CAPITULO 13. DISEÑAR APLICACIONES

Los pasos iniciales en el desarrollo de un sistema de software se conocen, generalmente, como las etapas de *análisis y diseño:* analizamos el problema y luego diseñamos una solución. El primer paso del diseño es de un nivel más alto que el diseño de clases que tratamos en el Capítulo 7: pensamos qué clases se deben crear para resolver nuestro problema y cómo deben interactuar exactamente. Una vez que tenemos una solución para este problema podemos continuar con el diseño de las clases individuales y comenzar a pensar en su implementación.

13.1 Análisis y diseño.

Usaremos un método bastante simple para orientar la tarea, que funcionará bien en problemas relativamente pequeños. Para descubrir las clases iniciales usamos el *método verbo/sustantivo* y luego usaremos *tarjetas CRC* para llevar adelante el diseño inicial de la aplicación.

13.1.1 El método verbo/sustantivo

Este método trata de identificar las clases y los objetos, y las asociaciones e interacciones entre ellos.

A partir de estos conceptos del lenguaje natural podemos ver que, en la descripción de un problema de programación los nombres generalmente se corresponden con las clases y con los objetos mientras que los verbos se corresponden con las cosas que hacen esos objetos, es decir, con los métodos. No precisamos una descripción más larga para ilustrar esta técnica; generalmente, la descripción necesita contener sólo unos pocos párrafos.

13.1.3 Descubrir clases

El primer paso en la identificación de las clases es recorrer la descripción y marcar todos los sustantivos y verbos que aparecen en el texto.

Los sustantivos que identificamos nos dan una primera aproximación de las clases de nuestro sistema. En un primer paso, podemos pensar en una clase por cada sustantivo. Este no es un método exacto, más adelante podríamos encontrar que necesitamos algunas clases adicionales o que algunos de nuestros sustantivos no son necesarios, pero esta es una cuestión que controlaremos más adelante. Es importante no excluir ninguno de los sustantivos de la descripción ya que todavía no tenemos información suficiente como para tomar una buena decisión.

13.1.4 Usar tarjetas CRC

El próximo paso en nuestro proceso de diseño es trabajar con las interacc iones entre nuestras clases; para hacerlo usaremos un método denominado *tarjetas CRC*.

La denominación CRC se basa en la tríada Clase/Responsabilidades/Colaboradores. La idea consiste en tomar tarjetas de cartulina y usar una tarjeta para cada clase. Es importa nte para esta actividad usar tarj etas rea les y fi sicas y no una computadora o una simple hoja de papel.
Cada tarjeta está dividida en tres áreas: en el área de la izquierda se escribe el nombre de la clase; en el área ubicada debajo de la ante rior se escriben las responsabilidades de dicha clase y en el área de la derecha se escriben los colaboradores de esta clase

13.1.5 Escenarios

Con el objetivo de deducir las interacciones necesarias entre las clases de nuestro sistema jugaremos con varios ***escenarios***. Un escenario es un ejemplo de la actividad que el sistema tiene que llevar adelante o proporcionar. En algunas ocasiones, se hace referencia a los escenarios como *casos de uso*. No usamos este término aquí porque generalmente se usa para denotar una manera más formal de la descripción de los escenarios.

El juego con los escenarios resulta mejor cuando se realiza en grupo. A cada miembro del grupo se le asigna una clase (o un número pequeño de clases) y esa persona cumple su rol diciendo en voz alta lo que la clase está actualmente haciendo. Mientras se juega con un escenario, los miembros registran en las tarjetas CRC cada cosa que se descubre

sobre la clase en acción: cuáles deben ser sus responsabilidades y qué otras clases colaboran con ella.

El trabajo con los escenarios funciona mejor cuando un grupo de personas se sienta alrededor de una mesa y mueve las tarjetas sobre ella. Las tarjetas que cooperan cercanamente se pueden ubicar juntas y más próximas para dar alguna impresión sobre el grado de acoplamiento del sistema.

