” o “set”. Por ejemplo, si debo poder acceder al número contenido en una celda, podrían llamarse “getNúmero” y “setNúmero”.

```

Fila >> inicializarConCeldas : colecciónCeldas
      self celdas := colecciónCeldas

```

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.

Ejercicio adicional

Como ejercicio adicional, vamos a seguir con nuestra aplicación bancaria, implementando el método *depositar* de una clase *Cuenta*. Para ver otro lenguaje, en este caso lo haremos en Java.

Las precondiciones de *depositar* son:

- *cuenta* debe referenciar un objeto existente
- El monto a depositar debe ser un número mayor que 0

Igual que pasó en Smalltalk, la primera precondición la chequea el entorno de ejecución y envía una excepción *NullPointerException* si no se cumple, así que no la tendremos en cuenta.

Las postcondiciones son:

- Si el monto a depositar no es un número mayor que 0, lanzaremos una excepción *MontoInvalido*
- Si las precondiciones se cumplen (camino feliz), el saldo de la cuenta debe verse incrementado en el monto pasado como argumento

Vamos entonces a ir escribiendo de a una prueba a la vez, y acompañando con el código respectivo en la clase *Cuenta*.

Como hicimos en Smalltalk, vamos a definir una clase de pruebas. En ella iremos volcando los métodos de prueba, y tendremos un método *main* desde el cual los iremos invocando. La clase de pruebas vacía será:

```

package carlosFontela.aplicacionBancaria.pruebas;
public class PruebasCuenta {
    public static void main (String[ ] parametros) {
    }
}

```

Lo que escribimos es la notación Java para un programa ejecutable: una clase (dentro de algún paquete) que contenga un método “estático” cuyo nombre sea *main* y con un arreglo de *String* como parámetro. O sea, por ahora no hicimos nada productivo.

Lo primero que deberíamos hacer, entonces, es agregar, dentro de *main*, la llamada a un método que chequee la primera postcondición: que se lance una excepción *MontoInvalido* si el monto pasado como parámetro es 0 o negativo. En el método *main* escribimos:

```

public static void main (String[ ] parametros) {
    new PruebasCuenta().siMontoEsNegativoDebeLanzarExcepcion();
}

```

Es decir, creamos un objeto de prueba y llamamos a un método que aún no hemos definido. El método en cuestión sería:

```

private void siMontoEsNegativoDebeLanzarExcepcion() {
    Cuenta cuenta = new Cuenta();
}

```

```

try {
    cuenta.depositar(-200);
}
catch (MontoInvalido e) {
    System.out.println("La prueba pasó: se pasó un monto negativo y
lanzó MontoInvalido");
    return;
}
System.out.println("La prueba NO pasó: se pasó un monto negativo pero
no lanzó MontoInvalido");
}

```

Es decir, el método empieza creando una cuenta y enviando el mensaje *depositar* con argumento -200, 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.

Atención:

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

Si compilamos la clase *PruebasCuenta*, vamos a encontrarnos con dos errores, que indican que no existen las clases *Cuenta* ni *MontoInvalido*.

En primer lugar creamos la clase *Cuenta*, vacía, con lo cual ya eliminamos el error. Pero surge un error nuevo: la clase *Cuenta* no tiene un método *depositar*. Lo creamos así dentro de la clase:

```

package carlosFontela.aplicacionBancaria;
public class Cuenta {
    public void depositar(int monto) {
    }
}

```

Todavía persiste el error de que no existe la clase *MontoInvalido*.

Importante:

En POO las excepciones son objetos, como ya dijimos. En Java, en particular, los objetos de excepción deben ser instancias de una clase que derive de *Exception* o *RuntimeException*.

A continuación creamos la clase de excepción:

```

package carlosFontela.aplicacionBancaria;
public class MontoInvalido extends RuntimeException {
}

```

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 la clase *Cuenta*: notemos que el método *depositar* está vacío. Por lo tanto, ¿qué debería ocurrir si corremos la prueba? Sí: la prueba debe fallar.

Veamos. Ejecutamos el programa de prueba y vemos en la consola:

```
La prueba NO pasó: se pasó un monto negativo pero no lanzó MontoInvalido
```

Escribamos ahora el código del método *depositar* de modo tal que arroje la excepción que esperamos:

```

public void depositar(int monto) {
    if (monto < 0)
        throw new MontoInvalido();
}

```

Ahora sí, al correr el programa, vemos en la consola:

```
La prueba pasó: se pasó un monto negativo y lanzó MontoInvalido
```

Sin embargo, no hemos comprobado qué pasa si el monto es cero. Si agregamos una prueba que lo chequee y la invocamos desde *main*:

```

private void siMontoEsCeroDebeLanzarExcepcion() {
    Cuenta cuenta = new Cuenta();
    try {
        cuenta.depositar(0);
    }
    catch (MontoInvalido e) {
        System.out.println("La prueba pasó: se pasó un monto 0 y lanzó
MontoInvalido");
        return;
    }
    System.out.println("La prueba NO pasó: se pasó un monto 0 pero no lanzó
MontoInvalido");
}

```

Al correr el programa obtenemos de nuevo el resultado de una prueba fallando:

```
La prueba pasó: se pasó un monto negativo y lanzó MontoInvalido
La prueba NO pasó: se pasó un monto 0 pero no lanzó MontoInvalido
```

Lo que nos falta es modificar el método *depositar* para que compruebe que el monto no sea cero:

```

public void depositar(int monto) {
    if (monto <= 0)
        throw new MontoInvalido();
}

```

Ahora sí, al correr el programa de prueba obtenemos dos salidas correctas en la consola:

```
La prueba pasó: se pasó un monto negativo y lanzó MontoInvalido
La prueba pasó: se pasó un monto 0 y lanzó MontoInvalido
```

Con lo que hicimos podemos dar por satisfecha la primera postcondición. Veamos ahora la segunda: si la precondición se cumple, el saldo de la cuenta debe verse incrementado en el monto pasado como argumento.

El método *main* de *PruebasCuenta* quedaría:

```

public static void main (String[ ] parametros) {
    new PruebasCuenta().siMontoEsNegativoDebeLanzarExcepcion();
    new PruebasCuenta().siMontoEsCeroDebeLanzarExcepcion();
    new PruebasCuenta().alDepositarSaldoDebeIncrementarseEnMonto();
}

```

Y el método *alDepositarSaldoDebeIncrementarseEnMonto* quedaría:

```
private void alDepositarSaldoDebeIncrementarseEnMonto() {
```

```

Cuenta cuenta = new Cuenta(); // el saldo inicial es 0
cuenta.depositar(100);
if (cuenta.getSaldo() == 100)
    System.out.println("La prueba pasó: el saldo se incrementó
correctamente al depositar");
else
    System.out.println("La prueba NO pasó: el saldo no se incrementó
correctamente al depositar");
}

```

Lo primero que ocurre es que no existe el método `getSaldo` en la clase `Cuenta`, lo cual hace que el programa no compile. Lo mínimo que tenemos que hacer para que compile es agregar un atributo `saldo` y el método que nos da su valor:

```

private int saldo;
public int getSaldo () {
    return this.saldo;
}

```

Ahora compila. Si corremos la prueba debería fallar (ya queda claro por qué, ¿no?). Efectivamente, en la consola leemos:

```

La prueba pasó: se pasó un monto negativo y lanzó MontoInvalido
La prueba pasó: se pasó un monto 0 y lanzó MontoInvalido
La prueba NO pasó: el saldo no se incrementó correctamente al depositar

```

Para hacer que pase, completamos el método `depositar`:

```

public void depositar(int monto) {
    if (monto <= 0)
        throw new MontoInvalido();
    this.saldo += monto;
}

```

Si ahora corremos el programa de pruebas obtenemos:

```

La prueba pasó: se pasó un monto negativo y lanzó MontoInvalido
La prueba pasó: se pasó un monto 0 y lanzó MontoInvalido
La prueba pasó: el saldo se incrementó correctamente al depositar

```

Ahora bien, hay un error sutil que nuestras pruebas no están detectando. ¿Dónde garantizamos que al crearse un objeto instancia de la clase `Cuenta` su saldo debe ser cero? Ante todo, deberíamos hacer una prueba que lo chequee:

```

private void alCrearUnaCuentaSuSaldoDebeSerCero() {
    Cuenta cuenta = new Cuenta();
    if (cuenta.getSaldo() == 0)
        System.out.println("La prueba pasó: el saldo empezó siendo 0");
    else
        System.out.println("La prueba NO pasó: el saldo no empezó siendo
0");
}

```

Si corremos la prueba, vemos que funciona. Pero no hemos escrito código: ¿por qué pasó esto?

Cuando estas cosas ocurren, deberíamos ponernos en guardia. Notemos que sin hacer nada, una nueva prueba pasa. Esto quiere decir que, de alguna manera, ya habíamos implementado esta

funcionalidad antes, sin quererlo.

En realidad, lo que está pasando son dos cosas que Java hace automáticamente:

- Una variable de tipo *int* siempre se inicializa en cero.
- Cuando no definimos un constructor para una clase, se crea uno por omisión, sin parámetros.

Como no nos parece una buena práctica de programación basarnos en estos automatismos del lenguaje, vamos a escribir un constructor en la clase *Cuenta*, que haga explícito el hecho de que el saldo debe comenzar en cero:

```
public Cuenta () {  
    this.saldo = 0;  
}
```

A continuación, y dado que recién hicimos un cambio en el código, no debemos olvidarnos de ejecutar nuevamente las pruebas, las que afortunadamente nos dan un resultado positivo:

```
La prueba pasó: se pasó un monto negativo y lanzó MontoInvalido  
La prueba pasó: se pasó un monto 0 y lanzó MontoInvalido  
La prueba pasó: el saldo se incrementó correctamente al depositar  
La prueba pasó: el saldo empezó siendo 0
```

Ahora sí estamos tranquilos.

Ejercicios propuestos

- 1) Resuelva el caso de la celda del Sudoku usando Java como lenguaje.
- 2) Resuelva el caso del depósito en una cuenta bancaria usando Smalltalk como lenguaje.
- 3) Escriba las pruebas necesarias para el diagrama de secuencia del problema de las colas en la empresa que cobra servicios públicos planteado en el capítulo anterior.

Recapitulación

El capítulo que estamos cerrando sirvió para establecer una manera de implementar objetos basada en la noción de contratos y en desarrollar en base a pruebas automatizadas escritas desde antes. Vimos también algunas de las ventajas y subproductos de este enfoque.

En el próximo capítulo comenzaremos a estudiar maneras de lograr colaboración entre objetos y cómo esas maneras tienen un correlato en la estructura de las relaciones entre objetos.

4. Los objetos colaboran

Contexto

Hasta el capítulo anterior, hemos hablado de POO y de objetos que colaboran. Sin embargo, no hemos todavía mostrado las posibles formas de colaboración.

Dado que este es un aspecto fundamental de la POO, veremos de qué manera colaboran los objetos en los distintos lenguajes, y la diferencia que esas maneras de colaborar nos son útiles en diferentes situaciones. Concretamente, veremos delegación y programación por diferencia, cuestiones de comportamiento que se apoyan en los aspectos estructurales llamados asociación y herencia.

Relaciones entre objetos

Los objetos interactúan

Volvamos a nuestro primer capítulo. Allí vimos, siguiendo el ejemplo del Sudoku, y también el de la aplicación bancaria, que los objetos no tienen sentido como entidades aisladas, sino que deben interactuar en escenarios en conjunto con otros objetos, recibiendo mensajes, respondiendo, enviando a su vez mensajes y recibiendo las respuestas a los mismos.

Un programa orientado a objetos, en definitiva, no es más que un conjunto de objetos enviando y recibiendo mensajes a y de otros objetos. Como vimos en un capítulo anterior, a los objetos que envían mensajes solicitando un servicio los llamamos clientes, mientras que a aquellos que reciben mensajes y brindan servicios los llamamos receptores o servidores.

Atención:

Todo mensaje tiene un objeto cliente y uno receptor. Pero los objetos no tienen por qué ser siempre servidores o clientes, sino que su condición cambia según el mensaje. Por ejemplo, un objeto puede ser el receptor de un mensaje y, para poder responder ese mensaje envía a su vez una solicitud a objeto, en cuyo caso se convierte en cliente.

Ahora bien, aun cuando reconozcamos que los objetos no viven aislados, nos faltan herramientas conceptuales para llevar estas ideas a la práctica. De hecho, en el último capítulo sólo nos dedicamos a implementar el comportamiento de objetos aislados, sin dependencias con otros objetos. El objetivo del presente capítulo es ver las maneras en que los objetos se relacionan unos con otros, y su implementación.

Yendo a lo práctico... o casi

Retomemos el ejercicio del Sudoku. En el juego de Sudoku tenemos un tablero con filas, columnas y cajas, que a su vez tienen celdas. A su vez, las celdas de una fila son las mismas que están en ciertas columnas y ciertas cajas. ¿Cómo resolvemos estas relaciones?

En POO lo mejor no es pensar en relaciones entre objetos per se, sino más bien ir encontrando estas relaciones a partir del comportamiento esperado en los escenarios. De esa manera, no tenemos relaciones tan complejas sin una razón para ello.

Volvamos a empezar entonces. En el escenario para determinar si podemos colocar un número en una celda teníamos las siguientes situaciones de mensajes entre objetos:

- InterfazSudoku >> puedoColocar (numero, filaCelda, columnaCelda)
- InterfazSudoku >> celda (filaCelda, columnaCelda)
- celda >> estaLibre
- InterfazSudoku >> fila (filaCelda)
- fila >> contiene (numero)
- InterfazSudoku >> columna (columnaCelda)
- columna >> contiene (numero)
- InterfazSudoku >> caja (filaCelda, columnaCelda)
- caja >> contiene (numero)

Ello nos llevaría a pensar que, para poder brindar dichos servicios, cada fila, cada columna y cada caja deben poder determinar cuáles son sus celdas, para poder responder si un determinado número está en ellas (mensaje *contiene*).

Como desarrollar un diagrama que muestre todas las celdas, filas, columnas y cajas va a ser virtualmente incomprendible, veamos cómo son las relaciones para una sola celda. El diagrama de objetos de la figura 4.1 muestra esta situación.

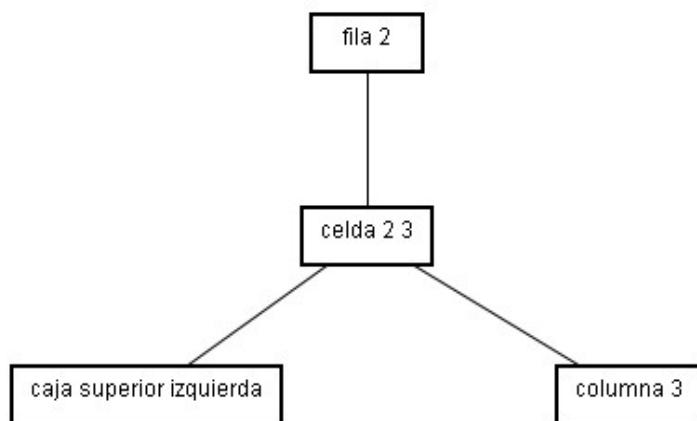


Figura 4.1 Relaciones de una celda

Ahora bien, cada una de las filas, columnas y cajas tiene que conocer a sus 9 celdas. Si en el diagrama anterior agregamos sólo una celda para cada fila, columna y caja, eligiéndolas cuidadosamente para evitar relaciones dobles, podemos mostrar parte de la red en la figura 4.2

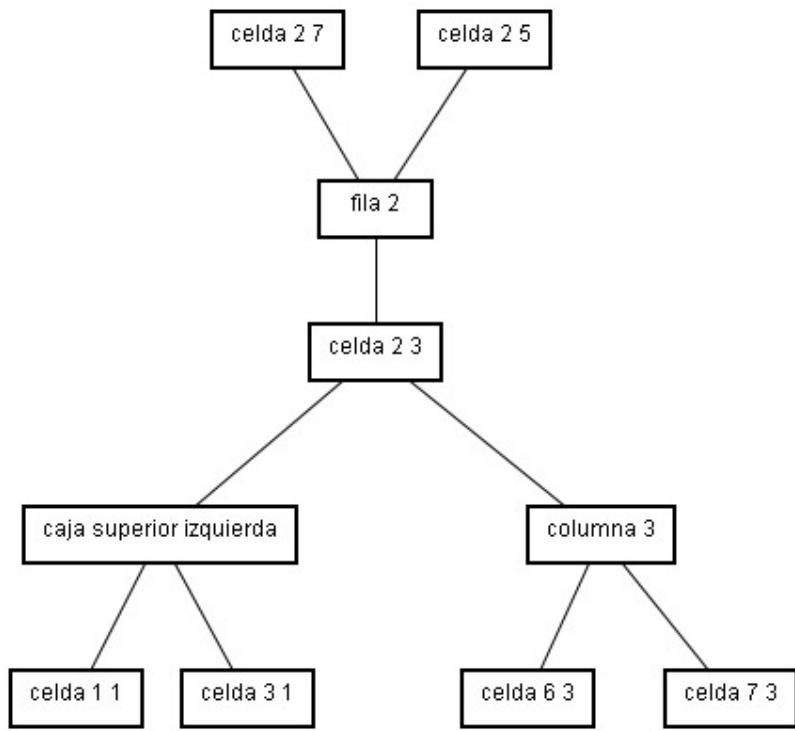


Figura 4.2 Relaciones de algunas celdas, filas, columnas y cajas

Si complejizamos un poco más el diagrama, agregando sólo algunas celdas y sus relaciones, ahora con menos cuidado, podemos ver una red más compleja aún en la figura 4.3.

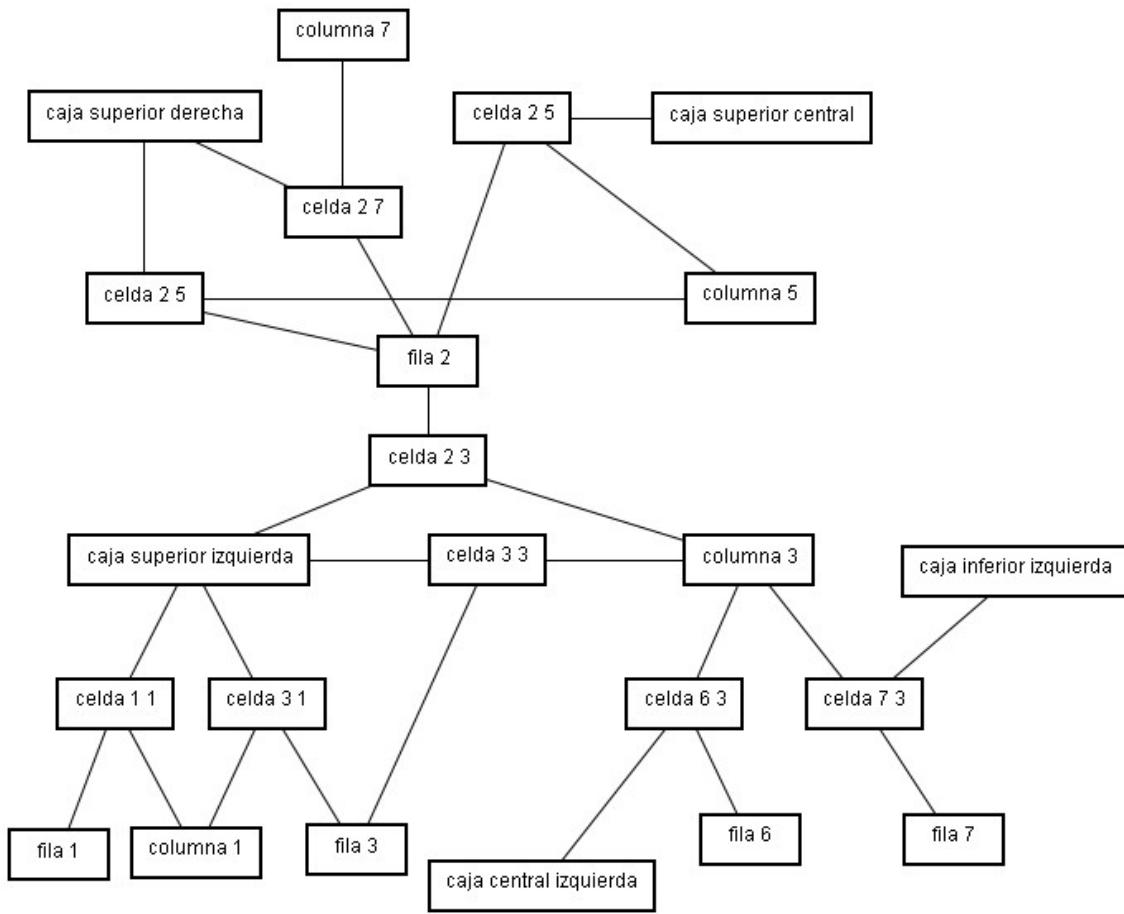


Figura 4.3 Relaciones de algunas celdas, filas, columnas y cajas más

Pero esta no es la red completa, ni mucho menos. Volvamos al principio, entonces.

El diagrama de secuencia que habíamos planteado era el de la figura 4.4.

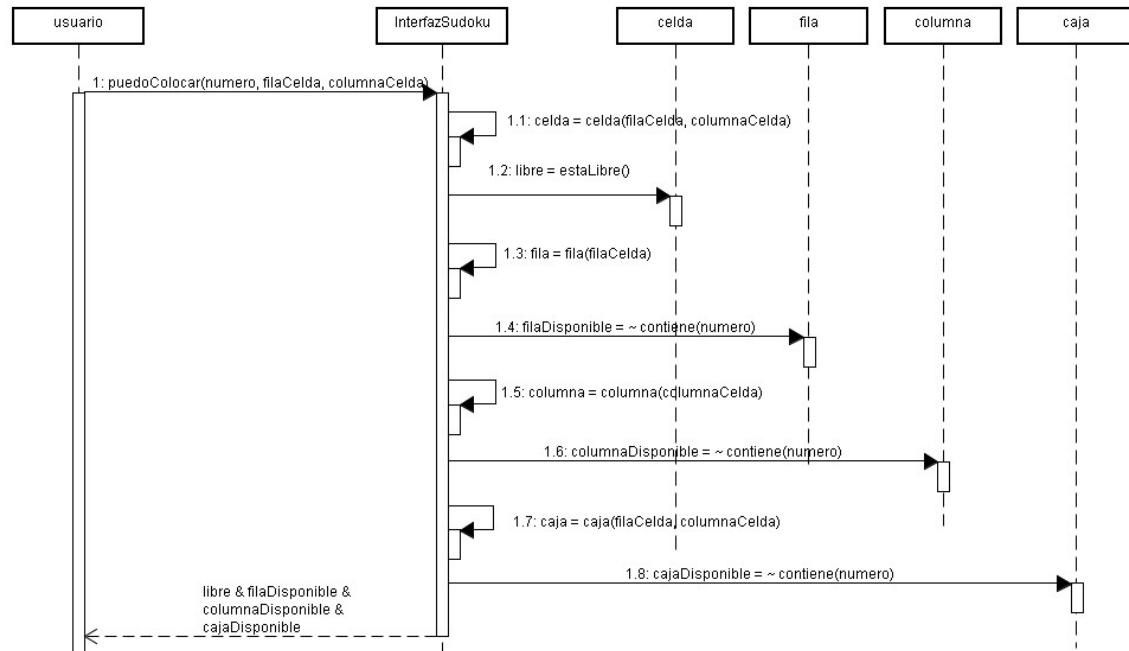


Figura 4.4 Repetición de la figura 2.3

Podemos pensar que el objeto *InterfazSudoku* tiene una referencia al objeto *Tablero*, que a su vez conoce sus filas, columnas y cajas, que a su vez referencian sus celdas.

Puede que el caso del Sudoku sea un poco especial, pero no es tan extraño que, en un sistema de tamaño mediano, nos encontremos con redes de objetos que son verdaderas telarañas con un gran número de objetos unidos por referencias.

Hagamos un diagrama de la estructura de objetos, como el de la figura 4.5.

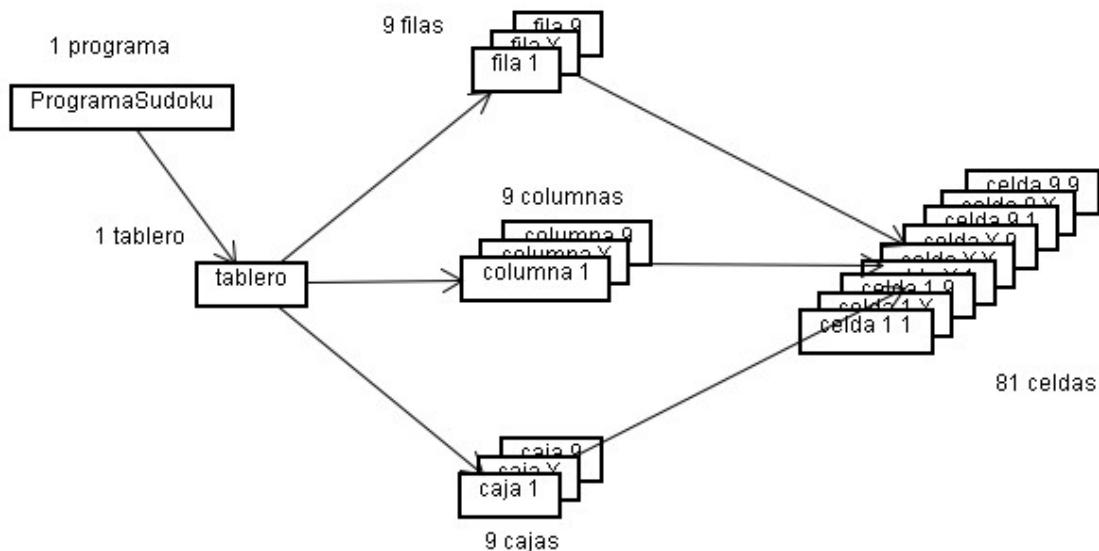


Figura 4.5 Estructura de objetos del Sudoku

Allí vemos los objetos principales, algunos formando colecciones, y también que las celdas de determinadas filas, columnas y cajas se superponen.

¿Cómo implementamos esto?

Ante todo, recordemos que en POO lo importante no es tanto la estructura de relacionamiento de los objetos, sino el comportamiento.

Por lo tanto, deberíamos ocuparnos de, siguiendo el diseño por contrato que vimos en el capítulo anterior, determinar los contratos de cada objeto a implementar.

Dejando de lado las celdas, cuyo comportamiento en este escenario ya implementamos en el capítulo anterior, debemos implementar los métodos cuyas firmas son las siguientes:

InterfazSudoku >> celda (filaCelda, columnaCelda)

InterfazSudoku >> fila (filaCelda)

fila >> contiene (numero)

InterfazSudoku >> columna (columnaCelda)

columna >> contiene (numero)

InterfazSudoku >> caja (filaCelda, celdaCelda)

caja >> contiene (numero)

Más el que provoca la invocación de todos los anteriores:

InterfazSudoku >> puedoColocar (numero, filaCelda, columnaCelda)

Por supuesto, también deberíamos implementar los constructores de estos objetos, de modo tal de poder escribir pruebas que tengan sentido. Como todo ese trabajo nos llevaría mucho código, y por lo tanto muchas páginas tediosas para el lector, vamos a implementar un solo método, trabajando en Smalltalk:

fila >> contiene (numero)

Procederemos a la manera del diseño por contrato. Las precondiciones son:

- 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 cumplen las precondiciones y el número argumento está en la fila, devolver verdadero.
- Si se cumplen las precondiciones y el número argumento no está en la fila, devolver falso.

La primera postcondición nos lleva a 3 pruebas muy similares a las que escribimos en el capítulo anterior para *colocarNumero*. No las transcribimos aquí para no aburrir a nuestros lectores.

La segunda postcondición nos hace escribir esta prueba:

```
contieneDevuelveTrueSiElNumeroEstaEnFila
    | fila colecciónAuxiliar |
    fila := Fila new.
    colecciónAuxiliar := OrderedCollection new.
    colecciónAuxiliar add: (Celda new colocarNúmero: 5).
    colecciónAuxiliar add: (Celda new colocarNúmero: 3).
    colecciónAuxiliar add: (Celda new).
    colecciónAuxiliar add: (Celda new).
    colecciónAuxiliar add: (Celda new colocarNúmero: 7).
    colecciónAuxiliar add: (Celda new colocarNúmero: 1).
    colecciónAuxiliar add: (Celda new).
    colecciónAuxiliar add: (Celda new colocarNúmero: 9).
    colecciónAuxiliar add: (Celda new).
    fila agregarCeldas: colecciónAuxiliar.
    (fila contiene: 5)
        ifTrue: [Transcript show: 'La prueba pasó: la fila tiene el número
y contiene devuelve true'; cr]
        ifFalse: [Transcript show: 'La prueba NO pasó: la fila tiene el
número y contiene devuelve false'; cr]
```

Y de la tercera postcondición se deriva la prueba siguiente:

```
contieneDevuelveFalseSiElNúmeroNoEstaEnFila
    | fila colecciónAuxiliar |
    fila := Fila new.
    colecciónAuxiliar := OrderedCollection new.
    colecciónAuxiliar add: (Celda new colocarNúmero: 5).
    colecciónAuxiliar add: (Celda new colocarNúmero: 3).
```

```

colecciónAuxiliar add: (Celda new).
colecciónAuxiliar add: (Celda new).
colecciónAuxiliar add: (Celda new colocarNúmero: 7).
colecciónAuxiliar add: (Celda new colocarNúmero: 1).
colecciónAuxiliar add: (Celda new).
colecciónAuxiliar add: (Celda new colocarNúmero: 9).
colecciónAuxiliar add: (Celda new).
fila agregarCeldas: colecciónAuxiliar.
(fila contiene: 8)
    ifTrue: [Transcript show: 'La prueba NO pasó: la fila no tiene
el número y contiene devuelve true'; cr]
    ifFalse: [Transcript show: 'La prueba pasó: la fila tiene no el
número y contiene devuelve false'; cr]

```

Por supuesto, el método *agregarCeldas* no está definido aún. Así que vamos a tener que hacerlo.

No obstante, no queremos abrumar a los lectores con decenas de líneas de código, así que confiamos en que comprendieron el método que estamos siguiendo, de escribir las pruebas, verificar que no pasen, escribir el código y verificar que finalmente pasen. Hacemos eso para el método *agregarCeldas* y luego lo vamos haciendo para el método *contiene*, que al final del proceso queda así:

```

contiene: numero
| encontrado |
( (numero < 1) | (numero > 9) )
    ifTrue: [ ValorInvalido new signal ].
encontrado := false.
celdas do: [:celda | (celda contiene: numero)
    ifTrue: [ encontrado := true ] ].
^encontrado.

```

Con esto ya tenemos suficiente material como para ir trabajando en conceptos.

Conceptualizando

Dependencia y asociación

Cuando un objeto conoce a otro y le puede enviar un mensaje decimos que hay una dependencia. Esta dependencia puede venir dada de tres maneras:

- Porque el objeto servidor se envía como argumento.
- Porque el objeto servidor se obtiene como respuesta al envío de un mensaje a otro objeto.
- Porque el objeto cliente tiene una referencia al servidor.

Definición: dependencia

Un objeto depende de otro cuando debe conocerlo para poder enviarle un mensaje.

Corolario:

Todo objeto cliente depende de su servidor.

En el caso particular de que el objeto cliente deba mantener una referencia al objeto servidor

(tercera de las posibilidades planteadas), decimos que la dependencia es una asociación, que es un caso especial – más fuerte – de dependencia.

Definición: asociación

Una asociación es una forma de dependencia en la que el objeto cliente tiene almacenada una referencia al objeto servidor.

Tanto las dependencias más débiles como las asociaciones tienen un sentido: decimos que el cliente depende del servidor, y no al revés. Para un mismo mensaje, lo habitual es que las dependencias sean unidireccionales.

Pero un objeto puede depender de otro y viceversa, a través de distintos mensajes. Si, en el caso del Sudoku, necesitásemos que una celda dependiese de su fila (por ejemplo, para saber cuál es su fila) y a su vez la fila dependiese de la celda (por ejemplo, porque necesita recorrer las celdas para ver si contiene un número determinado), la asociación sería bidireccional.

Atención:

Notemos que una asociación bidireccional puede ser un problema, ya que ambos objetos quedarían muy acoplados, y eso hace muy difícil implementarlos, probarlos y modificarlos por separado. A veces no hay más remedio, pero de poder evitarse, conviene evitarlo.

Interludio metodológico: ¿en qué orden probar cuando tenemos objetos que deben construirse luego?

Hay ocasiones en que venimos aplicando el enfoque basado en pruebas al construir un objeto, y ese objeto debe incluir una referencia a otro objeto, que aún no hemos implementado. Pasa algo parecido cuando estamos implementando un método y nos falta otro método en el cual el nuestro debe delegar. Notemos que nos ocurrió en el capítulo 3, al comenzar por implementar el método *estaLibre* de la clase *Celda*, más de una vez.

En general, tenemos tres opciones:

- Implementar el otro método u objeto inmediatamente, dejando en standby lo que veníamos haciendo y volviendo luego a donde estábamos. Es lo que hicimos cuando al necesitar *initialize* para *estaLibre*: lo implementamos y seguimos. En general, este enfoque funciona en casos en que lo que debemos construir sea pequeño y se pueda volver rápido al contexto previo.
- Implementar una versión mínima del método o clase como para poder seguir trabajando con lo que estábamos haciendo, dejando la implementación completa del método u objeto delegado para más adelante. Lo hicimos al abandonar la implementación del método *colocarNúmero* cuando se nos empezó a complicar. Este enfoque suele funcionar, pero hay que recordar que hay que volver en algún momento a lo que dejamos a medio camino.
- Diferir la implementación del método o clase auxiliar creando un objeto ficticio²⁵ y seguir en donde estábamos. Para esto existen muchas herramientas que facilitan la tarea. Dejaremos este tema aquí por exceder las pretensiones del libro.

²⁵ Un objeto ficticio es un objeto que se crea especialmente para simular la existencia de comportamiento que aún no ha sido implementado. Hay muchos tipos de objetos ficticios: un análisis detallado puede verse en [Meszaros 2007].

Digresión

Conceptualmente, si una prueba está probando el comportamiento de más de una clase, o incluso más de una responsabilidad de un objeto, es una **prueba de integración**, no una prueba unitaria.

Relaciones entre clases

En los lenguajes con clases, las relaciones entre objetos nos llevan a **relaciones entre clases**. Por ello, así como dibujamos ya varios diagramas de objetos de UML²⁶ para representar objetos en escenarios, podríamos hacer también diagramas de clases, como el de la figura 4.6

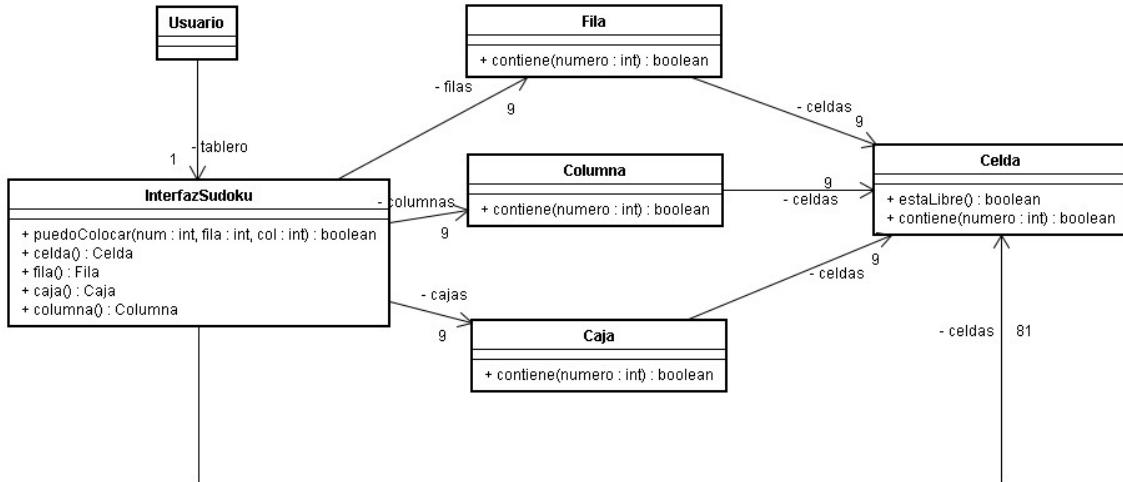


Figura 4.6 Relaciones entre clases

El diagrama de clases de la figura 4.6 es el que surge del escenario para el que hicimos el diagrama de secuencia. Lo que muestra el diagrama son clases, cuyas instancias son los objetos del diagrama de secuencia. Las líneas que vinculan las clases son relaciones entre las mismas, derivadas de las relaciones entre objetos. En este caso, las líneas continuas indican asociaciones. Las dependencias débiles no suelen representar en diagramas de clases, pero en caso de hacerlo se hace con líneas de puntos. Las flechas indican qué clases deben conocer a cuáles. Los números en el extremo de las flechas indican la cardinalidad, es decir, la cantidad de objetos de una clase que hay por cada uno de la otra. Los nombres sobre las flechas indican un nombre de asociación, que en términos de implementación suele terminar siendo el nombre de la referencia entre ambos objetos. Y en él se indican también los métodos de cada clase.

Colaboración por delegación

Más arriba dijimos que uno de los objetivos centrales de la POO era permitir la programación en base a componentes. Lo que no dijimos es cómo se ensamblaban esos componentes entre sí.

La manera más sencilla de vinculación es la simple asociación entre objetos.

Esta vinculación, que vista de manera estructural es una asociación, nos sirve para la delegación de comportamiento.

²⁶ Los diagramas de relaciones entre objetos que hicimos en este capítulo, los de secuencia de este capítulo y del anterior, e incluso los diagramas de comunicación del primer capítulo, son todos diagramas de objetos.

En efecto, en nuestro caso de estudio, dado que cada objeto *fila* tiene asociaciones con 9 objetos *celda*, puede delegar parte de su comportamiento. El caso más obvio de esta delegación es el mensaje *contiene*: notemos que cuando una fila recibe el mensaje *contiene*, no hace mucho más que enviar un mensaje – también llamado *contiene* – a cada una de las celdas a las que está asociada.

La vinculación por asociación está presente en todos los lenguajes de POO. Pero no es la única forma de lograr delegación. En realidad, cualquier dependencia débil sirve para delegar. Por ejemplo, si un objeto es pasado como argumento de un mensaje que le enviamos a otro objeto, el objeto argumento se puede convertir en receptor de mensajes del objeto que lo recibió. Lo mismo ocurre si un objeto se obtiene como resultado de un mensaje y luego se lo usa para delegación.

El mejor ejemplo de esto último lo tenemos en nuestro ejemplo de transferencia entre cuentas bancarias (recomendamos volver a mirar el diagrama de secuencia respectivo). En él, la aplicación, al recibir un mensaje *transferir*, pedía dos objetos, *cuentaDebitar* y *cuentaAcreditar*, cuyas referencias guarda en variables internas del método, y luego las usa para pasárselas los mensajes *extraer* y *depositar*.

El código de *transferir* podría ser:

```
public void transferir (int monto, String cbuDebitar, String cbuAcreditar) {  
    Cuenta cuentaDebitar = baseCuentas.buscarCuenta(cbuDebitar);  
    Cuenta cuentaAcreditar = baseCuentas.buscarCuenta(cbuAcreditar);  
    cuentaDebitar.extraer(monto);  
    cuentaAcreditar.depositar(monto);  
}
```

Programación por diferencia: herencia

En los lenguajes que trabajan con clases, hay una forma de vinculación de componentes que es la llamada **herencia**, o también relación de generalización-especialización.

Definición: herencia

La herencia es una relación entre clases, por la cual se define que una clase puede ser un caso particular de otra. A la clase más general la llamamos madre y a la más particular hija.

Corolario:

Cuando hay herencia, todas las instancias de la clase hija son también instancias de la clase madre.

El uso obvio que tiene esta propiedad es que podemos definir un comportamiento en la clase madre y, al considerar a todas las instancias de distintas clases hijas como instancias de la madre, usar ese comportamiento de la madre sin necesidad de escribir código en cada clase hija. Eso es lo que se llama programar por diferencia: en las clases hijas sólo incluimos los comportamientos que las diferencian de la clase madre.

En nuestro caso de estudio, por ejemplo, podríamos tener una clase *ColeccionCeldas* y que las clases *Fila*, *Columna* y *Caja* deriven de ella. Eso querría decir, en una mirada estática, que cualquier instancia de las clases *Fila*, *Columna* o *Caja* sería también una instancia de la clase *ColeccionCeldas*. Ahora bien, ¿para qué nos podría servir hacer eso?

En realidad, para bastante. Por ejemplo, el método *contiene* de la clase *Fila* sería idéntico si lo escribimos en las clases *Columna* y *Caja*. Por lo tanto, ¿no podríamos pasarlos a la clase

ColeccionCeldas – una vez creada, por supuesto – y sacarlos de *Fila*, de modo tal que al crear nuestras clases *Columna* y *Caja* no necesitemos escribirlos? Esto es, precisamente, programar por diferencia: hay métodos que ya tenemos en una clase y, como deseamos usarlos tal como están, no necesitamos volverlos a escribir.

Definición: programación por diferencia

Programamos por diferencia cuando indicamos que parte de la implementación de un objeto está definida en otro objeto, y por lo tanto sólo implementamos las diferencias específicas.

Cuando el compilador o intérprete de un lenguaje con herencia encuentra el envío de un mensaje a un objeto, y en la clase de ese objeto no hay un método para responder a ese mensaje, busca en la clase madre el método correspondiente. Si lo encuentra allí, lo usa. Si no lo encuentra allí, sube un nivel más en la jerarquía de herencia, y así sucesivamente. Si no existiera ese método en toda la jerarquía, se provocaría un error.

Llevada esta idea a un diagrama de UML, obtendríamos el de la figura 4.7.

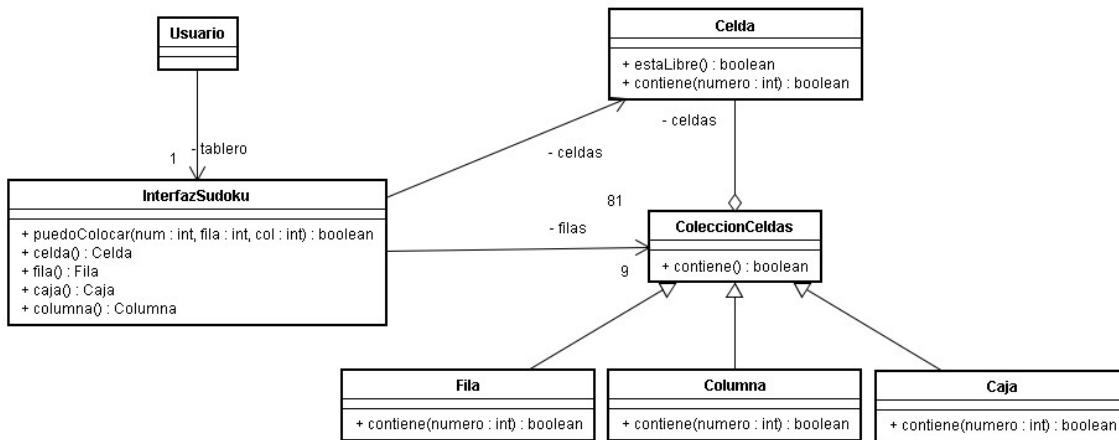


Figura 4.7 Herencia en diagrama de clases

En el diagrama se ven las relaciones de herencia entre las clases *Fila*, *Columna* y *Caja* y la clase más general *ColecciónCeldas*, con el uso de la flecha de punta triangular. Hemos eliminado las cardinalidades de las asociaciones para hacer el diagrama más legible. El mismo diagrama, al mostrar una asociación entre *ColecciónCeldas* y *Celda*, nos indica que también hay asociaciones desde toda clase descendiente de *ColecciónCeldas*, es decir, como ya sabíamos, desde las clases *Fila*, *Columna* y *Caja*.

La herencia resulta muy seductora. En efecto, ¿qué más cómodo que programar por diferencia? Pero como tantas ideas seductoras, hay que analizarla con cuidado.

Por ejemplo, con la idea de reutilizar, podemos pensar que la clase *ColecciónCeldas* podría usar parte del comportamiento de la clase *Celda*, y entonces pensar que deberíamos aplicar herencia. Pero está claro que una colección de celdas no es una celda, ni tampoco su comportamiento deriva del de la clase *Celda*.

Como regla, entonces, conviene hacer siempre este test a una relación para ver si aplicar o no la herencia: ¿necesitamos reutilizar la interfaz de una clase tal como está en otra clase, sin que nada me sobre? Si es así, puedo usar herencia, haciendo que la clase que va a reutilizar sea hija de la clase que provee el código que nos interesa. Si no, conviene reutilizar por delegación.

Recomendación:

Dejar la herencia solamente para aquellos casos en que la clase madre tenga una interfaz contenida en la interfaz de la clase hija. Es decir, que la clase madre no tenga métodos que sobren en la clase hija.

Una digresión final sobre el vocabulario con el que nos referimos a las clases que se relacionan en cualquier situación de herencia. A menudo se habla de clases madre e hijas, pero es común también hablar de clases base y derivadas, o ancestro y descendientes. En esta obra, usaremos cualquiera de estos pares de términos indistintamente.

Programación por diferencia: delegación de comportamiento

Los lenguajes que no tienen clases no pueden tener herencia. Sin embargo, se han ingenierado para definir algunos mecanismos alternativos. Por ejemplo, JavaScript permite la programación por diferencia con un mecanismo que llamamos delegación de comportamiento.

Veámoslo en un ejemplo. En JavaScript podemos definir un objeto que represente a la colección de celdas, así:

```
var ColeccionCeldas = {
    celdas: [],
    contieneNumero: function(numero) {
        for( var i in this.celdas ) {
            if ( this.celdas[i].numero === numero ) {
                return true;
            }
        }
        return false;
    }
};
```

Ahora bien, vimos también que, dado que todo objeto es además un prototipo, podemos definir otro objeto denominado *Fila*, a partir del anterior:

```
var Fila = Object.create(ColeccionCeldas);
```

Según lo que vimos en el capítulo 2, esto crea otro objeto que entiende los mismos mensajes que *ColeccionCeldas*. Pero, ¿cómo funciona esto?

En realidad, *ColeccionCeldas* tiene dos atributos y un método:

- *celdas*: atributo definido explícitamente
- *contieneNumero*: función definida explícitamente
- *__proto__*²⁷: atributo implícito en todo objeto JavaScript, que referencia a su prototipo

Mientras tanto, *Fila*, para el cual no hemos definido explícitamente métodos ni atributos, tiene al atributo *__proto__*, que referencia al objeto *ColeccionCeldas*.

En definitiva, lo que ocurre es que el objeto *Fila* tiene una referencia *__proto__* al objeto *ColeccionCeldas*, y hay una regla que dice que si un mensaje no es comprendido por el objeto receptor en forma directa, lo debe ir a buscar a su prototipo, que en este caso referencia al objeto *ColeccionCeldas*. O sea que no es más que una forma alternativa de delegación: lo que nosotros llamaremos delegación de comportamiento. En consecuencia, para *Fila* estará definido también

²⁷ En el estándar se denomina también [[prototype]].

el comportamiento que hace que pueda recibir el mensaje `contieneNumero`. Esto es una derivación de lo que vimos en el capítulo 2, cuando mostramos que un objeto delega su comportamiento en su prototipo.

Esto también es programar por diferencia: ahora todo objeto que refiere al prototipo `Fila` va a entender el mensaje `contieneNumero`, ya que el método está definido para el prototipo `ColeccionCeldas`, y `Fila` reutiliza dicho prototipo.

El diagrama de objetos de la figura 4.8 muestra cómo están vinculados los objetos.

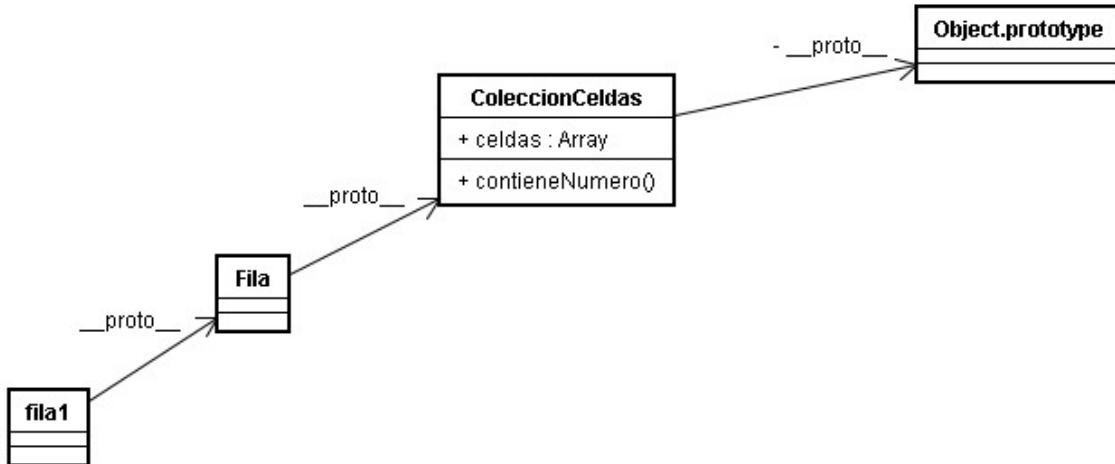


Figura 4.8 Vinculación de objetos en JavaScript

Pero aunque es programación por diferencia no es herencia en el sentido habitual. Recordemos que en JavaScript no hay clases, así que de ninguna manera puede haber relaciones entre clases. Lo único que hace el intérprete de JavaScript es configurar la propiedad `Fila.__proto__` para que refiera al objeto `ColeccionCeldas`, de modo que todos los métodos definidos en `ColeccionCeldas` sean accesibles desde el objeto `Fila`. Por eso `Fila` entiende los mensajes que entiende `ColeccionCeldas`.

Definición: delegación de comportamiento

Cuando deseamos programar por diferencia en JavaScript recurrimos a la delegación de comportamiento, por la cual un objeto delega su comportamiento en otro objeto prototípico sin necesidad de repetir las definiciones de los métodos.

Hay quienes intentan demostrar que esto no es más que otra manera de decir herencia. En este libro pretendemos dejar en claro que no es lo mismo, como tampoco es lo mismo todo el azúcar sintáctico que se agrega en cada versión de ECMAScript para hacer parecer que el lenguaje tiene herencia. Personalmente, como no creo que la herencia sea una virtud per sé de los lenguajes de POO, no creemos estar quitándole nada a la POO de JavaScript – uno de los pocos lenguajes que implementan POO sin clases, sólo con objetos – mostrando las cosas como son.

Redefinición

Los lenguajes de programación que tienen clases y herencia, permiten volver a definir métodos que ya estuvieran definidos en la clase base en sus clases derivadas. Esto se suele hacer cuando una clase derivada debe implementar un método que ya está definido en su clase base, pero no lo puede hacer tomándolo tal como está, porque necesita un comportamiento diferente.

Por ejemplo, en la Argentina, suele haber dos tipos de cuentas bancarias: la caja de ahorro, que

permite extraer solamente un monto inferior al saldo de la cuenta; y la cuenta corriente, en la cual se define un valor de giro en descubierto, y que por lo tanto permite extraer más que el saldo, usando ese crédito especial que es el giro en descubierto.

Si en nuestra aplicación bancaria dejásemos la noción de *Cuenta* para la caja de ahorro, y deseamos definir una nueva clase *CuentaCorriente*, podríamos hacerlo redefiniendo el método *extraer*. El diagrama de clases es el de la figura 4.9.

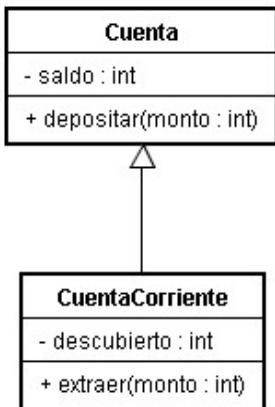


Figura 4.9 Herencia entre cuentas bancarias

Notemos en el diagrama que el método *extraer*, definido en la clase *Cuenta*, se vuelve a definir en la clase *CuentaCorriente*, porque su implementación es diferente, aunque en ambos casos estemos extrayendo dinero de una cuenta bancaria.

Importante:

La redefinición existe para definir un mismo comportamiento en una clase derivada, para el mismo mensaje de la clase base. Por lo tanto, la semántica o significado del mensaje se debe mantener. Si así no fuera, conviene definir un método diferente, con nombre diferente.

Desde el punto de vista del diseño por contrato, para que haya redefinición, es importante que se mantenga la firma del método. Las precondiciones y postcondiciones podrían variar ligeramente, aunque sin desnaturalizar el hecho de que el mensaje al cual estamos respondiendo es el mismo. Enfoques más estrictos afirman que las precondiciones sólo pueden ser más estrictas, lo cual no hemos respetado en nuestro ejemplo.

Clases abstractas

Supongamos que tomamos el ejemplo reciente de las clases en la aplicación bancaria y planteamos, para ser más coherentes, que exista una clase *Cuenta* y dos clases descendientes: *CajaAhorro* y *CuentaCorriente*. En ese caso, el diagrama de clases podría quedar así como en la figura 4.10.

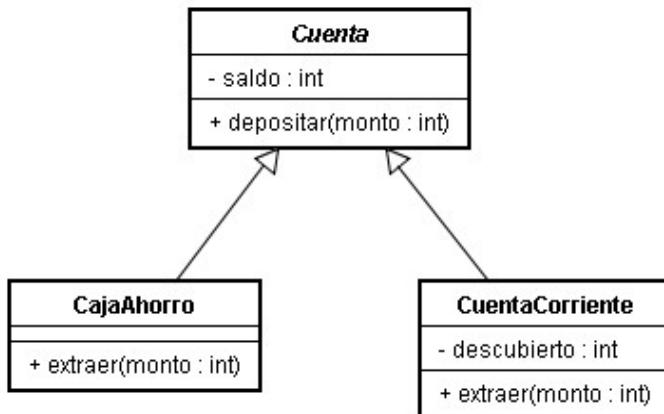


Figura 4.10 Herencia entre cuentas bancarias con una clase abstracta

En este caso particular, no habrá ninguna instancia que lo sea directamente de la clase *Cuenta*, ya que todas las instancias que respondan al concepto de cuenta lo serán antes de las clases *CajaAhorro* o *CuentaCorriente*. En este caso, decimos que la clase *Cuenta* es abstracta. En UML, como vemos en el diagrama, las clases abstractas se representan con su nombre en cursiva.

Definiciones: clase abstracta y clase concreta

Una clase es abstracta cuando no puede tener instancias en forma directa, habitualmente debido a que sus clases descendientes cubren todos los casos posibles.

Cuando queramos indicar una clase que no es abstracta (que puede tener instancias) la llamaremos clase concreta.

Métodos abstractos

Volvamos al caso anterior. Notemos que, al haber definido dos clases descendientes de *Cuenta*, que difieren en la implementación del método *extraer*, dicho método lo pasamos directamente a las mismas.

Sin embargo, hay ocasiones en las que deseamos indicar que toda clase derivada de una clase debe entender un mensaje, aun cuando en la clase en la que estemos trabajando no tenga sentido definirlo.

En este caso, usamos métodos abstractos.

Definiciones: método abstracto y método concreto

Un método es abstracto cuando no lo implementamos en una clase, pero sí deseamos que todas las clases descendientes puedan entender el mensaje.

Cuando queramos indicar un método que no es abstracto (que tiene implementación) lo llamaremos método concreto.

Como corolario, un método abstracto no va a tener implementación, sino que sólo se define su firma, para que las clases descendientes lo implementen en base a ella.

La figura 4.11 muestra este escenario, en el cual se ve que en UML el nombre de un método abstracto se escribe en cursiva.

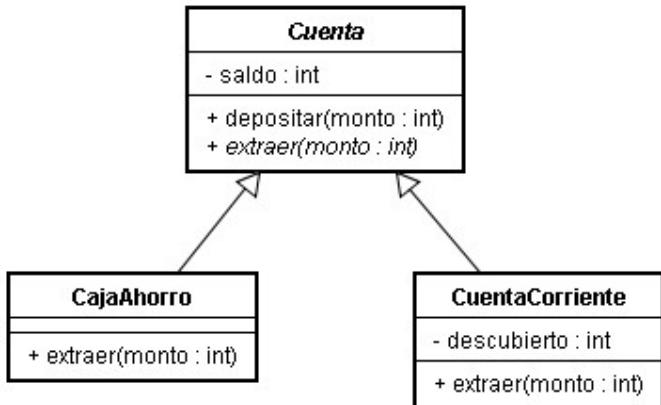


Figura 4.11 Método abstracto en cuentas bancarias

Cuestiones de implementación

Delegación y herencia en Smalltalk y Java

Como dijimos más arriba, hay dos herramientas para la utilización de componentes en los lenguajes con clases: la delegación y la herencia.

La delegación tiene una implementación muy directa en ambos lenguajes: basta recibir al objeto servidor como parámetro o como respuesta a un mensaje, o simplemente tener una referencia a él, para poder delegar.

Por ejemplo, si en el ejemplo de transferencia entre cuentas bancarias escribimos, en Java:

```

public void transferir (int monto, String cbuDebitar, String cbuAcreditar) {
    Cuenta cuentaDebitar = baseCuentas.buscarCuenta(cbuDebitar);
    Cuenta cuentaAcreditar = baseCuentas.buscarCuenta(cbuAcreditar);
    cuentaDebitar.extraer(monto);
    cuentaAcreditar.depositar(monto);
}

```

Lo que hacemos es obtener dos objetos, los referenciados por *cuentaDebitar* y por *cuentaAcreditar*, como resultado del envío del mensaje *buscarCuenta* al objeto referenciado por *baseCuentas*. Luego, usamos esos objetos (delegación) para enviarles los mensajes *extraer* y *depositar*.

En Smalltalk, el código equivalente sería, con un significado idéntico:

```

transferir: monto desde:cbuDebitar hacia:cbuAcreditar
cuentaDebitar := ( baseCuentas buscarCuenta: cbuDebitar ).
cuentaAcreditar := ( baseCuentas buscarCuenta: cbuAcreditar ).
cuentaDebitar extraer: monto.
cuentaAcreditar depositar: monto.

```

El caso de asociación se puede ver en este fragmento de código Smalltalk que hemos escrito antes en este mismo capítulo:

```

contiene: numero
| encontrado |
( (numero < 1) | (numero > 9) )

```

```

        ifTrue: [ ValorInvalido new signal ].
encontrado := false.
celdas do: [:celda | (celda contiene: numero)
        ifTrue: [ encontrado := true ] ].
^encontrado.

```

Aquí vemos cómo el objeto fila, delega en varias celdas invocando el método *contiene*, simplemente porque tiene una colección de celdas referenciadas desde su atributo *celdas*.

En Java, el equivalente sería:

```

public boolean contiene (int numero) {
    if ( (numero < 1) || (numero > 9) )
        throw new ValorInvalido();
    for (Celda celda : celdas)
        if ( celda.contiene(numero) )
            return true;
    return false;
}

```

La herencia también tiene una implementación sencilla. Por ejemplo, en Smalltalk se declaran las clases definiendo de qué otra clase heredan. La línea:

Error subclass: #ValorInvalido

Nos indica que *ValorInvalido* es una clase hija de *Error*.

Como en Smalltalk hay una clase madre por defecto de todas las demás, llamada *Object*, las clases más altas en la jerarquía se definen como ya vimos:

Object subclass: #Celda

En Java ocurre algo parecido, aunque la sintaxis es diferente. Por ejemplo:

class ValorInvalido extends RuntimeException { }

Indica que *ValorInvalido* es una clase hija de *RuntimeException*.

En Java también hay una clase madre por defecto, y también se denomina *Object*, aunque no se coloca este parentesco al definir una clase. Por ejemplo, se entiende que:

```

class Celda {
    ...
}

```

Está definiendo una clase *Celda* que, al no tener declarada su clase madre, es hija de *Object*.

Visibilidad

Los lenguajes de programación orientados a objetos prevén distintos tipos de visibilidad de los elementos del lenguaje: métodos, atributos y clases. Habitualmente hay tres niveles de visibilidad:

- **Pública:** el método, atributo o clase en cuestión se puede utilizar en cualquier parte del programa.
- **Privada:** el método, atributo o clase en cuestión se puede utilizar solamente dentro de su clase.
- **Protegida:** el método, atributo o clase en cuestión se puede utilizar solamente dentro de

su clase o una clase descendiente de ella.

En Java, los tres niveles de visibilidad pueden usarse, colocándoles las palabras *public*, *private* y *protected*, como venimos haciendo en los ejemplos. Además, existe un nivel más de visibilidad, “de paquete”, que permite que el elemento en cuestión se pueda usar solamente dentro del mismo paquete²⁸.

En Smalltalk hay menos control de la visibilidad por parte del programador: todos las clases y los métodos son públicos y los atributos son todos protegidos.

Un buen manejo de la visibilidad permite que el programador haga una buena administración del encapsulamiento, dando visibilidad solamente a aquellos elementos que necesitan ser usados desde afuera. Por ejemplo, si decidimos crear un método que sólo nos sirva para organizar nuestro código – y no porque queremos que implemente un mensaje que el objeto deba entender – podríamos definirlo como privado, y de esa manera mantenemos el encapsulamiento bajo control.

Recomendación: hacer un buen uso de los atributos de visibilidad para garantizar el encapsulamiento.

Métodos y clases abstractos

En Java, la declaración de un método como abstracto que hace añadiendo la palabra *abstract* en su firma, y cerrando con un punto y coma en vez de un bloque de código. He aquí un ejemplo:

```
public abstract void extraer (int monto);
```

Una clase que tiene algún método abstracto debe ser abstracta, por lo que hay que agregar la palabra *abstract* en su encabezado, como en el ejemplo:

```
public abstract class Cuenta {  
    ...  
}
```

De todas maneras, podría haber clases abstractas sin métodos abstractos.

En Smalltalk, un método abstracto se implementa enviando un mensaje que implica que las subclases deben implementarlo, como en el ejemplo:

```
extraer: monto  
    self subclassResponsibility.
```

En el caso de invocar este método, el objeto se envía a sí mismo el mensaje *subclassResponsibility*, que es el nombre de un método en la clase *Object* que provoca que se lance una excepción de tipo *SubclassResponsibility*.

Además, en Smalltalk, las clases abstractas son aquellas que tienen algún método abstracto, sin que tengamos que declararlo en forma explícita.

Ahora bien, que un método o una clase sean abstractos no tiene el mismo significado en ambos lenguajes, debido al momento en que hacen la comprobación de tipos. Smalltalk, como recién dijimos, maneja la cuestión en tiempo de ejecución, lanzando excepciones. En Java, en cambio, es el compilador el que chequea que no se llame un método abstracto o se intente instanciar una clase abstracta.

