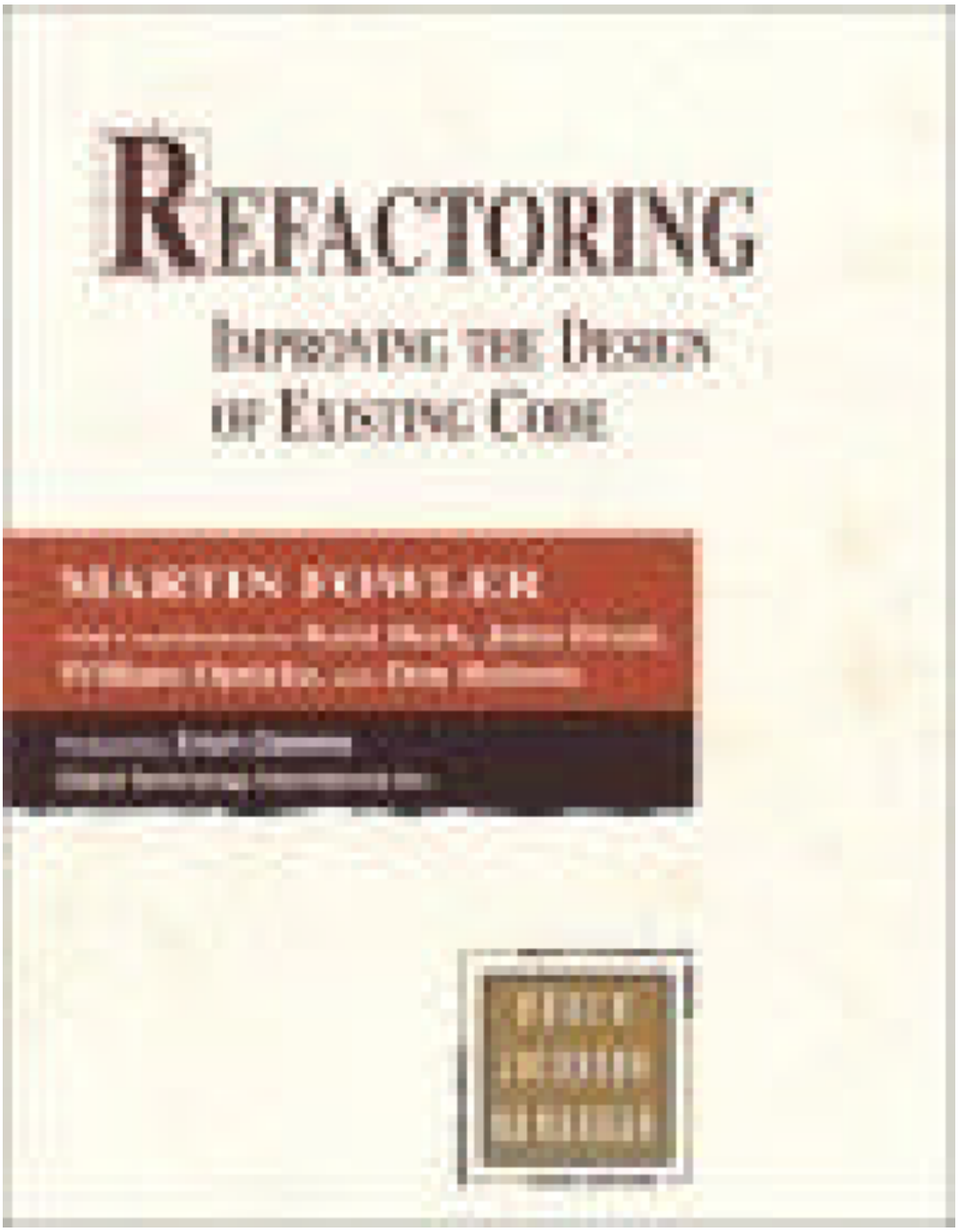
**Refactorización: mejorar el diseño del código existente**
por Martín Fowler, Kent Beck (Contribuyente), joh brant (Contribuyente), Guillermo por el camino, don roberts




***Otro lanzamiento estúpido 2002.☺***

Tu biblioteca de clases funciona, pero ¿podría ser mejor? *Refactorización: mejorar el diseño del código existente* muestra cómo *refactorización* puede hacer que el código orientado a objetos sea más simple y fácil de mantener. Hoy en día, la refactorización requiere considerables conocimientos de diseño, pero una vez que las herramientas estén disponibles, todos los programadores deberían poder mejorar su código utilizando técnicas de refactorización.

Además de una introducción a la refactorización, este manual proporciona un

catálogo de docenas de consejos para mejorar el código. Lo mejor de *Refactorización* es su presentación notablemente clara, junto con excelentes consejos prácticos del experto en objetos Martin Fowler. El autor también es una autoridad en patrones de software y UML, y esta experiencia ayuda a que esto sea una
mejor libro, uno que debería ser inmediatamente accesible para cualquier desarrollador orientado a objetos intermedio o avanzado. (Al igual que los patrones, cada consejo de refactorización se presenta con un nombre simple, una "motivación" y ejemplos que utilizan Java y UML).

Los primeros capítulos enfatizan la importancia de las pruebas en una refactorización exitosa. (Cuando mejora el código, debe realizar pruebas para verificar que todavía funciona). Después de la discusión sobre cómo detectar el "olor" del código incorrecto, los lectores llegan al corazón del libro, su catálogo de más de 70 "refactorizaciones": consejos para un diseño de clases mejor y más simple. Cada consejo está ilustrado con un código de "antes" y "después", junto con una explicación. Los capítulos posteriores ofrecen un vistazo rápido a la investigación sobre refactorización.

Al igual que los patrones de software, la refactorización puede ser una idea cuyo momento ha llegado. Este título innovador seguramente ayudará a llevar la refactorización a la programación principal.  Con sus consejos claros sobre un tema nuevo y candente, *Refactorización* Seguramente será una lectura esencial para cualquiera que escriba o mantenga software orientado a objetos. *--Richard Dragán*

**Temas cubiertos:** Refactorización, mejora del código de software, rediseño, consejos de diseño, patrones, pruebas unitarias, investigación de refactorización y herramientas.

***Noticias del libro, Inc.***
Una guía para refactorizar, el proceso de cambiar un sistema de software para que no altere el comportamiento externo del código pero mejore su estructura interna, para programadores profesionales. Los primeros capítulos cubren principios generales, fundamentos, ejemplos y pruebas.  El corazón del libro es un catálogo de refactorizaciones, organizado en capítulos sobre composición de métodos, movimiento de características entre objetos, organización de datos, simplificación de expresiones condicionales y tratamiento de generalizaciones.

# Prefacio

La "refactorización" se concibió en los círculos de Smalltalk, pero no pasó mucho tiempo antes de que llegara a otros campos de lenguajes de programación. Debido a que la refactorización es parte integral del desarrollo de marcos, el término surge rápidamente cuando los "frameworkers" hablan de su oficio. Surge cuando refinan sus jerarquías de clases y cuando se entusiasman con la cantidad de líneas de código que pudieron eliminar. Los creadores de marcos saben que un marco no será correcto la primera vez: debe evolucionar a medida que adquieren experiencia. También saben que el código se leerá y modificará con más frecuencia de la que se escribirá. La clave para mantener el código legible y modificable es la refactorización, para los marcos en particular, pero también para el software en general.

Entonces, ¿cuál es el problema? Simplemente esto: la refactorización es arriesgada. Requiere cambios en el código de trabajo que pueden introducir errores sutiles. La refactorización, si no se realiza correctamente, puede retrasar días, incluso semanas. Y la refactorización se vuelve más riesgosa cuando se practica de manera informal o ad hoc. Empiezas a profundizar en el código. Pronto descubres nuevas oportunidades de cambio y profundizas más. Cuanto más investigas, más cosas encuentras... y más cambios haces. Al final te hundes en un agujero del que no puedes salir. Para evitar cavar su propia tumba, la refactorización debe realizarse de forma sistemática. Cuando mis coautores y yo escribimos *patrones de diseño,* Mencionamos que los patrones de diseño proporcionan objetivos para las refactorizaciones. Sin embargo, identificar el objetivo es sólo una parte del problema; Transformar su código para llegar allí es otro desafío.

Martin Fowler y los autores contribuyentes hacen una contribución invaluable al desarrollo de software orientado a objetos al arrojar luz sobre el proceso de refactorización. Este libro explica los principios y las mejores prácticas de refactorización y señala cuándo y dónde debe comenzar a profundizar en su código para mejorarlo. La esencia del libro es un catálogo completo de refactorizaciones. Cada refactorización describe la motivación y la mecánica de una transformación de código probada. Algunas de las refactorizaciones, como Extraer método o Mover campo, pueden parecer obvias.

Pero no se deje engañar. Comprender la mecánica de dichas refactorizaciones es la clave para refactorizar de forma disciplinada. Las refactorizaciones de este libro le ayudarán a cambiar su código paso a paso, reduciendo así los riesgos de hacer evolucionar su diseño. Agregará rápidamente estas refactorizaciones y sus nombres a su vocabulario de desarrollo.

Mi primera experiencia con la refactorización disciplinada, "un paso a la vez", fue cuando estaba programando en pareja a 30.000 pies con Kent Beck. Se aseguró de que aplicáramos las refactorizaciones del catálogo de este libro paso a paso. Me sorprendió lo bien que funcionó esta práctica. No sólo aumentó mi confianza en el código resultante, sino que también me sentí menos estresado. Le recomiendo encarecidamente que pruebe estas refactorizaciones: usted y su código se sentirán mucho mejor con ello.



*—Erich Gamma*

*Objeto Tecnología Internacional, Inc.*

# Prefacio

Érase una vez un consultor que visitó un proyecto de desarrollo. El consultor miró parte del código que se había escrito; había una jerarquía de clases en el centro del sistema. Mientras recorría la jerarquía, el consultor vio que estaba bastante desordenada. Las clases de nivel superior hicieron ciertas suposiciones sobre cómo funcionarían las clases, suposiciones que estaban incorporadas en el código heredado. Sin embargo, ese código no se adaptaba a todas las subclases y fue anulado en gran medida. Si la superclase se hubiera modificado un poco, habría sido necesaria mucha menos anulación. En otros lugares, algunas de las intenciones de la superclase no se habían entendido adecuadamente y el comportamiento presente en la superclase se duplicó. En otros lugares, varias subclases hicieron lo mismo con código que claramente podía ascender en la jerarquía.

El consultor recomendó a la dirección del proyecto que se examinara y limpiara el código, pero la dirección del proyecto no pareció entusiasmada. El código parecía funcionar y había considerables presiones en el cronograma. Los directivos dijeron que lo abordarían más adelante.

El consultor también había mostrado lo que estaba pasando a los programadores que habían trabajado en la jerarquía. Los programadores estaban interesados y vieron el problema. Sabían que en realidad no era culpa suya; A veces se necesita un nuevo par de ojos para detectar el problema. Entonces los programadores pasaron uno o dos días limpiando la jerarquía. Cuando terminaron, los programadores habían eliminado la mitad del código de la jerarquía sin reducir su funcionalidad. Quedaron satisfechos con el resultado y descubrieron que resultaba más rápido y sencillo agregar nuevas clases a la jerarquía y utilizar las clases en el resto del sistema.

La dirección del proyecto no quedó contenta. Los horarios eran apretados y había mucho trabajo por hacer. Estos dos programadores habían pasado dos días haciendo un trabajo que no había servido para agregar las muchas funciones que el sistema tenía que ofrecer en unos pocos meses. El código antiguo había funcionado bien. Entonces el diseño era un poco más "puro", un poco más "limpio". El proyecto tenía que ofrecer un código que funcionara, no un código que agradara a un académico. El consultor sugirió que esta limpieza se hiciera en otras partes centrales del sistema. Una actividad de este tipo podría detener el proyecto durante una semana o dos. Toda esta actividad se dedicó a hacer que el código se viera mejor, no a que hiciera nada que no hiciera ya.

¿Cómo te sientes acerca de esta historia? ¿Cree que el consultor hizo bien en sugerir una mayor limpieza? ¿O sigues ese viejo dicho de ingeniería: "si funciona, no lo arregles"?

Debo admitir que hay cierta parcialidad aquí. Yo era ese consultor. Seis meses después, el proyecto fracasó, en gran parte porque el código era demasiado complejo para depurarlo o ajustarlo a un rendimiento aceptable.

Se contrató al consultor Kent Beck para reiniciar el proyecto, un ejercicio que implicó reescribir casi todo el sistema desde cero. Hizo varias cosas de manera diferente, pero una de las más importantes fue insistir en la limpieza continua del código mediante la refactorización. El éxito de este proyecto, y el papel que jugó la refactorización en este éxito, es lo que me inspiró a escribir este libro, para poder transmitir el conocimiento que Kent y otros han aprendido al utilizar la refactorización para mejorar la calidad del software.

## ¿Qué es la refactorización?

La refactorización es el proceso de cambiar un sistema de software de tal manera que no altere el comportamiento externo del código pero mejore su estructura interna. Es una forma disciplinada de limpiar el código que minimiza las posibilidades de introducir errores. En esencia, cuando refactorizas estás mejorando el diseño del código una vez escrito.

"Mejorar el diseño una vez escrito." Ésa es una expresión extraña. En nuestra comprensión actual del desarrollo de software, creemos que diseñamos y luego codificamos. Un buen diseño es lo primero y la codificación lo segundo. Con el tiempo el código se irá modificando, y la integridad del sistema, su estructura según ese diseño, se desvanecerá gradualmente. El código pasa lentamente de la ingeniería al pirateo.

La refactorización es lo opuesto a esta práctica. Con la refactorización puedes tomar un mal diseño, incluso un caos, y reelaborarlo en un código bien diseñado. Cada paso es simple, incluso simplista. Mueves un campo de una clase a otra, extraes algo de código de un método para convertirlo en su propio método y empujas algo de código hacia arriba o hacia abajo en una jerarquía. Sin embargo, el efecto acumulativo de estos pequeños cambios puede mejorar radicalmente el diseño. Es exactamente lo contrario de la noción normal de deterioro del software.

Con la refactorización se encuentra el equilibrio de los cambios en el trabajo. Usted descubre que el diseño, en lugar de ocurrir desde el principio, ocurre continuamente durante el desarrollo. Al construir el sistema se aprende cómo mejorar el diseño. La interacción resultante conduce a un programa con un diseño que se mantiene bien a medida que continúa el desarrollo.

## ¿Qué hay en este libro?

Este libro es una guía para la refactorización; Está escrito para un programador profesional. Mi objetivo es mostrarle cómo realizar la refactorización de forma controlada y eficiente. Aprenderá a refactorizar de tal manera que no introduzca errores en el código, sino que mejore metódicamente la estructura.

Es tradicional empezar los libros con una introducción. Aunque estoy de acuerdo con ese principio, no me resulta fácil introducir la refactorización con una discusión o definiciones generalizadas. Entonces comienzo con un ejemplo. Capítulo 1 toma un pequeño programa con algunos defectos de diseño comunes y lo refactoriza en un programa orientado a objetos más aceptable. En el camino vemos tanto el proceso de refactorización como la aplicación de varias refactorizaciones útiles. Este es el capítulo clave que debe leer si desea comprender de qué se trata realmente la refactorización.

En Capítulo 2 Cubro más sobre los principios generales de la refactorización, algunas definiciones y las razones para realizar la refactorización. Describo algunos de los problemas

con la refactorización. En Capítulo 3 Kent Beck me ayuda a describir cómo encontrar malos olores en el código y cómo limpiarlos mediante refactorizaciones. Las pruebas juegan un papel muy importante en la refactorización, por lo que Capítulo 4 describe cómo construir pruebas en código con un marco de pruebas Java simple de código abierto.

El corazón del libro, el catálogo de refactorizaciones, se extiende desde el Capítulo 5 hasta el Capítulo 12. Este no es de ninguna manera un catálogo completo. Es el comienzo de tal catálogo. Incluye las refactorizaciones que he anotado hasta ahora en mi trabajo en este campo. Cuando quiero hacer algo, como Reemplazar condicional con polimorfismo, el catálogo me recuerda cómo hacerlo de forma segura, paso a paso. Espero que esta sea la sección del libro a la que volverás a menudo.

En este libro describo el fruto de muchas investigaciones realizadas por otros. Los últimos capítulos son capítulos invitados de algunas de estas personas. Capítulo 13 Es de Bill Opdyke, quien describe los problemas que ha encontrado al adoptar la refactorización en el desarrollo comercial. Capítulo 14 es de don



Roberts y John Brant, quienes describen el verdadero futuro de las herramientas automatizadas de refactorización. He dejado la última palabra, Capítulo 15, al maestro del arte, Kent Beck.

### Refactorización en Java

En todo este libro utilizo ejemplos en Java. Por supuesto, la refactorización se puede realizar con otros lenguajes y espero que este libro sea útil para quienes trabajan con otros lenguajes. Sin embargo, sentí que sería mejor centrar este libro en Java porque es el lenguaje que mejor conozco. He agregado notas ocasionales para la refactorización en otros idiomas, pero espero que otras personas aprovechen esta base con libros dirigidos a idiomas específicos.

Para ayudar a comunicar mejor las ideas, no he utilizado áreas particularmente complejas del lenguaje Java. Por eso he evitado el uso de clases internas, reflexión, subprocesos y muchas otras de las características más potentes de Java. Esto se debe a que quiero centrarme en las refactorizaciones principales lo más claramente posible.

Debo enfatizar que estas refactorizaciones no se realizan teniendo en mente la programación concurrente o distribuida. Esos temas introducen preocupaciones adicionales que están más allá del alcance de este libro.

## ¿Quién debería leer este libro?

Este libro está dirigido a un programador profesional, alguien que se gana la vida escribiendo software. Los ejemplos y la discusión incluyen mucho código para leer y comprender. Todos los ejemplos están en Java. Elegí Java porque es un lenguaje cada vez más conocido que cualquier persona con experiencia en C puede entender fácilmente. También es un lenguaje orientado a objetos y los mecanismos orientados a objetos son de gran ayuda en la refactorización.

Aunque se centra en el código, la refactorización tiene un gran impacto en el diseño del sistema. Es vital que los diseñadores y arquitectos senior comprendan los principios de la refactorización y los utilicen en sus proyectos. La refactorización la realiza mejor un desarrollador respetado y experimentado. Un desarrollador de este tipo puede comprender mejor los principios detrás de la refactorización y adaptar esos principios al lugar de trabajo específico. Esto es particularmente

cierto cuando utilizas un lenguaje distinto de Java, porque tienes que adaptar los ejemplos que he dado a otros lenguajes.

Aquí le mostramos cómo aprovechar al máximo este libro sin leerlo todo.

- **Si quieres entender qué es la refactorización,** leer Capítulo 1; el ejemplo debería aclarar el proceso.
- **Si quieres entender por qué deberías refactorizar,** Lea los dos primeros capítulos. Ellos te dirán qué es la refactorización y por qué deberías hacerlo.
- **Si desea encontrar dónde debe refactorizar,** leer Capítulo 3. Le indica las señales que sugieren la necesidad de refactorizar.
- **Si realmente quieres hacer una refactorización,** Lea los primeros cuatro capítulos por completo. Luego omita la lectura del catálogo. Lea lo suficiente del catálogo para saber aproximadamente lo que contiene. No es necesario que comprenda todos los detalles. Cuando realmente necesite realizar una refactorización, lea la refactorización en detalle y utilícela como ayuda. El catálogo es una sección de referencia, por lo que probablemente no querrás leerlo de una sola vez. También deberías leer los capítulos invitados, especialmente Capítulo 15.

## Construir sobre los cimientos puestos por otros



Necesito decir ahora, para empezar, que tengo una gran deuda con este libro, una deuda con aquellos cuyo trabajo durante la última década ha desarrollado el campo de la refactorización. Lo ideal sería que uno de ellos hubiera escrito este libro, pero terminé siendo yo quien tenía el tiempo y la energía.

Dos de los principales defensores de la refactorización son **Ward Cunningham** y **Kent Beck.** Lo utilizaron como parte central de su proceso de desarrollo en los primeros días y han adaptado sus procesos de desarrollo para aprovecharlo. En particular, fue mi colaboración con Kent la que realmente me mostró la importancia de la refactorización, una inspiración que me llevó directamente a este libro.

**Ralph Johnson** Lidera un grupo en la Universidad de Illinois en Urbana-Champaign que se destaca por sus contribuciones prácticas a la tecnología de objetos. Ralph ha sido durante mucho tiempo un defensor de la refactorización y varios de sus alumnos han trabajado en el tema. **Bill Opdyke** Desarrolló el primer trabajo escrito detallado sobre refactorización en su tesis doctoral. **Juan Brant** y **Don Roberts** han ido más allá de escribir palabras y han escrito una herramienta, el Refactoring Browser, para refactorizar programas Smalltalk.

## Expresiones de gratitud

Incluso con toda esa investigación a la que recurrir, todavía necesitaba mucha ayuda para escribir este libro. En primer lugar, Kent Beck fue de gran ayuda. Las primeras semillas se plantaron en un bar de Detroit cuando Kent me habló de un artículo que estaba escribiendo para el *Informe de pequeña charla* [Beck, hanói]. No sólo me proporcionó muchas ideas para robar Capítulo 1 pero también me inició en la toma de notas de refactorizaciones. Kent también ayudó en otros lugares. Se le ocurrió la idea de los olores de código, me animó en varios puntos difíciles y, en general, trabajó conmigo para que este libro funcionara. No puedo evitar pensar que él mismo podría haber escrito este libro mucho mejor, pero tuve tiempo y sólo puedo esperar haber hecho justicia al tema.

Mientras escribía esto, quería compartir gran parte de esta experiencia directamente con usted, por lo que estoy muy agradecido de que muchas de estas personas hayan dedicado algún tiempo a agregar material a este libro. Kent Beck, John Brant, William Opdyke y Don Roberts han escrito o coescrito capítulos. Además, Rich Garzaniti y Ron Jeffries agregaron útiles recuadros laterales.

Cualquier autor le dirá que los revisores técnicos hacen mucho para ayudar en un libro como este. Como de costumbre, Carter Shanklin y su equipo en Addison-Wesley reunieron un gran panel de críticos duros. Estos fueron

- Ken Auer, Rolemodel Software, Inc.
- Joshua Bloch, Sun Microsystems, software Java
- John Brant, Universidad de Illinois en Urbana-Champaign
- Scott Corley, software de alto voltaje, Inc.
- Ward Cunningham, Cunningham y Cunningham, Inc.
- Stéphane Ducasse
- Erich Gamma, Object Technology International, Inc.
- Ron Jeffries
- Ralph Johnson, Universidad de Illinois
- Joshua Kerievsky, Lógica Industrial, Inc.
- Doug Lea, SUNY Oswego
- Sander Tichelaar

Todos ellos contribuyeron mucho a la legibilidad y precisión de este libro y eliminaron al menos algunos de los errores que pueden acechar en cualquier manuscrito. Me gustaría resaltar un par de sugerencias muy visibles que marcaron una diferencia en el aspecto del libro. Ward y Ron me pidieron que hiciera Capítulo



1 en el estilo de lado a lado. Joshua Kerievksy sugirió la idea de los bocetos de código en el catálogo.

Además del panel de revisión oficial, hubo muchos revisores no oficiales. Estas personas miraron el manuscrito o el trabajo en progreso en mis páginas web e hicieron comentarios útiles. Incluyen a Leif Bennett, Michael Feathers, Michael Finney, Neil Galarneau, Hisham Ghazouli, Tony Gould, John Isner, Brian Marick, Ralf Reissing, John Salt, Mark Swanson, Dave Thomas y Don Wells. Seguro que hay otros que se me han olvidado; Pido disculpas y ofrezco mi agradecimiento.

Un grupo de revisión particularmente entretenido es el infame grupo de lectura de la Universidad de Illinois en Urbana-Champaign. Debido a que este libro refleja gran parte de su trabajo, estoy particularmente agradecido por sus esfuerzos capturados en audio real. Este grupo incluye a Fredrico "Fred" Balaguer, John Brant,
Ian Chai, Brian Foote, Alejandra Garrido, Zhijiang "John" Han, Peter Hatch, Ralph Johnson, Songyu "Raymond" Lu, Dragos-Anton Manolescu, Hiroaki Nakamura, James Overturf, Don Roberts, Chieko Shirai, Les Tyrell y Joe Yoder.

Cualquier buena idea debe probarse en un sistema de producción serio. Vi que la refactorización tuvo un efecto enorme en el sistema de compensación integral de Chrysler (C3). Quiero agradecer a todos los miembros de ese equipo: Ann Anderson, Ed Anderi, Ralph Beattie, Kent Beck, David Bryant, Bob Coe, Marie DeArment, Margaret Fronczak, Rich

Garzaniti, Dennis Gore, Brian Hacker, Chet Hendrickson, Ron Jeffries, Doug Joppie, David Kim, Paul Kowalsky, Debbie Mueller, Tom Murasky, Richard Nutter, Adrian Pantea, Matt Saigeon, Don Thomas y Don Wells. Trabajar con ellos consolidó los principios y beneficios de la refactorización de primera mano. Observar su progreso mientras utilizan mucho la refactorización me ayuda a ver lo que la refactorización puede hacer cuando se aplica a un proyecto grande durante muchos años.

Nuevamente conté con la ayuda de J. Carter Shanklin en Addison-Wesley y su equipo: Krysia Bebick, Susan Cestone, Chuck Dutton, Kristin Erickson, John Fuller, Christopher Guzikowski, Simone Payment y Genevieve Rajewski. Trabajar con una buena editorial es un placer; Me brindaron mucho apoyo y ayuda.

Hablando de apoyo, quien más sufre con un libro es siempre el más cercano al autor, en este caso mi (ahora) esposa Cindy. Gracias por amarme incluso cuando estaba escondido en el estudio. Por mucho tiempo que dediqué a este libro, nunca dejé de distraerme pensando en ti.

*Martín Fowler*

*Melrose (Massachusetts)*

*fowler@acm.org*

http://www.martinfowler.com

http://www.refactoring.com



# Capítulo 1. Refactorización, un primer ejemplo

¿Cómo empiezo a escribir sobre refactorización? La forma tradicional de empezar a hablar de algo es resumir la historia, los principios generales y cosas por el estilo. Cuando alguien hace eso en una conferencia, me da un poco de sueño. Mi mente comienza a divagar con un proceso en segundo plano de baja prioridad que sondea al hablante hasta que da un ejemplo. Los ejemplos me despiertan porque es con ejemplos que puedo ver lo que está pasando. Con principios es demasiado fácil hacer generalizaciones y demasiado difícil descubrir cómo aplicar las cosas. Un ejemplo ayuda a aclarar las cosas.

Así que comenzaré este libro con un ejemplo de refactorización. Durante el proceso, les contaré mucho sobre cómo funciona la refactorización y les daré una idea del proceso de refactorización. Luego puedo proporcionar la introducción habitual al estilo de los principios.

Sin embargo, con un ejemplo introductorio me encuentro con un gran problema. Si elijo un programa grande, describirlo y cómo se refactoriza es demasiado complicado para que cualquier

lector pueda entenderlo. (Lo intenté e incluso un ejemplo un poco complicado ocupa más de cien páginas). Sin embargo, si elijo un programa que es lo suficientemente pequeño como para ser comprensible, no parece que valga la pena refactorizarlo.