Es muy común que los princIpiantes tomen atajos y no cuestionen y registren cada deta lle de la ejecución de un escenario. ¡Esto es peligroso! Pasan rápidamente a desarrollar el sistema en Java y si han quedado sin responder algunos detalles, es muy probable que las decisiones *ad hoc* se tomen en tiempo de implementación, lo que más tarde producirá malas elecciones.
También es común que los principiantes olviden algunos escenarios. El olvido de una parte del sistema antes de iniciar el di seño de las clases y su implementación puede acarrear una gran cantidad de trabajo más adelante, cuando el sistema esté parcia lmente implementado y deba ser modificado.

13.2 Diseño de Clases

Ahora es el momento del sigui ente gran paso: convertir las tarjetas CRC en clases de Java. Durante el ejercicio con las tarjetas CRC se logra una buena comprensión de la estructura de la aplicación y de la manera en que cooperan las clases para resolver las di stintas tareas del programa. Al atravesar los diferentes casos surge la necesidad de introducir clases adicionales (que generalmente se trata de clases que representan estructuras internas de datos) y también puede ocurrir que quede alguna tarjeta que representa una clase que jamás se usó, en cuyo caso, se puede eliminar.
A e ta altura, el reconocimiento de las clases que se deben implementar es trivial: las tarj etas nos muestran el conjunto completo de las clases que se necesitan. La decisión de la interfaz de cada clase, es decir, el .::onjunto de métodos públicos que debe tener una clase, es un poco más dificil, pero hemos dado un importante paso hacia adelante que es bueno. Si el juego con los escenarios estuvo bien hecho, las responsabi lidades anotadas en cada clase describen los métodos públicos de di chas clases y quizás también, algunos campos de instancia. Se deben evaluar las responsabilidades de cada clase de acuerdo con los principios del di seño de clases di scutidos en el Capítulo 7: di seño dirigido por responsabilidades, acoplamiento y cohes ión.

13.2.1 Diseñar interfaces de clases

Antes de comenzar a escribir el código de nue tra aplicación en Java, podemo usar nuevamente las taljetas para avanzar otro paso más en el diseño, mediante la traducción de las descripciones informales de las invocaciones a los métodos y el agregado
de los parámetros necesarios.
Para llegar a una descripción más formal podemos jugar nuevamente con los distintos escenarios, pero esta vez, en términos de llamadas a métodos, parámetros y va lores de retorno. La lógica y la estructura de la aplicación no debiera cambiar más, pero tratamos de anotar la información completa de las signaturas de los métodos y los campos de instancia. Llevamos a cabo esta tarea con un nuevo conjunto de tarjetas.

Una vez que se resolvió el último ejercicio propuesto, resulta fácil escribir la interfaz de cada clase. Podemos traducir las tarjetas directamente a Java. Típicamente, se deben crear todas las clases y se deben incluir *métodos stubs* para todos los métodos públicos que se deban escribir. Un *método stub* es un método que tiene la signatura correcta y el cuerpo vacío.

A muchos estudiantes les resulta tedioso realizar esta tarea detalladamente pero, al finalizar el proyecto, apreciarán el valor de estas actividades. Muchos equipos de desarrollo de software han enfatizado que el tiempo que se ahorra en la etapa de diseño, muchas veces se emplea en la corrección de errores u omisiones que no se descubrieron con la anterioridad suficiente

El diseño inicial es una de las partes más importantes del proyecto. El tiempo que se empleará en el diseño se debe planificar, como mínimo, similar al tiempo que se empleará en la implementación. El diseño de una aplicación no es anterior a la programación, ¡es la parte más importante de la programación!

Los errores del código propiamente dichos pueden solucionarse de manera bastante fácil. Los errores del diseño pueden ser, en el mejor de los casos, muy caros de corregir y en el peor de los casos, fatales para la aplicación una vez terminada. En algunos casos desafortunados, pueden ser prácticamente incorregibles (hay que parar y comenzar todo de nuevo).

13.2.2 Diseño de la interfaz de usuario

Hasta ahora, hemos dejado fuera de la discusión, el diseño de la interfaz de usuari0.
En este punto, tenemos que decidir detalladamente lo que los usuarios verán en la pantalla y las maneras en que interactuarán con nuestro sistema.
En una aplicación bien diseñada, la interfaz de usuario es muy independiente de la lógica subyacente de la aplicación, por lo que puede diseñarse independientemente del diseño de la estructura de cIases del resto del proyecto.