²⁸ Java agrupa las clases en paquetes, como se puede ver en el apéndice B. Si bien no hemos hecho énfasis en esta característica particular del lenguaje, permite un grado de abstracción interesante.

Constructores en situaciones de herencia y asociación

Recordemos que un constructor es un método que inicializa un objeto apenas creado. Cuando una clase hereda de otra, los constructores debieran invocarse en cascada en las clases descendientes.

En efecto, supongamos que tenemos una clase *Cuenta*, cuyos objetos son inicializados con el siguiente constructor, en Java:

```
public Cuenta (String titular) {  
    this.titular = titular;  
    this.saldo := 0;  
}
```

Si luego tenemos una clase *CuentaCorriente*, derivada de ésta, con el agregado del atributo *descubiertoAcordado*, podríamos creer que la inicialización podría hacerse con este constructor:

```
// OJO: lo que sigue está mal  
public CuentaCorriente (String titular, int descubiertoAcordado) {  
    this.titular = titular;  
    this.saldo := 0;  
    this.descubiertoAcordado := descubiertoAcordado;  
}
```

Sin embargo, si los atributos *titular* y *saldo* fuesen privados de la clase *Cuenta*, esto no va a ser posible²⁹. Por eso, la forma correcta de hacerlo sería:

```
public CuentaCorriente (String titular, int descubiertoAcordado) {  
    super(titular); // "super" invoca al constructor de la clase madre  
    this.descubiertoAcordado := descubiertoAcordado;  
}
```

En Smalltalk, el método *initialize* no tiene parámetros, como vimos, pero si se utilizan inicializadores ad hoc, conviene llamarlos de la misma manera que hacemos con los constructores en Java. Por ejemplo:

```
inicializarConTitular: titular conDescubierto: descubierto  
    "llamamos al inicializador de la clase madre:"  
    self inicializarConTitular: titular.  
    "inizializamos el atributo propio:"  
    self descubiertoAcordado := descubierto.
```

Hay quienes extienden esta idea de la inicialización en cascada en la jerarquía de herencia a una inicialización en cascada en situaciones de asociación. Pero es un error hacerlo siempre así.

Si por cada objeto que creásemos tuviésemos que crear en cascada los objetos referenciados, eso significaría que los objetos referenciados quedarían atados al objeto de arranque de la asociación. Si esto es lo que se quiere, que es lo que UML denomina “composición”, de acuerdo con ese enfoque. Pero si deseamos que los objetos referenciados puedan tener una existencia independiente del que los referencia, la receta ya no es válida.

Por ejemplo, nuestro ejemplo de Sudoku es elocuente en este caso: no podemos crear las celdas por cada creación de fila, columna o caja, ya que las mismas celdas son referenciadas desde

²⁹ Recordemos que un atributo privado no puede ser usado en ninguna otra clase, ni siquiera en una hija.

distintas filas, columnas y cajas.

En cambio, si los clientes de nuestra aplicación bancaria tuvieran un domicilio asociado que sólo puede corresponder a ese cliente, y ambos objetos quedarán siempre acoplados, podemos inicializarlos juntos y mantenerlos en sincronía durante su existencia. Eso es lo que llamamos composición.

Definición: composición

Decimos que hay composición entre dos objetos cuando ambos están tan acoplados que no se admite su existencia sin el otro. Como corolario, la creación de un objeto siempre llevaría a la creación del otro, y su destrucción también.

En esta obra vamos a evitar referirnos a la composición como concepto, salvo que lo necesitemos realmente.

Herencia y compatibilidad en lenguajes con comprobación de tipos estática

Hasta ahora, vimos que en los lenguajes de comprobación estática, la variable que contiene una referencia tenía que tener el mismo tipo que la clase del objeto en cuestión. Por ejemplo, en Java, escribíamos:

```
Cuenta c = new Cuenta();
```

Sin embargo, esto no tiene por qué ser así. Si *CuentaCorriente* es una subclase de *Cuenta*, dado que todas las instancias de *CuentaCorriente* son, en definitiva, instancias también de *Cuenta*, podríamos escribir:

```
Cuenta c = new CuentaCorriente();
```

Esto se debe a la existencia de un principio de diseño orientado a objetos denominado principio de sustitución, que fue enunciado por primera vez por Barbara Liskov.

Definición: principio de sustitución o de Liskov

Los objetos deberían poder ser reemplazados por instancias de sus subtipos sin que por ello el programa deje de ser correcto.

En términos del lenguaje de programación, podríamos enunciar el siguiente corolario:

Corolario:

Se puede almacenar, en una variable de un determinado tipo, referencias a objetos instancias de subclases del tipo de la variable.

Esto implica que un mismo objeto se podría referenciar desde variables de tipos diferentes, como se ve en este otro fragmento:

```
CuentaCorriente cc = new CuentaCorriente();
Cuenta c = cc;
```

Sin embargo, y si bien en ambos casos estamos referenciando el mismo objeto – una instancia de *CuentaCorriente* – el compilador de Java va a comprobar que la variable *cc* reciba cualquier mensaje de la clase *CuentaCorriente*, mientras que la variable *c* sólo va a poder recibir los mensajes que estén definidos en la clase *Cuenta*. Esto tiene algunas implicaciones adicionales, sobre las que volveremos en el capítulo de polimorfismo.

Una cuestión adicional. Mire el código que sigue y vea si le parece correcto:

```
CuentaCorriente cc = new CuentaCorriente();
```

```
Cuenta c = cc;  
CuentaCorriente otra = c;
```

Piense un poco antes de seguir leyendo...

No, no es correcto. Si tiene dudas, convéntase pretendiendo compilarlo. ¿Qué es lo que pasa? En definitiva, tanto *cc* como *c* referencian a un mismo objeto, que sin dudas es una instancia de *CuentaCorriente*. ¿Por qué entonces está mal la última línea?

La respuesta es más fácil si recordamos que el compilador de Java hace chequeo de tipos. En efecto, el compilador no puede saber qué referencia la variable *c*, más allá de que es una *Cuenta*. Cuando decimos que en *c* hay una referencia a una instancia de *CuentaCorriente*, nos referimos a lo que va a ocurrir en tiempo de ejecución, no en el momento en el que el compilador procesa estas líneas.

Para salvar estas situaciones se utiliza el “casteo”, barbarismo³⁰ que indica que le podemos pedir al compilador que suspenda la verificación de tipos. Por ejemplo, si escribimos:

```
CuentaCorriente cc = new CuentaCorriente ( );  
Cuenta c = cc;  
CuentaCorriente otra = (CuentaCorriente)c;
```

Lo que hacemos en este caso es decirle al compilador que sabemos que en *c* va a venir una referencia a una *CuentaCorriente*, y le pedimos que no haga el chequeo estático en este caso.

En efecto, en estos casos, si en tiempo de ejecución ocurre que en *c* viene una referencia a una *CuentaCorriente*, no habrá ningún problema. Si, en cambio, ocurriese que nos equivocamos y *c* nos trae una referencia a otro objeto que no sea una *CuentaCorriente*, vamos a obtener un error en tiempo de ejecución bajo la forma de una excepción de tipo *ClassCastException*.

Ejercicios adicionales

Construcción de objetos en cadena

Vamos a implementar la creación del tablero en la aplicación del Sudoku. Como ya dijimos, no es una tarea sencilla, ya que la sola construcción del tablero debería llevar a que queden creadas ya todas las celdas, filas, columnas y cajas. Además, la relación entre el tablero y las celdas es de composición (no pueden existir celdas sin tablero), lo mismo que entre el tablero y sus filas, columnas y cajas.

Lo primero que deberíamos hacer es definir la firma del mensaje de creación. Una posibilidad sería que se crease un tablero completo de Sudoku mediante un mensaje como el que sigue:

```
Tablero new cargar: # ( 5 3 0 0 7 0 0 0 0  
6 0 0 1 9 5 0 0 0  
0 9 8 0 0 0 0 6 0  
8 0 0 0 6 0 0 0 3  
4 0 0 8 0 3 0 0 1  
7 0 0 0 2 0 0 0 6
```

³⁰ “Casteo” no es una palabra castellana. Viene de forzar el término “to cast” del idioma inglés, que significa “hacer el papel de”.

0	6	0	0	0	0	2	8	0
0	0	0	4	1	9	0	0	5
0	0	0	0	8	0	0	7	9

Que representa el tablero de la figura 4.12.

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	

Figura 4.12 Tablero de Sudoku a construir

Lo que recibe como parámetro el mensaje cargar es un *Array* de Smalltalk.

Ahora tendríamos que definir con claridad cómo queremos que un tablero se relacione con sus celdas, filas, columnas y cajas. En principio, el tablero debe tener referencias a las 81 celdas, a sus 9 filas que referencian 9 celdas cada una, a sus 9 columnas que referencian 9 celdas cada una y a sus 9 cajas que también referencian 9 celdas cada una. ¡Pero no es que haya $81+9*9+9*9+9*9$ celdas! ¡Sólo hay 81 celdas que se referencian desde distintas colecciones!

En definitiva, todos los objetos están vinculados, como ya vimos en los diagramas de más arriba. Esto hace que, al construir un tablero, debamos construir también todos los demás objetos. Por lo tanto, el procedimiento para inicializar el tablero será:

- Crear 81 celdas.
- Referenciar las celdas desde el tablero.
- Crear las 9 filas, que referenciarán las celdas ya creadas.
- Crear las 9 columnas, que referenciarán las celdas ya creadas.
- Crear las 9 cajas, que referenciarán las celdas ya creadas.

Las filas y columnas las implementaremos como colecciones lineales de referencias a celdas. Las cajas y el propio tablero se suelen dibujar en forma matricial, y eso puede llevarnos a pensar que convendría implementarlos con matrices. Sin embargo, nos parece más simple implementar las cajas como colecciones lineales (su propiedad de matriz no se usa en ningún caso). El tablero podría implementarse como matriz. No obstante, al estar usando Smalltalk nos pareció más sencillo mantener una colección lineal y hacer las transformaciones que correspondieran en cada caso.

Ahora deberíamos definir las precondiciones del método *cargar*:

- El parámetro debe ser un arreglo lineal de 81 posiciones.
- Las 81 posiciones del arreglo deben contener números entre 0 y 9.

Las postcondiciones son bastante más complejas que en los casos que hemos visto hasta aquí:

- Si no se recibe un arreglo de 81 posiciones, lanzar una excepción de tipo *TamanioIncorrecto*.
- Si alguna de las posiciones del arreglo no es un número entre 0 y 9, lanzar una excepción de tipo *ValorInvalido*.
- Si se cumplen las precondiciones, se debe crear una colección de celdas, sin valor para las posiciones en que pasamos un 0 en el arreglo.
- Si se cumplen las precondiciones, se debe crear una colección de 81 celdas, con el valor pasado en el arreglo en caso de ser un valor entre 1 y 9.
- Si se cumplen las precondiciones, se debe crear una colección de 9 filas, siendo cada fila una colección de referencias a 9 celdas, que deben ser las celdas que se encuentren en esa fila.
- Si se cumplen las precondiciones, se debe crear una colección de 9 columnas, siendo cada columna una colección de referencias a 9 celdas, que deben ser las celdas que se encuentren en esa columna.
- Si se cumplen las precondiciones, se debe crear una colección de 9 cajas, siendo cada caja una colección de referencias a 9 celdas, que deben ser las celdas que se encuentren en esa caja.

Ahora debemos definir al menos una prueba unitaria por postcondición. Para la primera postcondición, escribimos las siguientes:

```
siArregloVacioCargarDebeLanzarExcepcion
    | tablero |
    tablero := Tablero new.
    [ tablero cargar: #() ]
        on: TamanioIncorrecto
        do: [ :e | Transcript show:'La prueba pasó: se pasó un arreglo vacío y lanzó TamanioIncorrecto'; cr. ^nil ].
    Transcript show:'La prueba NO pasó: se pasó un arreglo vacío y NO lanzó TamanioIncorrecto'; cr. ^nil
```

```
siArregloTieneMenosDe81elementosCargarDebeLanzarExcepcion
    | tablero |
    tablero := Tablero new.
    [ tablero cargar: #(5 3 0 0 7 0 0 0 0) ]
        on: TamanioIncorrecto
        do: [ :e | Transcript show:'La prueba pasó: se pasaron 9 elementos en el arreglo y lanzó TamanioIncorrecto'; cr. ^nil ].
    Transcript show:'La prueba NO pasó: se pasaron 9 elementos en el arreglo y NO lanzó TamanioIncorrecto'; cr. ^nil
```

```
siArregloTieneMasDe81elementosCargarDebeLanzarExcepcion
    | tablero |
    tablero := Tablero new.
    [ tablero cargar: #(5 3 0 0 7 0 0 0 0 5 3 0 0 7 0 0 0 0)
```

```

      5 3 0 0 7 0 0 0 0 5 3 0 0 7 0 0 0 0 0
      5 3 0 0 7 0 0 0 0 5 3 0 0 7 0 0 0 0 0
      5 3 0 0 7 0 0 0 0 5 3 0 0 7 0 0 0 0 0
      5 3 0 0 7 0 0 0 0 5 3 0 0 7 0 0 0 0 0
    ) ]
on: TamanioIncorrecto
do: [ :e | Transcript show:'La prueba pasó: se pasaron 90
elementos en el arreglo y lanzó TamanioIncorrecto'; cr. ^nil ].
Transcript show:'La prueba NO pasó: se pasaron 90 elementos en el
arreglo y NO lanzó TamanioIncorrecto'; cr. ^nil

```

Corremos las pruebas – una a una – y vemos que no pasan.

En consecuencia, agregamos el siguiente método en la clase *Tablero*:

```

cargar: numeros
  "verificación de tamaño:"
  ( numeros size ~= 81 )
  ifTrue: [ TamanioIncorrecto new signal ].

```

Ahora sí las pruebas pasan.

Para la segunda postcondición escribimos las siguientes pruebas:

```

siHayUnNumeroMayorQue9cargarDebeLanzarExcepcion
| tablero |
tablero := Tablero new.
[ tablero cargar: #( 5 3 0 0 7 0 0 0 0 5 3 0 0 7 0 0 0 0
  5 3 0 0 7 0 0 0 0 5 3 0 0 7 0 0 0 0
  5 3 0 0 7 0 0 0 0 5 3 0 0 7 0 0 0 13
  5 3 0 0 7 0 0 0 0 5 3 0 0 7 0 0 0 0
  5 3 0 0 7 0 0 0 0 ) ]
on: ValorInvalido
do: [ :e | Transcript show:'La prueba pasó: hay un 13 en el
arreglo y lanzó ValorInvalido'; cr. ^nil ].
Transcript show:'La prueba NO pasó: hay un 13 en el arreglo y NO lanzó
ValorInvalido'; cr. ^nil

```

```

siHayUnNumeroNegativoCargarDebeLanzarExcepcion
| tablero |
tablero := Tablero new.
[ tablero cargar: #( 5 3 0 0 7 0 0 0 0 5 3 0 0 7 0 0 0 0
  5 3 0 0 7 0 0 0 0 5 3 0 0 7 0 0 0 0
  5 3 0 0 7 0 0 0 0 5 3 0 0 7 0 0 0 -3
  5 3 0 0 7 0 0 0 0 5 3 0 0 7 0 0 0 0
  5 3 0 0 7 0 0 0 0 ) ]
on: ValorInvalido
do: [ :e | Transcript show:'La prueba pasó: hay un número negativo
en el arreglo y lanzó ValorInvalido'; cr. ^nil ].
Transcript show:'La prueba NO pasó: hay un número negativo en el arreglo
y NO lanzó ValorInvalido'; cr. ^nil

```

El código que logra que la prueba pase es:

```

cargar: numeros
    "verificación de tamaño:"
    ( numeros size ~= 81 )
        ifTrue: [ TamanoIncorrecto new signal ].
    "verificación de números entre 0 y 9:"
    numeros do: [ :num |
        ( (num < 0) | (num > 9) )
            ifTrue: [ ValorInvalido new signal ].
].

```

Ahora restan las demás postcondiciones. Para todas ellas, partiremos de un tablero correctamente construido, como el de más arriba. Para ello, escribimos un método interno a la clase *PruebasTablero*, que cargue ese tablero:

```

cargarTableroCorrecto
    tableroCorrecto := Tablero new cargar:
        # ( 5 3 0 0 7 0 0 0 0
            6 0 0 1 9 5 0 0 0
            0 9 8 0 0 0 0 6 0
            8 0 0 0 6 0 0 0 3
            4 0 0 8 0 3 0 0 1
            7 0 0 0 2 0 0 0 6
            0 6 0 0 0 0 2 8 0
            0 0 0 4 1 9 0 0 5
            0 0 0 0 8 0 0 7 9 ).

```

tableroCorrecto es un atributo (variable de instancia) de la clase *PruebasTablero*.

Debemos trabajar con cuidado aquí, ya que de la correcta construcción del tablero dependerá que muchas cuestiones del programa funcionen bien luego. Por ejemplo, no basta con asegurarnos de que en una celda quede determinado número, sino que debemos asegurarnos de que la celda en cuestión (y no una simple copia o una celda con el mismo valor) sea referenciada desde su fila, su columna y su caja.

Comencemos con la postcondición que chequea la correcta construcción de las celdas y su referenciación desde el tablero. Para ella escribiremos las siguientes pruebas:

```

siCargo3enCelda56debeQuedarCeldaConValor
    | celda |
    self cargarTableroCorrecto.
    celda := tableroCorrecto celdaFila:5 columna:6.
    ( celda contiene: 3 )
        ifTrue: [ Transcript show:'La prueba pasó: quedó una celda con
el número 3 en la celda 56'; cr. ^nil ]
        ifFalse: [ Transcript show:'La prueba NO pasó: NO quedó una celda
con el número 3 en la celda 56'; cr. ^nil ].

```

```

siCargo0enCelda29debeQuedarCeldaLibre
    | celda |
    self cargarTableroCorrecto.
    celda := tableroCorrecto celdaFila:2 columna:9.
    ( celda estaLibre )

```

```

        ifTrue: [ Transcript show:'La prueba pasó: la celda 29 quedó libre'; cr. ^nil ]
        ifFalse: [ Transcript show:'La prueba NO pasó: la celda 29 NO quedó libre'; cr. ^nil ].

```

Si ejecutamos las pruebas, las mismas fallan. Además, Pharo nos indica que no existe el método *celdaFila: columna:*. Escribimos este último así:

```

celdaFila: nFila columna: nColumna
| posicionLineal |
( (nFila < 1) | (nFila > 9) | (nColumna < 1) | (nColumna > 9) )
ifTrue: [ ValorInvalido new signal ].
posicionLineal := nColumna + (9* (nFila-1)).
^ (celdas at: posicionLineal).

```

Para que pasen las pruebas, modificamos el método *cargar*, que queda:

```

cargar: numeros
"recibe un Array con los números del tablero y crea las celdas, filas,
columnas y cajas"
"verificación de tamaño:"
( numeros size ~= 81 )
ifTrue: [ TamañoIncorrecto new signal ].
"verificación de números entre 0 y 9:"
numeros do: [ :num |
( (num < 0) | (num > 9) )
ifTrue: [ ValorInvalido new signal ].
].
celdas := OrderedCollection new.
"generación de celdas:"
numeros do: [ :num |
( num = 0 ) ifTrue: [ celdas add: (Celda new) ]
ifFalse: [ celdas add: (Celda new
colocarNúmero: num) ]].

```

Ahora sí, las pruebas pasan.

Lo siguiente que debemos probar es que cada fila quede con su contenido, pero que a la vez referencie la misma celda que se referencia directamente desde el tablero.

Para ello vamos a escribir la prueba que sigue:

```

loQueCargoEnCelda56deboEncontrarloEnFila5
| celdaEnTablero filaBuscar celdaEstaEnFila |
self cargarTableroCorrecto.
filaBuscar := tableroCorrecto fila: 5.
celdaEnTablero := tableroCorrecto celdaFila:5 columna:6.
celdaEstaEnFila := (filaBuscar contieneCelda: celdaEnTablero).
( celdaEstaEnFila )
ifTrue: [ Transcript show:'La prueba pasó: la celda desde el
tablero es la misma que a través de su fila'; cr. ^nil ]
ifFalse: [ Transcript show:'La prueba NO pasó: la celda desde el
tablero NO es la misma que a través de su fila'; cr. ^nil ].

```

Por supuesto, la prueba no pasa. Y eso ocurre porque no hemos implementado la creación de la

fila en el método *cargar*, pero también porque no tenemos el método *fila* en *Tablero*. El último lo escribimos así:

```
fila: posicion
  ^(filas at: posicion).
```

Y el método *cargar* lo modificamos, quedando así:

```
cargar: numeros
  "recibe un Array con los números del tablero y crea las celdas, filas,
  columnas y cajas"
  | unaFila colecciónParaFila |

  "verificación de tamaño:"
  ( numeros size ~= 81 )
    ifTrue: [ TamañoIncorrecto new signal ].

  "verificación de números entre 0 y 9:"
  numeros do: [ :num |
    ( (num < 0) | (num > 9) )
      ifTrue: [ ValorInvalido new signal ].
  ].

  "generación de celdas:"
  celdas := OrderedCollection new.
  numeros do: [ :num |
    ( num = 0 )      ifTrue: [ celdas add: (Celda new) ]
    ifFalse: [ celdas add: (Celda new
      colocarNúmero: num) ].
  ].

  "generación de filas:"
  filas := OrderedCollection new.
  (1 to: 9) do: [ :numFila |
    colecciónParaFila := OrderedCollection new.
    (((numFila-1)*9+1) to: (numFila*9)) do: [ :posición |
      colecciónParaFila add: (celdas at: posición) ].  

    unaFila := Fila new.
    unaFila agregarCeldas: colecciónParaFila.
    filas add: unaFila.
  ].
```

Ahora sí, las pruebas pasan.

No vamos a seguir escribiendo código aquí. Es obvio que lo que falta es muy parecido a lo anterior, pero para columnas y cajas, lo cual también alarga mucho el método *cargar*³¹. Una vez escritas las pruebas, los métodos auxiliares y el método *cargar*, terminaremos este ejercicio.

³¹ Ya volveremos sobre métodos largos en el capítulo 6.

Varios tipos de cuentas

Vamos a seguir implementando clases del ejercicio de la aplicación bancaria en Java. En esta ocasión vamos a trabajar con dos clases derivadas de *Cuenta*: *CajaAhorro* y *CuentaCorriente*, haciendo que la antigua clase *Cuenta* sea abstracta. Como dijimos más arriba, la diferencia entre una caja de ahorro y una cuenta corriente es que, mientras en la primera no podemos extraer más que el saldo de la cuenta, en una cuenta corriente podemos disponer de un acuerdo de giro en descubierto que nos permite extraer más que el saldo disponible, hasta un monto acordado con el banco.

El diagrama de clases de la figura 4.10 se reproduce aquí como figura 4.13, ya que va a guiar nuestra implementación.

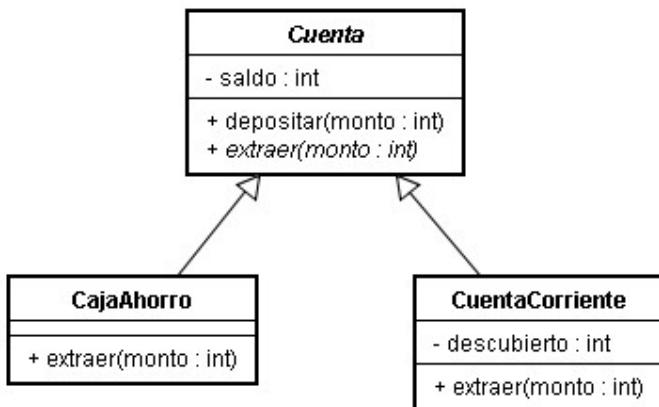


Figura 4.13 Repetición de la figura 4.10

Lo que dice este diagrama es que el método *depositar* sirve tal como lo hemos definido en la clase *Cuenta*, mientras que *extraer* debe ser definido en forma diferente en cada clase descendiente, manteniéndose abstracto en su clase madre.

Empecemos por cambiar la clase *Cuenta*, haciéndola abstracta y escribiendo un método *extraer*, también abstracto. Transcribimos únicamente las líneas que cambian:

```
public abstract class Cuenta {
    public abstract void extraer (int monto);
    ... todo el resto queda igual ...
}
```

Enseguida descubrimos que el compilador de Java nos muestra errores en nuestro programa de pruebas, ya que allí instanciábamos la clase *Cuenta*, que ahora quedó abstracta. Por lo tanto, cambiamos las líneas que dicen:

```
Cuenta cuenta = new Cuenta ( );
```

Por:

```
Cuenta cuenta = new CajaAhorro ( );
```

Por supuesto, esto nos lleva a un nuevo error: la clase *CajaAhorro* no existe. Para que pase la compilación, creamos la clase vacía, como derivada de *Cuenta*:

```
public class CajaAhorro extends Cuenta {
}
```

Sin embargo, sigue sin compilar: ¿qué ocurre? Lo que pasa es que el método *extraer*, definido

en *Cuenta* como abstracto, se hereda en *CajaAhorro*, y Java exige que una clase con métodos abstractos sea abstracta. Nosotros pretendemos que *CajaAhorro* sea una clase concreta, por lo que tenemos que redefinir – o, mejor dicho, implementar – el método *extraer*. Escribámoslo vacío para ver si compila:

```
public void extraer (int monto) { }
```

Si bien no es un método útil, dejó de ser abstracto, por lo que el programa compila sin problema.

El paso siguiente debería ser ejecutar las pruebas. Al ejecutar el programa *PruebasCuenta*, compilamos y corremos las pruebas. El resultado es que todas pasan, como se ve en la consola:

```
La prueba pasó: se pasó un monto negativo y lanzó MontoInvalido  
La prueba pasó: se pasó un monto 0 y lanzó MontoInvalido  
La prueba pasó: el saldo se incrementó correctamente al depositar  
La prueba pasó: el saldo empezó siendo 0
```

Esto es bueno y era lo esperado, ya que la caja de ahorro era la que tenía que tener el comportamiento que antes definimos en la clase *Cuenta*. Recordemos que teníamos implementados los métodos *getSaldo*, *depositar* y el constructor. Nos faltaría implementar el método *extraer*.

Las precondiciones de *extraer* son:

- El monto a extraer debe ser un número mayor que 0
- El monto a extraer debe ser menor o igual al saldo de la cuenta

Las postcondiciones son:

- Si el monto a extraer no es un número mayor que 0, lanzaremos una excepción *MontoInvalido*
- Si el monto a extraer es mayor que el saldo de la cuenta, lanzaremos una excepción *SaldoInsuficiente*
- Si las precondiciones se cumplen, el saldo de la cuenta se debe ver decrementado en el monto pasado como argumento

La primera precondición es idéntica a la del método *depositar*, así que no entraremos en detalles, escribiremos las pruebas respectivas y el método *extraer* quedará por el momento así:

```
public void extraer (int monto) {  
    if (monto <= 0)  
        throw new MontoInvalido(monto);  
}
```

La segunda postcondición hace que escribamos esta prueba:

```
private void enCajaAhorroIntentarExtraerMasQueSaldoDebeLanzarExcepcion () {  
    Cuenta cuenta = new CajaAhorro ();  
    cuenta.depositar(100);  
    try {  
        cuenta.extraer(200);  
    }  
    catch (SaldoInsuficiente e) {  
        System.out.println("La prueba pasó: se intentó extraer más que  
el saldo y lanzó SaldoInsuficiente");  
        return;  
    }  
}
```

```

    }
    System.out.println("La prueba NO pasó: se intentó extraer más que el
saldo pero no lanzó SaldoInsuficiente");
}

```

Como ya ocurrió en ocasiones similares, lo primero que nos avisa el compilador es que la clase *SaldoInsuficiente* no existe, por lo que la creamos como derivada de *RuntimeException* y recompilamos. A continuación, agregamos la invocación a *enCajaAhorroIntentarExtraerMasQueSaldoDebeLanzarExpcion* en el método *main*, y obtenemos lo que esperábamos: la prueba falla.

Por lo tanto, modificamos *extraer* para que pase:

```

public void extraer (int monto) {
    if (monto <= 0)
        throw new MontoInvalido(monto);
    if (monto > this.getSaldo())
        throw new SaldoInsuficiente();
}

```

Ahora ejecutamos la prueba y vemos que pasa.

Nos queda la última postcondición: el saldo de la cuenta debe verse decrementado en el monto pasado como argumento. Para implementarla, empezamos por escribir la prueba:

```

private void alExtraerSaldoDebeDecrementarseEnMonto() {
    Cuenta cuenta = new CajaAhorro (); // el saldo inicial es 0
    cuenta.depositar(700);
    cuenta.extraer(300);
    if (cuenta.getSaldo() == 400)
        System.out.println("La prueba pasó: el saldo se decrementó
correctamente al extraer");
    else
        System.out.println("La prueba NO pasó: el saldo no se
decrementó correctamente al extraer");
}

```

Al incorporar esta prueba en *main* y ejecutarlo, la misma falla. Proponemos entonces el siguiente código para que pase:

```

public void extraer (int monto) {
    if (monto <= 0)
        throw new MontoInvalido(monto);
    if (monto > this.getSaldo())
        throw new SaldoInsuficiente();
    this.saldo -= monto;
}

```

¡Pero esto no compila! ¿Qué pasó? El compilador de Java nos dice que el atributo *saldo* de la clase *Cuenta* no es visible aquí. Efectivamente, *saldo* era un atributo privado en *Cuenta*.

Recordar: Un atributo privado sólo se puede usar en la clase en la que está definido y en ninguna otra más, ni siquiera en una clase hija.

¿Qué debemos hacer? Ante todo, resistamos la tentación de agregar un atributo *saldo* en *CajaAhorro*, porque no es lo que necesitamos. El atributo es propio de cualquier cuenta, y si

estamos usando herencia es porque lo queremos en todas las clases descendientes, al igual que el método *getSaldo*.

Hay dos posibilidades: o cambiamos la visibilidad de *saldo* en *Cuenta* a protegida, o bien definimos un método *setSaldo*, también protegido. Ninguna es muy brillante, ya que, en el primer caso, estamos dando una visibilidad mayor a la deseada en primer término, y en el segundo, estamos definiendo un método nuevo, que no nos había sido necesario hasta ahora.

En el caso de hacer lo primero (declarar *saldo* como atributo protegido), la línea que no compilaba podría quedar como está:

```
this.saldo -= monto;
```

En el segundo (usando un método *setSaldo* en la clase *Cuenta*), la línea en cuestión quedaría:

```
this.setSaldo(this.getSaldo() - monto);
```

Ninguno de los dos nos parece suficientemente bueno, así que nos quedamos con la versión más sencilla, que es trabajar con un atributo *saldo* protegido en *Cuenta*. Ahora sí, ejecutamos las pruebas, y las mismas pasan.

Nos quedó entonces una clase *CajaAhorro* así:

```
package carlosFontela.aplicacionBancaria;
public class CajaAhorro extends Cuenta {
    public void extraer (int monto) {
        if (monto <= 0)
            throw new MontoInvalido(monto);
        if (monto > this.getSaldo())
            throw new SaldoInsuficiente();
        this.saldo -= monto;
    }
}
```

Pasemos ahora a la clase *CuentaCorriente*. En la misma, podemos usar todo lo que heredamos de *Cuenta*, salvo por el método *extraer*. De hecho, apenas definimos la clase *CuentaCorriente* como derivada de *Cuenta*, el compilador de Java nos avisa que debemos implementar *extraer*.

Para el método *extraer* de *CuentaCorriente*, las precondiciones son:

- El monto a extraer debe ser un número mayor que 0
- El monto a extraer debe ser menor o igual al saldo de la cuenta más el descubierto acordado

Y las postcondiciones:

- Si el monto a extraer no es un número mayor que 0, lanzaremos una excepción *MontoInvalido*
- Si el monto a extraer es mayor que el saldo de la cuenta más el descubierto acordado, lanzaremos una excepción *SaldoInsuficiente*
- Si las precondiciones se cumplen, el saldo de la cuenta debe verse decrementado en el monto pasado como argumento

Las dos primeras son similares al caso de la caja de ahorro, por lo que procedemos igual y ahorraremos la explicación. Para el caso de la tercera postcondición, proponemos la siguiente prueba:

```

private void enCuentaCorrienteIntentarExtraerMasQueDescubiertoDebeLanzarExcepcion() {
    CuentaCorriente cuenta = new CuentaCorriente ();
    cuenta.setDescubierto(1000);
    cuenta.depositar(100);
    try {
        cuenta.extraer(2000);
    }
    catch (SaldoInsuficiente e) {
        System.out.println("La prueba pasó: se intentó extraer más que el descubierto y lanzó SaldoInsuficiente");
        return;
    }
    System.out.println("La prueba NO pasó: se intentó extraer más que el descubierto pero no lanzó SaldoInsuficiente");
}

```

Ahora bien, esto no compila, porque el método *setDescubierto* no existe en *CuentaCorriente* ni en *Cuenta*. Lo deberíamos definir en *CuentaCorriente*. Luego de escribir las pruebas, ver que fallan y hacer que pasen, llegamos a que la clase queda así:

```

package carlosFontela.aplicacionBancaria;
public class CuentaCorriente extends Cuenta {
    private int descubiertoAcordado;
    public int getDescubierto() {
        return descubiertoAcordado;
    }
    public void setDescubierto (int descubiertoAcordado) {
        this.descubiertoAcordado = descubiertoAcordado;
    }
    public void extraer(int monto) {
    }
}

```

Ahora volvamos a la prueba. Sin embargo, si intentamos compilarla, seguimos con problemas: el compilador nos dice que *setDescubierto* no existe en *Cuenta*. ¿Cómo puede ser, si recién lo escribimos? A ver... un momento. Lo escribimos en la clase *CuentaCorriente*, no en *Cuenta*, porque sólo tiene sentido en *CuentaCorriente*. Pero luego pretendimos invocarlo con la variable *cuenta*, cuyo tipo es *Cuenta*. Como Java hace comprobación estática de tipos, esto no pasa: debemos cambiar la línea:

```
Cuenta cuenta = new CuentaCorriente ();
```

Por la línea:

```
CuentaCorriente cuenta = new CuentaCorriente ();
```

Ahora sí logramos que compile. Si ejecutamos las pruebas, la última que escribimos falla, como era de esperar. Escribamos entonces el siguiente código para el método *extraer*:

```

public void extraer(int monto) {
    if (monto <= 0)
        throw new MontoInvalido(monto);
    if (monto > this.getSaldo() + this.descubiertoAcordado)
        throw new SaldoInsuficiente();
}

```

```
        this.saldo -= monto;  
    }  
}
```

Si ahora ejecutamos la prueba, la misma pasa.

También habría que verificar que en las cuentas corrientes podemos extraer más allá del saldo, mientras tengamos descubierto disponible. Si no, estamos probando algo apenas diferente a lo que hicimos con las cajas de ahorro. Para eso, escribimos la prueba (notemos que en este caso el lanzamiento de la excepción sería un error):

```
private void alExtraerCuentaCorrientePuedoGirarEnDescubierto() {  
    CuentaCorriente cuenta = new CuentaCorriente ();  
    cuenta.setDescubierto(1000);  
    cuenta.depositar(100);  
    try {  
        cuenta.extraer(800);  
    }  
    catch (SaldoInsuficiente e) {  
        System.out.println("La prueba NO pasó: se intentó girar en  
descubierto y lanzó SaldoInsuficiente");  
        return;  
    }  
    System.out.println("La prueba pasó: permitió girar en descubierto y no  
lanzó SaldoInsuficiente");  
}
```

Si ejecutamos nuevamente el conjunto de las pruebas, todas pasan.

Ejercicios propuestos

Contenido todavía no disponible en esta versión del libro.

Recapitulación

En este capítulo nos hemos concentrado en las maneras de lograr colaboración entre objetos, con mecanismos tales como delegación y programación por diferencia. Hemos visto también que los lenguajes y sus características influyen en cómo se pueden implementar estos mecanismos, y también algunas consecuencias con implementaciones diferentes en cada uno.

En el próximo capítulo nos detendremos en las herramientas que se utilizan habitualmente para escribir pruebas unitarias automatizadas, y veremos cómo la implementación de esas herramientas se basa en los conceptos recién estudiados.

5. Hablemos de herramientas: frameworks xUnit y su uso

Contexto

Cuando presentamos diseño por contrato y desarrollo basado en pruebas, lo hicimos sin ninguna herramienta que nos ayudase. Como esta forma de trabajo es tan común, se han desarrollado herramientas muy cómodas en todos los lenguajes de POO. El objetivo de este capítulo es presentar esas herramientas y mostrar su funcionamiento.

La necesidad de herramientas

La necesidad de herramientas específicas

Ya desde el capítulo 3 venimos construyendo nuestra aplicación usando pruebas unitarias en código. Pero lo hicimos a puro pulmón, definiendo métodos que probaban los comportamientos de los objetos y luego emitían el informe en una consola. Esto no está mal: la automatización así realizada resulta muy buena, y si elegimos bien los mensajes a mostrar, son muy ilustrativos de lo que está pasando cuando una prueba falla. Sin embargo, podría ser mejor si en vez de volcar resultados por consola tuviésemos un resultado más gráfico.

Pero hay otra cuestión. A medida que fuimos escribiendo las pruebas, las fuimos reuniendo en alguna clase (como *PruebasCelda*) y además escribimos un método *ejecutarPruebas* en el cual fuimos colocando cada prueba para poder ejecutarlas todas juntas cada tanto. Esto también resulta una buena práctica. Pero sería mejor si no dependiese de nosotros el ir agregando cada prueba en el método *ejecutarPruebas*, ya que un olvido puede ser un problema serio a futuro.

Por suerte, hay gente que ya pensó en estos problemas y planteó soluciones. Una de las personas que más ayudaron en los inicios fue Kent Beck.

Primer intento: SUnit

En un antiguo artículo de 1989 (Simple Smalltalk Testing: With Patterns), Kent Beck presentó su idea de desarrollar un framework que facilitara la escritura de pruebas unitarias automatizadas en Smalltalk. En 1994 implementó su idea en un producto al que le puso por nombre SUnit.

La idea de SUnit es simple, y se basa en las siguientes premisas:

- Para escribir una clase de pruebas, debemos hacer que la misma derive de la clase *TestCase*.
- Cada método de prueba debe poder tener cualquier nombre, a condición de que empiece con la palabra *test*.
- Al escribir cada método, nos podemos ayudar con el comportamiento provisto en *TestCase*, que incluye algunos métodos de nombres *assert*, *deny*, *should*, etc.
- SUnit se ocupa de que todos los métodos *test* escritos hasta el momento en una clase derivada de *TestCase* se puedan ejecutar a la vez.

- Al correr un conjunto de pruebas en SUnit, una interfaz gráfica nos indica con color verde o rojo qué pruebas pasaron y cuáles no.

De esta manera, no nos podemos olvidar de incluir una prueba a ejecutar. Asimismo, el color es un indicativo gráfico de cómo salieron las pruebas. Además, al proveernos los métodos *assert* y afines, no tenemos que incluir los *if* que usábamos en el código de las pruebas de los capítulos anteriores.

Por ejemplo, los siguientes métodos de prueba, que en su momento escribimos:

```
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]
```

```
estaLibreDevuelveFalseLuegoDeColocarUnNumero
| celda |
celda := Celda new.
celda colocarNumero: 5.
(celda estaLibre)
    ifTrue: [Transcript show: 'La prueba NO pasó: estaLibre NO se da
cuenta de que la celda se ocupó'; cr]
    ifFalse: [Transcript show: 'La prueba pasó: estaLibre se da
cuenta de que la celda se ocupó'; cr]
```

```
getNumeroLanzaExcepcionSiNoTieneValor
| celda |
celda := Celda new.
[celda getNumero]
on: ValorInvalido
do: [ :e | Transcript show: 'La prueba pasó: se pidió el valor de
una celda recién creada y lanzó ValorInvalido'; cr. ^nil ].
Transcript show: 'La prueba NO pasó: se pidió el valor de una celda
recién creada y no lanzó ValorInvalido'; cr. ^nil
```

Quedarían escritos así en SUnit, dentro de una clase que derive de *TestCase*:

```
testUnaCeldaSeCreaLibre
| celda |
celda := Celda new.
self assert: (celda estaLibre) description: 'La prueba NO pasó: la
celda recién creada está ocupada'
```

```
testEstaLibreDevuelveFalseLuegoDeColocarUnNumero
| celda |
celda := Celda new.
celda colocarNumero: 5.
```

```

    self deny: (celda estaLibre) description: 'La prueba NO pasó: estaLibre
NO se da cuenta de que la celda se ocupó'

```

```

testGetNumeroLanzaExcepcionSiNoTieneValor
| celda |
celda := Celda new.
self should: [celda getNumero]
raise: ValorInvalido
description: 'La prueba NO pasó: se pidió el valor de una celda
recién creada y no lanzó ValorInvalido'

```

Como se ve, cambiamos los *if* por el envío de mensajes tipo *assert*, *deny* y *should/raise*, además de que tuvimos que agregar la palabra *test* al principio del nombre de cada método de prueba. Como ventajas, no hay más *if* y además no hay forma de olvidarse de incluir las llamadas a los métodos de prueba, ya que SUnit se ocupa de invocarlos por nosotros.

SUnit viene con una utilidad más. En los ejercicios que venimos haciendo desde el capítulo 3, vimos que muchos métodos de prueba comparten un mismo código al principio, que generalmente sirve para crear y dar valores iniciales a ciertos objetos sobre los que vamos a operar. Para facilitar esas tareas de inicialización, SUnit nos permite escribir un método *setUp*, que va a ser invocado automáticamente antes de cada prueba, y también un método *tearDown* que se va a invocar al terminar cada una. Nosotros no vamos a usar demasiado estos métodos, por dos razones: no siempre tenemos la misma inicialización en todos los métodos; y además, creemos que invocar explícitamente los métodos, usando asimismo nombres más descriptivos, queda más claro en las pruebas. Pero por supuesto, es una cuestión de gustos, y para quien guste, allí están.

El funcionamiento de SUnit se entiende mejor si miramos el diagrama de clases de la figura 5.1 y el de secuencia de la figura 5.2.

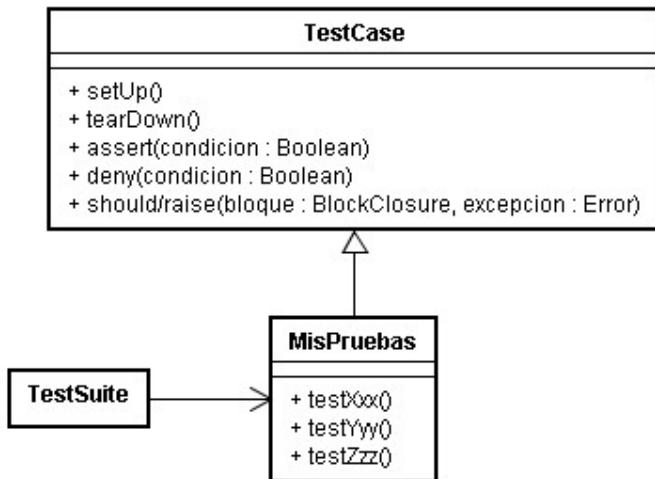


Figura 5.1 Diagrama de clases de SUnit

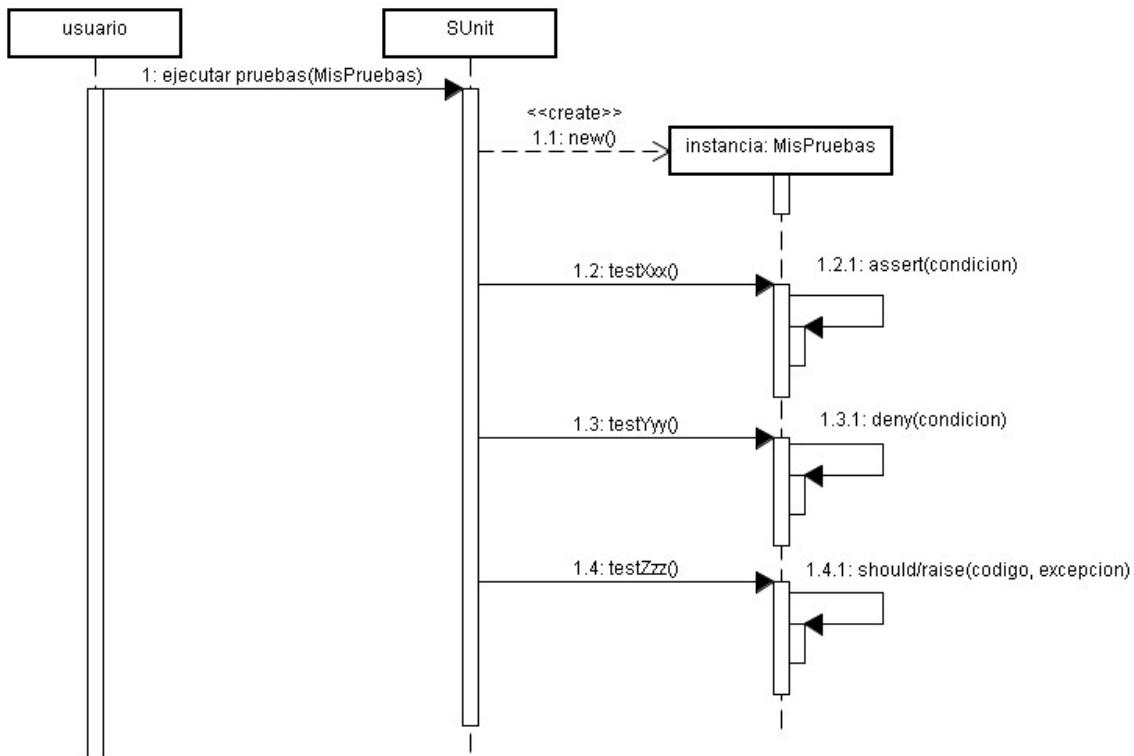


Figura 5.2 Diagrama de secuencia de SUnit

En definitiva, lo que ocurre es que el usuario le pide a SUnit que ejecute las pruebas de la clase derivada de *TestCase* que elija (en la figura, *MisPruebas*), SUnit crea una instancia de esa clase y llama sucesivamente a los métodos que encuentra en ella que comienzan con la palabra *test*. Cada uno de estos métodos, podrá hacer sus *assert*, sus *deny* o sus *should/raise*, lanzando una excepción si algo falla, que a su vez provocará que SUnit muestre o no el error.

Digresión:

SUnit es un **framework**.

La diferencia entre biblioteca y framework es muy sutil para muchos profesionales. Sin embargo, una biblioteca brinda servicios a través de clases y funciones listas para usar, mientras que un framework brinda servicios a través de comportamientos por defecto que se pueden cambiar o ampliar a través de clases provistas por el programador.

Dicho de otra manera, una biblioteca está lista para ser usada mediante la invocación de sus servicios, mientras que un framework es un programa que invoca a los comportamientos definidos por el programador.

Los herederos

Parece ser que el segundo framework de pruebas unitarias automatizadas fue JUnit, para el lenguaje Java. El mismo fue desarrollado por Erich Gamma y Kent Beck³². La idea del primer JUnit fue la misma de SUnit, solamente que con la notación de Java.

³² Cuentan los amantes de las epopeyas que escribieron el código de JUnit en un vuelo transatlántico.

Luego hubo desarrollos de estos frameworks – a los que genéricamente y con escasa originalidad se denomina xUnit – en varios lenguajes. NUnit es el de .NET, CppUnit el de C++, PyUnit el de Python, Test::Unit el de Ruby, JSUnit el de JavaScript, y también hay variantes más elaboradas, como TestNG para Java y MSTest para .NET.

Integración en los IDEs

Un paso interesante que se dio bastante naturalmente fue la integración de los frameworks xUnit en los entornos de desarrollo.

Por ejemplo, si el lector vino siguiendo lo que mostrábamos más arriba, escribiéndolo en Pharo, habrá visto que Pharo le ofrece la opción “Run tests” en el menú contextual de la clase de prueba. Al elegir esa opción, Pharo ejecuta todas las pruebas y nos muestra en verde o rojo si pasaron o no.

Lo mismo ocurre en Java, plataforma en la cual el archipopular entorno Eclipse viene con JUnit incluido y muestra con barras de colores la ejecución exitosa o fallida de las pruebas. Otros entornos de desarrollo para Java también incluyen JUnit.

Se imaginará el lector que otro tanto ocurre con muchos entornos de desarrollo y frameworks de pruebas unitarias automatizadas.

La evolución: NUnit y JUnit 4

Con el paso del tiempo, hubo desarrolladores que se quejaban de la necesidad de hacer que la clase de pruebas heredase de una clase en particular, sobre todo en lenguajes que sólo admiten herencia simple y eso les impedía heredar de cualquier otra clase. Otra queja se planteaba con la necesidad de tener que poner *test* delante del nombre de todos los métodos, lo cual afectaba a quienes escribían código en idiomas distintos del inglés, además de a quienes empezaron a cuestionar metodológicamente el uso de los frameworks xUnit (tema que analizaremos más adelante).

NUnit fue el primer framework que eliminó este requisito. Y al poco tiempo, JUnit 4 siguió sus pasos. A partir de allí, estos dos frameworks se basaron en anotaciones agregadas a las clases y a los métodos para indicar que se trataba de clases y métodos de prueba.

Veremos este cambio directamente en el ejercicio con JUnit en las próximas páginas.

Herramientas de cobertura

Otra necesidad que surgió a medida que se generalizaba el uso de los frameworks xUnit fue alguna herramienta que mostrase si el código era cubierto o no por las pruebas, a medida que éstas se ejecutaban. Así surgieron las herramientas de cobertura, que también fueron siendo incorporadas a los entornos de desarrollo.

Por ejemplo, tomando nuestras pruebas en JUnit de más adelante, y usando la herramienta EclEmma sobre Eclipse, podríamos ver el reporte de la figura 5.3.

The screenshot shows the Eclipse IDE interface. The top part displays the Java code for `CuentaCorriente.java`. The code includes methods for getting and setting the overdraft amount, checking if withdrawal is possible, and withdrawing money. Lines 8, 16, and 21 are highlighted in red, indicating they were not covered by tests. The bottom part shows the EclEmma coverage report table:

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
carlosFontela.aplicacionBancaria	81,4 %	79	18	97
CuentaCorriente.java	60,0 %	27	18	45
CajaAhorro.java	100,0 %	24	0	24
Cuenta.java	100,0 %	22	0	22
MontoInvalido.java	100,0 %	3	0	3
SaldoInsuficiente.java	100,0 %	3	0	3

Figura 5.3 Uso de EclEmma en Eclipse

El uso de herramientas de cobertura es bastante habitual hoy por hoy. Los análisis de cobertura se usan como una medida de la calidad de las pruebas: a mayor cobertura, las pruebas del programa son más exhaustivas, y por lo tanto existen menos situaciones de posibles errores que no están siendo probadas. No obstante, no hay que confundir con la calidad del programa: la cobertura mide sólo la calidad de las pruebas, indirectamente, y sólo muy subsidiariamente afecta la calidad del programa.

Por lo tanto, el nivel de cobertura no debería guiar el desarrollo. Sólo tiene sentido usarlo para ver, a posteriori, cuán cubiertos estamos ante regresiones. Porque si bien decimos que es una medida de la calidad de las pruebas, ni siquiera la mide en cuanto a su adecuación para guiar el desarrollo.

Otra buena idea es usar las métricas de cobertura para observar tendencias. Por ejemplo, si una determinada métrica de cobertura disminuye con el tiempo, se puede deber a que la aplicación creció sin que se escribieran las pruebas correspondientes, que algunas pruebas fueron eliminadas, o un poco de cada cosa.

Sin embargo, hay que tener cuidado, ya que el simple hecho de que una prueba recorra una línea de código no significa que todas las ramas y condiciones estén cubiertas.

Además, los reportes de porcentaje de cobertura son engañosos. Es común que algunos desarrolladores, con el objetivo de mejorar sus indicadores de cobertura, busquen introducir pruebas sencillas, que casi no agregan valor, pero que mejoren su grado de cobertura.

Por otro lado, si bien lograr un 100% parece ser a priori una buena meta, no es así teniendo en cuenta los costos frente a los beneficios que podemos obtener. Habitualmente, un objetivo

razonable es llegar a un 80% o 90%. Tampoco es bueno imponer un valor de grado de cobertura único para todos los desarrollos, como una condición de liberación del software. Lo más recomendable es hacer un análisis de costos y beneficios para cada caso.

Finalmente, aun en el marco de un mismo proyecto, suele haber zonas de diferente criticidad. Las zonas de mayor criticidad requerirán un grado de cobertura mayor, tal vez cercana al 100%, mientras que las zonas menos críticas tal vez admitan un grado de cobertura mucho menor.

Ejercicio adicional

Vamos a resolver en ejercicio de cuentas bancarias, tanto en Smalltalk como en Java, usando frameworks xUnit. Si bien vamos a seguir todos los pasos del método que hemos explicado en el capítulo 4, no los iremos transcribiendo exhaustivamente para no aburrir a los lectores.

Ejemplo con SUnit

El ejercicio a resolver ya lo hicimos en Java, aunque sin usar JUnit. Vamos a implementar las clases *Cuenta*, *CajaAhorro* y *CuentaCorriente*, con las mismas firmas, precondiciones y postcondiciones con que lo hicimos en Java, pero ahora en Smalltalk.

Al trabajar con SUnit, lo que debemos hacer es escribir una clase de pruebas, que llamaremos *PruebasCuentas*, derivada de la clase *TestCase*.

```
TestCase subclass: #PruebasCuentas
```

En cada método de prueba, que debe empezar con la palabra *test*, usaremos los métodos *assert* o *should/raise*, con el agregado de un parámetro *description*, que sirve para mostrar un mensaje en el caso de que la prueba no pase. Con todo esto en mente, definimos los métodos de la clase *PruebasCuentas*, de a uno a la vez, escribiendo el código que hace que esa prueba pase. El resultado de este proceso es (sólo mostramos las pruebas SUnit):

```
testAlCrearUnaCuentaSuSaldoDebeSerCero
    | cuenta |
    cuenta := CajaAhorro new.
    self assert: (cuenta getSaldo = 0) description: 'La prueba NO pasó: el saldo no empezó siendo 0'
```

```
testAlDepositarSaldoDebeIncrementarseEnMonto
    | cuenta |
    cuenta := CajaAhorro new. "el saldo comienza en cero"
    cuenta depositar: 100.
    self assert: (cuenta getSaldo = 100) description: 'La prueba NO pasó: el saldo no se incrementó correctamente al depositar'.
```

```
testIntentarDepositarMontoCeroDebeLanzarExcepcion
    | cuenta |
    cuenta := CajaAhorro new.
    self should: [cuenta depositar:0] raise: MontoInvalido description: 'La prueba NO pasó: se pasó un monto 0 y no lanzó MontoInvalido'.
```

```
testIntentarDepositarMontoNegativoDebeLanzarExcepcion
```

```

| cuenta |
cuenta := CajaAhorro new.
self should: [cuenta depositar: (-100)] raise: MontoInvalido
description: 'La prueba NO pasó: se pasó un monto 0 y no lanzó MontoInvalido'.

```

```

testAlExtraerSaldoDebeDisminuirEnMonto
| cuenta |
cuenta := CajaAhorro new.
cuenta depositar: 200.
cuenta extraer: 50.
self assert: (cuenta getSaldo = 150) description: 'La prueba NO pasó:
el saldo no disminuyó correctamente al extraer'.

```

```

testAlExtraerCuentaCorrientePuedoGirarEnDescubierto
| cuenta |
cuenta := CuentaCorriente new.
cuenta setDescubierto: 1000.
cuenta depositar: 100.
self shouldnt: [ cuenta extraer: 800 ] raise: SaldoInsuficiente
description: 'La prueba NO pasó: se intentó girar en descubierto y lanzó
SaldoInsuficiente'.

```

```

testIntentarExtraerMontoCeroDebeLanzarExcepcion
| cuenta |
cuenta := CajaAhorro new.
self should: [cuenta extraer: 0] raise: MontoInvalido description: 'La
prueba NO pasó: se pasó un monto 0 y no lanzó MontoInvalido'.

```

```

testIntentarExtraerMontoNegativoDebeLanzarExcepcion
| cuenta |
cuenta := CajaAhorro new.
self should: [cuenta extraer: (-100)] raise: MontoInvalido description:
'La prueba NO pasó: se pasó un monto 0 y no lanzó MontoInvalido'.

```

```

testIntentarExtraerMasQueSaldoCajaAhorroDebeLanzarExcepcion
| cuenta |
cuenta := CajaAhorro new.
cuenta depositar: 100.
self should: [ cuenta extraer: 200 ] raise: SaldoInsuficiente
description: 'La prueba NO pasó: se pasó un monto mayor que el saldo y no
lanzó SaldoInsuficiente'.

```

```

testIntentarExtraerMasQueDescubiertoCuentaCorrienteDebeLanzarExcepcion
| cuenta |
cuenta := CuentaCorriente new.
cuenta setDescubierto: 1000.

```

```

cuenta depositar: 100.
self should: [ cuenta extraer:2000 ] raise: SaldoInsuficiente
description: 'La prueba NO pasó: se pasó un monto mayor que el saldo y no
lanzó SaldoInsuficiente'.

```

Decíamos que las herramientas de pruebas unitarias automatizadas suelen venir integradas en los entornos de desarrollo. Por ejemplo, en Pharo, al terminar de escribir pruebas y código, se ve como en la figura 5.4.

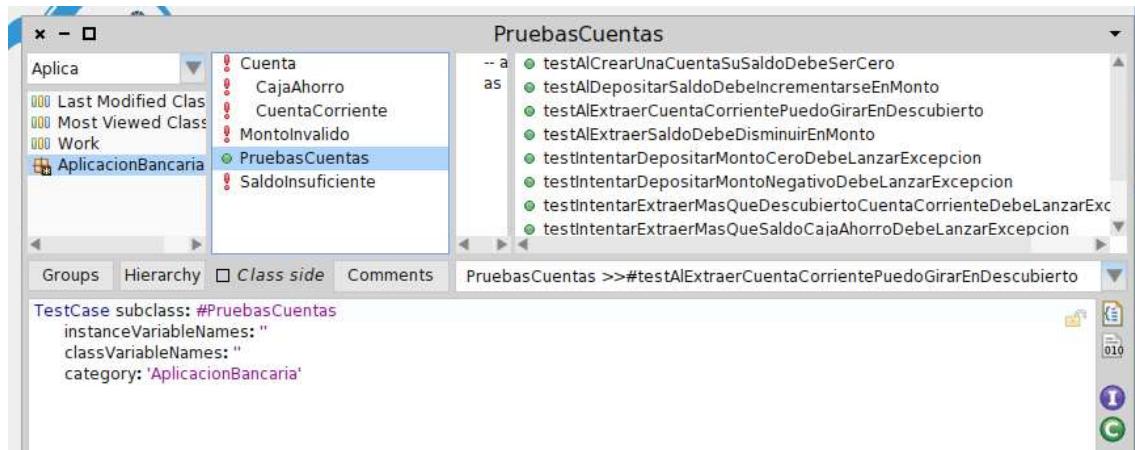


Figura 5.4 Integración de SUnit con Pharo

El punto verde al lado de la clase *PruebasCuentas* indica que es una clase de pruebas, cosa que descubre por sí mismo Pharo al ver que es hija de *TestCase*. Ese mismo punto puede usarse para correr todas las pruebas de la clase, haciendo un clic sobre el mismo. Los métodos de prueba, que Pharo identifica por empezar con la palabra *test*, también tienen un punto a la izquierda, que será verde si la prueba pasó, grisado si todavía no se ejecutó y rojo si la prueba falló. Por supuesto, este punto puede accionarse para ejecutar una prueba en forma aislada.

Hay otros mecanismos para ejecutar pruebas, como hacer clic en el botón derecho y elegir “Run tests” en el menú contextual. También se muestra un cartel verde o rojo bien visible que muestra si las pruebas pasaron o no. Invitamos al lector a abrir Pharo y probar todo esto para convencerse.

Ejemplo con JUnit

En Java ya tenemos programadas las tres clases desde el capítulo anterior. Por lo tanto, no tiene sentido ir siguiendo nuevamente el procedimiento que venimos utilizando.

Sin embargo, podemos seguir un procedimiento inverso, que es el que utilizan quienes agregan pruebas al código que no las tenía: en vez de usar las pruebas como garantía de buen funcionamiento del código (que en este caso ya sabemos que funciona), usar el código como garantía de buen funcionamiento de las pruebas. Por lo tanto, iremos escribiendo las pruebas de a una y verificando que pasen.

Vamos a usar JUnit 4 y Eclipse. En esta versión del popular framework, la clase de pruebas no tiene que tener ningún nombre en especial ni heredar de ninguna otra clase, sino que utiliza anotaciones.

Para decir que un método es una prueba se lo debe anotar con *@Test*. Si deseamos chequear el lanzamiento de una excepción podemos hacerlo mediante la misma anotación, con el agregado del parámetro *expected*. Con la sola presencia de un método anotado con *@Test*, Eclipse se da

cuenta de que es una clase de prueba.

Por ejemplo, escribamos la clase *PruebasCuentas* y el primero de los métodos de prueba (notemos que los métodos de prueba deben ser públicos, ya que serán utilizados por JUnit, desde afuera de la clase en la que están declarados):

```
package carlosFontela.aplicacionBancaria.pruebas;
import org.junit.Assert;
import org.junit.Test;
import carlosFontela.aplicacionBancaria.*;
public class PruebasCuentas {
    @Test
    public void alCrearUnaCuentaSuSaldoDebeSerCero() {
        Cuenta cuenta = new CajaAhorro ();
        Assert.assertEquals("La prueba NO pasó: el saldo no empezó siendo
0", 0, cuenta.getSaldo());
    }
}
```

Notemos que el método *assertEquals* (que sólo es uno de los tantos de la clase *Assert*), tiene tres parámetros: una descripción que sirve como mensaje para el caso de que la prueba falle, el valor esperado y el valor obtenido de la ejecución de la prueba.

Si ahora hacemos clic derecho, aparece el menú contextual que, como muestra la figura 5.5, nos permite ejecutar las pruebas.