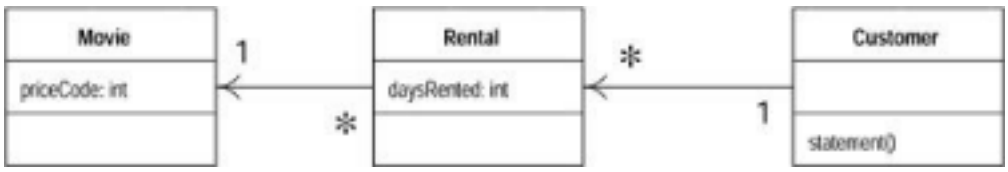
Por lo tanto, estoy en el clásico aprieto de cualquiera que quiera describir técnicas que sean útiles para programas del mundo real. Francamente no vale la pena el esfuerzo de hacer la refactorización que les voy a mostrar en un programa pequeño como el que voy a usar. Pero si el código que les muestro es parte de un sistema más grande, entonces la refactorización pronto se vuelve importante. Así que tengo que pedirles que miren esto y lo imaginen en el contexto de un sistema mucho más amplio.

## El punto de partida

El programa de muestra es muy simple. Es un programa para calcular e imprimir un extracto de los cargos de un cliente en un videoclub. El programa indica qué películas alquiló un cliente y por cuánto tiempo. Luego calcula los cargos, que dependen de cuánto tiempo se alquila la película, e identifica el tipo de película. Hay tres tipos de películas: regulares, infantiles y de estreno. Además de calcular los cargos, el estado de cuenta también calcula los puntos de alquiler frecuente, que varían dependiendo de si la película es un estreno nuevo.

Varias clases representan varios elementos de vídeo. Aquí hay un diagrama de clases para mostrarlos (Cifra 1.1).

**Figura 1.1. Diagrama de clases de las clases de punto de partida. Sólo se muestran las características más importantes. La notación es Lenguaje Unificado de Modelado UML [Fowler, UML].**




Mostraré el código de cada una de estas clases por turno.

### Película

La película es solo una clase de datos simple.



```
Película de clase pública {

public static final int NIÑOS = 2;
público estático final int REGULAR = 0;
público estático final int NEW_RELEASE = 1;

cadena privada _title;
privado int _priceCode;

Película pública (título de cadena, int código de precio) {
_título = título;
_priceCode = precioCode;
}
```

```
public int getPriceCode() {
devolver _priceCode;
}

setPriceCode público vacío (int arg) {
_priceCode = arg;
}

cadena pública getTitle (){
devolver _título;
};
}
```

## Alquiler

La clase de alquiler representa a un cliente que alquila una película.

```
clase Alquiler {
película privada _película;
privado int _daysRented;

Alquiler público(Película, int díasAlquilado) {
_pelicula = pelicula;
_diasAlquilado = diasAlquilado;
}
public int getDaysRented() {
return _daysRented;
}
Película pública getMovie() {
devolver _película;
}
}
```

## Cliente

La clase de cliente representa al cliente de la tienda. Al igual que las otras clases, tiene datos y descriptores de acceso:

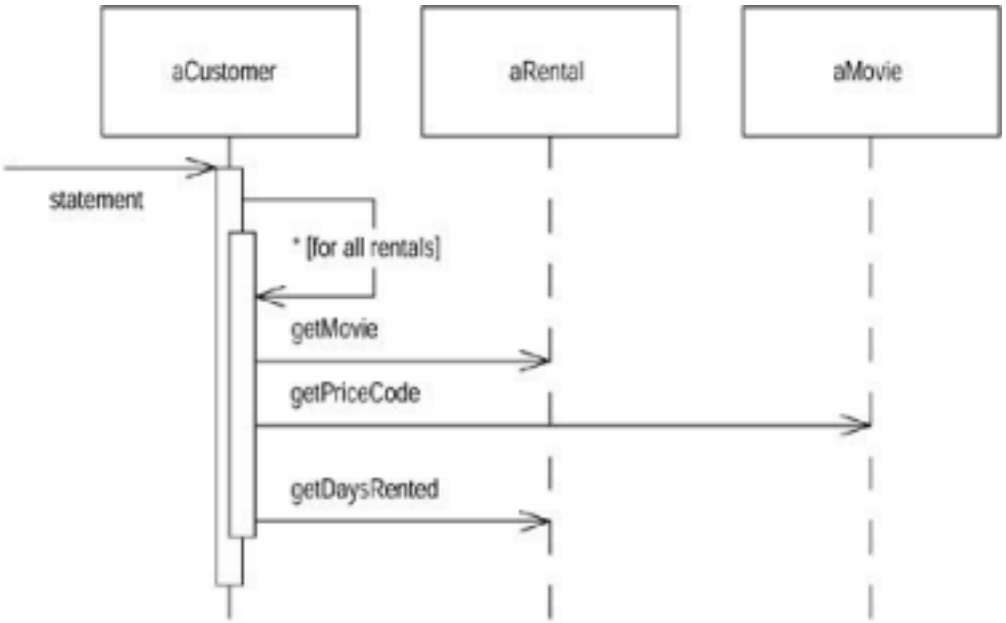


```
clase Cliente {
cadena privada _nombre;
Vector privado _rentals = nuevo Vector();

Cliente público (nombre de cadena){
_nombre = nombre;
};

addRental public void (argumento de alquiler) {
_alquileres.addElement(arg);
}
```

```
 cadena pública getName (){
 devolver _nombre;
 };
```

El cliente también tiene el método que produce una declaración. Figura 1.2 muestra las interacciones para este método. El cuerpo de este método está en la página opuesta.

**Figura 1.2. Interacciones para el método de declaración**




```
declaración de cadena pública () {
 doble cantidad total = 0;
 int puntos de alquiler frecuentes = 0;
 Alquileres de enumeración = _rentals.elements();
 Resultado de cadena = "Registro de alquiler para " + getName() +
"\n";  mientras (alquileres.hasMoreElements()) {
 duplicar estaCantidad = 0;
 Alquiler cada = (Alquiler) alquileres.nextElement();

 //determinar cantidades para cada línea
 cambiar (cada.getMovie().getPriceCode()) { caso
Película.REGULAR:
 estaCantidad += 2;
 si (cada.getDaysRented() > 2)
```



```
 estaCantidad += (cada.getDaysRented() - 2) * 1.5;  romper;
 caso Película.NEW_RELEASE:
 estaCantidad += cada.getDaysRented() * 3;  romper;
 caso Película.NIÑOS:
 estaCantidad += 1,5;
 si (cada.getDaysRented() > 3)
 estaCantidad += (cada.getDaysRented() - 3) * 1.5;  romper;

 }
```

```
 // agregar puntos de inquilino frecuente
 puntos de alquiler frecuentes ++;
 // agrega bonificación por el alquiler de un nuevo lanzamiento por dos
días if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
each.getDaysRented() > 1) frecuentesRenterPoints ++;

 //mostrar cifras de este alquiler
 resultado += "\t" + each.getMovie().getTitle()+ "\t" +
String.valueOf(thisAmount) + "\n";
 monto total += este monto;

 }
 //agregar líneas de pie de página
 resultado += "La cantidad adeuda es " + String.valueOf(totalAmount) +
"\n";
 resultado += "Has ganado " + String.valueOf(frequentRenterPoints) +
"puntos de inquilino frecuente";
 resultado de devolución;

 }
```

## Comentarios sobre el programa inicial

¿Cuáles son tus impresiones sobre el diseño de este programa? Lo describiría como no bien diseñado y ciertamente no orientado a objetos. Para un programa simple como este, eso realmente no importa. No hay nada de malo en una comida rápida y sucia. *simple* programa. Pero si este es un fragmento representativo de un sistema más complejo, entonces tengo algunos problemas reales con este programa. Esa larga rutina de declaraciones en la clase Cliente hace demasiado. Muchas de las cosas que hace realmente deberían ser realizadas por las otras clases.

Aun así el programa funciona. ¿No es esto sólo un juicio estético, un disgusto por el código feo? Lo es hasta que queramos cambiar el sistema. Al compilador no le importa si el código es feo o limpio. Pero cuando cambiamos el sistema, hay un ser humano involucrado, y a los humanos sí les importa. Un sistema mal diseñado es difícil de cambiar. Difícil porque es difícil determinar dónde se necesitan los cambios. Si es difícil saber qué cambiar, existe una gran posibilidad de que el programador cometa un error e introduzca errores.

En este caso tenemos un cambio que a los usuarios les gustaría hacer. Primero, quieren una declaración impresa en HTML para que pueda estar habilitada para la Web y ser totalmente compatible con las palabras de moda. Considere el impacto que tendría este cambio. Al mirar el código puedes ver que es



imposible reutilizar cualquiera de los comportamientos del método de declaración actual para una declaración HTML. Su único recurso es escribir un método completamente nuevo que duplique gran parte del comportamiento de la declaración. Ahora bien, por supuesto, esto no es demasiado oneroso. Puede simplemente copiar el método de declaración y realizar los cambios que necesite.

Pero, ¿qué sucede cuando cambian las reglas de cobro? tienes que arreglar ambos `declaración` y `htmlDeclaración` y asegúrese de que las correcciones sean consistentes. El problema de copiar y pegar código viene cuando tienes que cambiarlo más

tarde. Si está escribiendo un programa que no espera cambiar, entonces cortar y pegar está bien. Si el programa tiene una larga vida y es probable que cambie, entonces cortar y pegar es una amenaza.

Esto me lleva a un segundo cambio. Los usuarios quieren realizar cambios en la forma en que clasifican las películas, pero aún no han decidido qué cambio van a realizar. Tienen una serie de cambios en mente. Estos cambios afectarán tanto la forma en que se les cobra a los inquilinos por las películas como la forma en que se calculan los puntos de los inquilinos frecuentes. Como desarrollador experimentado, estás seguro de que, sea cual sea el esquema que se les ocurra a los usuarios, la única garantía que tendrás es que lo cambiarán nuevamente dentro de seis meses.

El método de declaración es donde se deben realizar los cambios para hacer frente a los cambios en las reglas de clasificación y cobro. Sin embargo, si copiamos la declaración a una declaración HTML, debemos asegurarnos de que cualquier cambio sea completamente consistente. Además, a medida que las reglas crezcan en complejidad, será más difícil determinar dónde realizar los cambios y realizarlos sin cometer un error.

Es posible que se sienta tentado a realizar la menor cantidad de cambios posibles en el programa; después de todo, funciona bien.  Recuerde el viejo dicho de ingeniería: "si no está roto, no lo arregles". Puede que el programa no esté roto, pero sí duele. Te está haciendo la vida más difícil porque te resulta difícil realizar los cambios que tus usuarios desean. Aquí es donde entra en juego la refactorización.

> **Consejo**
>
> *Cuando descubra que tiene que agregar una característica a un programa y el código del programa no está estructurado de una manera conveniente para agregar la característica, primero refactorice el programa para que sea más fácil agregar la característica y luego agregue la característica.*

## El primer paso en la refactorización

Siempre que hago una refactorización, el primer paso es siempre el mismo. Necesito crear un conjunto sólido de pruebas para esa sección de código. Las pruebas son esenciales porque, aunque sigo refactorizaciones estructuradas para evitar la mayoría de las oportunidades de introducir errores, sigo siendo humano y sigo cometiendo errores.  Por eso necesito pruebas sólidas.

Debido a que el resultado de la declaración produce una cadena, creo algunos clientes, le doy a cada cliente algunos alquileres de varios tipos de películas y genero las cadenas de declaración. Luego hago una comparación de cadenas entre la nueva cadena y algunas cadenas de referencia que he verificado manualmente. Configuré todas estas pruebas para poder ejecutarlas desde un comando Java en la línea de comando. Las pruebas tardan sólo unos segundos en ejecutarse y, como verá, las realizo con frecuencia.

Una parte importante de las pruebas es la forma en que informan sus resultados. O dicen "OK", lo que significa que todas las cadenas son idénticas a las cadenas de referencia, o imprimen una lista de fallas: líneas que resultaron diferentes. Por tanto, las pruebas son de autoverificación. Es vital realizar pruebas de autocomprobación. Si no lo hace, terminará perdiendo tiempo comparando algunos números de la prueba con algunos números de una libreta de escritorio, y eso lo ralentizará.

Mientras hacemos la refactorización, nos apoyaremos en las pruebas. Confiaré en las pruebas para saber si introduzco un error. Es esencial para la refactorización tener buenas pruebas. Vale la pena dedicar tiempo a crear las pruebas, porque las pruebas le brindan la seguridad que necesita para cambiar el programa más adelante. Esta es una parte tan importante de la refactorización que entraré en más detalles sobre las pruebas en Capítulo 4.

**Consejo**

*Antes de comenzar a refactorizar, verifique que tenga un conjunto sólido de pruebas. Estas pruebas deben ser de autocomprobación.*

## Descomponer y redistribuir el método del enunciado

El primer objetivo obvio de mi atención es el método de declaración demasiado largo. Cuando miro un método largo como ese, busco descomponer el método en partes más pequeñas. Los fragmentos de código más pequeños tienden a hacer que las cosas sean más manejables. Es más fácil trabajar con ellos y moverse.

La primera fase de las refactorizaciones en este capítulo muestra cómo divido el método largo y muevo las piezas a mejores clases. Mi objetivo es hacer que sea más fácil escribir un método de declaración HTML con mucha menos duplicación de código.

Mi primer paso es encontrar un grupo lógico de código y usarlo. Método de extracción. Una pieza obvia aquí es la declaración de cambio. Parece que sería una buena parte para extraerla en su propio método.

Cuando extraigo un método, como en cualquier refactorización, necesito saber qué puede salir mal. Si hago mal la extracción podría introducir un bug en el programa. Entonces, antes de refactorizar, necesito descubrir cómo hacerlo de forma segura. He realizado esta refactorización varias veces antes, así que anoté los pasos seguros en el catálogo.

Primero necesito buscar en el fragmento cualquier variable que tenga un alcance local para el método que estamos viendo, las variables y parámetros locales. Este segmento de código utiliza dos: `cada` y `estaCantidad.` De estos `cada` no es modificado por el código pero `esta cantidad` se modifica. Cualquier variable no modificada que pueda pasar como parámetro. Las variables modificadas necesitan más cuidado. Si solo hay uno, puedo devolverlo. La temperatura se inicializa a 0 cada vez que se realiza el ciclo y no se modifica hasta que el interruptor llega a ella. Entonces puedo asignar el resultado.

Las siguientes dos páginas muestran el código antes y después de la refactorización. El código anterior está a la izquierda, el código resultante a la derecha. El código que estoy extrayendo del original y cualquier cambio en el nuevo código que no creo que sea inmediatamente obvio están en negrita. A medida que continúe con este capítulo, continuaré con esta convención de izquierda-derecha.

```
declaración de cadena pública () {
 doble cantidad total = 0;
 int puntos de alquiler frecuentes = 0;
 Alquileres de enumeración = _rentals.elements();
```

```
 Resultado de cadena = "Registro de alquiler para " + getName() +
"\n";  mientras (alquileres.hasMoreElements()) {
 duplicar estaCantidad = 0;
 Alquiler cada = (Alquiler) alquileres.nextElement();

 //determinar cantidades para cada línea
 cambiar (cada.getMovie().getPriceCode()) {
```



```
 caso Película.REGULAR:
 estaCantidad += 2;
 si (cada.getDaysRented() > 2)
 estaCantidad += (cada.getDaysRented() - 2) * 1.5;  romper;
 caso Película.NEW_RELEASE:
 estaCantidad += cada.getDaysRented() * 3;  romper;
 caso Película.NIÑOS:
 estaCantidad += 1,5;
 si (cada.getDaysRented() > 3)
 estaCantidad += (cada.getDaysRented() - 3) * 1.5;  romper;

 }

 // agregar puntos de inquilino frecuente
 puntos de alquiler frecuentes ++;
 // agrega bonificación por el alquiler de un nuevo lanzamiento por dos
días if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
each.getDaysRented() >
1) puntos de alquiler frecuentes ++;

 //mostrar cifras de este alquiler
 resultado += "\t" + each.getMovie().getTitle()+ "\t" +
String.valueOf(thisAmount) +
"\norte";
 monto total += este monto;

 }
 //agregar líneas de pie de página
 resultado += "La cantidad adeuda es " + String.valueOf(totalAmount) +
"\n";
 resultado += "Has ganado " + String.valueOf(frequentRenterPoints) + "
inquilino frecuente
agujas";
 resultado de devolución;

 }
declaración de cadena pública () {
 doble cantidad total = 0;
 int puntos de alquiler frecuentes = 0;
 Alquileres de enumeración = _rentals.elements();
 Resultado de cadena = "Registro de alquiler para " + getName()
+ "\n";  mientras (alquileres.hasMoreElements()) {
 duplicar estaCantidad = 0;
 Alquiler cada = (Alquiler) alquileres.nextElement();
```

```
estaCantidad = cantidadPara(cada uno);

 // agregar puntos de inquilino frecuente
 puntos de alquiler frecuentes ++;
 // agrega bonificación por un alquiler de nueva versión de dos días if
((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
each.getDaysRented() > 1) frecuentaRenterPoints ++;

 //mostrar cifras de este alquiler
```



```
 resultado += "\t" + each.getMovie().getTitle()+ "\t" +
String.valueOf(thisAmount) + "\n";
 monto total += este monto;

 }
 //agregar líneas de pie de página
 resultado += "La cantidad adeuda es " + String.valueOf(totalAmount)
+ "\n";
 resultado += "Has ganado " + String.valueOf(frequentRenterPoints) +
"puntos de inquilino frecuente";
 resultado de devolución;

 }
}
cantidad int privada para (alquiler cada uno) {
 int estaCantidad = 0;
 cambiar (cada.getMovie().getPriceCode()) {
 caso Película.REGULAR:
 estaCantidad += 2;
 si (cada.getDaysRented() > 2)
 estaCantidad += (cada.getDaysRented() - 2) * 1.5;  romper;
 caso Película.NEW_RELEASE:
 estaCantidad += cada.getDaysRented() * 3;
 romper;
 caso Película.NIÑOS:
 estaCantidad += 1,5;
 si (cada.getDaysRented() > 3)
 estaCantidad += (cada.getDaysRented() - 3) * 1.5;  romper;

 }
 devolver esta cantidad;
}
```

Cada vez que hago un cambio como este, compilo y pruebo. No tuve un buen comienzo: las pruebas fracasaron. Un par de cifras de la prueba me dieron la respuesta incorrecta. Me quedé perplejo durante unos segundos y luego me di cuenta de lo que había hecho. Tontamente hice el tipo de devolución `cantidad para int` en lugar de `doble:`

```
 privado doble importePara(Alquiler cada uno) {
 doble estaCantidad = 0;
 cambiar (cada.getMovie().getPriceCode()) {
 caso Película.REGULAR:
 estaCantidad += 2;
```

```
si (cada.getDaysRented() > 2)
estaCantidad += (cada.getDaysRented() - 2) * 1.5;  romper;
caso Película.NEW_RELEASE:
estaCantidad += cada.getDaysRented() * 3;  romper;
caso Película.NIÑOS:
estaCantidad += 1,5;
si (cada.getDaysRented() > 3)
estaCantidad += (cada.getDaysRented() - 3) * 1.5;
```



```
romper;
}
devolver esta cantidad;
}
```

Es el tipo de error tonto que cometo a menudo y puede ser complicado localizarlo. En este caso, Java convierte dobles en enteros sin quejarse, pero redondeando alegremente [Java Spec]. Afortunadamente, en este caso fue fácil de encontrar porque el cambio fue muy pequeño y tuve un buen conjunto de pruebas. Aquí está la esencia del proceso de refactorización ilustrada por accidente. Como cada cambio es tan pequeño, cualquier error es muy fácil de encontrar. No pasas mucho tiempo depurando, incluso si eres tan descuidado como yo.

**Consejo**

*La refactorización cambia los programas en pequeños pasos. Si comete un error, es fácil encontrar el error.*

Como estoy trabajando en Java, necesito analizar el código para descubrir qué hacer con las variables locales. Sin embargo, con una herramienta esto se puede hacer realmente sencillo. Una herramienta de este tipo existe en Smalltalk, el navegador de refactorización. Con esta herramienta la refactorización es muy sencilla. Simplemente resalto el código, selecciono "Extraer método" en los menús, escribo el nombre del método y listo. Además, la herramienta no comete errores tontos como el mío. ¡Estoy esperando una versión de Java!

Ahora que he dividido el método original en partes, puedo trabajar en ellas por separado. No me gustan algunos de los nombres de variables en `cantidad para,` y este es un buen lugar para cambiarlos.

Aquí está el código original:

```
importe doble privadoPara(Alquiler cada uno) {
duplicar estaCantidad = 0;
cambiar (cada.getMovie().getPriceCode()) {
caso Película.REGULAR:
estaCantidad += 2;
si (cada.getDaysRented() > 2)
estaCantidad += (cada.getDaysRented() - 2) * 1.5;  romper;
caso Película.NEW_RELEASE:
estaCantidad += cada.getDaysRented() * 3;  romper;
```

```
caso Película.NIÑOS:
estaCantidad += 1,5;
si (cada.getDaysRented() > 3)
estaCantidad += (cada.getDaysRented() - 3) * 1.5;  romper;
}
devolver esta cantidad;
}
```

Aquí está el código renombrado:



```
importe doble privadoPor(Alquiler aAlquiler) {
doble resultado = 0;
cambiar (Alquiler.getPelícula().getPriceCode()) {
caso Película.REGULAR:
resultado += 2;
si (Alquiler.obtenerDíasAlquilado() > 2)
resultado += (aRental.getDaysRented() - 2) * 1.5;  romper;
caso Película.NEW_RELEASE:
resultado += Alquiler.getDíasAlquilado() * 3;
romper;
caso Película.NIÑOS:
resultado += 1,5;
si (Alquiler.obtenerDíasAlquilados() > 3)
resultado += (Alquiler.getDíasAlquilado() - 3) * 1.5;  romper;
}
devolver resultado;
}
```

Una vez que he cambiado el nombre, compilo y pruebo para asegurarme de que no he roto nada.

¿Vale la pena el esfuerzo de cambiar el nombre? Absolutamente. Un buen código debe comunicar claramente lo que está haciendo, y los nombres de las variables son la clave para limpiar el código. Nunca tengas miedo de cambiar los nombres de las cosas para mejorar la claridad. Con buenas herramientas para buscar y reemplazar, no suele ser difícil. La escritura y las pruebas sólidas resaltarán todo lo que se pierda. Recordar

> **Consejo**
>
> *Cualquier tonto puede escribir código que una computadora pueda entender. Los buenos programadores escriben código que los humanos pueden entender.*

El código que comunica su propósito es muy importante. A menudo refactorizo justo cuando leo algún código. De esa manera, a medida que comprendo el programa, incorporo ese conocimiento en el código para no olvidar lo que aprendí más adelante.

## Mover el cálculo de la cantidad

mientras miro `cantidad para`, puedo ver que utiliza información del alquiler, pero no

utiliza información del cliente.

```
Cliente de clase...
 importe doble privadoPara(Rental aRental) {
 resultado doble = 0;
 cambiar (aRental.getMovie().getPriceCode()) {
 caso Película.REGULAR:
 resultado += 2;
 si (aRental.getDaysRented() > 2)
 resultado += (aRental.getDaysRented() - 2) * 1.5;
```



```
romper;
 caso Película.NEW_RELEASE:
 resultado += aRental.getDaysRented() * 3;
 romper;
 caso Película.NIÑOS:
 resultado += 1,5;
 si (aRental.getDaysRented() > 3)
 resultado += (aRental.getDaysRented() - 3) * 1.5;  romper;
 }
 resultado de devolución;
 }
```

Esto inmediatamente me hace sospechar que el método está en el objeto equivocado. En la mayoría de los casos, un método debe estar en el objeto cuyos datos utiliza, por lo que el método debe trasladarse al alquiler. Para hacer esto utilizo Método de movimiento. Con esto, primero copia el código en alquiler, lo ajusta para que quepa en su nuevo hogar y lo compila de la siguiente manera:

```
Alquiler de clase...
 doble obtenerCarga() {
 resultado doble = 0;
 cambiar (getMovie().getPriceCode()) {
 caso Película.REGULAR:
 resultado += 2;
 si (getDaysRented() > 2)
 resultado += (getDaysRented() - 2) * 1.5;  romper;
  caso Película.NEW_RELEASE:
 resultado += getDaysRented() * 3;
 romper;
 caso Película.NIÑOS:
 resultado += 1,5;
 si (getDaysRented() > 3)
 resultado += (getDaysRented() - 3) * 1.5;  romper;
 }
 resultado de devolución;
 }
```

En este caso adaptarse a su nuevo hogar supone eliminar el parámetro. También cambié el nombre del método mientras hacía el movimiento.

Ahora puedo probar para ver si este método funciona. Para ello reemplazo el cuerpo de `Cantidad.del.clientePara` para delegar al nuevo método.

```
Cliente de clase...
 importe doble privadoPara(Rental aRental) {
 return aRental.getCharge();
 }
```

Ahora puedo compilar y probar para ver si he roto algo.



El siguiente paso es encontrar todas las referencias al método anterior y ajustar la referencia para usar el nuevo método, de la siguiente manera:

```
Cliente de clase...
 declaración de cadena pública () {
 doble cantidad total = 0;
 int puntos de alquiler frecuentes = 0;
 Alquileres de enumeración = _rentals.elements();
 Resultado de cadena = "Registro de alquiler para " + getName() +
"\n";  mientras (alquileres.hasMoreElements()) {
 duplicar estaCantidad = 0;
 Alquiler cada = (Alquiler) alquileres.nextElement();

estaCantidad = cantidadPara(cada uno);

 // agregar puntos de inquilino frecuente
 puntos de alquiler frecuentes ++;
 // agrega bonificación por el alquiler de un nuevo lanzamiento por dos
días if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
 each.getDaysRented() > 1) frecuentesRenterPoints ++;

 //mostrar cifras de este alquiler
 resultado += "\t" + each.getMovie().getTitle()+ "\t" +
String.valueOf(thisAmount) + "\n";
 monto total += este monto;

 }
 //agregar líneas de pie de página
 resultado += "La cantidad adeuda es " + String.valueOf(totalAmount) +
"\n";
 resultado += "Ganaste" +
String.valueOf(frequentRenterPoints) +
 "puntos de inquilino frecuente";
 resultado de devolución;

 }
```

En este caso este paso es fácil porque acabamos de crear el método y está en un solo lugar. Sin embargo, en general, es necesario realizar una "búsqueda" en todas las clases que podrían

estar usando ese método:

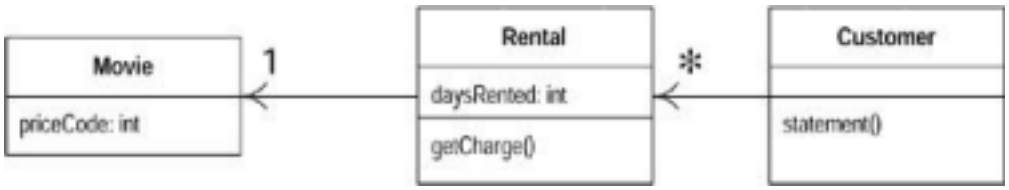
```
clase cliente
 declaración de cadena pública () {
 doble cantidad total = 0;
 int puntos de alquiler frecuentes = 0;
 Alquileres de enumeración = _rentals.elements();
 Resultado de cadena = "Registro de alquiler para " + getName() +
"\n";  mientras (alquileres.hasMoreElements()) {
 duplicar estaCantidad = 0;
 Alquiler cada = (Alquiler) alquileres.nextElement();

estaCantidad = cada.getCharge();

 // agregar puntos de inquilino frecuente
```



```
 puntos de alquiler frecuentes ++;
 // agrega bonificación por el alquiler de un nuevo lanzamiento por dos
días if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
 each.getDaysRented() > 1) frecuentesRenterPoints ++;

 //mostrar cifras de este alquiler
 resultado += "\t" + each.getMovie().getTitle()+ "\t" +
String.valueOf(thisAmount) + "\n";
 monto total += este monto;

 }
 //agregar líneas de pie de página
 resultado += "La cantidad adeuda es " + String.valueOf(totalAmount) +
"\n";
 resultado += "Ganaste " +
String.valueOf(frequentRenterPoints) +
 "puntos de inquilino frecuente";
 resultado de devolución;
 }
```

Cuando he hecho el cambio (Figura 1.3) lo siguiente es eliminar el método antiguo. El compilador debería decirme si me perdí algo. Luego pruebo para ver si he roto algo.