13.3 Documentación

Después de identificar las cIases y sus interfaces, y antes de comenzar con la implementación de los métodos de una cIase, se debe documentar la interfaz. Esto implica escribir un comentario de clase y comentarios de métodos en cada cIase del proyecto. Los comentarios deben ser descriptivos con la cantidad de detalle suficiente como para que se puedan identificar los principales propósitos de cada cIase y de cada método.

Al igual que el análisis y el diseño, la documentación es un área frecuentemente desdeñada por los principiantes. No es fácil que los programadores inexpertos vean los motivos por los que la documentación es tan importante. La razón es que los programadores inexpertos generalmente trabajan sobre proyectos que tienen sólo unas pocas clases y que se escriben en un período de unas pocas semanas o meses. Un programador puede contar con documentación pobre cuando trabaja sobre estos miniproyectos.

Sin embargo, aún los programadores experimentados frecuentemente se preguntan cómo es posible escribir la documentación antes que la implementación. Esto es así porque fallan al apreciar que la buena documentación hace foco en cuestiones de alto nivel, tales como qué hace una cIase o un método, antes que en cuestiones de bajo nivel tales como exactamente cómo lo hace. Esto es, generalmente, un síntoma de ver a la implementación como más importante que el diseño.

Si un desarrollador de software quiere avanzar hacia problemas más interesantes y comenzar a trabajar profesionalmente en aplicaciones de la vida real, no es poco usual que trabaje con docenas de personas sobre una aplicación durante varios años. La solución *ad hoc* de sólo «tener la documentación en su cabeza» no funcionará nunca más.

13.4 **Cooperación**

El desarrollo de software, generalmente, se hace en equipo. Un abordaje puro, orientado a objetos, proporciona un fuerte soporte al trabajo en equipo porque permite la separación del problema en componentes bajamente acopladas (clases) que pueden ser implementadas independjentemente.
Aunque el trabajo de diseño inicial es mejor cuando se realiza en grupo, llega el momento de dividirlo. Si la definición de las interfaces de las clases y la documentación está bien hecha, debe ser posible implementar las clases de manera independiente. Se pueden as ignar las clases a los programadores, quienes pueden trabajar a solas o en parejas.

13.5 Prototipos

En lugar de diseñar y luego construir la aplicación completa en un paso enorme, se usan los prototipos para investigar partes de un sistema.

Un prototipo es una versión de la aplicación en la que se simula una parte de ella, en vías de experimentar con las restantes partes. Por ejemplo, se podría implementar un prototipo para probar una interfaz gráfica de usuario. En este caso, la lógica de la apli cación podría no estar implementada apropiadamente y en cambio, podríamos escribir implementaciones simples de aquellos métodos que simulan la tarea.

Los prototipos nos permiten desarrollar rápidamente

un sistema ejecutable (pero no totalmente funcional), de modo que podamos investigar en la práctica di sti ntas partes de la aplicación. Sirve para proporcionar mayor compresión. en una etapa temprana del proceso de desarrollo. de la manera en que funcionará el sistema.

Los prototipos también son útiles para las clases independientes y ayudan al equipo en el proceso de desarrollo. Frecuentemente, cuando diferentes miembros de un equipo trabajan sobre diferentes clases, no todas las clases insumen la misma cantidad de tiempo en terminarse. En algunos casos, una clase dejada de lado, puede retrasar la continuación del desarrollo y la prueba de otras clases. En esos casos puede ser beneficioso escribir una clase prototipo. El prototipo tiene implementaciones de todos los métodos pero en lugar de contener implementaciones fina les y completas, el prototipo sólo simula la func ionalidad. La escritura de un prototipo puede ser posible rápidamente y el desarrollo de las clases cliente puede continuar usando el prototipo hasta que la clase se haya implementado en su totalidad.

un beneficio adicional de los prototipos es que puede brindar a los desarrolladores conocimientos profundos de cuestiones y problemas que no fueron considerados en un estado anterior.