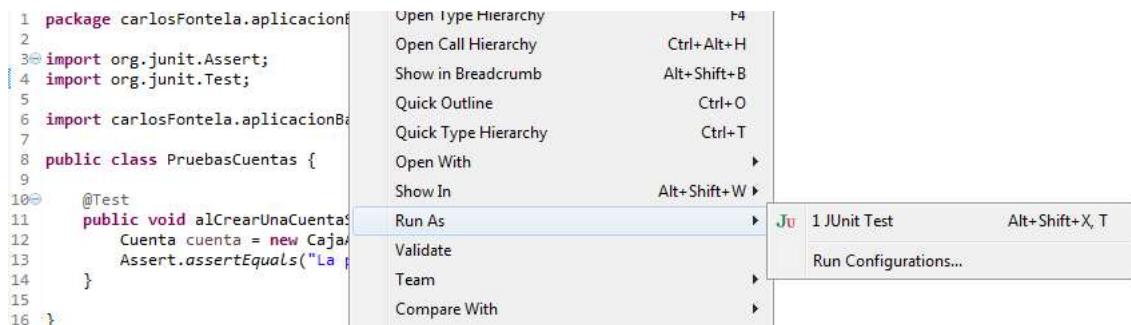


Figura 5.5 Integración de JUnit con Eclipse

Si ejecutamos la prueba, obtenemos la barra verde, como muestra la figura 5.6.

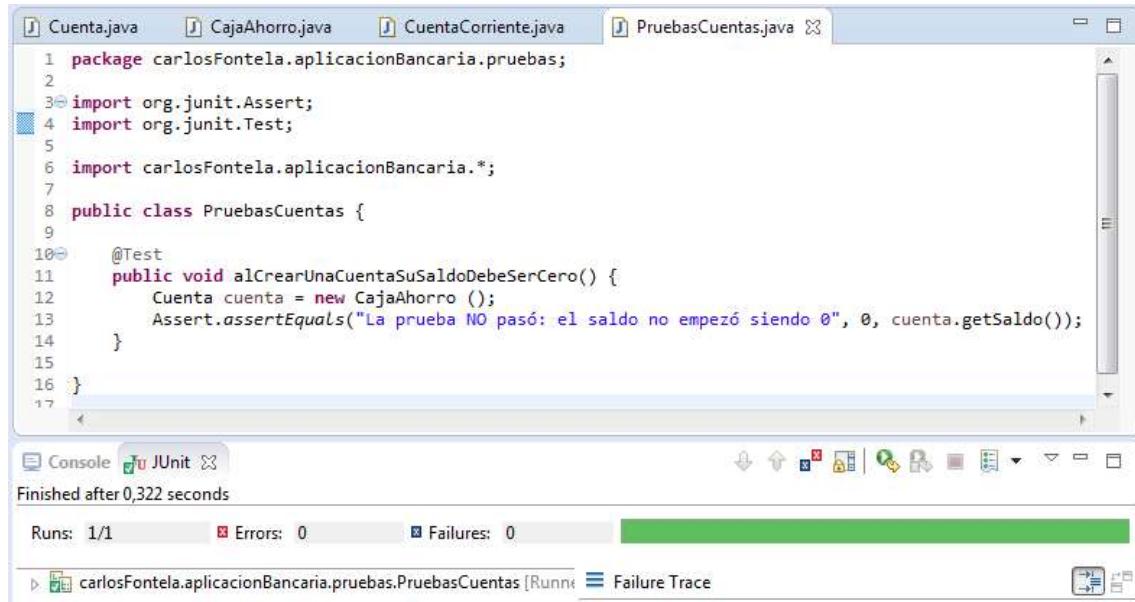


Figura 5.6 Integración de JUnit con Eclipse, con la barra verde

Sigamos escribiendo todos los métodos de prueba necesarios, verificando por cada uno que se obtenga la barra verde, hasta llegar a la clase *PruebasCuentas* que sigue:

```
package carlosFontela.aplicacionBancaria.pruebas;

import org.junit.Assert;
import org.junit.Test;
import carlosFontela.aplicacionBancaria.*;

public class PruebasCuentas {

    @Test
    public void alCrearUnaCuentaSuSaldoDebeSerCero() {
        Cuenta cuenta = new CajaAhorro ();
        Assert.assertEquals("La prueba NO pasó: el saldo no empezó siendo 0", 0, cuenta.getSaldo());
    }

    @Test (expected = MontoInvalido.class)
    public void alDepositarSiMontoEsNegativoDebeLanzarExcepcion() {
        Cuenta cuenta = new CajaAhorro ();
        cuenta.depositar(-200);
    }

    @Test (expected = MontoInvalido.class)
    public void alDepositarSiMontoEsCeroDebeLanzarExcepcion() {
        Cuenta cuenta = new CajaAhorro ();
        cuenta.depositar(0);
    }
}
```

```

    @Test
    public void alDepositarSaldoDebeIncrementarseEnMonto() {
        Cuenta cuenta = new CajaAhorro (); // el saldo inicial es 0
        cuenta.depositar(100);
        Assert.assertEquals("La prueba NO pasó: el saldo no se incrementó
correctamente al depositar", 100, cuenta.getSaldo());
    }

    @Test (expected = SaldoInsuficiente.class)
    public void enCajaAhorroIntentarExtraerMasQueSaldoDebeLanzarExpcion
() {
        Cuenta cuenta = new CajaAhorro ();
        cuenta.depositar(100);
        cuenta.extraer(200);
    }

    @Test (expected = SaldoInsuficiente.class)
    public void enCuentaCorrienteIntentarExtraerMasQueDescubiertoDebeLanzarExpcion() {
        CuentaCorriente cuenta = new CuentaCorriente ();
        cuenta.setDescubierto(1000);
        cuenta.depositar(100);
        cuenta.extraer(2000);
    }

    @Test (expected = MontoInvalido.class)
    public void alExtraerSiMontoEsNegativoDebeLanzarExpcion() {
        Cuenta cuenta = new CajaAhorro ();
        cuenta.extraer(-200);
    }

    @Test (expected = MontoInvalido.class)
    public void alExtraerSiMontoEsCeroDebeLanzarExpcion() {
        Cuenta cuenta = new CajaAhorro ();
        cuenta.extraer(0);
    }

    @Test
    public void alExtraerSaldoDebeDecrementarseEnMonto() {
        Cuenta cuenta = new CajaAhorro (); // el saldo inicial es 0
        cuenta.depositar(700);
        cuenta.extraer(300);
        Assert.assertEquals("La prueba NO pasó: el saldo no se decrementó
correctamente al extraer", 400, cuenta.getSaldo());
    }

    @Test
    public void alExtraerCuentaCorrientePuedoGirarEnDescubierto() {

```

```

        CuentaCorriente cuenta = new CuentaCorriente ();
        cuenta.setDescubierto(1000);
        cuenta.depositar(100);
        try {
            cuenta.extraer(800);
        }
        catch (SaldoInsuficiente e) {
            Assert.fail("La prueba NO pasó: se intentó girar en
descubierto y lanzó SaldoInsuficiente");
        }
    }
}

```

Algo más que nos muestra Eclipse es un área con las salidas de JUnit, como vemos en la figura 5.7, en la cual cada uno de los métodos de prueba que pasaron están acompañados de una viñeta de éxito en verde (en caso de no pasar, nos encontraríamos con una X en rojo).

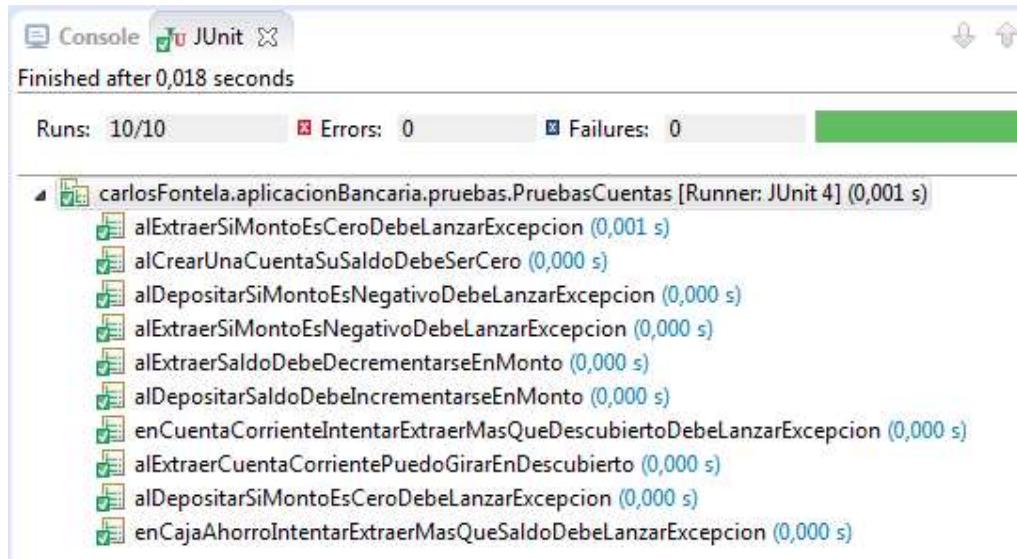


Figura 5.7 Integración de JUnit con Eclipse, mostrando método por método

Ejercicios propuestos

Contenido todavía no disponible en esta versión del libro.

Recapitulación

Este capítulo presentó la familia de herramientas xUnit, la más popular para la definición de pruebas unitarias automatizadas, de modo que ya estamos en condiciones de seguir nuestra práctica metodológica de una manera sencilla y cómoda.

El siguiente capítulo se va a detener en uno de los conceptos centrales de la POO, el polimorfismo, y veremos que cada lenguaje tiene maneras diferentes de implementarlo y también consecuencias derivadas de la elección de forma de implementación.

6. Polimorfismo: los objetos se comportan a su manera

Contexto

Hemos hablado de polimorfismo casi desde el comienzo del libro. Sin embargo, tratándose de un principio tan importante de la POO, no podíamos dejar de dedicarle un capítulo completo.

Encima, las decisiones de implementación de los lenguajes impactan en el comportamiento del polimorfismo. Ello ha llevado a algunos equívocos que debemos evitar, y en los cuales nos vamos a detener un poco.

De regreso al polimorfismo

¿Qué es polimorfismo?

En el capítulo 2 dijimos que el polimorfismo es la capacidad que tienen distintos objetos de responder de maneras diferentes a un mismo mensaje. Puede haber autores que no coincidan en un 100% con esta definición, pero para nosotros es suficiente.

Por ejemplo, en nuestra implementación del Sudoku, cuando enviamos el mensaje *contiene* a una celda, una caja, una fila o una columna, la implementación del método podrá – o no – ser diferente en cada caso, aunque la semántica (lo que pretendemos que ocurra como respuesta a ese mensaje) es la misma: que un valor booleano nos indique si el número pasado como parámetro se encuentra o no en esa celda, fila, columna o caja.

El nombre polimorfismo proviene de “muchas formas”, y se refiere a la variedad de maneras de responder a un mismo mensaje por parte de objetos distintos.

También en el ejercicio de las cuentas bancarias tenemos un ejemplo de polimorfismo: el mensaje *extraer*. Cuando ese mensaje se envíe a una caja de ahorro, verificará que el saldo sea mayor o igual que el monto a extraer; en cambio, si ese mensaje se le envía a una cuenta bancaria, la verificación a realizar será diferente, ya que tendrá en cuenta la posibilidad de giro en descubierto.

En estos dos casos, el polimorfismo es muy obvio. Veamos uno más sutil.

El método *contiene* de la clase *Celda* lo hemos escrito así:

```
contiene: valor
    | valorEnCelda |
    (libre)
        ifTrue: [ ^false ].
    [ valorEnCelda := self getNumero ]
        on: ValorInvalido
        do: [ ^false ].
    ^(valorEnCelda = valor)
```

Con esto decimos que cada vez que se envíe el mensaje *contiene* a cualquier instancia de la clase *Celda* va a responder con el mismo comportamiento.

Un momento... ¿responde realmente cada objeto de la misma manera? Dicho de otra manera,

¿responde igual una celda libre que una ocupada?

El código que se ejecuta en el caso de una celda que está libre es:

```
^false
```

Mientras que si la celda está ocupada se ejecuta el fragmento de código que sigue:

```
[ valorEnCelda := self getNumero ]
    on: ValorInvalido
    do: [ ^false ].
^(valorEnCelda = valor)
```

Algún lector avisado que a la vez no sepa tanto de objetos, diría que estamos siendo extremistas, mientras que un lector de mente más orientada a objetos diría que sí, que son dos comportamientos diferentes, que varían en función del estado del objeto.

Para ver de qué lado nos quedamos, dejemos Smalltalk y su implementación basada en clases por un instante: ¿cómo haríamos en un lenguaje como JavaScript?

Piense un poco antes de seguir leyendo...

En JavaScript podríamos pensar que tenemos dos prototipos:

```
celdaLibre = {
    contiene : function (valor) {
        return false;
    }
}
```

```
celdaOcupada = {
    contiene : function (valor) {
        return (this.numero == valor);
    }
}
```

Ahora son dos objetos diferentes, y ambos difieren en la respuesta que dan. ¿Hay o no polimorfismo? Según la definición de más arriba, sí, sin duda.

¿Cómo llevaríamos lo anterior, de un lenguaje con prototipos a un lenguaje basado en clases?

Piense un poco antes de seguir leyendo...

La respuesta es que en los lenguajes basados en clases, si queremos separar los dos comportamientos en respuestas diferentes, deberíamos definir dos clases, una para celdas libres y otra para celdas ocupadas.

Otra opción es guardar el estado de libre u ocupado en un objeto externo, y delegar en él el comportamiento, de modo que sería el estado el que se podría implementar con dos clases, como se muestra en la figura 6.1.

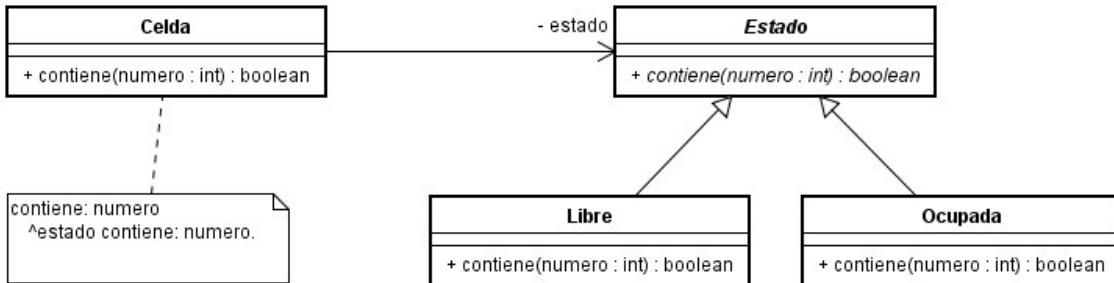


Figura 6.1 Estado representado mediante objetos

Tal vez para el no habituado a la POO parezca un poco excesivo, pero puede haber razones para hacerlo (hay patrones de diseño que se implementan así, como veremos más adelante).

Quienes piensan la POO desde la perspectiva de los lenguajes basados en clases, hablan de polimorfismo cuando distintos objetos responden de maneras diferentes a la llegada de un mensaje, basándose en la clase de la cual son instancias. Pero notemos que esto sólo es una limitación propia de los lenguajes basados en clases.

Hay definiciones aún más restrictivas de polimorfismo, como aquélla que dice que el polimorfismo se da solamente en presencia de herencia. Ya volveremos sobre esto para refutarlo, pero lo que sí podemos notar es que no es un concepto inequívoco.

Por lo tanto, establecemos la siguiente:

Definición (definitiva y personal): polimorfismo

Llamamos polimorfismo a la posibilidad de que distintos objetos respondan de manera diferente ante la llegada del mismo mensaje.

Definición: mensaje polimorfo

Un mensaje es polimorfo cuando la respuesta al mismo puede ser diferente en función del objeto receptor.

¿Por qué es importante el polimorfismo?

El polimorfismo permite que nos refiramos al mismo mensaje por su nombre y que – sin embargo – le demos implementaciones diferentes según el objeto que se trate. Eso tiene como ventaja, en primer lugar, la legibilidad del código, lo cual redunda en su facilidad de mantenimiento. En nuestros ejemplos, está claro lo que significa *contiene(valor)*, se refiera a una celda, una fila, una columna o una caja.

También evita la repetición de código, al permitir enviar el mismo mensaje a distintos objetos sin importar cómo van a responder. Por ejemplo, cuando escribimos, para un objeto fila:

```

contiene: numero
| encontrado |
( (numero < 1) | (numero > 9) )
    ifTrue: [ ValorInvalido new signal ].
encontrado := false.
celdas do: [:celda | (celda contiene: numero)
    ifTrue: [ encontrado := true ] ].
^encontrado.

```

No nos importa cómo está implementado el método *contiene* en cada celda. Podría ser un solo método (como hicimos en la versión inicial en Smalltalk) o dos métodos distintos (como mostramos en el caso de JavaScript), o incluso delegando en un objeto “estado”, como en el diagrama de la figura 6.1.

Asimismo, en ocasiones el polimorfismo lleva a que desaparezca código de comprobación como vimos en dos de los tres ejemplos recientes, donde la comprobación de si la celda está libre u ocupada la podemos remover del método, ya que la propia celda sabe si está o no ocupada. Esto también hace que el código sea más simple y sencillo de leer, y por lo tanto más económico de mantener.

Casos que se resuelven con polimorfismo

Hay algunos casos típicos que se resuelven con polimorfismo, y que mostraremos con el ejemplo de las cuentas bancarias.

El primer caso típico se da cuando tenemos que recorrer una colección y enviarle el mismo mensaje a todos los objetos de la misma.

Supongamos que se da el caso de que tenemos una lista de todas las cuentas del banco, y a todas ellas necesitamos debitarle una suma fija, digamos 100 pesos. El fragmento de código que resuelve esto podría ser:

```
listaCuentas do:  
    [ :c | c extraer:100 ].
```

En este caso, gracias al polimorfismo, no nos preocupa analizar si la cuenta en cuestión es una caja de ahorro o una cuenta corriente: con que exista el método *extraer* en dichas cuentas, lo invocamos y ya está.

El segundo caso se da cuando un mensaje enviado a un objeto debe a su vez delegar en otro mensaje al mismo objeto. Antes de mostrarlo, haremos un pequeño cambio a nuestro código. Recordemos que en nuestra clase *CajaAhorro*, el método *extraer* estaba implementado así:

```
extraer: monto  
    ( monto <= 0 ) ifTrue: [MontoInvalido new signal].  
    ( monto > (self getSaldo) ) ifTrue: [SaldoInsuficiente new signal].  
    saldo := saldo - monto
```

Y en la clase *CuentaCorriente*, así:

```
extraer: monto  
    ( monto <= 0 ) ifTrue: [MontoInvalido new signal].  
    ( monto > (self getSaldo + self getDescubierto) ) ifTrue:  
        [SaldoInsuficiente new signal].  
    saldo := saldo - monto
```

Sin embargo, acá hay repetición de código. En ambos métodos, lo único que cambia es el chequeo de la suficiencia del saldo. Si eso es así, podríamos colocar el método *extraer* directamente en la clase *Cuenta*, madre de ambas, así:

```
extraer: monto  
    ( monto <= 0 ) ifTrue: [MontoInvalido new signal].  
    ((self extraccionPosible:monto) not) ifTrue: [SaldoInsuficiente new signal].  
    saldo := saldo - monto
```

Y ahora tendríamos un método *extraccionPosible*, que en la clase *CajaAhorro* sería:

```
extraccionPosible: monto
    ^ ( monto <= (self getSaldo) ).
```

Y en la clase *CuentaCorriente* quedaría implementado así:

```
extraccionPosible: monto
    ^ ( monto <= (self getSaldo + self getDescubierto) ).
```

Ahora bien, esto es polimorfismo nuevamente. Un objeto de la clase *Cuenta* simplemente envía el mensaje *extraccionPosible*, y cada objeto responderá de manera diferente a ese mensaje, según se trate de una caja de ahorro o una cuenta corriente.

¡Un momento! Todo esto está muy bien, pero hay una manera un poco mejor de hacerlo: ¿y si en vez de *extraccionPosible* usamos un mensaje más simple, como *disponible*, sin argumentos? Como veremos al hablar de calidad de código, siempre es mejor que los métodos tengan la menor cantidad posible de parámetros, porque eso facilita la legibilidad y las pruebas.

Veamos cómo quedaría *extraer*:

```
extraer: monto
    ( monto <= 0 ) ifTrue: [MontoInvalido new signal].
    ( monto > disponible ) ifTrue: [SaldoInsuficiente new signal].
    saldo := saldo - monto
```

Y ahora tendríamos un método *disponible*, que en la clase *CajaAhorro* sería:

```
disponible
    ^ ( self getSaldo ).
```

Y en la clase *CuentaCorriente* quedaría implementado así:

```
disponible
    ^ ( self getSaldo + self getDescubierto ).
```

¿No es más legible?

Recomendación adicional:

En casos como este, es una buena práctica declarar al método *disponible* como abstracto en la clase madre, aunque en lenguajes de comprobación dinámica no es forzoso.

Es importante notar que el polimorfismo es una propiedad que se da sin que debamos hacer nada en especial, al menos en los lenguajes en los que estamos trabajando. Es prácticamente transparente, y hasta podríamos haber ignorado su existencia. Esta es una afirmación fuerte: ¿estamos seguros? Sigamos leyendo...

Polimorfismo y vinculación tardía

Habitualmente se vincula al polimorfismo con algo llamado *vinculación tardía o dinámica*. Es decir, al hablar de polimorfismo, se plantea que el mismo funciona porque la decisión del método a invocar se toma en tiempo de ejecución y no antes.

Hasta este momento, nosotros no hemos necesitado decir nada de esto, y sin embargo hablamos de polimorfismo. ¿Por qué?

Piense un poco antes de seguir leyendo...

En realidad, los lenguajes interpretados, como JavaScript o Python, no tienen otra posibilidad: sólo es posible determinar cuál método llamar durante la ejecución. A su vez, en el caso de los lenguajes de comprobación en tiempo de ejecución, como Smalltalk, la vinculación tardía se da

de modo natural, ya que al no comprobarse tipos en tiempo de compilación, lo que logramos es diferir la decisión hasta el tiempo de ejecución. Quedan, sin embargo, los lenguajes de comprobación estática: ¿qué pasará en ellos? La respuesta es: depende...

En Java, por ejemplo, la vinculación tardía es el comportamiento por defecto. De allí que en nuestros ejemplos en Java pudimos ignorar la existencia del polimorfismo o, lo que es lo mismo, no hicimos nada de manera explícita para que el comportamiento fuese polimorfo.

Sin embargo, hay una manera de indicar que no deseamos vinculación dinámica en Java: colocarle a la firma del método la cláusula *final*, que indicaría que el método no se puede redefinir, y por lo tanto el compilador puede vincular ese método en forma estática. No es algo deseable en la generalidad de los casos, pero a veces existen optimizaciones que incluyen este paso. Nosotros no lo haremos en este libro, entendiendo que el comportamiento esperado es la vinculación tardía.

Otros lenguajes, como ocurre con C# y C++, consideran a la vinculación estática como el comportamiento por defecto. En estos casos, cláusulas como *virtual* y *override* indican al compilador cuándo aplicar la vinculación dinámica. De allí que quienes usan lenguajes como estos se vean obligados a explicar el polimorfismo como una excepcionalidad que debe ser tenida en cuenta.

Digresión

Siendo el polimorfismo uno de los pilares de la POO, la manera en que lo implementan estos lenguajes (C++ y C#) nos parece una decisión que aporta más confusión que ventajas.

En definitiva, la necesidad de definir o no si un método debe ser polimorfo es una cuestión de implementación. En lenguajes interpretados es transparente, lo mismo que en los lenguajes de comprobación dinámica. Por otro lado, los lenguajes de comprobación estática pueden asumir el polimorfismo por defecto – como Java – o precisar que se haga en forma explícita.

Polimorfismo y herencia: ¿realmente deben ir juntos?

Otra extraña conexión que aparece cuando hablamos de POO es aquella entre polimorfismo y herencia. Incluso hay autores que afirman que el polimorfismo es una consecuencia de la herencia³³, y otros que – aun cuando admiten la posibilidad de polimorfismo sin herencia – proclaman que usar polimorfismo sin herencia es una mala idea. ¿De dónde vienen estas nociones excéntricas? Probablemente, de nuevo, de los lenguajes de comprobación estática. Veámoslo en Java, que es el lenguaje que conocemos.

El fragmento de código que recorre una lista de cuentas, traducido a Java y agregándole un poco de contexto, quedaría:

```
Collection <Cuenta> cuentas = new ArrayList <Cuenta> ( );
...
for (Cuenta c : cuentas)
    c.extraer(100);
```

Es decir, primero declaramos a la variable *cuentas* como una colección de instancias de *Cuenta*. Eso es información para el compilador, que va a comprobar que no coloquemos en la colección referenciada por *cuentas* nada que no sea una instancia de *Cuenta*.

Luego, en el ciclo que recorre la colección, debemos usar una variable para ir referenciando

³³ Cf. [Meyer 1994] y [Eckel 2003].

cada uno de los elementos. Como en Java cada variable debe tener un tipo, escribimos *Cuenta* delante de *c*. Esto va a hacer que el compilador haga comprobación de tipos sobre esa variable *c*. En la última línea, al enviar el mensaje *extraer*, el compilador va a chequear que la clase de la variable *c* tenga un método con ese nombre.

Como todas las comprobaciones se hacen en tiempo de compilación, y debido a que *CajaAhorro* y *CuentaCorriente* derivan de *Cuenta*, el compilador no tiene inconveniente alguno con el fragmento de código anterior, y en tiempo de ejecución *c* podrá referenciar tanto a instancias de *CajaAhorro* como a instancias de *CuentaCorriente*. En efecto, *c.extraer()* es un mensaje válido porque los objetos referenciados por *c* sólo van a poder ser instancias de *Cuenta* o de una clase descendiente.

Pero en Smalltalk no hay tanta verificación. Ya vimos el código equivalente en Smalltalk y no había necesidad de herencia:

```
cuentas := OrderedCollection new.  
...  
cuentas do:  
    [ :c | c extraer:100 ].
```

En este caso, el compilador no hace ninguna comprobación. Por lo tanto, en tiempo de ejecución la colección *cuentas* podrá tener cualquier tipo de objetos, y al ir enviando el mensaje *extraer* a cada elemento, no interesa si se trata o no de una instancia de *Cuenta*: sólo necesitamos saber que es un objeto que entiende el mensaje *extraer* con un parámetro. Por eso, en Smalltalk, el polimorfismo y la herencia no tienen por qué ir de la mano.

Pero, ¿por qué querríamos usar polimorfismo sin herencia? Veamos un ejemplo concreto. Supongamos que por cada depósito que se hace en una cuenta, el sistema debe enviar un aviso al cliente. Entonces, *depositar* podría quedar así:

```
depositar: monto  
    ( monto <= 0 ) ifTrue: [MontoInvalido new signal].  
    saldo := saldo + monto.  
    cliente notificarDeposito: monto.
```

Sería cuestión de hacer que el objeto *cliente* pudiera comprender un mensaje *notificarDeposito*, para ser notificado. Bien, hasta ahora no hay dificultades.

Ahora consideremos que la agencia de recaudación de impuestos (la AFIP en la Argentina) también requiere que se le notifiquen los depósitos recibidos en las cuentas, con fines de evaluar posibles evasiones impositivas. Ahora necesitamos que la agencia de recaudación también entienda el mensaje *notificarDeposito*.

Finalmente, asumamos que también el oficial de cuenta del cliente requiere ser notificado, con lo cual también debe recibir mensajes como *notificarDeposito*.

El diagrama de la figura 6.2 nos muestra este escenario.

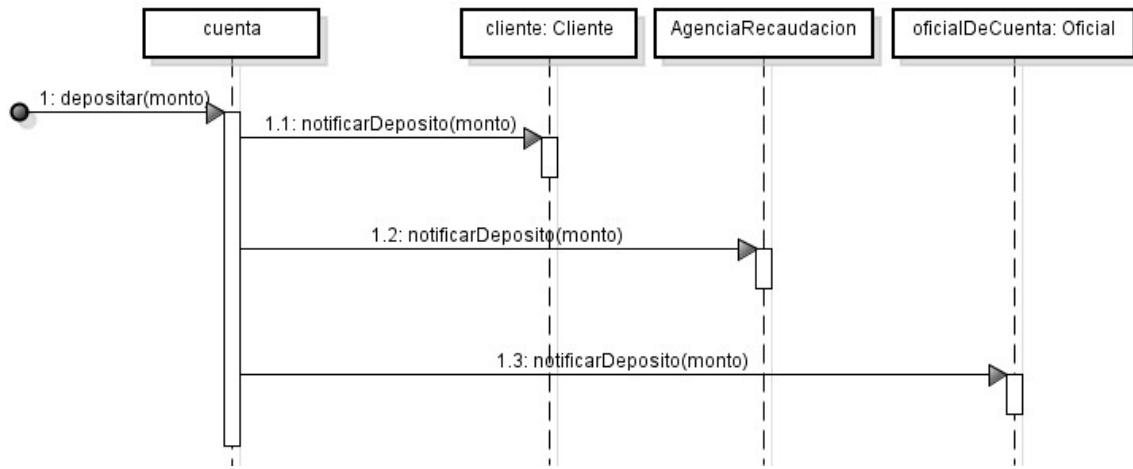


Figura 6.2 Notificaciones idénticas a instancias de clases disjuntas

Una forma de implementar esto es que la cuenta mantenga una lista de entidades a las que debe notificar. En Smalltalk, esto podría llevar a que el método *depositar* se escriba así:

```

depositar: monto
    ( monto <= 0 ) ifTrue: [MontoInvalido new signal].
    saldo := saldo + monto.
    entidadesNotificar do: [:e | e notificarDeposito: monto].

```

Nuevamente, vemos que en Smalltalk no hay problema, ya que el compilador no hará comprobación alguna, y en tiempo de ejecución deberíamos garantizar que cada uno de los elementos de la lista referenciada por *entidadesNotificar* entiende el mensaje *notificarDeposito*. Pero en Java necesitamos escribir algo como:

```

public void depositar(int monto) {
    if (monto <= 0)
        throw new MontoInvalido();
    this.saldo += monto;
    for (Entidad e : entidadesNotificar)
        e.notificarDeposito(monto);
}

```

El tema es que esto no funciona a menos que las clases *Cliente*, *AgenciaRecaudacion* y *Oficial* deriven todas de *Entidad*, y allí definir el método *notificarDeposito*, aunque más no sea como método abstracto.

Por esto es que hay autores que afirman que no conviene usar polimorfismo sin herencia. Lo que estos autores no ven es que es muy artificial que tres clases tan disjuntas como *Cliente*, *AgenciaRecaudacion* y *Oficial* deriven todas de una misma clase madre *Entidad* sólo porque necesitan un método en común. Además, en lenguajes de herencia simple, esa herencia nos impediría derivar de otras clases.

Lo que queremos hacer notar aquí es que el vínculo entre polimorfismo y herencia es, simplemente, una cuestión de implementación. Por eso es que – como veremos enseguida – varios lenguajes de comprobación estática de tipos hayan optado por tratar de solucionar el problema con alguna construcción especial.

Desde ya, en lenguajes sin clases – y por lo tanto sin herencia – como JavaScript, el polimorfismo se da sin herencia, de modo totalmente transparente.

Métodos abstractos y comprobación estática

Más arriba, cuando ilustramos uno de los casos típicos que funcionan gracias al polimorfismo, al mover el método *extraer* a la clase *Cuenta* dijimos que el método *disponible*, definido en las clases *CajaAhorro* y *CuentaCorriente*, convenía definirlo como abstracto en la clase *Cuenta*.

En un lenguaje como Smalltalk, esto es una recomendación. En efecto, si no lo hiciésemos así, no obligaríamos a las futuras clases hijas a implementar ese método. Y en ese caso, si alguna vez se invoca el método *extraer* para una instancia de una nueva clase que no entienda el mensaje *disponible*, vamos a tener una excepción en tiempo de ejecución.

Ahora bien, en Java no hay opción: debemos declarar *disponible* en *Cuenta*, al menos como método abstracto. ¿Por qué?

Piense un poco antes de seguir leyendo...

Lo que pasa es que el compilador, al ver que desde el método *extraer*, definido en la clase *Cuenta*, se invoca al método *disponible*, no va a poder comprobar que *disponible* exista en esa clase, y por lo tanto, no va a compilar. Por eso en los lenguajes de comprobación estática esta recomendación pasa a ser una obligación.

En definitiva, los métodos abstractos sirven, en los lenguajes de comprobación dinámica, como un medio para obligar a las clases descendientes a implementar ese comportamiento. En los lenguajes de comprobación estática, hay ocasiones en que no podemos sino definir ciertos métodos abstractos si queremos que funcione el polimorfismo.

Polimorfismo sin herencia en lenguajes de comprobación estática: interfaces

Como decíamos más arriba, varios lenguajes de comprobación estática de tipos han previsto el polimorfismo sin herencia. Java es uno de ellos, y el mecanismo que utiliza se denomina *interfaces*.

Podemos pensar en una *interfaz* como una clase especial:

- Abstracta
- Con todos los métodos abstractos
- Sin conservación de estado³⁴
- Que admite la herencia múltiple (esto es, que una clase puede derivar de una clase madre y una o más interfaces)
- Que puede ser heredada desde varias interfaces, con herencia simple

Otra manera de ver a una interfaz es simplemente como un conjunto de métodos abstractos.

En la jerga de POO, se suele decir que una clase *implementa* una interfaz, en vez de decir que deriva o hereda. Por las reglas de Java, de que toda clase con un método abstracto debe ser abstracta, se concluye lo que sigue, en forma sucesiva:

- Si una clase implementa una interfaz, hereda sus métodos, que pasan a ser abstractos salvo que se redefinan como concretos.
- Como una clase con métodos abstractos debe ser abstracta, si se desea que una clase que

³⁴ Esto es “casi” equivalente a decir que no tiene atributos. Sin embargo, en Java, está permitido que una interfaz tenga atributos, en la medida que los mismos almacenen constantes.

implementa una interfaz sea concreta, debemos redefinir (implementar) todos sus métodos.

- Una interfaz define un contrato para todas las clases que la implementan, ya que especifica los mensajes que las instancias de esas clases deben entender.
- Todas las clases concretas que implementan una interfaz tienen en común los métodos declarados en dicha interfaz.

Por ejemplo, si en Java reescribimos el método *depositar* de la siguiente manera:

```
public void depositar(int monto) {  
    if (monto <= 0)  
        throw new MontoInvalido();  
    this.saldo += monto;  
    for (Notificable e : entidadesNotificar)  
        e.notificarDeposito(monto);  
}
```

Notificable deberá ser definida así:

```
public interface Notificable {  
    public void notificarDeposito (int monto);  
}
```

Y en ese caso, *entidadesNotificar*, que será una instancia de *Collection<Notificable>*, admitirá como elementos a cualquier instancia de una clase que implemente la interfaz *Notificable*. Y como cualquier clase que implemente *Notificable* va a entender el mensaje *notificarDeposito*, esto funciona sin problemas.

Por ejemplo, si ahora deseamos que la clase *Cliente* implemente la interfaz, su declaración será:

```
public class Cliente implements Notificable { ... }
```

Por supuesto, *Cliente*, para ser concreta, necesita implementar el método *notificarDeposito*. Es decir:

```
public class Cliente implements Notificable {  
    ...  
    public void notificarDeposito (int monto) {  
        ...  
    }  
    ...  
}
```

El diagrama de clases correspondiente podría ser el de la figura 6.3, donde se ve que la implementación de interfaces se indica igual que la herencia, aunque con línea de puntos.

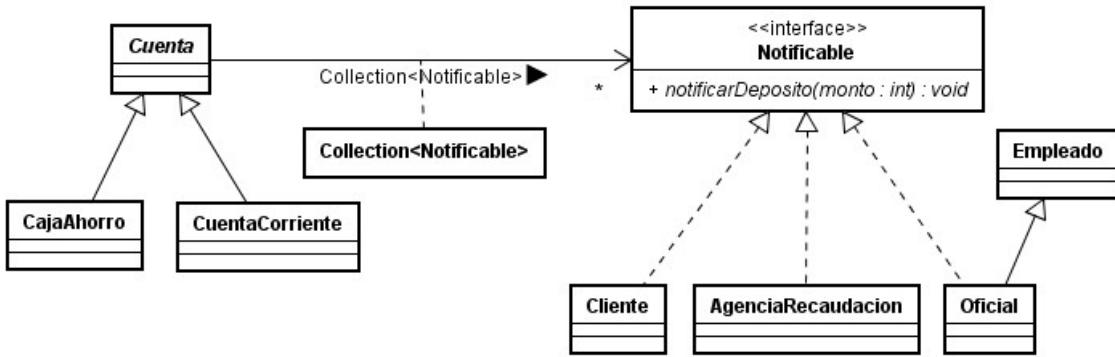


Figura 6.3 Interfaces en un diagrama de clases

Allí vemos cómo una clase, *Oficial*, hereda de la clase *Empleado* y a la vez implementa la interfaz *Notifiable*.

Algunas propiedades interesantes de las interfaces de Java

Ya hemos dicho que en Java está permitido almacenar la referencia a un objeto de una clase en una variable cuyo tipo es una clase ancestro de dicha clase. Eso vale tanto para **clases concretas** como abstractas. Por ejemplo, la inicialización que sigue es válida, según ya vimos:

```
Cuenta c = new CajaAhorro();
```

Por supuesto, esto que es válido para las clases, lo es también para las interfaces. Por ejemplo, teniendo en cuenta el diagrama de clases de la figura 6.3, podríamos escribir:

```
Notifiable n = new Oficial();
```

O incluso:

```
Oficial oc = new Oficial();
Empleado e = oc;
Notifiable n = oc;
```

En este último caso, hay un único objeto en tiempo de ejecución, que es accedido mediante tres referencias distintas, cada una almacenada en variables de distinto tipo. El tipo de cada variable va a servir para que el compilador compruebe el uso que hacemos del objeto. Por ejemplo, cuando usamos al objeto mediante la referencia almacenada en la variable *n*, el compilador va a restringir el uso: sólo podremos enviar el mensaje *notificarDeposito*. Si – en cambio – accedemos desde la variable *e*, vamos a poder enviarle los mensajes definidos en la clase *Empleado*. Mientras tanto, si accedemos al objeto mediante la variable *oc*, vamos a poder enviarle cualquier mensaje definido en la clase *Oficial*.

Esto mismo se puede aplicar a los parámetros. Por ejemplo, si en nuestra clase *Cuenta* tuviésemos un método cuya firma es:

```
public void agregarEntidad (Notifiable n)
```

Ese método va a poder recibir como argumento cualquier objeto instancia de una clase que implemente *Notifiable*. Obvio que eso limita lo que podemos hacer con ese objeto dentro del método *agregarEntidad*, pero también estamos pudiendo definir un método más genérico, que aplique para cualquier objeto de cualquier clase, con la sola condición de que implemente *Notifiable*, lo equivale a decir, para cualquier objeto que entienda el mensaje *notificarDeposito*.

Por lo tanto, el uso de interfaces como tipos de las variables lo que hace es definir qué mensajes

van a entender los objetos que referencien las mismas, y por lo tanto, vamos a poder construir código más genérico.

Un ejemplo de la biblioteca de Java es la propia *Collection* que usamos más arriba. En efecto, *Collection* es una interfaz, que define ciertos métodos. Por lo tanto, si bien podemos usar variables y parámetros cuyo tipo sea *Collection* – con lo cual estariamos restringiendo el uso de esa variable o parámetro al uso de los métodos definidos en la interfaz – a la hora de instanciar un objeto sólo puede ser mediante una clase.

Por eso podemos escribir:

```
ArrayList <Cuenta> cuentas = new ArrayList <Cuenta> ( );
// ArrayList es una clase, y cuentas entiende todos los mensajes de la clase
```

O bien:

```
Collection <Cuenta> cuentas = new ArrayList <Cuenta> ( );
// Collection es una interfaz: ahora cuentas está restringida a usar sólo
los métodos definidos en la interfaz
```

E incluso así, ya que *Collection* deriva de *Iterable*:

```
Iterable <Cuenta> cuentas = new ArrayList <Cuenta> ( );
// Iterable es una interfaz más restringida que Collection: ahora cuentas
está restringida a usar sólo el método definido en Iterable
```

Notemos que en Smalltalk nada de esto es necesario, ya que el compilador no hace comprobación de tipos.

Ejercicio adicional

Vamos a trabajar con el ejercicio del Sudoku, terminando de definir para las clases *Columna* y *Caja* lo que ya hicimos en la clase *Fila*.

Comencemos con el método *contiene* de la clase *Columna*. El procedimiento ya lo aplicamos en el capítulo 4, con la clase *Fila*, que es muy similar. El único cambio será que ahora vamos a usar SUnit.

Por lo tanto, definimos la clase *PruebasColumna*, como derivada de *TestCase*, y escribimos las pruebas que siguen:

```
testContieneDevuelveFalseSiElNumeroNoEstaEnCaja
    | columna colecciónAuxiliar |
    columna := Columna new.
    colecciónAuxiliar := OrderedCollection new.
    colecciónAuxiliar add: (Celda new colocarNúmero: 5).
    colecciónAuxiliar add: (Celda new colocarNúmero: 3).
    colecciónAuxiliar add: (Celda new).
    colecciónAuxiliar add: (Celda new).
    colecciónAuxiliar add: (Celda new colocarNúmero: 7).
    colecciónAuxiliar add: (Celda new colocarNúmero: 1).
    colecciónAuxiliar add: (Celda new).
    colecciónAuxiliar add: (Celda new colocarNúmero: 9).
    colecciónAuxiliar add: (Celda new).
    columna agregarCeldas: colecciónAuxiliar.
    self deny: (columna contiene: 8) description: 'La prueba NO pasó: la
columna no tiene el número y contiene devuelve true'.
```

```

testContieneDevuelveTrueSiElNumeroEstaEnColumna
    | columna colecciónAuxiliar |
    columna := Columna new.
    colecciónAuxiliar := OrderedCollection new.
    colecciónAuxiliar add: (Celda new colocarNúmero: 5).
    colecciónAuxiliar add: (Celda new colocarNúmero: 3).
    colecciónAuxiliar add: (Celda new).
    colecciónAuxiliar add: (Celda new).
    colecciónAuxiliar add: (Celda new colocarNúmero: 7).
    colecciónAuxiliar add: (Celda new colocarNúmero: 1).
    colecciónAuxiliar add: (Celda new).
    colecciónAuxiliar add: (Celda new colocarNúmero: 9).
    colecciónAuxiliar add: (Celda new).
    columna agregarCeldas: colecciónAuxiliar.
    self assert: (columna contiene: 5) description: 'La prueba NO pasó: la
columna tiene el número y contiene devuelve false'.

```

El código del método *contiene* será:

```

contiene: numero
    | encontrado |
    ( (numero < 1) | (numero > 9) )
        ifTrue: [ ValorInvalido new signal ].
    encontrado := false.
    celdas do: [:celda | (celda contiene: numero)
        ifTrue: [ encontrado := true ] ].
    ^encontrado.

```

Con un constructor:

```

initialize
    celdas := OrderedCollection new.

```

Pasemos a la clase *Caja* y su método *contiene*. Una caja, más allá de la forma que tenga en el Sudoku, no lineal sino cuadrada, puede verse como una colección de celdas. Si lo hacemos así, las pruebas serán idénticas a las de *Columna*, y por lo tanto también el método *contiene* y el constructor.

Siguiendo el mismo procedimiento, el código de los métodos de la clase *Caja* será:

```

contiene: numero
    | encontrado |
    ( (numero < 1) | (numero > 9) )
        ifTrue: [ ValorInvalido new signal ].
    encontrado := false.
    celdas do: [:celda | (celda contiene: numero)
        ifTrue: [ encontrado := true ] ].
    ^encontrado.

```

```

initialize
    celdas := OrderedCollection new.

```

Ahora debemos crear la clase abstracta *ColeccionCeldas*, con el método *contiene* abstracto, y hacer que las tres clases, *Fila*, *Columna* y *Caja*, hereden de ella. Lo hacemos, y en *ColeccionCeldas* definimos:

```
contiene: numero  
          self subclassResponsibility.
```

Luego cambiamos la declaración de cada clase. Por ejemplo, en *Fila* pondremos:

```
ColeccionCeldas subclass: #Fila
```

Lo mismo hacemos con *Columna* y *Caja*.

Bien: ¿ya está? Parece que nos estamos olvidando de algo. ¿Qué debemos hacer luego de cada cambio en una clase madre? Deberíamos ejecutar las pruebas de las clases hijas. Por lo tanto, las ejecutamos, y siguen funcionando bien.

Por el momento, el ejercicio sigue andando bien. Ya volveremos a él en el próximo capítulo.

Ejercicios propuestos

Contenido todavía no disponible en esta versión del libro.

Recapitulación

Este capítulo sirvió para volver sobre un concepto central de POO, el polimorfismo. También pudimos ver que la relación entre herencia y polimorfismo es más mítica que real, y que varios lenguajes tienen construcciones especiales para desacoplar ambos conceptos, como el muy especial de las interfaces.

En el próximo capítulo nos introduciremos a otra práctica metodológica, la refactorización, que se usa para evitar que el código degradado nos impida el mantenimiento de nuestros programas.

7. Paréntesis metodológico: refactorización y cierre de TDD

Contexto

Cuando trabajamos de la manera en que lo venimos haciendo en el libro, es posible que la calidad del código se vaya deteriorando, haciendo que el mismo sea cada vez más difícil de mantener. Este capítulo muestra una práctica metodológica, que se puede aplicar tanto en pequeñas dosis a lo largo del desarrollo como en gran escala cuando nuestra base de código ya está degradada.

Cómo evitar la degradación del código

Mantenimiento y entropía

Todo producto de software exitoso es modificado. La aseveración es de Fred Brooks [Brooks 1987], hace ya más de tres décadas, y se basa en:

- El software es un producto relativamente fácil de modificar.
- Los negocios exitosos tienden a innovar para mantenerse en el mercado.
- Los clientes que usan productos con los cuales están conformes, cada vez quieren más de esos productos.
- Si un producto es exitoso, deberá ser portado a nuevas plataformas a medida que las mismas van surgiendo.
- Si un producto es exitoso, deberá ser adaptado para cumplir nuevas reglamentaciones o costumbres.

Si hace más de tres décadas esto era así, hoy, cuando el software está detrás de toda actividad humana, de toda máquina, de todo proceso industrial y de nuestros aparatos más cotidianos, y cuando los usuarios se tornan cada vez más exigentes en sus demandas de funcionalidad, disponibilidad, experiencia de uso, eficiencia, etc., es todavía más cierta.

El problema es que, con cada cambio que le hacemos al software, éste se degrada en cuanto a calidad del código. No importa qué tan bien estuvo diseñado el producto al comienzo y cuánto hayamos hecho para prever los cambios, siempre va a ocurrir que algún cambio no haya sido previsto o que deba ser hecho contrarreloj o que alguna mano inexperta o que no estuvo en el diseño original, modifique ese diseño.

Todos conocemos suficientes casos como para ejemplificar. Los clientes son muy ingeniosos y nos piden cambios que no esperábamos. A veces es el mercado o alguna regulación los que nos obligan a introducir una modificación en un sentido no previsto y en un tiempo que no da para mantener la prolijidad. Los desarrolladores se encuentran en una situación de tensión entre realizar el cambio en el menor tiempo y con el menor costo posible, y mantener una buena calidad de código. Y pasa siempre que las personas que trabajan en los proyectos van rotando, con lo cual la visión inicial del diseño va quedando cada vez menos presente.

A medida que se introducen cambios sin cuidar el diseño, la calidad decae. Y cuando la calidad decae, se va haciendo cada vez más difícil modificar el código. Esto, a su vez, hace que los

nuevos cambios cada vez lo dejen en peor estado, en un proceso que se realimenta indefinidamente.

Si cada cambio degrada la calidad del código, muchos cambios llevarían la calidad a niveles cada vez peores. Esto es lo que se llama *entropía del software*: una degradación de la calidad inexorablemente continua y creciente.

¿Y qué podemos hacer?

Refactorización al rescate

Para mantener acotada la entropía en el código, hay una técnica que se llama refactorización.

Definición: refactorización como técnica

Refactorización es una técnica que busca mejorar la calidad del código con vistas a facilitar su mantenimiento, pero sin alterar el comportamiento observable del mismo.

También se usa el mismo término para referirse a cada uno de los cambios que se hacen en el mismo sentido.

Definición: refactorización como un cambio

Una refactorización es un cambio realizado sobre el código de un sistema de software de modo tal que no se afecte su comportamiento observable y con el sólo fin de mejorar la legibilidad, la facilidad de comprensión, la extensibilidad u otros atributos de calidad interna con vistas al mantenimiento.

Hay tres elementos en las definiciones anteriores:

- Se trata de un cambio hecho al código.
- Se hace para mejorar la calidad del código con el objetivo de facilitar su mantenimiento. Esto implica que el código debe entrar a la refactorización como está y debe salir de mejor calidad, entendiendo por mejor calidad aquella que hace más sencillas las modificaciones futuras.
- No altera el comportamiento observable. Esto implica que nada que se haga en una refactorización puede cambiar lo que el sistema venía haciendo. La única finalidad es la mejora del código, no el cambio de comportamiento.

Por lo tanto, una refactorización debería mejorar la calidad del código y –consecuentemente– limitar y disminuir la entropía. Cuanto más frecuentes sean las refactorizaciones, más controlada va a estar la entropía, y más sencillo será introducir nuevas modificaciones al sistema.

En la jerga de los desarrolladores, se dice que algo en el diseño “huele mal” cuando ciertas cosas en el código violan principios de diseño conocidos, de allí que a estas situaciones se las llame “oler” o “malos olores” (“smells” o “bad smells”, en inglés). Como pasa con el olor de los alimentos, un mal olor no es necesariamente un problema que amerita una acción, aunque sí es un poderoso indicio.

¿Ya hemos hecho alguna refactorización en el libro? Sí, así es, y varias. Tal vez una de las más notorias fue la extracción de parte del código del método *extraer* a un nuevo método, denominado *disponible*, en el ejercicio de la aplicación bancaria. Notemos que fue un cambio en el código, sin cambiar el comportamiento del método *extraer*, y que eliminó una repetición de código. Ya volveremos sobre este mismo ejemplo.

Condiciones de una refactorización segura

El problema con las refactorizaciones es que son riesgosas, ya que estamos cambiando código que sabemos que funciona por otro que – aunque presumimos que va a ser de mejor calidad – no sabemos si funcionará igual que antes.

Existen varios modos de asegurar la preservación del comportamiento luego de una refactorización, como los métodos analíticos y las pruebas. Los métodos analíticos, y en especial los de análisis estático, son los que utilizan las herramientas de refactorización automática.

Pero no toda refactorización es posible de resolver por métodos analíticos ni automatizable. Por eso se suelen usar pruebas para asegurar la preservación del comportamiento. En definitiva, para permitir refactorizaciones más seguras y confiables, una buena táctica es trabajar con pruebas unitarias automatizadas, escritas antes de refactorizar, y correrlas después para asegurarnos que nuestro programa sigue funcionando tal como lo hacía antes del cambio.

Definición (incompleta pero práctica):

Una refactorización es correcta si, luego de la misma, las pruebas que antes funcionaban siguen funcionando sin problema.

Por lo tanto, el procedimiento para hacer una refactorización con pruebas, es:

- Si no hay pruebas que verifiquen el comportamiento, escribirlas antes de la refactorización.
- Ejecutar las pruebas para asegurarse de que están funcionando bien.
- Realizar la refactorización.
- Volver a ejecutar las pruebas para asegurarse de que no hubo cambios de comportamiento.

El diagrama de actividades sería el de la figura 7.1.

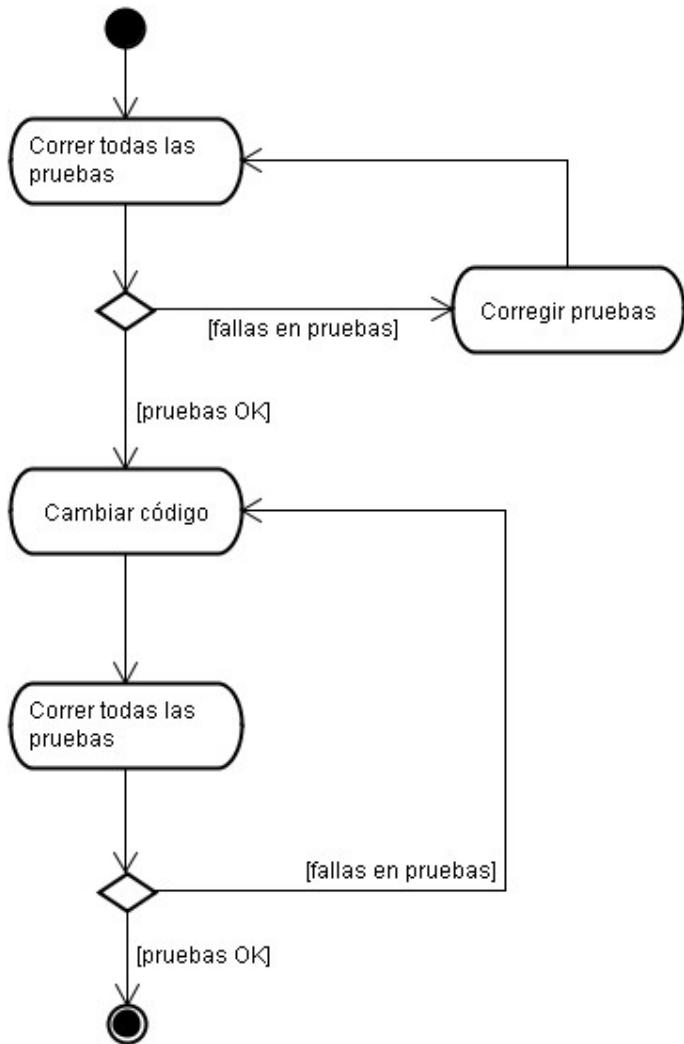


Figura 7.1 Procedimiento usual de una refactorización

Obviamente, para que esto funcione bien, tenemos que tener pruebas suficientes para comprobar el comportamiento observable. Si veníamos desarrollando en base a pruebas unitarias automatizadas, es muy probable que con las pruebas que ya tenemos nos alcance para verificar la corrección de las refactorizaciones.

Una buena práctica en el marco de las refactorizaciones, que si bien no asegura el resultado lo favorece, es hacer pequeñas iteraciones y probar el mantenimiento del comportamiento después de cada una de ellas. Como los entornos de desarrollo hacen algunas tareas en forma automática, a veces no es necesario descomponer en pasos tan pequeños, pero hay muchas situaciones que los entornos de desarrollo no resuelven, y en estos casos la recomendación sigue siendo esa: cambiar de a poco y probar luego de cada cambio, por pequeño que sea.

A qué llamamos observable

Venimos diciendo que una refactorización debe preservar el comportamiento observable. ¿Y qué es observable?

Opdyke [Opdyke 1992] propuso que, para verificar la invariancia de comportamiento, el

conjunto de entradas y el conjunto de salidas debería ser el mismo antes y después de una refactorización. Pero hay situaciones que no están contempladas en este enunciado. Por ejemplo, en sistemas de tiempo real las duraciones de los procesos son parte del comportamiento. En otros casos, puede que el uso de memoria se deba mantener. En sistemas que utilizan reflexión³⁵ o metaprogramación puede que necesitemos considerar al código como parte del comportamiento.

Por lo tanto, la palabra clave en las definiciones anteriores es “observable”. Lo que busca una refactorización es que se preserve lo que un usuario espera del sistema, que es lo que habitualmente llamamos requerimientos. Podríamos entonces definir una refactorización considerando esta premisa:

Definición: refactorización (basada en requerimientos)

Una refactorización es un cambio realizado sobre el código de un sistema de software, con el sólo fin de mejorar la calidad interna con vistas al mantenimiento, de modo tal que, luego de esa transformación, se sigan cumpliendo los requerimientos de la aplicación.

Discusión: requerimientos

La noción de que el software se construye en base a requerimientos de clientes y usuarios es antigua. Sin embargo, ha sido interpelada por los metodólogos más modernos que afirman que el cliente no conoce todo lo que quiere o necesita. No deseamos entrar en esa discusión en un libro de programación, y por eso dejaremos la definición anterior como está.

Simplificando métodos largos

Bueno, pero basta de teoría por un rato. ¿Qué pasa cuando un método es largo? ¿Qué pasa cuando un método está plagado de comentarios que separan sus partes? En el ejercicio del Sudoku tenemos un método así: el *cargar* de la clase *Tablero*.

Se trata, ante todo, de un método largo: la implementación completa tiene decenas de líneas. Por otro lado, el método sigue un mal olor famoso: está dividido por comentarios. Recordemos el método en Smalltalk:

```
cargar: numeros
    "recibe un Array con los números del tablero y crea las celdas, filas,
     columnas y cajas"

    "verificación de tamaño:"
    ... acá va el código que verifica el tamaño del arreglo ...

    "verificación de números entre 0 y 9:"
    ... acá va el código que verifica que los números en el Arreglo sean
     valores entre 0 y 9 ...

    "generación de celdas:"
    ... acá va el código que genera las 81 celdas y las referencia ...
```

³⁵ Reflexión es una técnica por la cual un programa observa y modifica su estructura de alto nivel. Veremos reflexión en el capítulo 9.

```

"generación de filas:"
... acá va el código que genera las 9 filas y referencia las celdas
...

"generación de columnas:"
... acá va el código que genera las 9 columnas y referencia las celdas
...

"generación de cajas:"
... acá va el código que genera las 9 cajas y referencia las celdas
...

```

Siguiendo el procedimiento de la figura 7.1, lo primero que debemos hacer es ejecutar las pruebas para ver que estén funcionando.

Pero ¿qué pruebas? Una posibilidad es ejecutar todas las pruebas de la aplicación. En nuestro caso, siendo todavía poco lo que hemos construido, podríamos hacerlo. En un caso más complejo, habría que tomarse el trabajo de ver qué pruebas cubren el método *cargar*, y ejecutar sólo esas pruebas³⁶.

Las pruebas que ejecutan el método *cargar* son:

- *siArregloTieneMenosDe8elementosCargarDebeLanzarExpcion*
- *siArregloTieneMasDe8elementosCargarDebeLanzarExpcion*
- *siArregloVacioCargarDebeLanzarExpcion*
- *siHayUnNumeroNegativoCargarDebeLanzarExpcion*
- *siHayUnNumeroMayorQue9cargarDebeLanzarExpcion*
- *siCargo3enCelda56debeQuedarceldaConValor*
- *siCargo0enCelda29debeQuedarceldalibre*
- *loQueCargoEnCelda56deboEncontrarloEnFila5*
- *loQueCargoEnCelda56deboEncontrarloEnColumna6*
- *loQueCargoEnCelda56deboEncontrarloEnCaja5*

Las ejecutamos y vemos que pasan. A continuación, y procurando realizar de a un cambio pequeño a la vez, vamos a crear un método por cada fragmento de código del método *cargar*.

Por ejemplo, el fragmento:

```

"verificación de tamaño:"
( numeros size ~= 81 )
    ifTrue: [ TamañoIncorrecto new signal ].

```

Lo podemos extraer, dejando la siguiente línea en *cargar*:

³⁶ Esto no suele ser tan sencillo, pues las herramientas de cobertura no hacen “cobertura inversa”, mostrándonos qué pruebas pasan por una línea de código, sino sólo qué líneas cubre cada prueba. Por lo tanto, un camino es ir probando prueba por prueba, lo cual es más complejo que ejecutar todas las pruebas del sistema. Un camino alternativo sería construir una herramienta que analice la cobertura inversa, como hicieron Carlos Fontela, Alejandra Garrido y Andrés Lange [FontelaASSE 2013]. La tercera, y tal vez la más usada, es usar la intuición, aunque es el camino más riesgoso.

```
self verificarTamanioArreglo: numeros.
```

Y creando un método nuevo:

```
verificarTamanioArreglo: arregloNumeros
    (arregloNumeros size ~= 81 )
        ifTrue: [ TamanioIncorrecto new signal ].
```

A continuación, ejecutamos las pruebas, y vemos que las mismas pasan. El siguiente paso es la extracción de la verificación de que los valores del arreglo sean números entre 0 y 9.

Para ello, escribimos la siguiente invocación en *cargar*:

```
self verificarNumerosEntre0y9enArreglo: numeros.
```

Y definimos otro método nuevo:

```
verificarNumerosEntre0y9enArreglo: arregloNumeros
    arregloNumeros do: [ :num |
        ( (num < 0) | (num > 9) )
            ifTrue: [ ValorInvalido new signal ].
    ].
```

Ejecutamos nuevamente las pruebas y verificamos que las mismas pasan. Lo siguiente sería separar la construcción de las celdas y su vinculación desde el tablero. Para ello, escribimos la siguiente invocación en *cargar*:

```
self generarCeldasVincularTablero: numeros.
```

Y procedemos igual que en los casos anteriores, escribiendo un nuevo método al que pasamos el código de generación de celdas. Por supuesto, a continuación, ejecutamos las pruebas para ver que pasen.

Así podemos seguir (lo hemos hecho, aunque no está transcripto aquí para no aburrir al lector) con la generación de filas, columnas y cajas. Luego de cada cambio, como ya es hábito, ejecutamos las pruebas. El resultado es un método *cargar* mucho más reducido y legible, y varios métodos aparte, cada uno haciendo su pequeña tarea. Omitimos escribir todos los métodos, dejando como ejemplo solamente al método *cargar*:

```
cargar: numeros
    "recibe un Array con los números del tablero y crea las celdas, filas,
    columnas y cajas"
    | ... declaraciones de variables locales ... |
    self verificarTamanioArreglo: numeros.
    self verificarNumerosEntre0y9enArreglo: numeros.
    self generarCeldasVincularTablero: numeros.
    self generarFilasVincularCeldas.
    self generarColumnasVincularCeldas.
    self generarCajasVincularCeldas.
```

Como vemos, resulta mucho más corto y se puede leer en unos pocos segundos.

Esta refactorización que hicimos suele denominarse *Extract Method*³⁷, y se basa en un olor que se suele llamar *Long Method*³⁸. Como se trata de una refactorización de catálogo, está incluida

³⁷ Para las refactorizaciones y los olores hemos mantenido los nombres originales en inglés. La traducción en este caso sería “extraer método”.

³⁸ La traducción al castellano sería “método largo”.

en varios entornos de desarrollo, entre ellos el Pharo que usamos nosotros en Smalltalk.

Como la refactorización es una práctica que existe hace ya mucho tiempo, Pharo, como muchos otros entornos de desarrollo, brinda una ayuda al programador automatizando las refactorizaciones. Eligiendo el trozo de código a extraer y haciendo clic con el botón derecho, aparece un menú contextual, del cual podemos elegir la opción “Source code refactoring”, ante lo cual se abre un nuevo menú, donde elegimos “Extract method”. Luego podemos elegir el nombre del método y sus parámetros, y Pharo hace el resto por nosotros. Por supuesto, conviene comprobar que la extracción haya sido bien hecha, ejecutando las pruebas que tenemos.