**Figura 1.3. Estado de las clases después de mover el método de carga.**




A veces dejo el método antiguo para delegar al nuevo método. Esto es útil si es un método público y no quiero cambiar la interfaz de la otra clase.

Ciertamente hay algo más que me gustaría hacer para `Alquiler.getCharge` pero lo dejaré por el momento y volveré a `Declaración.del.cliente.`

```
 declaración de cadena pública () {
 doble cantidad total = 0;
 int puntos de alquiler frecuentes = 0;
 Alquileres de enumeración = _rentals.elements();
 Resultado de cadena = "Registro de alquiler para " + getName() +
"\n";  mientras (alquileres.hasMoreElements()) {
 duplicar estaCantidad = 0;
 Alquiler cada = (Alquiler) alquileres.nextElement();

estaCantidad = cada.getCharge();

 // agregar puntos de inquilino frecuente
 puntos de alquiler frecuentes ++;
 // agrega bonificación por un alquiler de dos días de nueva versión
```



```
 if ((cada.getMovie().getPriceCode() == Película.NEW_RELEASE) &&
 each.getDaysRented() > 1) frecuentesRenterPoints ++;

 //mostrar cifras de este alquiler
 resultado += "\t" + each.getMovie().getTitle()+ "\t" +
String.valueOf(thisAmount) + "\n";
 monto total += este monto;

 }
 //agregar líneas de pie de página
 resultado += "La cantidad adeuda es " + String.valueOf(totalAmount) +
"\n";
 resultado += "Has ganado " + String.valueOf(frequentRenterPoints) +
 "puntos de inquilino frecuente";
 resultado de devolución;

 }
```

Lo siguiente que me llama la atención es que `esta cantidad` ahora es redundante. Se fija al resultado de `cada.carga` y no cambiarse después. Así puedo eliminar `esta cantidad` usando Reemplazar Temp con consulta:

```
 declaración de cadena pública () {
 doble cantidad total = 0;
 int puntos de alquiler frecuentes = 0;
 Alquileres de enumeración = _rentals.elements();
 Resultado de cadena = "Registro de alquiler para " + getName() +
"\n";  mientras (alquileres.hasMoreElements()) {
 Alquiler cada = (Alquiler) alquileres.nextElement();

 // agregar puntos de inquilino frecuente
 puntos de alquiler frecuentes ++;
 // agrega bonificación por el alquiler de un nuevo lanzamiento por dos
días if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
 each.getDaysRented() > 1) frecuentesRenterPoints ++;  //mostrar
```

```
cifras de este alquiler
 resultado += "\t" + each.getMovie().getTitle()+ "\t" +
String.valueOf
 (cada.getCharge()) + "\n";
 importe total += cada.getCharge();

 }
 //agregar líneas de pie de página
 resultado += "La cantidad adeuda es " + String.valueOf(totalAmount) +
"\n";
 resultado += "Has ganado " + String.valueOf(frequentRenterPoints) +
"puntos de inquilino frecuente";
 resultado de devolución;

 }

 }
```



Una vez que he realizado ese cambio, compilo y pruebo para asegurarme de que no he roto nada.

Me gusta deshacerme de variables temporales como esta tanto como sea posible. Las temperaturas suelen ser un problema porque hacen que se pasen muchos parámetros cuando no es necesario. Es fácil perder de vista para qué están ahí. Son particularmente insidiosos en métodos largos. Por supuesto, hay que pagar un precio por el rendimiento; aquí el cargo ahora se calcula dos veces. Pero es fácil optimizar eso en la clase de alquiler y se puede optimizar mucho más eficazmente cuando el código se factoriza correctamente. Hablaré más sobre ese tema más adelante en Refactorización y rendimiento en la página 69.

## Extracción de puntos de inquilino frecuente

El siguiente paso es hacer algo similar con los puntos de inquilinos frecuentes. Las reglas varían con la cinta, aunque hay menos variación que con la carga. Parece razonable echar la responsabilidad al alquiler. Primero necesitamos usar Método de extracción en la parte del código de puntos de inquilino frecuente (en negrita):

```
declaración de cadena pública () {
 doble cantidad total = 0;
 int puntos de alquiler frecuentes = 0;
 Alquileres de enumeración = _rentals.elements();
 Resultado de cadena = "Registro de alquiler para " + getName() +
"\n";  mientras (alquileres.hasMoreElements()) {
 Alquiler cada = (Alquiler) alquileres.nextElement();

 // agregar puntos de inquilino frecuente
 puntos de alquiler frecuentes ++;
 // agrega bonificación por un alquiler de nueva versión de dos días
if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
each.getDaysRented() > 1) frecuentaRenterPoints ++;

 //mostrar cifras de este alquiler
 resultado += "\t" + each.getMovie().getTitle()+ "\t" +
String.valueOf(each.getCharge())
+ "\n";
```

```
 importe total += cada.getCharge();

 }
 //agregar líneas de pie de página
 resultado += "La cantidad adeuda es " + String.valueOf(totalAmount) +
"\n";
 resultado += "Has ganado " + String.valueOf(frequentRenterPoints) +
"puntos de inquilino frecuente";
 resultado de devolución;

 }
}
```

Nuevamente analizamos el uso de variables de ámbito local. De nuevo `cada` se utiliza y se puede pasar como parámetro. La otra temperatura utilizada es `Puntos de alquiler frecuentes.` En este caso `puntos de alquiler frecuentes` tiene un valor de antemano. Sin embargo, el cuerpo del método extraído no lee el valor, por lo que no necesitamos pasarlo como parámetro siempre que usemos una asignación adjunta.



Hice la extracción, compilé y probé y luego hice un movimiento, compilé y probé nuevamente. Con la refactorización, los pequeños pasos son los mejores; de esa manera menos cosas tienden a salir mal.

```
Cliente de clase...
 declaración de cadena pública () {
 doble cantidad total = 0;
 int puntos de alquiler frecuentes = 0;
 Alquileres de enumeración = _rentals.elements();
 Resultado de cadena = "Registro de alquiler para " + getName() +
"\n";  mientras (alquileres.hasMoreElements()) {
 Alquiler cada = (Alquiler) alquileres.nextElement();
PuntosRentafrecuentes += cada.getPuntosRentantefrecuentes();

 //mostrar cifras de este alquiler
 resultado += "\t" + each.getMovie().getTitle()+ "\t" +
String.valueOf(each.getCharge()) + "\n";  importe total +=
cada.getCharge();
 }

 //agregar líneas de pie de página
 resultado += "La cantidad adeuda es " + String.valueOf(totalAmount) +
"\n";
 resultado += "Has ganado " + String.valueOf(frequentRenterPoints) +
 "puntos de inquilino frecuente";
 resultado de devolución;
 }

Alquiler de clase...
 int getPuntosRentadoresfrecuentes() {
 if ((getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
getDaysRented() > 1)
 devolver 2;
```

```
demás
devolver 1;
}
```

Resumiré los cambios que acabo de realizar con algunos diagramas del lenguaje de modelado unificado (UML) del antes y el después (Figuras 1.4 a 1.7). Nuevamente los diagramas de la izquierda son anteriores al cambio;  los de la derecha están detrás del cambio.

**Figura 1.4. Diagrama de clases antes de la extracción y movimiento del cálculo de puntos de inquilino frecuente.**

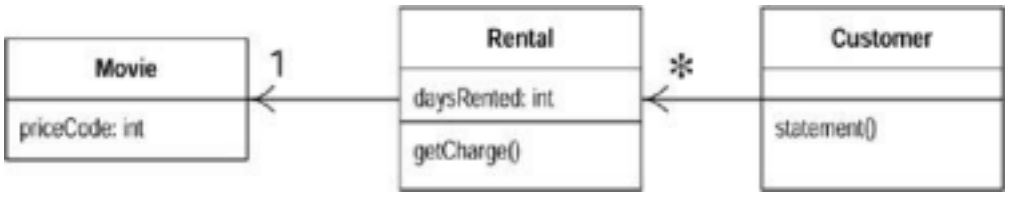



**Figura 1.5. Diagramas de secuencia antes de la extracción y movimiento del cálculo de puntos de inquilino frecuente.**



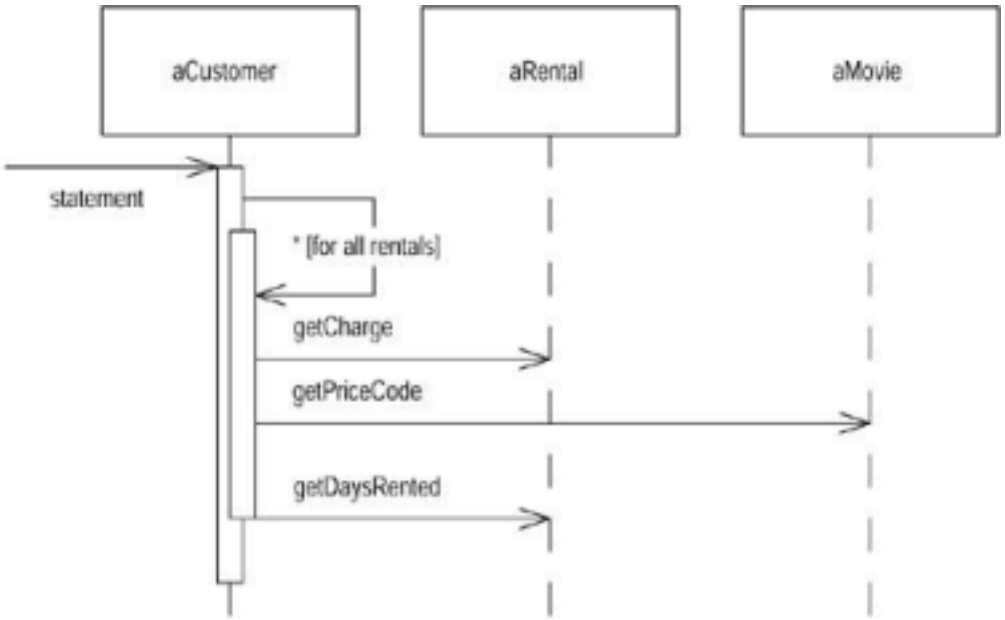



**Figura 1.6. Diagrama de clases después de la extracción y movimiento del cálculo de puntos de inquilino frecuente**

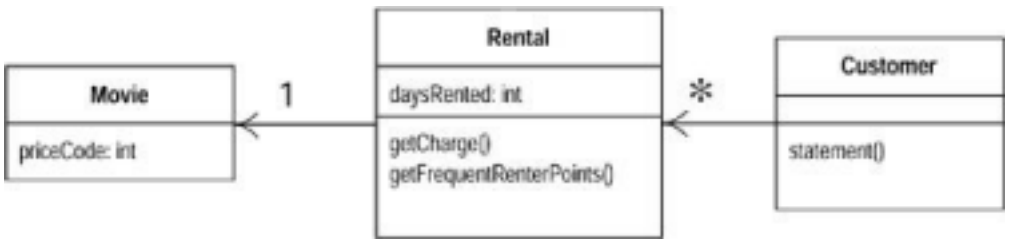
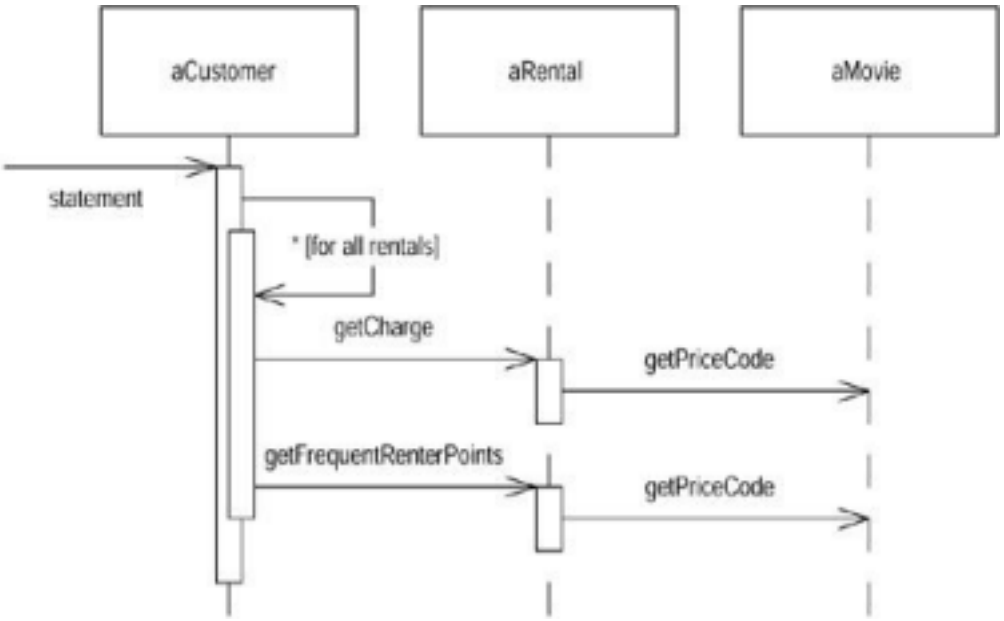



**Figura 1.7. Diagrama de secuencia antes de la extracción y movimiento del cálculo de**

**puntos de inquilino frecuente.**






## Eliminación de temperaturas

Como sugerí antes, las variables temporales pueden ser un problema. Son útiles sólo dentro de su propia rutina y, por lo tanto, fomentan rutinas largas y complejas. En este caso tenemos dos variables temporales, las cuales se utilizan para obtener un total de los alquileres adjuntos al cliente. Tanto la versión ASCII como la HTML requieren estos totales. me gusta usar Reemplazar temperatura con consulta reemplazar `monto total` y `puntos de alquiler frecuentes` con métodos de consulta. Las consultas son accesibles para cualquier método de la clase y, por lo tanto, fomentan un diseño más limpio sin métodos largos y complejos:

```
Cliente de clase...
```

```
 declaración de cadena pública () {
 doble cantidad total = 0;
 int puntos de alquiler frecuentes = 0;
 Alquileres de enumeración = _rentals.elements();
 Resultado de cadena = "Registro de alquiler para " + getName() +
"\n";  mientras (alquileres.hasMoreElements()) {
 Alquiler cada = (Alquiler) alquileres.nextElement();
PuntosRentafrecuentes += cada.getPuntosRentantefrecuentes();

 //mostrar cifras de este alquiler
 resultado += "\t" + each.getMovie().getTitle()+ "\t" +
String.valueOf(each.getCharge()) + "\n";  importe total +=
cada.getCharge();
 }

 //agregar líneas de pie de página
 resultado += "La cantidad adeuda es " + String.valueOf(totalAmount) +
"\n";
 resultado += "Has ganado " + String.valueOf(frequentRenterPoints) +
 "puntos de inquilino frecuente";
 resultado de devolución;
```



```
}
```

Empecé reemplazando `monto total` con un `cargar` método en el cliente:

```
Cliente de clase...

 declaración de cadena pública () {
 int puntos de alquiler frecuentes = 0;
 Alquileres de enumeración = _rentals.elements();
 Resultado de cadena = "Registro de alquiler para " + getName() +
"\n";  mientras (alquileres.hasMoreElements()) {
 Alquiler cada = (Alquiler) alquileres.nextElement();
PuntosRentafrecuentes += cada.getPuntosRentantefrecuentes();

 //mostrar cifras de este alquiler
 resultado += "\t" + each.getMovie().getTitle()+ "\t" +
String.valueOf(each.getCharge()) + "\n";  }

 //agregar líneas de pie de página
 resultado += "La cantidad adeuda es " +
Cadena.valorDe(obtenerCargaTotal()) + "\n";
 resultado += "Has ganado " + String.valueOf(frequentRenterPoints) +
 "puntos de inquilino frecuente";
 resultado de devolución;
 }

 doble privado getTotalCharge() {
 resultado doble = 0;
 Alquileres de enumeración = _rentals.elements();
 mientras (alquileres.hasMoreElements()) {
 Alquiler cada = (Alquiler) alquileres.nextElement();
```

```
resultado += cada.getCharge();
 }
 resultado de devolución;
 }
```

Este no es el caso más simple de Reemplazar temperatura con consulta `monto total` fue asignado dentro del bucle, por lo que tengo que copiar el bucle en el método de consulta.

Después de compilar y probar esa refactorización, hice lo mismo para `Puntos de alquiler frecuentes:`

```
 Cliente de clase...
 declaración de cadena pública () {
 int puntos de alquiler frecuentes = 0;
 Alquileres de enumeración = _rentals.elements();
 Resultado de cadena = "Registro de alquiler para " + getName() +
"\n";  mientras (alquileres.hasMoreElements()) {
 Alquiler cada = (Alquiler) alquileres.nextElement();
PuntosRentafrecuentes += cada.getPuntosRentantefrecuentes();

 //mostrar cifras de este alquiler
```



```
 resultado += "\t" + each.getMovie().getTitle()+ "\t" +
String.valueOf(each.getCharge()) + "\n";  }

 //agregar líneas de pie de página
 resultado += "La cantidad adeuda es " +
String.valueOf(getTotalCharge()) + "\n";
 resultado += "Has ganado " + String.valueOf(frequentRenterPoints) +
 "puntos de inquilino frecuente";
 resultado de devolución;
 }
 declaración de cadena pública () {
 Alquileres de enumeración = _rentals.elements();
 Resultado de cadena = "Registro de alquiler para " + getName() +
"\n";  mientras (alquileres.hasMoreElements()) {
 Alquiler cada = (Alquiler) alquileres.nextElement();

 //mostrar cifras de este alquiler
 resultado += "\t" + each.getMovie().getTitle()+ "\t" +
String.valueOf(each.getCharge()) + "\n";  }

 //agregar líneas de pie de página
 resultado += "La cantidad adeuda es " +
String.valueOf(getTotalCharge()) + "\n";
 resultado += "Ganaste" +
Cadena.valorDe(getTotalFrequentRenterPoints()) +
 "puntos de inquilino frecuente";
 resultado de devolución;
 }

 privado int getTotalFrequentRenterPoints(){
 resultado entero = 0;
```

```
 Alquileres de enumeración = _rentals.elements();
 mientras (alquileres.hasMoreElements()) {
 Alquiler cada = (Alquiler) alquileres.nextElement();
resultado += cada.getFrequentRenterPoints();
 }
 resultado de devolución;
 }
```

Las figuras 1.8 a 1.11 muestran el cambio de estas refactorizaciones en los diagramas de clases y el diagrama de interacción para el método de declaración.

**Figura 1.8. Diagrama de clases antes de la extracción de los totales.**

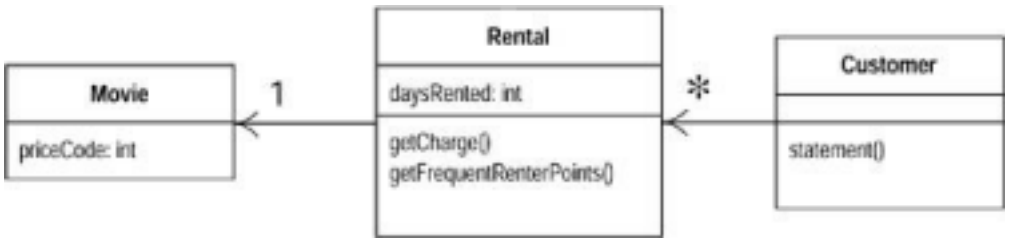





**Figura 1.9. Diagrama de secuencia antes de la extracción de los totales.**

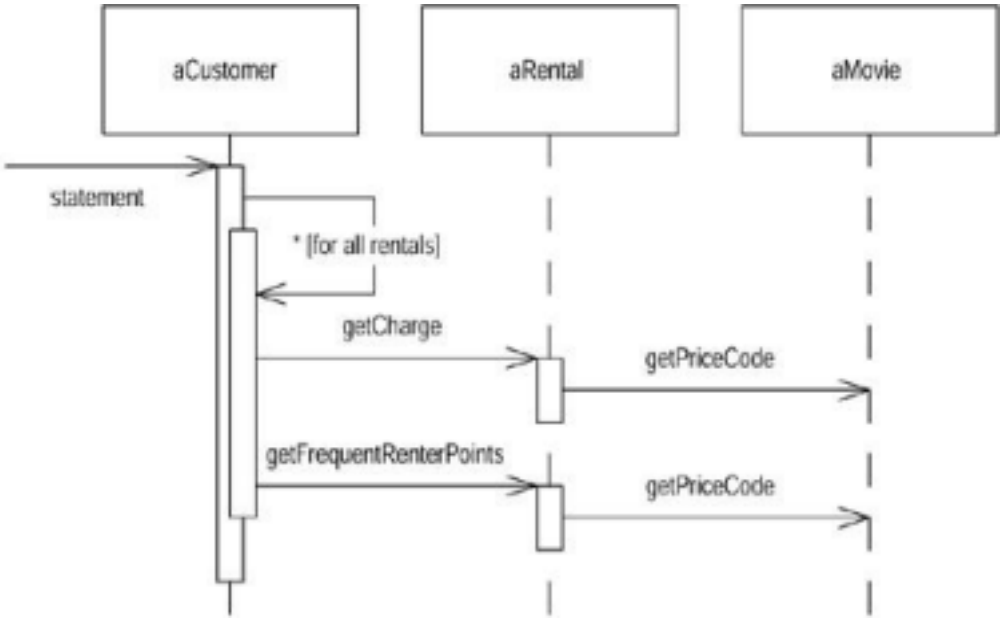



**Figura 1.10. Diagrama de clases después de la extracción de los totales.**

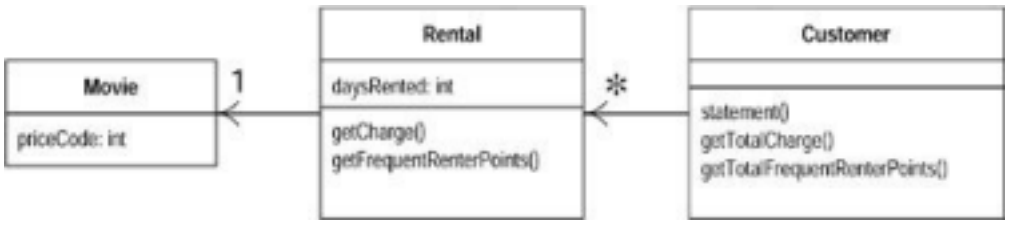
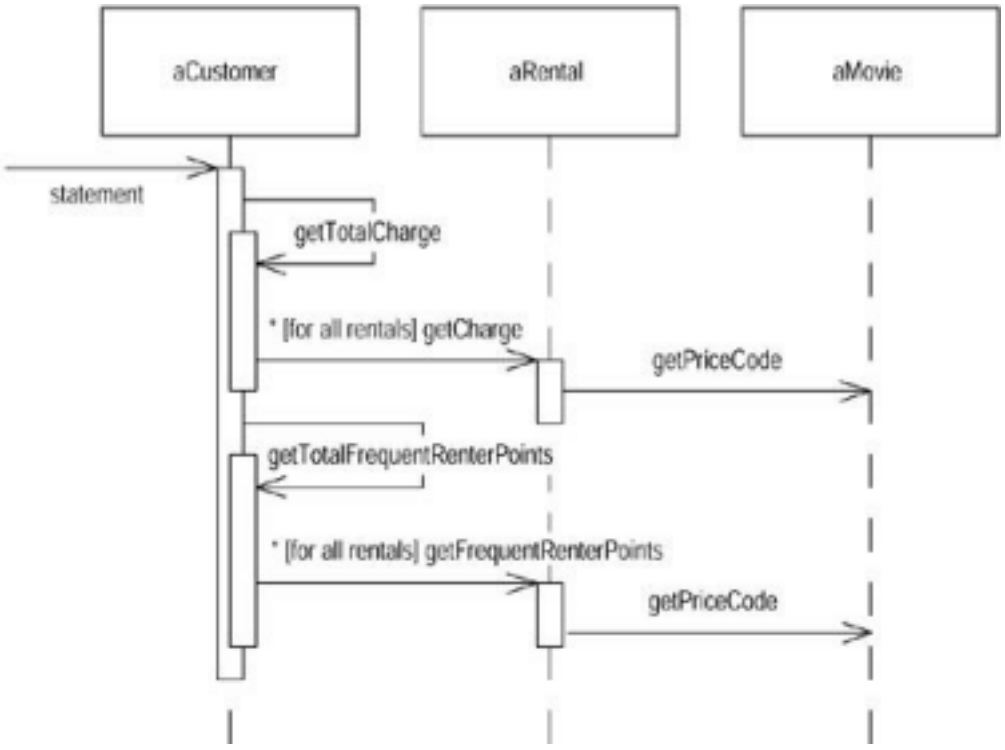



**Figura 1.11. Diagrama de secuencia después de la extracción de los totales.**




Vale la pena detenerse a pensar un poco en la última refactorización. La mayoría de las refactorizaciones reducen la cantidad de código, pero ésta la aumenta. Esto se debe a que Java 1.1 requiere muchas declaraciones para configurar un ciclo de suma. Incluso un bucle de suma simple con una línea de código por elemento necesita seis líneas de soporte a su alrededor. Es un modismo que es obvio para cualquier programador pero que de todos modos son muchas líneas.

La otra preocupación con esta refactorización radica en el rendimiento. El código antiguo ejecutó el bucle " while " una vez, el código nuevo lo ejecuta tres veces. Un bucle while que demore mucho tiempo podría afectar el rendimiento. Muchos programadores no harían esta

refactorización simplemente por este motivo. Pero tenga en cuenta las palabras *si* y *podría.* Hasta que no cree el perfil, no puedo saber cuánto tiempo se necesita para que el bucle se calcule o si el bucle se llama con suficiente frecuencia como para afectar el rendimiento general del sistema. No se preocupe por esto mientras refactoriza. Cuando optimice tendrá que preocuparse por ello, pero entonces estará en una posición mucho mejor para hacer algo al respecto y tendrá más opciones para optimizar de manera efectiva (consulte la discusión en la página 69).