13.6 Crecimiento del software

Para la construcción del software, existen varios modelos que se pueden aplicar, uno de los más comúnmente utilizados es el *modelo de cascada*, denominado así porque la actividad progresa de un nivel al siguiente, tal como el agua de una cascada, y no hay vueltas atrás.

13.6.1 Modelo de cascada

En el modelo de cascada, las di stintas fases del desarrollo del software se realizan siguiendo una secuencia determinada:
• análi sis del problema
• diseño del software
• implementación de los componentes del software
• prueba unitaria
• prueba integral
• entrega del sistema al cliente

Si se presenta algún problema en cualquiera de las fases, deberíamos regresar a la fase anterior para solucionarlo; por ejemplo, si alguna prueba demuestra la existencia de un fallo regresamos a la implementación, pero no existe ningún plan para revisitar las fases anteriores.

Este es, probablemente, el modelo más tradicional y conservador de desarrollo de software y se ha usado extensamente durante largo tiempo. Sin embargo, a lo largo de los años, se han descubierto numerosos inconvenientes en este modelo. Dos de las principales grietas son que asume que los desarrolladores comprenden por completo y detalladamente la funcionalidad del sistema desde el principio y que el sistema no va a cambiar con posterioridad a su entrega al cliente.

En la práctica, ambas presunciones, generalmente, no son ciertas. Es muy común que el diseño de la funciona lidad de un software no sea perfecto desde el principio, frecuentemente porque el cliente, quien conoce el dominio del problema, no sabe mucho de computación y los ingenieros de software, quienes saben cómo programar, tienen sólo un conocimiento limitado del dominio del problema.

13.6.2 Desarrollo iterativo

Una solución posible a los problemas que acarrea el modelo de cascada es el uso temprano de prototipos y la interacción frecuente con el cliente durante el proceso de desarrollo. Se construyen los prototipos de los sistemas, que no hacen demasiado pero que

nos dan una impresión de cómo se podría presentar el sistema y de lo que podría hacer, y regularmente, los clientes realizan comentarios sobre el diseño y la funcionalidad. Esta solución conduce a un proceso más circular que el modelo de cascada, el desarrollo del software se retroalimenta pasando varias veces por el circu ito análisis-di eñoimplementación de prototipo-cliente.
Otro enfoque captura la noción de que un buen software no se diseña sino que *crece.* La idea subyacente es diseñar inicialmente un sistema pequeño y prolijo, y ponerlo en funcionamiento para que pueda ser usado por usuarios finales. Luego, se van agregando gradualmente las características adicionales de una manera controlada (el software crece) y se alcanzan estados «finales» repetidamente y con bastante frecuencia (es decir, estados en los que el software es completamente usable y puede ser entregado a los clientes).
En realidad, el crecimiento del software no se contradice con el diseño del software; se diseña cuidadosamente cada etapa de crecimiento. Lo que se trata de hacer es no diseñar el sistema completo y correcto desde el inicio, aún más: ¡la noción de un sistema de software completo no existe en absoluto!
El modelo de cascada tradicional tiene como objetivo principal la liberación de un sistema completo. El modelo de crecimiento del software asume que no existen los sistemas completos que se usan indefinidamente y sin modificaciones; sólo hay dos cosas que le pueden ocurrir a un sistema de software: es continuamente mejorado y adaptado o desaparecerá.

En consecuencia, ciertas tareas y habilidades cobran mucha más importancia de la que podrían tener en el modelo de cascada. El mantenimiento del software, la lectura de código (en lugar de sólo su escritura), el diseño preparado para la extensibilidad, la documentación, la codificación que apunta a la legibilidad y muchas otras cuestiones que hemos mencionado en este libro resultan importantes a partir del hecho de que sabemos que vendrán otros después de nosotros que tendrán que adaptar y extender nuestro código.
La visión de una pieza de software como algo que continuamente crece, cambia y se adapta en lugar de ser una pieza estática de texto que se escribe y se preserva como una novela, determina nuestra visión sobre cómo debe escribirse un código de buena ca lidad. Todas las técnicas que hemos di scutido a lo largo de este libro apuntan a esto.