La figura 7.2 nos muestra los menús de Pharo.

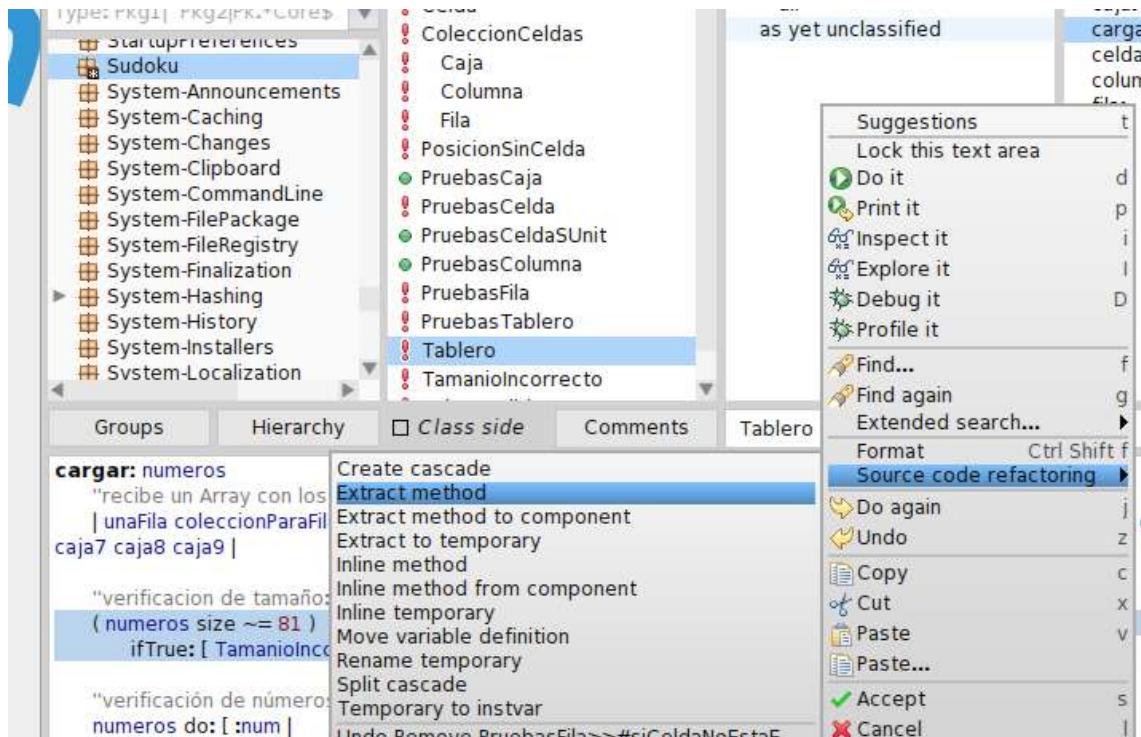


Figura 7.2 Refactorizando en Pharo

Discusión de diseño

Los nuevos métodos generados a partir de la refactorización no eran parte de la interfaz de la clase. Esto significa que no son parte del contrato entre el cliente y la clase Tablero. Por lo tanto, sólo existen como resultado de una refactorización, y no tiene sentido que sean accedidos desde otros objetos.

En lenguajes que permiten cambiar la visibilidad de los métodos, como Java, haríamos estos métodos privados. En Smalltalk no hay manera de hacer esto – aunque existen algunas convenciones que se usan para indicar que un método es privado – así que los dejaremos como están.

Extracción de código repetido

Hagamos otra refactorización un poco más laboriosa, pero conocida. Tan conocida que... ¡ya la hicimos!

En efecto, vamos a hacer la refactorización de extracción de código repetido en el método

extraer de las clases de cuentas en la aplicación bancaria en Java. Es cierto que ya lo hicimos, pero ahora vamos a concentrarnos en seguir el procedimiento.

Veamos. Antes de la refactorización teníamos tres métodos *extraer*. En la clase *Cuenta*:

```
public abstract void extraer(int monto);
```

En la clase *CajaAhorro*:

```
public void extraer (int monto) {
    if (monto <= 0)
        throw new MontoInvalido();
    if (monto > this.getSaldo())
        throw new SaldoInsuficiente();
    this.saldo -= monto;
}
```

En la clase *CuentaCorriente*:

```
public void extraer(int monto) {
    if (monto <= 0)
        throw new MontoInvalido();
    if (monto > this.getSaldo() + this.descubiertoAcordado)
        throw new SaldoInsuficiente();
    this.saldo -= monto;
}
```

Siguiendo el procedimiento de la figura 7.1, lo primero que debemos hacer es ejecutar las pruebas que cubran los tres métodos *extraer* para ver que estén funcionando.

Las pruebas que ejercitan esos métodos son:

- *enCajaAhorroIntentarExtraerMasQueSaldoDebeLanzarExcepcion*
- *enCuentaCorrienteIntentarExtraerMasQueDescubiertodebeLanzarExcepcion*
- *alExtraerSiMontoEsNegativoDebeLanzarExcepcion*
- *alExtraerSiMontoEsCeroDebeLanzarExcepcion*
- *alExtraerSaldoDebeDecrementarseEnMonto*
- *alExtraerCuentaCorrientePuedoGirarEnDescubiert*

Las ejecutamos y vemos que – como era de esperar – pasan.

A continuación, y procurando realizar de a un cambio pequeño a la vez, extraemos código en la clase *CajaAhorro*:

```
public void extraer(int monto) {
    if (monto <= 0)
        throw new MontoInvalido();
    if ( monto > disponible() )
        throw new SaldoInsuficiente();
    this.saldo -= monto;
}

public int disponible () {
    return this.getSaldo();
}
```

Ejecutamos las pruebas nuevamente. Como pasan, extraemos código en la clase *CuentaCorriente*:

```
public void extraer(int monto) {  
    if (monto <= 0)  
        throw new MontoInvalido();  
    if ( monto > disponible() )  
        throw new SaldoInsuficiente();  
    this.saldo -= monto;  
}  
public int disponible () {  
    return (this.getSaldo() + this.descubiertoAcordado);  
}
```

Verificamos una vez más que las pruebas pasen. Como pasan, y dado que el método *extraer* ahora es idéntico en ambas clases hermanas, lo movemos a la clase *Cuenta*. En la clase *Cuenta* queda:

```
public void extraer(int monto) {  
    if (monto <= 0)  
        throw new MontoInvalido();  
    if ( monto > disponible() )  
        throw new SaldoInsuficiente();  
    this.saldo -= monto;  
}
```

Pero si intentamos compilar lo anterior, ¡Eclipse nos dice que el método *disponible* no existe! En efecto, siendo Java un lenguaje de comprobación estática, debemos definir *disponible* para la clase *Cuenta*, aun cuando sea abstracto (ver capítulo de polimorfismo). Lo hacemos así entonces:

```
public abstract int disponible ();
```

Nuevamente ejecutamos las pruebas y vemos que pasan.

Como último paso, cambiamos el indicador de visibilidad del atributo *saldo*, de *protected* a *private*. Volvemos a ejecutar las pruebas y – al pasar – quedamos conformes con la refactorización realizada.

Atención:

Asegúrese de entender por qué hicimos este último cambio de visibilidad. Pista: vea el capítulo 4, cuando nos planteamos declarar a *saldo* como protegido, y finalmente no lo hicimos.

Esta refactorización que hicimos suele denominarse *Extract Method* seguida de *Pull Up Method*³⁹, y se basa en un olor que se suele llamar *Duplicated Code*⁴⁰. Como se trata de refactorizaciones de catálogo, están incluidas en varios entornos de desarrollo, entre ellos el Eclipse que usamos nosotros en Java.

La misma admite algunas variaciones:

- Si el código repetido hubiera estado en dos métodos de la misma clase, habríamos

³⁹ La traducción al castellano sería “extraer hacia arriba un método”.

⁴⁰ La traducción al castellano sería “código duplicado”.

podido extraer el código repetido a un método privado dentro de esa clase.

- Cuando el código repetido está en dos clases hermanas (como pasó en nuestro caso), lo extraemos en un método que pasamos a la clase madre, y que podría ser protegido si no necesitamos invocarlo desde afuera.
- Si el código repetido hubiera estado en clases que no tienen ancestros en común, habríamos podido extraer el código en un método y colocarlo en una clase generada especialmente, a la que las clases anteriores acceda por delegación.

Catálogos

Una vez establecida la refactorización como una práctica deseable, comenzaron a surgir algunos catálogos que muestran algunas refactorizaciones típicas, les dan un nombre y explican cómo realizarlas. Ya Opdyke [Opdyke 1992] presentó un catálogo de 23 refactorizaciones primitivas, a las que agregó tres refactorizaciones compuestas (sucesiones de refactorizaciones) a modo de ejemplo, trabajando con C++. Don Roberts [Roberts 1999] amplió las definiciones de las refactorizaciones primitivas agregándoles postcondiciones, y trabajando con un lenguaje diferente, Smalltalk.

Tal vez el catálogo más difundido en ambientes organizacionales sea el libro de Martin Fowler [Fowler 1999], que incluye 72 refactorizaciones típicas en Java, que el mismo autor complementa en su sitio web (<http://www.refactoring.com/>), agregando algunas más.

Siguiendo la línea de Fowler, Joshua Kerievsky [Kerievsky 2005], presenta un catálogo de refactorizaciones hacia patrones de diseño en Java, complementando una idea prevista en trabajos previos, de que el uso de patrones está mejor justificado cuando surge como necesidad durante una refactorización.

Lippert y Roock [LippertRoock 2006] han construido un catálogo de grandes refactorizaciones, aunque con un grado de detalle menor a los que se pueden encontrar en las publicaciones de Opdyke y Fowler.

Lo cierto es que los catálogos de refactorizaciones suelen venir acompañados de catálogos de olores, y asocian unos con otros. Fue Kent Beck [Fowler 1999] el primero que sugirió basarse en olores para detectar lugares de deseables refactorizaciones, y define a los olores como aquellas “estructuras en el código que sugieren la posibilidad de una refactorización”. Los catálogos de olores se han popularizado a partir del libro de Fowler [Fowler 1999]. Así como hay olores de código, el libro [LippertRoock 2006] introdujo los olores de arquitectura, que requieren refactorizaciones mayores.

Como dijimos más arriba, las refactorizaciones que realizamos más arriba son refactorizaciones de catálogo. Solamente que no en todos los catálogos se llaman igual. Por ejemplo, lo que para Fowler es *Pull Up Method*, para Opdyke es un caso particular de *Migrating Common Code to the Abstract Superclass*⁴¹.

Atención: los nombres importan

En programación y desarrollo de software, los nombres suelen ser importantes. Por eso es que hay ciertos algoritmos que se conocen por su nombre, y lo mismo ocurre con patrones de diseño, refactorizaciones de catálogo y olores. La razón para todo ello es que el uso consistente de nombres es fundamental para la comunicación entre profesionales usando una misma jerga.

⁴¹ La traducción al castellano sería “migrar código en común a una clase ancestro abstracta”.

Dicho sea de paso, lo mismo pasa en cualquier oficio o profesión.

TDD completo

Refactorización como parte de TDD

En el capítulo 3 introdujimos la práctica de escribir las pruebas en código y antes del propio código productivo. Dijimos en ese momento que esa forma de trabajar era parte de una práctica mayor, llamada Test-Driven Development (TDD).

La práctica de TDD, que ahora estamos en condiciones de entender mejor, incluye:

- Test-First
- Automatización
- Refactorización

Es decir, por cada ciclo de pruebas y código se hace una refactorización que vaya manteniendo la buena calidad del código generado.

El ciclo completo de TDD, incluyendo la refactorización, se muestra en la figura 7.3.

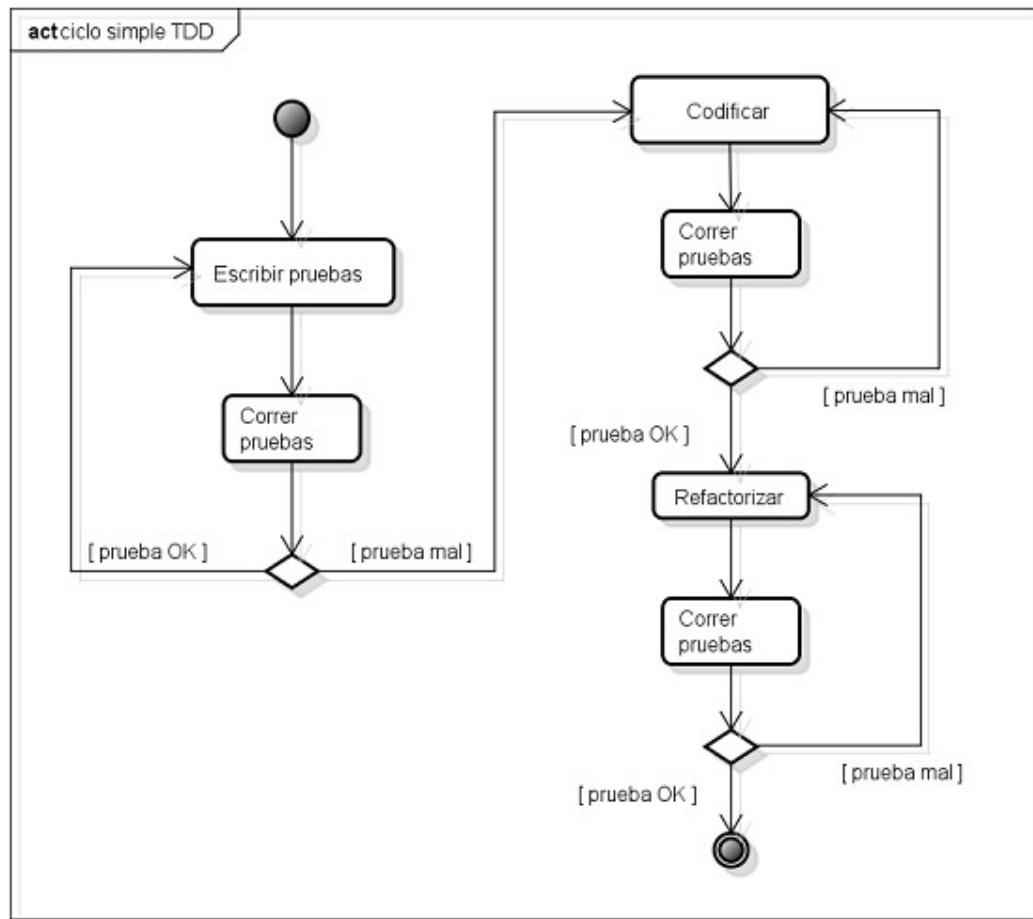


Figura 7.3 Ciclo completo de TDD

TDD y refactorización: un matrimonio bien avenido

Las prácticas de refactorización y TDD aparecen relacionadas en dos sentidos:

- El ciclo de TDD incluye una refactorización luego de hacer que las pruebas se ejecuten exitosamente. Esto logra un mejor diseño que el probablemente ingenuo que pudo haber surgido de una implementación cuyo único fin fuera que las pruebas corrieran sin problemas.
- Una refactorización segura exige la existencia de pruebas automáticas escritas con anterioridad, que permitan verificar que el comportamiento externo del código refactorizado sigue siendo el mismo que antes de comenzar el proceso.

La relación entre ambas prácticas es, por lo dicho, de realimentación positiva. Así como la refactorización precisa de pruebas como red de contención, TDD se apoya en la refactorización para eludir diseños triviales.

Así es como a veces ambas prácticas se citan en conjunto, y se confunden sus ventajas. Por ejemplo, cuando se menciona que TDD ayuda a reducir la entropía conforme pasa el tiempo, en realidad debería adjudicarse este mérito a las refactorizaciones que acompañan a los ciclos de TDD.

Sin embargo, la refactorización no necesita forzosamente del cumplimiento estricto del protocolo de TDD: sólo precisa que haya pruebas – en lo posible automatizadas – y que éstas hayan sido escritas antes de refactorizar. Por lo tanto, si en el desarrollo de una aplicación no se ha empleado TDD, al menos hay que escribir pruebas antes de la refactorización.

Refactorización y desarrollo incremental

La práctica de refactorización está muy asociada a Extreme Programming (XP) y a otros métodos de desarrollo ágil. En ellos, se hace especial hincapié en el carácter evolutivo del diseño. En especial, Kent Beck [Beck 2000] ha hecho énfasis en evitar grandes diseños realizados antes de comenzar a codificar.

Precisamente, la práctica de refactorización ayuda en este sentido del diseño evolutivo e incremental. Veamos.

Uno de los objetivos principales de un buen diseño es prever el cambio, y diseñar con los cambios potenciales en mente. Sin embargo, es imposible prever todo lo que se va a necesitar en el futuro y los cambios que vendrán. Por lo tanto, no podremos prever todo cambio en el diseño inicial de una aplicación. La práctica de refactorización permanente nos ayuda a que el diseño vaya evolucionando conforme los cambios van surgiendo.

Por otro lado, aun cuando los cambios fueran previsibles, es costoso introducir desde el primer momento toda la flexibilidad necesaria para afrontarlos. Además, aun los cambios más previsibles pueden no ocurrir, y ese caso habremos introducido una flexibilidad aparentemente útil que luego no resultó necesaria, y que mientras tanto encareció el desarrollo y el mantenimiento.

También los patrones de diseño⁴² han surgido para facilitar el cambio. Por eso, un enfoque incremental de introducción de patrones en forma evolutiva sería realizar las refactorizaciones hacia patrones antes de un cambio. De esa manera, se pospone el uso de un patrón hasta que

⁴² Un patrón de diseño es una solución exitosa para un problema frecuente de diseño de software. Veremos algo de patrones de diseño en el capítulo 10.

sea realmente necesario.

En definitiva, las buenas prácticas recomendarían esperar que el cambio sea necesario, en ese caso refactorizar para que el diseño se adapte mejor al cambio, y luego recién realizar el cambio en cuestión. De esta manera, el diseño surgirá en forma realmente evolutiva.

Otra manera de enunciarlo es que, para agregar nueva funcionalidad, primero hay que cambiar el código para facilitar el cambio funcional, sin cambiar el comportamiento; y recién luego cambiar el comportamiento.

Situaciones de refactorizaciones más complejas

Refactorizaciones que rompen las pruebas

Como dijimos, la práctica de refactorización descansa fuertemente en la existencia de pruebas unitarias automatizadas, que funcionan como red de seguridad que garantiza que el comportamiento de la aplicación no varía luego de una refactorización.

Sin embargo, esto puede ser ingenuo en algunas situaciones. En efecto, hay ocasiones en que las pruebas dejan de funcionar al realizar las refactorizaciones, con lo cual se pierde la sincronización entre código y pruebas, y la cualidad de red de seguridad de estas últimas. Esto resulta más evidente cuanto menor sea la granularidad de las pruebas y cuanto mayor sea el alcance de la refactorización.

El problema, entonces, surge porque hay situaciones en las cuales las pruebas dejan de funcionar, aun cuando se esté preservando el comportamiento observable, y por lo tanto ya no hacen las veces de una red de contención de la refactorización. Y sin red de contención, refactorizar se torna riesgoso y azaroso.

Por ejemplo, si cambiamos el nombre de un método, porque nos parece que el nombre previo no era el adecuado, los requerimientos del usuario se mantienen inalterados, y por lo tanto se trataría de una refactorización, aun cuando la prueba unitaria que verificara ese método dejase de funcionar porque invoca un método que no existe más.

Lo mismo va a ocurrir ante cualquier cambio de protocolo, como por ejemplo nombres de clases, cantidad de parámetros de métodos y añadido o eliminación de métodos o clases.

El ejercicio del final del capítulo muestra un ejemplo de este problema, aunque en un caso relativamente simple. Para casos más complejos, se aconseja ver mi tesis de maestría [FontelaTesis 2013].

El problema es que no se ha analizado con tanta profundidad el impacto de las refactorizaciones sobre las pruebas automáticas, que en ocasiones resultan afectadas por las propias refactorizaciones, causando que la funcionalidad no pueda ser verificada a posteriori. Uno de los trabajos más conocidos que plantean el problema es el de Uwe Pipka [Pipka 2002].

Se han propuesto algunas soluciones, entre las cuales destacan:

- *Test-First Refactoring*. Es el método que propone Pipka y que consiste en reescribir las pruebas para adaptarlas a las nuevas interfaces, comprobar que fallan y recién luego refactorizar. Notemos que no es un enfoque ciento por ciento seguro, pues ¿cómo podemos garantizar que la nueva versión de la prueba, obviamente diferente de la anterior, comprueba el mismo comportamiento que antes?
- *Refactorización asegurada por los clientes*. Fue propuesto muy informalmente por Lippert y Roock [LippertRoock 2006] y consiste en comprobar las pruebas de los

clientes. Si nuestra refactorización afecta a un objeto que debe preservar su comportamiento, entonces los objetos que a su vez son clientes de éste no deberían cambiar su comportamiento ni su protocolo. Por lo tanto, las pruebas de los objetos clientes⁴³ deberían seguir pasando sin problema.

- *Refactorización asegurada por pruebas de comportamiento.* Es el método que proponen Mugridge y Cunningham [Mugridge 2005], y que consiste en usar pruebas de mayor granularidad, probablemente las pruebas de comportamiento (BDD⁴⁴) o de aceptación (ATDD⁴⁵).
- *Refactorización asegurada por niveles de pruebas.* Propuesto por mí mismo en mi tesis de maestría [FontelaTesis 2013], es una técnica que permite alcanzar un grado de seguridad más alto en cuanto a la corrección, utilizando redes de seguridad en forma escalonada mediante pruebas cada vez más abarcativas cuando el nivel anterior no alcance, comenzando por pruebas unitarias, pasando luego a las de clientes y terminando con las de aceptación, verificando en cada caso la cobertura del código a refactorizar.

Grandes refactorizaciones

Existen ocasiones en las que los cambios de requerimientos o la necesidad de nuevas funcionalidades nos obliga a realizar previamente grandes refactorizaciones.

Si bien resulta difícil definir qué es “grande”, podemos seguir el criterio de [LippertRoock 2006], que clasifica de grandes refactorizaciones a aquellas que cumplen las tres condiciones que siguen:

- Se trata de iniciativas de larga duración (al menos mayor a un día de trabajo).
- Involucran a todo el equipo de desarrollo.
- No pueden ser descompuestas en refactorizaciones básicas seguras y de catálogo.

Algunos de los inconvenientes de estas refactorizaciones, que hacen que se las trate como casos aparte son:

- Las personas que deben trabajar en ellas tienden a perder la traza de la refactorización y se les hace difícil estimar el grado de avance.
- Como realizarlas lleva mucho tiempo, se las suele interrumpir, debiendo garantizar que el sistema quede en un estado válido y de preservación del comportamiento, aunque en estos estados intermedios, la calidad pueda ser peor que cuando se inició.
- Suelen ser difíciles prever todos los cambios intermedios, por lo que requieren desarrolladores experimentados y con mucha creatividad para resolver lo inesperado.

Por todas estas razones es que muchos desalientan la realización de grandes refactorizaciones, recomendando convertirlas en refactorizaciones compuestas de refactorizaciones simples. Sin

⁴³ Notemos que ya no estamos hablando de pruebas unitarias, sino de pruebas que cubren nuestro objeto desde objetos clientes.

⁴⁴ BDD es el acrónimo de Behaviour Driven Development, una técnica que veremos sucintamente en el capítulo 12.

⁴⁵ ATDD es el acrónimo de Acceptance Tests Driven Development, una técnica que veremos sucintamente en el capítulo 12.

embargo, esto no siempre es posible.

Otra manera de evitarlas es adoptar la práctica de realizar refactorizaciones pequeñas en forma permanente durante el desarrollo y luego de cada pequeño cambio, intentando prever modificaciones posteriores. Igual que antes, esto puede funcionar en la mayor parte de los casos, pero no en todos: a veces las grandes refactorizaciones surgen por grandes cambios en una aplicación, que aparecen sin aviso previo.

Un caso especial de refactorización de gran envergadura se da cuando realizamos cambios en repositorios persistentes, sobre todo bases de datos relacionales en sistemas orientados a objetos. En efecto, cuando cambiamos una base de datos, esto suele traer cambios importantes en los objetos que mapean a tablas de la base de datos.

Otras refactorizaciones que suelen convertirse en grandes refactorizaciones son las que afectan relaciones complejas entre objetos con muchos clientes, que a su vez cambian porque cambiaron sus colaboradores. También se suele llegar a grandes refactorizaciones cuando involucramos grupos de clases, tales como paquetes, subsistemas y capas de una aplicación.

No ha habido tantos catálogos de grandes refactorizaciones como sí los ha habido de las elementales. El libro [Lippert 2006] es una notable excepción. Lo cierto es que este tipo de refactorización tiene muchos puntos en común con la reingeniería de software⁴⁶.

Como práctica, es aconsejable comenzar por realizar refactorizaciones pequeñas en varios lugares antes de comenzar la de mayor envergadura, de modo de facilitar el trabajo en una segunda fase.

Refactorización de código legacy

*Legacy code*⁴⁷ es una expresión inglesa muy utilizada en la Ingeniería de Software, aunque bastante difícil de definir. Habitualmente se refiere a código antiguo, aunque ésta no es la característica que más nos interesa destacar.

La mejor definición de código legacy [BrodieStonebraker 1995] es que se trata de un sistema que “se resiste significativamente a la evolución y a la modificación para cumplir con nuevos y constantes cambios de requerimientos”. Feathers [Feathers 2005] define código legacy a aquél que se mantiene porque funciona, sin saber por qué funciona, y su principal distinción es no tener pruebas automatizadas.

En efecto, casi todos los procesos de desarrollo asumen que se desarrolla software desde cero, cuando en realidad la mayor parte del esfuerzo de desarrollo, globalmente hablando, se emplea en el mantenimiento de aplicaciones, algunas de ellas muy antiguas. Para hacer más sencillo el mantenimiento una buena idea es... es... ¡sí!: refactorizar antes.

Refactorización de pruebas automáticas

Si trabajamos con pruebas automatizadas, las pruebas también son código, y por lo tanto son susceptibles de refactorización. Si además suscribimos el argumento de que el código de las pruebas es parte del código del sistema, y que por lo tanto debe cumplir los mismos requisitos de calidad que el resto del código, la refactorización de pruebas pasa a ser importante, ya que

⁴⁶ Reingeniería es un término poco explicado, pero que alude a grandes cambios en la arquitectura del software. En este caso, estamos por definición en una refactorización, debido a que se busca preservar el comportamiento.

⁴⁷ La traducción textual sería “código legado o heredado”.

ayudaría a controlar la entropía del código de las pruebas.

Así es como hay autores [Meszaros 2007, Deursen 2001] que han tratado este tema y han descubierto algunos olores de pruebas, que suelen ser distintos de los olores del código común. Las refactorizaciones típicas de las pruebas también son distintas de las de código funcional, aunque hay pocos trabajos que las analicen y cataloguen.

Lo interesante de la refactorización de pruebas es que lleva a situaciones en las cuales no tendremos pruebas unitarias que nos sirvan para verificar la corrección de una refactorización. Esto es, no suele haber pruebas que garanticen la invariancia de una prueba después de un cambio: no hay pruebas de pruebas.

El camino más factible es considerar al código funcional como herramienta de verificación de las pruebas. Esto es lo que haremos sobre el final del próximo ejercicio.

Ejercicio adicional

El lector más perspicaz habrá notado lo parecidas que son las clases *Fila*, *Columna* y *Caja* en la aplicación de Sudoku. Cualquiera diría que sí, que son parecidas, y que precisamente por eso las hicimos derivar de una clase en común: *ColeccionCeldas*. Ahora bien, cuando se usa herencia de este modo, se supone que las clases hijas, teniendo algún comportamiento común como en este caso, tienen también diferencias entre ellas.

Sin embargo, si vemos lo que hemos implementado hasta ahora, las clases *Fila*, *Columna* y *Caja* son idénticas. En efecto, los métodos *initialize*, *agregarCeldas*, *contiene* y *contieneCeldas*, son exactamente iguales y por ello podríamos trasladarlos a la clase madre. Hagamos esa refactorización, que en el catálogo de [Fowler 1999] se denomina *Pull Up Method*.

Como siempre, debemos empezar por descubrir cuáles pruebas cubren el código a cambiar. En este caso, se trata de 11 métodos para cada clase, reunidos en las clases de prueba *PruebasFila*, *PruebasColumna* y *PruebasCaja*. Por como fuimos escribiéndolos, algunos arrojan sus resultados por consola y otros usan SUnit.

Ejecutamos todos estos métodos, y vemos que las pruebas pasan. Ahora vamos a ir introduciendo los cambios de a uno.

Lo primero que hacemos es pasar el método *initialize* a la clase *ColeccionCeldas* y eliminarlo de las tres clases hijas. Usamos nuevamente las facilidades de Pharo, que nos provee una opción de menú llamada “Push up”. Luego de hacer el cambio, ejecutamos las pruebas.

Aquí nos encontramos con un error. ¿Por qué? Ocurre que el método *initialize* de las tres clases hacía uso de un atributo *celdas*, que en la clase *ColeccionCeldas* no existe. Nos hemos apresurado, creyendo que Pharo iba a resolver eso también.

Pero no hay problema: declaramos el atributo *celdas* en *ColeccionCeldas* y lo eliminamos en las tres clases hijas. Volvemos a ejecutar las pruebas y vemos que pasan.

Sigamos con el método *agregarCeldas*. Procedemos igual, con la ayuda de Pharo, ejecutamos las pruebas y vemos nuevamente que todas pasan.

Seguimos luego con los métodos *contiene* y *contieneCelda*, con el mismo resultado.

Bien, lo hemos logrado. Sin embargo, llegamos a un resultado extraño: las tres clases hijas han quedado vacías. En un caso así, ¿qué debemos hacer? ¿para qué sirven las clases vacías?

Algunos lectores alegarán que las clases *Fila*, *Columna* y *Caja* son representaciones del modelo

de dominio, y por lo tanto son abstracciones útiles. Sin embargo, si el comportamiento de las mismas resulta idéntico – y por el momento así es – definir tres clases triplica el código productivo, el código de pruebas y obliga mantener una sobrecarga importante ante cualquier cambio futuro.

Eso nos hace pensar que debemos eliminar las tres clases y punto, quedándonos solamente con la clase *ColeccionCeldas*.

Importante

Notemos que no estamos diciendo que no deba haber objetos fila, columna y caja, referenciados desde un tablero, sino que los 27 objetos (9 filas, 9 columnas y 9 cajas), que seguiremos agrupando en tres colecciones de la clase *Tablero* que seguiremos referenciando desde los atributos *filas*, *columnas* y *cajas*, son instancias de un mismo tipo. De nuevo, aparece la fundamental distinción entre los objetos – el concepto central de la POO – y las clases – un concepto propio de algunas implementaciones del paradigma.

Procedamos entonces a refactorizar. En principio, las pruebas que deberíamos usar son las mismas de la refactorización anterior. Sin embargo, esas pruebas usan precisamente las clases que queremos eliminar. ¿Qué podemos hacer? Nos sentimos tentados de eliminar esas pruebas y cambiarlas por unas que usen la clase *ColeccionCeldas*. Con ello, sólo quedarían un tercio de las pruebas, haríamos los cambios, las ejecutamos para probar que sigan funcionando bien, y listo.

Sin embargo, un cambio así, aunque simple y aparentemente inocente, resulta riesgoso. Si cambiamos las pruebas que garantizan el comportamiento del sistema, ¿cómo podríamos asegurar que ese comportamiento se mantiene?

Una posibilidad es usar las pruebas de los clientes, como vimos antes en la discusión sobre cambios que rompen las pruebas. En este caso, tenemos una clase *Tablero*, que es cliente de nuestras tres clases a eliminar. Bastaría con probar con las pruebas de *Tablero*, ya que es la única clase cliente de los objetos filas, columnas y cajas, y el buen funcionamiento de *Tablero* debería bastar para dar por buenos los cambios en las clases de filas, columnas y cajas.

Una objeción posible es que la clase *Tablero* no está completa todavía, ya que sólo tenemos los métodos para cargarlo y verificar celdas, filas, columnas y cajas. Esto es así, sin duda. Por lo tanto, podríamos dejar todo como está – con tres clases vacías – y esperar a terminar la aplicación, ver en ese momento si siguen vacías, y recién allí eliminarlas. Una opción más audaz es limpiar las tres clases, que por el momento son inútiles, y reescribirlas sólo en caso de ser necesario más adelante.

Nos quedamos con la última opción, por dos motivos: queda un diseño más simple; y tampoco es de una complejidad superlativa agregar tres clases vacías más adelante. Por lo tanto, procederemos como dijimos: eliminamos las tres clases, usando las pruebas de la clase *Tablero* como red de seguridad.

Lo primero que hacemos es ejecutar las pruebas de la clase *PruebasTablero*, que vemos que funcionan bien.

Ahora eliminamos la clase *Fila*. Sabemos que las pruebas de la clase *PruebasFila* van a fallar, así que no nos molestamos en ejecutarlas. Sí ejecutamos las pruebas de la clase *PruebasTablero*, pero allí nos encontramos con que el método cargar intenta crear una instancia de *Fila* en la línea:

```
unaFila := Fila new.
```

Debemos cambiarla antes de seguir. La reemplazamos por:

```
unaFila := ColeccionCeldas new.
```

Ahora sí ejecutamos las pruebas y vemos que pasan. Hacemos lo mismo con las clases *Columna* y *Caja*, eliminándolas, y cambiando la creación en el método *cargar* de *Tablero*. Al ver que todas las pruebas pasan, terminamos.

¿Terminamos? No vayamos tan rápido...

Es cierto que hemos cambiado el código en el sentido que queríamos, y que ahora las pruebas de *Tablero* pasan. Pero hemos dejado de lado las pruebas de las clases *PruebasFila*, *PruebasColumna* y *PruebasCaja* que, por supuesto, no funcionan.

La tentación de parar aquí es fuerte. Sin embargo, si consideramos a las pruebas como parte del sistema – y lo son – no podemos deshacernos de las pruebas unitarias sin más. De hecho, notemos que las pruebas de la clase *PruebasTablero* no están funcionando como unitarias, ya que nos están sirviendo para probar más de un objeto a la vez. Por lo tanto, no tratemos de escabullirnos, y arreglemos las pruebas unitarias.

Ahora bien, al cambiar una clase, las pruebas unitarias nos sirven como red de contención. ¿Cuál será esa red en el caso de cambiar las propias pruebas? Bueno, sabemos que el sistema está bien – lo sabemos a través de la ejecución de las pruebas de *PruebasTablero* – así que si las pruebas modificadas funcionan bien, podemos suponer que están bien. En definitiva, la red de contención es el propio sistema.

Habiendo quedado una sola clase, *ColeccionCeldas*, podemos quedarnos con una sola clase de pruebas, que podemos llamar *PruebasColeccionCeldas*. Como las tres clases de prueba deberían probar lo mismo, simplemente renombramos *PruebasCaja* y eliminamos las otras dos clases de prueba. Revisamos, cambiando las apariciones de la clase *Caja* por *ColeccionCeldas* y, una vez que lo hicimos, ejecutamos las pruebas.

Ahora sí, al funcionar bien, hemos terminado: hemos eliminado tres clases productivas y tres clases de prueba, agregando una clase de pruebas nueva. En el primer paso, usamos pruebas de clientes para comprobar que el comportamiento no cambiase (*PruebasTablero*). En el segundo, el propio sistema nos aseguró la corrección de las nuevas pruebas unitarias (*PruebasColeccionCeldas*).

Ejercicios propuestos

Contenido todavía no disponible en esta versión del libro.

Recapitulación

En este capítulo, hemos visto refactorización como práctica metodológica, TDD en forma completa, y hemos analizado escenarios particulares de refactorización.

En el próximo ahondaremos sobre UML, la herramienta notacional que ya venimos usando, y que nos sirve para modelar nuestros programas orientados a objetos.

8. Hablemos de herramientas: UML y su uso

Contexto

UML es el acrónimo en inglés para “lenguaje unificado de modelado”. Surgió en 1995, por iniciativa de Grady Booch, James Rumbaugh e Ivar Jacobson, tres conocidos ingenieros de software que ya habían avanzado con sus propias notaciones de modelado. Precisamente, UML se define como “unificado”, porque surgió como síntesis de los mejores elementos de las notaciones previas. Hoy es un estándar de facto que se usa para modelar sistemas orientados a objetos.

Hasta este momento venimos usando UML cada vez que queremos mostrar ciertas cuestiones de nuestros ejemplos, pero no hemos analizado por qué lo hacemos, para qué, ni qué otras cosas podríamos mostrar.

Ante todo, tengamos en cuenta que este es un libro de programación, y por lo tanto el uso que hagamos de UML va a ser el que sea más útil para diseñar y programar. Hay otros usos de UML para otras disciplinas del desarrollo de software. Para quien quiera un buen resumen de UML en su conjunto recomiendo los libros de Fowler [Fowler 2004] y uno mío propio [Fontela 2011].

No todos los profesionales de desarrollo de software están de acuerdo con usar UML, o incluso otra notación basada en diagramas. Puede verse una pequeña discusión de esto en el apéndice C.

UML como herramienta de comunicación

Discutiendo el diseño

UML es – ante todo – una herramienta de comunicación. Sirve para comunicar en forma gráfica lo que contiene un sistema o algunas de sus partes y cómo se comporta.

En efecto, el uso más habitual de UML es plantear un diseño para ser discutido por un equipo de desarrollo antes de construir una parte del sistema. Esto tiene que ver con que hay muchas personas que se comunican mejor usando gráficos y otras imágenes que mediante código.

Eso hicimos nosotros en el libro en repetidas ocasiones, cuando antes de mostrar una solución en código usamos un diagrama de clases o uno de secuencia para ilustrar mejor de qué estábamos hablando. Eso mismo puede hacer un equipo de desarrollo, habitualmente mediante bosquejos a mano alzada sobre una pizarra o un papel. De ese modo, un desarrollador puede ir mostrándole a sus compañeros una idea de solución que a él se le ocurre. Y los compañeros podrán opinar escribiendo sobre el mismo diagrama o simplemente haciendo observaciones. La pizarra blanca y los marcadores de varios colores son los auxiliares fundamentales de quien usa UML con esta finalidad.

Como recién dijimos, este es el uso más habitual de UML, y en este caso suele hacerse un uso laxo de la notación, sin tanto detalle ni cuidado por la precisión. Cuando sólo interesa mostrar algunos aspectos, no siempre es importante mostrar visibilidad o todas las dependencias entre clases u objetos. Tampoco suele mostrar todos los elementos, sino sólo aquello que se quiere

discutir: tal vez un escenario de objetos o unas pocas clases. Incluso abundan los usos no ortodoxos de UML, mezclando objetos y clases o componentes y paquetes en un mismo diagrama.

Otra cosa que ocurre en estos casos es que los dibujos que se realizan suelen ser efímeros: una vez que se plasma el diseño en código, los diagramas pierden sentido, y suelen borrarse o descartarse.

Dicho todo lo anterior, hay personas que se sienten cómodas trabajando directamente sobre el código, y entienden que la mejor manera de comunicar es mediante el código en sí mismo. Si bien mi impresión personal es que se trata de una minoría, para esa minoría usar UML como herramienta de discusión no tiene sentido. Ver el apéndice C para más información.

Documentando lo realizado

Otro uso posible es documentar el sistema para su uso posterior, sea para mantenimiento o para el uso de un framework o biblioteca. En estos casos, se suelen usar herramientas que dejen diagramas más prolijos, pero tampoco se usa un nivel de detalle demasiado completo.

Por ejemplo, si lo que se quiere es mostrar el uso de un framework o biblioteca, no tiene sentido mostrar atributos y métodos privados, ni tampoco las interacciones entre objetos que no tengan necesidad de ser conocidas por los clientes. Probablemente, si la biblioteca es muy vasta, tal vez la idea sea mostrar solamente las clases y métodos más relevantes.

Si, en cambio, lo que se desea es documentar un sistema para mantenimiento posterior, el foco estará puesto en los detalles de implementación, sobre todo en los escenarios más complejos de interacciones.

Así, el uso de UML en estos casos es variado, pero suele ser más cuidado que cuando sólo se usa para discutir un diseño. Y también la perdurabilidad de los diagramas es mayor. No obstante, es importante no sobrecargar los diagramas con elementos innecesarios, recordando siempre que estamos intentando facilitar la comunicación entre humanos.

¿UML como lenguaje de programación?

El último caso es menos común, aunque suele utilizarse a veces.

Se trata de emplear a UML como una herramienta de desarrollo en sí misma. La más extrema de estas situaciones se da cuando se hace uso de la metodología conocida como MDD (Model Driven Development o “desarrollo guiado por modelos”). En este caso, se parte de modelos que surgen del análisis y, mediante una serie de pasos cuidadosamente controlados, se llega al código fuente del sistema, en forma automática.

Se ha criticado mucho esta forma de trabajo, después de la expectativa que se generó en la década de 1990 con las herramientas CASE⁴⁸ y su posterior fracaso, que ha convertido a la sigla CASE en poco menos que una mala palabra. Se ha dicho que el desarrollo de software es una actividad muy creativa, y que el uso de herramientas automáticas limita esa creatividad. También se ha puesto el énfasis en la mala calidad del código generado, que dificulta el mantenimiento en el caso de abandonar la herramienta. Y, además, se ha destacado la dificultad de atacar la complejidad de ciertas cuestiones de los dominios específicos. No obstante, sigue siendo válido explorar, al menos desde la investigación, la posibilidad de mecanizar todo lo que

⁴⁸ Acrónimo inglés para “Computer Aided Software Engineering” o “Ingeniería de Software Asistida por Computadora”.

se pueda del desarrollo de software, aprovechando así las ventajas de las computadoras para realizar tareas automáticamente.

Lo importante a destacar es que en este enfoque sí necesitamos colocar el máximo detalle posible en nuestros diagramas. Al fin y al cabo, si de los diagramas surge automáticamente el código, éstos deben tener el mismo nivel de detalle que el código. O, como dicen algunos autores, en estos casos “los diagramas son el código”, con lo que estamos usando a UML como un lenguaje de programación. Tampoco nos queda más remedio que utilizar herramientas para hacer los diagramas. Y, al igual que lo que ocurría antes, la claridad de los diagramas es fundamental para facilitar su mantenimiento.

UML: un lenguaje de modelos

Modelos: representación simple de algo complejo

La palabra **modelo** tiene varias acepciones en castellano. Nosotros usaremos la siguiente:

Definición: modelo

Un modelo es una descripción analógica para ayudar a visualizar algo que no se puede observar directamente, que se realiza con un propósito determinado y se destina a un público específico.

A modo de ejemplos, un mapa de transportes de una ciudad, es un modelo de la ciudad en cuestión; un plano de instalaciones sanitarias de un edificio, es un modelo de ese edificio; un dibujo de una nave intergaláctica, es un modelo de nave intergaláctica.

En los ejemplos anteriores, nos referimos a representaciones de cosas que no podemos observar, pero de las que nos damos una idea a partir del modelo.

Veamos las partes de nuestra definición:

- **Descripción analógica:** el modelo no es aquello que se quiere observar, sino una representación simplificada. Por esta razón, el mapa no es la ciudad, el plano sanitario no es la propia instalación sanitaria y el dibujo de la nave intergaláctica no es la propia nave.
- **De algo que no puede ser observado directamente:** el sistema de transportes de la ciudad no se puede ver, porque es un concepto abstracto que requiere de estudio y observación; las instalaciones sanitarias del edificio no se pueden ver a menos que rompamos las paredes y pisos del mismo; la nave intergaláctica no puede verse... porque aún no se construyó ninguna.
- **Se realiza con un propósito determinado:** el propósito o perspectiva es lo que determina para qué realizamos el modelo. Así, el mapa de transportes sirve para saber cómo llegar de un punto a otro de la ciudad usando el sistema de transportes; en el plano de instalaciones sanitarias se indica por dónde pasan las cañerías del edificio; el dibujo de la nave intergaláctica, para comunicar a otras personas cómo sería una posible nave.
- **Destinado a un determinado público:** es decir, existe un público potencial que va a usar el modelo en cuestión. Por ejemplo, el mapa de transportes se realiza para un ciudadano común que necesita moverse por la ciudad o para un planificador de transportes; el plano de instalaciones sanitarias le sirve a un plomero que desee hacer una refacción de un baño, entre otros; el dibujo de la nave intergaláctica puede tener un público al que le interese la ciencia ficción.

Notemos que las cosas que modelamos no tienen siquiera que existir. Tal vez lo primero que nos venga a la mente sea la imagen de la nave. Pero, incluso, el mapa de transportes o el plano de instalaciones sanitarias pueden referirse a situaciones hipotéticas (porque la ciudad está evaluando distintas alternativas de planeamiento del transporte) o futuras (porque el edificio todavía no se construyó).

La finalidad última de un modelo es la comunicación de algo: un proyecto, un concepto, la descripción física de algún elemento, etc.

Precisamente, por esta necesidad de comunicar y porque, además, se destina a un determinado público, con cierto propósito, un modelo se centra sólo en lo que se quiere comunicar y oculta los datos innecesarios.

Modelos de software

El software necesita modelos, en primer lugar, por las mismas razones que cualquier otra construcción humana: para comunicar de manera sencilla una idea abstracta, existente o no, o para describir un producto concreto existente.

En efecto, el modelo más detallado de un producto de software es el código fuente. Pero es como decir que el mejor modelo de un edificio es el edificio mismo; esto no nos sirve para concebirlo antes de la construcción ni para entender sus aspectos más ocultos con vistas al mantenimiento luego de terminado.

Sin embargo, en el software el modelado es aún más importante que en las otras ingenierías. Esto tiene varias razones de ser:

- El software es invisible e intangible: sólo se ve su comportamiento, sus efectos en el medio.
- El software es mucho más modificable que otros productos realizados por el hombre: esta maleabilidad es percibida por los ajenos a la industria, lo que provoca que haya un incentivo mucho más fuerte para modificarlo.
- El software se desarrolla por proyectos, no en forma repetitiva como los productos de la industria manufacturera. Esto hace que cada vez que construyamos un producto de software estemos enfrentándonos a un problema nuevo de diseño.
- El software es substancialmente complejo, con cientos o miles de partes interactuando, diferentes entre sí, y que pueden ir cambiando de estados a lo largo de su vida: esto hace que analizar un producto de software requiera mecanismos de abstracción y de un lenguaje para representarlo.

Diagramas de UML

Todos los diagramas

En UML 2.5 (la última versión a la fecha de escribir estas líneas) existen modelos estructurales y otros de comportamiento. En total son 13 tipos de diagramas.

Los modelos estáticos o estructurales sirven para modelar el conjunto de objetos, clases, relaciones y sus agrupaciones, presentes en un sistema. Por ejemplo, una empresa tiene clientes, proveedores, empleados; los empleados, que tienen un legajo y un sueldo, se asignan a proyectos; los proyectos pueden ser internos o externos; los segundos tienen clientes, mientras que los primeros, no; los proyectos externos tienen costo y precio de venta, mientras que los

internos solamente costo, etc.

Pero, además, existen cuestiones dinámicas o de comportamiento que definen cómo evolucionan esos objetos a lo largo del tiempo, y cuáles son las causas de esa evolución. Por ejemplo, un empleado puede pasar de un proyecto a otro; el sueldo de un empleado puede variar al recibir un ascenso; un proyecto puede pasar del estado de aprobado al de comenzado, o del de terminado al de aceptado; la preventa de un proyecto puede necesitar de ciertas actividades definidas en un flujo.

Los diagramas estructurales o estáticos de UML 2.5 son:

- Diagrama de casos de uso.
- Diagrama de objetos (estático).
- Diagrama de clases.
- Diagrama de paquetes.
- Diagrama de componentes.
- Diagrama de despliegue.
- Diagrama de estructuras compuestas.

Y los diagramas de comportamiento o dinámicos son:

- Diagrama de secuencia.
- Diagrama de comunicación.
- Diagrama de máquina de estados o de estados.
- Diagrama de actividades.
- Diagrama de visión global de la interacción.
- Diagrama de tiempos.

No todos los diagramas son igual de útiles, menos aún en un libro de programación. Por eso, nos detendremos solamente en algunos de ellos.

Representación de objetos

Un objeto se representa con un rectángulo, en el que figura el nombre del objeto y la clase de la cual éste es instancia, como se muestra en la figura 8.1.

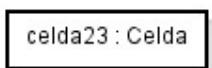


Figura 8.1 Un objeto

Hay ocasiones en que necesitamos modelar un objeto sin que nos interese conocer exactamente su clase. Y otras en las que nos interesa representar una instancia de una clase sin darle un nombre. Estas dos circunstancias están ejemplificadas en las figuras 8.2 y 8.3, respectivamente.

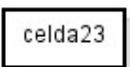


Figura 8.2 Un objeto del que no nos interesa su clase

: Celda

Figura 8.3 Una instancia de la que sólo conocemos su clase

Los objetos, en un sistema, pueden relacionarse con otros objetos. Esto se puede representar con un diagrama de objetos que muestre varios de ellos en un momento dado, con sus relaciones o enlaces, como se muestra en la figura 8.4.

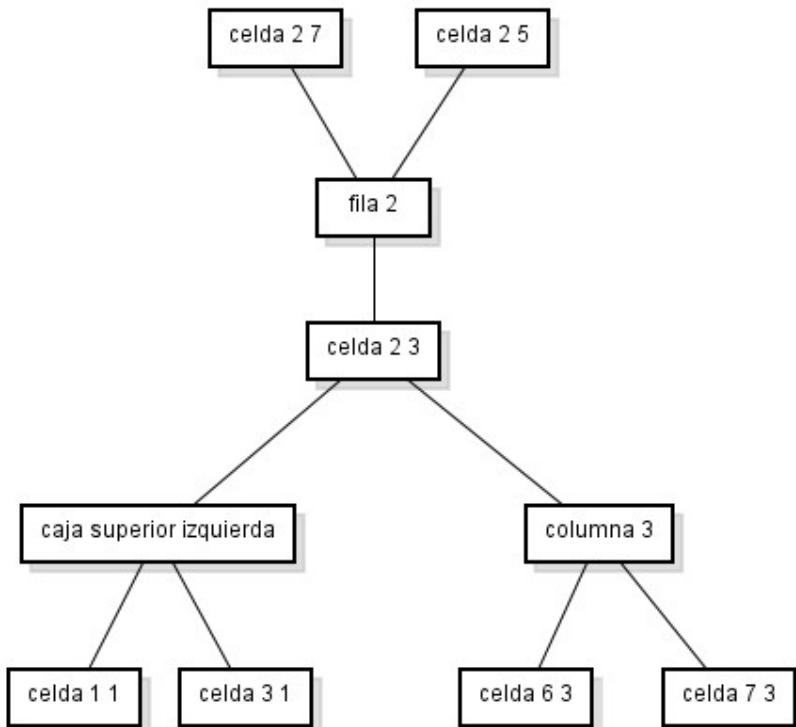


Figura 8.4 Diagrama de objetos

Los diagramas de objetos, indicando vínculos entre instancias y valores de atributos, pueden ser utilizados para razonar sobre el diseño. Un aspecto interesante de los mismos es que nos muestran algunos objetos y sus enlaces en un momento particular de la ejecución de la aplicación que nos interesa analizar.

Representación de comportamiento: **diagramas de secuencia y de comunicación**

Tal vez el mejor diagrama para encontrar métodos en objetos sea el diagrama de comunicación⁴⁹.

El diagrama de comunicación es un diagrama de objetos en un escenario, al que se le agregan los mensajes que los objetos se envían entre sí. La figura 8.5 muestra un diagrama de comunicación que ya hemos usado, con algunos agregados.

⁴⁹ El diagrama de comunicación de UML 2, es prácticamente el mismo que en versiones anteriores del lenguaje se llamaba diagrama de colaboración. Y si bien el nombre fue modificado porque en UML ya existía otro concepto denominado colaboración, casi todos los profesionales siguen refiriéndose al diagrama con el nombre antiguo. A pesar de ello, en aras de mantener el léxico oficial, en este libro vamos a llamarlo diagrama de comunicación.

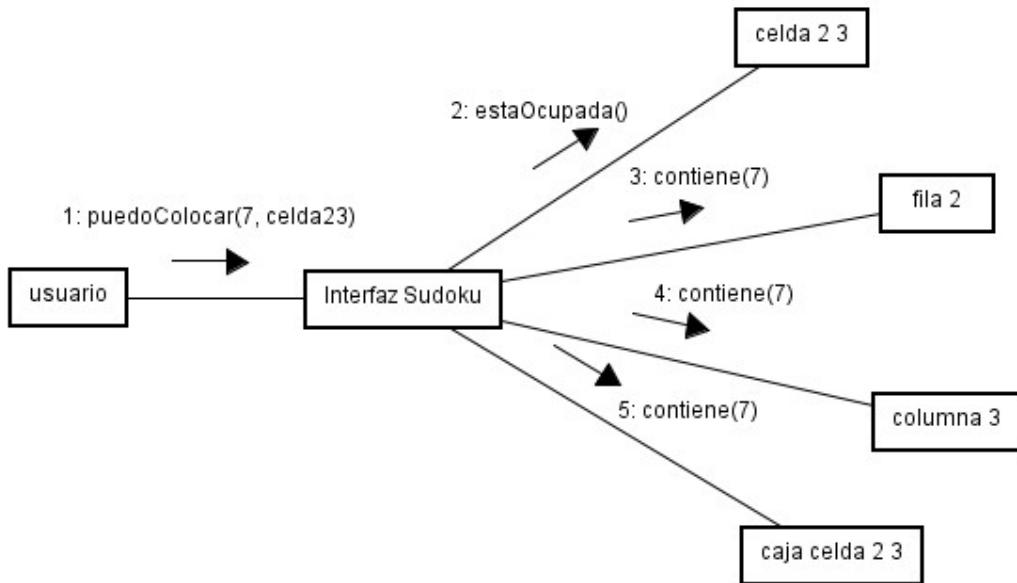


Figura 8.5 Diagrama de comunicación

Como decíamos, es un diagrama de objetos. Pero además de representar los vínculos entre objetos, muestra los mensajes que esos objetos se envían en el escenario en cuestión.

Los mensajes están representados con flechas sobre las líneas que denotan vínculos. Además, el orden de los mensajes se representa mediante el número que los precede que, como muestra el ejemplo, pueden ser anidados, para mostrar que un mensaje provoca el envío de otro. También es posible modelar el envío de un mensaje de un objeto a sí mismo.

Como también se puede ver, no es sencillo representar concurrencia en un diagrama de comunicación. Sí se puede representar el hecho de que un mensaje sea asíncrono, esto es, que el objeto que lo envió no se queda esperando la respuesta. Un mensaje asíncrono se representa con una flecha de punta abierta.

Decíamos que el diagrama de comunicación es ideal para encontrar métodos de las clases. Esto es así porque, al modelar los mensajes y las relaciones entre objetos en un escenario, nos indica qué métodos deben tener las clases de las cuales esos objetos son instancias.

Los diagramas de comunicación no permiten mostrar comportamiento condicional, concurrencia ni iteraciones, aunque hay muchos profesionales que han inventado sus propias notaciones para esto. Esto es así porque se suelen usar para modelar interacciones simples y secuenciales.

Ahora bien, el diagrama de comunicación es un caso particular de una familia de diagramas que se llaman **diagramas de interacción**. Otro miembro de la familia es el diagrama de secuencia.

Semánticamente hablando, los diagramas de comunicación y de secuencia son equivalentes. Sin embargo, hay algunas posibilidades de modelado del diagrama de secuencia que no están presentes en el de comunicación.

Por ejemplo, en la figura 8.6 vemos un diagrama de secuencia que ya conocemos.

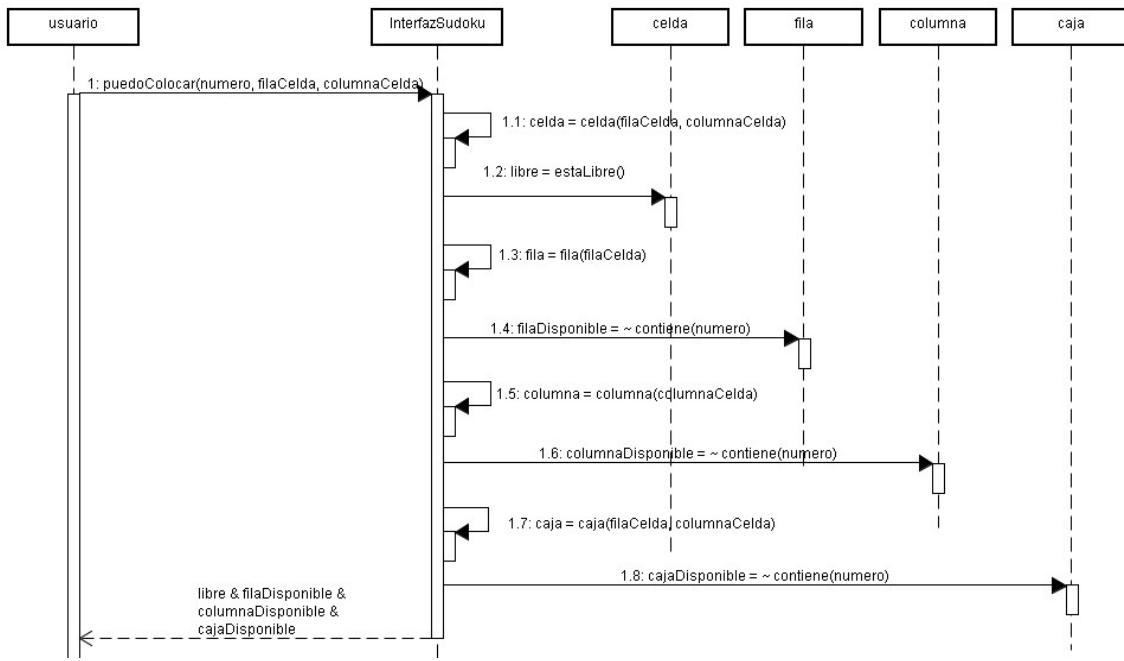


Figura 8.6 Diagrama de secuencia

Algunas de las cuestiones que se pueden observar en el diagrama son:

- Cada objeto se coloca arriba de una línea, llamada línea de vida. La misma puede tener un rectángulo cubriendola parcialmente, para indicar que en ese momento el objeto está activo.
- Cada mensaje se escribe en forma horizontal, entre la línea de vida del objeto que envía el mensaje y la de aquél que la recibe.
- Si un objeto se envía un mensaje a sí mismo, se puede dibujar como una flecha que vuelve sobre él mismo.
- El ordenamiento vertical representa el paso del tiempo. Esto es, un envío de mensaje que está más arriba en el diagrama quiere decir que ocurre antes que el que figura más abajo. Esta simbología hace que no sea necesario colocar números de orden a los mensajes, como sí era necesario en el diagrama de comunicación.
- La terminación de un mensaje con varios envíos de mensajes anidados se puede indicar con una flecha de línea punteada. Si bien esto no es obligatorio, a veces agrega claridad al diagrama. Nosotros usamos este recurso con el retorno del mensaje `puedoColocar`.

El diagrama de secuencia permite indicar también condicionales y ciclos iterativos. Si bien recomendamos tratar de evitar estas prácticas, para facilitar la lectura del diagrama, la figura 8.7, también conocida, muestra una guarda condicional [*no existe*] y algunas cosas más, como el estereotipo <<create>> para indicar que ese mensaje provoca la creación de un objeto (hay un <<destroy>> para la inversa y la posibilidad de mostrar la muerte de un objeto con una cruz en su línea de vida. Los estereotipos son opcionales, debido a que el significado de los mismos – creación o destrucción – puede ser deducido del diagrama sin ellos; sin embargo, es habitual colocarlos.

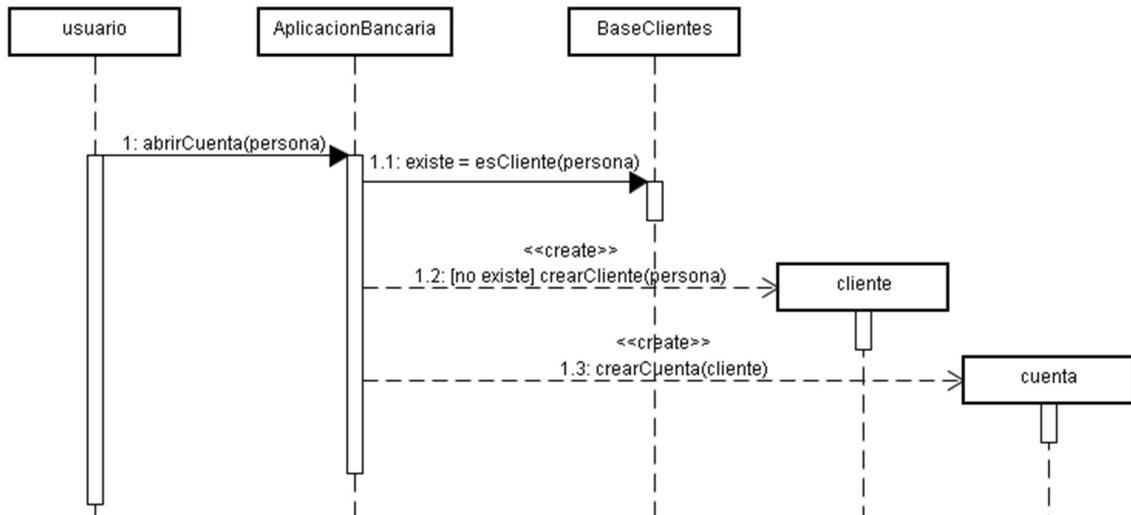


Figura 8.7 Diagrama de secuencia con guardas y estereotipos

Una iteración se puede representar con un asterisco.

En realidad, UML estricto exigiría usar marcos alrededor de las partes del diagrama que modela las acciones que se van a repetir, con el rótulo *loop* y alguna descripción de las condiciones del ciclo. De la misma manera se podría indicar un comportamiento condicional, pero en ese caso se utilizaría el rótulo *opt*. Pero nuestra notación parece más clara en los casos simples.

No obstante lo dicho, tal vez no sea la mejor idea mostrar condiciones o repeticiones en los diagramas de interacción. Lo mejor es representar escenarios simples que sean fácilmente comprendidos de un vistazo. Puede que las iteraciones no siempre convenga eliminarlas de los diagramas, pero a las condiciones suele ser conveniente separarlas en dos diagramas separados.

Otra característica posible de los diagramas de interacción (comunicación o secuencia) es mostrar cuándo un mensaje es asíncrono, es decir, el cliente no queda esperando una respuesta del receptor. Esta circunstancia se representa con una flecha de punta abierta. Por el momento no lo vamos a usar.