Estas consultas ahora están disponibles para cualquier código escrito en la clase de cliente. Se pueden agregar fácilmente a la interfaz de la clase en caso de que otras partes del sistema necesiten esta información. Sin consultas como estas, otros métodos tienen que lidiar con conocer los alquileres y construir los bucles. En un sistema complejo, eso conducirá a mucho más código que escribir y mantener.

Puedes ver la diferencia inmediatamente con el `htmlDeclaración.` Ahora estoy en el punto en el que me quito el sombrero de refactorización y me pongo el sombrero de función de adición. puedo escribir `htmlDeclaración` como sigue y agregue las pruebas apropiadas:

```
cadena pública htmlStatement() {
 Alquileres de enumeración = _rentals.elements();
 Resultado de cadena = "<H1>Alquileres para <EM>" + getName() +
"</EM></ H1><P>\n";
 mientras (alquileres.hasMoreElements()) {
 Alquiler cada = (Alquiler) alquileres.nextElement();
```



```
 //mostrar cifras para cada alquiler
 resultado += cada.getMovie().getTitle()+ ": " +
 String.valueOf(each.getCharge()) + "<BR>\n";  }
 //agregar líneas de pie de página
 resultado += "<P>Debes <EM>" + String.valueOf(getTotalCharge()) +
"</EM><P>\n";
 resultado += "En este alquiler obtuviste <EM>" +
 String.valueOf(getTotalFrequentRenterPoints()) + "</EM>
puntos de inquilino frecuente<P>";
 resultado de devolución;
 }
```

Al extraer los cálculos puedo crear el `htmlDeclaración` método y reutilice todo el código de cálculo que estaba en el método de declaración original. No copié ni pegué, por lo que si las reglas de cálculo cambian, solo tengo un lugar en el código al que ir. Cualquier otro tipo de declaración será realmente rápida y sencilla de preparar. La refactorización no tomó mucho tiempo. Pasé la mayor parte del tiempo averiguando qué hacía el código y habría tenido que hacerlo de todos modos.

Parte del código se copia de la versión ASCII, principalmente debido a la configuración del bucle. Una refactorización adicional podría limpiar eso. Los métodos de extracción para encabezado, pie de página y línea de detalle son una ruta que podría tomar. Puedes ver cómo hacer esto en el ejemplo de Método de plantilla de formulario. Pero ahora los usuarios vuelven a clamar. Se están preparando para cambiar la clasificación de las películas en la tienda. Todavía no está claro qué cambios quieren hacer, pero parece que se introducirán nuevas clasificaciones y las existentes bien podrían cambiarse. Los cargos y las asignaciones de puntos de inquilinos frecuentes para estas clasificaciones están por decidir. Por el momento, hacer este tipo de

cambios resulta incómodo. Tengo que entrar en los métodos de cargos y puntos de inquilinos frecuentes y modificar el código condicional para realizar cambios en las clasificaciones de películas. De vuelta con el sombrero de refactorización.

## Reemplazo de la lógica condicional del código de precio con polimorfismo

La primera parte de este problema es esa declaración de cambio. Es una mala idea hacer un cambio basado en un atributo de otro objeto. Si debe utilizar una declaración de cambio, debe hacerlo con sus propios datos, no con los de otra persona.

```
Alquiler de clase...
 doble obtenerCarga() {
 resultado doble = 0;
 cambiar (getMovie().getPriceCode()) {
 caso Película.REGULAR:
 resultado += 2;
 si (getDaysRented() > 2)
 resultado += (getDaysRented() - 2) * 1.5;  romper;
 caso Película.NEW_RELEASE:
 resultado += getDaysRented() * 3;
 romper;
 caso Película.NIÑOS:
 resultado += 1,5;
 si (getDaysRented() > 3)
 resultado += (getDaysRented() - 3) * 1.5;  romper;
  }
```



```
resultado de devolución;
 }
```

Esto implica que `obtener carga` debería pasar a la película:

```
clase de película...
 doble getCharge(int díasAlquilado) {
 resultado doble = 0;
 cambiar (getPriceCode()) {
 caso Película.REGULAR:
 resultado += 2;
 si (díasAlquilado > 2)
 resultado += (díasAlquilado - 2) * 1,5;
 romper;
 caso Película.NEW_RELEASE:
 resultado += díasAlquilado * 3;
 romper;
 caso Película.NIÑOS:
 resultado += 1,5;
 si (díasAlquilado > 3)
 resultado += (díasAlquilado - 3) * 1,5;
 romper;
```

```
}
resultado de devolución;
}
```

Para que esto funcione, tuve que pasar la duración del alquiler, que por supuesto son datos del alquiler. El método utiliza eficazmente dos datos: la duración del alquiler y el tipo de película. ¿Por qué prefiero pasar la duración del alquiler a la película en lugar del tipo de película al alquiler? Es porque los cambios propuestos tienen que ver con agregar nuevos tipos. La información de tipo generalmente tiende a ser más volátil. Si cambio el tipo de película, quiero el menor efecto dominó, por lo que prefiero calcular la carga dentro de la película.

Compilé el método en una película y luego cambié el `obtener carga` en alquiler para utilizar el nuevo método (Figuras 1.12 y 1.13):

**Figura 1.12. Diagrama de clases antes de mover métodos a la película.**

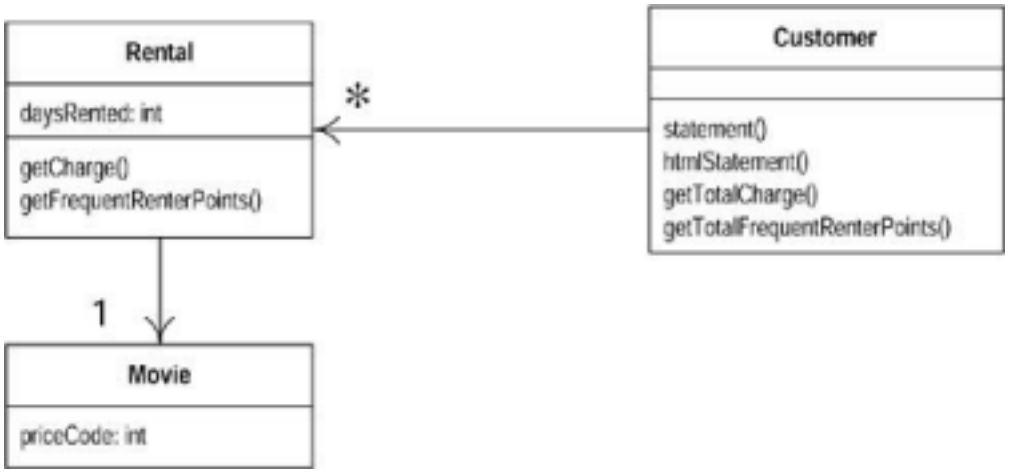
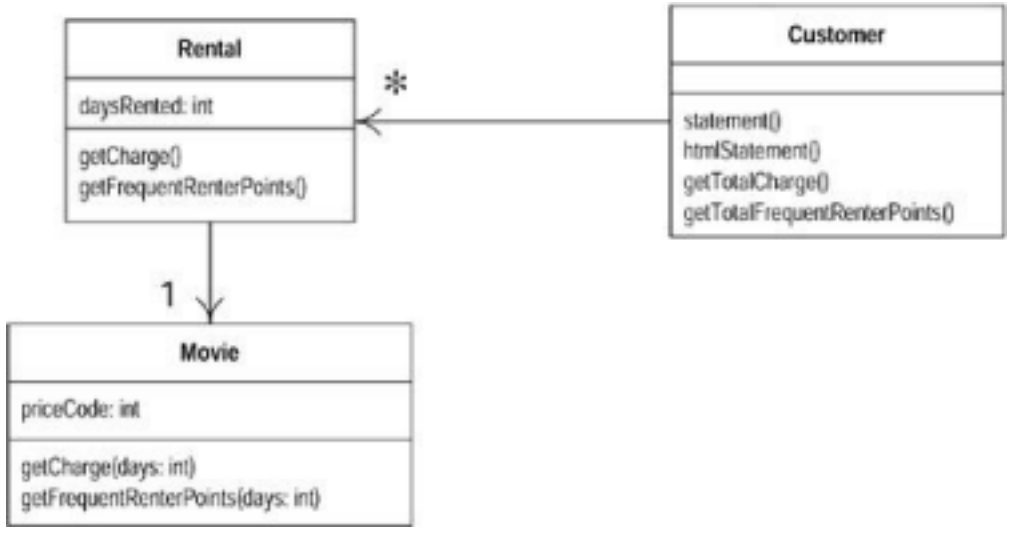





**Figura 1.13. Diagrama de clases después de mover métodos a la película**




```
Alquiler de clase...
doble obtenerCarga() {
return _movie.getCharge(_daysRented);
}
```

Una vez que he movido el `obtener carga` método, haré lo mismo con el cálculo de puntos de

inquilino frecuente. Eso mantiene juntas ambas cosas que varían con el tipo en la clase que tiene el tipo:

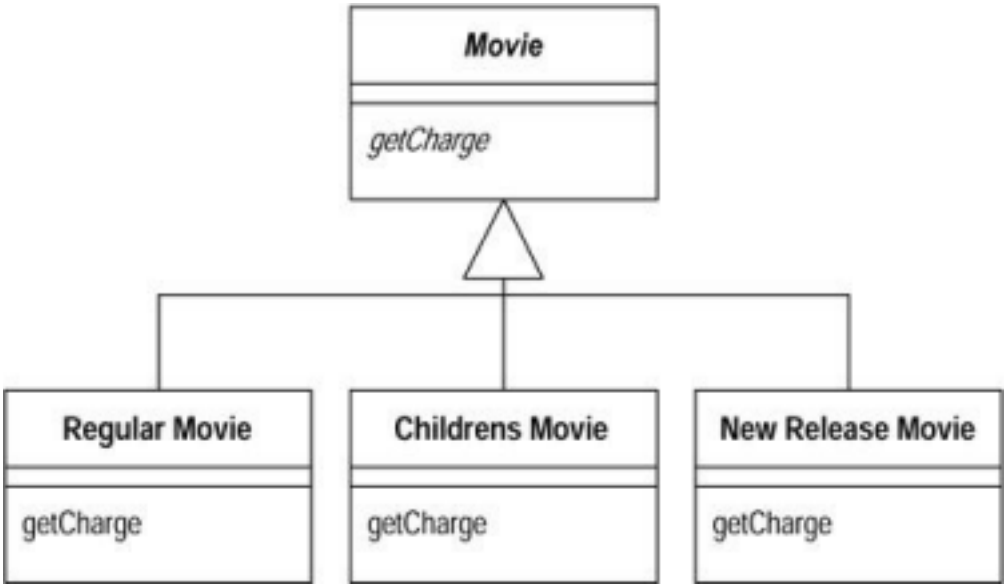
```
 Alquiler de clase...
  int getPuntosRentadoresfrecuentes() {
  if ((getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
getDaysRented() > 1)
  devolver 2;
  demás
  devolver 1;
  }

Alquiler de clases...
  int getPuntosRentadoresfrecuentes() {
  return _movie.getFrequentRenterPoints(_daysRented);  }
película de clase...


  int getFrequentRenterPoints(int díasAlquilados) {
  if ((getPriceCode() == Movie.NEW_RELEASE) && díasAlquilado > 1)
return 2;
  demás
  devolver 1;
  }
```

## Por fin… Herencia



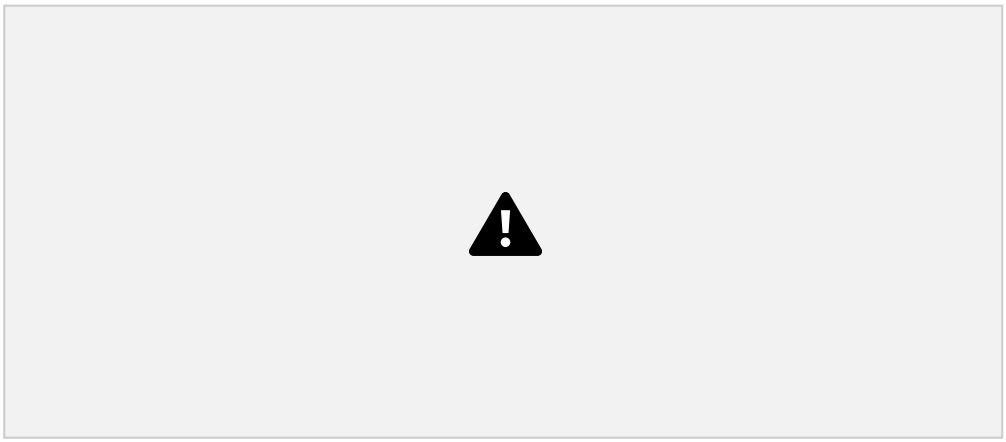
Disponemos de varios tipos de películas que tienen diferentes formas de responder a una misma pregunta. Esto suena como un trabajo para subclases. Podemos tener tres subclases de película, cada una de las cuales puede tener su propia versión de carga (Figura 1.14).

**Figura 1.14. Usando la herencia en la película**




Esto me permite reemplazar la declaración de cambio usando polimorfismo. Lamentablemente tiene un pequeño defecto: no funciona. Una película puede cambiar su clasificación durante su vida. Un objeto no puede cambiar de clase durante su vida. Sin embargo, existe una solución: el patrón estatal [la Banda de los Cuatro]. Con el patrón Estado las clases se ven como Figura 1.15.

**Figura 1.15. Usando el patrón Estado en una película**



Al agregar la dirección indirecta podemos hacer la subclasificación del objeto de código de precio y cambiar el precio cuando sea necesario.

Si está familiarizado con los patrones de la Banda de los Cuatro, puede preguntarse: "¿Es esto un estado o es una estrategia?" ¿La clase de precio representa un algoritmo para calcular el precio (en cuyo caso prefiero llamarlo Pricer o PricingStrategy), o representa un estado de la película (*Viaje a las estrellas X* es



una nueva versión). En esta etapa, la elección del patrón (y el nombre) refleja cómo desea pensar sobre la estructura. Por el momento estoy pensando en esto como un estado de la película. Si luego decido que una estrategia comunica mejor mi intención, la refactorizaré

cambiando los nombres.

Para introducir el patrón de estado, utilizaré tres refactorizaciones. Primero, moveré el comportamiento del código de tipo al patrón de estado con Reemplazar código de tipo con estado/estrategia. Entonces puedo usar Mover Método para mover la declaración de cambio a la clase de precio. Finalmente usaré Reemplazar condicional con polimorfismo para eliminar la declaración de cambio.

empiezo con Reemplazar código de tipo con estado/estrategia. Este primer paso es utilizar Ser Encapsular campo en el código de tipo para garantizar que todos los usos del código de tipo pasen por métodos de obtención y configuración. Debido a que la mayor parte del código proviene de otras clases, la mayoría de los métodos ya utilizan el método de obtención. Sin embargo, los constructores sí acceden al código de precio:

```
clase de película...
 Película pública (nombre de cadena, int código de precio) {
 _nombre = nombre;
 _priceCode = precioCode;
 }
```

Puedo usar el método de configuración en su lugar.

```
clase de película
 Película pública (nombre de cadena, int código de precio) {
 _nombre = nombre;
 setPriceCode(precioCódigo);
 }
```

Compilo y pruebo para asegurarme de no romper nada. Ahora agrego las nuevas clases. Proporciono el comportamiento del código de tipo en el objeto de precio. Hago esto con un método abstracto sobre precio y métodos concretos en las subclases:

```
precio de clase abstracta {
 abstracto int getPriceCode();
 }
 clase ChildrensPrice extiende Precio {
 int obtenerCódigoPrecio() {
 volver Película.NIÑOS;
 }
 }
 clase NewReleasePrice extiende Precio {
 int obtenerCódigoPrecio() {
 devolver Película.NEW_RELEASE;
 }
 }
 clase Precio Regular extiende Precio {
 int obtenerCódigoPrecio() {
 devolver Película.REGULAR;
 }
 }
```

Puedo compilar las nuevas clases en este punto.

Ahora necesito cambiar los accesores de la película para que el código de precio use la nueva clase:

```
public int getPriceCode() {
 devolver _priceCode;
 }
 setPriceCode público (int arg) {
 _priceCode = arg;
 }
 privado int _priceCode;
```

Esto significa reemplazar el campo del código de precio con un campo de precio y cambiar los accesores:

```
clase de película...
 public int getPriceCode() {
 return _price.getPriceCode();
 }
 setPriceCode público vacío (int arg) {
 cambiar (arg) {
 caso REGULAR:
 _precio = nuevo Precio Regular();
 romper;
 caso NIÑOS:
 _precio = nuevoPrecioNiños();
 romper;
 caso NEW_RELEASE:
 _precio = nuevo Precio de nuevo lanzamiento();
 romper;
 por defecto:
 throw new IllegalArgumentException("Código de precio incorrecto");  }
 }
 precio privado _precio;
```

Ahora puedo compilar y probar, y los métodos más complejos no se dan cuenta de que el

mundo ha cambiado. Ahora aplico Método de movimiento a `obtener cargo:`

```
clase de película...
doble getCharge(int díasAlquilado) {
resultado doble = 0;
cambiar (getPriceCode()) {
caso Película.REGULAR:
resultado += 2;
si (díasAlquilado > 2)
```

```
resultado += (díasAlquilado - 2) * 1,5;  romper;
caso Película.NEW_RELEASE:
resultado += díasAlquilado * 3;
romper;
caso Película.NIÑOS:
```



```
resultado += 1,5;
si (díasAlquilado > 3)
resultado += (díasAlquilado - 3) * 1,5;  romper;
}
resultado de devolución;
}
```

Es sencillo de mover:

```
clase de película...
doble getCharge(int díasAlquilado) {
return _price.getCharge(díasAlquilado);  }

clase Precio...
doble getCharge(int díasAlquilado) {
resultado doble = 0;
cambiar (getPriceCode()) {
caso Película.REGULAR:
resultado += 2;
si (díasAlquilado > 2)
resultado += (díasAlquilado - 2) * 1,5;  romper;
caso Película.NEW_RELEASE:
resultado += díasAlquilado * 3;
romper;
caso Película.NIÑOS:
resultado += 1,5;
si (díasAlquilado > 3)
resultado += (díasAlquilado - 3) * 1,5;  romper;
}
resultado de devolución;
}
```

Una vez movido puedo empezar a usarlo. Reemplazar condicional con polimorfismo:

```
clase Precio...
doble getCharge(int díasAlquilado) {
resultado doble = 0;
cambiar (getPriceCode()) {
caso Película.REGULAR:
resultado += 2;
si (díasAlquilado > 2)
 resultado += (díasAlquilado - 2) * 1,5;  romper;
caso Película.NEW_RELEASE:
resultado += díasAlquilado * 3;
romper;
caso Película.NIÑOS:
```

```
resultado += 1,5;
si (díasAlquilado > 3)
resultado += (díasAlquilado - 3) * 1,5;
```



```
romper;
}
resultado de devolución;
}
```

Hago esto tomando una parte de la declaración del caso a la vez y creando un método primordial. empiezo con `Precio Regular:`

```
clase RegularPrecio...
doble getCharge(int díasAlquilado){
resultado doble = 2;
si (díasAlquilado > 2)
resultado += (díasAlquilado - 2) * 1,5;
resultado de devolución;
}
```

Esto anula la declaración de caso principal, que dejo como está. Compilo y pruebo para este caso, luego tomo el siguiente tramo, compilo y pruebo. (Para asegurarme de que estoy ejecutando el código de la subclase, me gusta incluir un error deliberado y ejecutarlo para asegurar que las pruebas exploten. No es que esté paranoico ni nada por el estilo).

```
clase InfantilPrecio
doble getCharge(int díasAlquilado){
resultado doble = 1,5;
si (díasAlquilado > 3)
resultado += (díasAlquilado - 3) * 1,5;
resultado de devolución;
}

clase Nuevo Lanzamiento Precio...
doble getCharge(int díasAlquilado){
días de regresoAlquilado * 3;
}
```

Cuando haya hecho eso con todas las piernas, declaro el `Precio.getCharge` resumen del método:

```
clase Precio...
resumen doble getCharge(int díasAlquilado);
```

Ahora puedo hacer el mismo procedimiento con `obtener puntos de inquilino frecuentes:`

```
Alquiler de clase...
int getFrequentRenterPoints(int díasAlquilados) {
```

```
 if ((getPriceCode() == Movie.NEW_RELEASE) && díasAlquilado > 1)
return 2;
 demás
 devolver 1;
 }
```

Primero muevo el método a Precio:



```
Película de clase...
 int getFrequentRenterPoints(int díasAlquilados) {
 return _price.getFrequentRenterPoints(díasAlquilado);  }
 Precio de clase...
 int getFrequentRenterPoints(int díasAlquilados) {
 if ((getPriceCode() == Movie.NEW_RELEASE) && díasAlquilado > 1)
return 2;
 demás
 devolver 1;
 }
```

En este caso, sin embargo, no hago abstracto el método de la superclase. En lugar de eso, creo un método primordial para las nuevas versiones y dejo un método definido (como predeterminado) en la superclase:

```
Clase Nuevo Lanzamiento Precio
 int getFrequentRenterPoints(int díasAlquilados) {
 retorno (díasAlquilado > 1)? 2: 1;
 }

 Precio de clase...
 int getFrequentRenterPoints(int díasAlquilados){
 devolver 1;
 }
```

Poner el patrón estatal fue un gran esfuerzo. ¿Valió la pena? La ganancia es que si cambio el comportamiento del precio, agrego nuevos precios o agrego un comportamiento adicional dependiente del precio, el cambio será mucho más fácil de realizar. El resto de la aplicación no conoce el uso del patrón de estado. Por el pequeño comportamiento que tengo actualmente, no es gran cosa. En un sistema más complejo con aproximadamente una docena de métodos dependientes del precio, esto supondría una gran diferencia. Todos estos cambios fueron pequeños pasos. Parece lento escribirlo de esta manera, pero ni una sola vez tuve que abrir el depurador, por lo que el proceso fluyó bastante rápido. Me tomó mucho más tiempo escribir esta sección del libro que cambiar el código.

Ya he completado la segunda refactorización importante. Será mucho más fácil cambiar la estructura de clasificación de las películas y alterar las reglas de cobro y el sistema de puntos para inquilinos frecuentes. Figuras 1.16 y 1.17 Muestre cómo funciona el patrón estatal con información de precios.

**Figura 1.16. Interacciones usando el patrón de estado.**

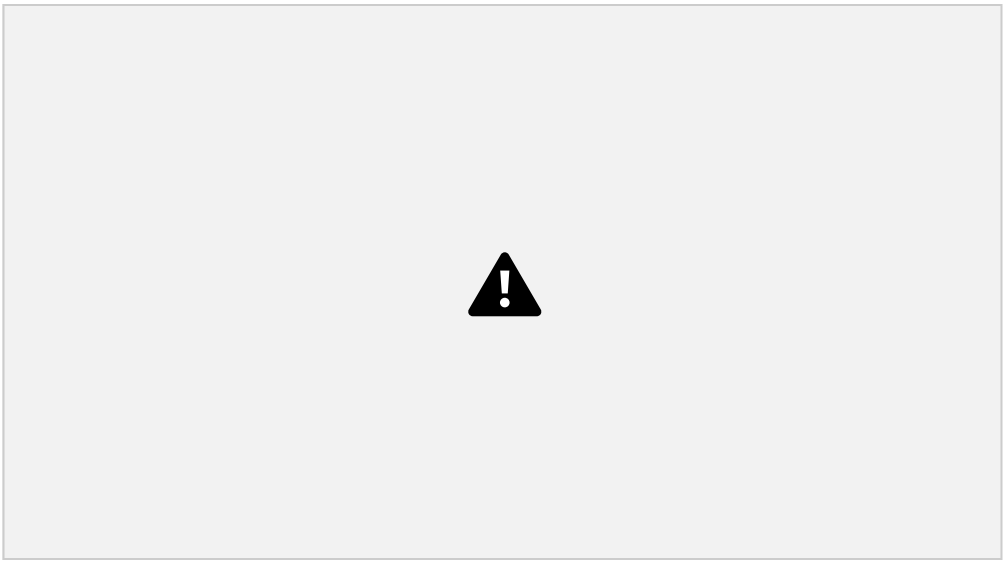
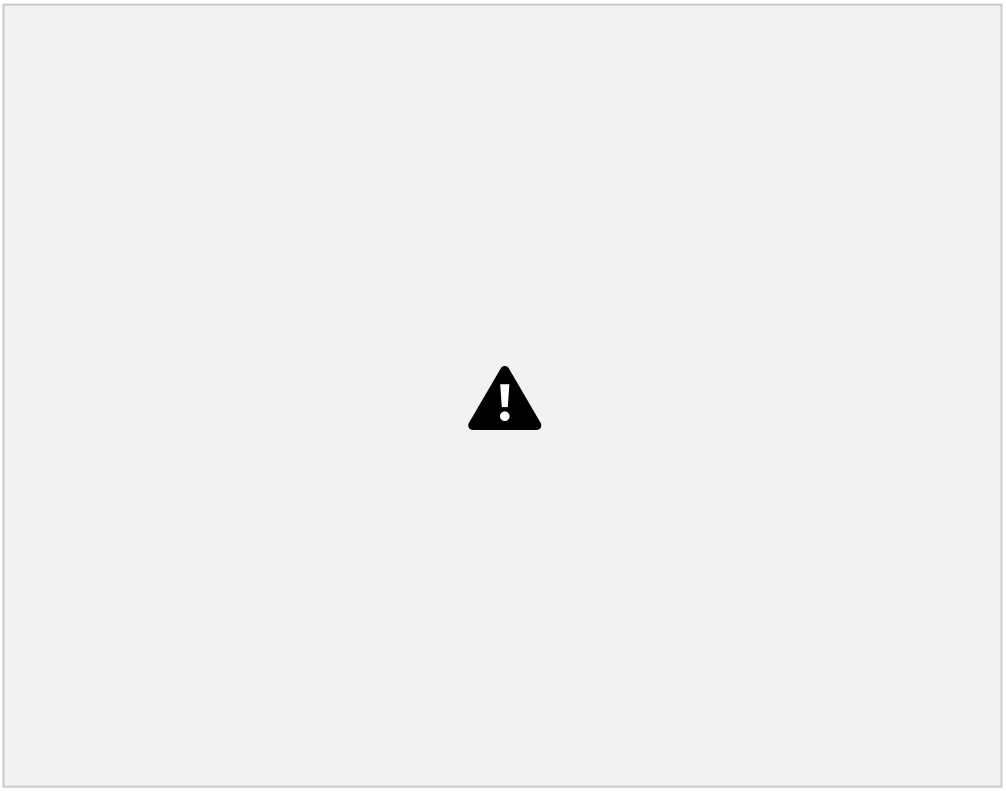


**Figura 1.17. Diagrama de clases después de agregar el patrón de estado**



## Pensamientos finales

Este es un ejemplo simple, pero espero que le dé una idea de cómo es la refactorización. He usado varias refactorizaciones, incluyendo Método de extracción, Método de movimiento, y Reemplazar condicional con polimorfismo. Todo esto conduce a responsabilidades mejor distribuidas y a un código más fácil de mantener. Se ve bastante diferente del código de estilo de procedimiento, y es necesario algo de tiempo para acostumbrarse. Pero una vez que te acostumbras, es difícil volver a los programas de procedimientos.