13.7 Usar patrones de diseño

A medida que nos volvemos más expertos y diseñamos sistemas de software de mayor envergadura, la implementación de las clases deja de ser el problema más dificultoso. La estructura del sistema, la complejidad de las relaciones entre las clases, se vuelve más complicada de diseñar y de comprender que el código de las clases individuales. Es lógico que tratemos de alcanzar los mismos objetivos para las estructuras de las clases que los que planteamos para el código: queremos reutilizar buena parte de nuestro trabajo y queremos permitir que otros comprendan lo que hemos hecho.
A nivel de las estructuras de clases, ambos objetivos se pueden lograr usando *patrones de diseño.*
Un patrón de diseño describe un problema común, que ocurre regularmente en el desarrollo del software y luego describe una solución general del problema que se puede usar en varios contextos diferentes. La solución de los patrones de di seño de software consiste, típicamente, en la descripción de un conjunto de clases y sus respectivas interacciones.
Los patrones de diseño colaboran en nuestra tarea de dos maneras. Primero, documentan buenas soluciones a problemas planteados, por lo tanto, estas soluciones se pueden reutilizar más adelante en problemas similares. En este caso, la reutilización no es a nivel código sino a nivel estructura de clases.
Segundo, los patrones de diseño tienen nombres y de esta manera establecen un vocabulario que ayuda a los diseñadores de software a hablar sobre sus diseños. Cuando los diseñadores experimentados discuten sobre la estructura de una aplicación, uno de ellos podría decir «Creo que aquí deberíamos usar un Singleton». Singleton es el nombre de un patrón de diseño ampliamente conocido por lo que si ambos diseñadores están familiarizados con este patrón, serán capaces de hablar sobre él a ese nivel, ahorrándose explicaciones de muchos detalles. De esta manera, el patrón de lenguaje introducido por los patrones de diseño comúnmente conocidos introduce otro nivel de abstracción, uno que nos permite sobrellevar la complejidad en sistemas cada vez más complejos.

13.7.1 Estructura de un patrón

Las descripciones de los patrones se registran, generalmente, mediante una plantilla que contiene un mínimo de información. La descripción de un patrón no sólo contiene información sobre la estructura de algunas clases sino que también incluye una descripción del problema(s) que este patrón resuelve y argumentos a favor o en contra del uso del patrón.
La descripción de un patrón incluye como mínimo:
• un *nombre* que se puede utilizar para hablar sobre el patrón convenientemente;
• una descripción del **problema** que resuelve el patrón (frecuentemente dividido en secciones como intento, motivación, pertinencia);
• una descripción de la **solución** (frecuentemente se describe la estructura, los participantes y los colaboradores);
• las **consecuencias** del uso del patrón, incluyendo los resultados y lo que se deja de lado.

13.7.2 Decorador

El patrón *Decorador* trata el problema de agregar funcionalidad a un objeto que ya existe. Asumimos que queremos un objeto que responda a las mismas llamadas a método (es decir, que tiene la misma interfaz) pero con un comportamiento adicional o alterado. También quisiéramos agregarlo a la interfaz existente.
Un camino sería utilizando herencia: una subclase puede sobrescribir la implementación de métodos y agregar métodos adicionales. Pero el uso de la herencia produce una solución estática: una vez que se crean los objetos no pueden cambiar su comportamiento.
Una solución más dinámica es el uso de un objeto Decorador. El Decorador es un objeto que encapsula un objeto existente y que puede usarse en lugar del origina l (generalmente implementa la misma interfaz). Luego, los clientes pueden comunicarse con el Decorador en lugar de hacerlo directamente con el objeto origina l (s in necesidad de conocer esta sustitución). El Decorador pasa las llamadas a método al objeto encapsulado pero puede llevar a cabo acciones adicionales. Podemos encontrar un ejemplo en la biblioteca de entrada/sa lida de Java donde se usa un Buffe redReader como un Decorador de un Reader (Figura 13.2). El BufferedReader implementa la misma interfaz que un Reader y se puede usar en lugar de éste que no utiliza un buffer, pero le agrega el comportamiento básico del Reader. En contraste con el uso de la herencia, los decoradores se pueden agregar a objetos ya existentes.