Los diagramas de secuencia son muy ricos, y admiten muchas variantes más. Hay quienes, para indicar que la llamada a un mensaje no es instantánea, sino que lleva un tiempo, realizan las flechas que los representan levemente oblicuas en vez de horizontales. Usando esta misma posibilidad, podríamos representar mensajes cuyas flechas se cruzan, cuando el comienzo de la llamada de uno es previo a otro, pero tarda más en llegar al receptor y puede adelantarse. No obstante, a pesar de que algunas personas usan estas facilidades, no es lo más habitual hacerlo.

Dado que los diagramas de secuencia son equivalentes en semántica a los de comunicación, una pregunta interesante para plantearse es cuándo conviene usar uno u otro. Lo cierto es que, por la forma en que se dibujan, los diagramas de secuencia son más aptos para representar el paso del tiempo y analizar temporalmente un escenario, mientras que los diagramas de comunicación resultan interesantes para hacer un primer esbozo de las relaciones entre objetos, brindando mayor libertad al permitir desplegar los objetos en dos dimensiones.

Hay ocasiones en que no nos interesa modelar el participante que envía el mensaje disparador de un diagrama de secuencia, sea porque es irrelevante para el modelo o porque no conocemos exactamente de qué objeto se trata. Esto es habitual cuando una aplicación brinda servicios que pueden invocarse desde contextos muy variados. UML llama a estos mensajes, **mensajes encontrados**, y los representa como provenientes de un punto sin nombre. En la figura 8.8 mostramos esta posibilidad.

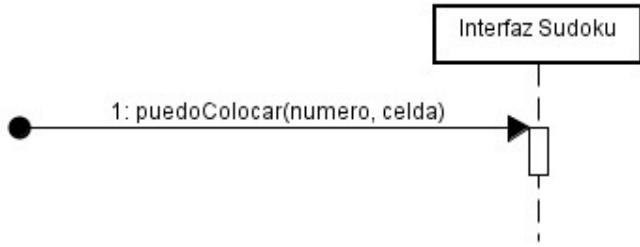


Figura 8.8 Mensaje encontrado

También hay ocasiones en que no nos interesa modelar quién recibe cierto mensaje. Esto último, un poco menos habitual, se podría usar para modelar un mensaje enviado a cualquier objeto que esté disponible para recibirllo. En términos de UML decimos que estamos en presencia de un **mensaje perdido**, que se modela como una flecha que termina en el vacío.

En teoría, un diagrama de secuencia permite diagramar cualquier algoritmo. Por eso, muchos profesionales caen en la tentación de usarlos para modelar código fuente. Pero mi opinión personal es que en general no se justifica, ya que el código fuente es también un modelo, y no tiene sentido repetirlo tanto en forma textual como gráfica. Hay quienes modelan con estos diagramas solamente el código fuente más intrincado o incomprensible. No obstante, aun en estos casos, es preferible mejorar la calidad del código antes de modelarlo con diagramas de secuencia detallados.

El diagrama de secuencia, tal como lo hemos venido estudiando, muestra tiempos solamente en el sentido de indicarnos el orden en que ocurren los envíos de mensajes. Sin embargo, UML admite mostrar restricciones temporales específicas, indicando cuánto tiempo debe pasar entre un mensaje y otro, o el momento exacto en que debe ocurrir un evento. En este libro no haremos uso de esta posibilidad.

La visión estática: diagrama de clases

La clase es una construcción de casi todos los lenguajes orientados a objetos. Esto hace que el diagrama de clases sea el diagrama estructural más importante a la hora de modelar diseño detallado y programación. Veamos el diagrama de la figura 8.9.

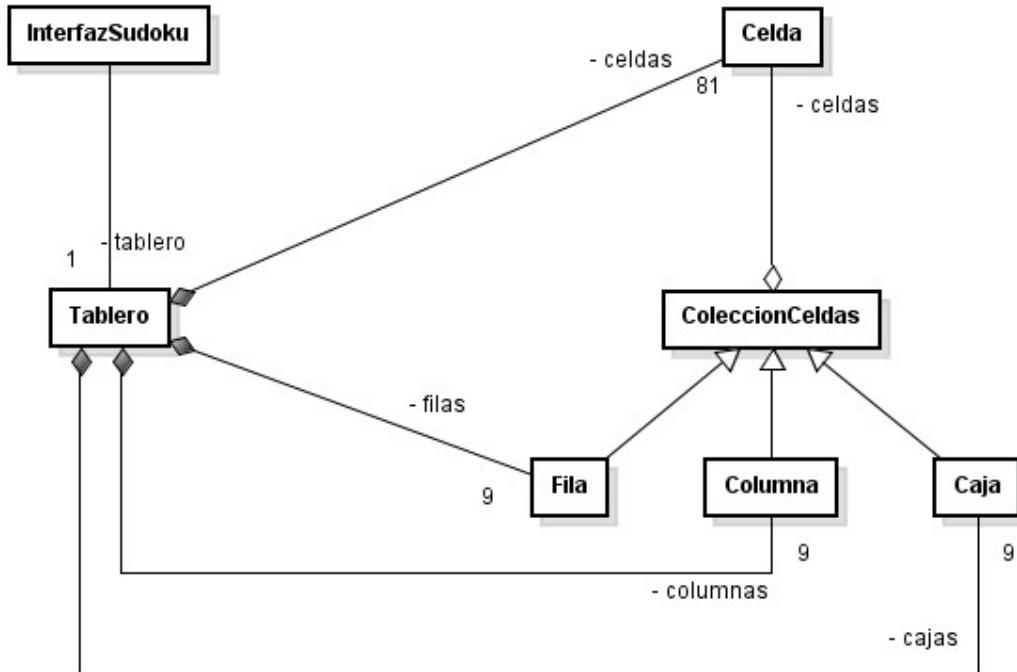


Figura 8.9 diagrama de clases simple

El diagrama que acabamos de realizar es todavía un poco diferente a los que veníamos haciendo, porque pretende ser más conceptual y menos detallado. Veamos.

- Cada clase se ha representado mediante un rectángulo con su nombre adentro.
- La asociación, representada por una simple línea, indica que una clase está relacionada con otra. Eso se muestra, por ejemplo, entre *InterfazSudoku* y *Tablero*.
- Las asociaciones tienen una **cardinalidad**, indicada por números o rangos al extremo de cada asociación. La cardinalidad de la relación nos informa con cuántos elementos de un concepto se relaciona cada elemento del otro. Por ejemplo, en el diagrama anterior, hemos indicado que *InterfazSudoku* tiene uno y sólo un *Tablero* y que un *Tablero* tiene 81 instancias de *Celda*.
- La cardinalidad de una relación puede expresarse de tres maneras: con un número, con un rango de números mediante el formato $n..m$, o mediante el uso del asterisco, que implica varios. Por ejemplo, si una cardinalidad se expresa como $1..3$, implica un rango de 1 a 3; si se expresa como $2..*$, implica que como mínimo es de 2, sin un máximo establecido; si se expresa sólo con un asterisco, tiene el significado de $0..*$.
- Habitualmente, uno de los extremos de asociación tiene cardinalidad 1. Esto surge directamente de la definición de cardinalidad, que indica cuántas instancias de una clase se relacionan con cada instancia de la otra. Por eso se suele asumir 1 cuando la cardinalidad no se indica, aunque esto no es normativo.
- La relación de generalización-especialización (o herencia, como se llama en los lenguajes) se indica con una línea terminada en flecha triangular vacía hacia la clase madre. Por ejemplo, en nuestro diagrama, estamos indicando que *Fila*, *Columna* y *Caja* son clases hijas de *ColeccionCeldas*.
- La relación de **agregación** (un término de UML que tratamos de no enfatizar mucho en

este libro) se representa mediante un rombo en el extremo de la asociación que corresponde al agregado, como hemos hecho al indicar que una instancia de *ColeccionCeldas* es una agregación de instancias de *Celda*.

- La relación de **composición** se representa mediante un rombo relleno en el extremo del contenedor, como hemos hecho al indicar que *InterfazSudoku* está compuesta por 9 instancias de *Fila*, 9 de *Columna*, 9 de *Caja* y 81 de *Celda*. El significado de la composición es que, cuando hay este tipo de agregación, las partes no pueden ser independientes del todo. Esto es, si deja de existir el todo, dejan de existir las partes, pues no tienen existencia independiente y el contenedor es responsable del ciclo de vida del objeto contenido. A nivel de implementación, esto es fuertemente dependiente del lenguaje. Por ejemplo, en los lenguajes que vinculan los objetos entre sí con un modelo de referencias, la composición es una decisión de diseño que el programador deberá implementar en código, y que muy probablemente afecte la manera de determinar igualdad, cómo hacer copias o cómo manejar la persistencia. En este libro no hemos hecho mucho énfasis en esas relaciones en los diagramas.
- Las asociaciones pueden tener un nombre en sí mismas (no hay en el diagrama) o definir un **rol** en uno de los extremos. Esto último lo hemos hecho al poner nombres en los extremos de las líneas que denotan asociaciones.

A modo de resumen, en la tabla que sigue mostramos las descripciones de distintos escenarios de los mecanismos de abstracción. Esta lista no pretende ser exhaustiva, pero muestra algunas de las situaciones más comunes y cómo se deberían modelar.

Relaciones entre clases		
Situación	Ejemplo en el diagrama	Relación
Las instancias de una clase son partes de una instancia de otra	ColeccionCeldas – Celda	Aggregación
Las instancias de una clase están contenidas en una instancia de otra	Tablero – Fila	Composición
Cada instancia de una clase contiene una relación (una referencia en nuestros lenguajes) con una instancia de la otra	InterfazSudoku – Tablero	Asociación
Una clase obtiene información de otra	No hay en el diagrama. Puede verse la línea de puntos terminada en flecha de las figuras 8.14 a 8.16	Dependencia
Una clase es un caso más general de otra clase (las instancias de una son un superconjunto de las instancias de la otra)	ColeccionCeldas – Fila	Especialización (inversa de generalización)
Una clase es un caso particular de otra clase (las instancias de una son siempre instancias de la otra)	Fila – ColeccionCeldas	Generalización (inversa de especialización)

Pero esto no es todo. Como hemos visto en capítulos anteriores, un diagrama de clases puede mostrar mucho más. Por ejemplo:

- Los métodos y los atributos de las instancias de cada clase. En estos casos, cada clase se representa como un rectángulo de 3 compartimientos. El primero es para el nombre de la clase, el segundo es para los atributos y el tercero para los métodos.
- También hemos visto que puede indicarse la navegabilidad, con una flecha en las asociaciones entre clases. Esto se hace para indicar cuál es la clase que necesita los servicios de la otra. Si hemos trabajado con diagramas de comunicación es fácil determinar la navegabilidad entre clases, ya que los envíos de mensajes entre objetos lo indican. Cuando no se indica la navegabilidad, UML explica que esta es bidireccional. Sin embargo, dado que tampoco indicamos navegabilidad cuando no queremos modelarla, es una buena práctica indicar la navegabilidad bidireccional con dos flechas en la relación.
- Una asociación entre dos clases, a nivel de implementación, significa que una clase tiene un atributo cuyo tipo es otra clase. La navegabilidad nos indica cuál es la clase del atributo, y el nombre del atributo es el rol de la asociación en el sentido de la navegabilidad.
- Y cuando queremos aclarar algo, podemos usar **notas**, que son rectángulos con texto adentro, vinculados mediante una línea de puntos.
- Las clases y métodos abstractos se escriben en cursiva. Hay personas que en vez de escribir las clases abstractas en cursiva, le agregan la propiedad *{abstract}* al lado del nombre. Esto es especialmente importante cuando, en vez de usar una herramienta de software modelamos a mano alzada para comunicarnos informalmente entre humanos, ya que no es sencillo escribir en cursiva en una pizarra o papel de forma que resulte clara esta distinción.
- Incluso podemos mostrar los tipos de métodos, parámetros y atributos y la visibilidad de atributos y métodos.

El diagrama de la figura 8.10 es más completo que el anterior, pues tiene todo este nivel de detalle.

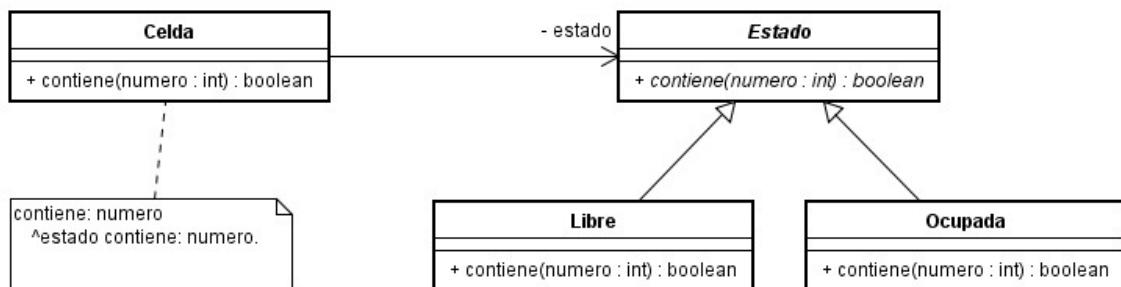


Figura 8.10 diagrama de clases en mayor nivel de detalle

En la misma figura vemos algo que, si bien venimos usando, no hemos explicitado aún: la visibilidad de un atributo o método, que se indica con los calificadores de visibilidad `+`, `-`, `#` y `~`.

Los calificadores de visibilidad tienen los siguientes significados:

Calificador	Significado
-------------	-------------

+	Visibilidad pública. Habitualmente significa la misma visibilidad que la clase que contiene el elemento. Esto es, el elemento es visible para cualquier otro elemento para el que la clase sea visible.
-	Visibilidad privada. Habitualmente significa que el elemento sólo es visible dentro de la clase que lo contiene. En algunos lenguajes sólo es visible para el objeto en sí mismo, no para la clase.
#	Visibilidad protegida. Habitualmente significa que el elemento sólo es visible dentro de la clase que lo contiene y las clases que heredan de ella.
~	Visibilidad de paquete. Especialmente introducido en UML para representar la visibilidad de paquete de Java: significa que el elemento sólo es visible para las clases del mismo paquete que la clase que lo contiene.

Hay más elementos interesantes de los diagramas de clases. El estereotipo `<<interface>>`, útil en los lenguajes que manejan interfaces, junto con la notación de realización o implementación, que se marca como una herencia de línea punteada, son muy útiles. Esto se ve en la figura 8.11.

También vemos allí la aparición de una clase de asociación y de una asociación calificada. En ambos casos, está marcando que la relación entre *Cuenta* y *Notifiable* se materializa mediante una *Collection*.

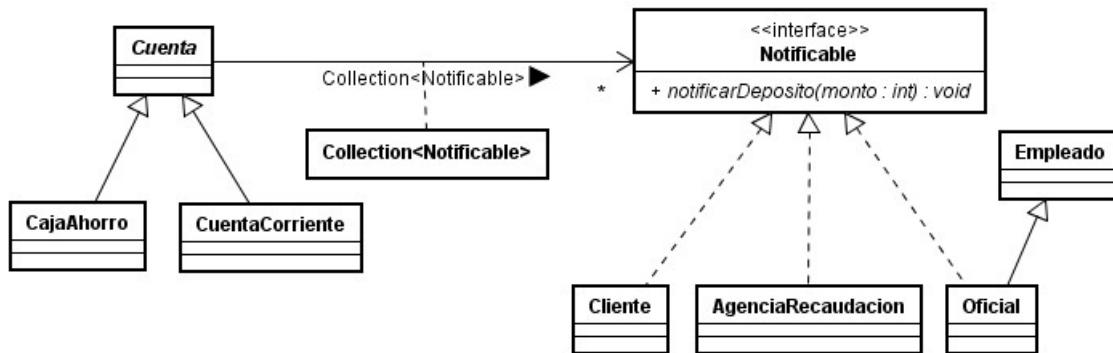


Figura 8.11 clases de asociación e interfaces

Hay cuestiones que UML provee para ciertos lenguajes en particular. Por ejemplo:

- Para los lenguajes que, como Smalltalk, manejan metaclasses – esto es, clases cuyas instancias son a su vez otras clases – existe el estereotipo `<<metaclass>>`.
- También se permite representar una clase con un tipo parámetro, para denotar lo que en este libro vemos como genericidad. La figura 8.12 muestra que la clase *ArrayList* de Java tiene un parámetro genérico *E*.
- También existe la posibilidad de representar clases internas. Esto, por supuesto, tiene diferentes significados según el lenguaje de programación.
- Incluso la particularidad de las clases internas anónimas, como existen en Java, se puede representar con el estereotipo `<<anonymus>>`. La figura 8.13 muestra que el iterador que permite recorrer un *ArrayList* es una clase interna y anónima que implementa la interfaz genérica *Iterator*.



Figura 8.12 Clase parametrizada

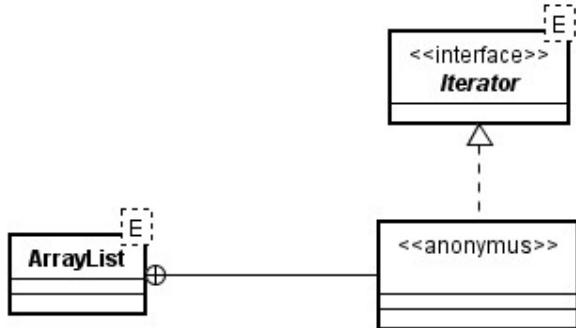


Figura 8.13 Clase interna anónima

La visión macro: paquetes

El diagrama de paquetes de UML es una herramienta que sirve para agrupar elementos estáticos y es, por definición, un elemento estructural.

Cuando decimos que sirve para agrupar elementos estáticos estamos englobando varios tipos de diagramas. Por ejemplo, un paquete podría agrupar clases, pero también objetos o casos de uso, e incluso otros paquetes.

Veamos. Si la aplicación se va a desarrollar siguiendo el patrón de arquitectura de tres capas, un diagrama de paquetes podría ser el de la figura 8.14.

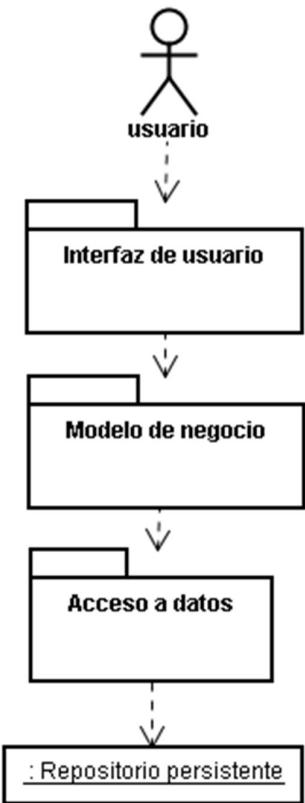


Figura 8.14 Diagrama de paquetes para una arquitectura de tres capas

Notemos los siguientes elementos básicos del diagrama de paquetes:

- Un paquete se representa como una carpeta con solapa. La solapa se puede utilizar para poner el nombre del paquete, aunque esto sólo se suele hacer en casos de necesitar escribir otras cosas dentro de la carpeta.
- Las dependencias entre paquetes se muestran como las dependencias simples del diagrama de clases.

Se dice que hay una dependencia entre paquetes cuando una modificación en un paquete puede provocar una modificación en el dependiente. El ejemplo del patrón de tres capas es bastante ilustrativo: la capa de interfaz de usuario depende de la capa lógica o de modelo, ésta depende de la de acceso a datos, y esta última del repositorio de datos persistentes.

Hay algunos elementos en el diagrama anterior que no son del todo parte de UML estándar. El símbolo que usamos para representar el repositorio persistente es el de un objeto, que no suelen representarse en diagramas de paquetes pero son bastante adecuados en diagramas de arquitectura como este. Respecto del usuario, lo hemos representado con el mismo esquema de persona que se usa para los actores en los diagramas de casos de uso. Esto es algo bastante usual, pero no del todo estándar.

Por supuesto, como decíamos más arriba, un paquete puede contener paquetes internos. En este caso, simplemente se dibuja el paquete interno dentro del externo y se lleva el nombre del más externo a la solapa. Eso lo hemos hecho en la figura 8.15, que muestra con un poco más de detalle la estructura lógica de un sistema.

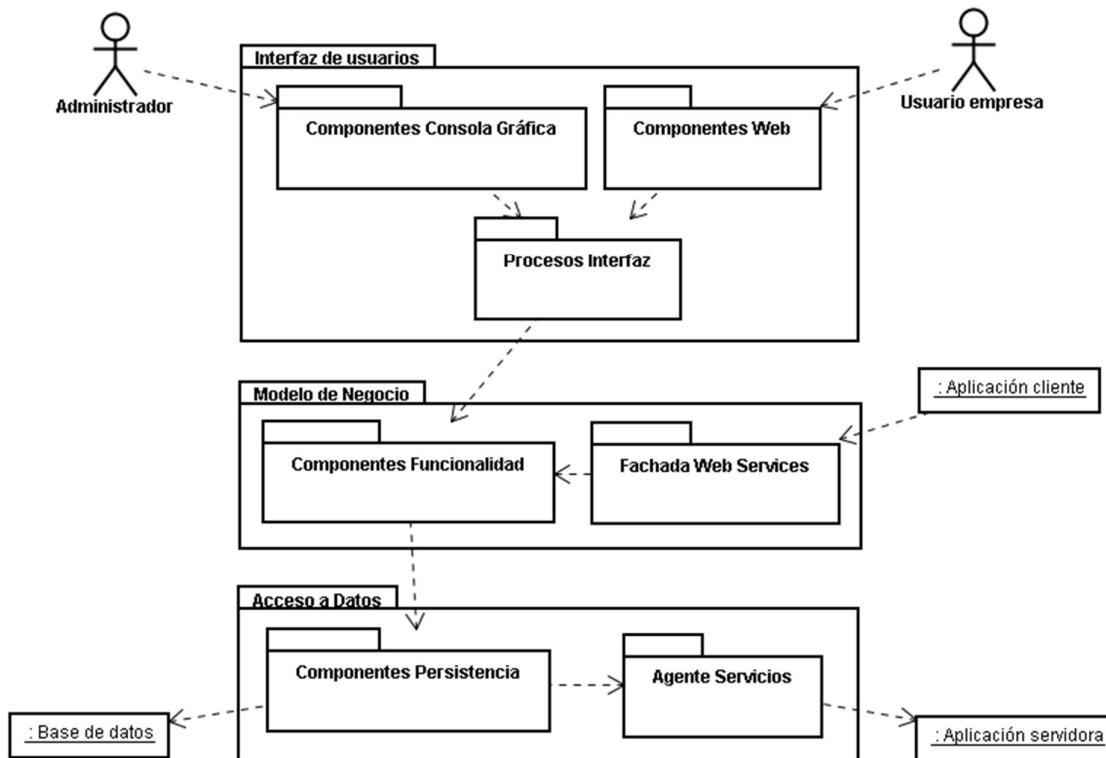


Figura 8.15 Diagrama con paquetes internos

También se puede indicar, en un diagrama de paquetes, el uso de bibliotecas externas y frameworks, con los estereotipos respectivos. La figura 6.3 muestra un caso más detallado de la figura 8.16, donde mostramos que para la persistencia estaremos usando el framework Hibernate, para servicios web el framework Axis, para componentes web el framework Struts y para componentes de consola gráfica la biblioteca Swing.

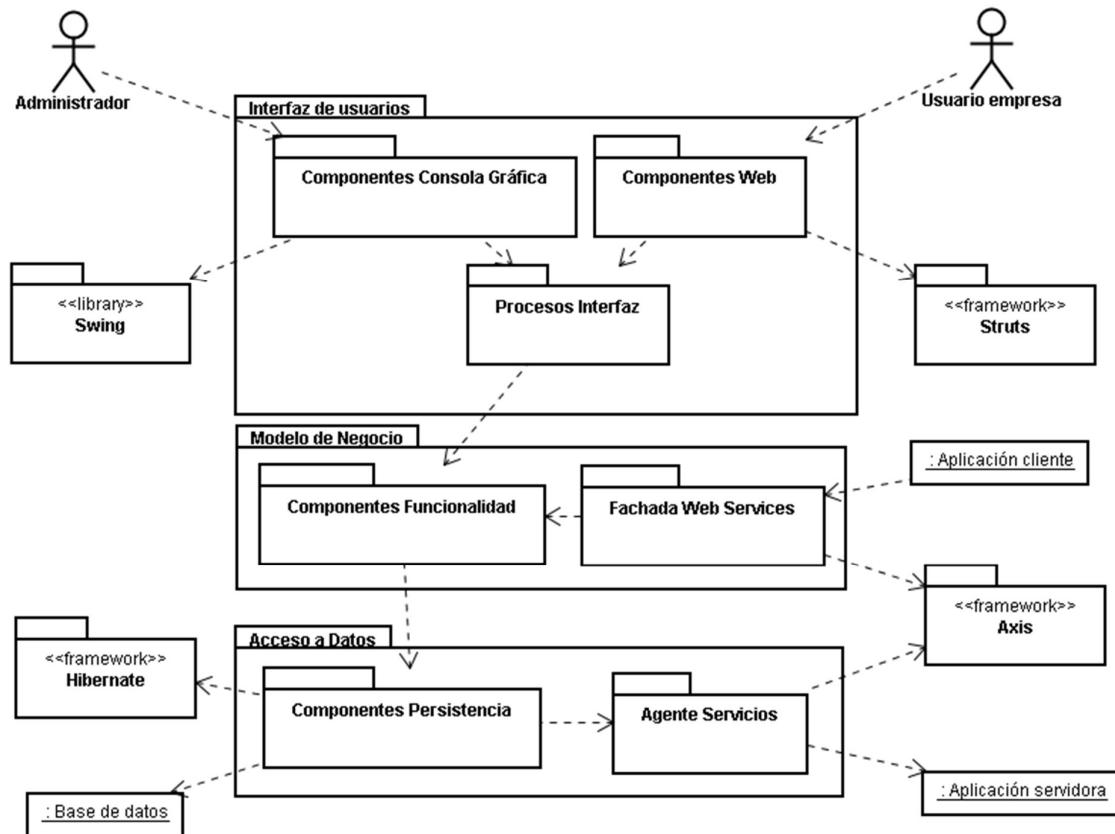


Figura 8.16 Diagrama con frameworks y bibliotecas

Hay mucho más que se puede avanzar con diagramas de paquetes. Entre ellos, hay distintos estereotipos estándar para las dependencias, aunque rara vez se usan. Y también tenemos la posibilidad de agrupar otros elementos, de los cuales se usan sobre todo las clases. La figura 8.17 muestra un diagrama de paquetes y clases combinados.

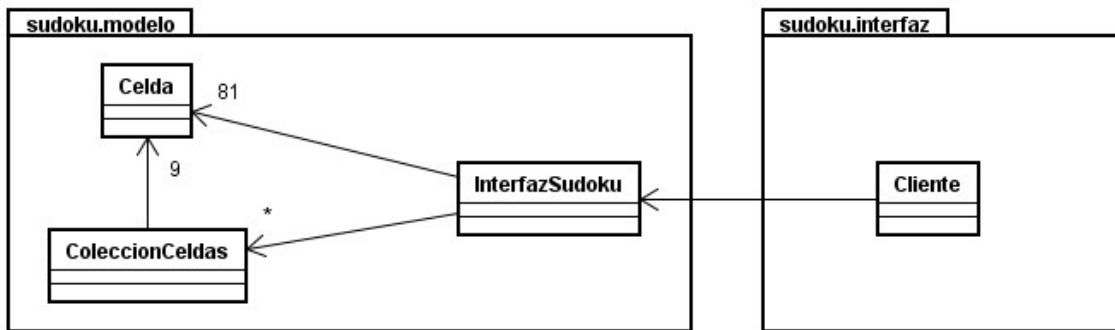


Figura 8.17 Paquetes y clases

También se pueden mostrar las clases internas de un paquete sin decir nada de sus interrelaciones ni de cómo se relacionan las clases entre paquetes, solamente colocando sus nombres. En este caso se puede también especificar si las clases internas son o no visibles para otros paquetes, con los atributos de visibilidad que vimos en los diagramas de clases. La figura 8.18 muestra el mismo diagrama anterior con este otro formato.

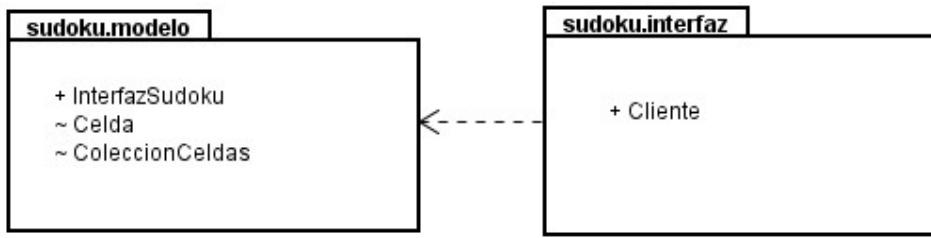


Figura 8.18 Visibilidad de elementos internos

Diagramas de actividades

El diagrama de actividades de UML es una herramienta sencilla para mostrar flujos de trabajo. Por ejemplo, cuando definimos TDD lo acompañamos con el diagrama de la figura 8.19.

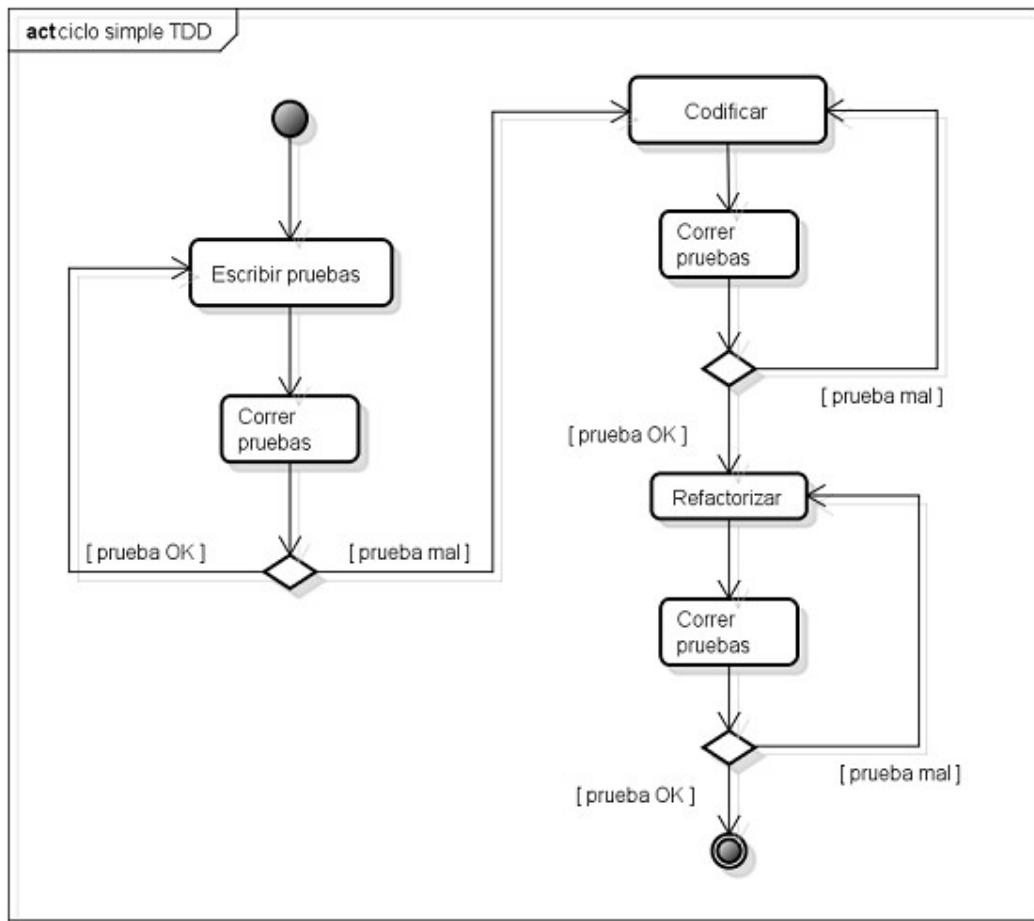


Figura 8.19 Diagrama de actividades de TDD

Notemos los elementos típicos de un diagrama de actividades:

- Los rectángulos de bordes redondeados son actividades o acciones en el flujo. Dentro de los mismos se coloca una descripción breve de la actividad.
- Las flechas indican el sentido del flujo.
- El comienzo y fin del flujo se indican con un círculo negro y un círculo blanco con uno negro concéntrico, respectivamente.

- Las bifurcaciones condicionales se especifican con un rombo, colocando la condición de las ramas – llamada condición de guarda – entre corchetes.

Hay otras características interesantes de los diagramas de actividades, como la posibilidad de indicar acciones concurrentes – que se podrían realizar al mismo tiempo o que no importa el orden en que se realicen – y la división en “calles” verticales para indicar los responsables de cada parte de la interacción. Ambas cosas se ven en la figura 8.20.

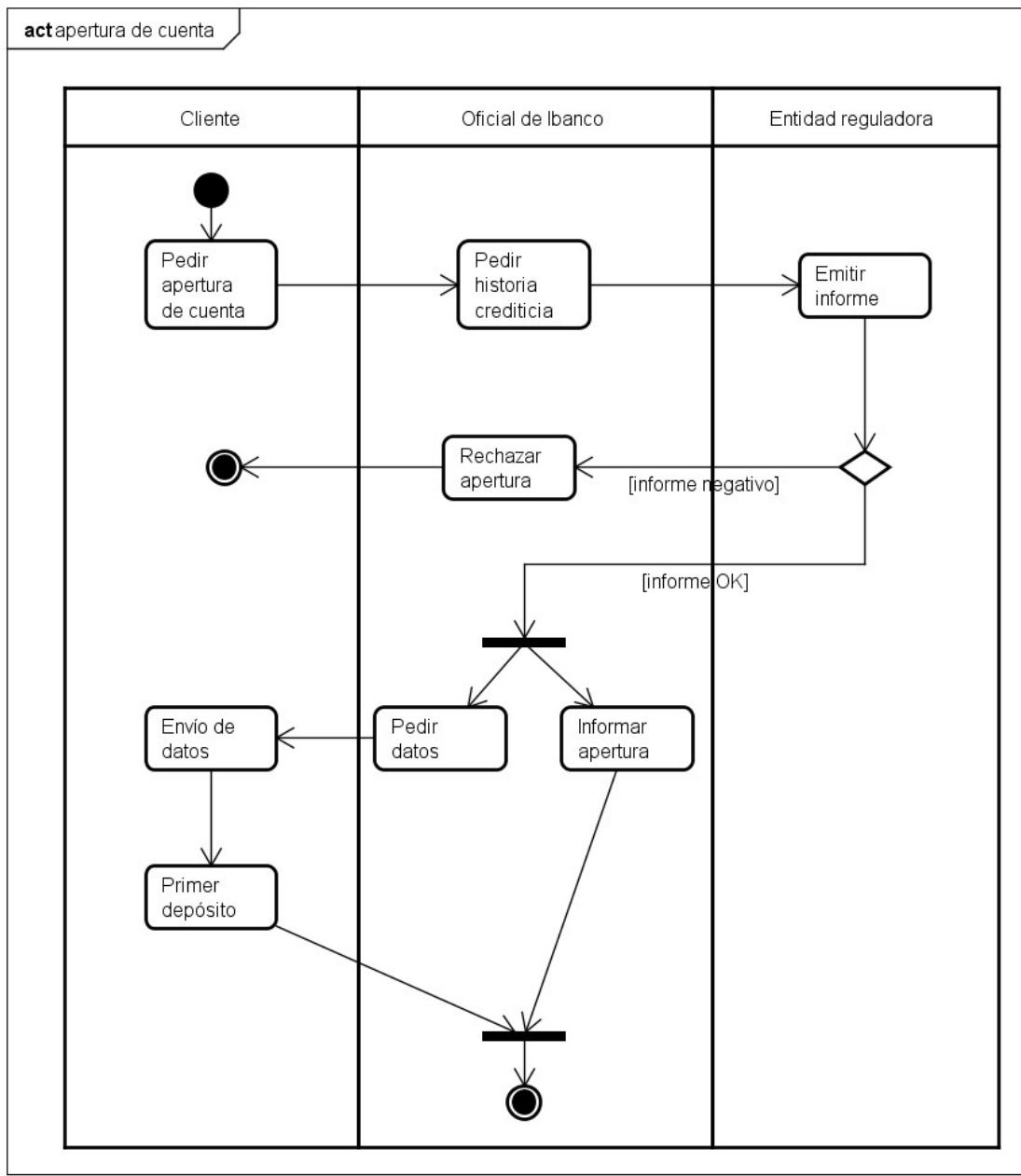


Figura 8.20 Diagrama de actividades con calles y concurrencia

Como vemos en el diagrama, las acciones concurrentes se dibujan naturalmente, con dos barras gruesas (llamadas barras de sincronización), una para indicar el comienzo de la concurrencia y otra para el fin (se los suele denominar conectores fork y join).

Al usar calles, como vemos en la figura, pudimos especificar quién hace cada acción. De otro modo, deberíamos haber colocado el nombre del sujeto en cada una de ellas.

Hay ocasiones en que puede ser conveniente abrir una actividad en varias sub-actividades, o reunir algunas actividades en una actividad compuesta. Lo hemos hecho en el capítulo que explicamos la técnica llamada ATDD.

Diagramas de estados

El diagrama de estados de UML es una herramienta que sirve para modelar cómo afecta un escenario a los estados que un objeto toma, en conjunto con los eventos que provocan las transiciones de estado. También es habitual llamarlo diagrama de máquina de estados, o máquina de estados a secas. En este libro usaremos siempre el nombre más habitual de diagrama de estados.

Los cambios de estado que sufren los objetos se deben a estímulos o mensajes recibidos de otros objetos, cuyos efectos UML llama **eventos**. Un evento indica la aparición de un estímulo que puede disparar una transición de estados. Es la especificación de un acontecimiento significativo.

Una **transición** entre estados es el paso de un estado a otro: un objeto que esté en un primer estado realizará ciertas acciones y entrará en un segundo estado cuando ocurra algún evento especificado y se satisfagan ciertas condiciones.

El diagrama de estados es un modelo dinámico que muestra los cambios de estado que sufre un objeto a través del tiempo. Se puede modelar toda la vida del objeto o su existencia dentro de un escenario particular.

El diagrama de estados, por lo tanto, es una abstracción que representa los estados, eventos y transiciones de estados para un objeto o un sistema: muestra la secuencia de estados en la vida de un objeto, los eventos que modifican estados y las acciones y respuestas del objeto.

La representación gráfica consiste en un grafo o red con nodos para los estados y arcos para las transiciones, además de un texto en correspondencia con los arcos que describen eventos, condiciones y acciones de las transiciones.

Por ejemplo, la figura 8.21 muestra un diagrama de los estados de la vida de una cuenta corriente en nuestro ejemplo de aplicación bancaria.

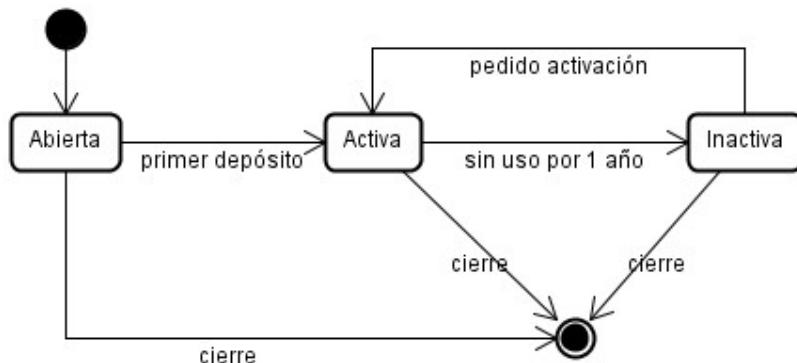


Figura 8.21 Diagrama de estados simple

Los elementos distintivos del diagrama de estados son:

- Los estados por los que pasa el objeto (la cuenta en este caso) se representan con

rectángulos de puntas redondeadas. En la figura vemos los estados *Abierta*, *Activa* e *Inactiva*.

- Los nodos inicial y final se representan igual que las actividades de inicio y fin del diagrama de actividades. Como en realidad no son estados, se usa simplemente el término nodo.
- Las transiciones entre estados se representan mediante líneas con flechas.
- Los eventos que provocan las transiciones se denotan con texto sobre las flechas. Por ejemplo, en la figura vemos los eventos *primer depósito* y *cierre*, entre otros. En casos más complejos, se puede colocar una leyenda del tipo: *evento [condición] / acción*. El significado de esta leyenda es que la transición se realiza cuando sucede el *evento* y se cumple la *condición*, realizando la *acción* durante la transición.

Hay cuestiones más complejas que pueden representarse en estos diagramas, que en este libro vamos a evitar. Por ejemplo:

- Un estado puede agrupar varios estados internos.
- Se puede mostrar la concurrencia de estados, para indicar que el objeto puede estar en más de un estado al mismo tiempo.
- También se pueden definir acciones internas a un estado, que se indican mediante las palabras entry (para indicar una acción que se realiza al ingresar al estado), do (para indicar acciones mientras se mantiene el estado) y exit (para indicar una acción de salida del estado). La figura 8.22 muestra un ejemplo de uso de estas cuestiones:

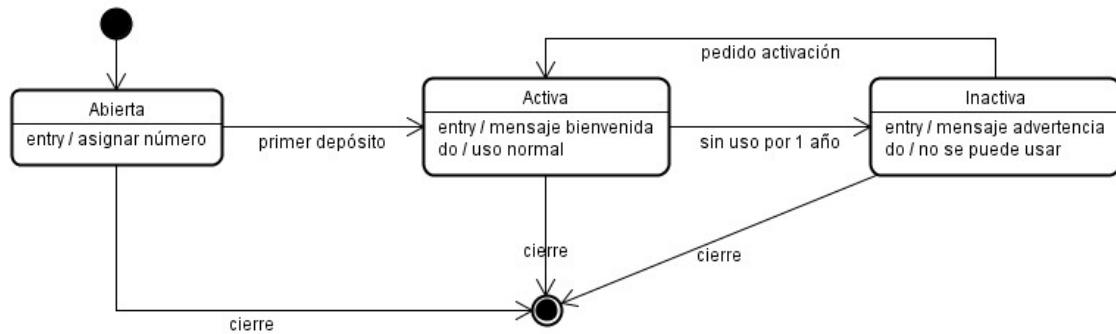


Figura 8.22 Actividades dentro de los estados

Como vemos, el diagrama de estados es un modelo sencillo. Por eso mismo, hay que tratar de que permanezca simple, cuidando el nivel de detalle, sin hacer diagramas muy complejos. Son ideales en el caso en que una especificación mediante estados es más ilustrativa que una que describa actividades, o cuando los objetos tienen un comportamiento complejo. De hecho, aunque hay mucho más, nos quedamos con un uso simple de estos diagramas.

Otros diagramas de UML

Como dijimos más arriba, hay muchos otros diagramas de UML. He aquí un resumen de los que no hemos analizado y su aplicación:

- Diagrama de casos de uso: es un diagrama pensado para modelar especificaciones de requerimientos. Más allá de tener un uso bastante amplio, no es tan útil como aparenta: sólo muestra los casos de uso, sus relaciones y otras cuestiones estáticas. De todas

maneras, es el de mayor uso entre los que exceden lo que vemos en este libro.

- Diagrama de despliegue: muestra nodos de hardware, sus relaciones, piezas de software y cómo se despliega el software en el hardware. Es similar a un diagrama de objetos, pero donde los objetos son nodos de hardware (computadoras, servidores, dispositivos en general).
- Diagrama de componentes: muestra relaciones entre componentes de software y sus interfaces. Es un buen complemento a los diagramas de clases y de paquetes, pero más útil cuando modelamos componentes concretos de software.
- Visión global de la interacción: es un diagrama de uso poco habitual, que combina actividades e interacciones en un único diagrama.
- Diagrama de tiempos: es un diagrama que sólo se usa como sucedáneo de los diagramas de secuencia, si interesa modelar restricciones temporales; es de uso poco habitual, pero interesante.
- Diagrama de estructura compuesta: diagrama de uso poco habitual que muestra la estructura interna de una clase o componente.

Ejercicio adicional

Contenido todavía no disponible en esta versión del libro.

Ejercicios propuestos

Contenido todavía no disponible en esta versión del libro.

Recapitulación

En este capítulo nos hemos detenido en UML, el lenguaje de modelado que es el estándar de facto en POO, y nos hemos concentrado en los tipos de diagramas más usuales en el diseño y la programación.

El próximo capítulo planteará una serie de temas que veníamos dejando de lado por no aportar al núcleo de POO, pero que son parte del paradigma y no podemos dejar de abordarlos en el libro.

9. Lo que fue quedando de lado

Contexto

Hasta este momento, al ir presentando los temas a lo largo del libro, hemos tratado de trabajar de una manera bien orientada a objetos y evitando entrar en cuestiones de más difícil abordaje. Este capítulo va a volver sobre algunos temas que han quedado tratados muy superficialmente y sobre otros que ni siquiera hemos visto, pero que son complementos interesantes a lo ya estudiado.

Necesariamente, se trata de un capítulo heterogéneo en su temática y con escasa vinculación entre los tópicos que se tratan: excepciones, atributos y métodos de clase, genericidad, información de tipos en tiempo de ejecución y reflexión.

Excepciones

Errores y excepciones

La idea de trabajar con excepciones tiene que ver con que, en el punto en que surge un error, no siempre se sabe qué decisión tomar. Supongamos el caso de una función de biblioteca que deba obtener el promedio de una serie de valores contenidos en un arreglo. ¿Qué se supone que debe hacer la función si el arreglo que se le pasa está vacío? El problema surge porque en el contexto de la función no hay suficiente información para resolver la anomalía, por lo que se debe salir de la misma hacia el módulo que la invocó informando esta situación, pero sin provocar la finalización del programa ni enviar mensajes a un presunto usuario del que no sabemos nada.

Notemos que hablamos de excepciones cuando el problema no se puede resolver en un determinado contexto. La idea es que cuando surge una excepción no hay forma de continuar y se debe elevar la misma a un contexto de nivel superior para que resuelva el inconveniente. Usando términos del modelo contractual, como ya dijimos, una excepción se provoca cuando no se cumple una precondición.

Una solución ingenua es pedirle ayuda al usuario final. Decimos que es ingenua debido a que no considera que hay muchos errores que un usuario final no es capaz de entender ni de resolver.

En la programación tradicional, el modo habitual de tratar situaciones de excepción fue devolviendo un valor especial, poniendo un valor en una variable global o mediante un parámetro ad hoc. Esto tenía más de un inconveniente.

En primer lugar, a menudo los programadores clientes no chequean estas variables o valores especiales. Esto puede parecernos un comportamiento equivocado, pero pasa a ser bastante lógico cuando para chequear cada valor de retorno hay que convertir al código en una maraña de verificaciones anidadas.

Por otro lado, en un sentido similar, el chequear las condiciones de error deja el código normal o básico muy acoplado con aquél que se ha escrito para manejar los errores, haciendo que se pierda la claridad del programa.

La idea de trabajar con excepciones es aislar el código que se usa para tratar problemas del que podemos llamar código básico. De esta manera el código básico queda más limpio. Además, brinda una forma unificada de reportar errores.

En términos de POO, adoptemos la siguiente definición:

Definición: excepción

Una excepción es un objeto que se usa para comunicar una situación excepcional desde un entorno que la detecta al ámbito desde el cual fue invocado.

Notemos que esta definición tiene varios elementos interesantes:

- Una excepción es un objeto: en efecto, las excepciones son objetos con comportamiento, identidad y estado. En los lenguajes con clases, son instancias de ciertas clases especiales.
- Se envía desde un entorno que la detecta; habitualmente hay un método que crea y lanza un objeto de excepción porque no puede resolver el problema en su ámbito.
- Se envía al ámbito desde el cual fue invocado: el objeto viaja desde el método que detectó el problema hasta el ámbito desde el cual ese método fue invocado. Este ámbito podrá recibir ese objeto y ver de qué manera tratar el problema.
- Se usa para comunicar una situación excepcional: su principal uso es la comunicación entre dos ámbitos, el que detecta el problema y el que debe lidiar con el mismo.

Bueno, hechas estas consideraciones, pasemos a estudiar excepciones más a fondo.

Lanzamiento de excepciones

Como dijimos, al surgir una situación excepcional, la idea es que el código que encuentra un problema y no lo pueda tratar eleve (lance, arroje) una excepción, que será atrapada o capturada por el módulo que invocó al método que está corriendo. Dicho módulo podrá manejar el problema como mejor le parezca.

Es decir, cuando se dispara una excepción, se corta el hilo de ejecución normal y se eleva la excepción hacia el contexto que invocó el método. De alguna manera, elevar una excepción es reconocer un fracaso en el módulo en cuestión, y como consecuencia se provoca una excepción en el módulo invocante, que es quien deberá manejarla. Quien toma el hilo de ejecución es ahora el manejador de excepciones del módulo invocante, y éste se ocupa de resolver el problema. Si no existiera un manejador de excepciones en el mismo, se dispara la misma excepción al nivel anterior, y así sucesivamente. Si la excepción no fuera capturada en ningún nivel, llegaría a salir del programa, abortándolo.

Por ejemplo, cuando en Smalltalk escribimos:

```
ValorInvalido new signal.
```

Lo que estamos haciendo es crear una excepción de la clase *ValorInvalido* (notemos el mensaje *new* enviado a la clase) y luego lanzarla (mediante el mensaje *signal*) a quien haya invocado el método en donde está este fragmento de código. Por supuesto, en algún lado debe estar definida la clase *ValorInvalido*, pero ese es un problema que abordaremos un poco más adelante.

En Java, el equivalente del código de más arriba sería:

```
throw new ValorInvalido();
```

Donde el fragmento *new ValorInvalido()* provoca la creación del objeto y la acción *throw* hace que la excepción vaya a parar a quien invocó el método.

La idea es disparar una excepción de una clase diferente por cada tipo de anomalía. En general es la propia clase de excepción la que se usa para determinar de qué error se trata por parte de

quien la recibe. Por eso es especialmente importante elegir un buen nombre para la clase de excepción.

Captura de excepciones y manejadores

El manejador es una porción del código que va a decidir qué hacer con la excepción que recibe.

En Smalltalk, por ejemplo, se trata mediante un mensaje `on: do:` enviado a un bloque de código. Por ejemplo, en el código que sigue:

```
[celda colocarNumero:10]
  on: ValorInvalido
    do: [:e | Transcript show:'Problemas'].
```

Provoca que, si al ejecutarse el bloque de código `[celda colocarNumero: 10]` recibimos una excepción de tipo `ValorInvalido`, se ejecute el bloque de código que le pasamos en el parámetro `do`, en el cual la variable `e` va a contener una referencia al objeto de excepción recibido. Luego de ello, el programa sigue con su flujo de ejecución.

En Java la semántica es bastante similar (aunque la sintaxis sea diferente). El siguiente código es equivalente al de más arriba en Smalltalk y el objeto de excepción también queda referenciado por la variable `e`:

```
try {
    celda.colocarNumero(10);
}
catch (ValorInvalido e) {
    System.out.println("Problemas");
}
```

Por supuesto, un manejador podría lanzar una excepción a su vez, como muestra el siguiente fragmento:

```
try {
    celda.colocarNumero(10);
}
catch (ValorInvalido e) {
    throw new IllegalArgumentException();
}
```

Restitución de recursos luego de la excepción

Veamos la siguiente situación en Java.

Hemos dicho que las excepciones interrumpen el flujo de ejecución normal. Si la excepción fue lanzada dentro de un bloque `try` y existe una cláusula `catch` a continuación que permita capturarla, la excepción se tratará en esta última cláusula; de no cumplirse alguno de esos dos supuestos, se interrumpirá el método en ejecución y la ejecución volverá al método que invocó al actual.

Pero, ¿qué ocurre si antes de provocarse la excepción se hubieran comprometido recursos que deben restituirse? Si el recurso comprometido fueran objetos creados con `new`, no hay inconveniente alguno, porque al lanzarse la excepción se pierden las referencias a esos objetos y el recolector de basura puede recogerlos. Sin embargo, si se hubiera comprometido una conexión de red o una sesión con una base de datos, no tendremos esa suerte.

Una primera solución al problema se muestra en el fragmento de código que sigue:

```
try {
    abrirSesionBD( );                      // comprometo recursos
    metodoConExcepciones( );                // posible excepción
    cerrarSesionBD( );                     // libero recursos si no hubo excepción
}
catch (IOException e) {
    tratarExcepcionIO();
    cerrarSesionBD( );                   // libero recursos
}
catch (SQLException e) {
    tratarExcepcionSQL();
    cerrarSesionBD( );                  // libero recursos
}
```

El problema con esta solución es la llamada recurrente al método que libera los recursos (*cerrarSesionBD*), lo cual, como siempre que se repite código, es una idea cuestionable.

Por suerte, en Java contamos con la cláusula *finally* para liberar recursos. De esta manera, el código anterior podría escribirse mejor así:

```
try {
    abrirSesionBD( );                      // comprometo recursos
    metodoConExcepciones( );                // posible excepción
}
catch (IOException e) {
    tratarExcepcionIO();
}
catch (SQLException e) {
    tratarExcepcionSQL();
}
finally {
    cerrarSesionBD( );                   // libero recursos
}
```

En Smalltalk no hay una construcción similar al *finally* de Java, pero puede implementarse.

Excepciones derivadas

Por supuesto, las clases de excepciones son como cualquier otra de POO. Por lo tanto, podemos heredar de clases de excepciones. Pero, ¿cuál sería la utilidad de hacer que una clase de excepción derive de otra?

Piense un poco antes de seguir leyendo...

Hay dos usos posibles y complementarios:

- Al definir la captura de una excepción de una clase, vamos a capturar también excepciones instancias de clases derivadas. Esto es una consecuencia directa de la herencia, por definición.
- Nos permite manejar una jerarquía rica de tipos de problemas.

Como las excepciones se evalúan en orden, es un error poner un manejador de excepciones de

una clase derivada luego del manejador de algún ancestro, pues nunca se lo invocará.

Excepciones en lenguajes sin verificación previa a la ejecución: el caso de Smalltalk

Smalltalk nos provee una jerarquía de excepciones según el diagrama de la figura 9.1.

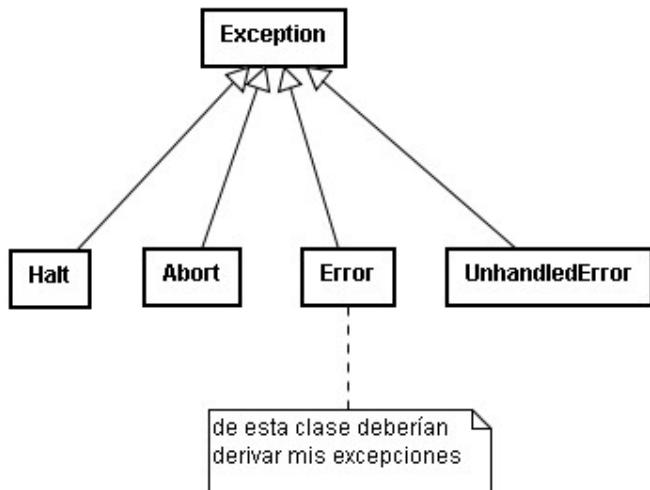


Figura 9.1 Clases de excepciones en Smalltalk

La documentación de Smalltalk nos dice que, al crear clases de excepción, lo hagamos derivando de *Error*, ya que las demás clases son para problemas del entorno.

Por lo tanto, basta una línea como la que sigue para definir una excepción:

```
Error subclass: ValorInvalido
```

Por supuesto, cualquier hija de una clase de excepción es también una clase de excepción. Por ejemplo:

```
ValorInvalido subclass: ValorNegativo
```

Declara una nueva clase de excepción, llamada *ValorNegativo*, que es derivada de *ValorInvalido*.

Excepciones en lenguajes de verificación estática y el caso extremo de Java

En Java las cosas son más complicadas. Como en Smalltalk, hay una jerarquía de excepciones definida en la plataforma, y también una recomendación: las excepciones que defina el programador deben derivar de *Exception* o alguna derivada de ésta.

La figura 9.2 muestra la jerarquía de excepciones en Java.

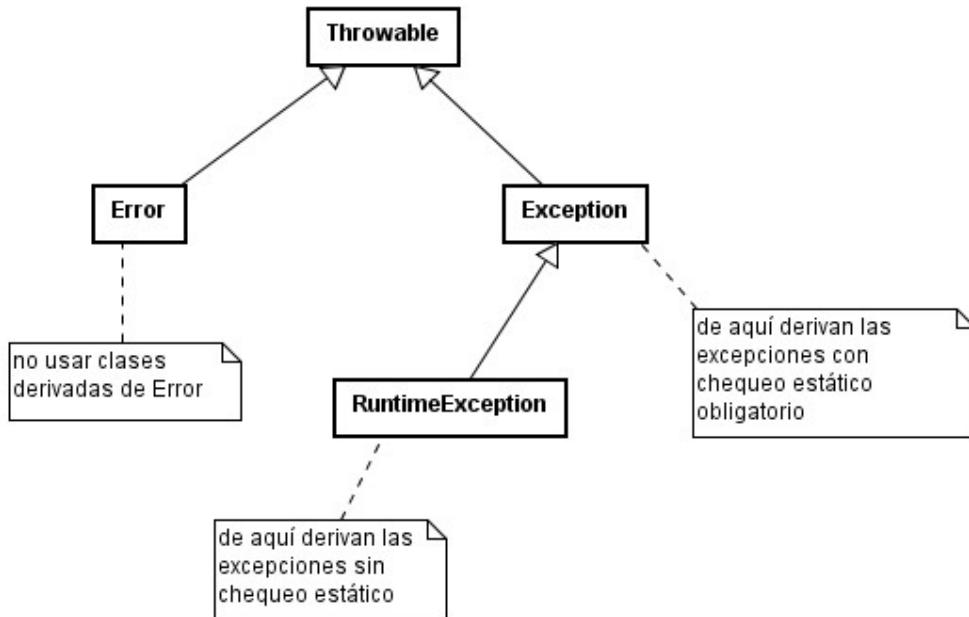


Figura 9.2 Clases de excepciones en Java

Las excepciones de clases descendientes de *Error* no se deberían capturar, pues indican problemas muy serios e irrecuperables. Tampoco podremos capturarlas cuando usamos la cláusula *catch* para una *Exception* genérica, pues no derivan de esa clase. Lo único que funciona con ellas es la cláusula *finally*.

Una primera particularidad de Java es que debemos tener un especial cuidado con las excepciones lanzadas o capturadas por constructores, pues un objeto no se crea realmente hasta que no finaliza correctamente su constructor. Esto hace que un constructor pueda haber comprometido recursos que luego no se utilicen por causa de la excepción.

Pero hay una diferencia fundamental respecto de Smalltalk y de muchos otros lenguajes, que hace que Java sea más seguro, y más molesto a la vez, en su tratamiento de las excepciones. Veamos.

Hay dos situaciones: la de las clases derivadas de *RuntimeException* y la de las derivadas directamente de *Exception*. El primer caso es idéntico a Smalltalk. Por eso, en nuestros ejemplos en Java hasta ahora, siempre declaramos que nuestras clases de excepción derivaban de *RuntimeException*.

Ahora bien, una excepción derivada de *Exception* tiene un significado diferente: no puede ser ignorada, ya que el compilador va a chequear que estemos tratándola.

Como la afirmación anterior puede ser un poco difícil de comprender veámoslo experimentalmente. Tomemos el ejemplo de aplicación bancaria y cambiemos la clase *SaldoInsuficiente*. Recordemos que estaba declarada así:

```
public class SaldoInsuficiente extends RuntimeException { }
```

Y cambiémosla para que herede directamente de *Exception*:

```
public class SaldoInsuficiente extends Exception { }
```

¿Qué pasó ahora? ¡El proyecto ya no compila! Eclipse marca un error en la clase *Cuenta*.

Miremos un poco el código y las afirmaciones anteriores aquí arriba a ver si entendemos qué

está ocurriendo.

Piense un poco antes de seguir leyendo...

Lo que pasa es que la clase *SaldoInsuficiente* se convirtió en un tipo de excepción que va a ser chequeada por el compilador (sólo las derivadas de *RuntimeException* no lo son). Hay dos cuestiones que se derivan de esto.

La primera es que el método que lanza la excepción debe declararlo explícitamente en su firma mediante la palabra *throws*:

```
public void extraer (int monto) throws SaldoInsuficiente {  
    if (monto <= 0)  
        throw new MontoInvalido();  
    if ( monto > disponible() )  
        throw new SaldoInsuficiente();  
    saldo -= monto;  
}
```

Con esto le decimos a cualquier otro método que invoque a *extraer*, que éste puede lanzar una instancia de *SaldoInsuficiente*. Pero esto no es todo: el compilador de Java nos obliga a capturar la excepción en el código que podría recibirla. De hecho, si miramos lo que nos dice Eclipse, vemos que ahora no compila la clase de prueba, y dice que porque no estamos manejando la excepción *SaldoInsuficiente*.

La forma más común de manejar esa posible excepción es armar un bloque *try-catch*, y eso debemos hacerlo en todas las pruebas que utilicen el método *extraer*. Por ejemplo, la pueba que antes era:

```
@Test  
public void alExtraerSaldoDebeDecrementarseEnMonto () {  
    Cuenta cuenta = new CajaAhorro (); // el saldo inicial es 0  
    cuenta.depositar(700);  
    cuenta.extraer(300);  
    Assert.assertEquals("La prueba NO pasó: el saldo no se decrementó  
correctamente al extraer", 400, cuenta.getSaldo());  
}
```

La escribimos así para que compile:

```
@Test  
public void alExtraerSaldoDebeDecrementarseEnMonto () {  
    Cuenta cuenta = new CajaAhorro (); // el saldo inicial es 0  
    cuenta.depositar(700);  
    try {  
        cuenta.extraer(300);  
    }  
    catch (SaldoInsuficiente e) {}  
    Assert.assertEquals("La prueba NO pasó: el saldo no se decrementó  
correctamente al extraer", 400, cuenta.getSaldo());  
}
```

Con ese bloque, la excepción ya se considera manejada. Por supuesto, tal vez no sea una buena práctica dejar vacío el bloque *catch*, ya que simplemente estamos capturando una excepción y no haciendo nada.

En general, un bloque *catch* tendría este aspecto:

```
catch (SaldoInsuficiente e) {  
    // acá va el código de manejo de la excepción  
}
```

No obstante, por tratarse de una prueba – y sólo por eso – podemos dejarlo como estaba: vacío.

Lo mismo deberíamos hacer en todos los métodos que invoquen a *extraer*.