La lección más importante de este ejemplo es el ritmo de la refactorización: prueba, pequeño cambio, prueba, pequeño cambio, prueba, pequeño cambio. Es ese ritmo el que permite que la refactorización avance de forma rápida y segura.

Si has llegado hasta aquí conmigo, deberías entender de qué se trata la refactorización. Ahora podemos pasar a algunos antecedentes, principios y teoría (¡aunque no demasiado!)

# Capítulo 2. Principios de la refactorización

El ejemplo anterior debería haberle dado una buena idea de lo que significa la refactorización. Ahora es el momento de dar un paso atrás y observar los principios clave de la refactorización y

algunas de las cuestiones en las que debe pensar al utilizar la refactorización.

## Definición de refactorización

Siempre desconfío un poco de las definiciones porque cada uno tiene las suyas, pero cuando escribes un libro puedes elegir tus propias definiciones. En este caso baso mis definiciones en el trabajo realizado por el grupo de Ralph Johnson y sus diversos asociados.

Lo primero que hay que decir al respecto es que la palabra *Refactorización* tiene dos definiciones según el contexto. Puede que esto le resulte molesto (a mí, ciertamente), pero sirve como otro ejemplo más de las realidades de trabajar con lenguaje natural.

La primera definición es la forma sustantiva.

> **Consejo**
>
> ***Refactorización*** *(sustantivo): un cambio realizado en la estructura interna del software para hacerlo más fácil de entender y más barato de modificar sin cambiar su comportamiento observable.*

Puede encontrar ejemplos de refactorizaciones en el catálogo, como Método de extracción y Levantar Campo. Como tal, una refactorización suele ser un pequeño cambio en el software, aunque una refactorización puede implicar otras. Por ejemplo, Extraer clase generalmente implica Método de movimiento y Mover Campo.

El otro uso de *refactorización* es la forma verbal

> **Consejo**
>
> ***Refactorizar*** *(verbo): reestructurar software aplicando una serie de refactorizaciones sin cambiar su comportamiento observable.*

Por lo tanto, podría dedicar algunas horas a la refactorización, durante las cuales podría aplicar un par de docenas de refactorizaciones individuales.

Me preguntaron: "¿La refactorización es solo limpiar el código?" En cierto modo la respuesta es sí, pero creo que la refactorización va más allá porque proporciona una técnica para limpiar el código de una manera más eficiente y controlada. Desde que uso la refactorización, he notado que limpio el código de manera mucho más efectiva que antes. Esto se debe a que sé qué refactorizaciones usar, sé cómo usarlas de manera que se minimicen los errores y pruebo en cada oportunidad posible.



Debería ampliar un par de puntos en mis definiciones. Primero, el propósito de la refactorización

es hacer que el software sea más fácil de entender y modificar. Se pueden realizar muchos cambios en el software que provocan poco o ningún cambio en el comportamiento observable. Sólo los cambios realizados para que el software sea más fácil de entender son refactorizaciones. Un buen contraste es la optimización del rendimiento. Al igual que la refactorización, la optimización del rendimiento no suele cambiar el comportamiento de un componente (aparte de su velocidad); sólo altera la estructura interna. Sin embargo, el propósito es diferente. La optimización del rendimiento a menudo hace que el código sea más difícil de entender, pero es necesario hacerlo para obtener el rendimiento que necesita.

Lo segundo que quiero resaltar es que la refactorización no cambia el comportamiento observable del software. El software sigue realizando la misma función que antes. Cualquier usuario, ya sea final u otro programador, no puede darse cuenta de que las cosas han cambiado.

### Los dos sombreros

Este segundo punto lleva a la metáfora de Kent Beck de los dos sombreros. Cuando utiliza la refactorización para desarrollar software, divide su tiempo entre dos actividades distintas: agregar funciones y refactorizar. Cuando agrega una función, no debería cambiar el código existente; simplemente estás agregando nuevas capacidades. Puede medir su progreso agregando pruebas y haciendo que funcionen. Cuando refactorizas, te aseguras de no agregar funciones; solo reestructuras el código. No agrega ninguna prueba (a menos que encuentre un caso que se le pasó por alto antes); solo reestructuras el código. No agrega ninguna prueba (a menos que encuentre un caso que se le pasó por alto antes); solo cambia las pruebas cuando es absolutamente necesario para hacer frente a un cambio en una interfaz.

A medida que desarrolla software, probablemente se encuentre cambiando de sombrero con frecuencia. Empiezas intentando agregar una nueva función y te das cuenta de que esto sería mucho más fácil si el código estuviera estructurado de manera diferente. Entonces intercambias sombreros y refactorizas por un tiempo. Una vez que el código está mejor estructurado, intercambia sombreros y agrega la nueva función. Una vez que consigues que la nueva función funcione, te das cuenta de que la codificaste de una manera que es difícil de entender, por lo que cambias de sombrero nuevamente y la refactorizas. Todo esto puede que te lleve sólo diez minutos, pero durante este tiempo siempre debes estar atento al sombrero que llevas puesto.

## ¿Por qué debería refactorizar?

No quiero proclamar la refactorización como la cura para todos los males del software. No es una "bala de plata". Sin embargo, es una herramienta valiosa, un par de alicates plateados que le ayudan a controlar bien su código. La refactorización es una herramienta que puede y debe usarse para varios propósitos.

### La refactorización mejora el diseño del software

Sin refactorización, el diseño del programa decaerá. A medida que las personas cambian el código (cambios para lograr objetivos a corto plazo o cambios realizados sin una comprensión completa del diseño del código), el código pierde su estructura. Se vuelve más difícil ver el diseño leyendo el código. Refactorizar es más bien como ordenar el código. Se trabaja para eliminar partes que realmente no están en el lugar correcto. La pérdida de la estructura del código tiene un efecto acumulativo. Cuanto más difícil es ver el diseño en el código, más difícil es preservarlo y más rápidamente se deteriora. La refactorización regular ayuda a que el código conserve su forma.

El código mal diseñado generalmente requiere más código para hacer las mismas cosas, a

menudo porque el código literalmente hace lo mismo en varios lugares. Por tanto, un aspecto importante para mejorar el diseño es eliminar el código duplicado. La importancia de esto radica en futuras modificaciones del código. Reducir la cantidad de código no hará que el sistema funcione más rápido, porque el efecto en la huella de los programas rara vez es significativo. Sin embargo, reducir la cantidad de código



hacer una gran diferencia en la modificación del código. Cuanto más código haya, más difícil será modificarlo correctamente. Hay más código para entender. Cambias este fragmento de código aquí, pero el sistema no hace lo que esperas porque no cambiaste ese fragmento de allí que hace prácticamente lo mismo en un contexto ligeramente diferente. Al eliminar los duplicados, te aseguras de que el código diga todo una vez y sólo una vez, lo cual es la esencia de un buen diseño.

## La refactorización hace que el software sea más fácil de entender

La programación es, en muchos sentidos, una conversación con una computadora. Usted escribe un código que le dice a la computadora qué hacer y ella responde haciendo exactamente lo que usted le dice. Con el tiempo, cierras la brecha entre lo que quieres que haga y lo que le dices que haga. Programar en este modo se trata de decir exactamente lo que quieres. Pero hay otro usuario de tu código fuente. Alguien intentará leer su código dentro de unos meses para realizar algunos cambios. Olvidamos fácilmente a ese usuario adicional del código, pero ese usuario es en realidad el más importante. ¿A quién le importa si la computadora necesita algunos ciclos más para compilar algo? Sí importa si a un programador le toma una semana realizar un cambio que le habría tomado sólo una hora si hubiera entendido su código.

El problema es que cuando intentas que el programa funcione, no estás pensando en el futuro desarrollador. Se necesita un cambio de ritmo para realizar cambios que hagan que el código sea más fácil de entender. La refactorización le ayuda a hacer que su código sea más legible. Al refactorizar, tiene un código que funciona pero que no está estructurado de manera ideal. Dedicar un poco de tiempo a la refactorización puede hacer que el código comunique mejor su propósito. Programar en este modo consiste en decir exactamente lo que quieres decir.

No estoy siendo necesariamente altruista con esto. A menudo, este futuro desarrollador soy yo. Aquí la refactorización es particularmente importante. Soy un programador muy vago. Una de mis formas de pereza es que nunca recuerdo cosas sobre el código que escribo. De hecho, intento deliberadamente no recordar nada que pueda buscar, porque tengo miedo de que mi cerebro se llene. Me propongo intentar poner todo lo que debo recordar en el código para no tener que recordarlo. De esa manera me preocupa menos que el viejo Peculier [Jackson] acabe con mis células cerebrales.

Esta comprensibilidad también funciona de otra manera. Utilizo la refactorización para ayudarme a comprender código desconocido. Cuando miro un código desconocido, tengo que intentar entender qué hace. Miro un par de líneas y me digo a mí mismo, oh sí, eso es lo que hace este fragmento de código. Con la refactorización no me detengo en la nota mental. De hecho, cambio el código para reflejar mejor mi comprensión y luego pruebo esa comprensión volviendo a ejecutar el código para ver si todavía funciona.

Al principio hago refactorizaciones como esta en pequeños detalles. A medida que el código se vuelve más claro, puedo ver cosas sobre el diseño que antes no podía ver. Si no hubiera cambiado el código, probablemente nunca habría visto estas cosas, porque simplemente no soy lo suficientemente inteligente como para visualizar todo esto en mi cabeza. Ralph Johnson describe estas primeras refactorizaciones como limpiar la suciedad de una ventana para poder

ver más allá. Cuando estudio código, encuentro que la refactorización me lleva a niveles más altos de comprensión que de otro modo perdería.

## La refactorización le ayuda a encontrar errores

La ayuda para comprender el código también me ayuda a detectar errores. Admito que no soy muy bueno encontrando errores. Algunas personas pueden leer un montón de código y ver errores, yo no. Sin embargo, encuentro que si refactorizo el código, trabajo profundamente para comprender lo que hace el código y vuelvo a incorporar esa nueva comprensión al código. Al aclarar la estructura del programa, aclaro ciertas suposiciones que he hecho, hasta el punto en que ni siquiera yo puedo evitar detectar los errores.



Me recuerda una afirmación que Kent Beck suele hacer sobre sí mismo: "No soy un gran programador; sólo soy un buen programador con grandes hábitos". La refactorización me ayuda a ser mucho más eficaz a la hora de escribir código sólido.

### La refactorización le ayuda a programar más rápido

Al final, todos los puntos anteriores se reducen a esto: la refactorización le ayuda a desarrollar código más rápidamente.

Esto suena contradictorio. Cuando hablo de refactorización, la gente puede ver fácilmente que mejora la calidad. Mejorar el diseño, mejorar la legibilidad, reducir errores, todo esto mejora la calidad. ¿Pero todo esto no reduce la velocidad del desarrollo?

Creo firmemente que un buen diseño es esencial para un desarrollo rápido de software. De hecho, el objetivo de tener un buen diseño es permitir un desarrollo rápido. Sin un buen diseño, puedes progresar rápidamente durante un tiempo, pero pronto el mal diseño empieza a frenarte. Dedicas tiempo a buscar y corregir errores en lugar de agregar nuevas funciones. Los cambios toman más tiempo a medida que intenta comprender el sistema y encontrar el código duplicado. Las nuevas funciones necesitan más codificación a medida que parcheas un parche que parchea un parche en la base del código original.

Un buen diseño es esencial para mantener la velocidad en el desarrollo de software. La refactorización le ayuda a desarrollar software más rápidamente, porque evita que el diseño del sistema se deteriore. Incluso puede mejorar un diseño.

## ¿Cuándo debería refactorizar?

Cuando hablo de refactorización, a menudo me preguntan cómo se debe programar. ¿Deberíamos dedicar dos semanas cada dos meses a la refactorización?

En casi todos los casos, me opongo a reservar tiempo para la refactorización. En mi opinión, la refactorización no es una actividad para la que se dedica tiempo. La refactorización es algo que se hace todo el tiempo en pequeñas ráfagas.  No decides refactorizar, lo haces porque quieres hacer otra cosa, y la refactorización te ayuda a hacer esa otra cosa.

### La regla de tres

Aquí hay una pauta que me dio Don Roberts: la primera vez que haces algo, simplemente hazlo. La segunda vez que haces algo similar, haces una mueca de dolor ante la duplicación, pero lo haces de todos modos. La tercera vez que haces algo similar, refactorizas.

**Consejo**

*Tres strikes y refactorizas.*

## Refactorizar cuando agrega una función

El momento más común para refactorizar es cuando quiero agregar una nueva característica a algún software. A menudo, la primera razón para refactorizar aquí es ayudarme a comprender algún código que necesito modificar. Es posible que este código haya sido escrito por otra persona o que yo lo haya escrito. Cada vez que tengo que pensar en



Cuando entiendo lo que hace el código, me pregunto si puedo refactorizar el código para que esa comprensión sea más evidente de inmediato. Luego lo refactorizo. Esto es en parte para la próxima vez que pase por aquí, pero principalmente es porque puedo entender más cosas si aclaro el código a medida que avanzo.

El otro factor de refactorización aquí es un diseño que no me ayuda a agregar una característica fácilmente. Miro el diseño y me digo a mí mismo: "Si tan solo hubiera diseñado el código de esta manera, agregar esta característica sería fácil". En este caso no me preocupo por mis fechorías pasadas: las arreglo refactorizando. Hago esto en parte para facilitar futuras mejoras, pero principalmente lo hago porque descubrí que es la forma más rápida. La refactorización es un proceso rápido y fluido. Una vez que haya refactorizado, agregar la función puede ser mucho más rápido y sin problemas.

## Refactorice cuando necesite corregir un error

Para corregir errores, gran parte del uso de la refactorización proviene de hacer que el código sea más comprensible. Mientras miro el código tratando de entenderlo, lo refactorizo para ayudar a mejorar mi comprensión. A menudo encuentro que este proceso activo de trabajar con el código ayuda a encontrar el error. Una forma de verlo es que si recibe un informe de error, es una señal de que necesita refactorizar, porque el código no era lo suficientemente claro como para que usted pudiera ver que había un error.

## Refactorice mientras realiza una revisión de código

Algunas organizaciones realizan revisiones periódicas del código; aquellos que no lo hacen harían mejor si lo hicieran. Las revisiones de código ayudan a difundir el conocimiento a través de un equipo de desarrollo. Las reseñas ayudan a los desarrolladores más experimentados a transmitir conocimientos a personas menos experimentadas. Ayudan a más personas a comprender más aspectos de un gran sistema de software. También son muy importantes a la hora de escribir código claro. Mi código puede parecer claro para mí, pero no para mi equipo. Eso es inevitable: es muy difícil para las personas ponerse en el lugar de alguien que no está familiarizado con las cosas en las que están trabajando. Las reseñas también brindan la oportunidad a más personas de sugerir ideas útiles. Solo se me ocurren un número limitado de

buenas ideas en una semana. Que otras personas contribuyan me hace la vida más fácil, por eso siempre busco muchas reseñas.

Descubrí que la refactorización me ayuda a revisar el código de otra persona. Antes de comenzar a utilizar la refactorización, podía leer el código, comprenderlo en cierta medida y hacer sugerencias. Ahora, cuando se me ocurren ideas, considero si se pueden implementar fácilmente en ese momento con la refactorización. Si es así, lo refactorizo. Cuando lo hago varias veces, puedo ver más claramente cómo se ve el código con las sugerencias implementadas. No tengo que imaginar cómo sería, puedo *ver* cómo es. Como resultado, puedo generar un segundo nivel de ideas que nunca habría realizado si no las hubiera refactorizado.

La refactorización también ayuda a que la revisión del código tenga resultados más concretos. No sólo hay sugerencias, sino que también se implementan muchas sugerencias en el momento. Terminas con una sensación de logro mucho mayor gracias al ejercicio.

Para que este proceso funcione, es necesario tener pequeños grupos de revisión. Mi experiencia sugiere que un revisor y el autor original trabajen juntos en el código. El revisor sugiere cambios y ambos deciden si los cambios se pueden refactorizar fácilmente. Si es así, realizan los cambios.

En el caso de revisiones de diseño más amplias, suele ser mejor obtener varias opiniones en un grupo más grande. Mostrar código a menudo no es el mejor dispositivo para esto. Prefiero los diagramas UML y recorrer escenarios con tarjetas CRC. Por eso hago revisiones de diseño con grupos y revisiones de código con revisores individuales.



Esta idea de revisión activa de código se lleva a su límite con la práctica de Programación en Parejas de Programación Extrema [Beck, XP]. Con esta técnica, todo desarrollo serio se realiza con dos desarrolladores en una máquina. En efecto, es una revisión continua del código incluida en el proceso de desarrollo, y la refactorización que se lleva a cabo también se incluye.

### Por qué funciona la refactorización

*Kent Beck*

Los programas tienen dos tipos de valor: lo que pueden hacer por usted hoy y lo que pueden hacer por usted mañana. La mayoría de las veces, cuando programamos, nos concentramos en lo que queremos que haga el programa hoy. Ya sea que corrijamos un error o agreguemos una característica, estamos haciendo que el programa actual sea más valioso al hacerlo más capaz.

No se puede programar por mucho tiempo sin darse cuenta de que lo que hace el sistema hoy es sólo una parte de la historia. Si puedes hacer el trabajo de hoy hoy, pero lo haces de tal manera que no es posible que puedas hacer el trabajo de mañana mañana, entonces pierdes. Sin embargo, observe que sabe lo que debe hacer hoy, pero no está muy seguro de lo que debe hacer mañana. Tal vez hagas esto, tal vez aquello, tal vez algo que aún no hayas imaginado.

Sé lo suficiente para hacer el trabajo de hoy. No sé lo suficiente para hacer el de mañana. Pero si sólo trabajo hoy, mañana no podré trabajar en absoluto.

La refactorización es una forma de salir del apuro. Cuando descubres que la decisión de ayer no tiene sentido hoy, cambias la decisión. Ahora puedes hacer el trabajo de hoy. Mañana, parte de tu comprensión de hoy te parecerá ingenua, así que cambiarás eso también.

¿Qué es lo que hace que sea difícil trabajar con los programas? Cuatro cosas en las que puedo pensar mientras escribo esto son las siguientes:

- Los programas que son difíciles de leer son difíciles de modificar.
- Los programas que tienen lógica duplicada son difíciles de modificar. • Los programas que requieren un comportamiento adicional que requiere cambiar el código en ejecución son difíciles de modificar.
- Los programas con lógica condicional compleja son difíciles de modificar.

Por lo tanto, queremos programas que sean fáciles de leer, que tengan toda la lógica especificada en un solo lugar, que no permitan cambios que pongan en peligro el comportamiento existente y que permitan que la lógica condicional se exprese de la manera más

simple posible.

La refactorización es el proceso de tomar un programa en ejecución y agregarle valor, no cambiando su comportamiento sino dándole más de estas cualidades que nos permiten continuar desarrollándonos rápidamente.



## ¿Qué le digo a mi jefe?

Cómo informarle a un gerente sobre la refactorización es una de las preguntas más comunes que me hacen. Si el gerente tiene conocimientos técnicos, presentar el tema puede no ser tan difícil. Si el gerente es *verdaderamente* orientado a la calidad, entonces lo que hay que destacar son los aspectos de calidad. Aquí, utilizar la refactorización en el proceso de revisión es una buena forma de trabajar las cosas. Montones de estudios muestran que las revisiones técnicas son una forma importante de reducir errores y así acelerar el desarrollo. Eche un vistazo a cualquier libro sobre revisiones, inspecciones o el proceso de desarrollo de software para conocer las citas más recientes. Estos deberían convencer a la mayoría de los gerentes del valor de las reseñas. Entonces es un breve paso introducir la refactorización como una forma de incluir comentarios de revisión en el código.

Por supuesto, muchas personas dicen que se guían por la calidad, pero más bien por el cronograma. En estos casos doy mi consejo más controvertido: ¡No lo cuentes!

¿Subversivo? No me parece. Los desarrolladores de software son profesionales. Nuestro trabajo es crear software eficaz lo más rápido posible. Mi experiencia es que la refactorización es de gran ayuda para crear software rápidamente. Si necesito agregar una nueva función y el diseño no se adapta al cambio, encuentro que es más rápido refactorizar primero y luego agregar la función. Si necesito corregir un error, necesito entender cómo funciona el software y creo que la refactorización es la forma más rápida de hacerlo. Un gerente basado en horarios quiere que haga las cosas de la manera más rápida posible; cómo lo hago es asunto mío. La forma más rápida es refactorizar; por lo tanto lo refactorizo.

### Indirección y refactorización

*Kent Beck*

*La informática es la disciplina que cree que todos los problemas se pueden resolver con una capa más de dirección indirecta.*
*—Dennis DeBruler*

Dado el enamoramiento de los ingenieros de software con la indirección, puede que no le sorprenda saber que la mayoría de las refactorizaciones introducen más indirección en un programa. La refactorización tiende a dividir objetos grandes en varios más pequeños y métodos grandes en varios más pequeños.

Sin embargo, la dirección indirecta es un arma de doble filo. Cada vez que divides una cosa en dos pedazos, tienes más cosas que gestionar. También puede hacer que un programa sea más difícil de leer cuando un objeto delega a un objeto que delega a un objeto. Entonces le gustaría minimizar la dirección indirecta.

No tan rápido, amigo. La dirección indirecta puede pagarse por sí sola. Estas son algunas de las formas.

- **Para permitir el intercambio de lógica.**

  Por ejemplo, un submétodo invocado en dos lugares diferentes o un método en una superclase compartida por todas las subclases.

- **Explicar la intención y la implementación por separado.**

Elegir el nombre de cada clase y el nombre de cada método le brinda la oportunidad de explicar lo que pretende. Los aspectos internos de la clase o método explican cómo se realiza la intención. Si los aspectos internos también están escritos en términos de intención en textos aún más pequeños
piezas, puede escribir código que comunique la mayor parte de la información importante sobre su propia estructura.

- **Para aislar el cambio.**

  Utilizo un objeto en dos lugares diferentes. Quiero cambiar el comportamiento en uno de los dos casos. Si cambio el objeto, me arriesgo a cambiar ambos. Entonces, primero creo una subclase y me refiero a ella en el caso de que esté cambiando. Ahora puedo modificar la subclase sin arriesgarme a un cambio involuntario en el otro caso.

- **Para codificar lógica condicional.**

  Los objetos tienen un mecanismo fabuloso, mensajes polimórficos, para expresar de manera flexible pero clara la lógica condicional. Al cambiar los condicionales explícitos de los mensajes, a menudo puedes reducir la duplicación, agregar claridad y aumentar la flexibilidad, todo al mismo tiempo.

Aquí está el juego de la refactorización: manteniendo el comportamiento actual del sistema, ¿cómo puede hacer que su sistema sea más valioso, ya sea aumentando su calidad o reduciendo su costo?

La variante más común del juego es mirar tu programa. Identifique un lugar donde falta uno o más de los beneficios de la indirección. Póngalo en esa dirección sin cambiar el comportamiento existente. Ahora tienes un programa más valioso porque tiene más cualidades que apreciaremos mañana.

Compare esto con un cuidadoso diseño inicial. El diseño especulativo es un intento de poner todas las buenas cualidades en el sistema antes de escribir cualquier código. Entonces el código se puede colgar simplemente en el robusto esqueleto. El problema con este proceso es que es muy fácil adivinar mal. Con la refactorización, nunca corres el riesgo de equivocarte por completo. El programa siempre se comporta al final como al

principio. Además, tienes la oportunidad de agregar cualidades valiosas al código.

Hay un segundo juego de refactorización, más raro. Identifique la indirección que no se amortiza por sí sola y elimínela. A menudo esto toma la forma de métodos intermedios que solían cumplir un propósito pero que ya no lo hacen. O podría ser un componente que esperaba que fuera compartido o polimórfico pero que resultó usarse en un solo lugar. Cuando encuentres una dirección indirecta parásita, elimínala. Una vez más, tendrás un programa más valioso, no porque haya



más de una de las cuatro cualidades enumeradas anteriormente, sino porque cuesta menos obtener la misma cantidad de las cualidades.

## Problemas con la refactorización

Cuando aprendes una nueva técnica que mejora enormemente tu productividad, es difícil ver cuándo no se aplica. Por lo general, lo aprendes dentro de un contexto específico, a menudo en un solo proyecto. Es difícil ver qué causa que la técnica sea menos efectiva, e incluso dañina. Hace diez años era así con los objetos. Si alguien me preguntaba cuándo no utilizar objetos, era difícil responder. No es que no pensara que los objetos tuvieran limitaciones; soy demasiado cínico para eso. Era sólo que no sabía cuáles eran esas limitaciones, aunque sí sabía cuáles eran los beneficios.

La refactorización es así ahora. Conocemos los beneficios de la refactorización. Sabemos que pueden marcar una diferencia palpable en nuestro trabajo. Pero no hemos tenido suficiente experiencia para ver dónde se aplican las limitaciones.

Esta sección es más corta de lo que me gustaría y más provisional. A medida que más personas aprendan sobre la refactorización, aprenderemos más. Para usted, esto significa que, si bien creo que debería intentar refactorizar para obtener las ganancias reales que puede proporcionar, también debe monitorear su progreso. Esté atento a los problemas que pueda estar introduciendo la refactorización. Háganos saber acerca de estos problemas. A medida que aprendamos más sobre la refactorización, encontraremos más soluciones a los problemas y aprenderemos qué problemas son difíciles de resolver.

### Bases de datos

Un área problemática para la refactorización son las bases de datos. La mayoría de las aplicaciones empresariales están estrechamente acopladas al esquema de base de datos que las respalda. Ésa es una de las razones por las que es difícil cambiar la base de datos. Otra razón es la migración de datos. Incluso si ha estratificado cuidadosamente su sistema para minimizar las dependencias entre el esquema de la base de datos y el modelo de objetos, cambiar el esquema de la base de datos lo obliga a migrar los datos, lo que puede ser una tarea

larga y complicada.

Con las bases de datos sin objetos, una forma de abordar este problema es colocar una capa separada de software entre su modelo de objetos y su modelo de base de datos. De esa manera puede aislar los cambios en los dos modelos diferentes. Al actualizar un modelo, no es necesario actualizar el otro. Simplemente actualiza la capa intermedia. Esta capa añade complejidad pero proporciona mucha flexibilidad. Incluso sin refactorizar, es muy importante en situaciones en las que tiene varias bases de datos o un modelo de base de datos complejo sobre el que no tiene control.

No es necesario comenzar con una capa separada. Puede crear la capa cuando note que partes de su modelo de objetos se vuelven volátiles. De esta manera obtendrá la mayor ventaja para sus cambios.