13.7.3 Singleton

Una situación común en muchos programas es la de tener un objeto del que debe existir sólo una instancia. Por ejemplo, en nuestro juego *world-oj:u/UI* queremos contar sólo con un único analizador. Si escribimos un entorno de desarrollo de software podríamos querer un único compilador o un único depurador.

El patrón *Singlelon* asegura que se creará una única instancia de una clase y que ésta proporcionará acceso unificado a la misma. En Java, se puede definir un Singleton mediante un constructor privado. Esto asegura que no pueda ser invocado fuera de la clase y por lo tanto, las clases cliente no pueden crear nuevas instancias. Podemos luego escribir código en la clase Singleton propiamente dicha para crear una única instancia y ofrecer acceso a ella

En este patrón:

• El constructor es privado, por lo que las instancias se pueden crear sólo mediante la clase propiamente dicha y tiene que ser en la parte estática de la clase (inicializaciones de campos estáticos o de métodos estáticos) ya que no existirá ninguna otra instancia.
• Se declara e inicializa un campo estático y privado con la única instancia del analizador.
• Se define el método estático getInstancia para proporcionar acceso a la instancia única.

13.7.4 Método Fábrica

El patrón *método Fábrica* provee una interfaz para crear objetos pero deja que las subclases decidan la clase específica de objeto que se crea. Típicamente, el cliente espera

una superclase o una interfaz del objeto actual y el método Fábrica provee las especializaciones.

Los iteradores de las colecciones son un ejemplo de esta técnica. Si tenemos una variable de tipo Collection podemos solicitar un iterador (usando el método iterator) y luego trabajar con dicho iterador.

Desde el punto de vista del cliente (en el código que se muestra en Código 13.2) estamos operando con objetos de tipo Collection e Iterator. En realidad, el tipo (dinámico) de la colección podría ser ArrayList, en cuyo caso el método i terator retorna un objeto de tipo ArrayListIterator; o podría ser un HashSet en donde iterator retorna un HashSetIterator. El método Fábrica se especializa en las subclases para retornar instancias especializadas del tipo de retorno «oficial».

13.7.5 Observador

El patrón
Observador proporciona una manera de llevar a cabo esta separación modelo-vista. En términos más generales: el patrón Observador define una relación uno a varios, por lo tanto, cuando un objeto cambia su estado podrían modificarse muchos otros objetos. Logra este efecto con un grado muy bajo de acoplamiento entre los observadores y el objeto observado.
Podemos ver a partir de esto que el patrón Observador no sólo soporta una vista desacoplada del modelo sino que también permite varias vistas diferentes (ya sean alternativas o simultáneas).

Para implementar el patrón Observador usamos dos clases abstractas: Observable y Observer[6]. La entidad observable (en nuestra simulación: el Campo) extiende la clase Observable y el observador (VisorDelSimulador) extiende la clase Observer

La clase Observable proporciona métodos a los observadores que les permite asociarse a la entidad observada. Esto asegura que el método update de los observadores se invoque cada vez que la entidad observada (el campo) invoca su método heredado not i f y. Los observadores actuales (los espectadores) pueden obtener un estado nuevo y actualizado del campo y mostrarlo nuevamente en la pantalla.

El patrón Observador también se puede usar para otros problemas distintos del que presenta la separación modelo-vista. Se puede aplicar siempre que el estado de uno o más objetos dependa del estado de otro objeto.

13.7.6 Resumen de patrones

Una vez que comprendemos cómo crear buenas implementaciones de clases con funcionalidad bien definida, podemos concentrarnos en decidir qué tipo de clases debemos tener en nuestra aplicación y cómo deben cooperar. Las buenas soluciones no siempre son obvias y por eso los patrones de diseño describen estructuras que han demostrado ser útiles una y otra vez para resolver tipos de problemas que se repiten.
A medida que adquiera más experiencia como desarrollador de software, empleará más tiempo en pensar sobre las estructuras de alto nivel en lugar de pensar en la implementación de métodos.