¿Cuál es el sentido que le han visto los diseñadores de Java a las excepciones chequeadas?

Piense un poco antes de seguir leyendo...

La obligación de tratar las excepciones de los métodos invocados tiene dos ventajas:

- Hace a los programas más robustos.
- Es el compilador el que nos avisa que no estamos capturando la excepción, por lo que no podemos olvidarnos y el problema nunca llega al tiempo de ejecución.

En definitiva, es una decisión en línea con la pretensión de que el compilador de Java verifique todo lo que pueda antes de la ejecución.

Pero también tiene la desventaja de hacer muy molesto el trabajo con ellas, como pudimos observar en la necesidad de agregar bloques *try-catch* en lugares donde no íbamos a hacer nada en caso de excepción. Por eso hay lenguajes que – si bien hacen verificación de tipos en tiempo de ejecución – no obligan al compilador a asegurarse la existencia de un manejador de excepciones, como es el caso de C#.

Atributos y métodos en excepciones creadas por el programador

La idea de que el programador cree sus propias clases de excepciones pretende indicar con precisión el tipo particular de error que un método puede elevar.

Pero como una excepción es un objeto, podemos aprovechar para almacenar datos en los atributos del objeto, que puedan ser utilizados por el contexto invocante como información adicional, que ayuden a encontrar con mayor facilidad el problema exacto.

Por ejemplo, si la excepción *SaldoInsuficiente* la declaramos así:

```
public class SaldoInsuficiente extends Exception {  
  
    private int monto;  
    private int saldo;  
  
    public SaldoInsuficiente (int monto, int saldo) {  
        this.monto = monto;  
        this.saldo = saldo;  
    }  
  
    public int getMonto() {  
        return this.monto;  
    }  
  
    public int getSaldo() {  
        return this.saldo;  
    }  
}
```

```
    }
}
```

Al lanzarla en el *extraer* de *Cuenta* deberíamos hacer:

```
if ( monto > disponible() )
    throw new SaldoInsuficiente(monto, this.saldo);
```

Y luego, al capturarla, podríamos usar alguno de los datos internos. Por ejemplo:

```
try {
    cuenta.extraer(300);
}
catch (SaldoInsuficiente e) {
    System.out.print("No se pudo extraer. El monto pedido es: ");
    System.out.print(e.getMonto());
    System.out.print(" y el saldo era: ");
    System.out.print(e.getSaldo());
    System.out.println();
}
```

¿Es esta una buena práctica? En principio podría ser, pero digamos que no es muy habitual. Lo más corriente es que los programadores clientes sólo usen la clase de la excepción para saber de qué problema se trata, por lo que esta práctica sofisticada no tiene demasiada aplicación práctica.

Es por eso que la mayoría de las clases de excepciones simplemente se definen vacías, sin atributos ni métodos, como esta excepción típica:

```
public class SaldoInsuficiente extends Exception { }
```

Observemos que lo único que la caracteriza como una excepción es que hereda de *Exception* o una derivada de ella.

De todas maneras, lo mejor es elegir como clase ancestro alguna más cercana en significado a lo que estamos haciendo, y que la incluya, de modo de respetar la relación “es un”, el principio básico de la herencia.

Pasando al lado de la clase

Comportamiento y estado de las clases

Empezamos el libro diciendo que la base de la POO es el concepto de objeto, e incluso explicamos que hay lenguajes que permiten definir todo a nivel de objeto. Son los objetos los que tienen comportamiento, identidad y la posibilidad de almacenar estado. Dijimos también que hay lenguajes que definen a la clase como el tipo de los objetos, y definen en la clase al comportamiento y la estructura en la que se almacena el estado. Pero la clase es una decisión de implementación.

Ahora bien: ¿puede una clase almacenar estado, no en sus objetos, sino en la clase como un todo? ¿Puede definirse comportamiento a nivel de la clase? En algunos lenguajes, sí.

Smalltalk, con su noción de “todo es un objeto” hace que las clases sean objetos. Por lo tanto, se pueden definir estados y comportamientos para el objeto clase. En Java, donde la clase es un concepto independiente, no es tan directo. Pero en ambos se puede.

A veces tiene sentido definir atributos que almacenen estado para la clase en sí misma. Y

también métodos que permitan operar sobre la clase como conjunto, y no sobre cada uno de sus métodos.

Eso es lo que veremos a continuación.

Métodos de clase

A ver, primero: ¿qué sentido tendría enviar un mensaje a una clase en vez de un objeto? La respuesta rápida es: no mucho, en POO son los objetos los que reciben mensajes.

Pero hay un caso que venimos viendo en Smalltalk desde el principio del libro. Cuando escribimos:

```
celda := Celda new.
```

Lo que estamos haciendo es enviar el mensaje *new* a la clase *Celda* (o al objeto *Celda*, que resulta ser una clase). Si eso es así es porque *new* es un mensaje enviado a la clase, y la implementación de la respuesta a ese mensaje es un **método de clase**.

Definición: método de clase

Un método de clase es un método cuyo fin es implementar la respuesta de un mensaje enviado a la clase.

Los usos de los métodos de clase, como dijimos recién, son escasos, pero a veces resultan necesarios.

Para definir un método de clase en Pharo, se tilda el ítem “Class side” y automáticamente nos aparece la plantilla de creación de métodos, pero con el encabezado “<Nombre de clase> class”, como muestra la figura 9.3.

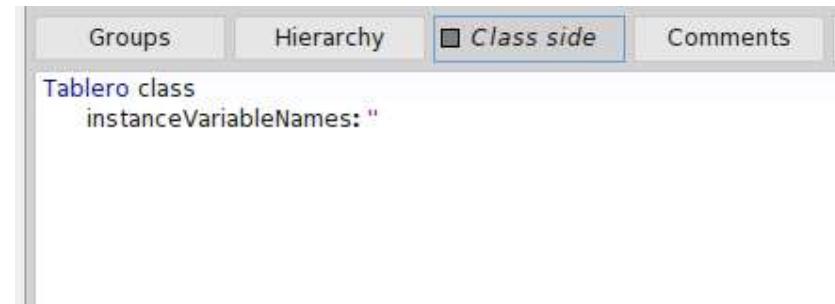


Figura 9.3 Métodos de clase en Pharo

En Java, los métodos de clase se conocen como métodos estáticos, y su declaración se hace mediante el agregado del modificador *static*, como en el ejemplo:

```
public static void metodoDeClase (int param) { ... }
```

Como los métodos de clase sólo actúan sobre la clase, y no sobre una instancia determinada, sólo pueden acceder a métodos y atributos de clase (ver más adelante). Tampoco tienen acceso a la referencia *this*.

Inicialización de objetos

En el capítulo de diseño por contrato introdujimos el concepto de invariante, como condiciones que debe cumplir un objeto durante toda su existencia. Mostramos también que eso afecta especialmente a los constructores, que deben crear objetos que cumplan su invariante desde el momento de su creación. Y vimos asimismo que en Smalltalk la solución no es del todo

satisfactoria. Repasemos un poco.

Si en la clase *Cuenta* tenemos como invariante que todo objeto debe tener un saldo mayor o igual que cero, y decidimos por lo tanto que toda instancia se cree con un saldo igual a cero, no nos sirve simplemente escribir:

```
cuenta := Cuenta new.
```

Pues en ese caso, el atributo *saldo* va a comenzar valiendo *nil* y no 0. Por suerte, Smalltalk provee un método *initialize*, definido en la clase *Object*, que se invoca desde *new* y que podemos redefinir en nuestra clase, lo cual funciona adecuadamente gracias al polimorfismo. Por ejemplo, si definimos:

```
Cuenta >> initialize  
    saldo := 0.
```

Habremos asegurado que toda instancia de *Cuenta* se cree con saldo 0.

Pero supongamos que tenemos un invariante más, que nos dice que el titular de una cuenta no puede quedar en *nil* ni ser la cadena de caracteres vacía, sino que debe asignarse en el momento de su creación.

Para esos casos hemos sugerido definir un método inicializador, como por ejemplo:

```
Cuenta >> inicializarConTitular: t  
    titular := t.
```

Ese método, se invocará al enviar un mensaje como:

```
c := Cuenta new inicializarConTitular: 'Carlos'.
```

Dado que el modo habitual de inicialización en Smalltalk es el método de clase *new*, hay personas que opinan que es mejor inicializar siempre con un método de clase. Para ello, definen un método de clase como el que sigue:

```
Cuenta class >> inicializarConTitular: t  
    | c |  
    c := Cuenta new.  
    c titular := t.  
    ^c.
```

Y la invocación la hacen:

```
c := Cuenta inicializarConTitular: 'Carlos'.
```

Por favor, asegúrese de entender la diferencia entre una y otra manera de inicializar. Ambas funcionan igual, pero el modo de invocación cambió, pues en el primer caso se usa el *new* seguido de un método de instancia, mientras que en el segundo no se usa *new* y se crea mediante un método de clase.

Piense un poco antes de seguir leyendo...

Personalmente, opino que es preferible usar métodos de instancia, como venimos haciendo desde el principio y vamos a seguir haciendo. Pero es una cuestión debatible...

En Java, no hay ninguno de estos problemas: toda clase tiene su constructor, que eventualmente puede tener parámetros, y si hay un constructor con parámetros deja de existir el constructor por defecto sin ellos. El caso anterior se resuelve simplemente así:

```
public Cuenta ( String titular ) {  
    this.saldo = 0;  
    this.titular = titular;
```

```
}
```

Atributos de clase

Tanto en Smalltalk como en Java se puede establecer estado para una clase y eso se hace mediante los **atributos de clase**.

Definición: atributo de clase

Un atributo de clase es un atributo cuyo fin es almacenar estado de la clase.

Como ya dijimos, dado que en Smalltalk las clases son objetos, esto no sería demasiado problema. En Pharo, por ejemplo, un atributo de clase se declara como los demás atributos, pero en la sección “class variable names” en vez de “instance variable names”.

En Java, un atributo de clase se define agregándole el modificador static, como en el fragmento que sigue:

```
private static int color;
```

Veamos un ejemplo. Supongamos que en nuestro ejemplo de aplicación bancaria queremos agregarle un número a cada cuenta, pero que este número no sea inicializado desde afuera al crear cada objeto, sino que comience en 1 y vaya incrementándose automáticamente por cada cuenta abierta. ¿Cómo haríamos algo así, si cada objeto debe ser autónomo de los demás objetos del sistema?

Piense un poco antes de seguir leyendo...

Aquí es donde podrían venir en ayuda nuestros atributos de clase. Por ejemplo, podríamos guardar en un atributo de la clase *Cuenta* el número de la última cuenta creada y a partir de allí ir incrementándolo con cada nueva cuenta que se crea. Notemos que, si bien el número de la cuenta se almacena junto con la propia instancia, el último número es un atributo de la clase.

Resolvámoslo en Java. ¿Qué es lo primero que debemos hacer? Claro: escribir una prueba.

La prueba es bastante elemental. Veamos: deberíamos agregar el siguiente método en la clase *PruebasCuentas*.

```
@Test
public void alCrearDosCuentasSucesivasSuNumeroDebeDiferirEn1 () {
    Cuenta c1 = new CajaAhorro ();
    Cuenta c2 = new CajaAhorro ();
    Assert.assertEquals ("La prueba NO pasó: se crearon dos cuentas
    sucesivas y los números NO difieren en 1",
        1, c2.getNumero() - c1.getNumero ());
}
```

A partir de allí definimos el método *getNumero* en la clase *Cuenta* para que compile:

```
public int getNumero () {
    return 0;
}
```

Ahora todo compila nuevamente, pero la prueba no pasa.

Tratemos de solucionar el problema. Agreguemos en la clase *Cuenta* dos atributos:

```
private static int ultimoNumero = 0;
private int numero;
```

Y cambiemos el constructor y `getNumero` para que queden así:

```
public Cuenta ( String titular ) {  
    ++ultimoNumero; // aquí estamos incrementando el atributo de clase  
    this.numero = ultimoNumero;  
    this.saldo = 0;  
    this.titular = titular;  
}  
  
public int getNumero() {  
    return this.numero;  
}
```

Si ahora corremos las pruebas, las mismas funcionan.

El diagrama de la figura 9.4 es el nuevo diagrama de clases de las cuentas bancarias. Nótese que los métodos y atributos de clase en UML se subrayan.

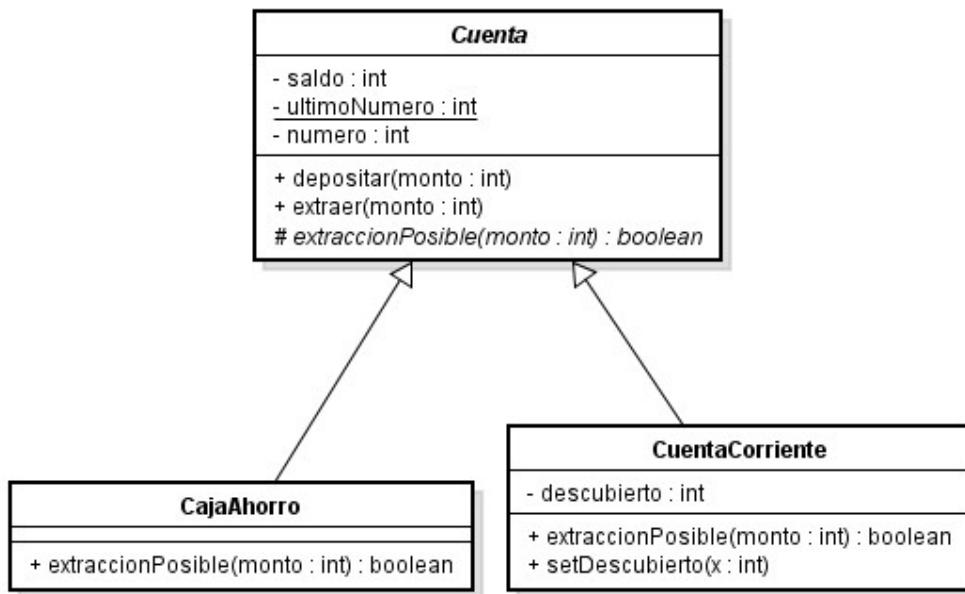


Figura 9.4 Métodos y atributos de clase

Genericidad

Antes de empezar: colecciones estilo Smalltalk

Smalltalk tiene una biblioteca de colecciones muy rica, que además aprovecha al máximo el polimorfismo y la herencia de raíz única. En efecto, los diseñadores del lenguaje, basándose en el hecho de que toda clase deriva de *Object*, implementaron colecciones cuyos elementos pueden ser de cualquier tipo.

Así, en la aplicación bancaria podríamos escribir:

```
listaCuentas := OrderedCollection new.
```

Y poner en esa colección cuentas de distintos tipos.

En el ejercicio del Sudoku podríamos hacer:

```
fila := OrderedCollection new.  
columna := OrderedCollection new.  
celda := OrderedCollection new.
```

Y en cada caso hacer una colección de celdas.

Esto resulta muy flexible para quienes debieron implementar las colecciones: como Smalltalk no hace verificación de tipos, podemos trabajar con las mismas clases de colecciones para celdas de Sudoku o cuentas bancarias.

El problema que podría haber es que una misma colección podría tener elementos de tipos bastante diferentes, y no hay forma de que el compilador nos advierta nada. Más aún, al extraer un elemento de la colección y pretender tratarlo como algo que no es, podrían ocurrir problemas en tiempo de ejecución. Pero eso va bien con la filosofía de Smalltalk, que nunca pretendió hacer esos controles.

Sin embargo, Java, en sus primeras versiones, siguió el mismo camino. Como todas las clases en Java tienen una clase ancestro por defecto, bastó con poner a *Object* como tipo de los elementos, y listo: todas las colecciones admiten cualquier tipo de datos.

Pero esta solución no se lleva bien con la pretensión de los diseñadores del lenguaje, que pretendieron crear un lenguaje con la mayor verificación de tipos posible en tiempo de compilación. Veamos:

```
List listaCuentas = new ArrayList( );  
CajaAhorro cuenta = new CajaAhorro();  
listaCuentas.add(cuenta);  
CajaAhorro otraReferencia = (CajaAhorro) (listaCuentas.get(0));
```

En efecto, en el arreglo *listaCuentas* puedo colocar elementos de cualquier tipo, y se almacenan como instancias de *Object*. Por lo tanto, al obtener un elemento, debo hacer un casteo de tipos para poder ver al objeto como lo que realmente es. Notemos que el casteo es inevitable, pues está indicando que el compilador no puede garantizar que no vaya a haber problemas en tiempo de ejecución.

Veamos, en cambio, este otro pedazo de código:

```
List listaCuentas = new ArrayList( );  
Date hoy = new Date( );  
listaCuentas.add(hoy);  
CajaAhorro otraReferencia = (CajaAhorro) (listaCuentas.get(0));
```

Claramente, estamos cometiendo un error, pues estamos pretendiendo ver como instancia de *CajaAhorro* lo que en realidad es una instancia de *Date*. Sin embargo, el compilador no va a detectar ningún problema, y éste va a saltar recién en tiempo de ejecución, con el lanzamiento de una *ClassCastException*.

Por lo tanto, hemos perdido la clásica seguridad de tipos que da el compilador en Java.

Lo que se buscó a partir de la versión 5 del lenguaje fue avanzar en el concepto de abstracción: pasar de la clásica abstracción de datos a la abstracción de tipos. Necesitamos – como autores de clases y métodos – poder escribir el mismo texto cuando podemos hacer la misma implementación de un concepto, aplicada a diferentes tipos de objetos.

Genericidad en lenguajes de verificación estática: el caso de Java

La idea es que haya clases y métodos que admitan tipos como parámetros. Habitualmente

llamamos clases genéricas y métodos genéricos a estas soluciones. Ya Ada implementó la idea de parametrización, luego adoptada también en el lenguaje C++. Eiffel planteó un cambio importante, con lo que Meyer denominó “genericidad restringida” [Meyer 1994], que luego pasó a la versión 5 de Java y la 2 de la plataforma .NET.

Lo que buscan estas soluciones, dicho brevemente, es permitir que los tipos puedan ser parámetros de clases, interfaces y métodos. En Java se coloca al tipo parámetro entre paréntesis angulares, como en *ArrayList<CajaAhorro>*, para indicar que esa colección sólo admite elementos que sean instancias de ese tipo parámetro.

Por ejemplo, más arriba mostramos el siguiente código, al “estilo Smalltalk”:

```
List listaCuentas = new ArrayList( );
CajaAhorro cuenta = new CajaAhorro();
// asigna una Cuenta a una variable de tipo Object:
listaCuentas.add(cuenta);
// casteo obligatorio para asignar lo que tengo en una variable Object
// a una variable de tipo CajaAhorro:
CajaAhorro otraReferencia = (CajaAhorro) (listaCuentas.get(0));
```

Ese mismo fragmento se podría escribir, con genericidad, así:

```
List<CajaAhorro> listaCuentas = new ArrayList<CajaAhorro> ( );
CajaAhorro cuenta = new CajaAhorro();
// en la asignación que sigue, el compilador verifica
// que el argumento sea una CajaAhorro:
listaCuentas.add(cuenta);
// no necesito el casteo porque sólo puede haber instancias de CajaAhorro:
CajaAhorro otraReferencia = listaCuentas.get(0);
```

Esta nueva versión es más robusta y más legible que la anterior, porque:

- No podemos agregar a *listaCuentas* nada que no sea una instancia de *CajaAhorro*, ya que el compilador nos lo impediría.
- El casteo no es necesario, pues el compilador puede asegurar que lo que se obtiene de *listaCuentas* es una instancia de *CajaAhorro*.

Java admite la genericidad con más de un tipo parámetro, como ocurre con la interfaz *Map*, que tiene dos parámetros *<K, V>*.

Implementación de clases y métodos genéricos

Por supuesto, podemos declarar nuestras propias clases genéricas. Por ejemplo, se podría implementar una lista circular genérica, así:

```
public class <E> ListaCircular {
    // definición de la clase
}
```

Para usar esta clase genérica como una lista de cajas de ahorro, deberíamos escribir, como es esperable:

```
ListaCircular < CajaAhorro > cuentas = new ListaCircular <CajaAhorro>();
```

También pueden implementarse métodos genéricos, como uno que tenga la firma siguiente:

```
public <E> void eliminarElemento (List<E> lista, int i) {
```

```
// código del método  
}
```

Para usar el método anterior, escribiríamos, por ejemplo:

```
x.eliminarElemento (listaConcreta, 6);
```

Y el compilador inferiría el tipo del parámetro *E* en base al tipo de los elementos de *listaConcreta*.

También podemos limitar los tipos admisibles como parámetro genérico. Por ejemplo, si escribimos:

```
public <T extends Comparable> void ordenar (T[] v) {  
    // código del método  
}
```

Cuando pretendamos usar este método, debemos utilizar un tipo argumento que implemente la interfaz *Comparable*. De no ser así, nuestro programa no compilará.

Sin embargo, las cosas se complican al ir avanzando. Supongamos que queremos un método de la aplicación bancaria que descuento 100 pesos a todas las cuentas del banco, de cualquier tipo. ¿Podría el código que sigue ser la solución?

```
public void descontarEnTodas (List<Cuenta> listaCuentas) {  
    for (Cuenta c : listaCuentas)  
        c.extraer(100);  
}
```

Piense un poco antes de seguir leyendo...

En principio pareciera que sí, y funciona si utilizamos una lista de instancias de *Cuenta* o sus descendientes sin problema. Pero nos llevaríamos una sorpresa al querer utilizarla pasándole una *List<CajaAhorro>*, aun cuando la clase *CajaAhorro* sea descendiente de *Cuenta*.

Para entender por qué ocurre esto, veamos el siguiente par de líneas:

```
List<CajaAhorro> listaCajasAhorro = new ArrayList<CajaAhorro>();  
List<Cuenta> listaCuentas = listaCajasAhorro;  
// ;error de compilación!
```

En realidad, lo que ocurre es que *List<CajaAhorro>* no es un caso particular de *List<Cuenta>* en el sentido del principio de sustitución. Un caso más claro de esto mismo se puede ver en el siguiente par de líneas:

```
listaCuentas.add(new Cuenta()); // hasta acá no hay problemas  
CajaAhorro c = listaCuentas.get(0); // ;no hay una instancia de CajaAhorro!
```

En realidad, nunca es cierto que los tipos *Clase<Derivada>* se puedan usar en vez de un tipo *Clase<Base>*, aun cuando *Derivada* sea descendiente de *Base*. Lo que se debe hacer en estos casos es utilizar un comodín, como en la siguiente versión de *descontarEnTodas*, que ahora sí funciona según lo que se espera:

```
public void descontarEnTodas (List<? extends Cuenta> listaCuentas) {  
    for (Cuenta c : listaCuentas)  
        c.extraer(100);  
}
```

En este caso, la leyenda “*? extends Cuenta*” significa que se admite un argumento *List* cuyo

tipo parámetro sea *Cuenta* o cualquier descendiente.

Podríamos seguir con los problemas de la genericidad en el marco de la orientación a objetos y las confusiones que genera. No obstante, la recomendación general sería usar genericidad solamente cuando es estrictamente necesaria y se conocen bien las implicancias de lo que estamos haciendo. Varios de los ejemplos anteriores se pueden resolver mejor sin usar genericidad.

Genericidad más allá de las colecciones

Hemos mostrado el uso de parámetros genéricos en el marco de las colecciones, que es su uso más común. Sin embargo, no está limitado a las colecciones. Un poco más adelante veremos una clase simple, *Class*, que tiene un parámetro genérico.

Información de tipos en tiempo de ejecución

Cómo se obtiene la información de tipos

Prácticamente en todos los lenguajes de programación se le puede preguntar a un objeto, en tiempo de ejecución, por la clase de la que es instancia.

Por ejemplo, si en Smalltalk escribimos:

```
[ x isKindOf: Cuenta ]
```

El resultado de la ejecución de ese bloque de código será un valor booleano que nos indica si el objeto referenciado por la variable *x* es instancia de la clase *Cuenta* o de alguna derivada.

Lo mismo significa el siguiente fragmento en Java:

```
( x instanceof Cuenta )
```

También se le puede preguntar a un objeto si es instancia de una clase exacta. En Smalltalk:

```
[ x class = CajaAhorro ]
```

O en Java:

```
( x.getClass == CajaAhorro.class )
```

Esta técnica se conoce como *RTTI*, que es el acrónimo en inglés para “información de tipos en tiempo de ejecución”⁵⁰. De todos modos, el uso de estas facilidades suele ser limitado. Veamos.

Por qué evitar la información de tipos en tiempo de ejecución

Venimos trabajando desde el principio sin necesitar preguntarle a un objeto de qué tipo es. Eso se debe a que – al menos hasta ahora – el polimorfismo venía resolviendo ese problema por nosotros: según la clase de la cual el objeto es instancia, la respuesta a un mensaje era acorde a esa clase, sin que nosotros tuviésemos que ocuparnos de nada.

Pero hay más: lo cierto es que escribir código para preguntar a un objeto, en tiempo de ejecución, de cuál clase es instancia, es una mala práctica. ¿Por qué?

El principal problema que suele causar el uso de RTTI es que compromete la extensibilidad de una manera que el polimorfismo resuelve automáticamente.

⁵⁰ Real-Time Type Information.

Por ejemplo, supongamos que – en vez de definir *disponible* en cada una de las clases hijas de *Cuenta*, lo hubiéramos definido directamente así en la clase madre:

```
protected int disponible ( ) {  
    if (this instanceof CajaAhorro)  
        return this.getSaldo();  
    else  
        return (this.getSaldo() +  
                ((CuentaCorriente)this).getDescubierto());  
}
```

La legibilidad empeoró, al aparecer ese feo indicador (*((CuentaCorriente)this)*), necesario para que el compilador entienda el mensaje *getDescubierto*, que sólo se aplica a instancias de *CuentaCorriente*.

Pero además, esto compromete la extensibilidad. ¿Qué pasaría si con el tiempo necesitamos un nuevo tipo de cuenta, cuya posibilidad de extracción se obtiene de una manera distinta?

Piense un poco antes de seguir leyendo...

En ese caso, deberíamos, no sólo agregar la nueva clase, sino modificar la clase *Cuenta* para prever esta nueva situación. Esto, además de molesto, viola el principio de diseño abierto-cerrado que vamos a ver en un capítulo posterior.

En la situación más habitual como programadores dentro del paradigma de objetos, tratamos de ignorar el tipo exacto de un objeto para poder abstraernos del mismo, y poder escribir código más genérico. El polimorfismo se ocupará del resto, trabajando a favor nuestro.

El abuso de uso de información de tipos en tiempo de ejecución es típico de programadores de otros paradigmas que se encuentran por primera vez con un lenguaje de POO. Precisamente por eso hemos dejado este tema hasta tan avanzado el libro: porque el uso del mismo es algo excepcional.

Modelo de objetos y clases en Smalltalk

Dijimos que en Smalltalk las clases son objetos. Pero si todo objeto es instancia de una clase: ¿de qué clases son instancias los objetos-clases? Más aún, vimos que tanto en Smalltalk como en Java todo objeto sabe, en tiempo de ejecución, de cuál clase es instancia: ¿cómo logra esa información?

Empecemos con el modelo de objetos de Smalltalk.

En Smalltalk cada clase es un objeto, y ese objeto es a su vez instancia de una **metaclase**.

Definición: metaclase

Una metaclase es una clase cuyas instancias son también clases.

Esas metaclases de Smalltalk son anónimas. Por lo tanto, para referirnos a la metaclase de *CuentaCorriente*, la nombraríamos como *CuentaCorriente class*. Las metaclases mantienen una jerarquía de herencia paralela a la jerarquía de clases del sistema. En la figura 9.5 vemos una jerarquía de clases y su jerarquía paralela de metaclases.

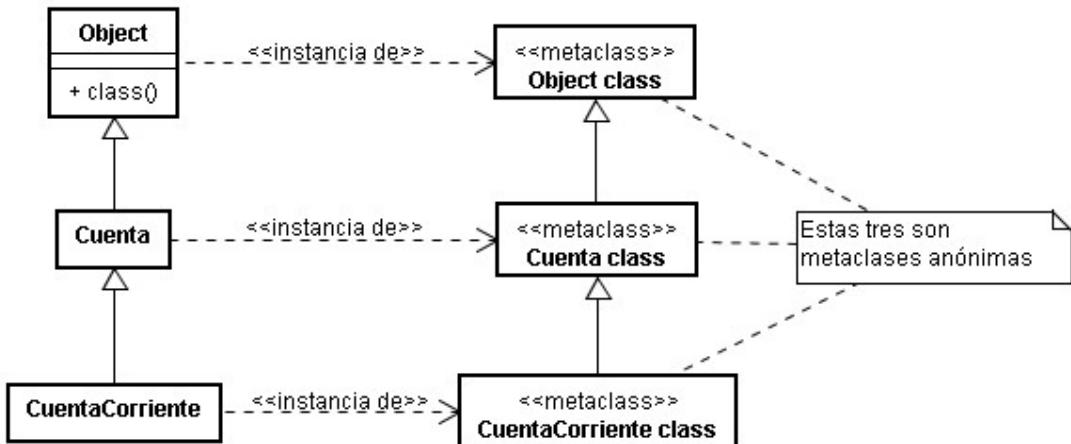


Figura 9.5 Jerarquía de clases y metaclases

Ahora bien, las metaclases son clases, todas ellas derivadas de la clase *Class*. Pero al ser clases son también objetos, y todas ellas son instancias de la clase *Metaclass*. Esto se ve en la figura 9.6.

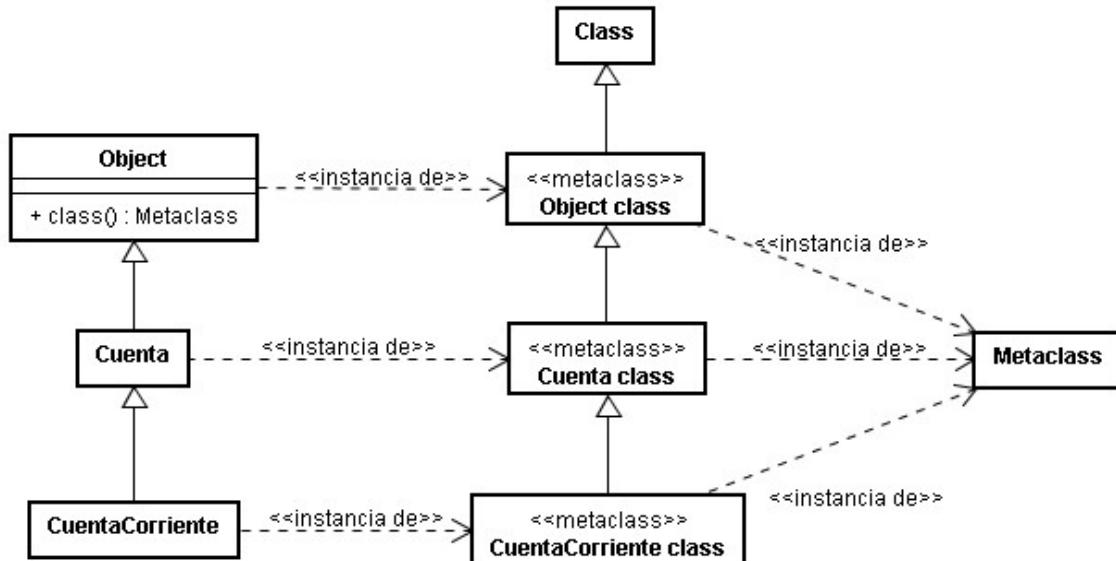


Figura 9.6 Jerarquía de clases y metaclases con las clases *Class* y *Metaclass*

Como en Smalltalk toda clase deriva de *Object*, la figura completa se muestra en la figura 9.7.

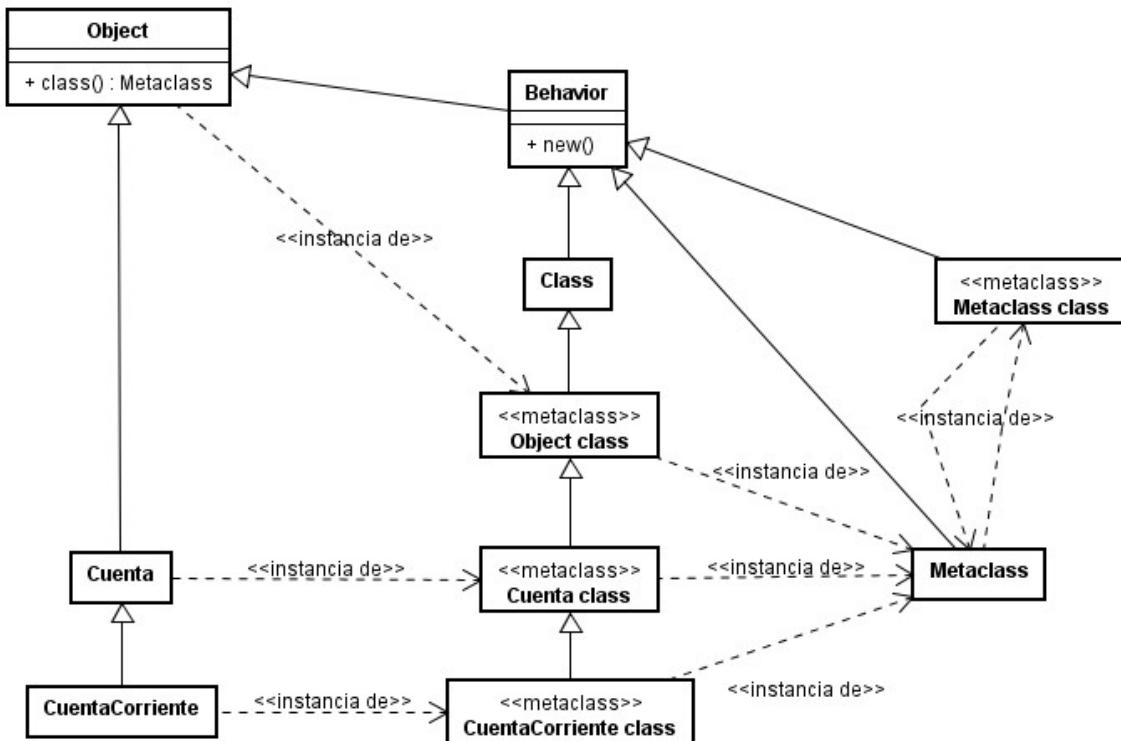


Figura 9.7 Jerarquía e instanciación completa de clases en Smalltalk

Modelo de objetos y clases en Java

En Java, el modelo es parecido al de Smalltalk, pero más simple, sobre todo por el hecho de que las clases no son objetos. Para ejemplificar, vamos a usar el ejemplo de nuestra aplicación bancaria, aunque sin la clase *CajaAhorro* y con la clase *Cuenta* como una clase concreta. Como objetos, vamos a usar una instancia de *Cuenta*, referenciada por la variable *cajaAhorro*, y una de *CuentaCorriente*, referenciada por *ctaCte*. La figura 9.8 muestra un diagrama de objetos y clases combinado con este escenario.

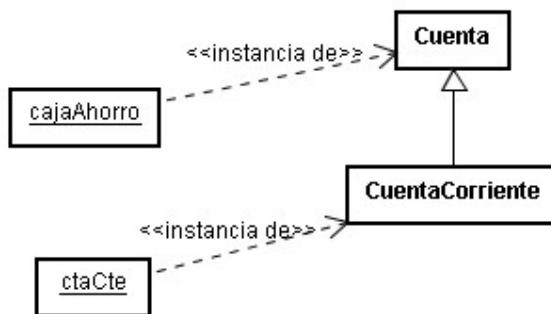


Figura 9.8 Objetos y clases de ejemplo

En primer lugar, hay una clase adicional, denominada *Class*, de forma tal que todo objeto tiene referencia a una instancia de *Class*: esa instancia es la que obtenemos mediante el método *getClass* que vimos más arriba. Como probablemente sea obvio, es la clase *Object* la que provee esta estructura. La figura 9.9 muestra estas relaciones, incluyendo dos instancias, una de la clase *Cuenta* y otra de *CuentaCorriente*.

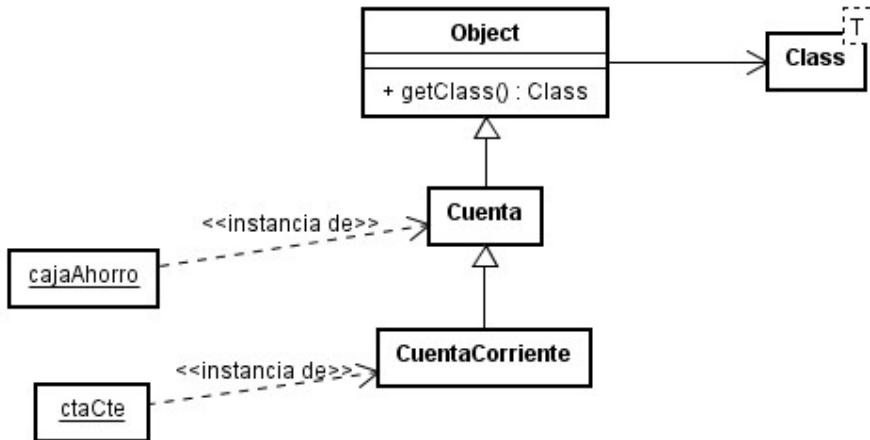


Figura 9.9 Clases *Object* y *Class* en Java

Ahora bien, ¿quiénes son las instancias de esa clase *Class* genérica y qué es ese extraño parámetro *T*? Veamos una sencilla definición, que se va a ir haciendo más compleja conforme vayamos avanzando.

Definición: *Class*

Las instancias de *Class* representan clases e interfaces de un programa que se está ejecutando.

Por ejemplo, el código que sigue imprime en consola el nombre de la clase de la cual es instancia el objeto parámetro (que, como sabemos, puede ser una instancia de cualquier clase):

```

void imprimirNombreClase (Object x) {
    String nombre = x.getClass().getName();
    System.out.println("El objeto es instancia de " + nombre);
}

```

El parámetro de *Class* es el tipo de la clase que modela cada instancia. Por ejemplo, una instancia de *CuentaCorriente* tiene una referencia a una instancia de *Class*<*CuentaCorriente*>.

La figura 9.10 pretende ilustrar un poco más. Aquí se ve un diagrama que combina clases y objetos, y en el cual las instancias de *Cuenta* y *CuentaCorriente* están relacionadas con instancias de *Class*. En la misma, se ve también que las propias instancias de *Class*, al ser objetos, tienen referencias a otra instancia de la clase *Class*.

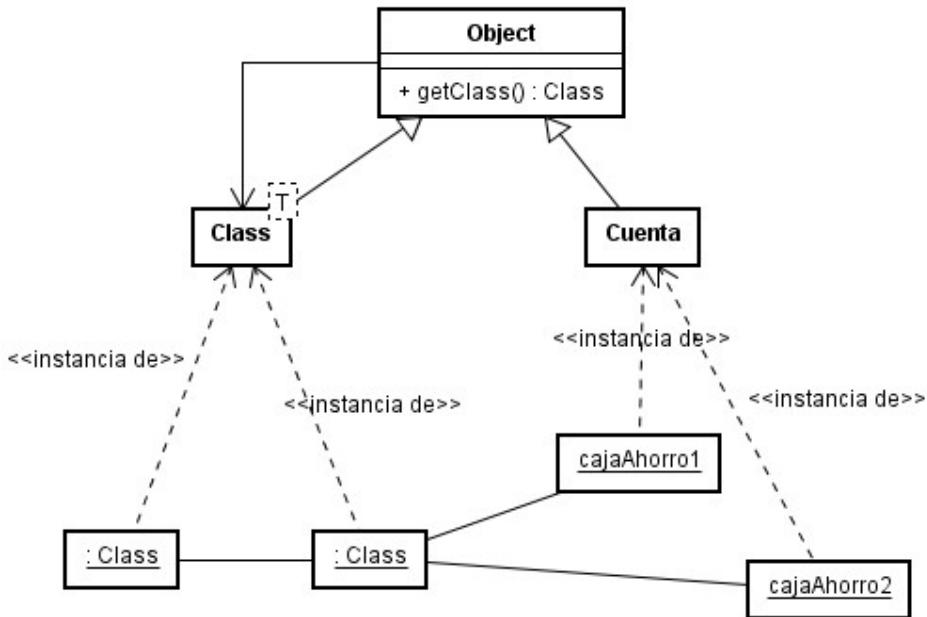


Figura 9.10 Relaciones entre objetos y entre clases en Java

Ahora bien, la clase *Class* no es una clase cualquiera. Por ejemplo, no tiene un constructor público, ya que sus instancias no pueden ser creadas por el programador. Como es de esperarse, es la propia JVM la que crea instancias de *Class*.

Reflexión y uso en Java

Dijimos que es una buena práctica evitar el uso de RTTI. Mostramos también cómo implementan Smalltalk y Java las relaciones entre objetos y clases. Pero, ¿qué uso concreto puede tener todo esto? El tema es que hay ocasiones en las que necesitamos conocer características particulares de una clase, e incluso darles uso.

Por ejemplo, ¿cómo funciona realmente SUnit? Dijimos en el capítulo respectivo que, cuando un usuario le pide a SUnit que ejecute las pruebas de la clase derivada de *TestCase* que elija, SUnit crea una instancia de esa clase y llama sucesivamente a los métodos que encuentra en ella que comienzan con la palabra *test*. Pero, ¿cómo encuentra esos métodos?

Piense un poco antes de seguir leyendo...

La respuesta está en una técnica denominada **reflexión**, que permite que le pidamos a una clase sus métodos (entre otras cosas), filtremos aquellos que empiezan con *test* y los invoquemos.

Definición: reflexión

Reflexión es la posibilidad que tiene un programa de examinar y modificar su estructura y comportamiento en tiempo de ejecución.

Algo parecido pasaba en JUnit 3. En JUnit 4, en vez de usar los nombres de los métodos para saber a cuáles invocar, se usan las anotaciones *@Test*, que también se pueden obtener por reflexión.

Vamos a ver reflexión sólo en Java, donde es un poco más sencilla y nos va a permitir entender los objetivos generales de esta técnica.

Para poder usar reflexión, cada instancia de la clase *Class* tiene referencias a instancias de las

clases *Method*, *Field*, *Constructor* y *Annotation*, entre otras. Como es relativamente obvio, así como dijimos que una instancia de *Class* representa a una clase o interfaz, una instancia de *Method* representa a un método, una instancia de *Field* a un atributo, una de *Constructor* a un constructor y una de *Annotation* a una anotación.

Por lo tanto, volviendo a los framework JUnit, si cada objeto *Class* tiene referencias a todos sus métodos, bastará con pedirle esas referencias, con ellas acceder a los objetos que representan métodos (instancias de *Method*) e invocarlos usando algún mensaje disponible en la clase *Method*.

La figura 9.11 muestra la clase *Class* y las clases que colaboran con ella para permitir la reflexión. Todas las clases del diagrama, a excepción de *Class* y *Object*, están en el paquete *java.lang.reflect*.

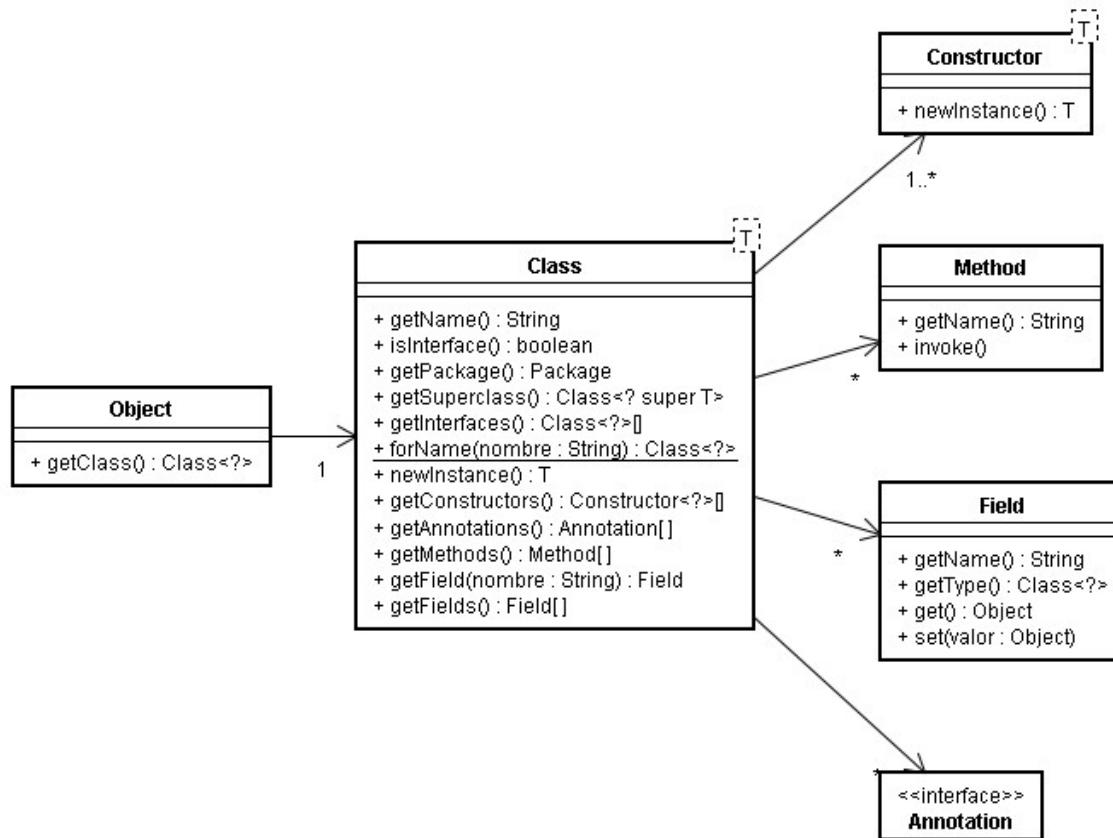


Figura 9.11 Clase *Class* y sus colaboradores

El diagrama de la figura 9.11 está muy simplificado: se han suprimido decenas de métodos, parámetros en algunos de ellos, excepciones lanzadas y clases e interfaces intermedias. No obstante, mirarlo un poco nos sirve para entender cómo funciona todo.

Otro ejemplo del uso de reflexión podría ser la pretensión de crear objetos en tiempo de ejecución en base a información que se maneja fuera del código. Veamos este método:

```

public class FabricaObjetos {
    ...
    public Object crearObjeto ( )
        throws ClassNotFoundException,
               InstantiationException,
               ...
  
```

```

        IllegalAccessException {
            String nombreClase = leerArchivo("configuracion.txt");
            Class<?> claseInstanciar = Class.forName(nombreClase);
            Object nuevo = claseInstanciar.newInstance();
            return nuevo;
        }
        ...
    }
}

```

Si luego hacemos:

```

FabricaObjetos f = new FabricaObjetos();
Object x = f.crearObjeto();

```

Vamos a obtener en *x* una instancia de una clase cuyo nombre viene en el archivo *configuracion.txt*.

La reflexión la suelen usar las bibliotecas estándar de Java en forma automática y es útil para implementar distintos tipos de frameworks, pero es muy raro que el programador deba utilizarla directamente si sólo está implementando cuestiones de negocio. Por eso sólo nos hemos detenido en sus aspectos esenciales y evitamos complejidades innecesarias.

Genericidad y RTTI en Java

¿Cómo implementa Java la genericidad que vimos más arriba? Veamos.

La JVM compila la clase genérica, generando una sola copia del código genérico en el archivo en código intermedio. Luego, la información de los tipos parámetro se usa solamente en tiempo de compilación. Por ejemplo, en el caso que ya vimos:

```

List<CajaAhorro> listaCuentas = new ArrayList<CajaAhorro>();
CajaAhorro cuenta = new CajaAhorro();
listaCuentas.add(cuenta);
CajaAhorro otraReferencia = listaCuentas.get(0);

```

En la primera línea, el compilador detecta que la variable *listaCuentas* es un *List*, cuyos elementos deben ser de tipo *CajaAhorro*. Por lo tanto, se asegura de que la llamada al método *add* (en la tercera línea) se haga con una variable de tipo *CajaAhorro* o alguna clase descendiente. A su vez, cuando se llama al método *get* (en la última línea), también tiene información de que – dado que sólo se le pudieron agregar instancias de *CajaAhorro* – lo que vamos a obtener va a ser una *CajaAhorro*, y no necesita casteo.

Sin embargo, una vez que termina la compilación de la aplicación, el compilador elimina toda información del tipo parámetro, convirtiendo nuestro *ArrayList<CajaAhorro>* en un simple *ArrayList* a secas. En el momento de llamar a *get*, incluye un casteo de tipos de *Object* a *CajaAhorro*. En definitiva, el tipo parámetro existe sólo hasta la compilación, y no llega al programa ejecutable.

Todo esto garantiza la ausencia de problemas en tiempo de ejecución, pues fue en tiempo de compilación que se aseguró que no se incluyeran otra cosa que instancias de *CajaAhorro* en el *ArrayList*.

La contra de esta implementación es que en tiempo de ejecución no hay manera de conocer nada de los tipos que figuraban como parámetros en el código fuente. Por lo tanto, si se le pregunta a la JVM la clase de una instancia que fue creada como *ArrayList<CajaAhorro>*, *ArrayList* o *ArrayList<Date>*, en todos los casos el método *getClass* devuelve una referencia

a la clase *ArrayList* a secas.

Tampoco se puede usar el operador *instanceof*:

```
// ¡error de compilación!
if (listaCuentas instanceof ArrayList<CajaAhorro>)
```

Lo mismo ocurriría si en un método tenemos un parámetro genérico T e intentamos hacer:

```
T x = (T) objeto;
```

Incluso los usos del tipo genérico *T* en código genérico se cambian por un tipo alto en la jerarquía, típicamente *Object*.

Por eso es común decir que – más que declarar una variable como de un tipo restringido – el programador expresa su “intención” de restringir el uso de esa variable, y el compilador responde asegurándole que puede hacerlo sin riesgos.

Ejercicios propuestos

Contenido todavía no disponible en esta versión del libro.

Recapitulación

El capítulo que estamos terminando nos ha servido para introducir varios temas relativamente heterogéneos, pero que habíamos dejado de lado para concentrarnos en POO en su forma más pura.

En el próximo se verán temas de Diseño Orientado a Objetos.

10. Diseño orientado a objetos

Contexto

En el resto de los capítulos hemos hablado de buenas prácticas metodológicas, de calidad de código y de diseño en varias oportunidades. Sin embargo, no hemos abordado el tema del diseño orientado a objetos en profundidad.

Si bien este es un libro de programación, la simbiosis creciente entre programación y diseño hace que nos debamos detener en algunas cuestiones, que son los temas de este capítulo. Veremos varios principios de diseño y algunos patrones de diseño clásicos. Cualquier profundización de estos temas requerirá consultar la bibliografía especializada, que la hay en abundancia.

Definición y objetivos del diseño

Llamamos **diseño** – al menos los informáticos – a la actividad del desarrollo de software que se centra en la manera en la que resolvemos problemas.

Si pensamos en la resolución de un problema cualquiera, éste suele tener cuatro partes:

- La identificación y delimitación del problema que queremos resolver: el “qué” o “los requisitos”.
- El establecimiento de una modalidad de resolución, y de los elementos que necesitaremos para resolverlo: el “cómo”, incluyendo el “con qué”.
- La resolución del mismo.
- La validación y la verificación contra los requisitos.

En el desarrollo de software, estas tres actividades se suelen denominar:

- Relevamiento y análisis (*qué* vamos a realizar).
- Diseño (*cómo* y *con qué* lo vamos a realizar).
- Construcción o implementación, que incluye la programación (la realización del producto esperado).
- Pruebas (verificación y validación).

El diseño, en consecuencia, es la actividad más ingenieril del desarrollo de software, la que debe elegir herramientas de todo tipo – lenguajes, tecnologías, estructuras de datos, algoritmos, y muchas otras más – para resolver el problema, y debe determinar cómo utilizarlas en conjunto. Como se ve, es una actividad muy vasta. Como toda tarea ingenieril, es importante basarse en experiencias previas – propias o ajenas – y en prácticas de probada efectividad en la resolución de problemas similares.

El objetivo de todo diseño es construir un sistema en código que satisfaga las especificaciones de los clientes, que sea fácil de adaptar y modificar, y comunique la intención del diseñador y el programador a otros potenciales lectores. En consecuencia, debe ser funcional, flexible, adaptable y legible. Debe lograr flexibilidad manteniendo al mínimo la complejidad. Según Martin [Martin 2003], “un sistema está bien diseñado si es fácil de comprender, fácil de cambiar y fácil de reutilizar”.

Muchos programadores, sobre todo si han trabajado en proyectos pequeños, no ven al diseño como una actividad separada de la programación. Al fin y al cabo, si quiero resolver el problema puedo partir del relevamiento y construir el software sin más preámbulo. ¿Será así realmente? Algunos opinan que la clave está en el tamaño del proyecto. Hagamos una analogía con la construcción.

Si una tarde de domingo me levanto de la siesta con ganas de hacer una cucha para el perro, voy a la ferretería, compro unas tablas, un serrucho y clavos; regreso a casa y me pongo a construirla. Como nunca he hecho una cucha, probablemente me salga mal. No obstante, no me frustraré por ello: puedo desarmarla y volver a construirla. Y si realmente arruiné todo, puedo volver a ir a la ferretería, comprar de nuevo lo que necesite, e intentarlo nuevamente. En alguno de estos ensayos, llegaré a construir algo pasable, o abandonaré el proyecto sin mayores consecuencias que una pequeña frustración.

Si, en cambio, quiero construir una torre de oficinas de veinte pisos, no me tomaré las cosas tan a la ligera. Deberé analizar alternativas de ubicación, me reuniré con otros pares más algún arquitecto para definir el aspecto del edificio y su funcionalidad, elegiré el tipo de materiales a emplear, dibujaré planos o haré que los dibujen, planificaré el proyecto, conseguiré financiación, contrataré personal especializado en distintas tareas, y finalmente dirigiré la construcción del mismo. ¿Por qué me tomo todo este trabajo? Simplemente, porque las tareas a realizar son tan complejas y costosas que no me puedo dar el lujo de que salgan mal y deba hacer todo de nuevo.

¿Es válida esta analogía? Para algunos sí, y para otros no. Lo que ocurre es que en el desarrollo de software, quienes hacen la tarea de construcción suelen ser técnicos y profesionales que a menudo están capacitados para las demás tareas también. Por eso, hay quienes han propuesto no hacer diseño previo a la construcción, sino dejar que el diseño evolucione con el código. Otros, más escépticos y formales, critican esta visión, diciendo que esto aplica muy bien a pequeños desarrollos – como en la analogía con la cucha del perro – pero que es impracticable en proyectos medianos y grandes.

La verdad es que ambos pueden tener razón. El diseño evolutivo podría ser riesgoso si la entropía del código aumenta hasta hacerlo inmantenible e inmanejable. En un proyecto chico esto puede no ser un problema, pero las cosas se complican en sistemas reales. Sin embargo, la solución de realizar todo el diseño por adelantado, choca a menudo con algunas dificultades propias del desarrollo de software, la más grave de las cuales es el cambio de requisitos del sistema una vez que lo estamos construyendo. En rigor de verdad, cuanto más grande es el proyecto en el que estamos embarcados, mayor es la probabilidad de que surjan cambios sobre la marcha. Y lo cierto es que varias décadas tratando de acotar el problema por el lado de impedir los cambios de requisitos, haciendo un concienzudo análisis previo, no han mostrado buenos resultados.

Pero hay algo que estamos dando por supuesto en nuestros razonamientos, que no tiene por qué ser así. Hemos mencionado al diseño y a la implementación como dos actividades distintas dentro del desarrollo de software. Pero que sean distintas no quiere decir, ni que deban realizarse separadamente.

De hecho, si bien los primeros esbozos de métodos de desarrollo planteaban que antes de comenzar a escribir la primera línea de código había que tener concluida la actividad de diseño completa, hace ya mucho tiempo que hemos debido flexibilizar esta cuestión. Un poco tímidamente al principio, se empezó a pensar en desarrollos iterativos y evolutivos, en los cuales se realiza un poco de diseño, al que le sigue un poco de programación, seguido de otro poco de diseño, y más programación, y así sucesivamente. Con el tiempo, esto se fue

formalizando, y surgieron métodos de desarrollo decididamente iterativos e incrementales.

Lo importante es entender que un diseño se desarrolla a lo largo del tiempo: no es algo que haya que hacer de entrada y para siempre.

Otra cuestión sujeta a debate es cuál es el resultado del diseño. Hay quienes piensan el Diseño (con mayúsculas) como un conjunto de diagramas muy formales. En realidad, esto tampoco tiene por qué ser tan dogmático: se pueden usar diagramas para diseñar, pero también se puede diseñar en y mediante el propio código. Todo depende de lo que queramos hacer.

Principios de diseño orientado a objetos

No repetir

El principio de diseño por excelencia es *no repetir*. Esta idea de no repetir se refiere, en primer lugar, al código idéntico. Si dos trozos de código son iguales, hay que refactorizar para que quede uno solo, probablemente en un método. También, al enunciar que no hay que repetir, nos referimos a código similar, que se puede colocar en un único método con los parámetros adecuados. De la misma manera, habría que tratar de no repetir patrones de agrupaciones de clases, generalizando lo más posible.

¿Para qué sirve no repetir? Un programador novato tal vez crea que lo que estamos recomendando es trabajar menos. Puede que en algunos casos sea así. Sin embargo, aun cuando ya el código repetido esté escrito, la recomendación es refactorizar y unificar. Claramente, en este caso estaríamos proponiendo trabajar más.

Es que la recomendación de no repetir tiene que ver, no con la primera escritura, sino con el mantenimiento futuro. Normalmente, el código igual o similar se refiere a los mismos requisitos, y por lo tanto suele ser necesario cambiarlo junto. Y no hay un error más común, ante un cambio de requisitos, que cambiar en un solo lugar lo que debiera haberse cambiado en dos o más. Si el código en cuestión es único, este peligro no existe.

Alta cohesión y bajo acoplamiento

En la programación modular existen dos características que hacen a la calidad de los módulos: la cohesión, que debe ser lo más alta posible, y el acoplamiento, que debe ser todo lo bajo que se pueda.

En POO, lograr alta cohesión y bajo acoplamiento a nivel de métodos es bastante sencillo. En efecto, los métodos son cohesivos casi por definición si seguimos las buenas prácticas, y el bajo acoplamiento suele ser fácil de lograr si se mantienen interfaces mínimas.

Sin embargo, habría que recordar que en POO la modularización se hace más bien mediante las clases, y ellas también deben cumplir los requisitos de acoplamiento y cohesión. Una clase tendrá alta cohesión cuando represente un tipo de objeto simple, o un agregado de tipos simples. Podemos garantizar una fuerte cohesión disminuyendo al mínimo las responsabilidades de una clase: si una clase tiene muchas responsabilidades probablemente haya que dividirla en dos o más. Del mismo modo, una clase tendrá bajo acoplamiento cuando tenga la menor interacción posible con otras clases. Esta dependencia significa que – si bien puede haber muchas clases que dependen de una – debería haber pocas dependencias hacia otras clases desde una sola. Para visualizar estas cuestiones, son de gran ayuda los diagramas de clases.

Importante:

Para lograr alta cohesión, nuestras clases deben tener pocas responsabilidades. Idealmente, una sola.

Para lograr bajo acoplamiento, cada clase debe depender de unas pocas clases más.

Otro módulo importante, aunque menos que la clase, es el paquete. Un paquete debe cumplir con estos mismos requisitos, sobre todo con el de acoplamiento débil, en el sentido de que debe tener vinculaciones mínimas con otros paquetes. En este caso, podemos ayudarnos con un diagrama de paquetes, que debido a que nos muestra dependencias entre conjuntos de clases, nos sirve para eliminar problemas de acoplamiento.

Única responsabilidad

Este principio fue enunciado por Meilir Page-Jones [PageJones 2000]. Según el mismo, cada clase debería tener una única responsabilidad. En general se lo alinea con el tema de cohesión, que implica que cada clase debe corresponder a una única abstracción.

Un corolario importante de este principio es el postulado de que cada clase tiene que tener una sola razón para cambiar.

Por ejemplo, recordemos lo que hicimos en nuestro ejemplo de cuentas bancarias cuando agregamos el número a las cuentas haciendo que el mismo se fuera incrementando de a uno. En ese caso, modificamos el constructor de *Cuenta* y le agregamos un atributo de clase a la misma. Pero al hacer eso estamos dándole a la clase *Cuenta* la responsabilidad de saber que los números de sus instancias se deben ir generando de esa manera, cuando en realidad conceptualmente una cuenta debería ser responsable solamente de mantener un número y un saldo, no de saber cómo se crean sus instancias. De hecho, si necesitásemos que algunas cuentas se fueran creando con números sucesivos y otras no, ese cambio haría que tuviésemos que cambiar la clase *Cuenta*, cuando en realidad el cambio es ajeno al concepto de cuenta.

Para separar esta responsabilidad de creación de cuentas, podríamos tener una clase *FabricaCuentas* con un método *crearCuenta*, que se ocupe de ello.

Empecemos por escribir la prueba para esta solución:

```
@Test
public void alCrearDosCuentasSucesivasSuNumeroDebeDiferirEn1 () {
    FabricaCuentas fabrica = new FabricaCuentas ();
    Cuenta c1 = fabrica.crearCuenta (TipoCuentas.CajaAhorro);
    Cuenta c2 = fabrica.crearCuenta (TipoCuentas.CuentaCorriente);
    Assert.assertEquals ("La prueba NO pasó: se crearon dos cuentas
    sucesivas y los números NO difieren en 1",
        1, c2.getNumero () - c1.getNumero ());
}
```

Para que compile, debemos crear la enumeración *TipoCuentas*:

```
package carlosFontela.aplicacionBancaria;
public enum TipoCuentas {
    CajaAhorro,
    CuentaCorriente
}
```

Ahora sí, escribamos la clase *FabricaCuentas*:

```
package carlosFontela.aplicacionBancaria;
```

```

public class FabricaCuentas {
    private int ultimoNumero = 0;

    public Cuenta crearCuenta (TipoCuentas tipo) {
        if (tipo == TipoCuentas.CajaAhorro)
            return new CajaAhorro(++ultimoNumero);
        else return new CuentaCorriente(++ultimoNumero);
    }
}

```

También debemos modificar los constructores de las tres clases de cuentas para que reciban el número como parámetro y eso a su vez hará que debamos cambiar las pruebas que utilizaban esos constructores.

Atención:

De todas maneras, persisten algunos problemas en esta solución. Notemos que podemos seguir creando cuentas por fuera de esta clase. También ese *if* es medio feo, además de complicar la evolución.

Encapsulamiento de lo que varía

Este principio a veces es presentado como una consecuencia del anterior, y lo cierto es que están muy relacionados.