Las bases de datos de objetos ayudan y obstaculizan. Algunas bases de datos orientadas a objetos proporcionan migración automática de una versión de un objeto a otra. Esto reduce el esfuerzo pero aún impone una penalización de tiempo mientras se realiza la migración. Cuando la migración no es automática, debe realizarla usted mismo, lo cual supone un gran esfuerzo. En esta situación hay que tener más cuidado con los cambios en la estructura de datos de las clases. Aún puedes mover el comportamiento libremente, pero debes tener más cuidado al mover campos. Es necesario utilizar descriptores de acceso para dar la ilusión de que los datos se han movido, incluso cuando no es así. Cuando esté bastante seguro de saber dónde deben estar los datos, podrá moverlos y migrarlos en un solo movimiento. Sólo es necesario cambiar los descriptores de acceso, lo que reduce el riesgo de problemas con errores.

## Cambiar interfaces



Una de las cosas importantes de los objetos es que le permiten cambiar la implementación de un módulo de software independientemente de cambiar la interfaz. Puedes cambiar de forma segura las partes internas de un objeto sin que nadie más se preocupe por ello, pero la interfaz es importante; cámbiala y cualquier cosa puede suceder.

Algo preocupante acerca de la refactorización es que muchas de las refactorizaciones cambian una interfaz. Algo tan simple como Cambiar nombre del método Se trata de cambiar una interfaz. Entonces, ¿qué efecto tiene esto en la preciada noción de encapsulación?

No hay problema en cambiar el nombre de un método si tiene acceso a todo el código que llama a ese método. Incluso si el método es público, siempre que pueda acceder y cambiar a todas las personas que llaman, puede cambiar el nombre del método. Sólo hay un problema si la interfaz está siendo utilizada por un código que no puede encontrar ni cambiar. Cuando esto sucede, digo que la interfaz se convierte en un *publicado interfaz* (un paso más allá de una interfaz pública). Una vez que publique una interfaz, ya no podrá cambiarla de forma segura y simplemente editar las personas que llaman. Necesitas un proceso algo más complicado.

Esta noción cambia la pregunta. Ahora el problema es: ¿Qué se hace con las refactorizaciones que cambian las interfaces publicadas?

En resumen, si una refactorización cambia una interfaz publicada, debe conservar tanto la interfaz antigua como la nueva, al menos hasta que sus usuarios hayan tenido la oportunidad de reaccionar al cambio. Afortunadamente, esto no es demasiado incómodo. Generalmente puedes arreglar las cosas para que la interfaz anterior siga funcionando. Intente hacer esto para que la

interfaz anterior llame a la nueva. De esta manera, cuando cambie el nombre de un método, conserve el anterior y deje que llame al nuevo. No copie el cuerpo del método, ya que eso le llevará por el camino de la condenación mediante código duplicado. También debe utilizar la función de desuso en Java para marcar el código como obsoleto. De esa manera las personas que llamen sabrán que algo está pasando.

Un buen ejemplo de este proceso son las clases de colección de Java. Los nuevos presentes en Java 2 reemplazan a los que se proporcionaron originalmente. Sin embargo, cuando se lanzaron los de Java 2, JavaSoft puso mucho esfuerzo en proporcionar una ruta de migración.

Por lo general, proteger las interfaces es factible, pero es una molestia. Tienes que construir y mantener estos métodos adicionales, al menos por un tiempo. Los métodos complican la interfaz, haciéndola más difícil de usar. Existe una alternativa: no publicar la interfaz. Ahora bien, no estoy hablando de una prohibición total, claramente hay que tener interfaces publicadas. Si está creando API para consumo externo, como lo hace Sun, entonces debe tener interfaces publicadas. Digo esto porque a menudo veo grupos de desarrollo que utilizan demasiado las interfaces publicadas. He visto un equipo de tres personas operar de tal manera que cada persona publicaba interfaces para las otras dos. Esto llevó a todo tipo de cambios para mantener las interfaces cuando hubiera sido más fácil ingresar al código base y realizar las ediciones. Las organizaciones con una noción demasiado fuerte de propiedad del código tienden a comportarse de esta manera. Usar interfaces publicadas es útil, pero tiene un costo. Así que no publiques interfaces a menos que realmente lo necesites. Esto puede significar modificar las reglas de propiedad de su código para permitir que las personas cambien el código de otras personas para admitir un cambio de interfaz. A menudo es una buena idea hacer esto con programación en pares.

**Consejo**

*No publique interfaces prematuramente. Modifique sus políticas de propiedad de código para facilitar la refactorización.*



Hay un área particular con problemas al cambiar interfaces en Java: agregar una excepción a la cláusula throws. Este no es un cambio de firma, por lo que no puede utilizar la delegación para cubrirlo. Sin embargo, el compilador no le permitirá compilar. Es difícil lidiar con este problema. Puede elegir un nuevo nombre para el método, dejar que el método anterior lo llame y convertir la excepción marcada en una no marcada. También puede generar una excepción no verificada, aunque luego perderá la capacidad de verificación. Al hacer esto, puede alertar a quienes llaman que la excepción se convertirá en una excepción marcada en el futuro. Luego tienen algo de tiempo para incluir los controladores en su código. Por esta razón, prefiero definir una excepción de superclase para un paquete completo (como SQLException para java.sql) y asegurarme de que los métodos públicos solo declaren esta excepción en su cláusula throws. De esa manera puedo definir excepciones de subclase si quiero, pero esto no afectará a una persona que llama y que sólo conoce el caso general.

## Cambios de diseño que son difíciles de refactorizar

¿Puede refactorizar para solucionar cualquier error de diseño, o algunas decisiones de diseño son tan importantes que no puede contar con la refactorización para cambiar de opinión más adelante? Ésta es un área en la que tenemos datos muy incompletos. Ciertamente, a menudo

nos han sorprendido situaciones en las que podemos refactorizar de manera eficiente, pero hay lugares donde esto es difícil. En un proyecto fue difícil, pero posible, refactorizar un sistema construido sin requisitos de seguridad en uno con buena seguridad.

En esta etapa mi enfoque es imaginar la refactorización. Al considerar alternativas de diseño, me pregunto qué tan difícil sería refactorizar un diseño en otro. Si me parece fácil, no me preocupo mucho por la elección, y elijo el diseño más sencillo, aunque no cubra todos los requisitos potenciales. Sin embargo, si no veo una forma sencilla de refactorizar, pongo más esfuerzo en el diseño. Creo que estas situaciones son minoritarias.

### ¿Cuándo no deberías refactorizar?

Hay ocasiones en las que no deberías refactorizar en absoluto. El ejemplo principal es cuando deberías reescribir desde cero. Hay ocasiones en las que el código existente es tan desordenado que, aunque podrías refactorizarlo, sería más fácil empezar desde el principio. Esta decisión no es fácil de tomar y admito que realmente no tengo buenas pautas para ello.

Una señal clara de la necesidad de reescribir es cuando el código actual simplemente no funciona. Puede descubrir esto sólo si intenta probarlo y descubre que el código está tan lleno de errores que no puede estabilizarlo. Recuerde, el código debe funcionar correctamente antes de refactorizar.

Una ruta de compromiso es refactorizar una gran pieza de software en componentes con una fuerte encapsulación. Luego podrá tomar una decisión entre refactorización y reconstrucción para un componente a la vez. Este es un enfoque prometedor, pero no tengo suficientes datos para escribir buenas reglas para hacerlo. Con un sistema heredado clave, esta sería sin duda una dirección atractiva a seguir.

El otro momento en el que debes evitar la refactorización es cuando estás cerca de una fecha límite. En ese momento, el aumento de productividad resultante de la refactorización aparecería después de la fecha límite y, por lo tanto, sería demasiado tarde. Ward Cunningham tiene una buena manera de pensar en esto. Describe la refactorización inacabada como un endeudamiento. La mayoría de las empresas necesitan algo de deuda para funcionar de manera eficiente. Sin embargo, con la deuda vienen los pagos de intereses, es decir, el costo adicional de mantenimiento y extensión causado por un código demasiado complejo. Puede soportar algunos pagos de intereses, pero si los pagos se vuelven demasiado grandes, se sentirá abrumado. Es importante gestionar su deuda, liquidando parte de ella mediante refactorización.

Sin embargo, excepto cuando esté muy cerca de una fecha límite, no debe posponer la refactorización porque no tiene tiempo. La experiencia con varios proyectos ha demostrado que un episodio de



La refactorización da como resultado una mayor productividad. No tener suficiente tiempo suele ser una señal de que es necesario realizar alguna refactorización.

## Refactorización y Diseño

La refactorización tiene un papel especial como complemento al diseño. Cuando aprendí a programar por primera vez, simplemente escribí el programa y lo logré. Con el tiempo aprendí que pensar en el diseño de antemano me ayudó a evitar costosos retrabajos. Con el tiempo me

acerqué más a este estilo de *diseño inicial.* Mucha gente considera que el diseño es la pieza clave y la programación sólo mecánica. La analogía es que el diseño es un dibujo de ingeniería y el código es el trabajo de construcción. Pero el software es muy diferente de las máquinas físicas. Es mucho más maleable y se trata de pensar. Como dice Alistair Cockburn: "Con el diseño puedo pensar muy rápido, pero mi pensamiento está lleno de pequeños agujeros".

Un argumento es que la refactorización puede ser una alternativa al diseño inicial. En este escenario no haces ningún diseño en absoluto. Simplemente codifica el primer enfoque que se le viene a la cabeza, lo hace funcionar y luego lo refactoriza para darle forma. En realidad, este enfoque puede funcionar. He visto a gente hacer esto y crear un software muy bien diseñado. Aquellos que apoyan la programación extrema [Beck, XP] a menudo son retratados como defensores de este enfoque.

Aunque solo refactorizar funciona, no es la forma más eficiente de trabajar. Incluso los programadores extremos hacen algo de diseño primero. Probarán varias ideas con tarjetas CRC o similares hasta que tengan una primera solución plausible. Sólo después de generar un primer intento plausible codificarán y luego refactorizarán. El punto es que la refactorización cambia el papel del diseño inicial. Si no se refactoriza, hay mucha presión para lograr que el diseño inicial sea correcto. La sensación es que cualquier cambio en el diseño posterior será costoso. De este modo, dedica más tiempo y esfuerzo al diseño inicial para evitar la necesidad de realizar dichos cambios.

Con la refactorización el énfasis cambia. Todavía haces diseño inicial, pero ahora no intentas encontrar *el* solución. En cambio, todo lo que desea es una solución razonable. Usted sabe que a medida que construye la solución, a medida que comprende más sobre el problema, se da cuenta de que la mejor solución es diferente de la que se le ocurrió originalmente. Con la refactorización esto no es un problema, porque ya no es costoso realizar cambios.

Un resultado importante de este cambio de énfasis es un mayor movimiento hacia la simplicidad del diseño. Antes de utilizar la refactorización, siempre buscaba soluciones flexibles. Con cualquier requisito, me preguntaría cómo cambiaría ese requisito durante la vida útil del sistema. Como los cambios de diseño eran costosos, buscaría construir un diseño que resistiera los cambios que podía prever. El problema de crear una solución flexible es que la flexibilidad cuesta. Las soluciones flexibles son más complejas que las simples. El software resultante es más difícil de mantener en general, aunque es más fácil de adaptar en la dirección que tenía en mente. Incluso allí, sin embargo, hay que entender cómo flexibilizar el diseño. En uno o dos aspectos esto no es gran cosa, pero se producen cambios en todo el sistema. Crear flexibilidad en todos estos lugares hace que el sistema general sea mucho más complejo y costoso de mantener. La gran frustración, por supuesto, es que toda esta flexibilidad no es necesaria. Algunas partes lo son, pero es imposible predecir qué piezas son. Para ganar flexibilidad, se ve obligado a aplicar mucha más flexibilidad de la que realmente necesita.

Con la refactorización, aborda los riesgos del cambio de manera diferente. Todavía piensas en cambios potenciales, todavía consideras soluciones flexibles. Pero en lugar de implementar estas soluciones flexibles, uno se pregunta: "¿Qué tan difícil será refactorizar una solución simple para convertirla en una solución flexible?" Si, como sucede la mayor parte del tiempo, la respuesta es "bastante fácil", entonces simplemente implemente la solución simple.

La refactorización puede conducir a diseños más simples sin sacrificar la flexibilidad. Esto hace que el proceso de diseño sea más fácil y menos estresante. Una vez que tenga un sentido amplio de las cosas que se pueden refactorizar fácilmente,



Ni siquiera pienses en las soluciones flexibles. Tienes la confianza para refactorizar si llega el

momento.  Construyes lo más simple que pueda funcionar. En cuanto al diseño flexible y complejo, la mayoría de las veces no lo necesitarás.

### Se necesita un tiempo para no crear nada

*Ron Jeffries*

El proceso de pago de Compensación Integral de Chrysler estaba funcionando demasiado lento. Aunque todavía estábamos en desarrollo, nos empezó a molestar, porque ralentizaba las pruebas.

Kent Beck, Martin Fowler y yo decidimos arreglarlo. Mientras esperaba que nos reuniéramos, especulaba, basándose en mi amplio conocimiento del sistema, sobre lo que probablemente lo estaba frenando. Pensé en varias posibilidades y hablé con la gente sobre los cambios que probablemente eran necesarios. Se nos ocurrieron algunas ideas realmente buenas sobre qué haría que el sistema fuera más rápido.

Luego medimos el rendimiento utilizando el perfilador de Kent. Ninguna de las posibilidades que había pensado tenía nada que ver con el problema.  En cambio, descubrimos que el sistema dedicaba la mitad de su tiempo a crear instancias de fecha. Aún más interesante fue que todas las instancias tenían el mismo par de valores.

Cuando analizamos la lógica de creación de fechas, vimos algunas oportunidades para optimizar cómo se crearon estas fechas. Todos estaban pasando por una conversión de cadenas a pesar de que no había entradas externas involucradas. El código solo usaba conversión de cadenas para facilitar la escritura. Quizás podríamos optimizar eso.

Luego vimos cómo se utilizaban estas fechas. Resultó que la gran mayoría de ellos estaban creando instancias de rango de fechas, un objeto con una fecha desde y hasta una fecha. Mirando un poco más a nuestro alrededor, nos dimos cuenta de que la mayoría de estos rangos de fechas estaban vacíos.

Mientras trabajábamos con el intervalo de fechas, utilizamos la convención de que cualquier intervalo de fechas que terminara antes de comenzar estaba vacío. Es una buena convención y encaja bien con el funcionamiento de la clase. Poco después de que comenzamos a usar esta convención, nos dimos cuenta de que simplemente crear un rango de fechas que comienza después de terminar no era un código claro, por lo que extrajimos ese comportamiento en un método de fábrica para rangos de fechas

vacíos.

Hicimos ese cambio para aclarar el código, pero recibimos una recompensa inesperada. Creamos un rango de fechas vacío constante y ajustamos el método de fábrica para devolver ese objeto en lugar de crearlo cada vez. Ese cambio duplicó la velocidad del sistema, lo suficiente para



las pruebas sean soportables. Nos llevó unos cinco minutos.

Había especulado con varios miembros del equipo (Kent y Martin niegan haber participado en la especulación) sobre lo que probablemente estaba mal en el código que conocíamos muy bien. Incluso habíamos esbozado algunos diseños de mejoras sin medir primero lo que estaba pasando.

Estábamos completamente equivocados. Aparte de tener una conversación realmente interesante, no estábamos haciendo ningún bien.

La lección es: incluso si sabes exactamente lo que está pasando en tu sistema, mide el rendimiento, no especules. Aprenderás algo, y nueve de cada diez veces, ¡no será que tenías razón!

## Refactorización y rendimiento

Una preocupación común con la refactorización es el efecto que tiene en el rendimiento de un programa. Para que el software sea más fácil de entender, a menudo se realizan cambios que harán que el programa se ejecute más lentamente. Ésta es una cuestión importante. No pertenezco a la escuela de pensamiento que ignora el rendimiento en favor de la pureza del diseño o con la esperanza de un hardware más rápido. El software ha sido rechazado por ser demasiado lento y las máquinas más rápidas simplemente mueven los postes. La refactorización ciertamente hará que el software funcione más lentamente, pero también hace que el software sea más susceptible de ajuste de rendimiento. El secreto del software rápido, en todos los contextos, excepto en los difíciles en tiempo real, es escribir primero software ajustable y luego ajustarlo para lograr una velocidad suficiente.

He visto tres enfoques generales para escribir software rápido. El más grave de ellos es el presupuesto de tiempo, que se utiliza a menudo en sistemas de tiempo real. En esta situación, a medida que descompone el diseño, le da a cada componente un presupuesto de recursos: tiempo y espacio. Ese componente no debe exceder su presupuesto, aunque se permite un mecanismo de intercambio de tiempos presupuestados. Un mecanismo de este tipo centra la atención en los tiempos difíciles de rendimiento. Es esencial para sistemas como los marcapasos, en los que los datos tardíos siempre son malos. Esta técnica es excesiva para otro tipo de sistemas, como los sistemas de información corporativos con los que trabajo habitualmente.

El segundo enfoque es el de atención constante. Con este enfoque, cada programador, todo el tiempo, hace todo lo que puede para mantener un alto rendimiento. Este es un enfoque común y tiene una atracción intuitiva, pero no funciona muy bien. Los cambios que mejoran el rendimiento normalmente hacen que sea más difícil trabajar con el programa. Esto ralentiza el desarrollo. Este sería un coste que valdría la pena pagar si el software resultante fuera más rápido, pero normalmente no lo es. Las mejoras de rendimiento se distribuyen por todo el programa y cada mejora se realiza con una perspectiva estrecha del comportamiento del programa.

Lo interesante del rendimiento es que si analizas la mayoría de los programas, descubres que pierden la mayor parte de su tiempo en una pequeña fracción del código. Si optimiza todo el código por igual, terminará con el 90 por ciento de las optimizaciones desperdiciadas, porque está optimizando código que no es
correr mucho. El tiempo dedicado a acelerar el programa, el tiempo perdido por falta de claridad, es tiempo perdido.

El tercer enfoque para mejorar el desempeño aprovecha esta estadística del 90 por ciento. En este enfoque, usted construye su programa de una manera bien factorizada sin prestar atención al rendimiento hasta que comienza una etapa de optimización del rendimiento, generalmente bastante avanzada en el desarrollo. Durante la etapa de optimización del rendimiento, se sigue un proceso específico para ajustar el programa.



Comienza ejecutando el programa bajo un generador de perfiles que monitorea el programa y le indica dónde está consumiendo tiempo y espacio. De esta manera podrás encontrar esa pequeña parte del programa donde se encuentran los puntos calientes de rendimiento. Luego, se concentra en esos puntos críticos de rendimiento y utiliza las mismas optimizaciones que usaría si estuviera utilizando el enfoque de atención constante. Pero debido a que estás centrando tu atención en un punto caliente, estás obteniendo mucho más efecto con menos trabajo. Aun así, sigue siendo cauteloso. Al igual que en la refactorización, los cambios se realizan en pequeños pasos. Después de cada paso, compila, prueba y vuelve a ejecutar el generador de perfiles. Si no ha mejorado el rendimiento, retira el cambio. Continúa el proceso de encontrar y eliminar puntos calientes hasta que obtenga el rendimiento que satisfaga a sus usuarios. McConnel [McConnel] brinda más información sobre esta técnica.

Tener un programa bien factorizado ayuda con este estilo de optimización de dos maneras. En primer lugar, le da tiempo para dedicarlo a ajustar el rendimiento. Como tiene un código bien factorizado, puede agregar funciones más rápidamente. Esto le da más tiempo para concentrarse en el rendimiento. (La creación de perfiles garantiza que usted concentre ese tiempo en el lugar correcto). En segundo lugar, con un programa bien factorizado tendrá una granularidad más fina para su análisis de rendimiento. Su generador de perfiles lo lleva a partes más pequeñas del código, que son más fáciles de ajustar. Debido a que el código es más claro, usted comprende mejor sus opciones y qué tipo de ajuste funcionará.

Descubrí que la refactorización me ayuda a escribir software rápido. Ralentiza el software a corto plazo mientras lo refactorizo, pero hace que el software sea más fácil de ajustar durante la optimización. Termino muy por delante.

## ¿De dónde vino la refactorización?

No he logrado precisar el nacimiento real del término. *refactorización.* Los buenos programadores ciertamente han dedicado al menos algún tiempo a limpiar su código. Lo hacen

porque han aprendido que el código limpio es más fácil de cambiar que el código complejo y desordenado, y los buenos programadores saben que rara vez escriben código limpio la primera vez.

La refactorización va más allá de esto. En este libro defiendo la refactorización como un elemento clave en todo el proceso de desarrollo de software. Dos de las primeras personas en reconocer la importancia de la refactorización fueron Ward Cunningham y Kent Beck, quienes trabajaron con Smalltalk desde la década de 1980 en adelante. Smalltalk es un entorno que incluso entonces era particularmente hospitalario para la refactorización. Es un entorno muy dinámico que le permite escribir rápidamente software altamente funcional. Smalltalk tiene un ciclo de compilación-enlace-ejecución muy corto, lo que facilita cambiar las cosas rápidamente. También está orientado a objetos y, por lo tanto, proporciona herramientas poderosas para minimizar el impacto del cambio detrás de interfaces bien definidas. Ward y Kent trabajaron duro para desarrollar un proceso de desarrollo de software orientado a trabajar con este tipo de entorno. (Kent actualmente se refiere a este estilo como *Programación extrema* [Beck, XP].) Se dieron cuenta de que la refactorización era importante para mejorar su productividad y desde entonces han estado trabajando con la refactorización, aplicándola a proyectos de software serios y refinando el proceso.

Las ideas de Ward y Kent siempre han tenido una fuerte influencia en la comunidad de Smalltalk y la noción de refactorización se ha convertido en un elemento importante en la cultura de Smalltalk. Otra figura destacada de la comunidad Smalltalk es Ralph Johnson, profesor de la Universidad de Illinois en Urbana-Champaign, famoso por ser miembro de la Banda de los Cuatro [Gang of Four]. Uno de los mayores intereses de Ralph es el desarrollo de marcos de software. Exploró cómo la refactorización puede ayudar a desarrollar un marco eficiente y flexible.

Bill Opdyke fue uno de los estudiantes de doctorado de Ralph y está particularmente interesado en los marcos. Vio el valor potencial de la refactorización y vio que podría aplicarse a mucho más que Smalltalk. Su experiencia se centró en el desarrollo de conmutadores telefónicos, en el que con el tiempo se acumula una gran complejidad y es difícil realizar cambios. La investigación doctoral de Bill analizó



refactorización desde la perspectiva de un creador de herramientas. Bill investigó las refactorizaciones que serían útiles para el desarrollo del marco C++ e investigó las refactorizaciones necesarias para preservar la semántica, cómo demostrar que preservaban la semántica y cómo una herramienta podría implementar estas ideas. La tesis doctoral de Bill [Opdyke] es el trabajo más sustancial sobre refactorización hasta la fecha. Él también contribuye Capítulo 13 a este libro.

Recuerdo haber conocido a Bill en la conferencia OOPSLA en 1992. Nos sentamos en un café y discutimos algunos de los trabajos que había realizado para construir un marco conceptual para la atención médica. Bill me habló de su investigación y recuerdo que pensé: "Interesante, pero en realidad no tan importante". ¡Me equivoqué!

John Brant y Don Roberts han llevado las ideas de herramientas de refactorización mucho más allá para producir Refactoring Browser, una herramienta de refactorización para Smalltalk. Ellos contribuyen Capítulo 14 a este libro, que describe con más detalle las herramientas de refactorización.

¿Y yo? Siempre me había inclinado por el código limpio, pero nunca lo había considerado *eso* importante. Luego trabajé en un proyecto con Kent y vi la forma en que utilizaba la

refactorización. Vi la diferencia que hizo en productividad y calidad. Esa experiencia me convenció de que la refactorización era una técnica muy importante. Sin embargo, me sentí frustrado porque no había ningún libro que pudiera darle a un programador en activo y ninguno de los expertos mencionados anteriormente tenía planes de escribir un libro así. Entonces, con su ayuda, lo hice.

### Optimización de un sistema de nómina

*Rico Garzaniti*

Habíamos estado desarrollando el Sistema de Compensación Integral de Chrysler durante bastante tiempo antes de comenzar a trasladarlo a GemStone. Naturalmente, cuando hicimos eso, descubrimos que el programa no era lo suficientemente rápido. Contratamos a Jim Haungs, un maestro en gemas, para que nos ayudara a optimizar el sistema.

Después de un tiempo con el equipo para aprender cómo funcionaba el sistema, Jim utilizó la función ProfMonitor de GemStone para escribir una herramienta de creación de perfiles que se conectó a nuestras pruebas funcionales. La herramienta mostraba la cantidad de objetos que se estaban creando y dónde se estaban creando.

Para nuestra sorpresa, el mayor problema resultó ser la creación de hilos. El mayor de los grandes fue la creación repetida de cadenas de 12.000 bytes. Este fue un problema particular porque la cuerda era tan grande que las instalaciones habituales de recolección de basura de GemStone no podían manejarla. Debido al tamaño, GemStone paginaba la cadena en el disco cada vez que se creaba. Resultó que las cadenas se estaban construyendo en nuestro marco IO, ¡y se estaban construyendo de tres en tres para cada registro de salida!

Nuestra primera solución fue almacenar en caché una única cadena de 12.000 bytes, lo que resolvió la mayor parte del problema. Más tarde, cambiamos el marco para escribir directamente en un archivo.

stream, lo que eliminó la creación incluso de una sola cadena.

Una vez que la enorme cadena estuvo fuera del camino, el generador de perfiles de Jim encontró problemas similares con algunas cadenas más pequeñas: 800 bytes, 500 bytes, etc. Convertirlos para usar la función de flujo de archivos también los resolvió.

Con estas técnicas mejoramos constantemente el rendimiento del sistema. Durante el desarrollo parecía que se necesitarían más de 1000 horas para ejecutar la nómina. Cuando realmente nos preparamos para empezar, nos llevó 40 horas. Después de un mes lo redujimos a alrededor de 18; cuando lo lanzamos teníamos 12 horas. Después de un año de ejecutar y mejorar el sistema para un nuevo grupo de empleados, se redujo a 9 horas.

Nuestra mayor mejora fue ejecutar el programa en múltiples subprocesos en una máquina multiprocesador. El sistema no fue diseñado teniendo en cuenta los subprocesos, pero debido a que estaba tan bien factorizado, solo nos tomó tres días ejecutarlo en múltiples subprocesos. Ahora la nómina tarda un par de horas en ejecutarse.

Antes de que Jim proporcionara una herramienta que mediera el sistema en funcionamiento real, teníamos buenas ideas sobre lo que estaba mal. Pero pasó mucho tiempo antes de que nuestras buenas ideas fueran las que debían implementarse. Las mediciones reales apuntaron en una dirección diferente y marcaron una diferencia mucho mayor.



# Capítulo 3. Malos olores en el código

*por Kent Beck y Martin Fowler*

> *Si huele mal, cámbialo.*
>
> —Abuela Beck, hablando sobre la filosofía de la crianza de los niños.