Según el mismo, hay que lograr que los cambios afecten a partes del sistema sin afectar al todo. La idea, entonces, es aislar los cambios. Con el mismo criterio, se suelen aislar las partes del sistema que estén desarrollados en otras tecnologías, paradigmas, las dependencias del hardware y las interfaces con otros sistemas.

Es un principio fundamental en una gran cantidad de patrones de diseño.

Principio abierto-cerrado

Este principio, que enunciara Bertrand Meyer [Meyer 1988], expresa que las clases tienen que estar cerradas para modificación, pero abiertas para reutilización. Esto es, habría que tratar de no modificar clases existentes, ya probadas, estables y en producción. Pero esto no impide que dichas clases puedan reutilizarse desde otras.

El uso más natural de este principio es la herencia, que permite agregar datos y comportamiento a clases existentes, aun cuando éstas estén cerradas. El problema con la herencia es que el agregado de comportamiento queda determinado en forma estática, en tiempo de compilación.

Mediante la delegación tenemos otra forma de reutilización, en este caso en forma potencialmente dinámica, ya que podemos decidir asociar un objeto a otro en tiempo de ejecución.

Minimizar la herencia de implementación

Este principio, que a veces se enuncia como “favorecer la delegación sobre la herencia”, llama bastante la atención de los más novatos en POO.

¿Por qué queríramos favorecer la delegación sobre la herencia al reutilizar código? Lo que ocurre es que, si bien herencia y delegación sirven ambas para la reutilización, la herencia es estática, mientras que la delegación otorga mayor flexibilidad, al permitir diferir hasta el tiempo

de ejecución el tipo de reutilización que se hará.

Por ejemplo, en Java, la implementación de concurrencia con múltiples hilos se puede realizar de dos maneras.

La manera más simple es usando herencia. En este caso, es cuestión de crear un hilo instanciando una clase derivada de *Thread* y redefiniendo el método *run*. El diagrama de clases de la figura 10.1 muestra este escenario.

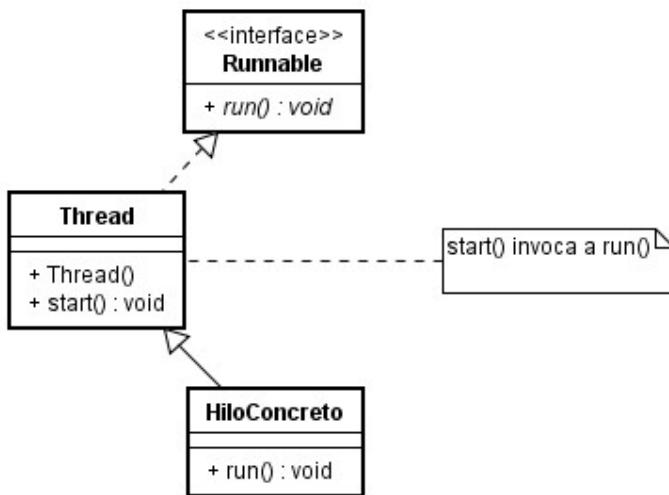


Figura 10.1 Concurrencia implementada por herencia

Cuando necesitemos lanzar un nuevo hilo de ejecución, teniendo en cuenta que el método *start* de *Thread* llama a *run* de *HiloConcreto* de manera polimorfa, lo haremos así:

```
new HiloConcreto().start();
```

Sin embargo, hay otra posibilidad, usando la propiedad de *Thread* de tener una referencia a una instancia de una clase que implemente *Runnable*. Esto se ve en el diagrama de clases de la figura 10.2.

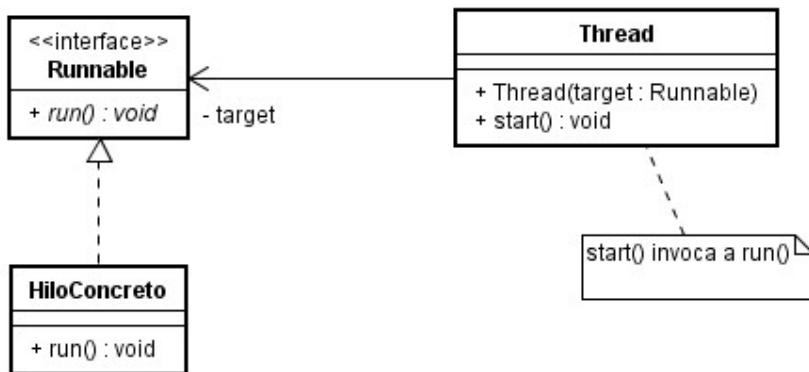


Figura 10.2 Concurrencia implementada por delegación

En este caso, el lanzamiento de un hilo nuevo se hace así:

```
new Thread(new HiloConcreto()).start();
```

¿Cuál de los dos mecanismos es preferible? En general, decimos que el segundo es más flexible que el primero, porque podemos cambiar la clase del hilo a lanzar de manera dinámica.

Por eso la herencia, a pesar de ser una herramienta importantísima de la POO, debe usarse con medida, y preferir siempre la delegación cuando sea una alternativa más o menos equivalente.

Atención:

Estas dos implementaciones de la reutilización de código corresponden a dos patrones de diseño que veremos un poco más adelante: Template Method y Strategy.

El próximo ítem, que habla del principio de sustitución, nos va a dar una nueva razón para evitar el uso indiscriminado de la herencia.

Sustitución de Liskov

Este principio, que lleva el nombre de quien lo enunció, Barbara Liskov [Liskov 1988], plantea que los subtipos deben ser sustituibles en todas partes por sus tipos base. La aplicación más elemental de este principio es el chequeo de la relación “es un”, es decir, que la subclase sea un subconjunto de la clase madre. La idea general es que las clases base no deben tener comportamientos que dependan de las clases derivadas, o incluso de su existencia.

Para entender un poco mejor por qué surgen confusiones al estudiar el principio de sustitución, tal vez sea bueno recordar que la famosa relación “es un” se refiere al comportamiento y no a la estructura. Quizá sería buena idea testear, además de la relación “es un”, la relación “se comporta como”.

Los corolarios del principio de sustitución son:

- Las precondiciones de un método no pueden ser más estrictas en una subclase de lo que son en su ancestro.
- Las postcondiciones de un método no pueden ser más laxas en una subclase de lo que son en su ancestro.
- Los invariantes de una clase deben ser al menos los mismos de la clase ancestro.
- Un método debe lanzar los mismos tipos de excepciones que en la clase ancestro, o a lo sumo excepciones derivadas de aquéllas.

Notemos que en nuestro ejemplo de aplicación bancaria, en el caso inicial en el que definimos el método *extraer* de modo diferente entre cuentas genéricas y cuentas corrientes, estábamos violando este principio, ya que la precondición respecto del límite del monto a extraer era más restrictiva en *CuentaCorriente* que en *Cuenta*. Las versiones posteriores solucionaron este problema.

Inversión de dependencia

Lo que este principio pretende es no depender de clases concretas volátiles. Esto es, no conviene que una clase herede o tenga una asociación hacia una clase concreta que tiene alta probabilidad de cambio.

Como en general las clases abstractas y las interfaces son más estables, conviene utilizarlas para herencia o delegaciones, en vez de las clases concretas.

Este principio a veces es enunciado como: “programe contra interfaces, no contra implementaciones”, aunque en este caso estamos poniendo el foco en el uso de un cliente.

Un ejemplo del uso de este principio en Java es el que sigue:

```
Collection c = new ArrayList( );
```

```
Iterator i = c.iterator();
```

Notemos que en ambos casos, los tipos de las variables son interfaces: *Collection* e *Iterator*. Incluso el uso de la clase *ArrayList* podría evitarse con el uso de algún patrón de diseño de creación de objetos como los que veremos luego.

Segregación de la interfaz

Este principio pretende que los clientes de una clase no dependan de métodos que no utilizan.

Así, si una clase tiene una referencia a, o hereda de, otra clase, de la cual sólo tiene sentido que utilice algunos de sus métodos, pero no todos, lo mejor sería separar la clase en cuestión en más de una, y colocar en una de ellas los métodos que utilice ese cliente.

Veámoslo con un ejemplo. Si tenemos un modelo de clases como el de la figura 10.3.

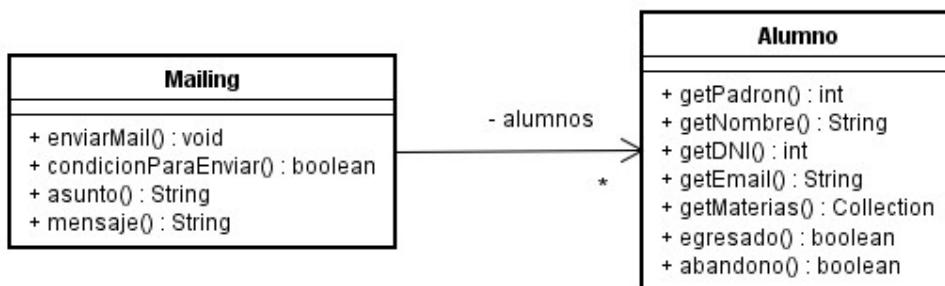


Figura 10.3 Para analizar dependencias de métodos

Y el código de la clase *Mailing* es como sigue:

```
public class Mailing {
    ...
    public void enviarMail ( ) {
        for (Alumno alumno : listaAlumnos)
            if ( condicionParaEnviar( ) )
                servicioMail.enviar(
                    alumno.getEmail( ),
                    alumno.getNombre( ),
                    asunto( ), mensaje( ) );
    }
}
```

Acá estamos violando claramente el principio. Sería mejor que el modelo fuese el de la figura 10.4.

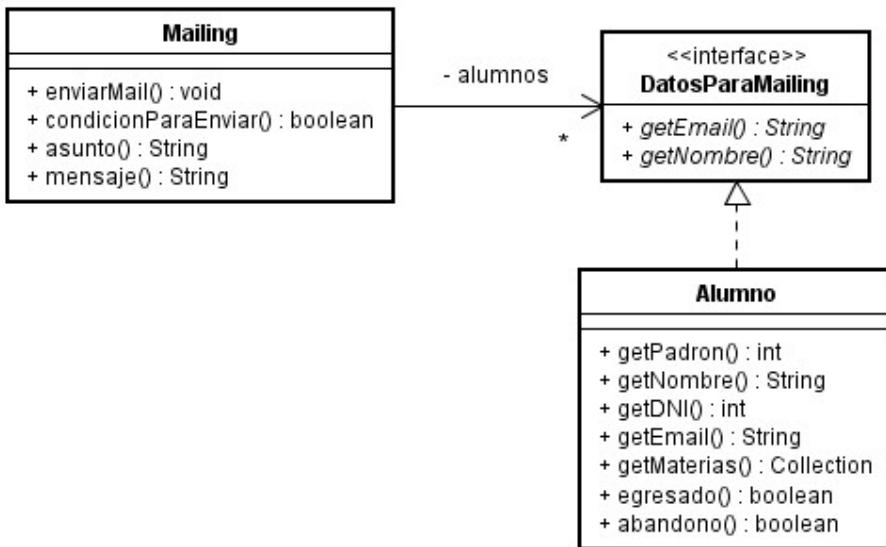


Figura 10.4 Interfaz necesaria ya segregada

Granularidad fina

La **granularidad** está definida por el tamaño de los objetos devueltos por los métodos de una clase.

Por ejemplo, si en nuestra clase *Cuenta* tuviésemos un método *getDatosCuenta*, que nos devolviera todos los datos agrupados de la cuenta, la granularidad es más gruesa que en el modo que venimos trabajando, con métodos *getNumero*, *getSaldo* y demás.

El principio de granularidad fina exige que la granularidad de las interfaces de clases sean lo más finas que se pueda, sin violar el debido encapsulamiento. Esto es así porque una granularidad más fina permite acceder a datos más desagregados.

Hay una objeción que se plantea a menudo contra este principio, y se refiere a los datos que deben viajar entre distintas aplicaciones. En efecto, si deseo invocar métodos sobre una aplicación remota, me interesa que la aplicación proveedora me devuelva un objeto de grano grueso mediante la llamada a un solo método, en vez de obligarme a llamar a varios métodos de grano fino, pues el consumo de ancho de banda es mucho mayor en el segundo caso.

La objeción es real, y si deseamos que los métodos de nuestra aplicación se puedan invocar desde otras, deberíamos prestarle atención. Sin embargo, hay dos cuestiones a tener en cuenta. La primera, que normalmente no deseamos exportar toda la funcionalidad hacia otras aplicaciones, sino sólo una parte. La segunda, que nada impide tener algunos métodos de grano grueso que estén implementados mediante varias llamadas a métodos de grano fino.

Por lo tanto, lo que se puede hacer es tener toda la funcionalidad implementada con granularidad fina, proveyendo adicionalmente algunos métodos de grano grueso para invocación externa, que deleguen a los métodos de grano fino. Cuando veamos el patrón Facade volveremos sobre este punto.

Recomendaciones para el diseño de paquetes

Cuando hablamos de diseño a nivel más alto que el de las clases y sus relaciones, el concepto que aflora naturalmente es el de paquete. Los paquetes permiten un nivel de abstracción más alto que las clases, de modo tal que es usual que, cuando deseamos tener una visión más

global de un sistema, pensemos en los paquetes que lo componen.

De allí que sea fundamental diseñar bien los paquetes y sus interacciones, para que ese nivel alto de abstracción sea más comprensible, y por lo tanto, más mantenible y escalable.

El principio del bajo acoplamiento, ya analizado, adquiere una fundamental relevancia en el diseño de paquetes. Decimos que hay dependencia entre paquetes cuando hay clases de un paquete que dependen de clases de otro paquete, sea por herencia, asociación o simple dependencia débil.

Algo de dependencia entre paquetes es obviamente inevitable, ya que las dependencias entre clases en muchos casos cruzan los límites de los paquetes. Lo que sí hay que lograr es que los cruces de dependencias entre paquetes sean la menor cantidad posible. El diagrama de paquetes de UML es una herramienta invaluable para analizar dependencias.

Otra recomendación habitual de diseño es construir las agrupaciones de clases de modo tal que sirvan para reutilizarse. Al fin y al cabo, los clientes reutilizan mucho más los grupos de clases que las clases aisladas. Por lo tanto, las clases que se prevé que se van a usar conjuntamente deberían colocarse en el mismo paquete. Asimismo, siguiendo en parte el principio de segregación de interfaz, conviene separar los paquetes por los clientes que los usan.

De la misma manera, si aplicamos algo similar al principio de única responsabilidad, las clases que cambian poco debieran ubicarse en paquetes separados de aquellas que cambian mucho.

También es deseable evitar las referencias cíclicas entre paquetes. Para esto contamos con algunas herramientas que nos ayudan a encontrar problemas, destacándose el diagrama de paquetes como herramienta gráfica.

Siguiendo el principio de inversión de dependencia, los paquetes de los cuales dependen muchos otros deben contener la mayor cantidad posible de clases abstractas e interfaces. Esto tiene como consecuencia que un paquete no dependa de otro menos estable. Otra forma de ver lo mismo es que cada paquete debe tener menor probabilidad de cambio que el paquete que depende de él.

Separación de incumbencias

Por suerte, hay una propuesta de diseño macro que es de aplicación casi permanente en las aplicaciones del mundo real, que llamamos **separación de incumbencias**.

La separación de incumbencias apunta a separar la aplicación por zonas de cambio. Así, si se espera que la interfaz de usuario cambie por razones diferentes, o más menudo, que la lógica de la aplicación, los componentes que implementen la interfaz de usuario deberían estar separados de los que implementen la lógica de negocio.

Las razones que pueden dar indicio de qué partes pueden evolucionar por separado – y en consecuencia tener que separarlas – son varias, pero destacan:

- Las tecnologías que se usen para cada parte sean o puedan ser diferentes.
- Las personas que desarrollan software se especialicen habitualmente en esas partes por separado.
- Las partes en cuestión tienen más de una posible forma de implementación, y es probable que se necesiten ambas.

- Las partes deban correr – hoy o en un futuro posible – en computadoras diferentes.

Siguiendo estas premisas, no es raro que en una aplicación puedan reconocerse los siguientes módulos, o al menos algunos de ellos:

- Lógica de negocio de la aplicación
- Presentación o interfaz de usuario
- Persistencia y acceso a datos
- Acceso de nuestra aplicación a servicios externos
- Acceso de otras aplicaciones a nuestros servicios
- Administración de seguridad

MVC, como veremos luego, es un patrón de separación de incumbencias.

Cuándo es malo un diseño

Una buena forma de recapitular sobre los principios de buen diseño es analizar qué cuestiones hacen malo a un diseño. Hay varias razones por las cuales un diseño puede ser malo. Lo que sigue son algunas señales, según [MartinUML]:

- Rigidez: dificultad de cambiar, porque cada vez que se modifica algo, esto provoca una secuencia interminable de cambios.
- Fragilidad: cada cambio provoca problemas en partes del código que no tienen que ver con el mismo.
- Se hace muy difícil descomponer el sistema en partes reutilizables.
- El ciclo editar-compilar-probar lleva mucho tiempo.
- Complejidad innecesaria: estructuras de código que no se necesitan, aunque puedan servir en el futuro.
- Repetición innecesaria: no se reutiliza, sino que se ha copiado partes de código en varios lugares.
- Opacidad: el programador se expresa – mediante el código – de formas ininteligibles.

Hay otros problemas típicos de diseño, que Martin Fowler [Fowler 1999] llama “malos olores”, que los buenos diseñadores enseguida detectan con su intuición. A continuación enumero algunos de los olores más usuales:

- Ciclos muy anidados, que se deberían convertir en métodos.
- Código duplicado, que causa modificaciones paralelas.
- Métodos complejos o muy largos.
- Clases con varias responsabilidades, o muy grandes.
- Abundancia de sentencias *switch* o *if* anidadas.
- Largas secuencias de llamadas sucesivas a métodos.
- Demasiados chequeos de referencias nulas.
- Clases sin comportamiento, que sólo tienen atributos y las propiedades que permiten

- acceder a ellos.
- Atributos no encapsulados.
- Métodos que usan más características de otras clases que de la suya propia.
- Uso de tipos primitivos o básicos para conceptos diversos.
- Comentarios que explican código difícil de leer.

Patrones de diseño

¿Qué es un patrón de diseño?

Un **patrón** es una solución no trivial a un problema en un determinado contexto. Para que sea útil, debe ser una solución probada a un problema que aparece con frecuencia.

En las disciplinas de desarrollo de software hay muchos niveles de patrones, según el grado de abstracción. Los patrones de despliegue – por ejemplo – especifican maneras de desplegar una aplicación en nodos de hardware. Los patrones de arquitectura definen relaciones entre paquetes o subsistemas. Los patrones de programación, o *idioms*, definen cuestiones de implementación de algoritmos y estructuras de datos. Como ejemplos, el algoritmo de burbujeo o la implementación de una búsqueda binaria en un determinado lenguaje, son patrones de programación.

Los **patrones de diseño** nos indican cómo utilizar clases y objetos de formas conocidas y estudiadas, de modo de adaptarlos a la resolución de parte de un problema o a un escenario particular. Es el concepto de reutilización llevado al diseño, presentado originalmente por Erich Gamma y otros tres autores [GoF 1994], conocidos como “la banda de los cuatro” (o GoF, por su sigla en inglés).

Un patrón de diseño se aplica a una sociedad de objetos y/o clases. De alguna manera, un patrón de diseño se resuelve mediante una colaboración, y consta tanto de aspectos estructurales como de comportamiento.

La idea es usar patrones que otros descubrieron, para aplicarlos en los futuros desarrollos, ya que la adopción de los mismos puede traer grandes ventajas. Entre ellas:

- Estructuras de diseño probadas previamente.
- Soluciones alternativas a las más frecuentes simples o ingenuas.
- Descripción de soluciones en base a combinaciones de patrones usuales.
- Fácil interpretación de colaboraciones ya conocidas.
- Facilidad de separar los aspectos que cambian de los que no cambian.
- Mayor nivel de abstracción.
- Introducen un lenguaje común para referirse a formas de construir software, elevando el nivel de abstracción de la conversación, lo que a su vez favorece la enseñanza, el aprendizaje y el trabajo en grupos.

En la misma línea, se suelen denominar **antipatrones** a los diseños que han arruinado proyectos. Los antipatrones se usan para saber qué es lo que no hay que hacer. Por eso, hay autores que han construido patrones haciendo lo contrario de los antipatrones.

Notemos que la idea de los patrones de diseño se viene aplicando desde siempre en otras

industrias. La industria de la construcción, por ejemplo, diseña escaleras y cajas de ascensor o distribuye sanitarios en un baño de tres o cuatro formas básicas. La idea es aplicar el mismo concepto en el desarrollo de software.

Como hay unos pocos patrones de uso general, no es difícil estudiarlos y conocerlos. A continuación, analizaremos algunos patrones más usuales.

Iterator

Los iteradores son objetos que recorren colecciones sin ser parte de las mismas. La idea de este patrón trabajar con un objeto que posea el conocimiento sobre cómo recorrer determinado tipo de colección, de forma tal que los clientes de este objeto no necesiten conocer la estructura interna de la colección.

Este patrón ha sido implementado en varios lenguajes de programación modernos. Está disponible para las colecciones de Smalltalk y Java. Sin embargo, no es un patrón complejo de implementar en cualquier lenguaje de POO. En la figura 10.5 se muestra el diagrama de clases típico.

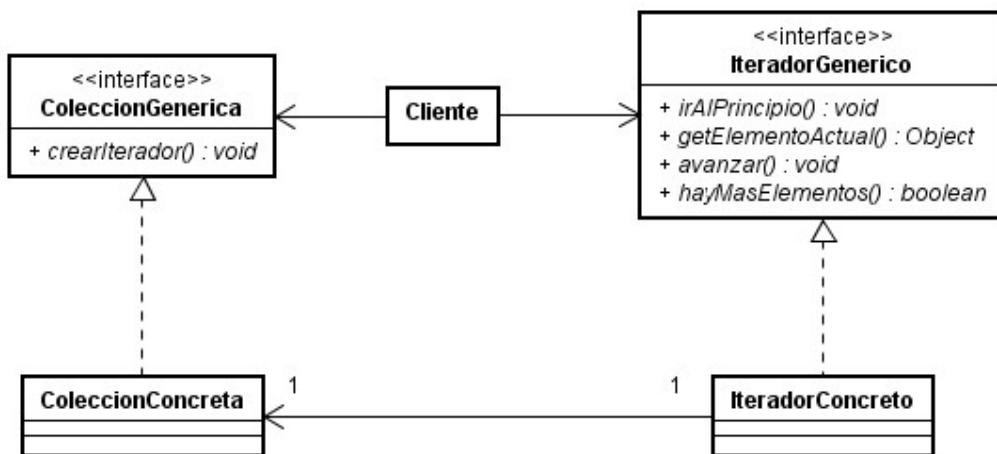


Figura 10.5 Diagrama de clases de una implementación de Iterator

Al implementar iteradores por fuera de las colecciones a recorrer, se mantiene en la colección solamente la responsabilidad de almacenar objetos, pero no la de saber cómo moverse a través de los mismos. De esta manera, la interfaz de la colección es más simple y consistente.

Los principios de diseño que satisface este patrón son:

- Única responsabilidad y alta cohesión, ya que tanto la colección como el iterador tienen un único comportamiento: almacenar objetos en el primer caso, y recorrer el agregado en el segundo.
- Inversión de dependencia, al utilizarse los iteradores por su interfaz.

Template Method

Este patrón lo venimos usando desde que vimos polimorfismo.

La idea del mismo es definir una clase abstracta, con un método plantilla o marco que contiene la lógica principal de un algoritmo, pero que delega en ciertos métodos – que se redefinen en clases descendientes – la implementación de cuestiones particulares. De este modo, la clase marco contiene la lógica general, y los clientes pueden implementar cuestiones particulares

creando clases derivadas de la misma.

En la figura 10.6 se muestra el diagrama de clases típico del patrón Template Method:

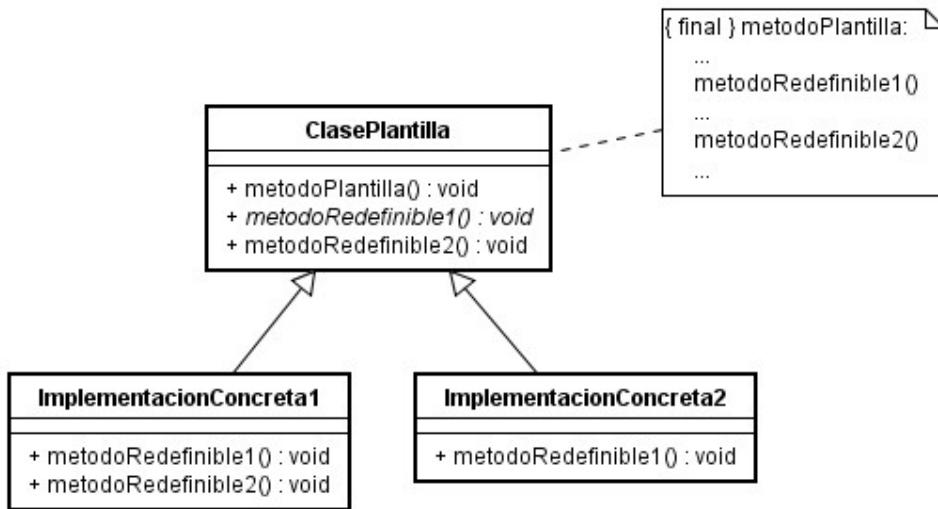


Figura 10.6 Diagrama de clases de una implementación de Template Method

La idea, entonces, es encapsular trozos de algoritmos en clases definidas por el cliente, respetando un marco definido en la clase plantilla. A veces a este patrón, o a la clase ancestro, se los denomina Application Framework, precisamente por esta condición de servir de marco general.

Un ejemplo de este patrón es la implementación por herencia de la concurrencia en Java, en la que la clase *Thread* es la clase plantilla, el método *start* hace de método plantilla y *run* es el método redefinible. La figura 10.7 muestra esto nuevamente.

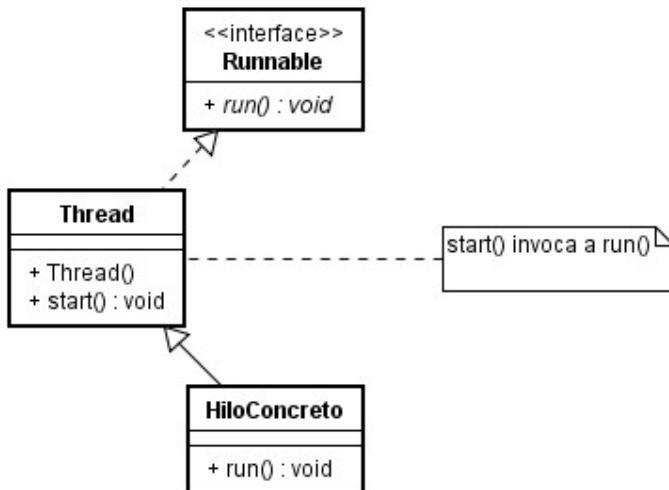


Figura 10.7 Concurrencia implementada con Template Method

Un buen uso del patrón exige que el método plantilla sea no redefinible. Si no fuera así, estaríamos tergiversando la intención del patrón, ya que permitiríamos un marco mutable, y ésa no es la idea.

Los que sí se deben poder redefinir son el resto de los métodos. Esto se puede lograr de dos maneras: o haciéndolos abstractos, obligando de esa manera a redefinirlos; o brindando una

implementación por defecto, o vacía, para que los clientes decidan si quieren o no redefinirlos. Lo que ocurre es que el método plantilla representa los invariantes del algoritmo, mientras que los métodos por él invocados son las partes que admiten variación.

A mayor granularidad del método marco – esto es, a mayor cantidad de métodos redefinibles que llame – mayor es la flexibilidad que dejamos a los clientes. Sin embargo, también es mayor la complejidad de uso, aun cuando definamos algunas implementaciones por defecto para simplificar su tarea.

Los principios de diseño que satisface este patrón son:

- Encapsulamiento de lo que varía, en este caso los métodos redefinibles.
- Abierto-cerrado, ya que se utiliza funcionalidad de la clase plantilla y se la modifica, pero sin modificarla.
- Alta o baja granularidad, según la conveniencia.

Factory Method

El patrón Factory Method pretende encapsular la creación de objetos de determinados tipos, todos descendientes de una misma clase abstracta o interfaz. De esta manera, un cliente puede obtener una referencia a un objeto del cual no conoce su clase, sino sólo una de sus interfaces posibles.

Por ejemplo, analicemos las siguientes líneas en Java:

```
Iterator i = agregado.iterator();
while (i.hasNext()) { ... i.next(); ... }
```

La verdad es que hay pocos fragmentos de código tan densos en conceptos complejos como éste.

Veamos. ¿Cuál es el tipo de la variable *agregado*? El objeto referenciado por *i*, ¿de qué clase es instancia? ¿Y el referenciado por *agregado*? ¿En qué clase está implementado el método *iterator*? ¿Y *next* y *hasNext*?

Lo único que sabemos por el momento, o podemos deducir, es:

- *agregado* es de tipo *Iterable* o algún derivado. Por lo tanto, el objeto referenciado por *agregado* es instancia de alguna clase que implemente *Iterable*, pero leyendo el fragmento anterior no podemos saber cuál es.
- *i* es de tipo *Iterator*. Por lo tanto, el objeto referenciado por *i* es instancia de alguna clase que implemente *Iterator*, pero no conocemos ninguna de esas clases ni tenemos forma de conocerlas.
- *iterator* es un método de la interfaz *Iterable*, que debe estar implementado en todas las clases que implementen dicha interfaz. Entre ellas, la clase de la cual sea instancia el objeto referenciado por *agregado*.
- *next* y *hasNext* son métodos de la interfaz *Iterator*, que deben estar implementados en todas las clases que implementen dicha interfaz. Entre ellas, la clase de la cual sea instancia el objeto referenciado por *i*.

Lo que ocurre es que el método *iterator* está encapsulando la creación de un objeto, instancia de una clase desconocida, que implementa la interfaz *Iterator*. Y lo encapsula de tal modo que, ni podemos crear objetos de esa clase mediante el operador *new*, ni tenemos posibilidad

de saber de qué clase se trata.

Esta es la idea del patrón Factory Method. Ahora bien, ¿qué ventaja tiene esto? La clave está en el análisis de tipos realizado más arriba.

En efecto, el par de líneas de código que analizamos antes se puede aplicar a cualquier agregado que sea instancia de una clase que implemente *Iterable*. No importa si agregado es un *ArrayList*, un *LinkedList*, un *TreeSet* o cualquier tipo de la plataforma Java o definido por el programador, siempre y cuando implemente *Iterable*. Esto nos permite cambiar la colección utilizada cuando lo deseemos, sin tener que cambiar el código en cuestión. Esto es en parte una ventaja del patrón Iterator, y en parte de Factory Method.

De hecho, si necesitásemos llamar al constructor del iterador que permite recorrer *ArrayList* cuando operamos con objetos *ArrayList*, y al constructor del iterador que permite recorrer *TreeSet* cuando operamos con objetos *TreeSet*, cada cambio de colección nos llevaría a un cambio en dos lugares: en la creación de la colección en sí, y en la creación del iterador.

Esta necesidad de dos cambios en el código para un mismo cambio de diseño no sólo es molesto, sino también peligroso, ya que podríamos introducir errores inadvertidos.

La implementación general del patrón Factory Method consiste en mantener una jerarquía de clases creadoras, paralela a la jerarquía de clases a crear, como muestra el diagrama de la figura 10.8.

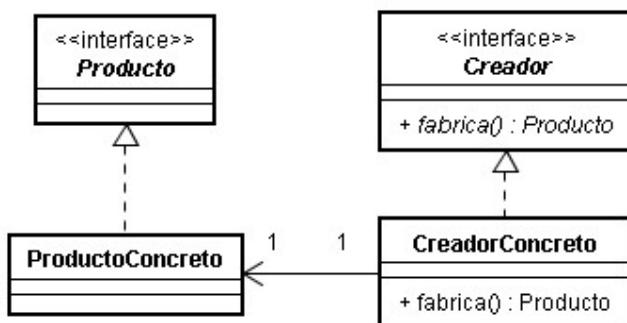


Figura 10.8 Implementación típica de Factory Method

Una forma alternativa de implementar Factory Method, es mediante reflexión, como hicimos al explicar ese concepto mediante la clase *FabricaObjetos*.

Los principios de diseño que satisface este patrón son:

- Inversión de dependencia, ya que los clientes sólo conocen la interfaz del objeto que crean.
- Bajo acoplamiento entre clientes y los objetos creados.

Observer

Éste es un patrón de gran utilidad en varios ambientes. Un ejemplo práctico son los ambientes operativos que manejan eventos. Otro es el patrón de arquitectura MVC. En general, es aplicable a cualquier desarrollo de sistemas en capas. Responde al paradigma suscriptor-productor. O sea, hay dos objetos: uno que es observado y otro u otros que observan, de modo que los segundos pueden cambiar su estado y comportamiento en función de cambios en el primero.

Veamos. La idea del patrón es manejar las situaciones en las cuales dos objetos – mutuamente

desacoplados – que necesitan actuar al unísono, de modo tal que, cuando uno cambie de estado, el otro sea notificado y pueda responder con cierto comportamiento preestablecido. Al primer objeto se lo llama observado o sujeto, y al segundo, observador. Habitualmente se admite que existan varios observadores por sujeto, sin que éste deba saberlo o varíe en algo su comportamiento por este motivo.

El paradigma típico representado por este patrón es el denominado de **subscripción y notificación**. El observador es quien se suscribe en el sujeto, para que éste lo notifique cuando haya cambios de estado que sean de su interés.

Habitualmente, la implementación se maneja mediante una especie de lista que mantiene el sujeto, pero que no administra directamente. Son los observadores los que se registran en esa lista. De ese modo, cuando el sujeto debe notificar un cambio de estado, lo hace mediante una llamada al método *actualizar* de todos los objetos referenciados desde la lista.

En la figura 10.9 se muestra el diagrama de secuencia de este escenario.

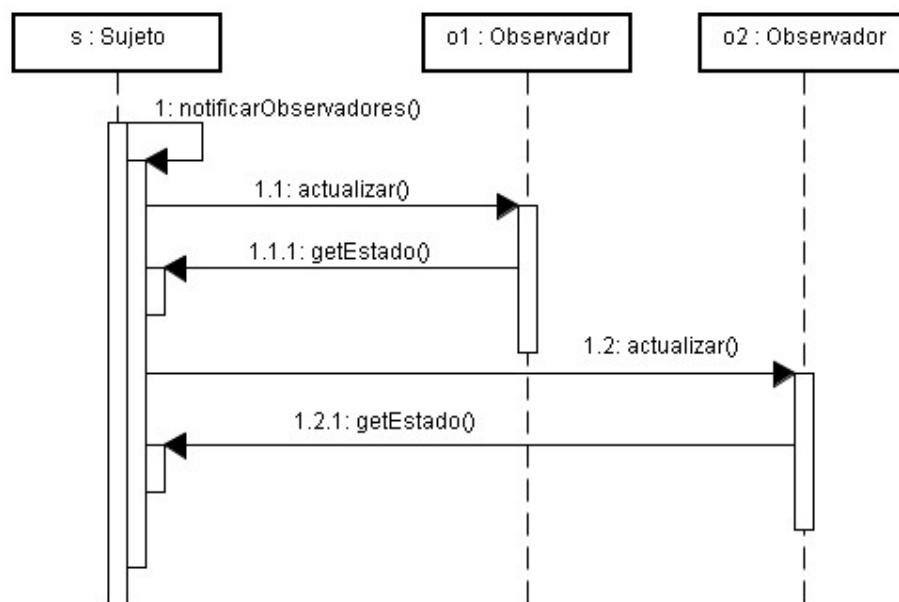


Figura 10.9 Diagrama de secuencia de una implementación de Observer

Como los observadores se pueden registrar y eliminar libremente de la lista, el sujeto no necesita conocer quiénes son, sino que se limita a recorrer la lista. La única condición que deben cumplir los observadores para anotarse en la lista de notificación del sujeto es tener un método *actualizar*, lo cual suele materializarse mediante la implementación de una interfaz. La figura 10.10 muestra el diagrama de clases típico.

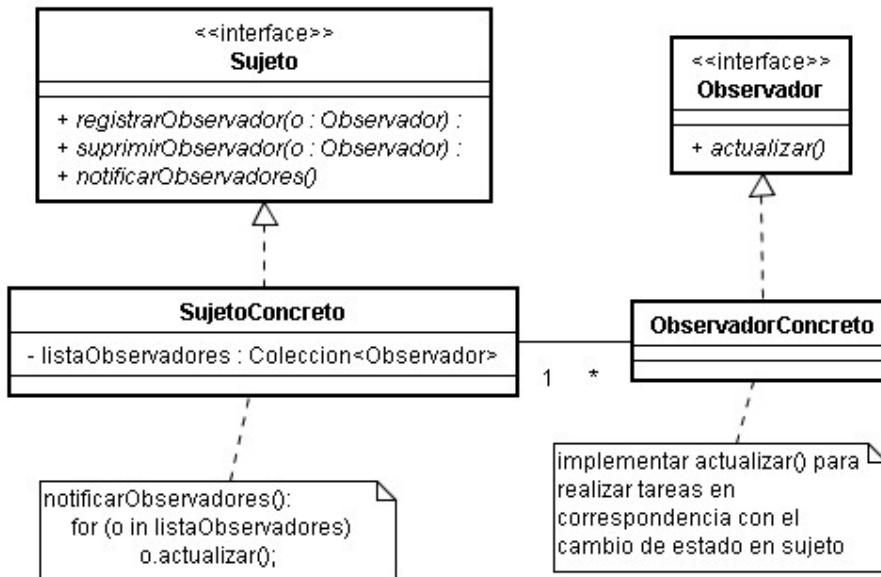


Figura 10.10 Diagrama de clases de la implementación típica de Observer

Por lo tanto, este modelo garantiza el bajo acoplamiento de sujeto y observadores, admite que los observadores sean de cualquier tipo, salvo por el hecho de que deben implementar la interfaz *Observador*, y soporta la suscripción de muchos observadores a la vez.

Hay tres maneras posibles de notificación:

- La primera, denominada “pull”, es la más típica, y se basa en que el sujeto sólo avisa a sus subscriptores que su estado cambió, sin enviar información adicional. Esto exige que, luego, los observadores le pidan su estado al sujeto, o los datos que les interesen. Si bien podría haber una inefficiencia derivada del doble llamado, este modelo es flexible y simple, y es el representado en los diagramas anteriores.
- La segunda es utilizar el modelo “push”, que consiste en que el sujeto envíe información sobre qué cambió, y cuál es el nuevo estado del emisor, de modo que no sea necesario que los observadores le pidan el estado más tarde. Este modelo también es flexible, aunque probablemente los observadores reciban mayor información que la necesaria.
- La tercera es que los observadores se subscriban a listas diferentes, por temas, y que las notificaciones les lleguen ante ciertos cambios de estado para una lista y ante otros para otras listas. Este modelo reduce mucho la cantidad de información que viaja, pero es bastante más compleja su implementación. A este modelo se lo llama “basado en eventos”, y se parece más al patrón Mediator que Observer.

Las aplicaciones de este patrón son diversas, desde el manejo de eventos en interfaces de usuario gráficas y sistemas de mensajería, hasta la implementación de patrones compuestos, como MVC. Incluso se lo suele usar en conjunto con el patrón Proxy en algunas situaciones.

Los principios de diseño que satisface este patrón son:

- Bajo acoplamiento, entre sujeto y observadores.
- Abierto-cerrado, ya que se pueden agregar observadores de tipos diferentes sin cambiar la clase del sujeto observado.
- Única responsabilidad, manteniendo separados a sujeto y observadores.

- Inversión de dependencia, pues los observadores se conocen por su interfaz.

Factory Class

Este no es uno de los patrones descriptos en [GoF 1994], pero algunos autores lo consideran una forma simplificada del Abstract Factory que sí es parte de los patrones canónicos. El mismo propone encapsular en una clase la creación de objetos, como hicimos en el ejemplo de la clase *FabricaObjetos* al presentar reflexión. El código, ya mostrado, era:

```
public class FabricaObjetos {
    ...
    public Object crearObjeto () throws ClassNotFoundException,
                                         InstantiationException,
                                         IllegalAccessException {
        String nombreClase = leerArchivo("configuracion.txt");
        Class<?> claseInstanciar = Class.forName(nombreClase);
        Object nuevo = claseInstanciar.newInstance();
        return nuevo;
    }
    ...
}
```

Este mismo ejemplo admite variantes, como lo sería la creación de cuentas bancarias, pasando el tipo de la misma como una cadena de caracteres parámetro. Por ejemplo, si para crear una instancia de *CuentaCorriente* deseásemos hacer:

```
Cuenta c =
    new FabricaCuentas().
    crearCuenta("carlosFontela.aplicacionBancaria.CuentaCorriente");
```

El código de la clase que permitiría que esto funcione sería:

```
public class FabricaCuentas {
    public Cuenta crearCuenta (String tipoCuenta)
        throws ClassNotFoundException,
               InstantiationException,
               IllegalAccessException {
        Class<?> claseInstanciar = Class.forName(tipoCuenta);
        Cuenta nueva = (Cuenta) (claseInstanciar.newInstance());
        return nueva;
    }
}
```

Por supuesto, hay más variantes de este patrón, incluyendo algunas que no necesiten reflexión.

Los principios de diseño que satisface este patrón son:

- Inversión de dependencia, ya que los clientes sólo conocen la interfaz del objeto que crean.
- Bajo acoplamiento entre clientes y los objetos creados.

Composite

Hay ocasiones en que un objeto se puede representar como un conjunto de objetos de su mismo tipo o familia de tipos. Y hay un patrón de diseño que se ocupa de este problema: Composite. Un caso particular del patrón Composite sería el caso en que, en un programa que maneja figuras geométricas, haya una clase *FiguraCompuesta*, que por un lado es una subclase de *Figura*, y por otro incluye instancias de *Figura* por agregación. Esto haría que tanto las figuras simples como las compuestas tuvieran la misma interfaz.

El caso general del patrón Composite, aplicado a ese problema, es el que figura en el diagrama de la figura 10.11.

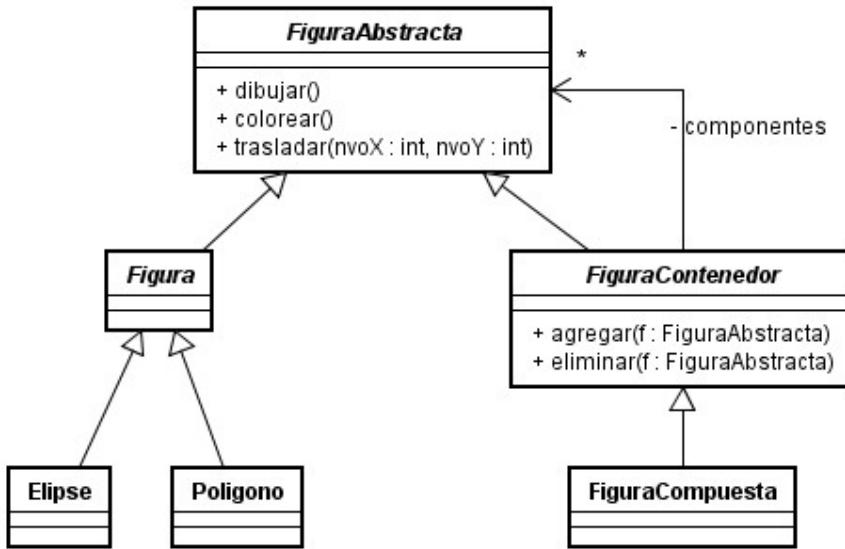


Figura 10.11 Diagrama de clases de una implementación del patrón Composite

Por favor, analice el lector las diferencias entre este diagrama y el de nuestra solución. Es importante – a esta altura sobre todo – que veamos si entendemos lo que nos dice este diagrama.

Analice un poco antes de seguir leyendo...

Veamos:

- *FiguraAbstracta*, *Figura* y *FiguraContenedor* son clases abstractas: no pueden tener instancias.
- Cada cliente accede a un objeto instancia de *FiguraAbstracta*. Este objeto es polimorfo, por lo que podría ser una instancia de cualquiera de sus descendientes concretas: *Elipse*, *Poligono* o *FiguraCompuesta*.
- Cada instancia de una clase derivada de *FiguraContenedor* (no de *FiguraContenedor* misma, pues ésta no tiene instancias) contiene una referencia a un objeto instancia de una clase descendiente de *FiguraAbstracta*. Como corolario, cada objeto *FiguraCompuesta* (descendiente de *FiguraContenedor*) puede contener referencias a *Elipse*, *Poligono*, e incluso otra *FiguraCompuesta*, en forma recursiva, pues todos descenden de *FiguraAbstracta*.

La idea, entonces, es representar componentes y contenedores mediante una interfaz común. Los agregados tienen referencias a objetos que implementen la interfaz. Asimismo, en

algunos casos conviene que los componentes tengan una referencia al agregado que los contiene.

Los principios de diseño que satisface este patrón son:

- Abierto-cerrado, ya que podemos usar y ampliar clases de componentes sin necesidad de modificarlas.
- Sustitución, al tratar los compuestos también como componentes, cuando es esperable que lo sean.

Command simplificado

El patrón Command sirve – nada más y nada menos – que para encapsular un método o un conjunto de ellos en un objeto, de modo tal que luego se los pueda usar en todas las ocasiones en que se usan los objetos.

Hay ocasiones en que resulta útil poder tratar a los métodos como objetos. Por ejemplo, podríamos necesitar:

- Parametrizar acciones a desarrollar, colocándolas en un objeto parámetro que contiene los métodos a invocar.
- Implementar acciones que luego se puedan revertir, con un par de métodos *hacer* y *deshacer*, que puedan definirse de formas alternativas. Si se desean varios niveles de *deshacer*, se puede trabajar con una pila de objetos Command.
- Priorizar peticiones poniéndolas en una cola de acciones.
- Implementar *callbacks* y eventos, permitiendo que un método sea parámetro de otro método.
- Implementar macros, que no son más que una agregación de objetos Command.

En la figura 10.12 mostramos una implementación de Command para modelar macros (nótese que una macro es una combinación de los patrones Composite y Command).

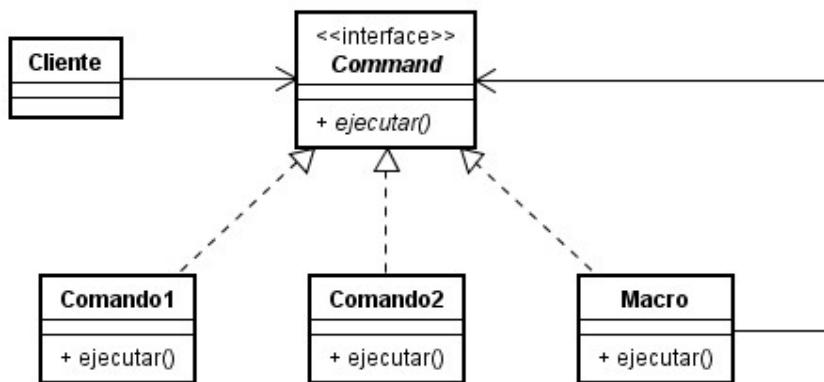


Figura 10.12 Diagrama de clases de una implementación de macros usando Command

Los objetos *Command* – instancias de clases que implementen la interfaz *Command* – son, por su misma definición, objetos sin estado, ya que su razón de ser no tiene que ver con almacenar dato alguno, sino solamente implementar uno u otro método. En definitiva, es un patrón simple que busca encapsular un método en un objeto.

Los principios de diseño que satisface este patrón son:

- Encapsulamiento de lo que varía, en este caso los posibles métodos a aplicar.
- Inversión de dependencia, ya que los distintos métodos están ocultos tras su interfaz.
- Extensión sin herencia de clases, ya que el acceso a los métodos encapsulados se hace por delegación.

Una advertencia importante. Lo que acabamos de explicar es sólo una versión simplificada de Command.

Strategy

A menudo necesitamos que el algoritmo que se emplea para resolver un problema se modifique en función de ciertas circunstancias que pueden variar dinámicamente. En estos casos, el patrón Strategy propone encapsular estas variaciones de algoritmos, definiendo clases abstractas o interfaces para los fragmentos que pueden cambiar.

Lo que hace el patrón, en definitiva, es encapsular varios algoritmos alternativos en diferentes clases y se ofrece una interfaz única para acceder a la funcionalidad ofrecida. La elección del algoritmo se vuelve transparente para el cliente, pudiendo depender del objeto e incluso variar en el tiempo.

Esto permite – además de la intención declarada de poder elegir un algoritmo en tiempo de ejecución – simplificar la clase cliente y reducir las estructuras de decisión en el mismo.

Por ejemplo, en el caso de un programa de TaTeTi, la decisión de qué estrategia aplicar en cada jugada debería depender del estado del tablero en cada momento. Strategy podría aportar una buena solución, si bien – como veremos luego – hay un patrón más adecuado.

Un caso más simple de lo mismo ocurre en la implementación de concurrencia por delegación. Recordemos que en ese caso, un nuevo hilo se lanzaba así:

```
new Thread(new HiloConcreto()).start();
```

Esto se basa en el hecho de que las instancias de *Thread* tienen una referencia a una instancia de una clase que implemente *Runnable*, como se ve en el diagrama de clases de la figura 10.13.

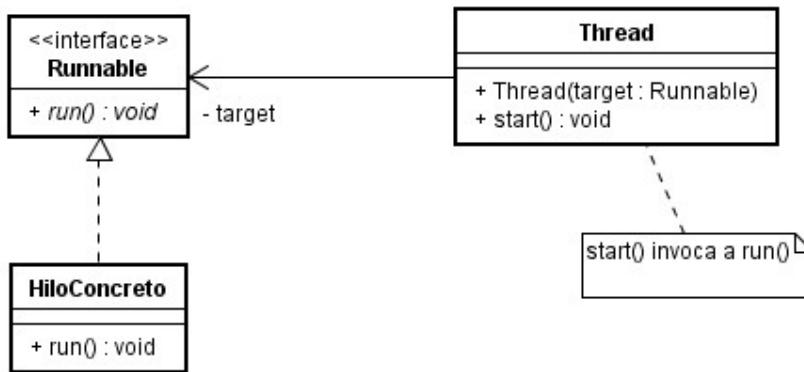


Figura 10.13 Concurrencia implementada con Strategy

Ya dijimos, al presentar el mecanismo, que esta implementación es más flexible que la basada en el patrón Template Method, ya que la elección del código a ejecutar se hace en forma dinámica, en tiempo de ejecución.

Los principios de diseño que satisface este patrón son:

- Encapsulamiento de lo que varía, al mantener los algoritmos en una jerarquía aparte.
- Inversión de dependencia, ya que utilizamos interfaces para acceder a los algoritmos.
- Extensión sin herencia de clases, al implementar las estrategias mediante una delegación.

Chain of Responsibility

Como dijimos recién, al explicar Strategy, en el caso de un programa de TaTeTi en el que deseemos dotar de cierta inteligencia a la computadora para jugar contra un humano, la decisión de qué estrategia aplicar en cada jugada debería depender del estado del tablero en cada momento. Por ejemplo, si el oponente está por ganar el juego, deberíamos intentar bloquear esa posibilidad. Si es la computadora la que está por ganar, debería poner su ficha de forma tal de ganarlo. En otros casos menos sencillos, deberíamos aplicar alguna heurística que defina la próxima jugada.

Esto se puede resolver con una cadena de decisiones sucesivas. Pero también podríamos pensar en armar una lista de estrategias a ir aplicando en forma sucesiva. Por ejemplo, siempre primero se intenta ganar el juego en la próxima jugada, luego bloquear una posible pérdida, luego... y así sucesivamente. El patrón que resuelve estos problemas se denomina Chain of Responsibility, y es una especie de Strategy anidados.

En nuestro caso, podríamos tener un método llamado *mejorCeldaParaJugar*, que invoque a un método *getProximaEstrategia*. Si esa estrategia no es aplicable, el propio *getProximaEstrategia* va a invocar a *getProximaEstrategia* en forma recursiva sobre el siguiente elemento de la lista de estrategias, y así sucesivamente hasta que la estrategia actual sea aplicable.

El diagrama de clases de la figura 10.14 modela este escenario.

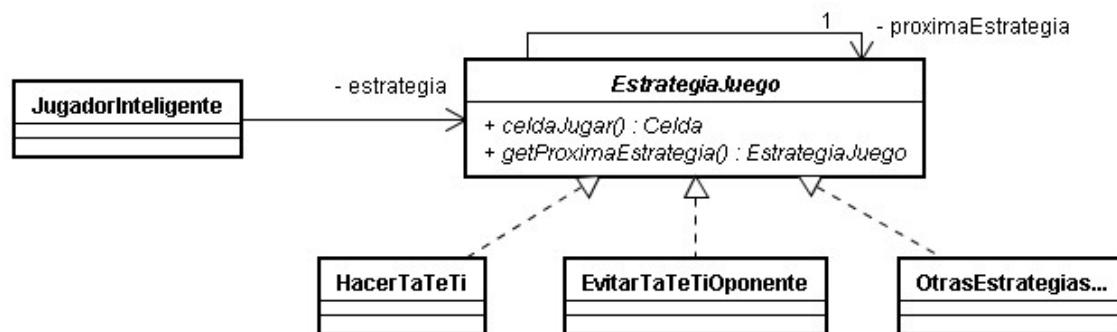


Figura 10.14 Estrategias encadenadas para jugar al TaTeTi

Los principios de diseño que satisface este patrón son:

- Encapsulamiento de lo que varía, al mantener los algoritmos en una jerarquía aparte.
- Extensión sin herencia de clases, al implementar las estrategias mediante una delegación.

Decorator

El patrón Decorator es bastante más complejo que los anteriores. Lo que busca es agregar comportamiento a uno o más métodos de una clase en forma dinámica, respetando simultáneamente el principio abierto-cerrado. Veamos la jerarquía de clases de la figura 10.15.

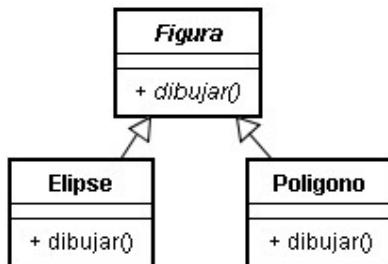


Figura 10.15 Diagrama de clases de una jerarquía de figuras

Supongamos ahora que deseamos contar cada vez que se llama el método *dibujar* en cualquier clase de las de la jerarquía. La solución con el patrón Decorator va a ser envolver los objetos instancias de *Elipse* o de *Poligono*, de modo tal que al método *dibujar* se le agregue comportamiento que incremente un contador.

El diagrama de clases de la solución sería el de la figura 10.16.

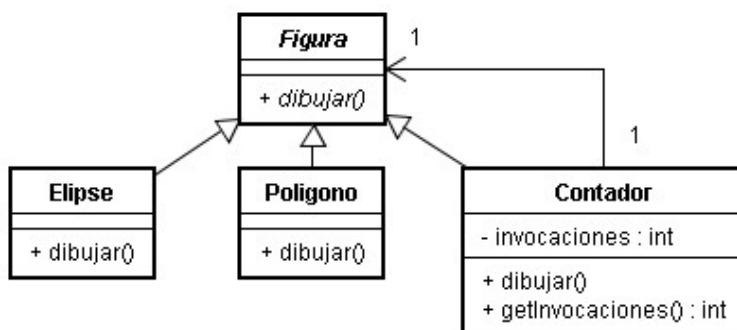


Figura 10.16 Jerarquía de figuras con contador implementado con Decorator

En ese caso, la instanciación de las clases habría que hacerlas así:

```
Figura f = new Contador (new Elipse( ) );
```

La pregunta que quedaría por hacer en este caso es: ¿y cómo evitamos que el programador cliente cree instancias de *Elipse* y *Poligono* sin envolverlas en una instancia de *Contador*?

Para eso tenemos respuesta con los patrones de creación: podemos encapsular la creación de figuras y allí envolver cada objeto en un Decorator. La solución de este problema queda como desafío para el lector.

Los principios de diseño que satisface este patrón son:

- Abierto-cerrado, ya que no necesitamos modificar las clases para agregar comportamiento a los métodos.
- Encapsulamiento de lo que varía, que en este caso son los decoradores que se agregan.
- Inversión de dependencia, ya que los clientes programan contra la interfaz más alta de la jerarquía.
- Extensión sin herencia de clases, ya que una solución más rígida pudo haber sido heredar comportamiento y agregarlo por redefinición, con el costo de que no podría hacerse en forma dinámica.

Como decíamos al principio, este patrón es bastante más complejo que lo presentado aquí, que es solamente un ejemplo de una forma simplificada del mismo.

State

La idea de este patrón es permitir que un objeto altere su comportamiento cuando cambia su estado interno, de modo tal que parezca cambiar su clase.

Por ejemplo, supongamos que queremos escribir la implementación de los posibles movimientos del juego de ajedrez, que según la pieza que se trate de mover serán diferentes. Lo haríamos mediante un método *movimientoLegal*, que verifique si un movimiento dado puede hacerse con una pieza a partir de una ubicación dada.

La solución más orientada a objetos parece ser la de definir una clase por tipo de pieza y que el método *movimientoLegal* sea diferente para cada clase. En este caso, todas las clases de pieza podrían extender a una clase abstracta *Pieza*.

El problema con esta solución es que en el ajedrez está permitido que un peón “corone”, esto es, que se convierta en reina o en cualquier otra pieza a voluntad del jugador, cuando llega a la última fila del tablero. En nuestra solución con una clase por tipo de pieza, la coronación provocaría que un objeto cambiase de clase, con todo el cambio de comportamiento asociado. Pero eso no es posible en los lenguajes en los que estamos trabajando.

La alternativa es recurrir a una solución menos elegante, sobre la base de que en realidad cada pieza no tiene un tipo diferente, sino que son simplemente instancias de la clase *Pieza*, cada una con un estado diferente. Esto deja una sola clase, con un atributo para guardar el estado, y un único método *movimientoLegal*, con un *switch* de ocho salidas, una por cada estado o tipo de pieza posible.

En estos casos, cuando nos encontramos con una solución poco elegante – la estructurada con el *switch* – como alternativa de una imposibilidad, es que debemos seguir pensando en opciones.

Y la opción que se plantea como solución es la que sigue: admitir que cada pieza es simplemente eso, sin ser cada una de un tipo diferente, pero delegar en otra clase, que podemos llamar *Estado*, la cuestión de almacenar el estado y el método *movimientoLegal*. Por lo tanto, cada pieza tendrá un estado, pero ese estado estará representado por un objeto con comportamiento. Mejor aún, *Estado* podría ser una interfaz, con una clase descendiente para cada pieza del ajedrez.

El diagrama de clases de la solución que acabamos de exponer es el de la Figura 10.17.

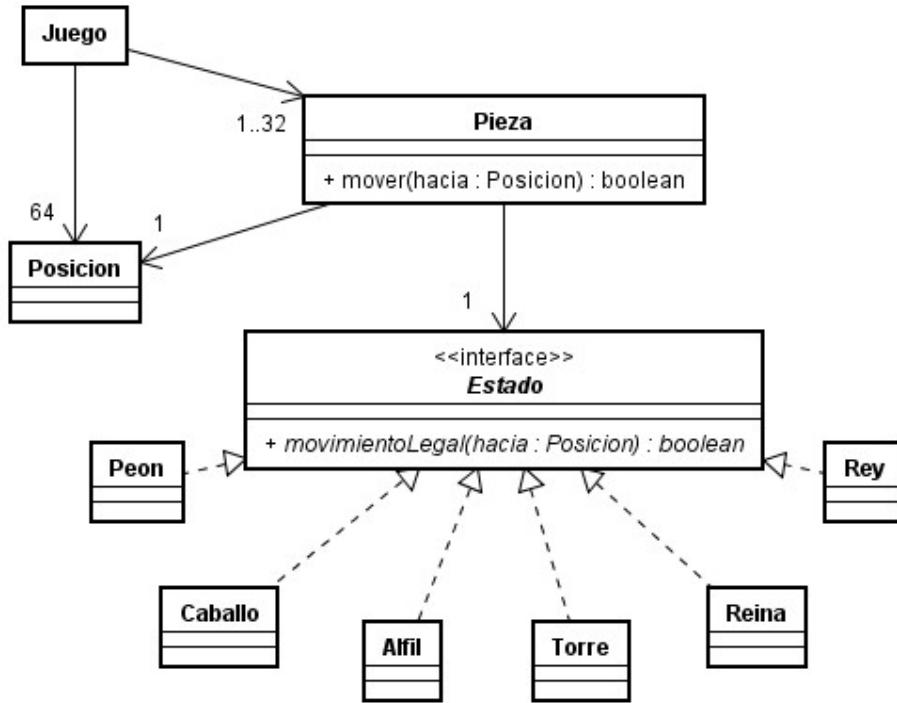


Figura 10.17 Diagrama de clases de la implementación de State para el ajedrez

Además, las clases descendientes de *Estado* no tiene sentido que tengan más de una instancia.

Ahora sí que estamos para implementar. Lo que sí deberíamos hacer es asegurarnos – antes de seguir adelante – de que entendemos el diagrama en cuestión. A esta altura es esencial asegurarnos de que podemos “leer” un diagrama, de modo tal de poder descubrir qué significa en términos de implementación. Si no, los diagramas no nos servirán.

Analice un poco antes de seguir leyendo...

El diagrama nos dice:

- Cada instancia de *Pieza* tiene una referencia a un objeto instancia de una clase que implementa la interfaz *Estado*. Esto implica que tiene una referencia a un objeto *Peon*, *Torre*, *Caballo*, etc.
- Las clases que implementan *Estado* sólo tienen una instancia. Esto implica que todas las piezas que estén en estado de *Peon* van a referenciar a la única instancia posible de esa clase. Lo mismo ocurre con todos los demás estados.

La implementación es sencilla. O mejor dicho, es tan compleja como complejo resulta expresar las reglas de ajedrez, pero no más que eso.

En la clase *Pieza* tendremos el método *mover*, que – simplificando un poco – será así:

```

public boolean mover (Posicion hacia) {
    // chequeo inicial de turno
    if (estado.movimientoLegal (this.posicion, hacia, color)) {
        // se realiza el movimiento
        // se verifica la condición de coronación
        // y se efectúa si corresponde
        // se cambia de turno
    }
}

```

```

        return true;
    }
    else return false;
}

```

A su vez, el método *movimientoLegal* estará definido en cada clase que implemente *Estado*, con las reglas del ajedrez.

Cuando haya que coronar una pieza, por ejemplo un peón que pasa a convertirse en reina, el código en cuestión será el siguiente:

```
this.estado = Reina.getEstado();
```

Lo que hemos hecho en esta solución es un caso de implementación del patrón State, que separa los posibles estados en otros objetos distintos de aquél que detenta este estado. El objetivo es poder implementar en forma simple los cambios de estado que implican cambios de comportamiento.

Un caso adicional del patrón State lo planteamos – sin nombrarlo – cuando nos planteamos, en el capítulo de polimorfismo, en el ejercicio del Sudoku, qué pasaría si separásemos en dos clases la celda libre y la celda ocupada. Allí dijimos que en JavaScript la implementación era la de dos prototipos distintos, pero en Smalltalk o Java deberíamos usar el patrón State.

La implementación de este patrón es prácticamente equivalente a la del patrón Strategy, pero sus propósitos son diferentes.

Los principios de diseño que satisface este patrón son:

- Encapsulamiento de lo que varía, en este caso los estados.
- Inversión de dependencia, ya que a los estados se accede mediante su interfaz.
- Extensión sin herencia de clases, ya que los estados se acceden mediante delegación.

Double Dispatch y Visitor

Hay situaciones en las cuales no resulta posible definir un cierto comportamiento en un único objeto.

Por ejemplo, supongamos que implementamos un juego de piedra, papel y tijera⁵¹. Las pruebas que proponemos son:

```

class PruebasPiedraPapelTijera {

    Material piedra = new Piedra();
    Material papel = new Papel();
    Material tijera = new Tijera();

    @Test
    public void piedraVsTijera() {
        Assert.assertEquals(piedra, piedra.ganadorContra(tijera));
        Assert.assertEquals(piedra, tijera.ganadorContra(piedra));
    }
}

```

⁵¹ Es un juego de manos en el que existen tres elementos: la piedra que vence a la tijera rompiéndola, la tijera que vence al papel cortándolo y el papel que vence a la piedra envolviéndola. Se utiliza para dirimir algún asunto, tal y como se hace a veces usando una moneda.

```

    }

    @Test
    public void piedraVsPapel() {
        Assert.assertEquals(papel, piedra.ganadorContra(papel));
        Assert.assertEquals(papel, papel.ganadorContra(piedra));
    }

    @Test
    public void papelVsTijera() {
        Assert.assertEquals(tijera, papel.ganadorContra(tijera));
        Assert.assertEquals(tijera, tijera.ganadorContra(papel));
    }

}

```

Una implementación posible usaría tres clases, una para cada material, y una interfaz *Material*, como las que siguen:

```

public interface Material {
    Material ganadorContra (Material material);
    Material ganadorContra (Piedra piedra);
    Material ganadorContra (Papel papel);
    Material ganadorContra (Tijera tijera);
}

```

```

public class Piedra implements Material {

    public Material ganadorContra (Material material) {
        return material.ganadorContra(this);
    }

    public Material ganadorContra (Piedra piedra) {
        return this;
    }

    public Material ganadorContra (Papel papel) {
        return papel;
    }

    public Material ganadorContra (Tijera tijera) {
        return this;
    }
}

```

```

public class Papel implements Material {

```

```

public Material ganadorContra (Material material) {
    return material.ganadorContra(this);
}

public Material ganadorContra (Piedra piedra) {
    return this;
}

public Material ganadorContra (Papel papel) {
    return this;
}

public Material ganadorContra (Tijera tijera) {
    return tijera;
}
}

```

```

public class Tijera implements Material {

    public Material ganadorContra (Material material) {
        return material.ganadorContra(this);
    }

    public Material ganadorContra (Piedra piedra) {
        return piedra;
    }

    public Material ganadorContra (Papel papel) {
        return this;
    }

    public Material ganadorContra (Tijera tijera) {
        return this;
    }
}

```

A este enfoque se lo llama de despacho doble, o por su nombre más habitual en inglés, Double Dispatch. En programación en general (no sólo POO), “despachar” es el procedimiento que se sigue en tiempo de ejecución para determinar qué función debe llamarse basándose en los parámetros que se reciben. En POO, y gracias al polimorfismo, en el caso del objeto receptor del mensaje (*self* o *this*, que no es otra cosa que un parámetro implícito), se despacha el método de la clase de la cual el objeto receptor es instancia (lo que hemos llamado vinculación tardía).

Lo habitual es que los lenguajes de POO sólo permitan este nivel de polimorfismo. Hay otros

lenguajes⁵², que permiten definir comportamiento polimorfo respecto de otros parámetros. A esto se lo llama despacho múltiple, o multimétodos.

Double Dispatch no es un patrón estrictamente hablando. Es una técnica que descubrió y presentó muy tempranamente Dan Ingalls, con el nombre de “polimorfismo múltiple” [Ingalls 1986]. En el catálogo inicial de [GOF 1994] figura otro patrón, un poco más complejo, llamado Visitor, y que usa como implementación típica Double Dispatch.

Visitor se usa cuando los objetos entre los cuales debemos hacer el despacho doble están en jerarquías distintas.

Si volvemos a nuestra ya conocida aplicación bancaria y le agregamos una jerarquía de clientes, podríamos llegar al diagrama de la figura x.x.

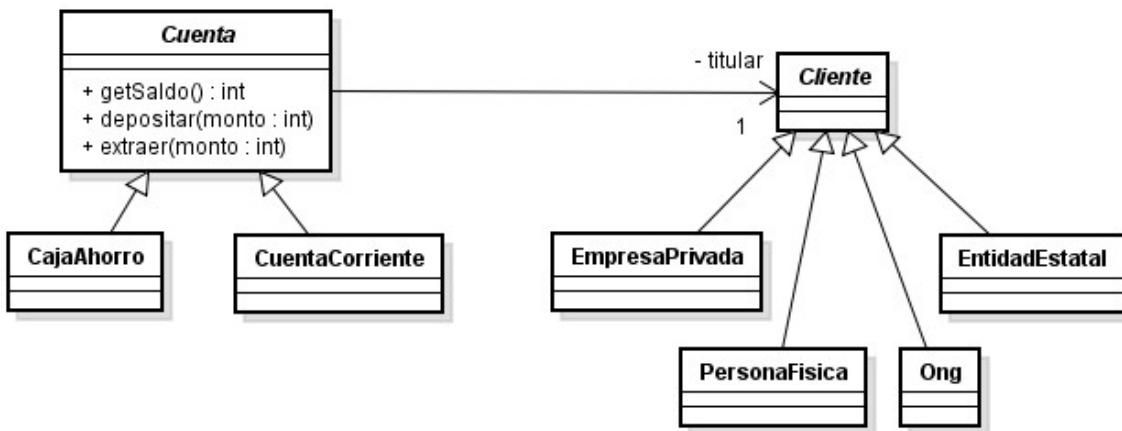


Figura x.x Diagrama de clases cuentas y clientes

Si además existiese una reglamentación que exige retener el pago de un impuesto por cada depósito en cuenta⁵³, que dependa a la vez del tipo de cuenta y del tipo de cliente, esto se podría resolver con el código que sigue.

En la clase *Cuenta*:

```

public void depositar(int monto) {
    if (monto <= 0)
        throw new MontoInvalido();
    this.saldo += monto;
    retenerImpuesto(monto);
}

private void retenerImpuesto (int montoDeposito) {
    int montoImpuesto = (int)
        (porcentajeImpuestoPorDeposito() * montoDeposito);
    if (montoImpuesto > 0)
        AgenciaRecaudacion.agencia().depositar(montoImpuesto);
}
  
```

⁵² Common Lisp es uno de ellos.

⁵³ Esta reglamentación existe en la República Argentina, en la forma de una percepción a cuenta de impuestos, aunque con porcentajes diferentes a los de este ejemplo.

```
protected abstract double porcentajeImpuestoPorDeposito();
```

En la clase *CajaAhorro*:

```
protected double porcentajeImpuestoPorDeposito() {
    if (this.getTitular() instanceof EmpresaPrivada)
        return 0.06;
    else // EntidadEstatal || PersonaFisica || Ong
        return 0.0;
}
```

En la clase *CuentaCorriente*:

```
protected double porcentajeImpuestoPorDeposito() {
    if (this.getTitular() instanceof EmpresaPrivada) {
        return 0.03;
    }
    else if (this.getTitular() instanceof PersonaFisica ||
              this.getTitular() instanceof Ong) {
        return 0.01;
    }
    else // EntidadEstatal
        return 0.0;
}
```

Ahora bien, esto hace que haya una violación del principio abierto-cerrado, ya que las clases de una jerarquía (*Cuenta*) deben conocer la existencia de las clases de otra (*Cliente*). Esto genera un problema si hay que modificar la jerarquía de clientes en algún momento.

Podríamos apoyarnos en el polimorfismo para resolver este problema, en vez de las acciones de selección (*if*) preguntando por el tipo de los objetos. Veamos la solución que sigue.

Cambiemos el método *porcentajeImpuestoPorDeposito* de la clase *CajaAhorro*:

```
protected double porcentajeImpuestoPorDeposito() {
    return this.getTitular().alicuotaCajaAhorro(this);
}
```

También en la clase *CuentaCorriente*:

```
protected double porcentajeImpuestoPorDeposito() {
    return this.getTitular().alicuotaCuentaCorriente(this);
}
```

Por otro lado, la clase *Cliente* debería tener dos métodos abstractos:

```
public abstract double alicuotaCajaAhorro (CajaAhorro cuenta);
public abstract double alicuotaCuentaCorriente (CuentaCorriente cuenta);
```

Que en la clase *EntidadEstatal* quedarían implementados así:

```
public double alicuotaCajaAhorro (CajaAhorro cuenta) {
    return 0.0;
}

public double alicuotaCuentaCorriente (CuentaCorriente cuenta) {
    return 0.0;
}
```

En la clase *EmpresaPrivada* así:

```
public double alicuotaCajaAhorro (CajaAhorro cuenta) {  
    return 0.06;  
}  
  
public double alicuotaCuentaCorriente (CuentaCorriente cuenta) {  
    return 0.03;  
}
```

Tanto la clase *Ong* como *PersonaFisica* lo implementarían así:

```
public double alicuotaCajaAhorro (CajaAhorro cuenta) {  
    return 0.0;  
}  
  
public double alicuotaCuentaCorriente (CuentaCorriente cuenta) {  
    return 0.01;  
}
```

Como vemos, en la solución hay dos objetos que pretenden resolver el problema y se reparten las responsabilidades. El objeto de la jerarquía de *Cuenta* le pide el valor de la alícuota a adoptar al objeto de la jerarquía de *Cliente*, que a su vez resuelve en base al tipo del objeto de *Cuenta* corresponda. De alguna manera, la responsabilidad está compartida por dos objetos, uno en cada jerarquía, que se pasan mensajes mutuamente. Esto es lo que hace que a esta solución se la haya denominado de despacho doble (o Double Dispatch).

En realidad, en Java, y dado que existe el mecanismo de sobrecarga, los métodos *alicuotaCajaAhorro* y *alicuotaCuentaCorriente* podrían denominarse ambos *alicuota*, ya que en realidad son diferentes por tener diferentes tipos en el parámetro. Por ejemplo, en *Cliente* podría quedar:

```
public abstract double alicuota (CajaAhorro cuenta);  
public abstract double alicuota (CuentaCorriente cuenta);
```

Y lo mismo ocurriría en las clases hijas.

Esto no ocurre en Smalltalk. Lo que sí es importante entender, aun en Java, es que no podemos unificar ambos métodos *alicuota* en uno solo, ya que el despacho doble funciona por una combinación de polimorfismo (un único método *porcentajeImpuestoPorDeposito* en la familia de *Cuenta*) y de sobrecarga (dos métodos *alicuota* en la familia de *Cliente*).

Sin embargo, hay un problema que subsiste: ¿qué pasaría si deseamos agregar un nuevo tipo de cuenta? Vamos a tener que modificar toda la familia de *Cliente* agregando un método más en todas las clases. Esto ocurre porque, si bien hemos eliminado las acciones de selección, reemplazándolas por una solución polimorfa, seguimos teniendo a una familia de clases dependiendo de la otra.

De todas maneras, esto de que un cambio en una jerarquía provoque un cambio en otra pasa con otros patrones (Iterator, por ejemplo). Por otro lado, suele ser más claro que otras alternativas de implementación, aunque habrá siempre que analizar si el problema que estamos resolviendo no admite implementaciones más simples.

Finalmente, la solución con el uso de *if* e información de tipos en tiempo de ejecución (RTTI) es mucho menos segura ante la necesidad de extensión, ya que nos podemos olvidar de algún *if*, y eso quedaría enterrado en el código.

Los principios de diseño que satisface este patrón son:

- Abierto – cerrado, ya que agregar nuevos casos sólo es necesario extender las clases existentes en una jerarquía y agregando métodos en la otra⁵⁴.

Adapter

El patrón Adapter tiene por propósito dar a un objeto una interfaz diferente a la que tiene.

Veámoslo con un caso práctico. Supongamos que una aplicación es cliente de un objeto provisto por un proveedor externo, y que en determinado momento se decide cambiar de proveedor. Obviamente, lo más probable es que la nueva clase tenga una interfaz diferente, y que no controlemos esa interfaz. ¿Qué pasa si no se desea cambiar el programa cliente? En ese caso, no podríamos cambiar ni cliente ni proveedor, debiendo crear un “adaptador”.

Para exemplificar aún más, pensemos en un problema no informático. Supongamos que necesito enchufar una heladera nueva en mi casa, pero que la misma viene con un enchufe de tipo estadounidense, mientras que los tomacorrientes de mi casa son del tipo argentino. Si no deseo cambiar ni el enchufe ni el tomacorriente, lo más sencillo es comprar un adaptador, al cual enchofo la heladera, y éste a su vez lo enchofo en el tomacorriente. El adaptador lo único que hace es cambiar la interfaz.

La idea es, entonces, que el cliente se comunique con el adaptador, llamando un método de una interfaz adaptadora. El objeto adaptador convierte la solicitud en una o varias llamadas sobre la interfaz del servidor, por delegación. Luego, el cliente recibe los resultados, sin saber qué traducción realizó el adaptador, o incluso podría desconocer que existe un adaptador.

Este proceso se ve bien en el diagrama de secuencia de la figura 10.18.

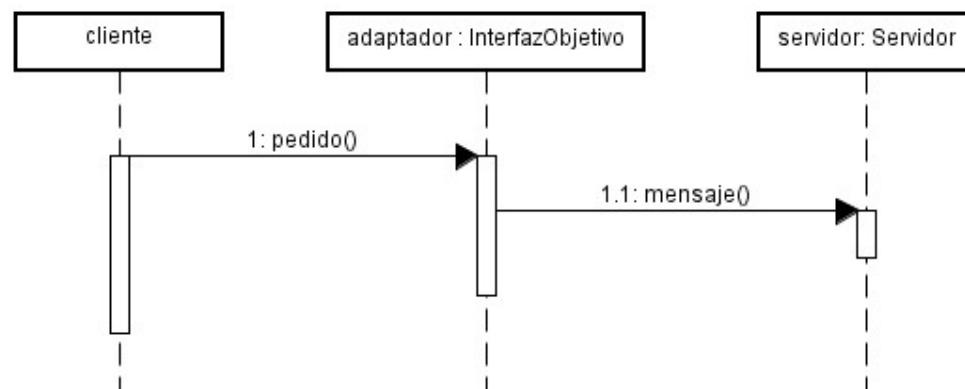


Figura 10.18 Diagrama de secuencia de un Adapter

Y el diagrama de clases que lo respalda podría ser el de la figura 10.19.

⁵⁴ Esto último se puede ver como una violación del principio abierto-cerrado, aunque en cualquier caso el compilador – en el caso de Java – nos va a forzar a no olvidarnos de implementarlo. Como dijimos, es importante verificar que no existen soluciones más sencillas.

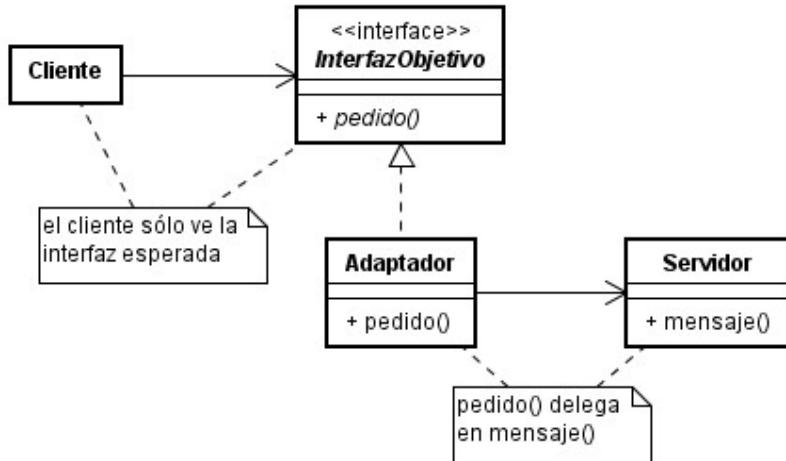


Figura 10.19 Diagrama de clases de un Adapter

El uso del patrón Adapter se justifica cuando se da alguna de las siguientes situaciones:

- Dos o más objetos tienen comportamiento similar con interfaces diferentes.
- El código cliente podría ser más simple si varios servidores tuvieran la misma interfaz.
- No se puede alterar la interfaz de alguna de los objetos que brindan servicios similares. Sea porque no tenemos el código fuente, porque es parte de una biblioteca de terceros o porque hay otros sistemas que la usan con su interfaz actual.

Notemos que Adapter sólo se usa para cambiar la interfaz, sin agregar comportamiento por parte del objeto intermediario. Si se quisiera agregar comportamiento, lo más atinado sería utilizar el patrón Decorator, ya visto. Si, en cambio, se desea simplificar una interfaz a varias clases complejas, debemos utilizar Facade, que veremos en seguida.

Los principios de diseño que satisface este patrón son:

- Inversión de dependencia, ya que la conexión se hace con una interfaz.
- Extensión sin herencia de clases, ya que el cliente utiliza el adaptador por delegación.
- Encapsulamiento de lo que varía, en este caso la interfaz del objetivo.
- Abierto-cerrado, ya que no modificamos la clase objetivo aunque necesitamos una interfaz diferente.

Facade

El patrón Facade también provee un cambio de interfaz, pero su propósito es diferente de Adapter.

La idea de Facade es simplificar el uso de un conjunto de clases de un subsistema, de modo tal que varias clases del subsistema se mapean en una sola que hace de fachada.

La figura 10.20 muestra un diagrama estático esquemático.

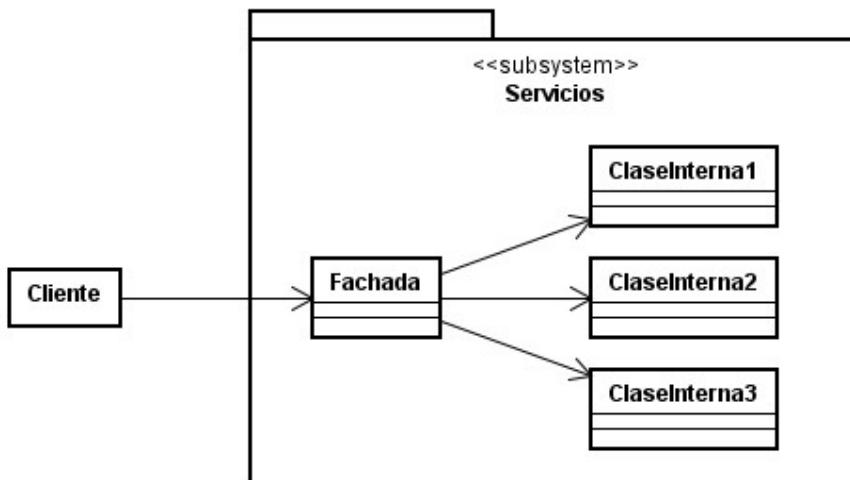


Figura 10.20 Diagrama estático de una implementación de Facade

La ventaja que provee el uso de una fachada sobre el acceso directo a las clases es, ante todo, la simplicidad. En efecto, sólo se utiliza una clase, en principio con menos métodos, lo cual hace más sencillo el aprendizaje, el primer uso y el mantenimiento. El acoplamiento también se reduce notablemente. Y tiene la ventaja – derivada de las anteriores – de que si se desean cambiar las interfaces de las clases internas del subsistema, podemos hacerlo sin problema, modificando solamente la fachada para que el cambio sea transparente a los clientes.

En cuanto a la implementación, la existencia de una fachada no obliga a ocultar las clases del subsistema a los clientes. Sin embargo, es conveniente hacerlo si la fachada provee acceso a toda la funcionalidad que se requiere por los posibles clientes. La posible implementación, en este caso, sería dar visibilidad de paquete a las clases internas, dejando como pública solamente a la clase *Fachada*.

Hay un uso de Facade que, si bien no es el propósito del patrón, se ha difundido bastante.

Ocurre que, cuando un sistema se conecta con una aplicación externa para solicitarle información, es conveniente que a través de la red viaje la mayor cantidad de datos cada vez, debido al costo de varias conexiones sucesivas. Por otro lado, es un principio de diseño normalmente aceptado que los métodos de las clases deben devolver datos lo más elementales posible, lo que se conoce como el principio de granularidad fina. ¿Cómo satisfacemos ambas necesidades, de granularidad fina para los clientes en la misma aplicación y granularidad gruesa para los clientes externos?

La respuesta es simple: colocamos una fachada, que sea utilizada por las aplicaciones externas, de granularidad gruesa. Y hacia adentro, esta clase *Fachada* puede realizar llamadas sobre métodos de grano fino.

Los principios de diseño que satisface este patrón son:

- Bajo acoplamiento entre cliente y proveedores, a través de la fachada.
- Segregación de la interfaz, definiendo la fachada para evitar que los clientes estén relacionados con métodos que no necesitan utilizar directamente.
- Encapsulamiento de lo que varía, en este caso las clases internas del subsistema.
- Granularidad fina hacia dentro del subsistema, y posiblemente gruesa hacia fuera.
- Abierto-cerrado, ya que no modificamos las clases objetivo aunque necesitemos una

interfaz diferente, más simple.

Proxy

Proxy es otro patrón de intermediación, en este caso sin cambio de interfaz.

La idea es que un objeto haga de intermediario entre un cliente y un objetivo, haciendo que todas las comunicaciones con el objetivo se deban realizar a través de su intermediario. De hecho, “proxy” en inglés significa “apoderado”. Ambos objetos, el intermediario y el objetivo, tienen la misma interfaz, de modo que el cliente pueda trabajar con aquél como si fuera el propio objetivo.

El diagrama de secuencia de la figura 10.21 muestra el escenario típico, en el cual el objeto proxy puede ejecutar algún comportamiento antes o después de invocar métodos sobre el objetivo.

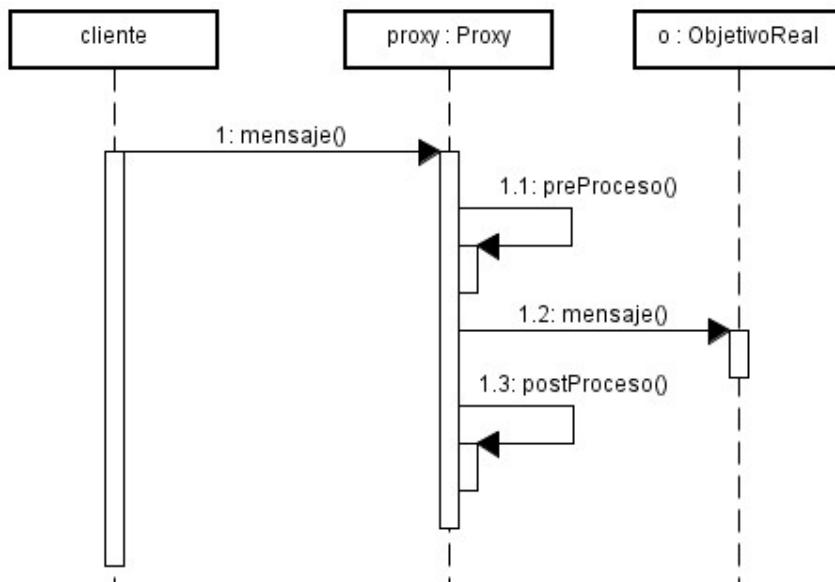


Figura 10.21 Diagrama de secuencia típico de un Proxy

Esta es una simplificación. De hecho, el intermediario muchas veces no llega a invocar los métodos sobre el objetivo, devolviendo lo que el cliente solicitó sin que medie ninguna comunicación con el objetivo.

El diagrama de clases sería el de la figura 10.22.

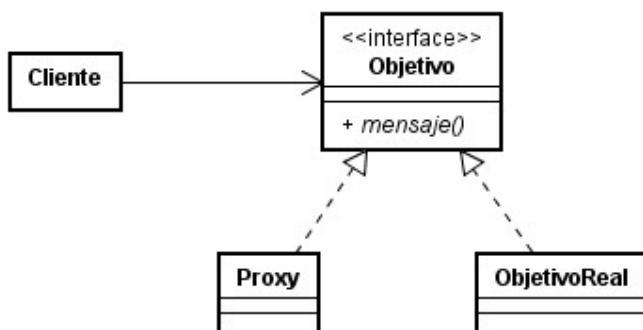


Figura 10.22 Diagrama de clases de una implementación típica de Proxy

En casi todas las implementaciones de Proxy, el cliente tiene una referencia a un *Objetivo*, y desconoce si lo está atendiendo una instancia de *Proxy* o de *ObjetivoReal*. Incluso al crearse el objeto de tipo *Objetivo*, se lo suele hacer mediante un patrón de creación, que le devuelve al cliente un objeto del cual sólo conoce la interfaz que implementa.

Los usos del patrón Proxy son muchos. Entre ellos están:

- Proxy remoto: el más usual, cuando el objetivo está en un sistema remoto. Suele servir como caché o copia local, evitando que haya que ir a buscar información cada vez al objeto remoto.
- Proxy virtual: cuando no se crea el objetivo sino hasta que sea realmente necesario, porque es costoso hacerlo.
- Proxy de protección: para controlar el acceso al objetivo.
- Proxy de sincronización: para gestionar acceso de varios clientes al objetivo.

Nótese que el intermediario no siempre se comunica con el objetivo. Por ejemplo, en el caso de un proxy remoto haciendo de caché, el intermediario va a recibir solicitudes dirigidas al objetivo, pero podrá responderlas él mismo si el objetivo no ha cambiado de estado desde la última solicitud. Para desacoplar aún más a ambos, podemos implementar un mecanismo de notificación desde el objetivo, usando el patrón Observer.

Los principios de diseño que satisface este patrón son:

- Bajo acoplamiento entre cliente y objetivo.
- Inversión de dependencia, ya que la conexión se hace con la interfaz objetivo, no contra el propio objetivo.
- Segregación de la interfaz, al no dar acceso a más objetos de los necesarios.

MVC

MVC (acrónimo inglés de “Model-View-Controller”) es un patrón macro que consiste en separar la aplicación en tres partes: modelo, vista y controlador. Véase la figura 10.23. Ha demostrado ser muy adecuado en aplicaciones con alta interacción con el usuario.

El **modelo** es el conjunto de componentes correspondientes a la lógica de la aplicación (estados y funcionalidad). La idea es que este modelo tenga un bajo acoplamiento con vistas y controladores. Toda la interacción se debe hacer mediante métodos de consulta, que informen el estado del modelo, comandos que permitan modificar dicho estado y mecanismos de notificación para informar a los observadores o vistas.

La **vista** es la parte del sistema que administra la visualización y presentación de la información. Su principal tarea es observar al modelo para actualizar las variaciones, de modo que la vista represente el estado del modelo, así como los cambios de estado que experimenta el mismo. Es un conjunto de componentes altamente dependiente del dispositivo y la tecnología de visualización, así como también del modelo, al que debe conocer bien. Si en el modelo se define una interfaz clara y estable, es fácil implementar múltiples vistas para un mismo modelo.

El **controlador** es el responsable de manejar el comportamiento global de la aplicación. Su principal tarea consiste en recibir los eventos del usuario y decidir qué es lo que se debe hacer, mapeándolos en comandos (mensajes) hacia el modelo, que eventualmente lo modifican. Como la vista, igualmente el controlador es altamente dependiente de los

dispositivos y mecanismos de interacción del usuario, y también debe conocer bien al modelo.

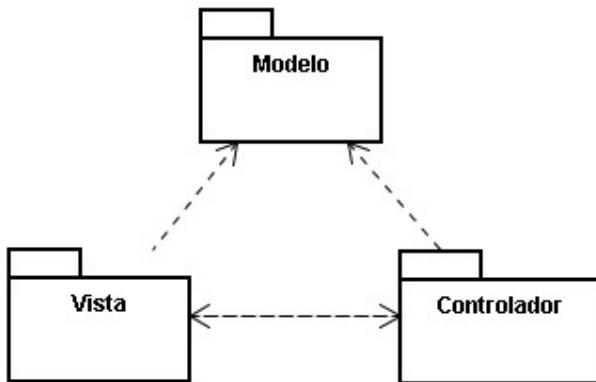


Figura 10.23 Diagrama de paquetes del patrón MVC

En un sistema con interfaz de usuario gráfica, una implementación simple de MVC consiste en un ciclo con la siguiente secuencia:

- El controlador captura los eventos del usuario para determinar qué objetos están siendo manipulados.
- Una vez que interpreta las acciones, envía mensajes al modelo para generar cambios de estado.
- El modelo realiza las transformaciones que sean necesarias, cambiando de estado si fuese necesario.
- El controlador o el modelo envían la notificación a la vista para que se actualice.
- Finalmente, la vista se actualiza para reflejar las novedades, consultando previamente al modelo.

Interesa analizar los mecanismos de notificación por los cuales el modelo puede dar a conocer sus cambios a las vistas (notemos que en cualquiera de los casos, el controlador también puede observar al modelo):

- Consulta de estado y respuesta. En este mecanismo, la vista consulta al modelo sobre su estado y se actualiza en consecuencia. El modelo desconoce qué pasa, limitándose a responder a los mensajes recibidos (comandos y consultas). Es la variante más en consonancia con la POO. Sin embargo, exige a la vista un mayor conocimiento del modelo que en las variantes que siguen.
- Mediante eventos. Ésta es una variante totalmente desacoplada, en la que las vistas escuchan y responden a los eventos de notificación de sus respectivos modelos, en la medida en que les interesa. El bajo acoplamiento se garantiza haciendo que las propias vistas sean quienes se suscriben en la lista del modelo, permitiendo un manejo totalmente dinámico. El mejor patrón para implementar este mecanismo es Mediator.
- Con observadores o vistas asociadas, conocidos por el modelo. Éste les envía activamente un mensaje de notificación, sin información, para que luego las vistas se preocupen de actualizarse pidiendo datos al modelo. Éste es un método fácil de implementar, y bastante flexible si utiliza eventos también. Es el mecanismo del patrón Observer, y es la manera más habitual de implementar

MVC.

Los tres mecanismos admiten trabajar con múltiples vistas.

Una cuestión a hacer notar es que, en las implementaciones de MVC para aplicaciones de escritorio con interfaz de usuario gráfica, la vista y el controlador están muy interrelacionados, pues ambos dependen de la interacción con el usuario. Sin embargo, no es así en aplicaciones web, donde la funcionalidad del controlador en parte la maneja el generador de páginas en el servidor: para ello existe un patrón de diseño del controlador denominado PageController [Fowler 2002].

En cualquier caso, vista y controlador deben comunicarse constantemente. Por eso, en general, se encuentran de a pares, de modo que hay un controlador por cada vista. Incluso hay otros modelos que implementan vista y controlador como una única entidad.

Si analizamos la separación de incumbencias, MVC aísla claramente la lógica de negocio (modelo) de la interfaz de usuario (vista y controlador). Como es un patrón especialmente pensado para aplicaciones de mucha interfaz de usuario, el acento está puesto en ella, al punto que la divide en dos partes, mientras todo el resto de la aplicación, incluyendo potencialmente la persistencia y los servicios, se maneja desde el modelo.

Separación en capas

El término capa hace referencia a las capas de una torta, como la torta milhoja argentina; una capa se apoya sobre la anterior, y sólo está en contacto con la capa que está apoyada en ella y la que se encuentra debajo. La figura 10.24 es un buen esquema.

La idea de los patrones basados en capas es que cada módulo – denominado capa – depende de una única capa inferior y brinda servicios a una o más capas superiores. Por lo tanto, cada capa conoce a la inmediata inferior, de la que depende, pero no puede conocer nada de las capas superiores que utilizan sus servicios, ni de aquellas capas inferiores no adyacentes.

A menudo se malinterpreta el modelo de capas, suponiendo que dicta un orden en el que se debe construir la aplicación. Esto es un error, y los que proclaman que el modelo de capas exige que las capas inferiores estén diseñadas para construir las superiores, confunden la idea de un patrón de arquitectura con un proceso de construcción.

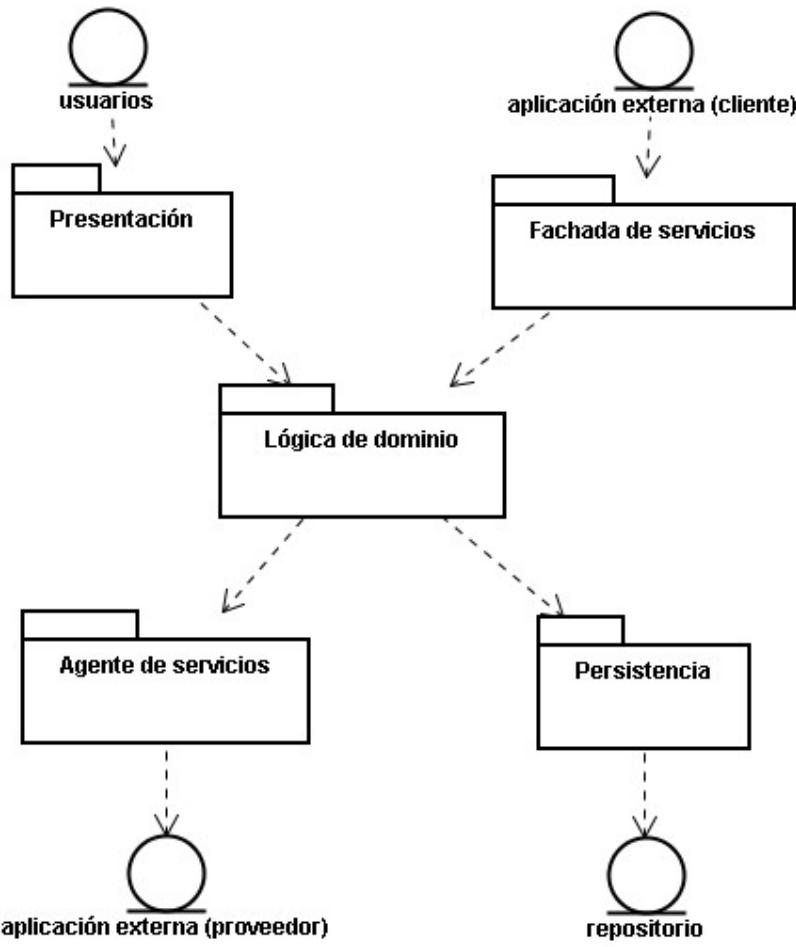


Figura 10.24 Diagrama de paquetes de una implementación de un sistema en capas

Las ventajas de este patrón son:

- Abstracción y encapsulamiento: cada capa se refiere a una abstracción diferente y puede ser analizada y comprendida sin saber cómo están implementadas el resto de las capas. Por ejemplo, puedo tener una capa de acceso a base de datos y otra de interacción con el usuario en una aplicación, y cada una es independiente de la otra.
- Bajo impacto de cambios: se puede cambiar la implementación de una capa sin afectar a las demás. En el ejemplo anterior, si cambio la base de datos por almacenamiento en archivos XML, sólo cambiaré la capa de persistencia, y el resto de la aplicación no se verá afectada.
- Bajo acoplamiento: la dependencia entre capas se mantiene al mínimo. Cada capa depende de una sola capa inferior.
- Una misma capa puede dar servicios a varias capas superiores colocadas en paralelo. Una misma capa de acceso a datos persistentes puede servir para que varias capas – o incluso varias aplicaciones – utilicen el almacenamiento de datos en cuestión.

Las desventajas, si bien son pocas, existen de todos modos:

- A veces un cambio en una capa inferior lleva a cambiar capas superiores que dependen de ella, con un efecto en cadena que puede resultar muy amplio.
- Muchas capas pueden dañar el desempeño por la necesidad de llamadas sucesivas de métodos.

Un caso particular de este patrón anterior es el de tres capas. La arquitectura de tres capas implica dividir el sistema en:

- Capa de interfaz de usuario
- Lógica de la aplicación
- Capa de acceso a datos

El polémico patrón Singleton

La idea de Singleton es construir una clase que sólo admita una instancia y dar un punto de acceso global a la esta última.

La solución es crear un solo objeto y reutilizarlo, manteniendo el constructor privado para que no se puedan crear otros objetos. Lo que hacemos, entonces, es encapsular el código que administra la reutilización.

A continuación se muestra un Singleton para la aplicación bancaria, implementado en Java.

```
public class Banco {

    // esta es la referencia a la única instancia de la clase:
    private static Banco instancia = null;

    // este es el método para acceder a la única instancia:
    public static Banco banco () {
        if (instancia == null)
            instancia = new Banco();
        return instancia;
    }

    private Collection<Cuenta> cuentas;
    private Collection<Cliente> clientes;

    private Banco() {
        this.cuentas = new ArrayList<Cuenta>();
        this.clientes = new ArrayList<Cliente>();
    }

    public Collection<Cuenta> getCuentas() {
        return cuentas;
    }

    public Collection<Cliente> getClientes() {
        return clientes;
    }
}
```

```

public void agregarCuenta (Cuenta c) {
    cuentas.add(c);
}

public void agregarCliente (Cliente c) {
    clientes.add(c);
}

public void eliminarCuenta (Cuenta c) {
    cuentas.remove(c);
}

public void eliminarCliente (Cliente c) {
    clientes.remove(c);
}
}

```

Si bien es uno de los patrones más típicos y fue incluido en [GoF 1994], no está exento de problemas, al punto que muchos lo consideran un antipatrón.

Entre los problemas que suele presentar el patrón Singleton se encuentran:

- Restricciones para la herencia: una clase con constructor privado no puede tener descendientes (al menos en Java y muchos otros lenguajes), lo cual es bastante razonable si se piensa un poco.
- Más restricciones para la herencia: un atributo de clase va a ser compartido por todas las clases de la familia, lo cual no es siempre lo que se quiere; además, los métodos de clase no se heredan ni se redefinen.
- Problemas de concurrencia: el uso del atributo *instancia* por más de una tarea a la vez puede traer problemas.
- Brinda un acceso global al objeto en cuestión, y esto incrementa el acoplamiento de las clases y disminuye la calidad del diseño.
- Rompe con el principio de única responsabilidad, ya que ahora la clase tiene tanto su responsabilidad específica como la de controlar la creación de objetos.

Estos problemas hacen que debamos replantearnos si utilizar Singleton toda vez que parece a priori una buena idea. Y aunque hay soluciones a la mayor parte de estos problemas, pueden ser muy costosas en claridad de código y desempeño en ejecución.

La recomendación es que se use únicamente si las instanciaciones frecuentes de una clase que conceptualmente tiene una sola instancia provocan problemas de desempeño o de integridad de datos, o cuando los objetos contienen estado que se puede compartir sin problema. Un caso particular de esto son los objetos que no guardan estado interno. Por eso se lo suele usar en conjunto con otros patrones, como Strategy, State y Command.

Por ejemplo, ¿tiene sentido en la aplicación bancaria? Todo depende del uso que debamos hacer del objeto *banco*. Si el uso es muy extendido desde otras clases del sistema, tal vez sí. Sin embargo, si no es tan extendido, es probable que en esos casos pueda simplemente pasarse como argumento a los métodos que lo necesiten. Por otro lado, así planteado, es una especie de súper-objeto accesible desde todos lados, lo cual se lleva bastante mal con la modularización que plantea la POO.

Resumen de patrones

Como ya dijimos, el uso de patrones facilita la tarea de diseño por utilizar soluciones ya conocidas, que no necesitan nuevo análisis, lo que a su vez simplifica la implementación. Si repasamos los patrones ya vistos nos daremos cuenta de que las soluciones que plantean no son las más intuitivas y directas, sino modelos bastante elaborados que han mostrado sus bondades.

A continuación, un cuadro resumen de los principales patrones estudiados:

Patrón	Otros nombres	Contexto o problema	Contras y limitaciones
Iterator	Iterador, Cursor	Necesitamos implementar recorridos en colecciones, con bajo acoplamiento y en forma externa a la propia colección	Exige crear una clase por colección
Template Method	Marco de aplicación, Application Framework	Tenemos un marco de algoritmo y deseamos implementar alternativas para ciertas partes del mismo	Tiene la rigidez que le impone la herencia; Cada variante del algoritmo necesita una clase nueva
Observer	Observador, Publicador-suscriptor	Necesitamos que un objeto sea notificado cuando cambia el estado de otro	No es sencillo implementar notificaciones en base a temas de interés
Factory Method	Método fábrica, Constructor virtual	Necesitamos encapsular la creación de objetos	Exige la definición de métodos especiales
Singleton		Necesitamos limitar a una la cantidad de instancias de una clase, brindando un punto de acceso global	Expone un objeto global; No admite herencia; Frágil en situaciones de concurrencia
Factory Class	Clase fábrica	Necesitamos encapsular la creación de objetos	Exige la definición de métodos especiales
Strategy	Estrategia	Deseamos poder elegir el algoritmo a usar en tiempo de ejecución	Cada variante del algoritmo necesita una clase nueva
Decorator	Decorador, Wrapper	Deseamos poder agregar comportamiento a métodos en forma dinámica	Complejidad de implementación
Command	Orden, Acción, Functor	Deseamos tratar a un método como si fuese un objeto	Debe definirse interfaces distintas para distintas firmas de métodos
State	Estado	Necesitamos que un objeto cambie su comportamiento al cambiar de estado	Si se desea una máquina de estados deben definirse reglas que le agregan complejidad
Composite	Contenedor, Compuesto, Agregado	Necesitamos tratar de la misma manera a los objetos contenidos y sus contenedores	Normalmente necesitamos más funcionalidad en contenedores que en contenidos
Adapter	Adaptador, Wrapper	Necesitamos una interfaz diferente para una clase	Tiende a usarse mal, agregando comportamiento en la clase adaptadora
Visitor	Double Dispatch, Doble despacho	Deseamos que el polimorfismo facilite implementar lógica que depende de dos jerarquías de clases.	Exige modificar dos jerarquías de clases en paralelo, lo cual limita la extensibilidad.
Chain of Responsibility	Cadena de Responsabilidad	Deseamos aplicar distintos algoritmos en forma sucesiva para resolver un problema	Cada variante del algoritmo necesita una clase nueva
Facade	Fachada	Deseamos brindar un acceso simplificado y unificado a un conjunto de funcionalidades de un subsistema	Cuando los servicios de un subsistema son muchos, la clase resultante es muy compleja
Proxy	Surrogate, Sucedáneo, Intermediario	Deseamos controlar el acceso a un objeto mediante un intermediario	Puede confundir al cliente

Como es lógico, los patrones de diseño no se utilizan en forma aislada, sino formando parte de aplicaciones. En algunos casos se utilizan también en conjunto con otros patrones. Ya lo vimos en el caso de State o Strategy junto con Singleton, en el de Command con Composite para implementar macros y el de Proxy con Observer para notificaciones en el caso de proxy

remoto con caché. Otro uso de patrones compuestos es en la construcción de patrones macro, como en el caso de MVC.

Los patrones de diseño son una herramienta muy interesante, pero que tiene sus detractores. Lo que éstos opinan es que hay personas que usan patrones por todos lados, como una solución en busca de un problema. Para evitar este inconveniente, lo mejor es tratar de diseñar y refactorizar aplicando principios y no patrones. Poco a poco, los patrones irán apareciendo, y en ese caso sería bueno cambiar nombres de clases, métodos y demás, para orientar al lector del código a que vea el uso de uno u otro patrón. Al fin y al cabo, los patrones no se inventaron, sino que se descubrieron y evolucionaron.

Ejercicios propuestos

Contenido todavía no disponible en esta versión del libro.

Recapitulación

En este capítulo dimos una pasada rápida por el tema de diseño orientado a objetos, sus principios y algunos patrones, con bastantes ejemplos de implementación, como corresponde en un libro de programación.

En el próximo, vamos a volver a nuestras prácticas metodológicas, ya no mirando solamente al desarrollo como escritura de código, sino como una actividad ingenieril que analiza al software como producto.

11. Entre dos paradigmas

Contexto

En el marco de lo que venimos sosteniendo en este libro, hay dos principios que hemos enunciado y que no estamos cumpliendo del todo.

Por un lado, está la afirmación de que en POO todo se basa en un mecanismo de objetos y mensajes. Pero esto no es totalmente cierto hasta este punto del libro: los métodos – o implementación de las respuestas a los mensajes – no son objetos. Algo ha hecho C# con la definición de un tipo de datos nuevo, los delegados, que permiten trabajar con métodos como si fueran objetos de pleno derecho, pero eso no es así en los lenguajes en los que estamos trabajando.

La otra cuestión tiene que ver con que venimos sosteniendo que – gracias a la POO – podemos evitar la duplicación de código. Sin embargo, como veremos enseguida, hay algunas situaciones en las que la duplicación de código se mantiene.

Este capítulo trata sobre expresiones lambda y los objetos de tipo bloque de código, que van a venir en nuestra ayuda para resolver los problemas planteados.

A pesar de todo... código duplicado

El problema

Supongamos que hacemos un poco más complejo nuestro modelo de la aplicación bancaria, hasta llegar al diagrama de clases de la figura 11.1.

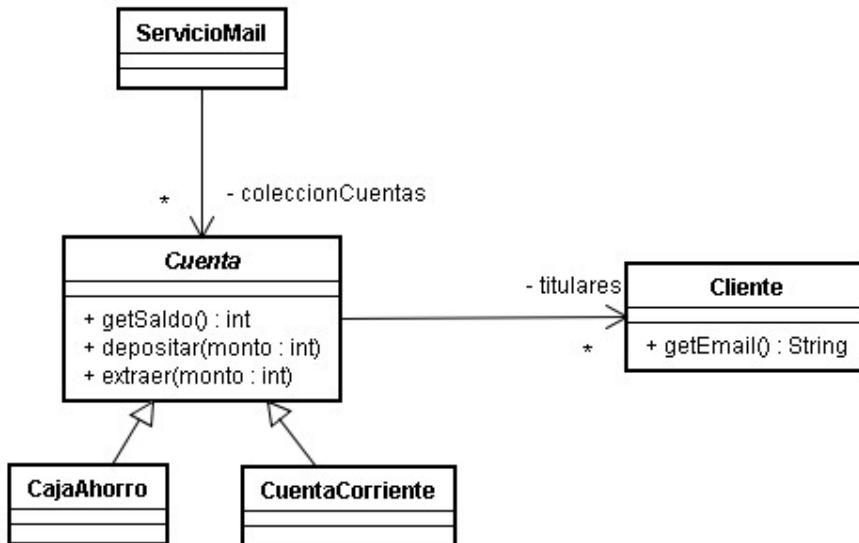


Figura 11.1 Diagrama de clases de la aplicación bancarias con clientes y servicios

Si, en este caso, decidísemos enviar correos electrónicos a los titulares de las cuentas con más de dos titulares y con más de 100.000 pesos de saldo, lo podríamos hacer, con los conocimientos que tenemos hasta ahora, con estos dos mensajes:

```
ServicioMail servicio = new ServicioMail(coleccionCuentas);
```

```

servicio.enviarMailCuentasMasDeNTitulares(3);
servicio.enviarMailCuentasPorSaldoMasQue(100000);

```

Y la implementación de los métodos en la clase *ServicioMail* sería:

```

public void enviarMailCuentasMasDeNTitulares (int desde) {
    for (Cuenta cuenta: colecciónCuentas) {
        if ( cuenta.getTitulares().size() >= desde ) {
            for (Cliente cliente: cuenta.getTitulares()) {
                enviarMail(cliente.getEmail());
            }
        }
    }
}

```

```

public void enviarMailCuentasPorSaldoMasQue (int saldoMinimo) {
    for (Cuenta cuenta: colecciónCuentas) {
        if (cuenta.getSaldo() >= saldoMinimo) {
            for (Cliente cliente: cuenta.getTitulares()) {
                enviarMail(cliente.getEmail());
            }
        }
    }
}

```

Acá tenemos mucho código repetido. En efecto, los dos métodos son idénticos en todo, salvo en el fragmento que verifica a los titulares de qué cuentas se les va a enviar el correo.

Sin embargo, con lo visto hasta ahora pareciera que no podemos evitar la repetición, ¿o sí?

Una solución posible

Sin embargo, con lo que ya sabemos, hay una posible solución. Por ejemplo, podríamos crear clases para encapsular el comportamiento que varía, siguiendo la propuesta del patrón Strategy.

Por ejemplo, si definimos la interfaz:

```

public interface CondicionCuenta {
    boolean test (Cuenta cuenta);
}

```

Luego podremos definir dos clases que implementen ese método *test*.

```

public class CondicionMasDe2Titulares implements CondicionCuenta {
    public boolean test(Cuenta cuenta) {
        return (cuenta.getTitulares().size() > 2);
    }
}

```

```

public class CondicionSaldoMayorQue100000 implements CondicionCuenta {
    public boolean test(Cuenta cuenta) {
        return (cuenta.getSaldo() >= 100000);
    }
}

```

```
}
```

Y a partir de allí definir un único método en *ServicioMail*:

```
public void enviarMailCuentasPorCondicion(CondicionCuenta condicion) {  
    for (Cuenta cuenta: colecciónCuentas) {  
        if (condicion.test(cuenta)) {  
            for (Cliente cliente: cuenta.getTitulares()) {  
                enviarMail(cliente.getEmail());  
            }  
        }  
    }  
}
```

Que para las dos primeras tareas que queríamos realizar se invocaría así:

```
servicio.enviarMailCuentasPorCondicion  
        (new CondicionMasDe2Titulares());  
servicio.enviarMailCuentasPorCondicion  
        (new CondicionSaldoMayorQue100000());
```

Ahora ya hemos minimizado el código repetido, al costo de declarar una clase para cada condición: cada clase encapsula un método.

Digresión

En C++ esto se puede resolver más fácilmente con punteros a funciones, que nos permiten pasar a las funciones como parámetros.

En C# esto se implementa con delegados, que son métodos que se pueden tratar como objetos.

Si queremos evitar declarar cada clase, Java nos permite usar un mecanismo que es el de la clase anónima, por la cual definimos una clase sin nombre que implementa una interfaz.

Definición:

Una **clase anónima** es una clase que no está asociada a un nombre, sino que sólo declara implementar una interfaz.

Por ejemplo, en el caso anterior, en vez de definir las dos clases y luego instanciarlas en la llamada a *enviarMailCuentasPorCondicion*, podríamos escribir ese código así:

```
servicio.enviarMailCuentasPorCondicion (new CondicionCuenta() {  
    public boolean test(Cuenta cuenta) {  
        return (cuenta.getTitulares().size() > 2);  
    }  
});  
  
servicio.enviarMailCuentasPorCondicion (new CondicionCuenta() {  
    public boolean test(Cuenta cuenta) {  
        return (cuenta.getSaldo() >= 100000);  
    }  
});
```

Aquí las clases quedan definidas al decir *new CondicionCuenta()* seguido del cuerpo de una clase. Recordemos que *CondicionCuenta* es una interfaz, así que lo que se está instanciando no es *CondicionCuenta* sino clases anónimas que implementan dicha interfaz.

Esto es útil si el único uso que le queremos dar a la clase es ese. Si no, conviene definir una clase e instanciarla cada vez que la necesitemos, como hicimos en la implementación anterior con el patrón Strategy.

Expresiones lambda

La última versión de los mensajes que envían correos están escritos de una forma tal que pareciera que estamos pasando métodos como argumentos⁵⁵. Esto ocurre porque estamos usando clases anónimas que implementan una interfaz funcional.

Definición:

Una **interfaz funcional** es una interfaz que tiene solamente un método.

El único problema de esta implementación es que no resulta demasiado clara para leer.

En su versión 8, Java nos provee una forma más legible de hacer lo mismo, mediante expresiones lambda.

Definición:

Una **expresión lambda** es una clase anónima basada en una interfaz funcional con una notación simplificada.

Usando expresiones lambda, las dos últimas invocaciones a *enviarMailCuentasPorCondicion* se escribirían así:

```
servicio.enviarMailCuentasPorCondicion(  
    (Cuenta cuenta) -> cuenta.getTitulares().size() > 2  
);  
  
servicio.enviarMailCuentasPorCondicion(  
    (Cuenta cuenta) -> cuenta.getSaldo() >= 100000  
);
```

Notemos que estamos tratando porciones de código como objetos que pueden ser pasados como argumentos.

Incluso podemos hacerlo más legible con la ayuda de genericidad. Hay una interfaz *java.util.function.Predicte* en Java 8 definida así:

```
public interface Predicte<T> {  
    boolean test (T t);  
}
```

Por lo tanto, si en vez de usar nuestra interfaz *CondicionCuenta*, usamos *Predicte*, nuestro método *enviarMailCuentasPorCondicion* quedaría así:

```
public void enviarMailCuentasPorCondicion(Predicte<Cuenta> condicion) {  
    for (Cuenta cuenta: colecciónCuentas) {  
        if (condicion.test(cuenta)) {  
            for (Cliente cliente: cuenta.getTitulares()) {  
                enviarMail(cliente.getEmail());  
            }  
        }  
    }  
}
```

⁵⁵ En realidad, estamos enviando como argumento un objeto que encapsula un método.

```
        }
    }
}
```

Y como el compilador de Java podría inferir el tipo genérico, el código de invocación sería aún más sencillo:

```
servicio.enviarMailCuentasPorCondicion(
    cuenta -> cuenta.getTitulares().size() > 2
);
servicio.enviarMailCuentasPorCondicion(
    cuenta -> cuenta.getSaldo() >= 100000
);
```

Si bien hicimos un ejemplo sencillo, las funciones lambda funcionan en todos los casos que estemos usando interfaces funcionales.

Por otro lado, Java 8 tiene varias interfaces funcionales predefinidas y varias posibilidades más, pero vamos a dejar el tema aquí porque excede la pretensión con la que escribimos este libro.

La visión de Smalltalk: bloques de código que son objetos

Smalltalk también tiene métodos anónimos, que este lenguaje denomina **bloques de código** y que, formalmente hablando son bloques de código de alcance léxico⁵⁶.

Hasta este punto del libro, esto nos ha permitido trabajar en Smalltalk con ciclos y condicionales que no necesitan construcciones del lenguaje por fuera del modelo de objetos y mensajes.

Por ejemplo, si escribimos:

```
(a > b) ifTrue: [Transcript show: a printString].
```

Estamos enviando el mensaje *ifTrue* a una instancia de *Boolean*, enviando como parámetro un bloque de código.

Los bloques de código son, entonces, objetos que admiten determinados mensajes. Por ejemplo, el mensaje *value* nos da el valor de un bloque. Si escribimos:

```
[ 2 + 2 ] value.
```

El resultado es el número 4.

También sabemos que los bloques de código permiten parámetros. Así lo hicimos cuando trabajamos colecciones o capturamos excepciones. Un uso más sencillo sería el uso de *value* con parámetros:

```
[:x | x + 2] value: 5.
```

Ahora el resultado es 7.

Existe también la posibilidad de usar varios parámetros, como en:

```
[:x :y | x + y] value: 5 value: 7.
```

Que tiene por resultado 12.

⁵⁶ En inglés, “lexically scoped block closures”.

Otro uso de los bloques, como decíamos, es el trabajo con excepciones, como en :

```
on: ClaseExcepcion do: bloque.
```

O incluso una variante que no hemos usado, que abre un proceso aparte en el caso que se lance una excepción:

```
on: ClaseExcepcion fork: otroProceso.
```

Hay varios usos más en concurrencia y en colecciones.

Los bloques pueden tener variables locales y acceder también a objetos que estén fuera de su ámbito.

Incluso mediante bloques podemos acceder al contexto de ejecución, por ejemplo mediante las clases *CompiledMethod*, *MethodContext*, etc., que se pueden consultar en la documentación oficial de Smalltalk o de Pharo.

Nos hemos mantenido en los ejemplos simples para no complicar más el libro. Lo importante es recordar que en Smalltalk se puede trabajar con bloques como las expresiones lambda de Java, recordando que en Smalltalk los bloques son objetos de pleno derecho, con comportamiento, identidad, y también pueden almacenar estado.

¿Qué tiene que ver esto con la programación funcional?

¿Es todo esto programación funcional, como se oye y lee por allí?

No, no y no. La programación funcional es un paradigma distinto al de POO, que nos abstendremos de analizar aquí, en un libro de POO, pero que excede en mucho al uso de construcciones como las funciones como objetos, los bloques de código o las expresiones lambda. Algunas características inherentes a la programación funcional son:

- Transparencia referencial
- Funciones de orden superior
- Funciones puras sin efectos secundarios y de resultado constante

Lo único que hemos logrado en este capítulo es mostrar la ventaja de trabajar con código o métodos como si fueran objetos y de ese modo evitar repetición de código y hacerlo más claro. Todo parecido con el paradigma funcional es sólo eso: un parecido.

Ejercicios propuestos

Contenido todavía no disponible en esta versión del libro.

Recapitulación

Contenido todavía no disponible en esta versión del libro.

12. Cierre metodológico: mirando al software como producto

Contenido todavía no disponible en esta versión del libro.

13. Apéndice A: Elementos básicos de Smalltalk

Contenido todavía no disponible en esta versión del libro.

14. Apéndice B: Elementos básicos de Java

Contenido todavía no disponible en esta versión del libro.

15. Apéndice C: Algunas objeciones a las prácticas y herramientas de este libro

UML

La crítica ágil

Contenido todavía no disponible en esta versión del libro.

NoUML

Contenido todavía no disponible en esta versión del libro.

UML en la pizarra

Contenido todavía no disponible en esta versión del libro.

TDD

Contenido todavía no disponible en esta versión del libro.

Bibliografía

[Beck 2000] “Extreme programming explained: embrace change”, Kent Beck, Addison-Wesley Professional, 2000.

[BrodieStonebraker 1995] “Migrating Legacy Systems”, M. Brodie, M. Stonebraker, Morgan Kaufman, 1995.

[Brooks 1987] “No Silver Bullet: Essence and Accidents of Software Engineering”, Fred Brooks Jr., IEEE Computer 20.4 (1987): 10-19. 1987.

[Deursen 2001] “Refactoring test code”, Arie Van Deursen, Leon Moonen, Alex van den Bergh, Gerard Kok, Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001) (pp. 92-95), 2001.

[Eckel 2003] “Thinking in JAVA”, Bruce Eckel, Prentice Hall Professional, 2003.

[Feathers 2005] "Working Effectively with Legacy Software", Michael Feathers, Prentice Hall - Pearson Education, Inc., 2005.

[Fontela 2008] “Orientación a objetos: diseño y programación”, Carlos Fontela, Nueva Librería, 2008.

[Fontela 2010] “Orientación a objetos con Java y UML - 2da. edición”, Carlos Fontela, Nueva Librería, 2010.

[Fontela 2011] “UML Modelado de Software para Profesionales”, Carlos Fontela, Alfaomega 2011; Marcombo 2012.

[FontelaTesis 2013] “Cobertura entre pruebas a distintos niveles para refactorizaciones más seguras”, Carlos Fontela, tesis de maestría en Universidad Nacional de La Plata, 2013.

[FontelaASSE 2013] “Hacia un enfoque metodológico de cobertura múltiple para refactorizaciones más seguras”, Carlos Fontela, Alejandra Garrido, Andrés Lange, Proceedings of the 14th Argentine Symposium on Software Engineering (ASSE 2013), Córdoba, Argentina, 2013.

[Fowler 1999] “Refactoring: improving the design of existing code”, Martin Fowler, Kent Beck, Addison-Wesley Professional, 1999.

[Fowler 2002] “Patterns of enterprise application architecture”, Martin Fowler, Addison-Wesley Longman Publishing Co., Inc., 2002.

[Fowler 2004] “UML distilled: a brief guide to the standard object modeling language”, Martin Fowler, Addison-Wesley Professional, 2004.

[GoF 1994] “Design Patterns: Elements of reusable object-oriented software”, Erich Gamma, John Vlissides, Ralph Johnson, Richard Helm, Addison-Wesley, 1994.

[Ingalls 1986] “A Simple Technique for Handling Multiple Polymorphism”, Daniel Ingalls, Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages and Applications, pages 347–349, November 1986. Printed as SIGPLAN Notices, 21(11).

[Kerievsky 2005] “Refactoring to patterns”, Joshua Kerievsky, Pearson Deutschland GmbH, 2005.

[LippertRoock 2006] “Refactoring in large software projects: performing complex restructurings successfully”, Martin Lippert, Stefan Roock, John Wiley & Sons, 2006.

- [Liskov 1988] “Keynote address-data abstraction and hierarchy”, Barbara Liskov, ACM Sigplan Notices, 23(5), 17-34, 1988.
- [Martin 2003] “UML for Java programmers”, Robert Martin, Prentice Hall PTR, 2003.
- [Meszaros 2007] “xUnit test patterns: Refactoring test code”, Gerard Meszaros, Pearson Education, 2007.
- [Meyer 1988] “Object-oriented software construction”, Bertrand Meyer, Prentice Hall, 1988.
- [Meyer 1993] “Towards an object-oriented curriculum”, Bertrand Meyer, Journal of Object-Oriented Programming, 6(2):76–81, May 1993.
- [Meyer 1994] “Reusable software : the base object-oriented component libraries”, Bertrand Meyer, Hemel Hempstead: Prentice-Hall, 1994.
- [Mugridge 2005] “Fit for developing software: framework for integrated tests”, Rick Mugridge, Ward Cunningham, Pearson Education, 2005.
- [Opdyke 1992] “Refactoring object-oriented frameworks”, William F. Opdyke, tesis de doctorado en University of Illinois at Urbana-Champaign, 1992.
- [PageJones 2000] “Fundamentals of object-oriented design in UML”, Meilir Page-Jones, Larry Constantine, Addison-Wesley Professional, 2000.
- [Pipka 2002] “Refactoring in a “test first”-world”, Jens Uwe Pipka, Proceedings Third International Conference in eXtreme Programming and Flexible Processes in Software Engineering, 2002.
- [Roberts 1999] “Practical analysis for refactoring”, Don Roberts, Ralph Johnson, University of Illinois at Urbana-Champaign, 1999.