A estas alturas ya tienes una buena idea de cómo funciona la refactorización. Pero sólo porque sepas cómo no significa que sepas cuándo. Decidir cuándo comenzar la refactorización y cuándo detenerla es tan importante para la refactorización como saber cómo operar la mecánica de una refactorización.

Ahora viene el dilema. Es fácil explicarle cómo eliminar una variable de instancia o crear una jerarquía. Éstas son cuestiones sencillas. Tratar de explicar cuándo debes hacer estas cosas no es tan sencillo. En lugar de apelar a una vaga noción de estética de la programación (que

francamente es lo que solemos hacer los consultores), quería algo un poco más sólido.

Estaba reflexionando sobre este delicado tema cuando visité a Kent Beck en Zurich. Quizás estaba bajo la influencia de los olores de su hija recién nacida en ese momento, pero se le ocurrió la idea de describir el "cuándo" de la refactorización en términos de olores. "Huele", dices, "¿y eso se supone que es mejor que una vaga estética?" Bueno, sí. Analizamos una gran cantidad de código, escrito para proyectos que abarcan desde un gran éxito hasta un casi muerto. Al hacerlo, hemos aprendido a buscar ciertas estructuras en el código que sugieren (a veces piden a gritos) la posibilidad de refactorizar. (Cambiamos a "nosotros" en este capítulo para reflejar el hecho de que Kent y yo escribimos este capítulo juntos. Puedes notar la diferencia porque los chistes divertidos son míos y los demás son suyos).

Una cosa que no intentaremos hacer aquí es brindarle criterios precisos sobre cuándo debe realizarse una refactorización. Según nuestra experiencia, ningún conjunto de métricas rivales informaba la intuición humana. Lo que haremos será darle indicaciones de que hay un problema que puede solucionarse mediante una refactorización. Tendrá que desarrollar su propio sentido de cuántas variables de instancia son demasiadas variables de instancia y cuántas líneas de código en un método son demasiadas líneas.

Debería utilizar este capítulo y la tabla en el interior de la contraportada como una manera de inspirarse cuando no esté seguro de qué refactorizaciones hacer. Lea el capítulo (o hojee la tabla) para intentar identificar qué es lo que está oliendo, luego vaya a las refactorizaciones que sugerimos para ver si le ayudarán. Es posible que no encuentre el olor exacto que pueda detectar, pero con suerte le indicará la dirección correcta.

## Código duplicado

El número uno en el desfile apestoso es el código duplicado. Si ve la misma estructura de código en más de un lugar, puede estar seguro de que su programa será mejor si encuentra una manera de unificarlos.

El problema de código duplicado más simple es cuando tienes la misma expresión en dos métodos de la misma clase. Entonces todo lo que tienes que hacer es Método de extracción e invocar el código desde ambos lugares.

Otro problema de duplicación común es cuando tienes la misma expresión en dos subclases hermanas. Puedes eliminar esta duplicación usando Método de extracción entonces en ambas clases Campo de dominadas. Si el código es similar pero no igual, debes usar Método de extracción para separar los bits similares de los bits diferentes. Entonces descubrirás que puedes usar Plantilla de formulario



Método. Si los métodos hacen lo mismo con un algoritmo diferente, puede elegir el más claro de los dos algoritmos y utilizar Algoritmo de sustitución.

Si tiene código duplicado en dos clases no relacionadas, considere usar Extraer clase en una clase y luego usar el nuevo componente en la otra. Otra posibilidad es que el método realmente pertenezca sólo a una de las clases y deba ser invocado por la otra clase o que el método pertenezca a una tercera clase a la que ambas clases originales deberían hacer referencia. Tienes que decidir dónde tiene sentido el método y asegurarte de que esté allí y en ningún otro lugar.

## Método largo

Los programas objeto que viven mejor y más tiempo son aquellos con métodos cortos. Los programadores nuevos en objetos a menudo sienten que nunca se realiza ningún cálculo, que los programas de objetos son secuencias interminables de delegación. Sin embargo, cuando uno ha vivido con un programa de este tipo durante algunos años, se da cuenta de lo valiosos que son todos esos pequeños métodos. Todos los beneficios de la indirección (explicación, compartir y elegir) están respaldados por pequeños métodos (consulte Indirección y refactorización en la página 61).

Desde los primeros días de la programación, la gente se ha dado cuenta de que cuanto más largo es un procedimiento, más difícil es de entender. Los lenguajes más antiguos conllevaban una sobrecarga en las llamadas a subrutinas, lo que disuadía a la gente de utilizar métodos pequeños. Los lenguajes OO modernos prácticamente han eliminado esa sobrecarga para las llamadas en proceso. Todavía hay una sobrecarga para el lector del código porque tiene que cambiar de contexto para ver qué hace el subprocedimiento. Los entornos de desarrollo que le permiten ver dos métodos a la vez ayudan a eliminar este paso, pero la verdadera clave para facilitar la comprensión de los métodos pequeños es una buena denominación. Si tienes un buen nombre para un método, no necesitas mirar el cuerpo.

El efecto neto es que debería ser mucho más agresivo con los métodos de descomposición. Una heurística que seguimos es que cada vez que sentimos la necesidad de comentar algo, escribimos un método. Dicho método contiene el código que se comentó, pero recibe el nombre de la intención del código y no de cómo lo hace. Podemos hacer esto en un grupo de líneas o en tan solo una línea de código. Hacemos esto incluso si la llamada al método es más larga que el código que reemplaza, siempre que el nombre del método explique el propósito del código. La clave aquí no es la longitud del método sino la distancia semántica entre lo que hace el método y cómo lo hace.

El noventa y nueve por ciento de las veces, todo lo que tienes que hacer para acortar un método es Método de extracción. Encuentre partes del método que parezcan ir bien juntas y cree un método nuevo.

Si tiene un método con muchos parámetros y variables temporales, estos elementos obstaculizan la extracción de métodos. Si intentas usar *método de extracción,* terminas pasando tantos parámetros y variables temporales como parámetros al método extraído que el resultado apenas es más legible que el original. A menudo puedes usar Reemplazar temperatura con consulta para eliminar las temperaturas. Se pueden reducir largas listas de parámetros con Introducir parámetro Objeto y Preservar el objeto completo.

Si lo has intentado y todavía tienes demasiadas temperaturas y parámetros, es hora de sacar la artillería pesada: Reemplazar método con objeto de método.

¿Cómo se identifican los grupos de código que se van a extraer? Una buena técnica es buscar comentarios. A menudo señalan este tipo de distancia semántica. Un bloque de código con un comentario que indica lo que está haciendo se puede reemplazar por un método cuyo nombre se base en el comentario. Incluso vale la pena extraer una sola línea si necesita explicación.



Los condicionales y los bucles también dan señales de extracciones. Usar Descomponer

condicional para tratar con expresiones condicionales. Con bucles, extraiga el bucle y el código dentro del bucle en su propio método.

## Clase grande

Cuando una clase intenta hacer demasiado, a menudo aparece como demasiadas variables de instancia. Cuando una clase tiene demasiadas variables de instancia, el código duplicado no se queda atrás.

Puede Extraer clase para agrupar varias de las variables. Elija variables para combinarlas en el componente que tenga sentido para cada una. Por ejemplo, es probable que "depositAmount" y "depositCurrency" pertenezcan juntos en un componente. De manera más general, los prefijos o sufijos comunes para algún subconjunto de variables en una clase sugieren la oportunidad de un componente. Si el componente tiene sentido como subclase, encontrará Extraer subclase muchas veces es más fácil.

A veces una clase no utiliza todas sus variables de instancia todo el tiempo. Si es así, es posible que pueda Extraer clase o Extraer subclase muchas veces.

Al igual que con una clase con demasiadas variables de instancia, una clase con demasiado código es un caldo de cultivo ideal para el código duplicado, el caos y la muerte. La solución más simple (¿hemos mencionado que nos gustan las soluciones simples?) es eliminar la redundancia en la clase misma. Si tienes quinientas líneas
métodos con mucho código en común, es posible que pueda convertirlos en cinco métodos de diez líneas con otros diez métodos de dos líneas extraídos del original.

Al igual que con una clase con una gran cantidad de variables, la solución habitual para una clase con demasiado código es Extraer clase o Extraer subclase. Un truco útil es determinar cómo los clientes usan la clase y usar Extraer interfaz para cada uno de estos usos. Eso puede darle ideas sobre cómo dividir aún más la clase.

Si su clase grande es una clase GUI, es posible que necesite mover datos y comportamiento a un objeto de dominio separado. Esto puede requerir mantener algunos datos duplicados en ambos lugares y mantenerlos sincronizados. Datos observados duplicados sugiere cómo hacer esto. En este caso, especialmente si está utilizando componentes antiguos de Abstract Windows Toolkit (AWT), puede seguir esto eliminando la clase GUI y reemplazándola con componentes Swing.

## Lista larga de parámetros

En nuestros primeros días de programación nos enseñaron a pasar como parámetros todo lo que necesitaba una rutina. Esto era comprensible porque la alternativa eran los datos globales, y los datos globales son malos y normalmente dolorosos. Los objetos cambian esta situación porque si no tienes algo que necesitas, siempre puedes pedirle a otro objeto que te lo consiga. Por lo tanto, con los objetos no se pasa todo lo que el método necesita; en lugar de eso, pasa lo suficiente para que el método pueda llegar a todo lo que necesita. Gran parte de lo que necesita un método está disponible en la clase anfitriona del método. En los programas orientados a objetos las listas de parámetros tienden a ser mucho más pequeñas que en los programas tradicionales.

Esto es bueno porque las listas de parámetros largas son difíciles de entender, porque se vuelven inconsistentes y difíciles de usar, y porque siempre las cambias a medida que necesitas más datos. La mayoría de los cambios se eliminan pasando objetos porque es mucho más probable que necesite realizar solo un par de solicitudes para obtener un nuevo dato.

Usar Reemplazar parámetro con método cuando puede obtener los datos en un parámetro realizando una solicitud de un objeto que ya conoce. Este objeto puede ser un campo o puede ser otro



parámetro. Usar Preservar el objeto completo tomar un montón de datos obtenidos de un objeto y reemplazarlos con el objeto mismo. Si tiene varios elementos de datos sin ningún objeto lógico, utilice Introducir objeto de parámetro.

Hay una excepción importante a la hora de realizar estos cambios. Esto es cuando explícitamente no desea crear una dependencia del objeto llamado al objeto más grande. En esos casos, descomprimir los datos y enviarlos como parámetros es razonable, pero preste atención al dolor que implica. Si la lista de parámetros es demasiado larga o cambia con demasiada frecuencia, debe reconsiderar su estructura de dependencia.

## Cambio divergente

Estructuramos nuestro software para facilitar el cambio; después de todo, el software debe ser blando. Cuando hacemos un cambio queremos poder saltar a un único punto claro en el sistema y realizar el cambio. Cuando no puedes hacer esto, estás oliendo uno de dos puntos picantes estrechamente relacionados.

El cambio divergente ocurre cuando una clase comúnmente cambia de diferentes maneras por diferentes razones. Si observa una clase y dice: "Bueno, tendré que cambiar estos tres métodos cada vez que obtenga una nueva base de datos; tengo que cambiar estos cuatro métodos cada vez que haya un nuevo instrumento financiero", probablemente se encuentre en una situación en la que dos objetos son mejores que uno. De esta manera, cada objeto cambia sólo como resultado de un tipo de cambio. Por supuesto, a menudo descubres esto sólo después de haber agregado algunas bases de datos o instrumentos financieros. Cualquier cambio para manejar una variación debería cambiar una sola clase, y todo lo escrito en la nueva clase debería expresar la variación. Para limpiar esto, identifica todo lo que cambia por una causa particular y usa Extraer clase para juntarlos a todos.

## Cirugía de escopeta

La cirugía de escopeta es similar al cambio divergente pero es todo lo contrario. Hueles esto cuando cada vez que haces algún tipo de cambio, tienes que hacer muchos pequeños cambios en muchas clases diferentes. Cuando los cambios están por todos lados, son difíciles de encontrar y es fácil pasar por alto un cambio importante.

En este caso desea utilizar Método de movimiento y Mover campo para poner todos los cambios en una sola clase. Si ninguna clase actual parece buena candidata, cree una. A menudo puedes usar Clase en línea para reunir un montón de comportamientos. Obtienes una pequeña dosis de cambio divergente, pero puedes afrontarlo fácilmente.

El cambio divergente es una clase que sufre muchos tipos de cambios, y la cirugía de escopeta es un cambio que altera muchas clases. De cualquier manera, desea organizar las cosas de modo que, idealmente, exista un vínculo uno a uno entre los cambios y las clases comunes.

## Envidia de funciones

El objetivo de los objetos es que son una técnica para empaquetar datos con los procesos utilizados en esos datos. Un olor clásico es un método que parece más interesado en una clase distinta de aquella en la que realmente se encuentra. El foco más común de la envidia son los datos. Hemos perdido la cuenta de las veces que hemos visto un método que invoca media docena de métodos de obtención en otro objeto para calcular algún valor. Afortunadamente la cura es obvia, el método claramente quiere estar en otra parte, así que usas Método de movimiento para llegar allí. A veces sólo una parte del método sufre de envidia; en ese caso usar Método de extracción en la parte celosa y Método de movimiento para darle la casa de sus sueños.



Por supuesto, no todos los casos son sencillos. A menudo, un método utiliza características de varias clases, entonces, ¿con cuál debería convivir? La heurística que utilizamos es determinar qué clase tiene la mayoría de los datos y poner el método con esos datos. Este paso suele ser más fácil si Método de extracción se utiliza para dividir el método en pedazos que van a diferentes lugares.

Por supuesto, existen varios patrones sofisticados que rompen esta regla. De la Banda de los Cuatro [Banda de los Cuatro] Estrategia y Visitante inmediatamente me vienen a la mente. La Autodelegación [Beck] de Kent Beck es otra. Los utilizas para combatir el olor a cambio divergente. La regla general fundamental es juntar cosas que cambien juntas. Los datos y el comportamiento que hace referencia a esos datos suelen cambiar juntos, pero hay excepciones. Cuando ocurren excepciones, movemos el comportamiento para mantener los cambios en un solo lugar. La estrategia y el visitante le permiten cambiar el comportamiento fácilmente, porque aíslan la pequeña cantidad de comportamiento que debe anularse, a costa de una mayor indirección.

## Grupos de datos

Los elementos de datos tienden a ser como niños; les gusta estar juntos en grupos. A menudo verás los mismos tres o cuatro elementos de datos juntos en muchos lugares: campos en un par de clases, parámetros en muchas firmas de métodos. Los grupos de datos que se encuentran juntos realmente deberían convertirse en su propio objeto. El primer paso es buscar dónde aparecen los grupos como campos. Usar Extraer clase en los campos para convertir los grupos en un objeto. Luego, preste atención a las firmas de métodos usando Introducir objeto de parámetro o Preservar el objeto completo para adelgazarlos. El beneficio inmediato es que puede reducir muchas listas de parámetros y simplificar la llamada a métodos. No se preocupe por los grupos de datos que utilizan sólo algunos de los campos del nuevo objeto. Siempre que reemplace dos o más campos con el nuevo objeto, saldrá adelante.

Una buena prueba es considerar eliminar uno de los valores de datos: si hiciera esto, ¿tendrían algún sentido los demás? Si no es así, es una señal segura de que tienes un objeto que se muere por nacer.

Reducir las listas de campos y de parámetros ciertamente eliminará algunos malos olores, pero una vez que tengas los objetos, tendrás la oportunidad de hacer un buen perfume. Ahora puede buscar casos de envidia de funciones, que le sugerirán comportamientos que pueden trasladarse a sus nuevas clases. En poco tiempo estas clases serán miembros productivos de la sociedad.

## Obsesión primitiva

La mayoría de los entornos de programación tienen dos tipos de datos. Los tipos de registros le permiten estructurar los datos en grupos significativos. Los tipos primitivos son tus componentes básicos. Los registros siempre conllevan una cierta cantidad de gastos generales. Pueden referirse a tablas en una base de datos, o pueden resultar difíciles de crear cuando solo las desea para una o dos cosas.

Una de las cosas valiosas de los objetos es que desdibujan o incluso rompen la línea entre clases primitivas y grandes. Puedes escribir fácilmente pequeñas clases que no se pueden distinguir de los tipos integrados del idioma. Java tiene primitivas para números, pero las cadenas y las fechas, que son primitivas en muchos otros entornos, son clases.

Las personas nuevas en el mundo de los objetos suelen ser reacias a utilizar objetos pequeños para tareas pequeñas, como clases de dinero que combinan número y moneda, rangos con una parte superior y una inferior, y cadenas especiales como números de teléfono y códigos postales. Puedes salir de la cueva al mundo de objetos con calefacción central usando Reemplazar valor de datos con objeto en valores de datos individuales. Si el valor de los datos es un código de tipo, utilice Reemplazar código de tipo con clase si el valor no afecta el comportamiento. Si tiene condicionales que dependen del código de tipo, utilice Reemplazar tipo Código con subclases o Reemplazar código de tipo con estado/estrategia.



Si tiene un grupo de campos que deben ir juntos, use Extraer clase. Si ve estas primitivas en las listas de parámetros, pruebe con una dosis civilizadora de Introducir objeto de parámetro. Si te encuentras separando una matriz, usa Reemplazar matriz con objeto.

## Declaraciones de cambio

Uno de los síntomas más obvios del código orientado a objetos es su relativa falta de declaraciones de cambio (o caso). El problema con las declaraciones de cambio es esencialmente el de la duplicación. A menudo encontrará la misma declaración de cambio dispersa en un programa en diferentes lugares. Si agrega una nueva cláusula al cambio, debe encontrar todos estos cambios, declaraciones y cambiarlas. La noción de polimorfismo orientada a objetos ofrece una manera elegante de abordar este problema.

La mayoría de las veces que ve una declaración de cambio, debería considerar el polimorfismo. La cuestión es dónde debería producirse el polimorfismo. A menudo, la declaración de cambio activa un código de tipo. Quiere el método o clase que aloja el valor del código de tipo. Entonces usa Método de extracción para extraer la declaración de cambio y luego Método de movimiento para llevarlo a la clase donde se necesita el polimorfismo. En ese momento tienes que decidir si Reemplazar código de tipo con subclases o Reemplazar Escriba código con estado/estrategia. Cuando haya configurado la estructura de herencia, puede utilizar Reemplazar condicional con polimorfismo.

Si solo tiene unos pocos casos que afectan a un único método y no espera que cambien, entonces el polimorfismo es excesivo. En este caso Reemplazar parámetro con métodos explícitos es una buena opción. Si uno de sus casos condicionales es nulo, intente Introducir objeto nulo.

## Jerarquías de herencia paralela

Las jerarquías de herencia paralelas son en realidad un caso especial de cirugía de escopeta.

En este caso, cada vez que creas una subclase de una clase, también tienes que crear una subclase de otra. Puede reconocer este olor porque los prefijos de los nombres de clases en una jerarquía son los mismos que los prefijos en otra jerarquía.

La estrategia general para eliminar la duplicación es asegurarse de que las instancias de una jerarquía se refieran a instancias de la otra. si usas Método de movimiento y Mover campo, la jerarquía de la clase de referencia desaparece.

## Clase perezosa

Cada clase que creas cuesta dinero para mantenerla y comprenderla. Una clase que no está haciendo lo suficiente para pagarse a sí misma debería ser eliminada. A menudo, esta podría ser una clase que solía pagarse pero que se ha reducido con la refactorización. O podría ser una clase que se agregó debido a cambios planificados pero no realizados. De cualquier manera, dejas que la clase muera con dignidad. Si tienes subclases
que no están haciendo lo suficiente, intenta usar Contraer jerarquía. Los componentes casi inútiles deben someterse a Clase en línea.

## Generalidad especulativa

Brian Foote sugirió este nombre para un olor al que somos muy sensibles. Lo entiendes cuando la gente dice: "Oh, creo que algún día necesitamos la capacidad para este tipo de cosas" y, por lo tanto, quiere todo tipo de ganchos y casos especiales para manejar cosas que no son necesarias. El resultado suele ser más difícil de entender y mantener. Si se utilizara toda esta maquinaria, valdría la pena. Pero si no lo es, no lo es. La maquinaria simplemente estorba, así que deshazte de ella.



Si tienes clases abstractas que no hacen mucho, usa Contraer jerarquía. La delegación innecesaria se puede eliminar con Clase en línea. Los métodos con parámetros no utilizados deben estar sujetos a Eliminar parámetro. Los métodos nombrados con nombres abstractos impares deben bajar a la tierra con Cambiar nombre del método.

La generalidad especulativa se puede detectar cuando los únicos usuarios de un método o clase son casos de prueba. Si encuentra dicho método o clase, elimínelo y el caso de prueba que lo ejercita. Si tiene un método o clase que sirve de ayuda para un caso de prueba que ejerce una funcionalidad legítima, debe dejarlo ahí, por supuesto.

## Campo Temporal

A veces ves un objeto en el que se establece una variable de instancia sólo en determinadas circunstancias. Este código es difícil de entender porque se espera que un objeto necesite todas sus variables. Tratar de entender por qué una variable está ahí cuando parece no usarse puede volverte loco.

Usar Extraer clase para crear un hogar para los pobres huérfanos variables. Coloque todo el código relacionado con las variables en el componente. También puede eliminar el código condicional utilizando Introducir objeto nulo para crear un componente alternativo para cuando las variables no sean válidas.

Un caso común de campo temporal ocurre cuando un algoritmo complicado necesita varias

variables. Como el implementador no quería pasar una lista enorme de parámetros (¿quién quiere?), los puso en campos. Pero los campos son válidos sólo durante el algoritmo; en otros contextos son simplemente confusos. En este caso puedes usar *Extraer clase* con estas variables y los métodos que las requieren. El nuevo objeto es un objeto de método [Beck].

## Cadenas de mensajes

Se ven cadenas de mensajes cuando un cliente solicita a un objeto otro objeto, luego el cliente solicita otro objeto más, luego el cliente solicita otro objeto más, y así sucesivamente. Es posible que los vea como una larga lista de métodos getThis o como una secuencia de tiempos. Navegar de esta manera significa que el cliente está acoplado a la estructura de la navegación. Cualquier cambio en las relaciones intermedias hace que el cliente tenga que cambiar.

El movimiento a utilizar aquí es Ocultar delegado. Puedes hacer esto en varios puntos de la cadena. En principio, puedes hacer esto con cada objeto de la cadena, pero hacerlo a menudo convierte a cada objeto intermedio en un intermediario. A menudo, una mejor alternativa es ver para qué se utiliza el objeto resultante. Vea si puede usar Método de extracción tomar una parte del código que lo usa y luego Método de movimiento para empujarlo hacia abajo en la cadena. Si varios clientes de uno de los objetos de la cadena quieren navegar el resto del camino, agregue un método para hacerlo.

Algunas personas consideran que cualquier cadena de métodos es algo terrible. Somos conocidos por nuestra moderación tranquila y razonada. Bueno, al menos en este caso lo somos.

## intermediario

Una de las características principales de los objetos es la encapsulación: ocultar detalles internos al resto del mundo. La encapsulación suele venir acompañada de delegación. Le preguntas a una directora si está libre para una reunión; ella delega el mensaje a su diario y te da una respuesta. Todo muy bien. No es necesario saber si la directora utiliza una agenda, un aparato electrónico o una secretaria para realizar un seguimiento de sus citas.



Sin embargo, esto puede ir demasiado lejos. Observas la interfaz de una clase y descubres que la mitad de los métodos están delegando a esta otra clase. Después de un tiempo es hora de usar. Eliminar intermediario y habla con el objeto que realmente sabe lo que está pasando. Si sólo unos pocos métodos no sirven de mucho, utilice En línea Método para alinearlos con la persona que llama. Si hay un comportamiento adicional, puede utilizar Reemplazar Delegación con Herencia convertir al intermediario en una subclase del objeto real. Eso le permite ampliar el comportamiento sin perseguir toda esa delegación.

## Intimidad inapropiada

A veces las clases se vuelven demasiado íntimas y dedican demasiado tiempo a profundizar en las partes privadas de los demás. Puede que no seamos mojigatos cuando se trata de personas, pero creemos que nuestras clases deben seguir reglas estrictas y puritanas.

Las clases demasiado íntimas deben disolverse como se hacía con los amantes en la antigüedad. Usar Método de movimiento y Mover campo separar las piezas para reducir la intimidad. Vea si puede organizar una Cambiar asociación bidireccional a unidireccional. Si

las clases tienen intereses comunes, utilice Extraer clase poner lo común en un lugar seguro y hacer clases honestas de ellos. O usar Ocultar delegado dejar que otra clase actúe como intermediaria.

La herencia a menudo puede conducir a un exceso de intimidad. Las subclases siempre sabrán más sobre sus padres de lo que a sus padres les gustaría que supieran. Si es hora de salir de casa, postula Reemplazar Delegación con Herencia.

## Clases alternativas con diferentes interfaces

Usar Cambiar nombre del método sobre cualquier método que haga lo mismo pero que tenga firmas diferentes para lo que hacen. A menudo esto no es suficiente. En estos casos las clases aún no están haciendo lo suficiente. seguir usando Método de movimiento trasladar conductas a las clases hasta que los protocolos sean los mismos. Si tiene que mover código de forma redundante para lograr esto, es posible que pueda usar Extracto Superclase para expiar.

## Clase de biblioteca incompleta

A menudo se promociona la reutilización como el propósito de los objetos. Creemos que la reutilización está sobrevalorada (simplemente usamos). Sin embargo, no podemos negar que gran parte de nuestra habilidad de programación se basa en clases de biblioteca, por lo que nadie puede decir si hemos olvidado nuestros algoritmos de clasificación.

Los creadores de clases bibliotecarias rara vez son omniscientes. No los culpamos por eso; después de todo, rara vez podemos idear un diseño hasta que lo hayamos construido en su mayor parte, por lo que los creadores de bibliotecas tienen un trabajo realmente difícil. El problema es que a menudo es de mala educación, y generalmente imposible, modificar una clase de biblioteca para que haga algo que le gustaría que hiciera. Esto significa que tácticas probadas y verdaderas como Método de movimiento mentira inútil.

Disponemos de un par de herramientas especiales para este trabajo. Si solo hay un par de métodos que le gustaría que tuviera la clase de biblioteca, use Introducir método extranjero. Si hay una gran cantidad de comportamiento adicional, es necesario Introducir la extensión local.

## Clase de datos

Estas son clases que tienen campos, métodos de obtención y configuración para los campos, y nada más. Estas clases son tontas poseedoras de datos y es casi seguro que están siendo manipuladas en demasiadas ocasiones.



detalle por otras clases. En etapas tempranas estas clases pueden tener campos públicos. Si es así, debe solicitarlo inmediatamente. Encapsular campo antes de que alguien se dé cuenta. Si tiene campos de colección, verifique si están encapsulados correctamente y se aplican Colección encapsulada si no lo son. Usar Eliminar método de configuración en cualquier campo que no deba modificarse.

Busque dónde otras clases utilizan estos métodos de obtención y configuración. Intenta usar Mover Método para mover el comportamiento a la clase de datos. Si no puedes mover un método completo, usa Extracto Método para crear un método que se pueda mover. Después

de un tiempo puedes empezar a usar Ocultar método en los captadores y definidores.

Las clases de datos son como niños. Están bien como punto de partida, pero para participar como objeto adulto, deben asumir cierta responsabilidad.

## Legado rechazado

Las subclases heredan los métodos y datos de sus padres. Pero ¿qué pasa si no quieren o no necesitan lo que se les da? Reciben todos estos fantásticos regalos y eligen sólo unos pocos para jugar.

La historia tradicional es que esto significa que la jerarquía está equivocada. Necesitas crear una nueva clase de hermanos y usar Método de empuje hacia abajo y Campo de empuje hacia abajo para enviar todos los métodos no utilizados al hermano. De esa manera el padre posee sólo lo que es común. A menudo escucharás consejos de que todas las superclases deben ser abstractas.

Lo adivinarás por nuestro uso sarcástico de *tradicional* que no vamos a aconsejar esto, al menos no todo el tiempo. Creamos subclases para reutilizar un poco de comportamiento todo el tiempo y consideramos que es una forma perfectamente buena de hacer negocios. Hay un olor, no lo podemos negar, pero normalmente no es un olor fuerte. Por eso decimos que si el legado rechazado causa confusión y problemas, siga el consejo tradicional.
Sin embargo, no sienta que tiene que hacerlo todo el tiempo. Nueve de cada diez veces este olor es demasiado débil para que valga la pena limpiarlo.

El olor a legado rechazado es mucho más fuerte si la subclase está reutilizando el comportamiento pero no quiere soportar la interfaz de la superclase. No nos importa rechazar implementaciones, pero rechazar la interfaz nos pone en alto. En este caso, sin embargo, no juegues con la jerarquía; quieres destriparlo aplicando Reemplazar herencia con delegación.

## Comentarios

No te preocupes, no estamos diciendo que la gente no deba escribir comentarios. En nuestra analogía olfativa, los comentarios no son un mal olor; de hecho son un olor dulce. La razón por la que mencionamos los comentarios aquí es que los comentarios a menudo se utilizan como desodorante. Es sorprendente la frecuencia con la que miras un código con muchos comentarios y te das cuenta de que los comentarios están ahí porque el código es malo.

Los comentarios nos llevan a un código incorrecto que tiene todos los malos olores que hemos discutido en el resto de este capítulo. Nuestra primera acción es eliminar los malos olores mediante refactorización. Cuando terminamos, a menudo encontramos que los comentarios son superfluos.

Si necesita un comentario para explicar qué hace un bloque de código, intente Método de extracción. Si el método ya está extraído pero aún necesita un comentario para explicar lo que hace, use Cambiar nombre del método. Si necesita establecer algunas reglas sobre el estado requerido del sistema, use Introducir Afirmación.

**Consejo**

> *Cuando sienta la necesidad de escribir un comentario, primero intente refactorizar el código para que cualquier comentario se vuelva superfluo.*

Un buen momento para utilizar un comentario es cuando no sabes qué hacer. Además de describir lo que está sucediendo, los comentarios pueden indicar áreas en las que no estás seguro. Un comentario es un buen lugar para decir. *por qué* hiciste algo. Este tipo de información ayuda a futuros modificadores, especialmente a los olvidadizos.

# Capítulo 4. Pruebas de construcción

Si desea refactorizar, la condición previa esencial es tener pruebas sólidas. Incluso si tienes la suerte de tener una herramienta que pueda automatizar las refactorizaciones, aún necesitarás pruebas. Pasará mucho tiempo antes de que todas las refactorizaciones posibles puedan automatizarse en una herramienta de refactorización.

No veo esto como una desventaja. Descubrí que escribir buenas pruebas acelera

enormemente mi programación, incluso si no estoy refactorizando. Esto fue una sorpresa para mí y es contradictorio para muchos programadores, por lo que vale la pena explicar por qué.

## El valor del código de autoprueba

Si observa cómo la mayoría de los programadores pasan su tiempo, encontrará que escribir código en realidad es una fracción bastante pequeña. Se dedica algo de tiempo a descubrir qué debería estar sucediendo, algo de tiempo a diseñar, pero la mayor parte del tiempo se dedica a depurar. Estoy seguro de que todos los lectores pueden recordar largas horas de depuración, a menudo hasta bien entrada la noche. Cada programador puede contar la historia de un error que tardó un día entero (o más) en encontrar. Arreglar el error suele ser bastante rápido, pero encontrarlo es una pesadilla.  Y luego, cuando solucionas un error, siempre existe la posibilidad de que aparezca otro y que ni siquiera lo notes hasta mucho más tarde. Luego pasas años encontrando ese error.

El evento que me inició en el camino hacia la autoevaluación del código fue una charla en OOPSLA en el 92. Alguien (creo que fue Dave Thomas) dijo casualmente: "Las clases deberían contener sus propias pruebas". Me pareció una buena forma de organizar las pruebas. Lo interpreté como que cada clase debería tener su propio método (llamado *prueba*) que se puede utilizar para probarse a sí mismo.

En ese momento también estaba en el desarrollo incremental, así que intenté agregar métodos de prueba a las clases a medida que completaba cada incremento. El proyecto en el que estaba trabajando en ese momento era bastante pequeño, por lo que publicábamos incrementos aproximadamente cada semana. Ejecutar las pruebas se volvió bastante sencillo, pero aunque eran fáciles de ejecutar, seguían siendo bastante aburridos. Esto se debió a que cada prueba generaba resultados en la consola que tenía que verificar. Ahora soy una persona bastante vaga y estoy dispuesto a trabajar bastante duro para evitar el trabajo. Me di cuenta de que en lugar de mirar la pantalla para ver si imprimía alguna información del modelo, podía hacer que la computadora hiciera esa prueba. Todo lo que tenía que hacer era poner el resultado que esperaba en el código de prueba y hacer una comparación. Ahora podía ejecutar el método de prueba de cada clase y simplemente imprimiría "OK" en la pantalla si todo estaba bien. La clase estaba ahora en autoevaluación.

> **Consejo**
>
> *Asegúrese de que todas las pruebas sean completamente automáticas y de que verifiquen sus propios resultados.*

Ahora era fácil ejecutar una prueba, tan fácil como compilar. Entonces comencé a ejecutar pruebas cada vez que compilaba. Pronto comencé a notar que mi productividad se había disparado. Me di cuenta de que no estaba dedicando tanto tiempo a depurar. Si agregaba un error detectado en una prueba anterior, aparecería tan pronto como ejecutara esa prueba. Como la prueba había funcionado antes, sabría que el error estaba en el trabajo que había realizado desde la última vez que hice la prueba. Debido a que realicé las pruebas con frecuencia, solo habían transcurrido unos minutos. Entonces supe que la fuente del error era el código que acababa de escribir.  Debido a que ese código estaba fresco en mi mente y era una cantidad pequeña, el error fue fácil de encontrar. Los errores que antes tardaban una hora o más en detectarse ahora tardaban un par de minutos como máximo. No solo había creado clases de autoevaluación, sino que al ejecutarlas con frecuencia tenía un potente detector de errores.

Cuando me di cuenta de esto, me volví más agresivo al hacer las pruebas. En lugar de esperar el final del incremento, agregaría las pruebas inmediatamente después de escribir un poco de función. Todos los días agregaba un par de funciones nuevas y pruebas para probarlas. Hoy en día casi nunca dedico más de unos minutos a depurar.

**Consejo**

*Un conjunto de pruebas es un potente detector de errores que reduce el tiempo que lleva encontrar errores.*

Por supuesto, no es tan fácil persuadir a otros a seguir este camino. Escribir las pruebas implica mucho código adicional que escribir. A menos que haya experimentado realmente la forma en que acelera la programación, la autoevaluación no parece tener sentido. A esto no ayuda el hecho de que muchas personas nunca hayan aprendido a redactar exámenes o incluso a pensar en ellos. Cuando las pruebas son manuales, resultan tremendamente aburridas. Pero cuando son automáticas, las pruebas pueden ser bastante divertidas de escribir.

De hecho, uno de los momentos más útiles para escribir pruebas es antes de empezar a programar. Cuando necesite agregar una función, comience escribiendo la prueba. Esto no es tan atrasado como parece. Al escribir la prueba, se pregunta qué se debe hacer para agregar la función. Escribir la prueba también se concentra en la interfaz más que en la implementación (siempre es algo bueno). También significa que tiene un punto claro en el que ha terminado de codificar: cuando la prueba funciona.

Esta noción de pruebas frecuentes es una parte importante de la programación extrema [Beck, XP]. El nombre evoca nociones de programadores que son hackers rápidos y relajados. Pero los programadores extremos son probadores muy dedicados. Quieren desarrollar software lo más rápido posible y saben que las pruebas le ayudan a hacerlo lo más rápido posible.

Ya basta de polémica. Aunque creo que todo el mundo se beneficiaría si escribiera código de autoprueba, ese no es el objetivo de este libro. Este libro trata sobre la refactorización. La refactorización requiere pruebas. Si desea refactorizar, debe escribir pruebas. Este capítulo le ofrece un comienzo para hacer esto para Java. Este no es un libro de pruebas, por lo que no entraré en muchos detalles. Pero con las pruebas descubrí que una cantidad notablemente pequeña puede tener beneficios sorprendentemente grandes.

Como ocurre con todo lo demás en este libro, describo el enfoque de prueba mediante ejemplos. Cuando desarrollo código, escribo las pruebas sobre la marcha. Pero a menudo, cuando trabajo con personas en la refactorización, tenemos un cuerpo de código que no se prueba automáticamente en el que trabajar. Entonces, primero tenemos que hacer que el código se autopruebe antes de refactorizar.

El modismo estándar de Java para las pruebas es testing main. La idea es que cada clase debería tener una función principal que pruebe la clase. Es una convención razonable (aunque no se respeta mucho), pero puede resultar incómoda. El problema es que dicha convención dificulta la ejecución de muchas pruebas.
fácilmente. Otro enfoque es crear clases de prueba independientes que funcionen en un

marco para facilitar las pruebas.

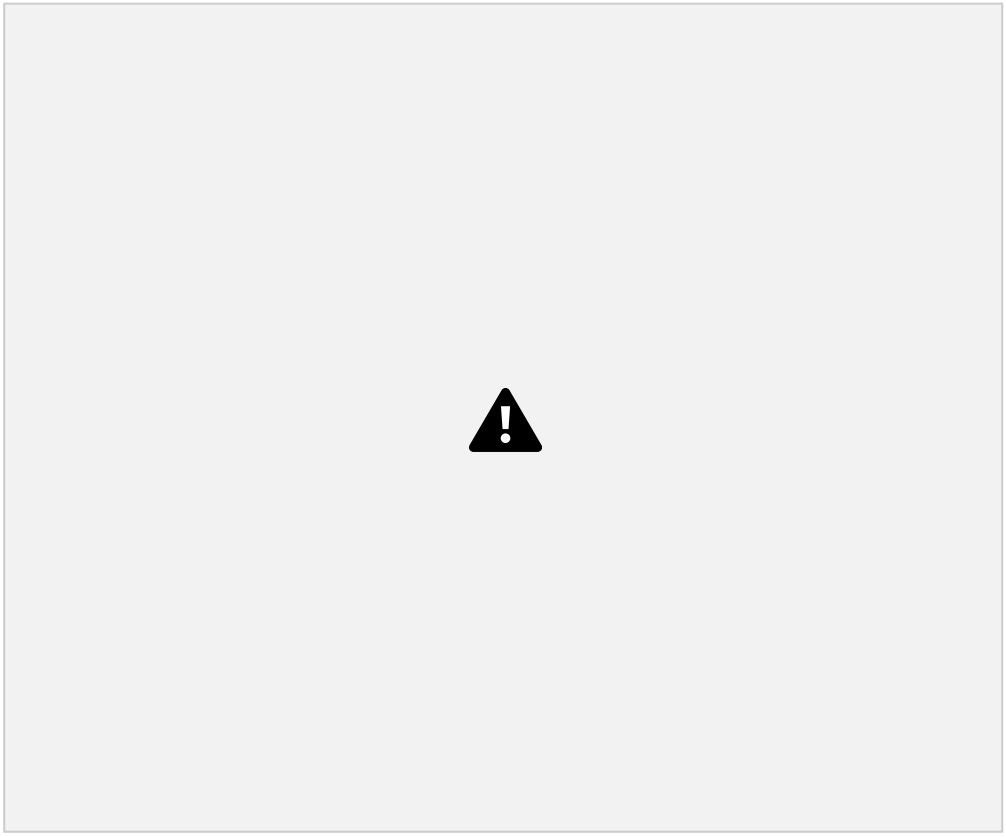
## El marco de pruebas JUnit

El marco de pruebas que utilizo es JUnit, un marco de pruebas de código abierto desarrollado por Erich Gamma y Kent Beck [JUnit]. El marco es muy simple, pero le permite hacer todas las cosas clave que necesita para las pruebas. En este capítulo utilizo este marco para desarrollar pruebas para algunas clases de io.



Para empezar creo una clase FileReaderTester para probar el lector de archivos. Cualquier clase que contenga pruebas debe subclasificar la clase de caso de prueba del marco de pruebas. El marco utiliza el patrón compuesto [Gang of Four] que le permite agrupar pruebas en conjuntos (Figura 4.1). Estos conjuntos pueden contener casos de prueba sin procesar u otros conjuntos de casos de prueba. Esto facilita la creación de una variedad de conjuntos de pruebas de gran tamaño y la ejecución de las pruebas automáticamente.

**Figura 4.1. La estructura compuesta de las pruebas.**



```
clase FileReaderTester extiende TestCase {
 public FileReaderTester (nombre de cadena) {
 super(nombre);
 }
 }
```

La nueva clase debe tener un constructor. Después de esto puedo comenzar a agregar algún código de prueba. Mi primer trabajo es configurar el dispositivo de prueba. Un dispositivo de

prueba son esencialmente los objetos que actúan como muestras para la prueba. Como estoy leyendo un archivo, necesito configurar un archivo de prueba, de la siguiente manera:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Bradman | 99,94 | 52 | 80 | 10 | 6996 | 334 | 29 |
| abadejo | 60,97 | 23 | 41 | 4 | 2256 | 274 | 7 |
| cabeza | 60,83 | 22 | 40 | 4 | 2256 | 270* | 10 |
| Sutcliffe | 60,73 | 54 | 84 | 9 | 4555 | 194 | 16 |

Para seguir utilizando el archivo, preparo el dispositivo. La clase de caso de prueba proporciona dos métodos para manipular el dispositivo de prueba: `configuración` crea los objetos y `demoler` los elimina. Ambos son



implementado como métodos nulos en el caso de prueba. La mayoría de las veces no es necesario derribar el archivo (el recolector de basura puede hacerlo), pero es aconsejable usarlo aquí para cerrar el archivo, de la siguiente manera:

```
clase FileReaderTester...
 configuración de vacío protegido() {
 intentar {
 _input = nuevo FileReader("datos.txt");
 } captura (FileNotFoundException e) {
 lanzar una nueva RuntimeException ("no se puede abrir el archivo de
prueba");  }
 }

 desgarro del vacío protegido() {
 intentar {
 _input.close();
 } captura (IOException e) {
 lanzar una nueva RuntimeException ("error al cerrar el archivo de
prueba");
 }
 }
```

Ahora que tengo el dispositivo de prueba instalado, puedo comenzar a escribir pruebas. El primero es probar el método de lectura. Para hacer esto leo algunos caracteres y luego compruebo que el carácter que leo a continuación es el correcto:

```
testRead nulo público () lanza IOException {
 char ch = '&';
 para (int i=0; i < 4; i++)
 ch = (char) _input.read();
 afirmar('d' == ch);
 }
```

La prueba automática es el método de afirmación. Si el valor dentro de la afirmación es verdadero, todo está bien. De lo contrario señalamos un error. Más adelante muestro cómo lo hace el marco.

Primero describo cómo ejecutar la prueba.

El primer paso es crear un conjunto de pruebas. Para hacer esto, cree un método llamado *suite:*

```
clase FileReaderTester...
 conjunto de pruebas estáticas públicas () {
 Suite TestSuite= nueva TestSuite();
 suite.addTest(new FileReaderTester("testRead"));  suite de
regreso;
 }
```

Este conjunto de pruebas contiene sólo un objeto de caso de prueba, una instancia de `Probador de lectura de archivos.` Cuando creo un caso de prueba, le doy al constructor un argumento de cadena, que es el nombre del método que voy a probar. Esto crea un objeto que prueba ese método. La prueba está vinculada al objeto mediante la capacidad de reflexión de Java. Puede echar un vistazo al código fuente descargado para descubrir cómo lo hace. Simplemente lo trato como magia.



Para ejecutar las pruebas, utilice una clase TestRunner independiente. Hay dos versiones de TestRunner: una usa una GUI genial y la otra una interfaz de caracteres simple. Puedo llamar a la versión de la interfaz del personaje en el archivo principal:

```
clase FileReaderTester...
 público estático vacío principal (String[] args) {
 junit.textui.TestRunner.run (suite());
 }
```

El código crea el ejecutor de pruebas y le indica que pruebe la clase FileReaderTester. Cuando lo ejecuto veo

```
.
 Tiempo: 0,110

 OK (1 pruebas)
```

JUnit imprime un período para cada prueba que se ejecuta (para que pueda ver el progreso). Le indica cuánto tiempo se han ejecutado las pruebas. Luego dice "OK" si nada sale mal y le indica cuántas pruebas se han ejecutado. Puedo realizar mil pruebas y, si todo va bien, lo veré bien. Esta simple retroalimentación es esencial para la autocomprobación del código. Sin él, nunca ejecutará las pruebas con la suficiente frecuencia. Con él, puede realizar una gran cantidad de pruebas, salir a almorzar (o a una reunión) y ver los resultados cuando regrese.

**Consejo**

*Ejecute sus pruebas con frecuencia. Localice las pruebas cada vez que las compile: cada prueba al menos todos los días.*

Al refactorizar, ejecuta sólo unas pocas pruebas para ejercitar el código en el que está trabajando. Sólo puedes ejecutar unos pocos porque deben ser rápidos; de lo contrario, te ralentizarán y te verás tentado a no ejecutarlos. No cedas a esa tentación: retribución. *voluntad* seguir.

¿Qué pasa si algo sale mal? Lo demostraré poniendo un error deliberado, de la siguiente manera:

```
testRead nulo público () lanza IOException {
 char ch = '&';
 para (int i=0; i < 4; i++)
 ch = (char) _input.read();
 afirmar('2' == ch); //error deliberado
 }
```

El resultado se ve así:

```
.F
 Tiempo: 0,220

 !!!FALLOS!!!
```



```
Resultados de la prueba:
 Ejecutar: 1 Fallos: 1 Errores: 0
 Hubo 1 fallo:
 1) FileReaderTester.testRead
 prueba.framework.AssertionFailedError
```

El marco me alerta del error y me dice qué prueba falló. Sin embargo, el mensaje de error no es particularmente útil. Puedo mejorar el mensaje de error usando otra forma de afirmación.

```
testRead nulo público () lanza IOException {
 char ch = '&';
 para (int i=0; i < 4; i++)
 ch = (char) _input.read();
 afirmarEquals('m',ch);
 }
```

La mayoría de las afirmaciones que hace comparan dos valores para ver si son iguales. Entonces el marco incluye `afirmarEquals.` Esto es conveniente; se utiliza `es igual()` sobre objetos y `==` sobre valores, algo que a menudo olvido hacer. También permite un mensaje de error más significativo:

```
.F
 Tiempo: 0,170
```

```
!!!FALLOS!!!
Resultados de la prueba:
Ejecutar: 1 Fallos: 1 Errores: 0
Hubo 1 fallo:
1) FileReaderTester.testRead "esperado:"m"pero era:"d""
```

Debo mencionar que a menudo cuando escribo pruebas, empiezo por hacerlas fallar. Con el código existente, lo cambio para que falle (si puedo tocar el código) o pongo un valor esperado incorrecto en la afirmación. Hago esto porque me gusta demostrarme a mí mismo que la prueba realmente se ejecuta y que en realidad está probando lo que se supone que debe hacer (por eso prefiero cambiar el código probado si puedo). Esto puede ser paranoia, pero usted puede realmente confundirse cuando las pruebas prueban algo distinto de lo que usted cree que están probando.

Además de detectar fallas (afirmaciones que resultan falsas), el marco también detecta errores (excepciones inesperadas). Si cierro la transmisión y luego intento leerla, debería obtener una excepción. Puedo probar esto con

```
testRead nulo público () lanza IOException {
 char ch = '&';
 _input.close();
 para (int i=0; i < 4; i++)
 ch = (char) _input.read(); // lanzará una excepción
afirmarEquals('m',ch);
 }
```

Si ejecuto esto obtengo

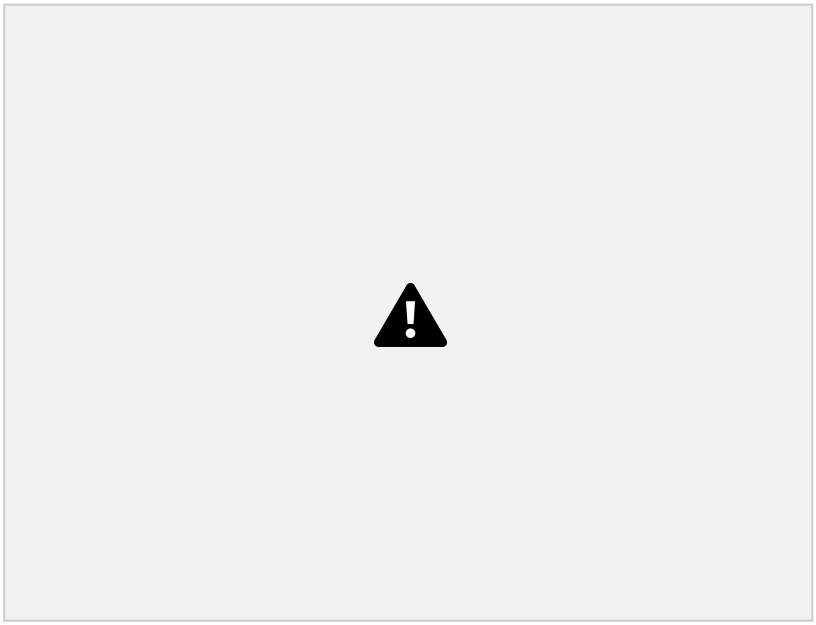


```
.Y

Tiempo: 0,110

!!!FALLOS!!!
Resultados de la prueba:
Ejecutar: 1 Fallos: 0 Errores: 1
Hubo 1 error:
1) FileReaderTester.testRead
java.io.IOException: transmisión cerrada
```

Es útil diferenciar fallas y errores, porque tienden a aparecer de manera diferente y el proceso de depuración es diferente.

JUnit también incluye una bonita GUI (Figura 4.2). La barra de progreso se muestra en verde si se pasan todas las pruebas y en rojo si hay alguna falla. Puede dejar la GUI activada todo el tiempo y el entorno vincula automáticamente cualquier cambio en su código. Esta es una forma muy conveniente de ejecutar las pruebas.

**Figura 4.2. La interfaz gráfica de usuario de JUnit**



## Pruebas unitarias y funcionales

Este marco se utiliza para pruebas unitarias, por lo que debo mencionar la diferencia entre pruebas unitarias y pruebas funcionales. Las pruebas de las que estoy hablando son *pruebas unitarias.* Los escribo para mejorar mi productividad como programador. Hacer feliz al departamento de control de calidad es sólo un efecto secundario. Las pruebas unitarias están altamente localizadas. Cada clase de prueba funciona dentro de un solo paquete. Prueba las interfaces con otros paquetes, pero más allá de eso asume que el resto simplemente funciona.

*Pruebas funcionales* Eres un animal diferente. Están escritos para garantizar que el software en su conjunto funcione.  Proporcionan garantía de calidad al cliente y no se preocupan por la productividad del programador.  Deberían ser desarrollados por un equipo diferente, uno que se deleite en encontrar errores. Este equipo utiliza herramientas y técnicas pesadas para ayudarlos a lograr esto.



Las pruebas funcionales suelen tratar todo el sistema como una caja negra en la medida de lo posible. En un sistema basado en GUI, operan a través de la GUI. En un programa de actualización de archivos o bases de datos, las pruebas simplemente analizan cómo se cambian los datos para ciertas entradas.

Cuando los evaluadores funcionales, o los usuarios, encuentran un error en el software, se necesitan al menos dos cosas para solucionarlo. Por supuesto, debes cambiar el código de producción para eliminar el error. Pero también deberías agregar una prueba unitaria que exponga el error. De hecho, cuando recibo un informe de error, empiezo escribiendo una prueba unitaria que hace que el error surja. Escribo más de una prueba si necesito limitar el alcance del error o si
puede haber fallas relacionadas. Utilizo las pruebas unitarias para ayudar a identificar el error y asegurarme de que un error similar no vuelva a pasar mis pruebas unitarias.

**Consejo**

*Cuando reciba un informe de error, comience escribiendo una prueba unitaria que exponga el error.*

El marco JUnit está diseñado para escribir pruebas unitarias. Las pruebas funcionales suelen realizarse con otras herramientas. Las herramientas de prueba basadas en GUI son buenos ejemplos. Sin embargo, a menudo escribirá sus propias herramientas de prueba específicas de la aplicación que harán que sea más fácil administrar casos de prueba que los scripts GUI solos. Puedes realizar pruebas funcionales con JUnit, pero normalmente no es la forma más eficiente. Para propósitos de refactorización, cuento con las pruebas unitarias: el amigo del programador.

## Agregar más pruebas

Ahora deberíamos seguir añadiendo más pruebas. El estilo que sigo es observar todas las cosas que la clase debe hacer y probar cada una de ellas para detectar cualquier condición que pueda causar que la clase falle. Esto no es lo mismo que "probar todos los métodos públicos", como defienden algunos programadores. Las pruebas deben estar impulsadas por el riesgo; Recuerde, está intentando encontrar errores ahora o en el futuro. Por eso no pruebo descriptores de acceso que simplemente leen y escriben un campo. Como son tan simples, no es probable que encuentre ningún error allí.

Esto es importante porque intentar escribir demasiadas pruebas generalmente conduce a no escribir las suficientes. A menudo he leído libros sobre pruebas y mi reacción ha sido la de alejarme de la montaña de cosas que tengo que hacer para realizar las pruebas. Esto es contraproducente, porque te hace pensar que para probar hay que trabajar mucho. Obtiene muchos beneficios de las pruebas incluso si solo realiza unas pocas pruebas. La clave es probar las áreas que más le preocupan que salgan mal. De esa manera obtendrá el mayor beneficio por su esfuerzo de prueba.

**Consejo**

*Es mejor escribir y ejecutar pruebas incompletas que no ejecutar pruebas completas.*

Por el momento estoy mirando el método de lectura. ¿Qué más debería hacer? Una cosa que dice es que vuelve. `-1` al final del archivo (no es un protocolo muy bueno en mi opinión, pero supongo que eso lo hace más natural para los programadores de C). Probémoslo. Mi editor de texto me dice que hay 141 caracteres en el archivo, así que aquí está la prueba: