Refactorización: mejora del diseño del código existente

por Martin Fowler , Kent Beck (Colaborador), John Brant (Colaborador), William Opdyke , don Roberts

Otro estúpido lanzamiento 2002 ☺

Para todas las personas que no tienen dinero para comprar un buen libro

Página 2

La biblioteca de tu clase funciona, pero ¿podría ser mejor? *Refactorización: mejora del diseño de El Código existente* muestra cómo la *refactorización* puede hacer que el código orientado a objetos sea más simple y fácil mantener. Hoy la refactorización requiere un considerable conocimiento del diseño, pero una vez que las herramientas estén disponibles, todos los programadores deberían poder mejorar su código utilizando la refactorización técnicas

Además de una introducción a la refactorización, este manual proporciona un catálogo de docenas de consejos. para mejorar el código. Lo mejor de la *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 mejor libro, uno que debería ser accesible de inmediato para cualquier intermedio o avanzado desarrollador orientado a objetos. (Al igual que los patrones, cada punta de refactorización se presenta con un nombre simple, una "motivación" y ejemplos usando Java y UML.)

Los primeros capítulos enfatizan la importancia de las pruebas en una refactorización exitosa. (Cuando tú Para mejorar el código, debe probar para verificar que todavía funciona.) Después de la discusión sobre cómo Para 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 clase mejor y más simple. Cada punta se ilustra con código "antes" y "después", junto con una explicación. Los capítulos posteriores proporcionan un vistazo rápido en la refactorización de la investigación.

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

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

Book News, Inc.

Una guía para refactorizar, el proceso de cambiar un sistema de software para que no altere el comportamiento externo del código mejora su estructura interna, para profesionales programadores Los primeros capítulos cubren principios generales, fundamentos, ejemplos y pruebas. El corazón del libro es un catálogo de refactorizaciones, organizado en capítulos sobre composición métodos, mover características entre objetos, organizar datos, simplificar condicional expresiones y tratar con generalizaciones

Página 3

Página 4

Página 6

Prefacio

La "refactorización" se concibió en los círculos de Smalltalk, pero no pasó mucho tiempo antes de que llegara a otros campamentos de lenguaje de programación. Debido a que la refactorización es parte integral del desarrollo del marco, el término aparece rápidamente cuando los "armadores" hablan de su oficio. Surge cuando ellos refinar sus jerarquías de clase y cuando deliran sobre cuántas líneas de código pudieron Eliminar. Los frameworkers saben que un framework no será correcto la primera vez, debe evolucionar a medida que ganan experiencia. También saben que el código se leerá y modificará con más frecuencia. de lo que se escribirá La clave para mantener el código legible y modificable es la refactorización, para frameworks, en particular, pero también para software en general.

¿Entonces, cuál es el problema? Simplemente esto: la refactorización es arriesgada. Requiere cambios en el código de trabajo que Puede introducir errores sutiles. Refactorizar, si no se hace correctamente, puede retrasar días, incluso semanas. Y la refactorización se vuelve más riesgosa cuando se practica de manera informal o ad hoc. Empiezas a cavar en el código. Pronto descubre nuevas oportunidades de cambio y profundiza. Cuanto más cavas, Cuantas más cosas aparezcas ... y más cambios hagas. Finalmente te cavas en un agujero del que no puedes salir. Para evitar cavar su propia tumba, se debe refactorizar sistemáticamente. Cuando mis coautores y yo escribimos *Patrones de diseño,* mencionamos ese diseño patrones proporcionan objetivos para refactorizaciones. Sin embargo, identificar el objetivo es solo una parte de problema; transformar su código para que llegue allí es otro desafío.

Martin Fowler y los autores contribuyentes hacen una contribución invaluable a la orientación a objetos. desarrollo de software al arrojar luz sobre el proceso de refactorización. Este libro explica el principios y mejores prácticas de refactorización, y señala cuándo y dónde debe comenzar cavando en su código para mejorarlo. En el núcleo del libro hay un catálogo completo de refactorizaciones. Cada refactorización describe la motivación y la mecánica de una transformación de código probada. Algunos de las refactorizaciones, como Método de extracción o Mover campo, puede parecer obvio.

Pero no te dejes engañar. Comprender la mecánica de tales refactorizaciones es la clave para refactorizar en de manera disciplinada Las refactorizaciones en este libro lo ayudarán a cambiar su código en un pequeño paso en un tiempo, reduciendo así los riesgos de evolucionar su diseño. Agregará rápidamente estas refactorizaciones y sus nombres para tu vocabulario de desarrollo.

Mi primera experiencia con la refactorización disciplinada, "paso a paso", fue cuando estaba en pareja. programación a 30,000 pies con Kent Beck. Se aseguró de que apliquemos refactorizaciones de esto catálogo de libros paso a paso. Me sorprendió lo bien que funcionó esta práctica. No solo mi confianza en el aumento del código resultante, también me sentí menos estresado. Te recomiendo que lo pruebes Estas refactorizaciones: usted y su código se sentirán mucho mejor por ello.

Página 7

—Erich Gamma

Object Technology International, Inc.

Página 8

Prefacio

Érase una vez, un consultor hizo una visita a 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 deambulaba por la jerarquía, el consultor vio que era bastante desordenado. Lo mas alto-
las clases de nivel hicieron ciertas suposiciones sobre cómo funcionarían las clases, suposiciones que
fueron incorporados en código heredado. Sin embargo, ese código no se adaptaba a todas las subclases y era
anulado en gran medida. Si la superclase se hubiera modificado un poco, entonces mucho menos se anularía
Habría sido necesario. En otros lugares, parte de la intención de la superclase no había sido
correctamente entendido, y el comportamiento presente en la superclase fue duplicado. En otros lugares
Varias subclases hicieron lo mismo con el código que claramente podría moverse hacia arriba en la jerarquía.

El consultor recomendó a la gerencia del proyecto que el código sea revisado y limpiado
arriba, pero la gestión del proyecto no parecía entusiasta. El código parecía funcionar y allí
fueron considerables presiones de horario. Los gerentes dijeron que lo conseguirían en algún momento
punto posterior

El consultor también les mostró a los programadores que habían trabajado en la jerarquía lo que era
pasando Los programadores estaban interesados y vieron el problema. Sabían que no era realmente su
culpa; a veces se necesita un nuevo par de ojos para detectar el problema. Entonces los programadores pasaron un
uno o dos días limpiando la jerarquía. Cuando terminaron, los programadores habían eliminado
la mitad del código en la jerarquía sin reducir su funcionalidad. Estaban satisfechos con el resultado.
y descubrí que se hizo más rápido y fácil agregar nuevas clases a la jerarquía y usar
las clases en el resto del sistema.

La gestión del proyecto no estaba satisfecha. Los horarios eran ajustados y había mucho trabajo por hacer
hacer. Estos dos programadores habían pasado dos días haciendo un trabajo que no había hecho nada para agregar el
Muchas características que el sistema tenía que entregar en unos pocos meses. El viejo código había funcionado bien.
Así que el diseño era un poco más "puro", un poco más "limpio". El proyecto tuvo que enviar el código que funcionó,
No es un código que complacería a un académico. El consultor sugirió que se realizara esta limpieza.
en otras partes centrales del sistema. Tal actividad podría detener el proyecto por una semana o dos. Todas
esta actividad se dedicó a hacer que el código se vea mejor, no a hacer que haga algo que no hizo
ya lo hacen.

¿Cómo te sientes acerca de esta historia? ¿Crees que el consultor tenía razón al sugerir más limpieza?
¿arriba? ¿O sigues ese viejo adagio de ingeniería, "si funciona, no lo arregles"?

Debo admitir que hay algunos prejuicios aquí. Yo fui ese consultor. Seis meses después, el proyecto fracasó, en general.
parte porque el código era demasiado complejo para depurar o sintonizar con un rendimiento aceptable.

El consultor Kent Beck fue contratado 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
importante era insistir en la limpieza continua del código mediante refactorización. El éxito de
Este proyecto, y la refactorización del papel desempeñado en este éxito, es lo que me inspiró a escribir este libro, así que
que podría transmitir el conocimiento que Kent y otros han aprendido al usar 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 altera el
El comportamiento externo del código mejora su estructura interna. Es una forma disciplinada de limpiar
código que minimiza las posibilidades de introducir errores. En esencia, cuando refactorizas eres
mejorar el diseño del código después de que se haya escrito.

"Mejora del diseño después de que se haya escrito". Ese es un extraño giro de la frase. En nuestra corriente
comprensión del desarrollo de software creemos que diseñamos y luego codificamos. Un bien
el diseño es lo primero, y la codificación es lo segundo. Con el tiempo, el código se modificará y el
La integridad del sistema, su estructura de acuerdo con ese diseño, se desvanece gradualmente. El código lentamente
se hunde desde la ingeniería hasta la piratería.

Refactorizar es lo contrario de esta práctica. Con la refactorización puedes tomar un mal diseño, caos
incluso, y reelaborarlo en un código bien diseñado. Cada paso es simple, incluso simplista. Te mueves un
de una clase a otra, extraiga un código de un método para convertirlo en su propio método, y
empujar un código hacia arriba o hacia abajo en una jerarquía. Sin embargo, el efecto acumulativo de estos pequeños cambios puede
Mejorar radicalmente el diseño. Es el reverso exacto de la noción normal de deterioro del software.

Con la refactorización encontrará el equilibrio de los cambios de trabajo. Encuentras ese diseño, en lugar de ocurrir
todo por adelantado, ocurre continuamente durante el desarrollo. Aprende de la construcción del sistema cómo
Mejorar el diseño. La interacción resultante conduce a un programa con un diseño que se mantiene bien
El desarrollo continúa.

¿Qué hay en este libro?

Este libro es una guía para refactorizar; Está escrito para un programador profesional. Mi objetivo es mostrar
cómo refactorizar de manera controlada y eficiente. Aprenderás a refactorizar en tal
de manera que no se introducen errores en el código sino que se mejora metódicamente la estructura.

Es tradicional comenzar libros con una introducción. Aunque estoy de acuerdo con ese principio, no lo encuentro
refactorización fácil de introducir con una discusión generalizada o definiciones. Entonces empiezo con un
ejemplo. El 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
y la aplicación de varias refactorizaciones útiles. Este es el capítulo clave para leer si quieres
Entiende de qué se trata la refactorización.

En el Capítulo 2 cubro más de los principios generales de refactorización, algunas definiciones y los
razones para refactorizar. Describo algunos de los problemas con la refactorización. En el capítulo 3 Kent
Beck me ayuda a describir cómo encontrar malos olores en el código y cómo limpiarlos con
refactorizaciones. Las pruebas juegan un papel muy importante en la refactorización, por lo que el Capítulo 4 describe cómo
cree pruebas en código con un marco de prueba Java de código abierto simple.

El corazón del libro, el catálogo de refactorizaciones, se extiende desde el Capítulo 5 hasta el Capítulo 12.
De ninguna manera es un catálogo completo. Es el comienzo de tal catálogo. Incluye
Las refactorizaciones que he escrito hasta ahora en mi trabajo en este campo. Cuando quiero hacer
algo, como Reemplazar condicional con polimorfismo , el catálogo me recuerda cómo
hágalo de manera 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 invitados.
capítulos de algunas de estas personas. El Capítulo 13 es de Bill Opdyke, quien describe los problemas que él
ha llegado a adoptar la refactorización en el desarrollo comercial. Capítulo 14 es de Don

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

Refactorización en Java

Para todo este libro, uso ejemplos en Java. La refactorización puede, por supuesto, hacerse con otros idiomas, y espero que este libro sea útil para quienes trabajan con otros idiomas. Sin embargo, yo Sentí que sería mejor enfocar este libro en Java porque es el lenguaje que mejor conozco. yo tengo agregué notas ocasionales para refactorizar en otros idiomas, pero espero que otras personas construyan sobre esta fundación con libros destinados a idiomas específicos.

Para ayudar a comunicar mejor las ideas, no he usado áreas particularmente complejas de Java idioma. Así que he evitado usar clases internas, reflexión, hilos y muchos otros Las características más potentes de Java. Esto se debe a que quiero centrarme en las refactorizaciones centrales tan claramente como puedo.

Debo enfatizar que estas refactorizaciones no se realizan con concurrencia o distribución programación en mente. Esos temas presentan 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 escribe software para ganarse la vida. los Los ejemplos y la discusión incluyen mucho código para leer y comprender. Los ejemplos están todos en Java. Elegí Java porque es un lenguaje cada vez más conocido que puede ser fácilmente entendido por cualquier persona con experiencia en C. También es un lenguaje orientado a objetos, y Los mecanismos orientados son de gran ayuda en la refactorización.

Aunque se centra en el código, la refactorización tiene un gran impacto en el diseño del sistema. Está Es vital para los diseñadores y arquitectos senior comprender los principios de refactorización y utilizar ellos en sus proyectos. La refactorización es mejor introducida por un desarrollador respetado y experimentado. Tal desarrollador puede comprender mejor los principios detrás de la refactorización y adaptar esos principios para el lugar de trabajo específico. Esto es particularmente cierto cuando estás usando un idioma diferente que Java, porque tienes que adaptar los ejemplos que he dado a otros idiomas.

Aquí se explica cómo aprovechar al máximo este libro sin leerlo todo.

- **Si desea comprender qué es la refactorización,** lea el Capítulo 1 ; el ejemplo debería aclarar el proceso
- **Si quiere entender por qué debería refactorizar,** lea los primeros dos capítulos. Ellos le dirá qué es la refactorización y por qué debería hacerlo.
- **Si desea encontrar dónde debe refactorizar,** lea el Capítulo 3 . Te dice los signos eso sugiere la necesidad de refactorizar.
- **Si realmente desea refactorizar,** lea los primeros cuatro capítulos por completo. Entonces salta-lee el catálogo. Lea lo suficiente del catálogo para saber aproximadamente qué hay allí. Tú No tiene que entender todos los detalles. Cuando realmente necesitas llevar a cabo un refactorización, lea la refactorización en detalle y úsela para ayudarlo. El catálogo es una referencia. sección, por lo que probablemente no querrá leerlo de una vez. También deberías leer al invitado capítulos, especialmente el Capítulo 15 .

Sobre la base de los cimientos puestos por otros

Necesito decir ahora, al principio, que tengo una gran deuda con este libro, una deuda con aquellos cuyo trabajo en la última década ha desarrollado el campo de la refactorización. Idealmente uno de ellos Debería haber escrito este libro, pero terminé siendo el que tenía el tiempo y la energía.

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

Ralph Johnson dirige un grupo en la Universidad de Illinois en Urbana-Champaign que es notable 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 detallado trabajo escrito sobre refactorización en su tesis doctoral. **John Brant** y **Don Roberts se** han ido más allá de escribir palabras para escribir una herramienta, el Refactoring Browser, para refactorizar Smalltalk programas

Expresiones de gratitud

Incluso con toda esa investigación a la que recurrir, todavía necesitaba mucha ayuda para escribir este libro. Primero y sobre todo, Kent Beck fue de gran ayuda. Las primeras semillas se plantaron en un bar en Detroit cuando Kent me contó sobre un artículo que estaba escribiendo para el *Informe Smalltalk* [Beck, hanoi]. No solo proporcionó Muchas ideas para robar para el Capítulo 1, pero también me ayudaron a tomar notas de refactorizaciones. Kent también ayudó en otros lugares. Se le ocurrió la idea de los olores de código, me animó a varios puntos fijos, y generalmente trabajó conmigo para hacer que este libro funcione. No puedo evitar pensar él mismo podría haber escrito este libro mucho mejor, pero tuve el tiempo y solo puedo esperar haberlo hecho el sujeto justicia.

Mientras escribía esto, quería compartir gran parte de esta experiencia directamente con usted, así que estoy muy agradecido. que muchas de estas personas han pasado algún tiempo agregando material a este libro. Kent Beck, John Brant, William Opdyke y Don Roberts tienen capítulos escritos o coescritos. Adicionalmente, Rich Garzaniti y Ron Jeffries han agregado barras laterales útiles.

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 narices duras revisores Éstas eran

- 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 agregaron 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 destacar un par de muy visibles sugerencias que marcaron una diferencia en el aspecto del libro. Ward y Ron me hicieron hacer el capítulo

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

Además del panel de revisión oficial, hubo muchos revisores no oficiales. Estas personas miraron en el manuscrito o en el trabajo en progreso en mis páginas web e hice comentarios útiles. Ellos 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 Estoy seguro de que hay otros a quienes he 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 necesita ser probada en un sistema de producción serio. Vi refactorización tener un enorme efecto sobre el sistema de compensación integral de Chrysler (C3). Quiero agradecer a todos 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. Trabajando con ellos cimenté los principios y beneficios de refactorizarme de primera mano.
Ver su progreso a medida que usan la refactorización me ayuda a ver qué puede hacer la refactorización cuando se aplica a un gran proyecto durante muchos años.

Nuevamente tuve 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 un buen editor es un placer; proporcionaron una gran cantidad de Apoyo y ayuda.

Hablando de apoyo, la mayor víctima de un libro es siempre la más cercana al autor, en este caso mi (ahora) esposa Cindy. Gracias por amarme incluso cuando estaba escondido en el estudio. Como mucho vez que escribí en este libro, nunca dejé de distraerme pensando en ti.

Martin 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 comenzar a hablar sobre algo es para delinear la historia, principios generales y similares. Cuando alguien hace eso en una conferencia, yo tener un poco de sueño Mi mente comienza a divagar con un proceso de fondo de baja prioridad que sondea el orador hasta que él o ella dé un ejemplo. Los ejemplos me despiertan porque es con ejemplos que puedo ver lo que está pasando Con los principios es demasiado fácil hacer generalizaciones, demasiado difícil averiguar cómo aplicar las cosas. Un ejemplo ayuda a aclarar las cosas.

Así que voy a comenzar este libro con un ejemplo de refactorización. Durante el proceso te contaré muchas cosas sobre cómo funciona la refactorización y darle una idea del proceso de refactorización. Entonces puedo Proporcionar la introducción habitual de estilo de principios.

Con un ejemplo introductorio, sin embargo, me encuentro con un gran problema. Si elijo un programa grande, describirlo y cómo se refactoriza es demasiado complicado para cualquier lector. (Lo intenté e incluso un ejemplo un poco complicado tiene más de cien páginas.) Sin embargo, si elijo un programa que es lo suficientemente pequeño como para ser comprensible, la refactorización no parece que valga la pena.

Por lo tanto, estoy en el clásico vínculo de cualquiera que quiera describir técnicas que sean útiles para programas mundiales Francamente, no vale la pena hacer la refactorización que les voy a mostrar en un pequeño programa como el que voy a usar. Pero si el código que te estoy mostrando es parte de un sistema más grande, entonces la refactorización pronto se vuelve importante. Así que tengo que pedirte que mires esto e imagínelo en el contexto de un sistema mucho más grande.

El punto de partida

El programa de muestra es muy simple. Es un programa para calcular e imprimir una declaración de un Cargos del cliente en una tienda de videos. Se le dice al programa qué películas alquiló un cliente y por cuánto tiempo. Luego calcula los cargos, que dependen de cuánto tiempo se alquila la película, y identifica el tipo de película. Hay tres tipos de películas: regulares, infantiles y nuevos estrenos. Además de calcular los cargos, la declaración también calcula los puntos frecuentes de arrendamiento, que varían dependiendo de si la película es un nuevo lanzamiento.

Varias clases representan varios elementos de video. Aquí hay un diagrama de clase para mostrarlos (Figura 1.1)

Figura 1.1. Diagrama de clases de las clases de punto de partida. Solo las características más importantes son exhibidos. La notación es Unified Modeling Language UML [Fowler, UML].

Mostraré el código para cada una de estas clases a su vez.

Película

La película es solo una clase de datos simple.

```
Película de clase pública {

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

   cadena privada _title;
   private int _priceCode;

   Public Movie (String title, int priceCode) {
        _title = title;
        _priceCode = priceCode;
   }

   public int getPriceCode () {
        return _priceCode;
   }

   public void setPriceCode (int arg) {
      _priceCode = arg;
   }

   public String getTitle () {
        return _title;
   };
}
```

Alquiler

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

```
Alquiler de clase {
     Película privada _película;
     privado int _daysRented;

     Alquiler público (película, días int alquilados) {
        _movie = película;
        _daysRented = daysRented;
     }
     public int getDaysRented () {
        return _daysRented;
     }
     película pública getMovie () {
        volver _película;
     }
}
```

Cliente

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

```
clase Cliente {
   cadena privada _name;
   Private Vector _rentals = new Vector ();

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

   public void addRental (Argumento de alquiler) {
      _rentals.addElement (arg);
   }
   public String getName () {
        return _name;
   };
```

El cliente también tiene el método que produce una declaración. La 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 () {
          double totalAmount = 0;
          int frecuenteRenterPoints = 0;
          Enumeración alquileres = _rentals.elements ();
          String result = "Registro de alquiler para" + getName () + "\ n";
          while (rentals.hasMoreElements ()) {
               double thisAmount = 0;
               Alquiler de cada = (Alquiler) rentals.nextElement ();

               // determina cantidades para cada línea
               switch (each.getMovie (). getPriceCode ()) {
                    caso Movie.REGULAR:
                         thisAmount + = 2;
                         if (each.getDaysRented ()> 2)
```


Página 16

```
                              thisAmount + = (each.getDaysRented () - 2) * 1.5;
```

```
                    rotura;
                case Movie.NEW_RELEASE:
                    thisAmount + = each.getDaysRented () * 3;
                    rotura;
                película de caso. NIÑOS:
                    thisAmount + = 1.5;
                    if (each.getDaysRented ()> 3)
                        thisAmount + = (each.getDaysRented () - 3) * 1.5;
                    rotura;

            }

            // agrega puntos frecuentes de arrendatario
            nterrenterPoints ++;
            // agregue bonificación por un nuevo alquiler de dos días
            if ((each.getMovie (). getPriceCode () == Movie.NEW_RELEASE)
&&
each.getDaysRented ()> 1) frecuenteRenterPoints ++;

            // muestra las cifras de este alquiler
            resultado + = "\ t" + each.getMovie (). getTitle () + "\ t" +
String.valueOf (thisAmount) + "\ n";
            totalAmount + = thisAmount;

        }
        // agrega líneas de pie de página
        resultado + = "El monto adeudado es" + String.valueOf (totalAmount) +
"\norte";
        resultado + = "Ganaste" + String.valueOf (frecuenteRenterPoints)
+
"puntos frecuentes de arrendamiento";
        resultado de retorno;

    }
```

Comentarios sobre el programa de inicio

¿Cuáles son sus 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 importar. No hay nada malo con un programa *simple* rápido y sucio . Pero si este es un representante fragmento de un sistema más complejo, entonces tengo algunos problemas reales con este programa. Así de largo La rutina de declaración en la clase Cliente hace demasiado. Muchas de las cosas que hace deberían Realmente sea hecho por las otras clases.

Aun así el programa funciona. ¿No es esto solo un juicio estético, una aversión al código feo? Está hasta que queramos cambiar el sistema. Al compilador no le importa si el código es feo o está limpio. Pero cuando cambiamos el sistema, hay un humano involucrado, y a los humanos les importa. Un mal El sistema diseñado es difícil de cambiar. Difícil porque es difícil averiguar dónde están los cambios necesario. Si es difícil averiguar qué cambiar, existe una gran posibilidad de que el programador cometerá un error e introducirá errores.

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

imposible reutilizar ninguno 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 declaración. Ahora, por supuesto, esto no es demasiado oneroso. Simplemente puede copiar el método de declaración y

haz los cambios que necesites.

Pero, ¿qué sucede cuando cambian las reglas de carga? Tienes que arreglar tanto la `declaración` como `htmlStatement` y asegúrese de que las correcciones sean consistentes. El problema con copiar y pegar el código viene cuando tienes que cambiarlo más tarde. Si está escribiendo un programa que no espera para cambiar, luego cortar y pegar está bien. Si el programa tiene una larga vida y es probable que cambie, entonces corte y pegar es una amenaza.

Esto me lleva a un segundo cambio. Los usuarios desean realizar cambios en la forma en que clasifican películas, pero aún no han decidido el cambio que van a hacer. Tienen un numero de cambios en la mente. Estos cambios afectarán tanto la forma en que se cobra a los inquilinos por las películas como la forma en que se calculan los puntos de arrendatario frecuentes. Como desarrollador experimentado, está seguro de que cualquiera sea el esquema que los usuarios propongan, la única garantía que tendrán es que lo harán cámbielo nuevamente dentro de seis meses.

El método de declaración es donde los cambios tienen que hacerse para lidiar con los cambios en clasificación y reglas de carga. Sin embargo, si copiamos la declaración a una declaración HTML, nosotros necesita asegurarse de que cualquier cambio sea completamente consistente. Además, a medida que crecen las reglas la complejidad será más difícil de averiguar dónde hacer los cambios y más difícil de hacer ellos sin cometer un error.

Es posible que sienta la tentación de realizar la menor cantidad posible de cambios en el programa; después de todo, funciona bien. Recuerde el viejo adagio de ingeniería: "si no está roto, no lo arregles". El programa puede no ser roto, pero duele. Le está haciendo la vida más difícil porque le resulta difícil hacer Cambios que tus usuarios quieren. Aquí es donde entra en juego la refactorización.

Propina

Cuando encuentre que tiene que agregar una función a un programa, y el código del programa no es estructurado de manera conveniente para agregar la función, primero refactorice el programa para hacerlo fácil de agregar la función, luego agregue la función.

El primer paso en la refactorización

Cada vez que realizo una refactorización, el primer paso es siempre el mismo. Necesito construir 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 lo tanto, necesito pruebas sólidas.

Debido a que el resultado de la declaración produce una cadena, creo algunos clientes, le doy a cada cliente un pocos alquileres de varios tipos de películas y generar cadenas de declaración. Entonces hago una cuerda comparación entre la nueva cadena y algunas cadenas de referencia que he verificado manualmente. lo puse arriba todas estas pruebas para que pueda ejecutarlas desde un comando Java en la línea de comandos. Los exámenes Tardo solo unos segundos en ejecutarse y, como verá, los ejecuto a menudo.

Una parte importante de las pruebas es la forma en que informan sus resultados. O bien dicen "OK", lo que significa que todas las cadenas son idénticas a las cadenas de referencia, o imprimen una lista de fallas: líneas que resultó diferente. Las pruebas son, por lo tanto, autoverificables. Es vital realizar pruebas de autocomprobación. Si no, terminas pasando el tiempo comprobando algunos números de la prueba contra algunos números de una almohadilla de escritorio, y eso te ralentiza.

Página 18

A medida que realizamos la refactorización, nos apoyaremos en las pruebas. Voy a confiar en las pruebas para decirme si presento un error. Es esencial para la refactorización que tenga buenas pruebas. Merece la pena dedicar tiempo a crear las pruebas, porque las pruebas le brindan la seguridad que necesita para cambiar El programa más tarde. Esta es una parte tan importante de la refactorización que entro más en detalle en las pruebas

en el Capítulo 4 .

Propina

Antes de comenzar a refactorizar, verifique que tenga un conjunto sólido de pruebas. Estas pruebas debe autoverificarse.

Descomponiendo y redistribuyendo el método de declaración

El primer objetivo obvio de mi atención es el método de declaración demasiado largo. Cuando miro un largo método así, estoy buscando descomponer el método en piezas más pequeñas. Piezas más pequeñas de El código tiende a hacer las cosas más manejables. Son más fáciles de trabajar y moverse.

La primera fase de las refactorizaciones en este capítulo muestra cómo dividir el método largo y mover las piezas a mejores clases. Mi objetivo es facilitar la escritura de un método de declaración HTML con mucho menos duplicación de código.

Mi primer paso es encontrar un grupo lógico de código y usar el Método de extracción . Una pieza obvia aquí es La declaración de cambio. Parece que sería una buena porción extraerlo en su propio método.

Cuando extraigo un método, como en cualquier refactorización, necesito saber qué puede salir mal. Si hago el extracción mal, podría introducir un error en el programa. Entonces, antes de hacer la refactorización, necesito averiguar cómo hacerlo de manera segura. He hecho esta refactorización varias veces antes, así que he escrito Los pasos seguros en el catálogo.

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

Las siguientes dos páginas muestran el código antes y después de refactorizar. 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 El nuevo código que no creo que sea inmediatamente obvio está en negrita. Mientras sigo con esto capítulo, continuaré con esta convención izquierda-derecha.

```
declaración de cadena pública () {
     double totalAmount = 0;
     int frecuenteRenterPoints = 0;
     Enumeración alquileres = _rentals.elements ();
     String result = "Registro de alquiler para" + getName () + "\ n";
     while (rentals.hasMoreElements ()) {
          double thisAmount = 0;
          Alquiler de cada = (Alquiler) rentals.nextElement ();

          // determina cantidades para cada línea
          switch (each.getMovie (). getPriceCode ()) {
```

18 años

Página 19

```
caso Movie.REGULAR:
     thisAmount + = 2;
     if (each.getDaysRented ()> 2)
          thisAmount + = (each.getDaysRented () - 2) * 1.5;
     rotura;
```

```
                    case Movie.NEW_RELEASE:
                         thisAmount + = each.getDaysRented () * 3;
                         rotura;
                    película de caso. NIÑOS:
                         thisAmount + = 1.5;
                         if (each.getDaysRented ()> 3)
                              thisAmount + = (each.getDaysRented () - 3) * 1.5;
                         rotura;

                }

               // agrega puntos frecuentes de arrendatario
               nterrenterPoints ++;
               // agregue bonificación por un nuevo alquiler de dos días
               if ((each.getMovie (). getPriceCode () == Movie.NEW_RELEASE)
&& each.getDaysRented ()>
1) frecuenteRenterPoints ++;

               // muestra las cifras de este alquiler
               resultado + = "\ t" + each.getMovie (). getTitle () + "\ t" +
String.valueOf (thisAmount) +
"\norte";
               totalAmount + = thisAmount;

          }
          // agrega líneas de pie de página
          resultado + = "El monto adeudado es" + String.valueOf (totalAmount) +
"\norte";
          resultado + = "Ganaste" + String.valueOf (frecuenteRenterPoints)
+ "inquilino frecuente
puntos";
          resultado de retorno;

     }
declaración de cadena pública () {
        double totalAmount = 0;
        int frecuenteRenterPoints = 0;
        Enumeración alquileres = _rentals.elements ();
        String result = "Registro de alquiler para" + getName () + "\ n";
        while (rentals.hasMoreElements ()) {
             double thisAmount = 0;
             Alquiler de cada = (Alquiler) rentals.nextElement ();

             thisAmount = amountFor (cada uno);

             // agrega puntos frecuentes de arrendatario
             nterrenterPoints ++;
             // agregue bonificación por un nuevo alquiler de dos días
             if ((each.getMovie (). getPriceCode () == Movie.NEW_RELEASE) &&
                  each.getDaysRented ()> 1) frecuenteRenterPoints ++;

             // muestra las cifras de este alquiler
```


Página 20

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

        }
        // agrega líneas de pie de página
```

```
"\norte";resultado + = "El monto adeudado es" + String.valueOf (totalAmount) +
        resultado + = "Ganaste" + String.valueOf (frequencyRenterPoints) +
              "puntos frecuentes de arrendamiento";
        resultado de retorno;

    }
}
Private Int amountFor (Alquiler de cada uno) {
    int thisAmount = 0;
    switch (each.getMovie (). getPriceCode ()) {
         caso Movie.REGULAR:
              thisAmount + = 2;
              if (each.getDaysRented ()> 2)
                   thisAmount + = (each.getDaysRented () - 2) * 1.5;
              rotura;
         case Movie.NEW_RELEASE:
              thisAmount + = each.getDaysRented () * 3;
              rotura;
         película de caso. NIÑOS:
              thisAmount + = 1.5;
              if (each.getDaysRented ()> 3)
                   thisAmount + = (each.getDaysRented () - 3) * 1.5;
              rotura;

    }
    devuelve thisAmount;
}
```

Cada vez que hago un cambio como este, compilo y pruebo. No tuve un buen comienzo: el
las pruebas explotaron. Un par de cifras de la prueba me dieron la respuesta incorrecta. Estaba desconcertado por unos pocos segundos luego me di cuenta de lo que había hecho. Tontamente hice la cantidad del tipo de retorno `para int` en lugar de `doble`:

```
    cantidad doble privada para (Alquiler de cada) {
        double thisAmount = 0;
        switch (each.getMovie (). getPriceCode ()) {
            caso Movie.REGULAR:
                thisAmount + = 2;
                if (each.getDaysRented ()> 2)
                    thisAmount + = (each.getDaysRented () - 2) * 1.5;
               rotura;
            case Movie.NEW_RELEASE:
                thisAmount + = each.getDaysRented () * 3;
                rotura;
            película de caso. NIÑOS:
                thisAmount + = 1.5;
                if (each.getDaysRented ()> 3)
                    thisAmount + = (each.getDaysRented () - 3) * 1.5;
```


Página 21

```
                rotura;
        }
        devuelve thisAmount;
    }
```

Es el tipo de error tonto que suelo cometer, y puede ser un dolor rastrearlo. En este caso Java
convierte dobles a ints sin quejarse pero redondeando alegremente [Java Spec]. Afortunadamente fue

fácil de encontrar en este caso, porque el cambio fue muy pequeño y tuve un buen conjunto de pruebas. Aquí está
La esencia del proceso de refactorización ilustrado por accidente. Debido a que cada cambio es tan pequeño,
Cualquier error es muy fácil de encontrar. No pasas mucho tiempo depurando, incluso si eres tan
descuidado como soy.

Propina

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

Como estoy trabajando en Java, necesito analizar el código para descubrir qué hacer con el local
variables Con una herramienta, sin embargo, esto puede hacerse realmente simple. Tal herramienta existe en
Smalltalk, el navegador de refactorización. Con esta herramienta, la refactorización es muy simple. Solo resalto el
código, seleccione "Método de extracción" de los menús, escriba el nombre de un método y listo. Además,
la herramienta no comete errores tontos como el mío. Estoy deseando una versión de Java!

Ahora que he dividido el método original en fragmentos, puedo trabajar en ellos por separado. Yo no
como algunos de los nombres de variables en `cantidad`, y este es un buen lugar para cambiarlos.

Aquí está el código original:

```
cantidad doble privada para (Alquiler de cada) {
     double thisAmount = 0;
     switch (each.getMovie (). getPriceCode ()) {
          caso Movie.REGULAR:
               thisAmount + = 2;
               if (each.getDaysRented ()> 2)
                    thisAmount + = (each.getDaysRented () - 2) * 1.5;
               rotura;
          case Movie.NEW_RELEASE:
               thisAmount + = each.getDaysRented () * 3;
               rotura;
          película de caso. NIÑOS:
               thisAmount + = 1.5;
               if (each.getDaysRented ()> 3)
                    thisAmount + = (each.getDaysRented () - 3) * 1.5;
               rotura;
     }
     devuelve thisAmount;
}
```

Aquí está el código renombrado:

Página 22

```
cantidad doble privada para (Alquiler aRental ) {
     doble resultado = 0;
     switch ( aRental. getMovie (). getPriceCode ()) {
          caso Movie.REGULAR:
               resultado + = 2;
               if ( aRental. getDaysRented ()> 2)
                    resultado + = (aRental.getDaysRented () - 2) * 1.5;
               rotura;
```

```
            case Movie.NEW_RELEASE:
                resultado + = aRental. getDaysRented () * 3;
                rotura;
            película de caso. NIÑOS:
                resultado + = 1.5;
                if ( aRental. getDaysRented ()> 3)
                    resultado + = ( aRental. getDaysRented () - 3) * 1.5;
                rotura;
        }
        resultado de retorno ;
    }
```

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

¿Vale la pena cambiar el nombre? Absolutamente. Un buen código debe comunicar lo que está haciendo claramente, y los nombres de variables son una clave para borrar el código. Nunca tengas miedo de cambiar los nombres de las cosas a Mejora la claridad. Con buenas herramientas para buscar y reemplazar, generalmente no es difícil. Mecanografía fuerte y las pruebas resaltarán todo lo que te pierdas. Recuerda

Propina

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 solo cuando estoy leyendo algún código De esa manera, a medida que adquiero comprensión sobre el programa, incorporo esa comprensión en el código para más tarde, así que no olvido lo que aprendí.

Mover el cálculo del importe

Cuando miro `cantidad`, puedo ver que usa información del alquiler, pero no usa Información del cliente.

```
clase Cliente ...
cantidad doble privada para (Alquiler aRental) {
    doble resultado = 0;
    switch (aRental.getMovie (). getPriceCode ()) {
        caso Movie.REGULAR:
            resultado + = 2;
            if (aRental.getDaysRented ()> 2)
                resultado + = (aRental.getDaysRented () - 2) * 1.5;
```



```
            rotura;
        case Movie.NEW_RELEASE:
            resultado + = aRental.getDaysRented () * 3;
            rotura;
        película de caso. NIÑOS:
            resultado + = 1.5;
            if (aRental.getDaysRented ()> 3)
                resultado + = (aRental.getDaysRented () - 3) * 1.5;
            rotura;
    }
```

```
        resultado de retorno;
    }
```

Esto levanta inmediatamente mis sospechas de que el método está en el objeto equivocado. En la mayoría de los casos, un El método debe estar en el objeto cuyos datos utiliza, por lo tanto, el método debe moverse al alquiler. Para hacer esto, uso el método Move . Con esto, primero copie el código al alquiler, ajústelo a cabe en su nuevo hogar y compila de la siguiente manera:

```
Alquiler de clase ...
   doble getCharge () {
        doble resultado = 0;
        switch (getMovie (). getPriceCode ()) {
             caso Movie.REGULAR:
                  resultado + = 2;
                  if (getDaysRented ()> 2)
                       resultado + = (getDaysRented () - 2) * 1.5;
                  rotura;
             case Movie.NEW_RELEASE:
                  resultado + = getDaysRented () * 3;
                  rotura;
             película de caso. NIÑOS:
                  resultado + = 1.5;
                  if (getDaysRented ()> 3)
                       resultado + = (getDaysRented () - 3) * 1.5;
                  rotura;
        }
        resultado de retorno;
   }
```

En este caso, adaptarse a su nuevo hogar significa eliminar el parámetro. También cambié el nombre del método como hice el movimiento

Ahora puedo probar para ver si este método funciona. Para hacer esto, reemplazo el cuerpo de `Customer.amountFor` para delegar al nuevo método.

```
clase Cliente ...
   cantidad doble privada para (Alquiler aRental) {
        devuelve aRental.getCharge ();
  }
```

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

Página 24

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

```
  clase Cliente ...
       declaración de cadena pública () {
            double totalAmount = 0;
            int frecuenteRenterPoints = 0;
            Enumeración alquileres = _rentals.elements ();
            String result = "Registro de alquiler para" + getName () + "\ n";
            while (rentals.hasMoreElements ()) {
```

```
                double thisAmount = 0;
                Alquiler de cada = (Alquiler) rentals.nextElement ();

                thisAmount = amountFor (cada uno);

                // agrega puntos frecuentes de arrendatario
                nterrenterPoints ++;
                // agregue bonificación por un nuevo alquiler de dos días
                if ((each.getMovie (). getPriceCode () == Movie.NEW_RELEASE)
&&
                    each.getDaysRented ()> 1) frecuenteRenterPoints ++;

                // muestra las cifras de este alquiler
                resultado + = "\ t" + each.getMovie (). getTitle () + "\ t" +
                     String.valueOf (thisAmount) + "\ n";
                totalAmount + = thisAmount;

            }
            // agrega líneas de pie de página
            resultado + = "El monto adeudado es" + String.valueOf (totalAmount) +
"\norte";
            resultado + = "Ganaste" +
String.valueOf (frequencyRenterPoints) +
                  "puntos frecuentes de arrendamiento";
            resultado de retorno;

        }
```

En este caso, este paso es fácil porque acabamos de crear el método y está en un solo lugar. En en general, sin embargo, debe hacer un "hallazgo" en todas las clases que podrían estar usando ese método:

```
  cliente de clase
       declaración de cadena pública () {
            double totalAmount = 0;
            int frecuenteRenterPoints = 0;
            Enumeración alquileres = _rentals.elements ();
            String result = "Registro de alquiler para" + getName () + "\ n";
            while (rentals.hasMoreElements ()) {
                 double thisAmount = 0;
                 Alquiler de cada = (Alquiler) rentals.nextElement ();

                 thisAmount = each.getCharge ();

                 // agrega puntos frecuentes de arrendatario
```


Página 25

```
                 nterrenterPoints ++;
                 // agregue bonificación por un nuevo alquiler de dos días
                 if ((each.getMovie (). getPriceCode () == Movie.NEW_RELEASE)
&&
                     each.getDaysRented ()> 1) frecuenteRenterPoints ++;

                 // muestra las cifras de este alquiler
                 resultado + = "\ t" + each.getMovie (). getTitle () + "\ t" +
                      String.valueOf (thisAmount) + "\ n";
                 totalAmount + = thisAmount;

            }
            // agrega líneas de pie de página
```

```
            resultado + = "El monto adeudado es" + String.valueOf (totalAmount) +
"\norte";
            resultado + = "Ganaste" +
String.valueOf (frequencyRenterPoints) +
                  "puntos frecuentes de arrendamiento";
            resultado de retorno;
        }
```

Cuando hice el cambio (Figura 1.3), lo siguiente es eliminar el método anterior. los
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 anterior para delegarlo al nuevo. Esto es útil si es público
método y no quiero cambiar la interfaz de la otra clase.

Ciertamente hay algo más que me gustaría hacerle a `Rental.getCharge` pero lo dejaré para el
momento y regreso a la declaración del `cliente`.

```
    declaración de cadena pública () {
         double totalAmount = 0;
         int frecuenteRenterPoints = 0;
         Enumeración alquileres = _rentals.elements ();
         String result = "Registro de alquiler para" + getName () + "\ n";
         while (rentals.hasMoreElements ()) {
              double thisAmount = 0;
              Alquiler de cada = (Alquiler) rentals.nextElement ();

              thisAmount = each.getCharge ();

              // agrega puntos frecuentes de arrendatario
              nterrenterPoints ++;
              // agregue bonificación por un nuevo alquiler de dos días
```



```
              if ((each.getMovie (). getPriceCode () == Movie.NEW_RELEASE)
&&
                  each.getDaysRented ()> 1) frecuenteRenterPoints ++;

              // muestra las cifras de este alquiler
              resultado + = "\ t" + each.getMovie (). getTitle () + "\ t" +
                   String.valueOf (thisAmount) + "\ n";
              totalAmount + = thisAmount;

         }
         // agrega líneas de pie de página
         resultado + = "El monto adeudado es" + String.valueOf (totalAmount) +
"\norte";
         resultado + = "Ganaste" + String.valueOf (frecuenteRenterPoints)
```

```
+
                "puntos frecuentes de arrendamiento";
            resultado de retorno;

     }
```

Lo siguiente que me sorprende es que `thisAmount` ahora es redundante. Se establece en el resultado de `cada carga` y no cambia después. Por lo tanto, puedo eliminar `thisAmount` usando Reemplazar Temp con consulta :

```
      declaración de cadena pública () {
           double totalAmount = 0;
           int frecuenteRenterPoints = 0;
           Enumeración alquileres = _rentals.elements ();
           String result = "Registro de alquiler para" + getName () + "\ n";
           while (rentals.hasMoreElements ()) {
                Alquiler de cada = (Alquiler) rentals.nextElement ();

                // agrega puntos frecuentes de arrendatario
                nterrenterPoints ++;
                // agregue bonificación por un nuevo alquiler de dos días
                if ((each.getMovie (). getPriceCode () == Movie.NEW_RELEASE)
&&
                    each.getDaysRented ()> 1) frecuenteRenterPoints ++;
                // muestra las cifras de este alquiler
                resultado + = "\ t" + each.getMovie (). getTitle () + "\ t" +
String.valueOf
                    ( each.getCharge () ) + "\ n";
                totalAmount + = each.getCharge ();

           }
           // agrega líneas de pie de página
           resultado + = "El monto adeudado es" + String.valueOf (totalAmount) +
"\norte";
           resultado + = "Ganaste" + String.valueOf (frecuenteRenterPoints)
                 + "puntos frecuentes de alquiler";
           resultado de retorno;

      }

 }
```


Una vez que hice ese cambio, lo compilé y probé para asegurarme de que no había roto nada.

Me gusta deshacerme de variables temporales como esta tanto como sea posible. Temps son a menudo un
El problema es que hacen que se pasen muchos parámetros cuando no es necesario.
Puede perder fácilmente la noción de para qué sirven. Son particularmente insidiosos en mucho tiempo
métodos Por supuesto, hay un precio de rendimiento a pagar; aquí el cargo ahora se calcula dos veces.
Pero es fácil optimizar eso en la clase de alquiler, y puede optimizar mucho más efectivamente
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 arrendatario frecuente

El siguiente paso es hacer algo similar para los puntos frecuentes de alquiler. Las reglas varían con la cinta,
aunque hay menos variación que con la carga. Parece razonable poner la responsabilidad

En el alquiler. Primero, debemos usar el Método de extracción en la parte del código de puntos de arrendatario frecuente (en negrita):

```
     declaración de cadena pública () {
          double totalAmount = 0;
          int frecuenteRenterPoints = 0;
          Enumeración alquileres = _rentals.elements ();
          String result = "Registro de alquiler para" + getName () + "\ n";
          while (rentals.hasMoreElements ()) {
               Alquiler de cada = (Alquiler) rentals.nextElement ();

              // agrega puntos frecuentes de arrendatario
              nterrenterPoints ++;
              // agregue bonificación por un nuevo alquiler de dos días
              if ((each.getMovie (). getPriceCode () == Movie.NEW_RELEASE)
                  && each.getDaysRented ()> 1) frecuenteRenterPoints ++;

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

          }
          // agrega líneas de pie de página
          resultado + = "El monto adeudado es" + String.valueOf (totalAmount) +
"\norte";
          resultado + = "Ganaste" + String.valueOf (frecuenteRenterPoints)
                + "puntos frecuentes de alquiler";
          resultado de retorno;

     }
}
```

Una vez más nos fijamos en el uso de variables de ámbito local. De nuevo, `cada uno` se usa y se puede pasar como un parámetro La otra temperatura utilizada es `frecuenteRenterPoints.` En este caso `frecuenteRenterPoints` tiene un valor de antemano. El cuerpo del método extraído. sin embargo, no lee el valor, por lo que no necesitamos pasarlo como parámetro siempre que usemos una tarea anexa.

Página 28

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

```
clase Cliente ...
     declaración de cadena pública () {
          double totalAmount = 0;
          int frecuenteRenterPoints = 0;
          Enumeración alquileres = _rentals.elements ();
          String result = "Registro de alquiler para" + getName () + "\ n";
          while (rentals.hasMoreElements ()) {
               Alquiler de cada = (Alquiler) rentals.nextElement ();
              frecuentesRenterPoints + = each.getFrequentRenterPoints ();

               // muestra las cifras de este alquiler
               resultado + = "\ t" + each.getMovie (). getTitle () + "\ t" +
```

```
                String.valueOf (each.getCharge ()) + "\ n";
            totalAmount + = each.getCharge ();
        }

        // agrega líneas de pie de página
        resultado + = "El monto adeudado es" + String.valueOf (totalAmount) +
"\norte";
        resultado + = "Ganaste" + String.valueOf (frecuenteRenterPoints)
+
              "puntos frecuentes de arrendamiento";
        resultado de retorno;
    }

Alquiler de clase ...
  int getFrequentRenterPoints () {
       if ((getMovie (). getPriceCode () == Movie.NEW_RELEASE) &&
getDaysRented ()> 1)
           retorno 2;
       más
           retorno 1;
  }
```

Resumiré los cambios que acabo de hacer con un lenguaje de modelado unificado anterior y posterior (UML) diagramas (Figuras 1.4 a 1.7). De nuevo, los diagramas de la izquierda están antes del cambio; los de la derecha están después del cambio.

Figura 1.4. Diagrama de clases antes de la extracción y movimiento de los puntos frecuentes de alquiler cálculo

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

Figura 1.6. Diagrama de clases después de la extracción y movimiento de los puntos frecuentes de alquiler cálculo

Figura 1.7. Diagrama de secuencia antes de la extracción y movimiento del inquilino frecuente cálculo de puntos

Página 30

Eliminar templos

Como sugerí antes, las variables temporales pueden ser un problema. Son útiles solo dentro de su rutina propia, y así fomentan rutinas largas y complejas. En este caso tenemos dos variables temporales, que se utilizan para obtener un total de los alquileres adjuntos a cliente. Tanto las versiones ASCII como HTML requieren estos totales. Me gusta usar Reemplazar Temp con consulta para reemplazar `totalAmount` y `frecuenteRentalPoints` con métodos de consulta.

Las consultas son accesibles a cualquier método en la clase y, por lo tanto, fomentan un diseño más limpio sin métodos largos y complejos:

```
 clase Cliente ...
      declaración de cadena pública () {
           double totalAmount = 0;
           int frecuenteRenterPoints = 0;
           Enumeración alquileres = _rentals.elements ();
           String result = "Registro de alquiler para" + getName () + "\ n";
           while (rentals.hasMoreElements ()) {
                Alquiler de cada = (Alquiler) rentals.nextElement ();
                frecuentesRenterPoints + = each.getFrequentRenterPoints ();

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

           // agrega líneas de pie de página
           resultado + = "El monto adeudado es" + String.valueOf (totalAmount) +
"\norte";
           resultado + = "Ganaste" + String.valueOf (frecuenteRenterPoints)
+
             "puntos frecuentes de arrendamiento";
           resultado de retorno;
```



```
      }
```

Comencé reemplazando `totalAmount` con un método de `carga` para el cliente:

```
 clase Cliente ...

      declaración de cadena pública () {
           int frecuenteRenterPoints = 0;
           Enumeración alquileres = _rentals.elements ();
           String result = "Registro de alquiler para" + getName () + "\ n";
           while (rentals.hasMoreElements ()) {
                Alquiler de cada = (Alquiler) rentals.nextElement ();
                frecuentesRenterPoints + = each.getFrequentRenterPoints ();

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

```
            // agrega líneas de pie de página
            resultado + = "El monto adeudado es" +
String.valueOf ( getTotalCharge () ) + "\ n";
            resultado + = "Ganaste" + String.valueOf (frecuenteRenterPoints)
+
              "puntos frecuentes de arrendamiento";
            resultado de retorno;
      }

     privado doble getTotalCharge () {
            doble resultado = 0;
            Enumeración alquileres = _rentals.elements ();
            while (rentals.hasMoreElements ()) {
                  Alquiler de cada = (Alquiler) rentals.nextElement ();
                  resultado + = each.getCharge ();
            }
            resultado de retorno;
      }
```

Este no es el caso más simple de Reemplazar temp con Query `totalAmount` se asignó a dentro del bucle, así que tengo que copiar el bucle en el método de consulta.

Después de compilar y probar esa refactorización, hice lo mismo para `frecuenteRenterPoints`:

```
 clase Cliente ...
      declaración de cadena pública () {
            int frecuenteRenterPoints = 0;
            Enumeración alquileres = _rentals.elements ();
            String result = "Registro de alquiler para" + getName () + "\ n";
            while (rentals.hasMoreElements ()) {
                  Alquiler de cada = (Alquiler) rentals.nextElement ();
                  frecuentesRenterPoints + = each.getFrequentRenterPoints ();

                  // muestra las cifras de este alquiler
```


Página 32

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

            // agrega líneas de pie de página
            resultado + = "El monto adeudado es" +
String.valueOf (getTotalCharge ()) + "\ n";
            resultado + = "Ganaste" + String.valueOf (frecuenteRenterPoints)
+
              "puntos frecuentes de arrendamiento";
            resultado de retorno;
      }
      declaración de cadena pública () {
            Enumeración alquileres = _rentals.elements ();
            String result = "Registro de alquiler para" + getName () + "\ n";
            while (rentals.hasMoreElements ()) {
                  Alquiler de cada = (Alquiler) rentals.nextElement ();

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

```
            }

            // agrega líneas de pie de página
            resultado + = "El monto adeudado es" +
String.valueOf (getTotalCharge ()) + "\ n";
            resultado + = "Ganaste" +
String.valueOf ( getTotalFrequentRenterPoints () ) +
                          "puntos frecuentes de arrendamiento";
          resultado de retorno;
      }

       private int getTotalFrequentRenterPoints () {
             int resultado = 0;
            Enumeración alquileres = _rentals.elements ();
            while (rentals.hasMoreElements ()) {
                 Alquiler de cada = (Alquiler) rentals.nextElement ();
                 resultado + = each.getFrequentRenterPoints ();
            }
            resultado de retorno;
       }
```

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

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.

34

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

La otra preocupación con esta refactorización radica en el rendimiento. El antiguo código ejecutó el "while" bucle una vez, el nuevo código lo ejecuta tres veces. Un bucle while que lleva mucho tiempo puede afectar actuación. Muchos programadores no harían esta refactorización simplemente por esta razón. Pero nota las palabras *si* y *poder.* Hasta que perfile no puedo decir cuánto tiempo se necesita para que el ciclo calcular o si el ciclo se llama con la frecuencia suficiente para afectar el rendimiento general de sistema. No se preocupe por esto mientras refactoriza. Cuando optimices tendrás que preocuparte por pero estará en una posición mucho mejor para hacer algo al respecto y tendrá más opciones para optimizar de manera efectiva (vea la discusión en la página 69).

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

Puede ver la diferencia inmediatamente con `htmlStatement`. Ahora estoy en el punto donde yo me quito el sombrero refactorizador y me pongo el sombrero de la función de agregar. Puedo escribir `htmlStatement` como sigue y agrega las pruebas apropiadas:

```
   public String htmlStatement () {
        Enumeración alquileres = _rentals.elements ();
        String result = "<H1> Alquileres para <EM>" + getName () + "</EM> </
H1> <P> \ n ";
        while (rentals.hasMoreElements ()) {
             Alquiler de cada = (Alquiler) rentals.nextElement ();
```


Página 35

```
             // muestra las cifras de cada alquiler
             resultado + = each.getMovie (). getTitle () + ":" +
                              String.valueOf (each.getCharge ()) + "<BR> \ n";
        }
        // agrega líneas de pie de página
        resultado + = "<P> Debe <EM>" + String.valueOf (getTotalCharge ()) +
"</EM> <P> \ n";
        resultado + = "En este alquiler ganó <EM>" +
             String.valueOf (getTotalFrequentRenterPoints ()) +
             "</EM> puntos frecuentes de alquiler <P>";
        resultado de retorno;
   }
```

Al extraer los cálculos, puedo crear el método `htmlStatement` y reutilizar todos los código de cálculo que estaba en el método de declaración original. No copié y pegué, así que si el cambio de reglas de cálculo Solo tengo un lugar en el código para ir. Cualquier otro tipo de declaración Será realmente rápido y fácil de preparar. La refactorización no tardó mucho. Pasé la mayor parte del tiempo averiguar lo que hizo el código, y habría tenido que hacer eso de todos modos.

Parte del código se copia de la versión ASCII, principalmente debido a la configuración del bucle. Promover, adicional refactorizar podría limpiar eso. Los métodos de extracción para encabezado, pie de página y línea de detalle son uno ruta que podría tomar. Puede ver cómo hacerlo en el ejemplo del Método de plantilla de formulario . Pero ahora los usuarios están clamando de nuevo. Se están preparando para cambiar la clasificación de la peliculas en la tienda. Todavía no está claro qué cambios quieren hacer, pero parece nuevo

Se introducirán clasificaciones y las existentes podrían modificarse. Los cargos y
Se deben decidir las asignaciones frecuentes de puntos de arrendatario para estas clasificaciones. En el momento,
hacer este tipo de cambios es incómodo. Tengo que entrar en el cargo y en el punto de alquiler frecuente
métodos y alterar el código condicional para realizar cambios en las clasificaciones de películas. De vuelta con el
sombrero refactorizador.

Sustitución de la lógica condicional en el código de precios 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 usar una declaración de cambio, debe estar en sus propios datos, no
en el de otra persona.

```
Alquiler de clase ...
   doble getCharge () {
        doble resultado = 0;
        switch (getMovie (). getPriceCode ()) {
             caso Movie.REGULAR:
                  resultado + = 2;
                  if (getDaysRented ()> 2)
                       resultado + = (getDaysRented () - 2) * 1.5;
                  rotura;
             case Movie.NEW_RELEASE:
                  resultado + = getDaysRented () * 3;
                  rotura;
             película de caso. NIÑOS:
                  resultado + = 1.5;
                  if (getDaysRented ()> 3)
                       resultado + = (getDaysRented () - 3) * 1.5;
                  rotura;
        }
```



```
        resultado de retorno;
  }
```

Esto implica que `getCharge` debería pasar a la película:

```
clase de cine ...
   double getCharge (int daysRented) {
        doble resultado = 0;
        switch (getPriceCode ()) {
             caso Movie.REGULAR:
                  resultado + = 2;
                  if (días alquilados> 2)
                       resultado + = (días alquilados - 2) * 1.5;
                  rotura;
             case Movie.NEW_RELEASE:
                  resultado + = días alquilados * 3;
                  rotura;
             película de caso. NIÑOS:
                  resultado + = 1.5;
                  if (días alquilados> 3)
                       resultado + = (días alquilados - 3) * 1.5;
                  rotura;
        }
        resultado de retorno;
   }
```

Para que esto funcione, tuve que pasar la duración del alquiler, que por supuesto son datos del alquiler.
El método utiliza efectivamente 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. Escriba la información en general
tiende a ser más volátil. Si cambio el tipo de película, quiero el menor efecto dominó, así que prefiero
Calcule la carga dentro de la película.

`Compilé` el método en una película y luego cambié `getCharge` en el alquiler para usar el nuevo
Método (Figuras 1.12 y 1.13):

Figura 1.12. Diagrama de clases antes de pasar los métodos a la película

Page 37

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

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

Una vez que haya movido el método `getCharge` , haré lo mismo con el punto de inquilino frecuente

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

```
 Alquiler de clase ...
  int getFrequentRenterPoints () {
       if ((getMovie (). getPriceCode () == Movie.NEW_RELEASE) &&
getDaysRented ()> 1)
           retorno 2;
       más
           retorno 1;
  }

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


  int getFrequentRenterPoints (int daysRented) {
       if ((getPriceCode () == Movie.NEW_RELEASE) && daysRented> 1)
            retorno 2;
       más
           retorno 1;
 }
```

Por fin ... herencia

Tenemos varios tipos de películas que tienen diferentes formas de responder la misma pregunta. Esta
Suena como un trabajo para subclases. Podemos tener tres subclases de películas, cada una de las cuales puede
tiene su propia versión de carga (Figura 1.14).

Figura 1.14. Usar herencia en película

Esto me permite reemplazar la declaración de cambio mediante el uso de polimorfismo. Lamentablemente tiene un ligero

fallar no funciona. Una película puede cambiar su clasificación durante su vida útil. Un objeto no puede cambiar su clase durante su vida útil. Hay una solución, sin embargo, el patrón de Estado [Banda de los Cuatro]. Con el patrón State, las clases se ven como en la Figura 1.15 .

Figura 1.15. Usando el patrón de estado en la película

Al agregar la indirección podemos hacer la subclasificación del objeto del código de precio y cambiar el precio siempre que lo necesitemos.

Si está familiarizado con los patrones de la Banda de los Cuatro, puede preguntarse: "¿Es este un estado o es un estrategia? "¿La clase de precio representa un algoritmo para calcular el precio (en cuyo caso I prefiere llamarlo Pricer o PricingStrategy), o representa un estado de la película (*Star Trek X* es

Página 39

Un nuevo lanzamiento). En esta etapa, la elección del patrón (y el nombre) refleja cómo quieres pensar Sobre la estructura. Por el momento estoy pensando en esto como un estado de la película. Si luego decido un La estrategia comunica mejor mi intención, voy a refactorizar para hacer esto cambiando los nombres.

Para presentar el patrón de estado, usaré tres refactorizaciones. Primero moveré el comportamiento del código de tipo en el patrón de estado con Reemplazar código de tipo con Estado / Estrategia . Entonces puedo usar Move Método para mover la declaración de cambio a la clase de precio. Finalmente usaré Reemplazar condicional con polimorfismo para eliminar la declaración de cambio.

Comienzo con Reemplazar código de tipo con Estado / Estrategia . Este primer paso es usar Self Encapsule el campo en el código de tipo para garantizar que todos los usos del código de tipo pasen por y métodos de configuración. Como la mayoría del código proviene de otras clases, la mayoría de los métodos ya usa el método de obtención. Sin embargo, los constructores acceden al código de precio:

```
clase de cine ...
      Public Movie (String name, int priceCode) {
         _name = nombre;
         _priceCode = priceCode;
      }
```

Puedo usar el método de configuración en su lugar.

```
clase de cine
     Public Movie (String name, int priceCode) {
        _name = nombre;
        setPriceCode (priceCode);
```

```
    }
```

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

```
clase abstracta precio {
   resumen int getPriceCode ();
}
clase ChildrensPrice extiende el precio {
   int getPriceCode () {
        película de regreso. NIÑOS;
  }
}
clase NewReleasePrice extiende el precio {
   int getPriceCode () {
        volver Movie.NEW_RELEASE;
  }
}
clase RegularPrice extiende el precio {
   int getPriceCode () {
        volver Movie.REGULAR;
  }
}
```


Page 40

Puedo compilar las nuevas clases en este momento.

Ahora necesito cambiar los accesos de la película por el código de precio para usar la nueva clase:

```
   public int getPriceCode () {
        return _priceCode;
   }
   public setPriceCode (int arg) {
        _priceCode = arg;
   }
   private int _priceCode;
```

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

```
clase de cine ...
   public int getPriceCode () {
       return _price.getPriceCode ();
   }
   public void setPriceCode (int arg) {
        interruptor (arg) {
        caso REGULAR:
            _price = new RegularPrice ();
            rotura;
            caso NIÑOS:
            _price = new ChildrensPrice ();
            rotura;
            caso NEW_RELEASE:
            _price = new NewReleasePrice ();
```

```
            rotura;
        defecto:
            lanzar una nueva 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 el Método Move para `getCharge`:

```
clase de cine ...
  double getCharge (int daysRented) {
       doble resultado = 0;
       switch (getPriceCode ()) {
            caso Movie.REGULAR:
                 resultado + = 2;
                 if (días alquilados> 2)
                      resultado + = (días alquilados - 2) * 1.5;
                 rotura;
            case Movie.NEW_RELEASE:
                 resultado + = días alquilados * 3;
                 rotura;
            película de caso. NIÑOS:
```



```
                 resultado + = 1.5;
                 if (días alquilados> 3)
                      resultado + = (días alquilados - 3) * 1.5;
                 rotura;
       }
       resultado de retorno;
  }
```

Es simple de mover:

```
clase de cine ...
  double getCharge (int daysRented) {
       return _price.getCharge (daysRented);
  }

clase Precio ...
  double getCharge (int daysRented) {
       doble resultado = 0;
       switch (getPriceCode ()) {
            caso Movie.REGULAR:
                 resultado + = 2;
                 if (días alquilados> 2)
                      resultado + = (días alquilados - 2) * 1.5;
                 rotura;
            case Movie.NEW_RELEASE:
                 resultado + = días alquilados * 3;
                 rotura;
            película de caso. NIÑOS:
                 resultado + = 1.5;
                 if (días alquilados> 3)
                      resultado + = (días alquilados - 3) * 1.5;
```

```
                rotura;
        }
        resultado de retorno;
    }
```

Una vez que se mueve, puedo comenzar a usar Reemplazar condicional con polimorfismo :

```
clase Precio ...
  double getCharge (int daysRented) {
       doble resultado = 0;
       switch (getPriceCode ()) {
            caso Movie.REGULAR:
                 resultado + = 2;
                 if (días alquilados> 2)
                      resultado + = (días alquilados - 2) * 1.5;
                 rotura;
            case Movie.NEW_RELEASE:
                 resultado + = días alquilados * 3;
                 rotura;
            película de caso. NIÑOS:
                 resultado + = 1.5;
                 if (días alquilados> 3)
                      resultado + = (días alquilados - 3) * 1.5;
```


Page 42

```
                rotura;
       }
       resultado de retorno;
  }
```

Lo hago tomando una parte de la declaración del caso a la vez y creando un método de anulación. yo Comience con `RegularPrice`:

```
clase RegularPrice ...
     double getCharge (int daysRented) {
          doble resultado = 2;
          if (días alquilados> 2)
               resultado + = (días alquilados - 2) * 1.5;
          resultado de retorno;
     }
```

Esto anula la declaración del caso principal, que simplemente dejo como está. Compilo y pruebo esto luego tome el siguiente tramo, compile y pruebe. (Para asegurarme de que estoy ejecutando el código de la subclase, yo quisiera lanzar un error deliberado y ejecutarlo para garantizar que las pruebas exploten. No es que sea paranoico o cualquier cosa.)

```
clase ChildrensPrice
     double getCharge (int daysRented) {
          doble resultado = 1.5;
          if (días alquilados> 3)
               resultado + = (días alquilados - 3) * 1.5;
          resultado de retorno;
     }

clase NewReleasePrice ...
```

```
    double getCharge (int daysRented) {
        días de regreso Alquilado * 3;
    }
```

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

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

Ahora puedo hacer el mismo procedimiento con `getFrequentRenterPoints`:

```
Alquiler de clase ...
  int getFrequentRenterPoints (int daysRented) {
       if ((getPriceCode () == Movie.NEW_RELEASE) && daysRented> 1)
           retorno 2;
       más
           retorno 1;
  }
```

Primero muevo el método a Precio:

Page 43

```
Película de clase ...
  int getFrequentRenterPoints (int daysRented) {
          return _price.getFrequentRenterPoints (daysRented);
  }
Precio de clase ...
  int getFrequentRenterPoints (int daysRented) {
       if ((getPriceCode () == Movie.NEW_RELEASE) && daysRented> 1)
            retorno 2;
       más
           retorno 1;
  }
```

En este caso, sin embargo, no hago que el método de la superclase sea abstracto. En cambio, creo una anulación
Método para nuevos lanzamientos y deje un método definido (por defecto) en la superclase:

```
Clase NewReleasePrice
  int getFrequentRenterPoints (int daysRented) {
          retorno (días alquilados> 1)? 2: 1;
  }

Precio de clase ...
    int getFrequentRenterPoints (int daysRented) {
         retorno 1;
    }
```

Poner en el patrón de estado fue todo un esfuerzo. ¿Valió la pena? La ganancia es que si cambio alguno de
comportamiento del precio, agregar nuevos precios o agregar un comportamiento adicional dependiente del precio, el cambio será mucho
Más fácil de hacer. El resto de la aplicación no sabe sobre el uso del patrón de estado. por
La pequeña cantidad de comportamiento que tengo actualmente no es gran cosa. En un sistema más complejo con un
Una docena de métodos que dependen del precio, esto haría una gran diferencia. Todos estos cambios fueron
pasos pequeños. Parece lento escribirlo de esta manera, pero ni una sola vez tuve que abrir el depurador, así que
El proceso realmente fluyó bastante rápido. Me tomó mucho más tiempo escribir esta sección del libro.

de lo que hizo para cambiar el código.

Ahora he completado la segunda refactorización importante. Va a ser mucho más fácil cambiar el estructura de clasificación de películas, y para alterar las reglas de cobro y el punto de alquiler frecuente sistema. Las figuras 1.16 y 1.17 muestran cómo funciona el patrón de estado con la información de precios.

Figura 1.16. Interacciones usando el patrón de estado

Figura 1.17. Diagrama de clases después de la adición del patrón de estado

Pensamientos finales

Este es un ejemplo simple, pero espero que te dé la sensación de cómo es la refactorización. he usado varias refactorizaciones, incluido el método de extracción , el método de movimiento y el reemplazo condicional con el polimorfismo . Todo esto lleva a responsabilidades mejor distribuidas y un código que es más fácil mantener. Se ve bastante diferente del código de estilo de procedimiento, y eso requiere algo de tiempo Acostumbrado a. Pero una vez que está acostumbrado, es difícil volver a los programas de procedimientos.

Página 45

La lección más importante de este ejemplo es el ritmo de refactorización: prueba, pequeño cambio, prueba, pequeño cambio, prueba, pequeño cambio. Es ese ritmo el que permite que la refactorización se mueva rápidamente y sin peligro.

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

Página 46

Capítulo 2. Principios en la refactorización

El ejemplo anterior debería haberle dado una buena idea de lo que se trata la refactorización. Ahora
Es hora de dar un paso atrás y mirar los principios clave de refactorización y algunos de los problemas que
Es necesario pensar en el uso de refactorización.

Definición de refactorización

Siempre desconfío un poco de las definiciones porque todos tienen las suyas, pero cuando escribes un
libro puedes elegir tus propias definiciones. En este caso baso mis definiciones en el trabajo
hecho por el grupo de Ralph Johnson y una variedad de asociados.

Lo primero que hay que decir al respecto es que la palabra *Refactorización* tiene dos definiciones dependiendo de
contexto. Puede encontrar esto molesto (ciertamente lo hago), pero sirve como otro ejemplo más de la
realidades de trabajar con lenguaje natural.

La primera definición es la forma del sustantivo.

Propina

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 Pull Up
Campo . Como tal, una refactorización suele ser un pequeño cambio en el software, aunque una refactorización
puede involucrar a otros. Por ejemplo, Extract Class generalmente involucra Move Method y Move
Campo .

El otro uso de la *refactorización* es la forma verbal

Propina

> ***Refactor*** *(verbo): para reestructurar el software aplicando una serie de refactorizaciones sin cambiando su comportamiento observable.*

Por lo tanto, puede pasar algunas horas refactorizando, durante las cuales puede aplicar un par de docenas 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 de manera controlada. Desde que utilizo la refactorización, noté que limpio mucho más código efectivamente que lo hice antes. Esto se debe a que sé qué refactorizaciones usar, sé cómo usar ellos de una manera que minimiza los errores, y los pruebo en cada oportunidad posible.

Debería amplificar un par de puntos en mis definiciones. Primero, el propósito de refactorizar es hacer El software es más fácil de entender y modificar. Puede hacer muchos cambios en el software que hacer poco o ningún cambio en el comportamiento observable. Solo cambios realizados para hacer el software más fáciles de entender son las refactorizaciones. Un buen contraste es la optimización del rendimiento. Me gusta refactorización, la optimización del rendimiento no suele cambiar el comportamiento de un componente (aparte de su velocidad); solo 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 debe hacerlo para obtener Rendimiento que necesita.

Lo segundo que quiero destacar es que la refactorización no cambia el comportamiento observable del software El software todavía lleva a cabo la misma función que antes. Cualquier usuario, ya sea un usuario final u otro programador, no puede decir que las cosas han cambiado.

Los dos sombreros

Este segundo punto conduce a la metáfora de dos sombreros de Kent Beck. Cuando usa refactorización para desarrolla software, divide su tiempo entre dos actividades distintas: agregar funciones y refactorización. Cuando agrega una función, no debería cambiar el código existente; solo estás agregando Nuevas capacidades. Puede medir su progreso agregando pruebas y haciendo que las pruebas funcionen. Cuando refactoriza, hace un punto de no agregar función; solo reestructura el código. Tú no agregue ninguna prueba (a menos que encuentre un caso que perdió antes); solo reestructura el código. No agrega ninguna prueba (a menos que encuentre un caso que se perdió antes); solo cambias las pruebas cuando absolutamente necesario para hacer frente a un cambio en una interfaz.

A medida que desarrolles software, probablemente te encuentres intercambiando sombreros con frecuencia. Empiezas intentando para agregar una nueva función, y se da cuenta de que esto sería mucho más fácil si el código estuviera estructurado diferentemente. Entonces intercambias sombreros y refactorizas por un tiempo. Una vez que el código está mejor estructurado, usted intercambia sombreros y agrega la nueva función. Una vez que tienes la nueva función funcionando, te das cuenta de que codificado de una manera que es incómoda de entender, por lo que intercambia sombreros nuevamente y refactoriza. Todo esto puede tomar solo diez minutos, pero durante este tiempo siempre debes saber qué sombrero eres vistiendo.

¿Por qué deberías refactorizar?

No quiero proclamar la refactorización como la cura para todos los problemas de software. No es una "bala de plata". Sin embargo, es un valiosa herramienta, un par de alicates plateados que te ayudan a mantener un buen control de tu código. Refactorizar es un herramienta que puede y debe usarse para varios propósitos.

La refactorización mejora el diseño de software

Sin refactorizar, el diseño del programa decaerá. A medida que la gente cambia de código, cambia a realizar objetivos a corto plazo o cambios realizados sin una comprensión completa del diseño de la código: el código pierde su estructura. Se hace más difícil ver el diseño leyendo el código. Refactorizar es como ordenar el código. Se trabaja para eliminar los bits que no están realmente 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, cuanto más difícil sea preservarlo, y más rápidamente se descompone. Regular La refactorización 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 lo tanto, un aspecto importante de mejorar El diseño es eliminar el código duplicado. La importancia de esto radica en futuras modificaciones a la código. Reducir la cantidad de código no hará que el sistema se ejecute más rápido, porque el efecto en La huella de los programas rara vez es significativa. Sin embargo, reducir la cantidad de código

47

48

Hacer una gran diferencia en la modificación del código. Cuanto más código haya, más difícil será modificar correctamente Hay más código para entender. Cambias este bit de código aquí, pero el el sistema no hace lo que esperas porque no cambiaste un poco por allá que hace mucho lo mismo en un contexto ligeramente diferente. Al eliminar los duplicados, se asegura de que El código lo dice todo una vez y solo una vez, que es la esencia del buen diseño.

La refactorización hace que el software sea más fácil de entender

La programación es en muchos sentidos una conversación con una computadora. Escribes código que le dice al computadora qué hacer, y responde haciendo exactamente lo que le dices. A tiempo cierras la brecha entre lo que quieres que haga y lo que le dices que haga. La programación en este modo tiene que ver con diciendo exactamente lo que quieres. Pero hay otro usuario de su código fuente. Alguien intentará lea su código en unos meses para hacer algunos cambios. Olvidamos fácilmente ese usuario adicional de el código, sin embargo, ese usuario es en realidad el más importante. A quién le importa si la computadora tarda ¿Más ciclos para compilar algo? Importa si un programador tarda una semana en hacer un cambio que hubiera tomado solo una hora si hubiera entendido su código.

El problema es que cuando intentas que el programa funcione, no estás pensando en eso futuro desarrollador Se necesita un cambio de ritmo para hacer cambios que hagan que el código sea más fácil entender. La refactorización lo ayuda a hacer que su código sea más legible. Al refactorizar tienes código que funciona pero no está idealmente estructurado. Un poco de tiempo refactorizado puede hacer que el código mejor comunicar su finalidad. La programación en este modo se trata de decir exactamente lo que media.

No estoy siendo necesariamente altruista sobre esto. A menudo este futuro desarrollador soy yo. Aquí refactorizando Es particularmente importante. Soy un programador muy vago. Una de mis formas de pereza es que nunca recuerda cosas sobre el código que escribo. De hecho, intento deliberadamente no recordar nada que pueda Miro hacia arriba, porque me temo que mi cerebro se llenará. Intento poner todo lo que debería recordar en el código para no tener que recordarlo. De esa manera estoy menos preocupado por Old Peculier [Jackson] matando mis células cerebrales.

Esta comprensibilidad también funciona de otra manera. Utilizo refactorización para ayudarme a entender que no estoy familiarizado código. Cuando miro un código desconocido, tengo que tratar de entender lo que hace. Miro un par de líneas y decirme a mí mismo, oh sí, eso es lo que está haciendo 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 lo pruebo comprensión volviendo a ejecutar el código para ver si aún funciona.

Al principio realizo una refactorización como esta en pequeños detalles. A medida que el código se aclara, veo que puedo ver cosas sobre el diseño que no podía ver antes. Si no hubiera cambiado el código, probablemente nunca lo haría

He visto estas cosas, porque no soy lo suficientemente inteligente como para visualizar todo esto en mi cabeza. Ralph Johnson describe estas refactorizaciones tempranas como limpiar la suciedad de una ventana para que pueda ver más allá. Cuando estoy estudiando el código, encuentro que la refactorización me lleva a niveles más altos de comprensión que de lo contrario me perdería.

Refactorizar te ayuda a encontrar errores

La ayuda para comprender el código también me ayuda a detectar errores. Admito que no soy terriblemente bueno para encontrar loco. Algunas personas pueden leer un trozo de código y ver errores, yo no. Sin embargo, me parece que si yo refactorizar código, trabajo profundamente para entender lo que hace el código, y pongo ese nuevo entendiendo de nuevo en el código. Al aclarar la estructura del programa, aclaro ciertos Suposiciones que he hecho, hasta el punto de que ni siquiera yo puedo evitar detectar los errores.

Página 49

Me recuerda una declaración que Kent Beck hace a menudo sobre sí mismo: "No soy un gran programador; Solo soy un buen programador con excelentes hábitos ". La refactorización me ayuda a ser mucho más eficaz en escribir código robusto

Refactorizar te ayuda a programar más rápido

Al final, todos los puntos anteriores se reducen a esto: la refactorización lo ayuda a desarrollar más código con rapidez.

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

Creo firmemente que un buen diseño es esencial para un rápido desarrollo de software. De hecho, el Todo el punto de tener un buen diseño es permitir un rápido desarrollo. Sin un buen diseño, tú puede progresar rápidamente por un tiempo, pero pronto el mal diseño comienza a ralentizarlo. Tu gastas tiempo para encontrar y corregir errores en lugar de agregar una nueva función. Los cambios tardan más en intentarlo entienda el sistema y encuentre el código duplicado. Las nuevas funciones necesitan más codificación a medida que parches sobre 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. Refactorizar te ayuda desarrollar software más rápidamente, porque evita que el diseño del sistema se deteriore. Puede Incluso mejorar un diseño.

¿Cuándo deberías refactorizar?

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

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

La regla de tres

Aquí hay una guía que Don Roberts me dio: la primera vez que haces algo, simplemente lo haces. los la segunda vez que haces algo similar, haces una mueca por la duplicación, pero haces lo duplicado de todas formas. La tercera vez que haces algo similar, refactorizas.

Propina

Tres golpes y refactoriza.

Refactorizar cuando agrega función

El momento más común para refactorizar es cuando quiero agregar una nueva función a algún software. A menudo
La primera razón para refactorizar aquí es para ayudarme a comprender algún código que necesito modificar. Este código
puede haber sido escrito por otra persona, o puede que lo haya escrito. Cada vez que tengo que pensar

Página 50

entiendo lo que está haciendo el código, me pregunto si puedo refactorizar el código para hacer eso
comprensión 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
a lo largo.

El otro impulsor de la refactorización aquí es un diseño que no me ayuda a agregar una función fácilmente. yo miro
en el diseño y decirme a mí mismo: "Si tan solo hubiera diseñado el código de esta manera, agregar esta característica
sea fácil ". En este caso no me preocupo por mis fechorías pasadas, las arreglo refactorizando. Lo hago en parte
para facilitar futuras mejoras, pero principalmente lo hago porque he descubierto que es la forma más rápida.
La refactorización es un proceso rápido y sin problemas. Una vez que he refactorizado, agregar la función puede ser muy útil.
Más rápido y sin problemas.

Refactorizar cuando necesita corregir un error

Al corregir errores, gran parte del uso de la refactorización proviene de hacer que el código sea más comprensible. Como yo
mire el código tratando de entenderlo, 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 obtienes un informe de error, es una señal de que necesitas refactorizar, porque el código no estaba claro
suficiente para que veas que hubo un error.

Refactorizar mientras hace una revisión de código

Algunas organizaciones hacen revisiones regulares de códigos; aquellos que no lo harían mejor si lo hicieran. Código
Las revisiones ayudan a difundir el conocimiento a través de un equipo de desarrollo. Los comentarios ayudan a los más experimentados
Los desarrolladores transmiten sus conocimientos a personas menos experimentadas. Ayudan a más personas a entender más
aspectos de un gran sistema de software. También son muy importantes para escribir código claro. Mi código
Puede parecer claro para mí, pero no para mi equipo. Eso es inevitable: es muy difícil para la gente poner
en el lugar de alguien que no está familiarizado con las cosas en las que está trabajando. Comentarios también
Brinde la oportunidad a más personas de sugerir ideas útiles. Solo puedo pensar en tantos buenos
Ideas en una semana. Hacer que otras personas contribuyan me hace la vida más fácil, así que siempre busco muchos
comentarios

Descubrí que la refactorización me ayuda a revisar el código de otra persona. Antes de comenzar a usar
refactorizando, pude leer el código, entender cierto grado y hacer sugerencias. Ahora
Cuando se me ocurren ideas, considero si se pueden implementar fácilmente en ese momento
con refactorización. Si es así, refactorizo. Cuando lo hago varias veces, puedo ver más claramente cuál es el código
parece con las sugerencias en su lugar. No tengo que imaginar cómo sería, puedo *ver*
como es. Como resultado, puedo llegar a un segundo nivel de ideas que nunca tendría
me di cuenta si no hubiera refactorizado.

La refactorización también ayuda a que la revisión del código tenga resultados más concretos. No solo hay

sugerencias, pero también muchas sugerencias se implementan allí y luego. Terminas con mucho más de una sensación de logro del ejercicio.

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

Con revisiones de diseño más grandes, a menudo es mejor obtener varias opiniones en un grupo más grande. Demostración El código a menudo no es el mejor dispositivo para esto. Prefiero diagramas UML y recorriendo escenarios con tarjetas CRC. Así que diseño revisiones con grupos y revisiones de código con revisores individuales.

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

Por qué funciona la refactorización

Kent Beck

Los programas tienen dos tipos de valor: lo que pueden hacer por usted hoy y qué pueden hacer por ti mañana. La mayoría de las veces cuando estamos programación, estamos enfocados en lo que queremos que haga el programa hoy. Ya sea que estemos arreglando un error o agregando una función, estamos haciendo programa más valioso haciéndolo más capaz.

No puedes programar mucho tiempo sin darte cuenta de lo que hace el sistema hoy es solo una parte de la historia. Si puedes hacer el trabajo de hoy hoy, pero lo haces de tal manera que no puedes conseguir el trabajo de mañana hecho mañana, entonces pierdes. Tenga en cuenta, sin embargo, que sabe lo que debe hacerlo hoy, pero no está seguro del mañana. Tal vez tu haz esto, tal vez aquello, tal vez algo que aún no has imaginado.

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

Refactorizar es una forma de salir del vínculo. Cuando encuentres que ayer la decisión no tiene sentido hoy, tú cambias la decisión. Ahora tu puede hacer el trabajo de hoy. Mañana, parte de su comprensión a partir de hoy parecerá ingenuo, así que también cambiarás eso.

¿Qué es lo que hace que los programas sean difíciles de trabajar? Cuatro cosas que puedo pensar mientras escribo esto son los siguientes:

- Λοσ programas que son difíciles de leer son difíciles de modificar.
- Λοσ programas que tienen lógica duplicada son difíciles de modificar.
- Programas que requieren un comportamiento adicional que requiere que usted cambiar el código de ejecución es difícil de modificar.

• Λοσ programas con lógica condicional compleja son difíciles de modificar.

Entonces, queremos programas que sean fáciles de leer, que tengan toda la lógica especificada en un solo lugar, que no permiten que los cambios pongan en peligro comportamiento existente, y eso permite que la lógica condicional se exprese como Simplemente como sea posible.

Refactorizar es el proceso de tomar un programa en ejecución y agregarlo a su valor, no cambiando su comportamiento sino dándole más de estos cualidades que nos permiten seguir desarrollándonos a toda velocidad.

¿Qué le digo a mi gerente?

Cómo decirle a un gerente sobre la refactorización es una de las preguntas más comunes que me han hecho. Si el gerente es técnicamente inteligente, presentar el tema puede no ser tan difícil. Si el gerente es *genuinamente* orientado a la calidad, lo que hay que destacar son los aspectos de calidad. Aquí usando refactorización en El proceso de revisión es una buena manera de trabajar las cosas. Toneladas de estudios muestran que las revisiones técnicas son Una forma importante de reducir errores y acelerar el desarrollo. Echa un vistazo a cualquier libro sobre revisiones, inspecciones o el proceso de desarrollo de software para las últimas citas. Estos deberían convencer a la mayoría de los gerentes del valor de las revisiones. Es entonces un paso corto para introducir la refactorización como una forma de obtener comentarios de revisión en el código.

Por supuesto, muchas personas dicen que están motivadas por la calidad, pero están más motivadas por el horario. En estos Casos Doy mi consejo más controvertido: ¡No lo cuentes!

¿Subversivo? No lo creo. Los desarrolladores de software son profesionales. Nuestro trabajo es construir efectivo software tan rápido como podamos. Mi experiencia es que la refactorización es una gran ayuda para crear software con rapidez. Si necesito agregar una nueva función y el diseño no se adapta al cambio, creo que es más rápido refactorizar primero y luego agregar la función. Si necesito corregir un error, necesito entender cómo el software funciona, y creo que la refactorización es la forma más rápida de hacerlo. Un gerente programado quiere que haga las cosas lo más rápido que pueda; cómo lo hago es mi negocio. La forma más rápida es refactor Por lo tanto, refactorizo.

Indirección y Refactorización

Kent Beck

La informática es la disciplina que cree que todos los problemas pueden ser resuelto con una capa más de indirección. —Dennis DeBruler

Dada la infatuación de los ingenieros de software con indirección, puede que no sorprenda aprenderá que la mayoría de las refactorizaciones introducen más indirección en un programa. La refactorización tiende a dividir objetos grandes en varios más pequeños unos y grandes métodos en varios más pequeños.

Sin embargo, la indirecta es una espada de dos filos. Cada vez que rompes uno cosa en dos partes, tienes más cosas que administrar. También puede hacer un programa más difícil de leer cuando un objeto delega a un objeto que delega a un objeto Entonces le gustaría minimizar la indirección.

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

formas.

- **Para permitir compartir la 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.**

Page 53

Elegir el nombre de cada clase y el nombre de cada método le da la oportunidad de explicar lo que pretende. Los internos de la clase o método explican cómo se realiza la intención. Si el las partes internas también están escritas en términos de intención en aún más pequeñas piezas, puedes escribir código que comunique la mayoría de Información importante sobre su propia estructura.

- **Para aislar el cambio.**

  Yo uso un objeto en dos lugares diferentes. Quiero cambiar el comportamiento en uno de los dos casos. Si cambio el objeto, me arriesgo cambiando ambos. Así que primero hago una subclase y me refiero a ella en el caso que está cambiando Ahora puedo modificar la subclase sin arriesgando un cambio inadvertido al otro caso.

- **Para codificar la lógica condicional.**

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

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

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

Contrasta esto con un cuidadoso diseño inicial. El diseño especulativo es un Intente poner todas las buenas cualidades en el sistema antes de que cualquier código sea escrito. Entonces el código se puede colgar en el robusto esqueleto. los El problema con este proceso es que es demasiado fácil adivinar mal. Con refactorizando, nunca estás en peligro de estar completamente equivocado. los El programa siempre se comporta al final como lo hizo al principio. En

Además, tiene la oportunidad de agregar cualidades valiosas al código.

Hay un segundo juego de refactorización más raro. Identificar indirección que no es pagar por sí mismo y sacarlo. A menudo esto toma la forma de intermedio métodos que solían tener un propósito pero ya no lo hacen. O podría ser un componente que esperaba que fuera compartido o polimórfico pero que resultó Para ser utilizado en un solo lugar. Cuando encuentre indirección parasitaria, tómela fuera. Nuevamente, tendrá un programa más valioso, no porque haya

más de una de las cuatro cualidades mencionadas anteriormente, pero porque cuesta menos indirección para obtener la misma cantidad de las cualidades.

Problemas con la refactorización

Cuando aprende una nueva técnica que mejora enormemente su productividad, es difícil ver cuándo no se aplica. Por lo general, lo aprende dentro de un contexto específico, a menudo solo un solo proyecto. Es difícil para ver qué hace que la técnica sea menos efectiva, incluso dañina. Hace diez años era como eso con los objetos. Si alguien me preguntaba cuándo no usar objetos, era difícil responder. No fue que no creía que los objetos tuvieran limitaciones, soy demasiado cínico para eso. Era solo que no sabía cuáles eran esas limitaciones, aunque sabía cuáles eran los beneficios.

La refactorización es así ahora. Conocemos los beneficios de la refactorización. Sabemos que pueden hacer un Diferencia palpable a nuestro trabajo. Pero no hemos tenido una experiencia lo suficientemente amplia como para ver dónde Se aplican limitaciones.

Esta sección es más corta de lo que me gustaría y es más tentativa. A medida que más personas aprenden sobre refactorizando, aprenderemos más. Para ti esto significa que si bien creo que deberías intentarlo refactorizando las ganancias reales que puede proporcionar, también debe monitorear su progreso. Tener cuidado de problemas que la refactorización puede estar presentando. Háganos saber acerca de estos problemas. Mientras aprendemos más sobre refactorización, encontraremos más soluciones a los problemas y aprenderemos sobre qué Los problemas son difíciles de resolver.

Bases de datos

Un área problemática para la refactorización son las bases de datos. La mayoría de las aplicaciones comerciales están estrechamente unidas a El esquema de la base de datos que los soporta. Esa es una razón por la cual la base de datos es difícil de cambio. Otra razón es la migración de datos. Incluso si ha colocado cuidadosamente su sistema en capas para minimizar las dependencias entre el esquema de la base de datos y el modelo de objetos, cambiando el El esquema de la base de datos lo obliga a migrar los datos, lo que puede ser una tarea larga y difícil.

Con bases de datos no objetivas, una forma de abordar este problema es colocar una capa de software separada entre su modelo de objeto y su modelo de base de datos. De esa manera puede aislar los cambios a Los dos modelos diferentes. A medida que actualiza un modelo, no necesita actualizar el otro. Tu solo Actualiza la capa intermedia. Tal capa agrega complejidad pero le da mucha flexibilidad. Incluso sin refactorizar es muy importante en situaciones en las que tiene múltiples bases de datos o un modelo de base de datos complejo sobre el que no tiene control.

No tiene que comenzar con una capa separada. Puede crear la capa a medida que observa partes de su modelo de objeto volviéndose volátil. De esta manera, obtiene el mayor apalancamiento para sus cambios.

Las bases de datos de objetos ayudan y dificultan. Algunas bases de datos orientadas a objetos proporcionan información automática. migración de una versión de un objeto a otro. Esto reduce el esfuerzo pero aún impone un tiempo penalización mientras se realiza la migración. Cuando la migración no es automática, debe hacer lo

migrar a ti mismo, lo cual es un gran esfuerzo. En esta situación, debes tener más cuidado con cambios en la estructura de datos de las clases. Todavía puedes mover libremente el comportamiento, pero tienes ser más cauteloso con los campos en movimiento. Necesitas usar accesores para dar la ilusión de que el los datos se han movido, incluso cuando no lo han hecho. Cuando esté bastante seguro de saber dónde deben estar los datos para ser, puede mover y migrar los datos en un solo movimiento. Solo los accesos necesitan cambiar, reduciendo el riesgo de problemas con errores.

Interfaces cambiantes

Página 55

Una de las cosas importantes sobre los objetos es que le permiten cambiar la implementación de un módulo de software por separado de cambiar la interfaz. Puede cambiar las partes internas de forma segura un objeto sin que nadie más se preocupe por él, pero la interfaz es importante: cambie eso y cualquier cosa puede suceder.

Algo que es inquietante acerca de la refactorización es que muchas de las refactorizaciones cambian interfaz. Algo tan simple como Rename Method se trata de cambiar una interfaz. Y qué ¿Esto le hace a la preciada noción de encapsulación?

No hay problema para cambiar el nombre de un método si tiene acceso a todo el código que lo llama método. Incluso si el método es público, siempre que pueda contactar y cambiar a todas las personas que llaman, usted Puede cambiar el nombre del método. Hay un problema solo si la interfaz está siendo utilizada por el código que usted No puede encontrar y cambiar. Cuando esto sucede, digo que la interfaz se convierte en una *publicación interfaz* (un paso más allá de una interfaz pública). Una vez que publica una interfaz, ya no puede cámbielo de forma segura y solo edite las llamadas. Necesitas un proceso algo más complicado.

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

En resumen, si una refactorización cambia una interfaz publicada, debe conservar tanto la interfaz anterior y el nuevo, al menos hasta que los usuarios hayan tenido la oportunidad de reaccionar al cambio. Por suerte, Esto no es demasiado incómodo. Por lo general, puede organizar las cosas para que la interfaz anterior siga funcionando. Intentar haga esto para que la interfaz anterior llame a la nueva interfaz. De esta manera cuando cambia el nombre de un método, conserve el antiguo y simplemente deje que llame al nuevo. No copie el cuerpo del método, eso te lleva por el camino de la condenación a través de un código duplicado. También deberías usar el recurso de desaprobación en Java para marcar el código como desaprobado. De esa manera, las personas que llaman sabrán que Algo está arriba.

Un buen ejemplo de este proceso son las clases de colección de Java. Los nuevos presentes en Java 2 Reemplazar los que se proporcionaron originalmente. Sin embargo, cuando se lanzaron los Java 2, JavaSoft hizo un gran esfuerzo para proporcionar una ruta de migración.

La protección de las interfaces generalmente es factible, pero es una molestia. Tienes que construir y mantener estos extras métodos, al menos por un tiempo. Los métodos complican la interfaz y hacen que sea más difícil de usar. Ahí es una alternativa: no publique la interfaz. Ahora no estoy hablando de una prohibición total aquí, claramente Tienes que tener interfaces publicadas. Si está creando API para consumo externo, como Sun lo hace, entonces tienes que tener interfaces publicadas. Digo esto porque a menudo veo desarrollo grupos que usan interfaces publicadas demasiado. He visto a un equipo de tres personas operar en tales una manera en que cada persona publicó interfaces con las otras dos. Esto llevó a todo tipo de giros a mantener las interfaces cuando hubiera sido más fácil ingresar a la base del código y realizar las ediciones. Las organizaciones con una noción demasiado fuerte de propiedad del código tienden a comportarse de esta manera. Utilizando Las interfaces publicadas son útiles, pero tienen un costo. Así que no publique interfaces a menos que Realmente lo necesito. Esto puede significar modificar las reglas de propiedad de su código para permitir que las personas cambien código de otras personas para admitir un cambio de interfaz. A menudo es una buena idea hacer esto con programación en pareja.

Propina

No publique interfaces prematuramente. Modifique las políticas de propiedad de su código para suavizar refactorización.

Hay un área particular con problemas al cambiar las interfaces en Java: agregar una excepción a La cláusula de tiros. Esto no es un cambio en la firma, por lo que no puede usar la delegación para cubrirlo. los Sin embargo, el compilador no lo dejará compilar. Es difícil lidiar con este problema. Puedes elegir un nuevo nombre para el método, deje que el método anterior lo llame y convierta el marcado en un desmarcado excepción. También puede lanzar una excepción no verificada, aunque luego pierde la comprobación capacidad. Cuando hace esto, puede alertar a las personas que llaman que la excepción se convertirá en marcada excepción en una fecha futura. Luego tienen tiempo para poner los controladores en su código. Para esto razón por la que prefiero definir una excepción de superclase para un paquete completo (como SQLException para java.sql) y asegúrese de que los métodos públicos solo declaren esta excepción en su cláusula throws. Ese puedo definir excepciones de subclase si lo deseo, pero esto no afectará a la persona que llama que solo conoce Sobre el caso general.

Cambios de diseño que son difíciles de refactorizar

¿Puede refactorizar su salida de cualquier error de diseño, o son algunas decisiones de diseño tan centrales? que no puedes contar con refactorizar para cambiar de opinión más tarde? Esta es un área en la que tenemos Datos muy incompletos. Ciertamente, a menudo nos hemos sorprendido por 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, pregunto yo mismo lo difícil que sería refactorizar de un diseño a otro. Si parece fácil, no me preocupo demasiado por la elección, y elijo el diseño más simple, incluso si no cubre todos los requisitos potenciales Sin embargo, si no puedo ver una manera simple de refactorizar, entonces pongo más esfuerzo en el diseño Encuentro que tales situaciones son minoritarias.

¿Cuándo no deberías refactorizar?

Hay momentos en que no debe refactorizar en absoluto. El ejemplo principal es cuando deberías reescribir desde cero en su lugar. Hay momentos en que el código existente es un desastre que aunque podría refactorizarlo, sería más fácil comenzar desde el principio. Esta decisión no es uno fácil de hacer, 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. Puedes descubrir esto solo tratando de probarlo y descubriendo que el código está tan lleno de errores que no puede estabilizarlo. Recuerde, el código tiene que funcionar principalmente correctamente antes de refactorizar.

Una ruta de compromiso es refactorizar una gran pieza de software en componentes con fuerte encapsulación Luego puede tomar una decisión de refactor versus reconstrucción para un componente a la vez hora. Este es un enfoque prometedor, pero no tengo suficientes datos para escribir buenas reglas para hacerlo. ese. Con un sistema heredado clave, esta sería una dirección atractiva para tomar.

La otra vez que debe evitar la refactorización es cuando está cerca de una fecha límite. En ese punto el La ganancia de productividad de la refactorización aparecería después de la fecha límite y, por lo tanto, sería demasiado tarde. Sala Cunningham tiene una buena manera de pensar en esto. Describe la refactorización inacabada como entrar en deuda. La mayoría de las empresas necesitan alguna deuda para funcionar de manera eficiente. Sin embargo, con la deuda viene pagos de intereses, es decir, el costo adicional de mantenimiento y extensión causado por excesivamente complejo código. Puede pagar algunos pagos de intereses, pero si los pagos se vuelven demasiado grandes, será

abrumado. Es importante administrar su deuda, pagando parte de ella mediante la refactorización.

Sin embargo, salvo cuando esté muy cerca de una fecha límite, no debe posponer la refactorización porque no tienes tiempo La experiencia con varios proyectos ha demostrado que una serie de

La refactorización da como resultado una mayor productividad. No tener suficiente tiempo generalmente es una señal de que Necesito hacer algunas refactorizaciones.

Refactorización y Diseño

La refactorización tiene un papel especial como complemento del diseño. Cuando aprendí a programar, simplemente escribí el programa y me abrí paso a través de él. Con el tiempo aprendí que pensar en el El diseño de antemano me ayudó a evitar costosas modificaciones. Con el tiempo me metí más en este estilo *inicial diseño.* Muchas personas consideran que el diseño es la pieza clave y que la programación solo es mecánica. los 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 Alistair Cockburn dice: "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 tu no hagas ningún diseño en absoluto. Simplemente codifique el primer enfoque que se le ocurra, entiéndalo trabajando, y luego refactorizarlo en forma. En realidad, este enfoque puede funcionar. He visto a gente hacer esto y salir con un software muy bien diseñado. Los que apoyan Extreme La programación [Beck, XP] a menudo se retrata como abogando por este enfoque.

Aunque solo funciona la refactorización, no es la forma más eficiente de trabajar. Incluso el Los programadores extremos primero hacen un diseño. Probarán varias ideas con tarjetas CRC o el como hasta que tengan una primera solución plausible. Solo después de generar un primer disparo plausible podrán código y luego refactorizar. El punto es que la refactorización cambia el papel del diseño inicial. Si tu no refactorice, hay mucha presión para lograr ese diseño inicial correcto. El sentido es que cualquier Los cambios en el diseño más adelante serán costosos. Por lo tanto, pones más tiempo y esfuerzo en Diseño inicial para evitar la necesidad de tales cambios.

Con la refactorización el énfasis cambia. Todavía haces un diseño inicial, pero ahora no intentas encontrar *La* solución. En cambio, todo lo que quieres es una solución razonable. Sabes que a medida que construyes el solución, a medida que comprende más sobre el problema, se da cuenta de que la mejor solución es diferente del que originalmente se te ocurrió. Con la refactorización esto no es un problema, porque ya no lo es caro hacer los 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 lo haría me pregunto cómo cambiaría ese requisito durante la vida útil del sistema. Porque el diseño los cambios eran caros, buscaría construir un diseño que resistiera los cambios que podría prever. El problema con la construcción de una solución flexible es que los costos de flexibilidad. Flexible Las soluciones son más complejas que las simples. El software resultante es más difícil de mantener. en general, aunque es más fácil flexionar en la dirección que tenía en mente. Incluso allí, sin embargo, tú Hay que entender cómo flexionar el diseño. Para uno o dos aspectos esto no es gran cosa, pero los cambios ocurren en todo el sistema. Construir flexibilidad en todos estos lugares hace que sistema mucho más complejo y costoso de mantener. La gran frustración, por supuesto, es que todo Esta flexibilidad no es necesaria. Algo de eso es, pero es imposible predecir qué piezas son esas. A para obtener flexibilidad, se ve obligado a poner mucha más flexibilidad de la que realmente necesita.

Con la refactorización, usted aborda los riesgos del cambio de manera diferente. Todavía piensas en potencial cambios, aún considera soluciones flexibles. Pero en lugar de implementar estas soluciones flexibles, te preguntas: "¿Qué tan difícil va a ser refactorizar una solución simple en el flexible?

solución? "Si, como sucede la mayoría de las veces, la respuesta es" bastante fácil ", entonces simplemente implementa La solución simple.

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

Ni siquiera pienses en las soluciones flexibles. Tienes la confianza para refactorizar si llega el momento. Construyes lo más simple que posiblemente pueda funcionar. En cuanto al diseño flexible y complejo, la mayoría de el tiempo que no lo vas a necesitar.

Se tarda un tiempo en crear nada

Ron Jeffries

Se estaba ejecutando el proceso de pago de compensación integral de Chrysler Demasiado lento. Aunque todavía estábamos en desarrollo, comenzó a molestarnos, porque estaba ralentizando las pruebas.

Kent Beck, Martin Fowler y yo decidimos que lo arreglaríamos. Mientras esperaba nos reunimos, estaba especulando, sobre la base de mi extensa conocimiento del sistema, sobre lo que probablemente lo estaba ralentizando. yo Pensé en varias posibilidades y conversé con la gente sobre los cambios. eso probablemente era necesario. 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. Ninguno de los Las posibilidades que había pensado tenían algo que ver con el problema. En cambio, descubrimos que el sistema pasaba la mitad de su tiempo creando instancias de fecha. Aún más interesante fue que todas las instancias tenían el mismo par de valores

Cuando miramos la lógica de creación de fecha, vimos algunas oportunidades para optimizar cómo se crearon estas fechas. Todos iban a través de una conversión de cadena aunque no haya entradas externas involucrado. El código solo estaba usando la conversión de cadenas para la conveniencia de mecanografía. Tal vez podríamos optimizar eso.

Luego vimos cómo se usaban estas fechas. Resultó que la gran mayoría de ellos estaban creando instancias de rango de fechas, un objeto con una fecha de inicio y una fecha de inicio. Mirando un poco más, nosotros ¡Me di cuenta de que la mayoría de estos rangos de fechas estaban vacíos!

Mientras trabajábamos con el rango de fechas, usamos la convención de que cualquier fecha el rango que terminó antes de comenzar estaba vacío. Es una buena convención y encaja bien con el funcionamiento de la clase. Poco después comenzamos a usar esto convención, nos dimos cuenta de que solo creamos un intervalo de fechas que comienza después los extremos no eran código claro, por lo que extrajimos ese comportamiento en una fábrica método para intervalos de fechas vacíos.

Hicimos ese cambio para aclarar el código, pero recibimos un

pago inesperado Creamos un intervalo de fechas vacío constante y
ajustó el método de fábrica para devolver ese objeto en lugar de crearlo
cada vez. Ese cambio duplicó la velocidad del sistema, suficiente para

Las pruebas para ser soportable. Nos llevó unos cinco minutos.

Había especulado con varios miembros del equipo (Kent y Martin
negar participar en la especulación) sobre lo que probablemente estaba mal con el código
Lo supimos muy bien. Incluso habíamos esbozado algunos diseños para
mejoras sin medir primero lo que estaba sucediendo.

Estábamos completamente equivocados. Aparte de tener un muy interesante
conversación, no estábamos haciendo nada bueno.

La lección es: incluso si sabe exactamente lo que está sucediendo en su sistema,
Mida el rendimiento, no especule. 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 hacer
el software es más fácil de entender, a menudo realiza cambios que harán que el programa se ejecute
más lentamente. Este es un tema importante. No soy de la escuela de pensamiento que ignora
rendimiento a 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 de la portería. Refactorizar ciertamente lo hará
hace que el software vaya más lento, pero también hace que el software sea más susceptible al rendimiento
sintonización. El secreto del software rápido, en todos los contextos en tiempo real, excepto en el difícil, es escribir software ajustable
primero y luego para ajustarlo a suficiente velocidad.

He visto tres enfoques generales para escribir software rápido. El más serio de estos es el tiempo.
presupuesto, usado a menudo en sistemas difíciles en tiempo real. En esta situación, al descomponer el diseño
Usted le da a cada componente un presupuesto de recursos: tiempo y huella. Ese componente no debe
exceder su presupuesto, aunque se permite un mecanismo para intercambiar tiempos presupuestados. Tal
El mecanismo centra la atención en los tiempos de rendimiento difíciles. Es esencial para sistemas como
marcapasos cardíacos, en los que los datos tardíos siempre son malos. Esta técnica es excesiva para otros tipos
de sistemas, como los sistemas de información corporativos con los que suelo trabajar.

El segundo enfoque es el enfoque de atención constante. Con este enfoque, cada programador,
todo el tiempo, hace todo lo que puede para mantener el rendimiento alto. Este es un enfoque común
y tiene una atracción intuitiva, pero no funciona muy bien. Cambios que mejoran el rendimiento.
por lo general, hace que sea más difícil trabajar con el programa. Esto ralentiza el desarrollo. Esto seria un costo
Vale la pena pagar si el software resultante fuera más rápido, pero generalmente no lo es. El desempeño
las mejoras se extienden por todo el programa, y cada mejora se realiza con un estrecho
perspectiva del comportamiento del programa.

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

El tercer enfoque para mejorar el rendimiento aprovecha esta estadística del 90 por ciento. En
Con este enfoque, construye su programa de una manera bien pensada sin prestar atención a

rendimiento hasta que comience una etapa de optimización del rendimiento, generalmente bastante tarde en el desarrollo. Durante la etapa de optimización del rendimiento, sigue un proceso específico para ajustar el programa.

Usted comienza ejecutando el programa bajo un generador de perfiles que lo monitorea y le dice dónde Está consumiendo tiempo y espacio. De esta manera puede encontrar esa pequeña parte del programa donde rendimiento puntos calientes mienten. Luego te enfocas en esos puntos críticos de rendimiento y usas el mismo optimizaciones que usaría si estuviera usando el enfoque de atención constante. Pero porque tu están enfocando su atención en un punto caliente, están teniendo mucho más efecto por menos trabajo. Aún así Sigues siendo cauteloso. Al igual que en la refactorización, realiza los cambios en pequeños pasos. Después de cada paso tu compilar, probar y volver a ejecutar el generador de perfiles. Si no ha mejorado el rendimiento, retrocede cambio. Continúa el proceso de búsqueda y eliminación de puntos calientes hasta obtener el rendimiento que satisface a sus usuarios. McConnel [McConnel] da más información sobre esto técnica.

Tener un programa bien diseñado ayuda con este estilo de optimización de dos maneras. Primero te da tiempo para dedicar al ajuste del rendimiento. Debido a que tiene un código bien factorizado, puede agregar funcionar más rápido Esto le da más tiempo para concentrarse en el rendimiento. (Perfilar te asegura enfoca ese tiempo en el lugar correcto.) Segundo, con un programa bien diseñado tienes más precisión granularidad para su análisis de rendimiento. Su generador de perfiles lo lleva a partes más pequeñas del código, que son más fáciles de sintonizar Debido a que el código es más claro, usted comprende mejor su opciones y de qué tipo de ajuste funcionará.

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

¿De dónde vino la refactorización?

No he logrado precisar el verdadero nacimiento del término *refactorización.* Buenos programadores ciertamente pasé al menos un tiempo limpiando su código. Hacen esto porque tienen aprendí que el código limpio es más fácil de cambiar que el código complejo y desordenado, y bueno los programadores saben que rara vez escriben código limpio la primera vez.

La refactorización va más allá de esto. En este libro estoy abogando por la refactorización como un elemento clave en el Todo el proceso de desarrollo de software. Dos de las primeras personas en reconocer la importancia de refactorizando fueron Ward Cunningham y Kent Beck, quienes trabajaron con Smalltalk desde la década de 1980 adelante. Smalltalk es un entorno que incluso entonces era particularmente hospitalario para la refactorización. Está Un entorno muy dinámico que le permite escribir rápidamente software altamente funcional. Charla tiene un ciclo muy corto de compilación-enlace-ejecución, lo que facilita cambiar las cosas rápidamente. Está también orientado a objetos y, por lo tanto, proporciona herramientas poderosas para minimizar el impacto del cambio detrás Interfaces bien definidas. Ward y Kent trabajaron duro para desarrollar un desarrollo de software proceso 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 hemos estado trabajando con la refactorización, aplicándola a software serio proyectos y perfeccionamiento del proceso.

Las ideas de Ward y Kent siempre han tenido una fuerte influencia en la comunidad de Smalltalk, y el La noción de refactorización se ha convertido en un elemento importante en la cultura Smalltalk. Otro líder figura en la comunidad Smalltalk es Ralph Johnson, profesor de la Universidad de Illinois en Urbana-Champaign, famosa por ser una de las pandillas de los cuatro [pandilla de los cuatro]. Uno de los de Ralph El mayor interés está en el desarrollo de marcos de software. Exploró cómo la refactorización puede ayudar Desarrollar un marco eficiente y flexible.

Bill Opdyke fue uno de los estudiantes de doctorado de Ralph y está particularmente interesado en los marcos. Él vio el valor potencial de la refactorización y vio que podría aplicarse a mucho más de Charla. Su experiencia fue en el desarrollo de conmutadores telefónicos, en los que una gran cantidad de

la complejidad se acumula con el tiempo y los cambios son difíciles de realizar. La investigación doctoral de Bill examinó

Página 61

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

Recuerdo haber conocido a Bill en la conferencia de OOPSLA en 1992. Nos sentamos en un café y discutimos parte del trabajo que había realizado en la construcción de un marco conceptual para la atención médica. Bill me habló de Su investigación, y recuerdo haber pensado: "Interesante, pero no tan importante". Chico era yo ¡incorrecto!

John Brant y Don Roberts han llevado las ideas de herramientas a la refactorización mucho más lejos para producir el Refactoring Browser, una herramienta de refactorización para Smalltalk. Contribuyen el Capítulo 14 a este libro, que describe además las herramientas de refactorización.

¿Y yo? Siempre había sentido inclinado a limpiar el código, pero nunca había considerado que sea *que* importante. Luego trabajé en un proyecto con Kent y vi cómo usaba la refactorización. Vi la diferencia hecho en productividad y calidad. Esa experiencia me convenció de que refactorizar era muy técnica importante Sin embargo, estaba frustrado porque no había ningún libro que pudiera dar a un programador de trabajo, y ninguno de los expertos anteriores tenía planes para escribir tal libro. Entonces, con su ayuda, lo hice.

Optimizar un sistema de nómina

Rich Garzaniti

Habíamos estado desarrollando la compensación integral de Chrysler Sistema durante bastante tiempo antes de comenzar a moverlo a GemStone. Naturalmente, cuando hicimos eso, descubrimos que el programa no era rápido suficiente. Trajimos a Jim Haungs, un maestro GemSmith, para ayudarnos optimizar el sistema

Después de un poco de 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 muestra el número de objetos que fueron siendo creado y donde fueron creados.

Para nuestra sorpresa, el mayor delincuente resultó ser la creación de instrumentos de cuerda. El más grande de los grandes fue la creación repetida de 12,000 bytes instrumentos de cuerda. Este era un problema particular porque la cadena era tan grande que Las instalaciones habituales de recolección de basura de GemStone no lo resolverían. Debido al tamaño, GemStone estaba paginando la cadena al disco cada vez fue creado. Resultó que las cuerdas se estaban construyendo en nuestro IO framework, y se estaban construyendo de tres en tres para cada salida ¡grabar!

Nuestra primera solución fue almacenar en caché una sola cadena de 12,000 bytes, que resolvió la mayoría del problema. Más tarde, cambiamos el marco para escribir directamente en un archivo

stream, que eliminó la creación de incluso una cadena.

Una vez que la enorme cuerda estaba fuera del camino, el perfilador de Jim encontró algo similar problemas con algunas cadenas más pequeñas: 800 bytes, 500 bytes, etc.
La conversión de estos para usar la función de flujo de archivos también los resolvió.

Con estas técnicas, mejoramos constantemente el rendimiento de
sistema. Durante el desarrollo parecía que tomaría más de 1,000
horas para ejecutar la nómina. Cuando realmente nos preparamos para comenzar, tomó 40 horas Después de un mes lo redujimos a alrededor de 18; cuando lanzamos nosotros tenía 12 años. Después de un año de funcionamiento y mejora del sistema para un nuevo grupo de empleados, se redujo a 9 horas.

Nuestra mayor mejora fue ejecutar el programa en múltiples hilos en
Una máquina multiprocesador. El sistema no fue diseñado con hilos en
importa, pero como estaba tan bien factorizado, nos llevó solo tres días
ejecutar en múltiples hilos. Ahora la nómina tarda un par de horas en ejecutarse.

Antes de que Jim proporcionara una herramienta que midiera el sistema en operación real, Teníamos buenas ideas sobre lo que estaba mal. Pero pasó mucho tiempo antes nuestras buenas ideas fueron las que debieron implementarse. El Real las medidas apuntaban en una dirección diferente y hacían mucho más grande diferencia.

Capítulo 3. Malos olores en el código

por Kent Beck y Martin Fowler

> *Si apesta, cámbialo.*
>
> —Grandma Beck, hablando sobre la filosofía de la crianza de los hijos.

A estas alturas ya tienes una buena idea de cómo funciona la refactorización. Pero solo porque sabes cómo no significa que sabes cuándo. Decidir cuándo comenzar a refactorizar y cuándo parar es tan importante a refactorizar 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. Estos son asuntos simples. Tratar de explicar cuándo debes hacer estas cosas no es tan cortado y secado. En lugar de apelar a una vaga noción de estética de programación (que francamente es lo que solemos hacer los consultores), quería algo un poco más sólido.

Estaba reflexionando sobre este tema complicado cuando visité a Kent Beck en Zurich. Quizás estaba bajo el influencia de los olores de su hija recién nacida en ese momento, pero se le ocurrió la idea describiendo el "cuándo" de la refactorización en términos de olores. "Huele", dices, "y eso se supone ser mejor que una estética vaga? "Bueno, sí. Observamos muchos códigos, escritos para proyectos que abarcan desde una gran variedad de éxitos hasta casi muertos. Al hacerlo, hemos aprendido a buscar ciertas estructuras en el código que sugieren (a veces gritan por) la posibilidad de refactorización. (Estamos cambiando a "nosotros" en este capítulo para reflejar el hecho de que Kent y yo escribimos Este capítulo conjuntamente. Puedes notar la diferencia porque los chistes son míos y los demás son de él.)

Una cosa que no intentaremos hacer aquí es darle criterios precisos para cuando una refactorización está vencida. En Nuestra experiencia no tiene un conjunto de métricas que compitan con la intuición humana. Lo que haremos es darte indicaciones de que hay problemas que pueden resolverse mediante una refactorización. Tendrás que desarrollar tu propio sentido de cuántas variables de instancia son demasiadas variables de instancia y cuántas líneas de código en un método son demasiadas líneas.

Debe usar este capítulo y la tabla en la contraportada interior como una forma de darle inspiración cuando no estás seguro de qué refactorizaciones hacer. Lea el capítulo (o hojee la tabla) para intenta identificar qué es lo que estás oliendo, luego ve a las refactorizaciones que sugerimos para ver si ellos te ayudarán. Es posible que no encuentre el olor exacto que puede detectar, pero con suerte debería señalar usted en la dirección correcta

Código duplicado

El número uno en el desfile 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 tiene la misma expresión en dos métodos de la misma clase. Entonces todo lo que tienes que hacer es extraer el método e invocar el código de ambos lugares.

Otro problema común de duplicación es cuando tienes la misma expresión en dos hermanos subclases Puede eliminar esta duplicación utilizando el Método de extracción en ambas clases y luego Pull Up Field . Si el código es similar pero no es el mismo, debe usar el Método de extracción para separe los bits similares de los diferentes bits. Entonces puede encontrar que puede usar la Plantilla de formulario

Página 64

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

Si ha duplicado el código en dos clases no relacionadas, considere usar Extract Class en una clase y luego use el nuevo componente en el otro. Otra posibilidad es que el método realmente pertenece solo a una de las clases y debe ser invocado por la otra clase o que el método pertenece a una tercera clase a la que deberían referirse ambas clases originales. Tienes que decida dónde tiene sentido el método y asegúrese de que esté allí y en ningún otro lugar.

Método largo

Los programas de objetos que viven mejor y más largos son aquellos con métodos cortos. Programadores nuevos a los objetos a menudo sienten que nunca se lleva a cabo ningún cálculo, que los programas de objetos son infinitos secuencias de delegación. Sin embargo, cuando ha vivido con un programa de este tipo durante algunos años, aprendes cuán valiosos son todos esos pequeños métodos. Todos los beneficios de la indirección: explicación, compartir y elegir: son compatibles con pequeños métodos (ver Indirección y Refactorización en la página 61).

Desde los primeros días de la programación, las personas se han dado cuenta de que cuanto más largo es un procedimiento, el Más difícil es entender. Los idiomas más antiguos llevaban una sobrecarga en las llamadas de subrutina, que disuadió a las personas de los pequeños métodos. Los lenguajes modernos de OO prácticamente han eliminado eso gastos generales para llamadas en proceso. Todavía hay una sobrecarga para el lector del código porque usted tiene que cambiar el contexto para ver qué hace el subprocedimiento. Entornos de desarrollo que permiten ver dos métodos a la vez ayuda a eliminar este paso, pero la clave real para facilitar entender pequeños métodos es un buen nombre. Si tiene un buen nombre para un método que no necesita mirar el cuerpo

El efecto neto es que deberías ser mucho más agresivo con los métodos de descomposición. UNA Lo heurístico que seguimos es que cada vez que sentimos la necesidad de comentar algo, escribimos un método en lugar. Tal método contiene el código que se comentó pero se nombra después de la intención del código en lugar 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 explica 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.

Noventa y nueve por ciento del tiempo, todo lo que tiene que hacer para acortar un método es el Método de extracción . Encontrar partes del método que parecen ir bien juntas y hacer un nuevo método.

Si tiene un método con muchos parámetros y variables temporales, estos elementos entran en forma de extraer métodos. Si intentas usar el *Método de extracción,* terminas pasando muchas de las parámetros y variables temporales como parámetros para el método extraído de que el resultado es apenas más legible que el original. A menudo puede usar Reemplazar temp con consulta para Eliminar las temperaturas. Las listas largas de parámetros se pueden reducir con Introducir parámetro Objeto y preservar todo el objeto .

Si lo ha intentado y todavía tiene demasiadas temperaturas y parámetros, es hora de salir del artillería pesada: Reemplazar Método con Método Objeto .

¿Cómo identifica los grupos de código para extraer? Una buena técnica es buscar comentarios. A menudo señalan este tipo de distancia semántica. Un bloque de código con un comentario que te dice lo que está haciendo puede ser reemplazado por un método cuyo nombre se basa en el comentario. Incluso un Vale la pena extraer una sola línea si necesita explicación.

Página 65

Los condicionales y los bucles también dan señales de extracciones. Use 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 La clase tiene demasiadas variables de instancia, el código duplicado no puede estar muy lejos.

Puede extraer la clase para agrupar varias variables. Elija variables para ir juntas El componente que tiene sentido para cada uno. Por ejemplo, "depositAmount" y "depositCurrency" es probable que pertenezcan juntos en un componente. Más generalmente, prefijos o sufijos comunes para algún subconjunto de las variables en una clase sugiere la oportunidad para un componente. Si el componente tiene sentido como una subclase, encontrará Extraer subclase a menudo es más fácil.

Algunas veces una clase no usa todas sus variables de instancia todo el tiempo. Si es así, puede a Extraer la clase o Extraer Subclase muchas veces.

Al igual que con una clase con demasiadas variables de instancia, una clase con demasiado código es la mejora primaria terreno para código duplicado, caos y muerte. La solución más simple (¿hemos mencionado que nosotros como soluciones simples?) es eliminar la redundancia en la clase misma. Si tienes quinientas líneas métodos con mucho código en común, puede 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 ya sea para extraer clase o extraer subclase . Un truco útil es determinar cómo los clientes usan el clase y utilizar la interfaz de extracción para cada uno de estos usos. Eso puede darte ideas sobre cómo puede romper aún más la clase.

Si su clase grande es una clase GUI, es posible que necesite mover datos y comportamiento a un dominio separado objeto. Esto puede requerir mantener algunos datos duplicados en ambos lugares y mantener los datos en sincronizar Los datos observados duplicados sugieren cómo hacer esto. En este caso, especialmente si eres usando componentes anteriores de Windows Resumen Toolkit (AWT), puede seguir esto quitando el Clase GUI y su sustitución por componentes Swing.

Lista larga de parámetros

En nuestros primeros días de programación, nos enseñaron a pasar como parámetros todo lo que un rutina. Esto era comprensible porque la alternativa eran los datos globales, y los datos globales son malos. y usualmente doloroso. Los objetos cambian esta situación porque si no tienes algo que necesitas, siempre puedes pedirle a otro objeto que te lo traiga. Así, con los objetos no pasas todo el método necesita; en su lugar, pasa lo suficiente para que el método pueda llegar a todo lo que necesita. UNA gran parte de lo que necesita un método está disponible en la clase de host del método. En programas orientados a objetos. Las listas de parámetros tienden a ser mucho más pequeñas que en los programas tradicionales.

Esto es bueno porque las largas listas de parámetros son difíciles de entender, porque se convierten en inconsistente y difícil de usar, y porque los está cambiando para siempre ya que necesita más datos. La mayoría de los cambios se eliminan al pasar objetos porque es mucho más probable que necesite hacer solo un par de solicitudes para obtener un nuevo dato.

Use Reemplazar parámetro con método cuando pueda obtener los datos en un parámetro haciendo una solicitud de un objeto que ya conoces. Este objeto podría ser un campo o podría ser otro

Página 66

parámetro. Use Preserve Whole Object para tomar un montón de datos recogidos de un objeto y reemplácelo con el objeto mismo. Si tiene varios elementos de datos sin objeto lógico, use Introducir objeto de parámetro .

Hay una excepción importante para hacer estos cambios. Esto es cuando explícitamente no desea crear una dependencia del objeto llamado al objeto más grande. En esos casos desempacando Los datos y enviarlos como parámetros son razonables, pero preste atención al dolor involucrado. Si la lista de parámetros es demasiado larga o cambia con demasiada frecuencia, debe repensar su dependencia estructura.

Cambio divergente

Estructuramos nuestro software para facilitar el cambio; después de todo, el software está destinado a ser suave. Cuando nosotros hacer un cambio que queremos poder saltar a un solo punto claro en el sistema y hacer que el cambio. Cuando no puedes hacer esto, estás oliendo una de las dos pungencias estrechamente relacionadas.

El cambio divergente ocurre cuando una clase se cambia comúnmente de diferentes maneras para diferentes razones. Si miras una clase y dices: "Bueno, tendré que cambiar estos tres métodos cada cada vez que obtengo una nueva base de datos; Tengo que cambiar estos cuatro métodos cada vez que hay un nuevo instrumento financiero ", es probable que tenga una situación en la que dos objetos son mejores que uno. forma en que cada objeto se cambia solo como resultado de un tipo de cambio. Por supuesto, a menudo descubres esto solo después de haber agregado algunas bases de datos o instrumentos financieros. Cualquier cambio para manejar un la variación debería cambiar una sola clase, y todo el tipeo en la nueva clase debería expresar el variación. Para limpiar esto, identifica todo lo que cambia por una causa particular y usa Extrae la clase para ponerlos todos juntos.

Cirugía de escopeta

La cirugía de escopeta es similar al cambio divergente, pero es lo contrario. Lo hueles cuando cada vez realiza un tipo de cambio, tiene que hacer muchos pequeños cambios en muchas clases diferentes. Cuando los cambios están por todas partes, son difíciles de encontrar y es fácil pasar por alto un importante cambio.

En este caso, desea utilizar Move Method y Move Field para poner todos los cambios en un solo clase. Si ninguna clase actual parece un buen candidato, cree una. A menudo puedes usar la clase en línea para reunir un montón de comportamiento. Obtiene una pequeña dosis de cambio divergente, pero puede lidiar fácilmente con eso.

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

Característica de la envidia

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 que no sea la en realidad está adentro. El foco más común de la envidia son los datos. Hemos perdido la cuenta de los tiempos 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 otro lugar, así que usa Move Method para llegar allí. A veces, solo una parte del método sufre de envidia; en eso use el Método de extracción en la parte celosa y el Método de mudanza para darle la casa de sus sueños.

Por supuesto, no todos los casos se cortan y secan. A menudo, un método utiliza características de varias clases, por lo que ¿con cuál debería vivir? La heurística que utilizamos es determinar qué clase tiene la mayor parte de datos y poner el método con esos datos. Este paso a menudo se hace más fácil si se usa el Método de extracción para romper el método en pedazos que van a diferentes lugares.

Por supuesto, hay varios patrones sofisticados que rompen esta regla. De la banda de los cuatro [Pandilla de los cuatro] La estrategia y el visitante saltan inmediatamente a la mente. Autodelegación de Kent Beck [Beck] es otro. Los usas para combatir el olor de cambio divergente. La regla de oro fundamental es juntar cosas que cambian juntas. Datos y el comportamiento que hace referencia a esos datos. Por lo general, cambian juntos, pero hay excepciones. Cuando ocurren las excepciones, movemos el comportamiento para mantener los cambios en un solo lugar. Strategy y Visitor te permiten cambiar el comportamiento fácilmente, porque aíslan la pequeña cantidad de comportamiento que debe anularse, a costa de Más indirección.

Grupos de datos

Los elementos de datos tienden a ser como los niños; les gusta andar 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. Grupos de datos que se juntan realmente deberían para convertirse en su propio objeto. El primer paso es buscar dónde aparecen los grupos como campos. Use Extract Class en los campos para convertir los grupos en un objeto. Luego dirija su atención a firmas de método que utilizan Introducir objeto de parámetro o Conservar objeto completo para adelgazar ellos abajo. El beneficio inmediato es que puede reducir muchas listas de parámetros y simplificar Llamada al método. No se preocupe por los grupos de datos que usan solo algunos de los campos del nuevo objeto. Mientras 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, ¿harían los otros? ¿cualquier sentido? Si no lo hacen, es una señal segura de que tienes un objeto que muere por nacer.

Reducir las listas de campos y las listas de parámetros ciertamente eliminará algunos malos olores, pero una vez que tenga los objetos, tienes la oportunidad de hacer un buen perfume. Ahora puede buscar casos de presentan envidia, lo que sugerirá un comportamiento que se puede trasladar a sus nuevas clases. Pronto 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 registro le permiten estructurar datos en grupos significativos. Los tipos primitivos son sus bloques de construcción. Los registros siempre llevan un cierta cantidad de gastos generales. Pueden significar tablas en una base de datos, o pueden ser incómodas crea cuando quieras para solo una o dos cosas.

Una de las cosas valiosas de los objetos es que difuminan o incluso rompen la línea entre lo primitivo y clases más grandes. Puede escribir fácilmente pequeñas clases que no se pueden distinguir de las integradas Tipos de lenguaje. Java tiene primitivas para números, pero cadenas y fechas, que son primitivas en muchos otros entornos, son clases.

Las personas nuevas en los objetos generalmente son reacias a usar objetos pequeños para tareas pequeñas, como el dinero clases que combinan número y moneda, rangos con mayúsculas y minúsculas, y especiales cadenas como números de teléfono y códigos postales. Puedes salir de la cueva al mundo de objetos con calefacción central mediante el uso de Reemplazar valor de datos con objeto en datos individuales valores. Si el valor de los datos es un código de tipo, use Reemplazar código de tipo con clase si el valor lo hace No afecta el comportamiento. Si tiene condicionales que dependen del código de tipo, use Reemplazar tipo Código con subclases o Reemplazar código de tipo con estado / estrategia .

Si tiene un grupo de campos que deberían ir juntos, use Extraer clase . Si ves estos primitivas en las listas de parámetros, pruebe una dosis civilizadora de Introducir objeto de parámetro . Si tu encuentras usted mismo separando una matriz, use Reemplazar matriz con objeto .

Cambiar declaraciones

Uno de los síntomas más obvios del código orientado a objetos es su falta comparativa de cambio (o caso) declaraciones. El problema con las declaraciones de cambio es esencialmente el de la duplicación. A menudo tu encuentre la misma declaración de interruptor dispersa sobre un programa en diferentes lugares. Si agrega un nuevo cláusula para el cambio, debe encontrar todos estos cambios, declaraciones y cambiarlos. El objeto- La noción orientada del polimorfismo le brinda una forma elegante de tratar este problema.

La mayoría de las veces que ve una declaración de cambio debería considerar el polimorfismo. El problema es dónde El polimorfismo debe ocurrir. A menudo, la instrucción switch activa un código de tipo. Usted quiere El método o clase que aloja el valor del código de tipo. Entonces use el método de extracción para extraer el interruptor y luego Mover método para llevarlo a la clase donde se necesita el polimorfismo. A ese punto tiene que decidir si reemplazar el código de tipo con subclases o reemplazar Código de tipo con estado / estrategia . Cuando haya configurado la estructura de herencia, puede usar Reemplazar condicional con polimorfismo .

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

Jerarquías de herencia paralelas

Las jerarquías de herencia paralelas son realmente un caso especial de cirugía de escopeta. En este caso, cada cada vez que crea una subclase de una clase, también debe crear una subclase de otra. Usted puede reconocer este olor porque los prefijos de los nombres de clase 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 uno jerarquía se refiere a instancias de la otra. Si usa Move Method y Move Field , la jerarquía en la clase de referencia desaparece.

Clase perezosa

Cada clase que creas cuesta dinero para mantener y comprender. Una clase que no está haciendo lo suficiente. pagar por sí mismo debe ser eliminado. A menudo, esta podría ser una clase que solía pagar pero tiene reducido con refactorización. O podría ser una clase que se agregó debido a cambios que fueron planeados pero no hechos. De cualquier manera, dejas que la clase muera con dignidad. Si tienes subclases que no están haciendo lo suficiente, intente usar Collapse Hierarchy . Los componentes casi inútiles deberían ser sometido 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 necesitamos la capacidad de hacer este tipo de cosas algún día" y por eso queremos todo tipo de ganchos y estuches especiales para manejar cosas que no son necesarias. El resultado a menudo es 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 se interpone en el camino, así que deshazte de ella.

Si tiene clases abstractas que no están haciendo mucho, use Collapse Hierarchy . Innecesario
la delegación se puede eliminar con la clase en línea . Los métodos con parámetros no utilizados deben ser
sujeto a Eliminar parámetro . Se deben traer métodos nombrados con nombres abstractos impares
Bajar a la tierra con el método Rename .

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

Campo temporal

A veces ve un objeto en el que una variable de instancia se establece solo en ciertas circunstancias.
Este código es difícil de entender, porque espera que un objeto necesite todas sus variables.
Intentar entender por qué una variable está allí cuando parece que no se usa puede volverte loco.

Use Extract Class para crear un hogar para las variables huérfanas pobres. Pon todo el código que concierne
Las variables en el componente. También puede eliminar el código condicional utilizando
Introduzca 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.
Debido a que el implementador no quería pasar una gran lista de parámetros (¿quién lo hace?), Puso
ellos en los campos. Pero los campos son válidos solo durante el algoritmo; en otros contextos son solo
Claramente confuso. En este caso, puede usar *Extract Class* con estas variables y los métodos que
exigirlos. El nuevo objeto es un objeto de método [Beck].

Cadenas de mensajes

Usted ve cadenas de mensajes cuando un cliente le pregunta a un objeto por otro objeto, que el cliente luego
pide otro objeto más, que luego el cliente pide otro objeto más, y así sucesivamente.
Puede ver esto como una larga línea de métodos getThis, o como una secuencia de temperaturas. Navegando esto
way significa que el cliente está acoplado a la estructura de la navegación. Cualquier cambio al intermedio
las relaciones hacen que el cliente tenga que cambiar.

El movimiento para usar aquí es Ocultar delegado . Puede hacer esto en varios puntos de la cadena. En
En principio, puede hacer esto a cada objeto de la cadena, pero hacerlo a menudo convierte cada intermedio
objetar en un intermediario. A menudo, una mejor alternativa es ver para qué se utiliza el objeto resultante.
Vea si puede usar el método de extracción para tomar una parte del código que lo usa y luego
Mover Método para empujarlo hacia abajo de la cadena. Si varios clientes de uno de los objetos de la cadena quieren
para 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 calma,
moderación razonada Bueno, al menos en este caso lo somos.

Hombre medio

Una de las características principales de los objetos es la encapsulación, que oculta los detalles internos del resto de los objetos.
mundo. La encapsulación a menudo viene con delegación. Le preguntas a un director si ella es libre para un
reunión; ella delega el mensaje en su diario y le da una respuesta. Todo bien y bien.
No es necesario saber si el director usa un diario, un artilugio electrónico o una secretaria para
realizar un seguimiento de sus citas.

Sin embargo, esto puede ir demasiado lejos. Miras la interfaz de una clase y encuentras que la mitad de los métodos son delegando a esta otra clase. Después de un tiempo, es hora de usar Remove Middle Man y hablar con el objeto que realmente sabe lo que está pasando. Si solo unos pocos métodos no están haciendo mucho, use Inline Método para incorporarlos a la persona que llama. Si hay un comportamiento adicional, puede usar Reemplazar Delegación con herencia para convertir al intermediario en una subclase del objeto real. Ese le permite extender el comportamiento sin perseguir toda esa delegación.

Intimidad inapropiada

A veces las clases se vuelven demasiado íntimas y pasan demasiado tiempo profundizando en cada una. partes privadas de otros. Puede que no seamos prudentes cuando se trata de personas, pero creemos que nuestras clases debe seguir reglas estrictas y puritanas.

Las clases demasiado íntimas deben dividirse como los amantes de la antigüedad. Usar método de movimiento y Mover campo para separar las piezas para reducir la intimidad. Vea si puede organizar un Cambie Asociación bidireccional a Unidireccional . Si las clases tienen común intereses, use Extract Class para poner la comunidad en un lugar seguro y hacer clases honestas de ellos. O use Hide Delegate para dejar que otra clase actúe como intermediario.

La herencia a menudo puede conducir a una intimidad excesiva. Las subclases siempre van a saber más sobre sus padres de lo que a ellos les gustaría que supieran. Si es hora de salir de casa, aplique Reemplazar Delegación con herencia .

Clases alternativas con diferentes interfaces

Use el método Rename en cualquier método que haga lo mismo pero que tenga firmas diferentes para lo que hacen. A menudo esto no llega lo suficientemente lejos. En estos casos las clases aún no están suficiente. Siga usando el Método Move para mover el comportamiento a las clases hasta que los protocolos sean mismo. Si tiene que mover código de forma redundante para lograr esto, puede usar Extract Superclase para expiar.

Clase de biblioteca incompleta

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

Los constructores de clases de biblioteca rara vez son omniscientes. No los culpamos por eso; después de todo, podemos rara vez descubrimos un diseño hasta que lo hemos construido en su mayoría, por lo que los creadores de bibliotecas tienen un trabajo realmente difícil. los El problema es que a menudo es una mala forma, y generalmente imposible, modificar una clase de biblioteca para hacer algo que te gustaría que hiciera. Esto significa que las tácticas tales como verdaderos probada y método Move mentira inútil.

Tenemos un par de herramientas especiales para este trabajo. Si solo hay un par de métodos que Si desea que la clase de la biblioteca tenga, utilice Introducir método extranjero . Si hay una carga completa de extra comportamiento, necesita Introducir extensión local .

Clase de datos

Estas son clases que tienen campos, métodos de obtención y configuración para los campos, y nada más. Dichas clases son titulares de datos tontos y casi seguramente se están manipulando en demasiadas

detalle por otras clases. En las primeras etapas, estas clases pueden tener campos públicos. Si es así, deberías aplique inmediatamente el campo Encapsulado antes de que alguien lo note. Si tiene campos de recopilación, marque para ver si están correctamente encapsulados y aplicar Encapsulate Collection si no lo están.
Use el Método de configuración de eliminación en cualquier campo que no deba cambiarse.

Busque dónde otras clases utilizan estos métodos de obtención y configuración. Intenta usar Move Método para mover el comportamiento a la clase de datos. Si no puede mover un método completo, use Extraer Método para crear un método que se pueda mover. Después de un tiempo, puede comenzar a usar el Método Ocultar en los captadores y colocadores.

Las clases de datos son como los niños. Están bien como punto de partida, pero para participar como adultos objetar, necesitan asumir cierta responsabilidad.

Legado rechazado

Las subclases heredan los métodos y datos de sus padres. Pero, ¿y si no quieren o necesita lo que se les da? Se les dan todos estos grandes regalos y eligen solo algunos para jugar.

La historia tradicional es que esto significa que la jerarquía está mal. Necesitas crear un nuevo hermano clase y utilizar Empuje hacia abajo Método y empuje hacia abajo Campo de empujar todos los métodos utilizados a el hermano. De esa manera, el padre tiene solo lo que es común. A menudo escuchará consejos de que todos Las superclases deben ser abstractas.

Adivinará por nuestro uso sarcástico de lo *tradicional* que no vamos a aconsejar esto, al menos no todos hora. Hacemos subclases para reutilizar un poco de comportamiento todo el tiempo, y nos parece una forma perfectamente buena de hacer negocios. Hay un olor, no podemos negarlo, pero generalmente no es un olor fuerte. Entonces decimos que si el legado rechazado está causando confusión y problemas, siga los consejos tradicionales.
Sin embargo, no sienta que tiene que hacerlo todo el tiempo. Nueve de cada diez veces este olor es demasiado débil para ser Vale la pena limpiarlo.

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

Comentarios

No se preocupe, no estamos diciendo que la gente no deba escribir comentarios. En nuestra analogía olfativa, los comentarios no son mal olor; de hecho son un olor dulce. La razón por la que mencionamos comentarios Aquí es que los comentarios a menudo se usan como desodorante. Es sorprendente la frecuencia con la que miras con atención código comentado y observe que los comentarios están ahí porque el código es malo.

Los comentarios nos llevan a un código incorrecto que tiene todas las fallas podridas que hemos discutido en el resto de este capítulo. Nuestra primera acción es eliminar los malos olores refactorizando. Cuando terminamos, a menudo encuentra que los comentarios son superfluos.

Si necesita un comentario para explicar lo que hace un bloque de código, pruebe el Método de extracción . Si el método ya se extrajo, pero aún necesita un comentario para explicar lo que hace, use el método Rename .
Si necesita establecer algunas reglas sobre el estado requerido del sistema, use Introducir Afirmación .

Propina

Cuando sienta la necesidad de escribir un comentario, primero intente refactorizar el código para que El comentario se vuelve superfluo.

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

Capítulo 4. Pruebas de construcción

Si desea refactorizar, la condición previa esencial es tener pruebas sólidas. Incluso si eres afortunado suficiente para tener una herramienta que pueda automatizar las refactorizaciones, aún necesita pruebas. Será 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 contraintuitivo para muchos programadores, por lo que vale la pena explicar por qué.

El valor del código de autocomprobación

Si observa cómo la mayoría de los programadores pasan su tiempo, encontrará que escribir código en realidad es una fracción bastante pequeña Se pasa algún tiempo averiguando qué debería estar pasando, algún tiempo es pasó diseñando, pero la mayoría del tiempo se dedica a la depuración. Estoy seguro de que cada lector puede recordar mucho horas de depuración, a menudo hasta bien entrada la noche. Cada programador puede contar una historia de un error que se llevó un día entero (o más) para encontrar. Solucionar el error suele ser bastante rápido, pero encontrarlo es una pesadilla. Y luego, cuando arreglas un error, siempre existe la posibilidad de que aparezca otro y que Puede que ni lo note hasta mucho más tarde. Luego pasas años buscando ese error.

El evento que me inició en el camino hacia el código de autoevaluación fue una charla en OOPSLA en el '92. Alguien (Creo que fue Dave Thomas) dijo sin rodeos: "Las clases deben contener sus propias pruebas". Ese Me pareció una buena forma de organizar las pruebas. Interpreté eso como diciendo que cada clase debería tiene su propio método (llamado *prueba*) que puede usarse para probarse a sí mismo.

En ese momento también estaba en desarrollo incremental, así que intenté agregar métodos de prueba a las clases como Completé cada incremento. El proyecto en el que estaba trabajando en ese momento era bastante pequeño, así que sacamos incrementos cada semana más o menos. Ejecutar las pruebas se volvió bastante sencillo, pero aunque eran fáciles de ejecutar, las pruebas aún eran bastante aburridas. Esto fue porque cada La prueba produjo una salida a la consola que tuve que verificar. Ahora soy una persona bastante perezosa y soy preparado para trabajar bastante duro para evitar el trabajo. Me di cuenta de que en lugar de mirar la pantalla para ver si imprime alguna información del modelo, podría hacer que la computadora haga eso prueba. Todo lo que tenía que hacer era poner la salida que esperaba en el código de prueba y hacer una comparación. Ahora yo podría ejecutar el método de prueba de cada clase, y simplemente imprimiría "OK" en la pantalla si todo estuviera bien. los la clase ahora se autocomprobaba.

Propina

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

Ahora era fácil ejecutar una prueba, tan fácil como compilar. Entonces comencé a hacer pruebas cada vez que compilado Pronto comencé a notar que mi productividad se había disparado. Me di cuenta de que no era dedicando mucho tiempo a la depuración. Si agrego un error que fue detectado por una prueba anterior, sería aparecer tan pronto como ejecuté esa prueba. Como la prueba había funcionado antes, sabría que El error estaba en el trabajo que había hecho desde la última prueba. Debido a que ejecutaba las pruebas con frecuencia, solo unas pocas Los minutos habían transcurrido. Por lo tanto, sabía 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 pequeña cantidad, el error fue fácil de encontrar. Loco que una vez había tardado una hora o más en encontrar, ahora tomó un par de minutos como máximo. No solo si yo construí clases de autoevaluación, pero al ejecutarlas con frecuencia tuve un potente detector de errores.

Cuando noté esto, me volví más agresivo al hacer las pruebas. En lugar de esperar el final de incremento, agregaría las pruebas inmediatamente después de escribir un poco de función. Todos los días agregaría

Un par de nuevas características y las pruebas para probarlas. En estos días casi nunca paso más de un unos minutos de depuración

Propina

Un conjunto de pruebas es un potente detector de errores que decapita el tiempo que lleva encontrar loco.

Por supuesto, no es tan fácil persuadir a otros para que sigan esta ruta. Escribir las pruebas es mucho código extra para escribir. A menos que haya experimentado la forma en que acelera la programación, las pruebas no parecen tener sentido. Esto no es ayudado por el hecho de que muchas personas nunca han aprendí a escribir exámenes o incluso a pensar en exámenes. Cuando las pruebas son manuales, son intestinales. desgarradoramente aburrido. Pero cuando son automáticos, las pruebas pueden ser bastante divertidas de escribir.

De hecho, uno de los momentos más útiles para escribir pruebas es antes de comenzar a programar. Cuando tú necesita agregar una función, comience escribiendo la prueba. Esto no es tan atrasado como parece. Al escribir el prueba se pregunta qué debe hacerse para agregar la función. Escribir el examen también se concentra en la interfaz en lugar de la implementación (siempre es algo bueno). También significa 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 sueltos. Pero programadores extremos son probadores muy dedicados. Quieren desarrollar software lo más rápido posible, y saben que Las pruebas lo ayudan a ir lo más rápido posible.

Eso es suficiente de la polémica. Aunque creo que todos se beneficiarían escribiendo autoevaluaciones código, no es el punto de este libro. Este libro trata sobre la refactorización. La refactorización requiere pruebas. Si quieres refactorizar, tienes que escribir pruebas. Este capítulo le ofrece un comienzo para hacer esto para Java. Este no es un libro de prueba, así que no voy a entrar en muchos detalles. Pero con las pruebas he encontrado que un Una cantidad notablemente pequeña puede tener beneficios sorprendentemente grandes.

Como con todo lo demás en este libro, describo el enfoque de prueba usando ejemplos. Cuando yo desarrollar código, escribo las pruebas a medida que avanzo. Pero a menudo, cuando estoy trabajando con personas en la refactorización, nosotros tener un cuerpo de código que no sea de autoevaluación para trabajar. Entonces, primero tenemos que hacer que el código se pruebe por sí mismo antes de refactorizar.

El idioma estándar de Java para las pruebas es la prueba principal. La idea es que cada clase debería tener un función principal que prueba la clase. Es una convención razonable (aunque no se honra mucho), pero Puede volverse incómodo. El problema es que tal convención hace que sea difícil ejecutar muchas pruebas fácilmente. Otro enfoque es construir clases de prueba separadas que funcionen en un marco para hacer Prueba más fácil.

El marco de prueba JUnit

El marco de prueba que uso es JUnit, un marco de prueba de código abierto desarrollado por Erich Gamma y Kent Beck [JUnit]. El marco es muy simple, pero te permite hacer todas las claves cosas que necesitas para probar. En este capítulo utilizo este marco para desarrollar pruebas para algunos io clases

Página 75

Para comenzar, 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 prueba. El marco utiliza el compuesto patrón [Gang of Four] que le permite agrupar pruebas en suites (Figura 4.1). Estas suites pueden

contener los casos de prueba sin procesar u otros conjuntos de casos de prueba. Esto facilita la creación de una gama de grandes conjuntos de pruebas y ejecutar las pruebas de forma automática.

Figura 4.1. La estructura compuesta de las pruebas.

```
La 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 un código de prueba. Mi primer trabajo es configurar el dispositivo de prueba. Un accesorio de prueba es 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
Pollock                 60,97          23     41     4 4    2256        274         7 7
Headley                 60,83          22     40     4 4    2256        270 *       10
Sutcliffe               60,73          54     84     9 9    4555        194         dieciséis
```

Para seguir usando el archivo, preparo el accesorio. La clase de caso de prueba proporciona dos métodos para manipule el dispositivo de prueba: `setUp` crea los objetos y `tearDown los` elimina. Ambos son

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

```
   clase FileReaderTester ...
        configuración vacía protegida () {
             tratar {
                  _input = new FileReader ("data.txt");
             } catch (FileNotFoundException e) {
                  lanzar una nueva RuntimeException ("no se puede abrir el archivo de prueba");
             }
        }

        desgarro vacío protegido () {
             tratar {
                  _input.close ();
             } catch (IOException e) {
                  lanzar nueva RuntimeException ("error al cerrar la prueba
archivo");
             }
        }
```

Ahora que tengo el dispositivo de prueba en su lugar, puedo comenzar a escribir pruebas. Lo primero es probar la lectura.
método. Para hacer esto, leo algunos caracteres y luego verifico que el siguiente que leí es el
el correcto:

```
        public void testRead () arroja 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 otra manera
Señalamos un error. Muestro cómo el marco hace eso más tarde. 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 () {
             TestSuite suite = nuevo TestSuite ();
             suite.addTest (nuevo FileReaderTester ("testRead"));
             suite de retorno;
        }
```

Este conjunto de pruebas contiene solo un objeto de caso de prueba, una instancia de `FileReaderTester.` Cuando yo
crear un caso de prueba, le doy al constructor un argumento de cadena, que es el nombre del método que soy
va a probar Esto crea un objeto que prueba ese método. La prueba está ligada al objeto.
a través de la capacidad de reflexión de Java. Puede echar un vistazo al código fuente descargado para calcular
a ver cómo lo hace. Solo lo trato como magia.

Page 77

Para ejecutar las pruebas, use una clase TestRunner separada. Hay dos versiones de TestRunner: una
utiliza una GUI genial, la otra una interfaz de caracteres simple. Puedo llamar a la versión de interfaz de personaje
en general:

```
clase FileReaderTester ...
    public static void main (String [] args) {
        junit.textui.TestRunner.run (suite ());
}
```

El código crea el corredor de prueba y le dice que pruebe la clase FileReaderTester. Cuando lo corro veo

```
.
Tiempo: 0.110

OK (1 pruebas)
```

JUnit imprime un período para cada prueba que se ejecuta (para que pueda ver el progreso). Te dice cuánto tiempo Las pruebas han corrido. Luego dice "OK" si nada sale mal y le dice cuántas pruebas han sido correr. Puedo ejecutar mil pruebas, y si todo va bien, lo veré bien. Esta simple retroalimentación es esencial para el código de autocomprobación. Sin él, nunca ejecutará las pruebas con la frecuencia suficiente. Con ella puedes realice una gran cantidad de pruebas, salga a almorzar (o una reunión) y vea los resultados cuando regrese.

Propina

Ejecute sus pruebas con frecuencia. Localice las pruebas siempre que compile, cada prueba al menos cada día.

Al refactorizar, ejecuta solo unas pocas pruebas para ejercitar el código en el que está trabajando. Usted puede corre solo unos pocos porque deben ser rápidos: de lo contrario, te ralentizarán y te sentirás tentado no para ejecutarlos. No ceder a esa tentación-retribución *se* siga.

¿Qué ocurre si algo va mal? Lo demostraré poniendo un error deliberado, como sigue:

```
public void testRead () arroja 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
Hora: 0.220

!!! FALLAS !!!
```



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

El marco me alerta sobre la falla y me dice qué prueba falló. El mensaje de error no es

particularmente útil, sin embargo. Puedo mejorar el mensaje de error utilizando otra forma de afirmación.

```
public void testRead () arroja IOException {
     char ch = '&';
     para (int i = 0; i <4; i ++)
          ch = (char) _input.read ();
     ClaimEquals ('m', ch);
}
```

La mayoría de las afirmaciones que haces están comparando dos valores para ver si son iguales. Entonces el El marco incluye `afirmar iguales.` Esto es conveniente; usa `equals ()` en objetos y == en valores, que a menudo olvido hacer. También permite un mensaje de error más significativo:

```
.F
Tiempo: 0.170

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

Debo mencionar que, a menudo, cuando escribo pruebas, empiezo haciéndolas fallar. Con el código existente I o cámbielo para que falle (si puedo tocar el código) o ponga un valor esperado incorrecto en el afirmar. Hago esto porque me gusta probarme a mí mismo que la prueba realmente se ejecuta y la prueba es en realidad probando lo que se supone que debe hacer (por eso prefiero cambiar el código probado si puedo). Esto puede ser paranoia, pero realmente puede confundirse cuando las pruebas prueban algo diferente de lo que crees que están probando.

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

```
public void testRead () arroja IOException {
     char ch = '&';
     _input.close ();
     para (int i = 0; i <4; i ++)
          ch = (char) _input.read ();                // lanzará una excepción
     ClaimEquals ('m', ch);
}
```

Si ejecuto esto me sale

Página 79

```
.MI

Tiempo: 0.110

!!! FALLAS !!!
Resultados de la prueba:
Ejecutar: 1 Fallos: 0 Errores: 1
Hubo 1 error:
```

```
1) FileReaderTester.testRead
java.io.IOException: flujo cerrado
```

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

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

Figura 4.2. La interfaz gráfica de uso r de JUnit

Pruebas unitarias y funcionales

Este marco se utiliza para pruebas unitarias, por lo que debería mencionar la diferencia entre pruebas unitarias y
pruebas funcionales Las pruebas de las que hablo son *pruebas unitarias.* Los escribo para mejorar mi productividad
como programador Hacer feliz al departamento de garantía de calidad es solo un efecto secundario. Pruebas unitarias
están altamente localizados Cada clase de prueba funciona dentro de un solo paquete. Prueba las interfaces a otros
paquetes, pero más allá de eso supone que el resto simplemente funciona.

Las pruebas funcionales son 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.
Deben ser desarrollados por un equipo diferente, uno que se deleita en encontrar errores. Este equipo usa
herramientas y técnicas pesadas para ayudarlos a hacer esto.

Las pruebas funcionales generalmente tratan todo el sistema como una caja negra tanto como sea posible. En una GUI
sistema basado, operan a través de la GUI. En un programa de actualización de archivos o bases de datos, las pruebas solo
mire cómo se cambian los datos para ciertas entradas.

Cuando los probadores funcionales, o los usuarios, encuentran un error en el software, se necesitan al menos dos cosas para solucionarlo
eso. Por supuesto, debe cambiar el código de producción para eliminar el error. Pero también deberías agregar
Una prueba unitaria que expone el error. De hecho, cuando recibo un informe de error, comienzo escribiendo una prueba unitaria que
hace que el error salga a la superficie. Escribo más de una prueba si necesito reducir el alcance del error, o si
Puede haber fallas relacionadas. Utilizo las pruebas unitarias para ayudar a localizar el error y asegurar que

un error similar no supera las pruebas de mi unidad nuevamente.

Propina

Cuando obtenga 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 a menudo se realizan con otras herramientas. Las herramientas de prueba basadas en GUI son buenos ejemplos. A menudo, sin embargo, escribirás tu propio herramientas de prueba específicas de la aplicación que facilitan la gestión de casos de prueba que las secuencias de comandos GUI solas. Puede realizar pruebas funcionales con JUnit, pero generalmente no es la forma más eficiente. por Para fines de refactorización, cuento con las pruebas unitarias: el amigo del programador.

Agregar más pruebas

Ahora deberíamos continuar agregando más pruebas. El estilo que sigo es mirar todas las cosas que la clase debe hacer y probar cada uno de ellos en busca de cualquier condición que pueda causar que la clase falle. Esto es no es lo mismo que "probar todos los métodos públicos", que defienden algunos programadores. Las pruebas deberían ser conducido por el riesgo; recuerde, está tratando de encontrar errores ahora o en el futuro. Entonces no pruebo accesores que solo leen y escriben un campo. Debido a que son tan simples, no es probable que encuentre un error ahí.

Esto es importante porque intentar escribir demasiadas pruebas generalmente lleva a no escribir lo suficiente. He a menudo leo libros sobre pruebas, y mi reacción ha sido alejarme de la montaña de cosas que Hay que hacer para probar. Esto es contraproducente, porque te hace pensar que para probar tienes que hacer mucho trabajo Obtiene muchos beneficios de las pruebas, incluso si solo realiza una pequeña prueba. La clave es para probar las áreas que más le preocupan que salgan mal. De esa manera obtienes el mayor beneficio por tu esfuerzo de prueba.

Propina

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 devuelve `-1` al final del archivo (no es un protocolo muy agradable en mi opinión, pero supongo que eso lo hace más natural para programadores en C). Probémoslo. Mi editor de texto me dice que hay 141 caracteres en el archivo, así que aquí está la prueba:

Página 81

```
public void testReadAtEnd () lanza IOException {
   int ch = -1234;
   para (int i = 0; i <141; i ++)
         ch = _input.read ();
   ClaimEquals (-1, ch);
}
```

Para ejecutar la prueba, debo agregarla a la suite:

```
conjunto de pruebas estáticas públicas () {
   TestSuite suite = nuevo TestSuite ();
   suite.addTest (nuevo FileReaderTester ("testRead"));
   suite.addTest (nuevo FileReaderTester ("testReadAtEnd"));
   suite de retorno;
}
```

Cuando se ejecuta esta suite, le dice a cada una de sus pruebas de componentes (los dos casos de prueba) que se ejecuten. Cada prueba case ejecuta setUp, el cuerpo del código de prueba en el método de prueba, y finalmente tearDown. Está
Es importante ejecutar setUp y tearDown cada vez para que las pruebas estén aisladas unas de otras.
Eso significa que podemos ejecutarlos en cualquier orden y no importa.

Es difícil recordar agregar las pruebas al método de la suite. Afortunadamente, Erich Gamma y Kent
Beck es tan vago como yo, por lo que proporcionaron una forma de evitar eso. Un constructor especial para el
Test Suite toma una clase como parámetro.

Este constructor crea un conjunto de pruebas que contiene un caso de prueba para cada método que comienza con *prueba de* palabras Si sigo esa convención, puedo reemplazar mi main con

```
public static void main (String [] args) {
   junit.textui.TestRunner.run (nuevo TestSuite (FileReaderTester.class));
}
```

De esa manera, cada prueba que escribo se agrega a la suite.

Un truco clave con las pruebas es buscar condiciones de contorno. Para la lectura, los límites serían
primer carácter, el último carácter y el carácter después del último carácter:

```
public void testReadBoundaries () throwsIOException {
      ClaimEquals ("read first char", 'B', _input.read ());
      int ch;
      para (int i = 1; i <140; i ++)
            ch = _input.read ();
      ClaimEquals ("leer el último carácter", '6', _ input.read ());
      ClaimEquals ("leer al final", - 1, _input.read ());
   }
```

Tenga en cuenta que puede agregar un mensaje a la afirmación que se imprime si la prueba falla.

Propina

Page 82

Piense en las condiciones de contorno bajo las cuales las cosas pueden salir mal y concéntrese tus pruebas allí.

Otra parte de la búsqueda de límites es buscar condiciones especiales que puedan hacer que la prueba fallar. Para los archivos, los archivos vacíos son siempre una buena opción:

```
testEmptyRead () vacío público arroja IOException {
```

```
    Archivo vacío = archivo nuevo ("empty.txt");
    FileOutputStream out = nuevo FileOutputStream (vacío);
    out.close ();
    FileReader en = nuevo FileReader (vacío);
    ClaimEquals (-1, in.read ());
  }
```

En este caso, estoy creando un accesorio adicional solo para esta prueba. Si necesito un archivo vacío para más tarde, puedo muévelo a un dispositivo normal moviendo el código a la configuración.

```
configuración vacía protegida () {
   tratar {
       _input = new FileReader ("data.txt");
       _empty = newEmptyFile ();
   } catch (IOException e) {
        lanzar nueva RuntimeException (e.toString ());
   }
}

privado FileReader newEmptyFile () lanza IOException {
   Archivo vacío = archivo nuevo ("empty.txt");
   FileOutputStream out = nuevo FileOutputStream (vacío);
   out.close ();
   return newFileReader (vacío);
}

testEmptyRead () vacío público arroja IOException {
   afirmarEquals (-1, _empty.read ());
}
```

¿Qué sucede si lees después del final del archivo? De nuevo `-1` debería ser devuelto, y aumento una de las otras pruebas para probar que:

```
      public void testReadBoundaries () throwsIOException {
        ClaimEquals ("read first char", 'B', _input.read ());
        int ch;
        para (int i = 1; i <140; i ++)
              ch = _input.read ();
        ClaimEquals ("read last char", '6', _input.read ());
        ClaimEquals ("leer al final", - 1, _input.read ());
        ClaimEquals ("readpast end", -1, _input.read ());
   }
```


Page 83

Observe cómo estoy jugando el papel de un enemigo para codificar. Estoy pensando activamente en cómo puedo romper eso. Creo que ese estado mental es productivo y divertido. Se entrega a la parte mezquina de mi Psique.

Cuando realice pruebas, no olvide comprobar que los errores esperados se producen correctamente. Si intentas leer una secuencia después de que se cierra, debe obtener una IOException. Esto también debe ser probado:

```
  public void testReadAfterClose () throwsIOException {
    _input.close ();
    tratar {
           _input.read ();
```

```
        fail ("no hay excepción para leer pasado final");
    } catch (IOException io) {}
}
```

Cualquier otra excepción que no sea la IOException producirá un error de la manera normal.

Propina

No olvide probar que se producen excepciones cuando se espera que las cosas salgan mal.

Desarrollar las pruebas continúa en esta línea. Lleva un tiempo pasar por la interfaz para algunas clases para hacer esto, pero en el proceso realmente entiendes la interfaz de la clase. En particular, ayuda pensar en las condiciones de error y las condiciones de contorno. Esa es otra ventaja para escribir pruebas mientras escribe código, o incluso antes de escribir el código de producción.

A medida que agrega más clases de probadores, puede crear otras clases de probadores que combinen suites de Clases múltiples. Esto es fácil de hacer porque un conjunto de pruebas puede contener otros conjuntos de pruebas. Así tu puede tener una clase de prueba maestra:

```
clase MasterTester extiende TestCase {
  public static void main (String [] args) {
       junit.textui.TestRunner.run (suite ());
  }
  conjunto de pruebas estáticas públicas () {
       Resultado TestSuite = nuevo TestSuite ();
       result.addTest (nuevo TestSuite (FileReaderTester.class));
       result.addTest (nuevo TestSuite (FileWriterTester.class));
       // y así...
       resultado de retorno;
  }
}
```

¿Cuándo te detienes? Estoy seguro de que ha escuchado muchas veces que no puede probar que un programa no tiene errores de prueba. Eso es cierto, pero no afecta la capacidad de las pruebas para acelerar la programación. He visto varias propuestas de reglas para garantizar que haya probado todas las combinaciones de todo. Sus Vale la pena echarles un vistazo, pero no dejes que te afecten. Hay un punto de rendimientos decrecientes con las pruebas, y existe el peligro de que al intentar escribir demasiadas pruebas, te conviertas desanimados y terminan no escribiendo ninguno. Debes concentrarte en dónde está el riesgo. Mira el

codifique y vea dónde se vuelve complejo. Mire la función y considere las áreas probables de error. Sus pruebas no encontrarán todos los errores, pero a medida que refactorice, comprenderá el programa mejor y así encontrar más errores. Aunque siempre comienzo a refactorizar con un conjunto de pruebas, invariablemente añadir a medida que avanzo.

Propina

No deje que el miedo a que las pruebas no puedan atrapar todos los errores le impida escribir las pruebas que atrapará la mayoría de los errores.

Una de las cosas difíciles sobre los objetos es que la herencia y el polimorfismo pueden hacer que las pruebas más difícil, porque hay muchas combinaciones para probar. Si tienes tres clases abstractas que colaborar y cada uno tiene tres subclases, tienes nueve alternativas pero veintisiete combinaciones. No siempre trato de probar todas las combinaciones posibles, pero intento probar cada alternativa. Se reduce al riesgo en las combinaciones. Si las alternativas son razonablemente independientes el uno del otro, no es probable que pruebe cada combinación. Siempre existe el riesgo de que yo perder algo, pero es mejor pasar un tiempo razonable para atrapar la mayoría de los errores que gastar edades tratando de atraparlos a todos.

Una diferencia entre el código de prueba y el código de producción es que está bien copiar y editar el código de prueba. Cuando trato con combinaciones y alternativas, a menudo hago eso. Primero tome "evento de pago regular" ahora tome "antigüedad" y "discapacitados antes de fin de año". Ahora hazlo sin "antigüedad" y "desactivado antes de fin de año", y así sucesivamente. Con alternativas simples como esa encima de un accesorio razonable, puedo generar pruebas muy rápidamente. Entonces puedo usar la refactorización para factorizar verdaderamente artículos comunes más tarde.

Espero haberte dado una idea de cómo escribir exámenes. Puedo decir mucho más sobre este tema, pero eso sería oscurecer el mensaje clave. Construya un buen detector de errores y ejecútelo con frecuencia. Es una herramienta maravillosa para cualquier desarrollo y es una condición previa para la refactorización.

Capítulo 5. Hacia un catálogo de refactorizaciones

Los capítulos 5 a 12 forman un catálogo inicial de refactorizaciones. Han crecido de las notas que hice en refactorización en los últimos años. Este catálogo no es completo ni hermético, pero debería proporcionar un punto de partida sólido para su propio trabajo de refactorización.

Formato de las refactorizaciones

Cuando describo las refactorizaciones en este y otros capítulos, uso un formato estándar. Cada refactorización tiene cinco partes, como sigue:

- Comienzo con un **nombre.** El nombre es importante para construir un vocabulario de refactorizaciones. Esta es el nombre que uso en otra parte del libro.

- Sigo el nombre con un breve **resumen** de la situación en la que necesita la refactorización.
  y un resumen de lo que hace la refactorización. Esto te ayuda a encontrar una refactorización más
  con rapidez.
- La **motivación** describe por qué se debe realizar la refactorización y describe
  circunstancias en las que no se debe hacer.
- La **mecánica** es una descripción concisa, paso a paso, de cómo llevar a cabo
  refactorización.
- Los **ejemplos** muestran un uso muy simple de la refactorización para ilustrar cómo funciona.

El resumen incluye una breve descripción del problema con el que la refactorización le ayuda, un breve descripción de lo que haces, y un boceto que muestra un simple ejemplo de antes y después. A veces uso código para el boceto y, a veces, Unified Modeling Language (UML), dependiendo de cuál parece transmitir mejor la esencia de la refactorización. (Todos los diagramas UML en este libro se extrae de la perspectiva de implementación [Fowler, UML].) Si ha visto el refactorizando antes, el boceto debería darle una buena idea de qué se trata la refactorización. Si no probablemente necesitará trabajar con el ejemplo para tener una mejor idea.

La mecánica proviene de mis propias notas para recordar cómo hacer la refactorización cuando no he Lo hice por un tiempo. Como tales, son algo breves, generalmente sin explicaciones de por qué los pasos se hacen de esa manera. Doy explicaciones más amplias en el ejemplo. De esta manera el la mecánica son notas breves a las que puede referirse fácilmente cuando conoce la refactorización pero necesita mirar subir los pasos (al menos así es como los uso). Probablemente necesites leer el ejemplo cuando primero haga la refactorización.

He escrito la mecánica de tal manera que cada paso de cada refactorización es tan pequeño como posible. Destaco la forma segura de refactorizar, que consiste en dar pasos muy pequeños y prueba después de cada uno. En el trabajo, generalmente tomo pasos más grandes que algunos de los pasos de bebé descritos, pero si me encuentro con un error, retrocedo el paso y tomo los pasos más pequeños. Los pasos incluyen un número de referencias a casos especiales. Los pasos también funcionan como una lista de verificación; A menudo olvido estos cosas yo mismo.

Los ejemplos son del tipo de libro de texto ridículamente simple. Mi objetivo con el ejemplo es ayudar explique la refactorización básica con mínimas distracciones, así que espero que perdone la simplicidad. (Ellos ciertamente no son ejemplos de un buen diseño de objetos comerciales.) Estoy seguro de que podrá aplicarlos a tus situaciones más complejas. Algunas refactorizaciones muy simples no tienen ejemplos porque no pensé que un ejemplo agregaría mucho.

En particular, recuerde que los ejemplos se incluyen solo para ilustrar el que se refactoriza en discusión. En la mayoría de los casos, todavía hay problemas con el código al final, pero solucionarlos

Los problemas requieren otras refactorizaciones. En algunos casos en los que las refactorizaciones a menudo van juntas, yo llevar ejemplos de una refactorización a otra. En la mayoría de los casos, dejo el código tal como está después del refactorización individual. Hago esto para hacer que cada refactorización sea autónoma, porque la función principal de El catálogo es como referencia.

No tome ninguno de estos ejemplos como sugerencias sobre cómo diseñar objetos de pedido o empleados. Estos ejemplos están ahí solo para ilustrar las refactorizaciones, nada más. En particular Observe que en los ejemplos utilizo el `doble` para representar los valores monetarios. He hecho esto solo para simplifique los ejemplos, ya que la representación no es importante para la refactorización. Creo firmemente desaconsejar el uso de `dobles` por dinero en software comercial. Cuando represento dinero uso el patrón Cantidad [Fowler, AP].

Cuando estaba escribiendo este libro, Java 1.1; era la versión que más se usaba en comerciales trabajo. Por lo tanto, la mayoría de mis ejemplos usan Java 1.1; Esto es más notable en mi uso de colecciones. Cuando llegué al final del libro, Java 2 se hizo más disponible. No siento que sea necesario cambie todos los ejemplos, ya que las colecciones son secundarias a la refactorización. Sin embargo hay

algunas refactorizaciones, como Encapsulate Collection , que son diferentes en Java 2. En tales casos
He explicado los casos de Java 2 y Java 1.1.

Utilizo el **`código en negrita`** para resaltar el código modificado donde está enterrado entre el código que no tiene
ha cambiado y puede ser difícil de detectar. No uso negrita para todos los códigos modificados,
porque demasiado derrota el propósito.

Encontrar referencias

Muchas de las refactorizaciones requieren que encuentre todas las referencias a un método, un campo o una clase. Cuando
Si haces esto, alista la computadora para que te ayude. Al usar la computadora, reduce sus posibilidades de
falta una referencia y generalmente puede hacer la búsqueda mucho más rápido de lo que lo haría si fuera
simplemente mirar el código.

La mayoría de los idiomas tratan los programas de computadora como archivos de texto. Tu mejor ayuda aquí es un texto adecuado
buscar. Muchos entornos de programación le permiten buscar texto en un solo archivo o en un grupo de archivos.
El control de acceso de la función que está buscando le indicará el rango de archivos que necesita
buscar.

No solo busque y reemplace a ciegas. Inspeccione cada referencia para asegurarse de que realmente se refiere a la cosa
estás reemplazando Puedes ser inteligente con tu patrón de búsqueda, pero siempre reviso mentalmente
asegúrese de que estoy haciendo el reemplazo correcto. Si puede usar el mismo nombre de método en diferentes
clases o métodos de diferentes firmas en la misma clase, hay muchas posibilidades de que
Lo entenderé mal.

En un lenguaje fuertemente tipado, puede dejar que el compilador lo ayude a cazar. A menudo puedes
elimine la característica anterior y deje que el compilador encuentre las referencias colgantes. Lo bueno de
esto es que el compilador capturará cada referencia colgante. Sin embargo, hay problemas con esto
técnica.

Primero, el compilador se confundirá cuando una característica se declare más de una vez en un
jerarquía de herencia. Esto es particularmente cierto cuando está mirando un método que se anula
varias veces. Si está trabajando en una jerarquía, use la búsqueda de texto para ver si hay alguna otra
La clase declara el método que está manipulando.

El segundo problema es que el compilador puede ser demasiado lento para ser efectivo. Si es así, use una búsqueda de texto
primero; al menos el compilador verifica dos veces tu trabajo. Esto solo funciona cuando tiene la intención de eliminar

la característica. A menudo desea ver todos los usos para decidir qué hacer a continuación. En estos casos tu
tiene que usar la alternativa de búsqueda de texto.

Un tercer problema es que el compilador no puede detectar los usos de la API de reflexión. Esta es una razón para estar
cauteloso de usar la reflexión. Si su sistema usa reflexión, tendrá que usar búsquedas de texto para encontrar
cosas y poner peso adicional en sus pruebas. En varios lugares sugiero compilar
sin probar en situaciones en las que el compilador generalmente detecta errores. Si usa la reflexión, todos
tales apuestas están apagadas, y debe probar con muchas de estas compilaciones.

Algunos entornos Java, especialmente VisualAge de IBM, están siguiendo el ejemplo de Smalltalk
navegador. Con estos, usa las opciones del menú para buscar referencias en lugar de usar búsquedas de texto.
Estos entornos no usan archivos de texto para guardar el código; usan una base de datos en memoria. Obtener
acostumbrado a usar estos elementos de menú y los encontrará a menudo superiores al texto no disponible
buscar.

¿Qué tan maduras son estas refactorizaciones?

Cualquier autor técnico tiene el problema de decidir cuándo publicar. Cuanto antes publiques, el
Las personas más rápidas pueden aprovechar las ideas. Sin embargo, las personas siempre están aprendiendo. Si tu publique ideas a medias demasiado pronto, las ideas pueden estar incompletas e incluso generar problemas para aquellos que intentan usarlos.

La técnica básica de refactorización, tomar pequeños pasos y probar a menudo, ha sido bien probada durante muchos años, especialmente en la comunidad Smalltalk. Así que estoy seguro de que la idea básica de La refactorización es muy estable.

Las refactorizaciones en este libro son mis notas sobre las refactorizaciones que uso. Los he usado todos. Sin embargo, hay una diferencia entre usar una refactorización y reducirla a Pasos mecánicos que doy aquí. En particular, ocasionalmente ve problemas que surgen solo en Circunstancias muy específicas. No puedo decir que he tenido mucha gente trabajando desde estos pasos para detectar muchos de estos tipos de problemas. A medida que usa las refactorizaciones, tenga en cuenta lo que es haciendo. Recuerde que, como trabajar con una receta, debe adaptar las refactorizaciones a su circunstancias. Si se encuentra con un problema interesante, envíeme un correo electrónico e intentaré transmitirlo. Estas circunstancias para los demás.

Otro aspecto a recordar acerca de estas refactorizaciones es que se describen con un solo proceso de software en mente. Con el tiempo, espero ver las refactorizaciones descritas para su uso con concurrente y programación distribuida. Tales refactorizaciones serán diferentes. Por ejemplo, en un solo proceso software, nunca debe preocuparse con qué frecuencia llama a un método; Las llamadas al método son baratas. Con software distribuido, sin embargo, los viajes de ida y vuelta deben ser minimizados. Hay diferentes refactorizaciones para esos sabores de programación, pero esos son temas para otro libro.

Muchas de las refactorizaciones, como Reemplazar código de tipo con estado / estrategia y forma El Método de plantilla consiste en introducir patrones en un sistema. Como la pandilla esencial de los cuatro el libro dice: "Patrones de diseño ... proporcionan objetivos para sus refactorizaciones". Hay una relacion natural entre patrones y refactorizaciones. Los patrones están donde quieres estar; refactorizaciones son formas de llegar desde otro lugar. No tengo refactorizaciones para todos los patrones conocidos en este libro, no incluso para todos los patrones de Gang of Four [Gang of Four]. Este es otro aspecto de la incompletitud de este catálogo. Espero que algún día la brecha se cierre.

Cuando utilice las refactorizaciones, tenga en cuenta que son un punto de partida. Sin duda encontrarás huecos en ellos. Los estoy publicando ahora porque, aunque no son perfectos, creo que

Page 88

Son útiles. Creo que te darán un punto de partida que mejorará tu capacidad de refactorizar eficientemente. Eso es lo que hacen por mí.

A medida que use más refactorizaciones, espero que comience a desarrollar las suyas propias. Espero los ejemplos aquí lo motivará y le dará un punto de partida sobre cómo hacerlo. Soy consciente de la hecho de que hay muchas más refactorizaciones que las que describí. Si se te ocurre algunos, por favor envíeme un correo electrónico.

Page 89

Capítulo 6. Métodos de composición

Una gran parte de mi refactorización es componer métodos para empaquetar el código correctamente. Casi todos los tiempo los problemas provienen de métodos que son demasiado largos. Los métodos largos son problemáticos porque a menudo contienen mucha información, que queda oculta por la lógica compleja que generalmente se obtiene arrastrado hacia adentro. La refactorización clave es el método de extracción , que toma un grupo de código y lo convierte en Su propio método. El método en línea es esencialmente lo contrario. Toma una llamada al método y la reemplaza con el cuerpo del código. Necesito el método en línea cuando he realizado múltiples extracciones y me doy cuenta algunos de los métodos resultantes ya no están tirando de su peso o si necesito reorganizar el camino He desglosado los métodos.

El mayor problema con el método de extracción es tratar con variables locales, y las temperaturas son una de Las principales fuentes de este problema. Cuando estoy trabajando en un método, me gusta Reemplazar temp con Consulta para deshacerte de cualquier variable temporal que pueda eliminar. Si la temperatura se usa para muchas cosas, Primero uso la variable temporal dividida para hacer que la temperatura sea más fácil de reemplazar.

A veces, sin embargo, las variables temporales están demasiado enredadas para reemplazarlas. Necesito reemplazar Método con Método Objeto . Esto me permite romper incluso el método más enredado, en el

costo de introducir una nueva clase para el trabajo.

Los parámetros son menos problemáticos que los temporales, siempre que no los asigne. Si lo haces, tu necesita eliminar asignaciones a parámetros .

Una vez que el método se desglosa, puedo entender cómo funciona mucho mejor. También puedo encontrar que El algoritmo se puede mejorar para hacerlo más claro. Luego uso Algoritmo sustituto para presentar El algoritmo más claro.

Método de extracción

Tiene un fragmento de código que se puede agrupar.

Convierta el fragmento en un método cuyo nombre explique el propósito del método.

```
vacío printOwing (cantidad doble) {
     printBanner ();

     // imprimir detalles
     System.out.println ("nombre:" + _nombre);
     System.out.println ("cantidad" + cantidad);
}
```

```
vacío printOwing (cantidad doble) {
     printBanner ();
     printDetails (cantidad);
}

vacío printDetails (cantidad doble) {
```


Página 90

```
     System.out.println ("nombre:" + _nombre);
     System.out.println ("cantidad" + cantidad);
}
```

Motivación

El método de extracción es una de las refactorizaciones más comunes que hago. Miro un método que es demasiado largo o mire el código que necesita un comentario para comprender su propósito. Luego giro ese fragmento de código en su propio método.

Prefiero métodos cortos y bien nombrados por varias razones. Primero, aumenta las posibilidades de que otros Los métodos pueden usar un método cuando el método es de grano fino. En segundo lugar, permite el nivel superior métodos para leer más como una serie de comentarios. Anular también es más fácil cuando los métodos son De grano fino.

Lleva un poco acostumbrarse si está acostumbrado a ver métodos más grandes. Y pequeños métodos realmente funciona solo cuando tienes buenos nombres, por lo que debes prestar atención a los nombres. Personas a veces me preguntan qué longitud busco en un método. Para mí la longitud no es el problema. La clave es

La distancia semántica entre el nombre del método y el cuerpo del método. Si la extracción mejora claridad, hazlo, incluso si el nombre es más largo que el código que has extraído.

Mecánica

- Cree un nuevo método y asígnele el nombre según la intención del método (asígnele el nombre lo hace, no por cómo lo hace).

  ? *Si el código que desea extraer es muy simple, como un solo mensaje o llamada de función, debe extraerlo si el nombre del nuevo El método revelará la intención del código de una mejor manera. Si no puedes proponga un nombre más significativo, no extraiga el código.*

- Copie el código extraído del método de origen en el nuevo método de destino.
- Escanee el código extraído en busca de referencias a cualquier variable que tenga un alcance local para método fuente Estas son variables locales y parámetros del método.
- Vea si alguna variable temporal se usa solo dentro de este código extraído. Si es así, declararlos en el método de destino como variables temporales.
- Observe si alguna de estas variables de ámbito local se modifica por el extraído código. Si se modifica una variable, vea si puede tratar el código extraído como una consulta y asigne el resultado a la variable en cuestión. Si esto es incómodo, o si hay más de una de esas variables, no puede extraer el método tal como está. Es posible que deba usar Split Variable temporal e intente nuevamente. Puede eliminar variables temporales con Reemplace Temp con Query (vea la discusión en los ejemplos).
- Pase al método de destino como parámetros variables de ámbito local que se leen desde código extraído
- Compile cuando haya tratado con todas las variables de ámbito local.
- Reemplace el código extraído en el método de origen con una llamada al método de destino.

  ? *Si ha movido alguna variable temporal al método de destino, mire para ver si se declararon fuera del código extraído. Si entonces, ahora puede eliminar la declaración.*

Page 91

- Compilar y probar.

Ejemplo: sin variables locales

En el caso más simple, el método de extracción es trivialmente fácil. Toma el siguiente método:

```
vacío printOwing () {

     Enumeración e = _orders.elements ();
     doble pendiente = 0.0;

     // imprimir banner
     System.out.println ("**************************");
     System.out.println ("***** El cliente debe ******");
     System.out.println ("**************************");

     // calcular pendiente
     while (e.hasMoreElements ()) {
          Ordenar cada = (Ordenar) e.nextElement ();
          sobresaliente + = each.getAmount ();
     }
```

```
        // imprimir detalles
        System.out.println ("nombre:" + _nombre);
        System.out.println ("cantidad" + pendiente);
    }
```

Es fácil extraer el código que imprime el banner. Acabo de cortar, pegar y hacer una llamada:

```
    vacío printOwing () {

        Enumeración e = _orders.elements ();
        doble pendiente = 0.0;

        printBanner ();

        // calcular pendiente
        while (e.hasMoreElements ()) {
            Ordenar cada = (Ordenar) e.nextElement ();
            sobresaliente + = each.getAmount ();
        }

        // imprimir detalles
        System.out.println ("nombre:" + _nombre);
        System.out.println ("cantidad" + pendiente);
    }

    vacío printBanner () {
        // imprimir banner
        System.out.println ("**************************");
        System.out.println ("***** El cliente debe ******");
```


Página 92

```
        System.out.println ("**************************");
    }
```

Ejemplo: uso de variables locales

¿Entonces, cuál es el problema? El problema son las variables locales: parámetros pasados al original método y temporales declarados dentro del método original. Las variables locales solo tienen alcance en ese método, así que cuando uso el método de extracción , estas variables me causan trabajo extra. En algunos incluso me impiden hacer la refactorización.

El caso más fácil con las variables locales es cuando las variables se leen pero no se modifican. En este caso Solo puedo pasarlos como un parámetro. Entonces, si tengo el siguiente método:

```
    vacío printOwing () {

        Enumeración e = _orders.elements ();
        doble pendiente = 0.0;

        printBanner ();
```

```
        // calcular pendiente
        while (e.hasMoreElements ()) {
             Ordenar cada = (Ordenar) e.nextElement ();
             sobresaliente + = each.getAmount ();
        }

        // imprimir detalles
        System.out.println ("nombre:" + _nombre);
        System.out.println ("cantidad" + pendiente);
    }
```

Puedo extraer la impresión de detalles con un método con un parámetro:

```
    vacío printOwing () {

        Enumeración e = _orders.elements ();
        doble pendiente = 0.0;

        printBanner ();

        // calcular pendiente
        while (e.hasMoreElements ()) {
             Ordenar cada = (Ordenar) e.nextElement ();
             sobresaliente + = each.getAmount ();
        }

        printDetails (sobresaliente);
    }
```


Página 93

```
    vacío printDetails (doble pendiente) {
        System.out.println ("nombre:" + _nombre);
        System.out.println ("cantidad" + pendiente);
    }
```

Puede usar esto con tantas variables locales como necesite.

Lo mismo es cierto si la variable local es un objeto e invoca un método de modificación en el variable. Una vez más, puede pasar el objeto como parámetro. Solo tienes que hacer algo diferente si realmente asigna a la variable local.

Ejemplo: reasignación de una variable local

Es la asignación a variables locales lo que se vuelve complicado. En este caso solo estamos hablando sobre las temperaturas Si ve una asignación a un parámetro, debe usar Eliminar inmediatamente Asignaciones a parámetros .

Para las temperaturas asignadas, hay dos casos. El caso más simple es aquel en el que La variable es una variable temporal que se usa solo dentro del código extraído. Cuando eso sucede, puedes mueve la temperatura al código extraído. El otro caso es el uso de la variable fuera del código. Si la variable no se utiliza después de extraer el código, puede hacer el cambio solo en el extraído código. Si se usa después, debe hacer que el código extraído devuelva el valor modificado de

variable. Puedo ilustrar esto con el siguiente método:

```
vacío printOwing () {

     Enumeración e = _orders.elements ();
     doble pendiente = 0.0;

     printBanner ();

     // calcular pendiente
     while (e.hasMoreElements ()) {
          Ordenar cada = (Ordenar) e.nextElement ();
          sobresaliente + = each.getAmount ();
     }

     printDetails (sobresaliente);
}
```

Ahora extraigo el cálculo:

```
vacío printOwing () {
     printBanner ();
     doble pendiente = getOutstanding ();
     printDetails (sobresaliente);
}
```


Page 94

```
doble getOutstanding () {
     Enumeración e = _orders.elements ();
     doble pendiente = 0.0;
     while (e.hasMoreElements ()) {
          Ordenar cada = (Ordenar) e.nextElement ();
          sobresaliente + = each.getAmount ();
     }
     retorno sobresaliente;
}
```

La variable de enumeración se usa solo en el código extraído, por lo que puedo moverla completamente dentro del Nuevo método. La variable sobresaliente se usa en ambos lugares, por lo que necesito volver a ejecutarla desde método extraído Una vez que he compilado y probado la extracción, cambio el nombre del valor devuelto para seguir mi convención habitual:

```
doble getOutstanding () {
     Enumeración e = _orders.elements ();
     doble resultado = 0.0;
     while (e.hasMoreElements ()) {
          Ordenar cada = (Ordenar) e.nextElement ();
          resultado = each.getAmount ();
     }
     resultado de retorno ;
```

```
}
```

En este caso, la variable pendiente se inicializa solo a un valor inicial obvio, por lo que puedo inicializar solo dentro del método extraído. Si algo más involucrado le sucede a la variable, tengo que Pase el valor anterior como parámetro. El código inicial para esta variación podría verse así:

```
vacío printOwing (doble anteriorAmount) {

     Enumeración e = _orders.elements ();
     doble pendiente = previoAmount * 1.2;

     printBanner ();

     // calcular pendiente
     while (e.hasMoreElements ()) {
          Ordenar cada = (Ordenar) e.nextElement ();
          sobresaliente + = each.getAmount ();
     }

     printDetails (sobresaliente);
}
```

En este caso, la extracción se vería así:

Page 95

```
vacío printOwing (doble anteriorAmount) {
     doble pendiente = previoAmount * 1.2;
     printBanner ();
     pendiente = getOutstanding (pendiente);
     printDetails (sobresaliente);
}

double getOutstanding (double initialValue) {
     resultado doble = valor inicial;
     Enumeración e = _orders.elements ();
     while (e.hasMoreElements ()) {
          Ordenar cada = (Ordenar) e.nextElement ();
          resultado + = each.getAmount ();
     }
     resultado de retorno;
}
```

Después de compilar y probar esto, aclaro la forma en que se inicializa la variable pendiente:

```
vacío printOwing (doble anteriorAmount) {
     printBanner ();
     doble pendiente = getOutstanding (previousAmount * 1.2);
     printDetails (sobresaliente);
```

```
}
```

En este punto puede que se pregunte: "¿Qué sucede si se necesita más de una variable?
regresado?

Aquí tienes varias opciones. La mejor opción generalmente es elegir un código diferente para extraer. yo prefiero un método para devolver un valor, por lo que trataría de organizar varios métodos para los diferentes valores. (Si su idioma permite parámetros de salida, puede usarlos. Prefiero usar solo devolver valores tanto como sea posible)

Las variables temporales a menudo son tan abundantes que hacen que la extracción sea muy incómoda. En estos casos Intento reducir las temperaturas utilizando Reemplazar temperatura con consulta . Si todo lo que hago las cosas son aún incómodo, recurro a Reemplazar Método con Método Objeto . A esta refactorización no le importa cuántos temporarios tienes o qué haces con ellos.

Método en línea

El cuerpo de un método es tan claro como su nombre.

Coloque el cuerpo del método en el cuerpo de sus llamantes y elimine el método.

```
int getRating () {
     return (moreThanFiveLateDeliveries ())? 2: 1;
}
```



```
boolean moreThanFiveLateDeliveries () {
     return _numberOfLateDeliveries> 5;
}
```

```
int getRating () {
     return (_numberOfLateDeliveries> 5)? 2: 1;
}
```

Motivación

Un tema de este libro es usar métodos cortos nombrados para mostrar su intención, porque estos Los métodos conducen a un código más claro y más fácil de leer. Pero a veces te encuentras con un método en que el cuerpo es tan claro como el nombre. O refactoriza el cuerpo del código en algo que es tan claro como el nombre. Cuando esto sucede, debe deshacerse del método. La indirecta puede ser útil, pero la indirecta innecesaria es irritante.

Otro momento para usar el método en línea es cuando tiene un grupo de métodos que parecen mal factorizado Puede incorporarlos a todos en un gran método y luego volver a extraer los métodos. Kent Beck encuentra que a menudo es bueno hacer esto antes de usar Reemplazar Método con Método Objeto . Usted en línea las diversas llamadas realizadas por el método que tienen el comportamiento que desea tener en el objeto del método. Es más fácil mover un método que mover el método y sus métodos llamados.

Comúnmente uso el método en línea cuando alguien usa demasiada indirección y parece que cada método hace una delegación simple a otro método, y me pierdo en toda la delegación. En En estos casos, parte de la indirección vale la pena, pero no toda. Al tratar de alinearme puedo salir los útiles y eliminar el resto.

Mecánica

- Verifique que el método no sea polimórfico.

> ? *No en línea si las subclases anulan el método; no pueden anular un método que no está ahí.*

- Encuentra todas las llamadas al método.
- Reemplace cada llamada con el cuerpo del método.
- Compilar y probar.
- Eliminar la definición del método.

Escrito de esta manera, el método en línea es simple. En general no lo es. Podría escribir páginas sobre cómo manejar la recursividad, múltiples puntos de retorno, alinearse en otro objeto cuando no tiene accesorios y similares. La razón por la que no lo hago es que si te encuentras con estas complejidades, No debería hacer esta refactorización.

Temperatura en línea

Tiene una temperatura que se asigna una vez con una expresión simple, y la temperatura se está poniendo en forma de otras refactorizaciones.

Reemplace todas las referencias a esa temperatura con la expresión.

```
double basePrice = anOrder.basePrice ();
return (basePrice> 1000)
```

```
return (anOrder.basePrice ()> 1000)
```

Motivación

La mayoría de las veces, *Inline Temp* se usa como parte de Reemplazar temp con consulta , por lo que La motivación está ahí. El único momento en que *Inline Temp* se usa por sí solo es cuando encuentra una temperatura que es asignado el valor de una llamada al método. A menudo, esta temperatura no está haciendo ningún daño y puedes déjalo ahí. Si la temperatura se interpone en el camino de otras refactorizaciones, como el Método de extracción , es hora de ponerlo en línea.

Mecánica

- Declare la temperatura como final si aún no está, y compile.

? *Esto comprueba que la temperatura solo se asigna realmente una vez.*

- Encuentre todas las referencias a la temperatura y reemplácelas con el lado derecho del asignación.
- Compilar y probar después de cada cambio.
- Eliminar la declaración y la asignación de la temp.
- Compilar y probar.

Reemplazar temp con consulta

Está utilizando una variable temporal para contener el resultado de una expresión.

Extrae la expresión en un método. Reemplace todas las referencias a la temperatura con la expresión. los El nuevo método se puede utilizar en otros métodos.

```
double basePrice = _quantity * _itemPrice;
if (precio base> 1000)
     volver basePrecio * 0.95;
más
     volver basePrecio * 0.98;
```



```
if (basePrice ()> 1000)
     return basePrice () * 0.95;
más
     return basePrice () * 0.98;
...
precio base doble () {
     return _quantity * _itemPrice;
}
```

Motivación

El problema con las temperaturas es que son temporales y locales. Porque solo se pueden ver en En el contexto del método en el que se utilizan, las temperaturas tienden a fomentar métodos más largos, porque esa es la única forma en que puedes alcanzar la temperatura. Al reemplazar la temperatura con un método de consulta, Cualquier método en la clase puede obtener la información. Eso ayuda mucho a encontrar un limpiador código para la clase.

Reemplazar Temp con Query a menudo es un paso vital antes del Método de extracción . Las variables locales lo hacen difícil de extraer, así que reemplace tantas variables como sea posible con consultas.

Los casos directos de esta refactorización son aquellos en los que las temperaturas se asignan solo una vez y aquellos en los que la expresión que genera la asignación está libre de efectos secundarios. Otro Los casos son más complicados pero posibles. Es posible que deba usar dividir variable temporal o separar Consulta desde Modifier primero para facilitar las cosas. Si la temperatura se usa para recopilar un resultado (como sumando en un bucle), necesita copiar algo de lógica en el método de consulta.

Mecánica

Aquí está el caso simple:

- Busque una variable temporal asignada a una vez.

  ? *Si se establece una temperatura más de una vez, considere* dividir la variable temporal .

- Declarar la temperatura como final.
- Compilar.

  ? *Esto asegurará que la temperatura solo se asigne una vez.*

- Extraiga el lado derecho de la tarea en un método.

  ? *Inicialmente marque el método como privado. Puede encontrar más uso para él más tarde, pero puedes relajar fácilmente la protección más tarde.*

  ? *Asegúrese de que el método extraído no tenga efectos secundarios, es decir, no Modificar cualquier objeto. Si no está libre de efectos secundarios, use la* Consulta separada del modificador .

- Compilar y probar.
- Use Reemplazar temp con consulta en la temp.

Los tiempos a menudo se usan para almacenar información resumida en bucles. Se puede extraer todo el ciclo. en un método; Esto elimina varias líneas de código ruidoso. A veces se puede usar un ciclo para sumar valores múltiples, como en el ejemplo de la página 26. En este caso, duplique el ciclo para cada temp para que pueda reemplazar cada temporal con una consulta. El bucle debe ser muy simple, por lo que hay poco peligro al duplicar el código.

Puede que le preocupe el rendimiento en este caso. Al igual que con otros problemas de rendimiento, déjelo deslizarse por el momento. Nueve de cada diez veces, no importará. Cuando importa, arreglará el problema durante la optimización. Con su código mejor factorizado, a menudo encontrará más poderoso optimizaciones, que te habrías perdido sin refactorizar. Si lo peor llega a ser peor, es Muy fácil volver a poner la temperatura.

Ejemplo

Comienzo con un método simple:

```
doble getPrice () {
     int basePrice = _quantity * _itemPrice;
     doble discountFactor;
     if (basePrice> 1000) discountFactor = 0.95;
     else discountFactor = 0.98;
     return basePrice * discountFactor;
}
```

Me inclino a reemplazar ambas temperaturas, una a la vez.

Aunque está bastante claro en este caso, puedo probar que solo se asignan una vez declarando ellos como final:

```
doble getPrice () {
     final int basePrice = _quantity * _itemPrice;
     doble descuento final Factor;
     if (basePrice> 1000) discountFactor = 0.95;
     else discountFactor = 0.98;
     return basePrice * discountFactor;
}
```

Compilar me alertará sobre cualquier problema. Primero hago esto, porque si hay un problema, yo no debería estar haciendo esta refactorización. Reemplazo las temperaturas una a la vez. Primero extraigo la mano derecha lado de la tarea:

```
doble getPrice () {
     final int basePrice = basePrice ();
     doble descuento final Factor;
```


Página 100

```
     if (basePrice> 1000) discountFactor = 0.95;
     else discountFactor = 0.98;
     return basePrice * discountFactor;
}

private int basePrice () {
return _quantity * _itemPrice;
}
```

Compilo y pruebo, luego comienzo con Reemplazar temp con consulta . Primero reemplazo el primero referencia a la temperatura:

```
doble getPrice () {
     final int basePrice = basePrice ();
     doble descuento final Factor;
     if ( basePrice () > 1000) discountFactor = 0.95;
     else discountFactor = 0.98;
     return basePrice * discountFactor;
}
```

Compila, prueba y haz lo siguiente (suena como una persona que llama en un baile en línea). Porque es el último, yo También elimine la declaración temporal:

```
doble getPrice () {
     doble descuento final Factor;
     if (basePrice ()> 1000) discountFactor = 0.95;
     else discountFactor = 0.98;
```

```
        return basePrice () * discountFactor;
    }
```

Con eso desaparecido, puedo extraer discountFactor de manera similar:

```
    doble getPrice () {
        doble double discountFactor = discountFactor ();
        return basePrice () * discountFactor;
    }

    private double discountFactor () {
        if (basePrice ()> 1000) devuelve 0,95;
        de lo contrario, devuelve 0,98;
    }
```

Vea cómo hubiera sido difícil extraer `discountFactor` si no hubiera reemplazado `basePrice` con una consulta.

Page 101

El método `getPrice` termina de la siguiente manera:

```
    doble getPrice () {
        return basePrice () * discountFactor ();
    }
```

Introducir explicando la variable

Tienes una expresión complicada.

Ponga el resultado de la expresión, o partes de la expresión, en una variable temporal con un nombre

eso explica el propósito.

```
  if ((platform.toUpperCase (). indexOf ("MAC")> -1) &&
       (browser.toUpperCase (). indexOf ("IE")> -1) &&
        wasInitialized () && resize> 0)
{
       // hacer algo
}
```

```
    boolean final isMacOs = platform.toUpperCase (). indexOf ("MAC")>
-1;
    boolean final isIEBrowser = browser.toUpperCase (). indexOf ("IE")>
-1;
    final booleano wasResized = redimensionar> 0;
```

```
if (isMacOs && isIEBrowser && wasInitialized () && wasResized) {
    // hacer algo
}
```

Motivación

Las expresiones pueden volverse muy complejas y difíciles de leer. En tales situaciones variables temporales Puede ser útil dividir la expresión en algo más manejable.

Introducir Explicar variable es particularmente valioso con la lógica condicional en la que es útil tome cada cláusula de una condición y explique qué significa la condición con una temp. Otro caso es un algoritmo largo, en el que cada paso en el cálculo se puede explicar con un temperatura

Introducir Explicar variable es una refactorización muy común, pero confieso que no lo uso tanto. Casi siempre prefiero usar el método de extracción si puedo. Una temperatura es útil solo dentro del contexto de Un método Un método es utilizable en todo el objeto y para otros objetos. Hay veces,

sin embargo, cuando las variables locales dificultan el uso del Método de extracción . Ahí es cuando uso Introducir explicando la variable .

Mecánica

- Declare una variable temporal final y configúrela como el resultado de parte del complejo. expresión.
- Reemplace la parte del resultado de la expresión con el valor de la temp.

> ? *Si la parte del resultado de la expresión se repite, puede reemplazar el Se repite uno a la vez.*

- Compilar y probar.
- Repita para otras partes de la expresión.

Ejemplo

Comienzo con un cálculo simple:

```
precio doble () {
// el precio es el precio base - descuento por cantidad + envío
return _quantity * _itemPrice -
     Math.max (0, _cantidad - 500) * _itemPrice * 0.05 +
     Math.min (_cantidad * _itemPrice * 0.1, 100.0);
}
```

Puede ser simple, pero puedo hacerlo más fácil de seguir. Primero identifico el precio base como la cantidad veces el precio del artículo. Puedo convertir esa parte del cálculo en una temperatura:

```
precio doble () {
     // el precio es el precio base - descuento por cantidad + envío
     precio base doble final = _cantidad * _itemPrice;
     volver basePrecio -
          Math.max (0, _cantidad - 500) * _itemPrice * 0.05 +
          Math.min (_cantidad * _itemPrice * 0.1, 100.0);
}
```

La cantidad por el precio del artículo también se usa más tarde, por lo que también puedo sustituirla por la temperatura allí:

```
precio doble () {
     // el precio es el precio base - descuento por cantidad + envío
     precio base doble final = _cantidad * _itemPrice;
     volver basePrecio -
          Math.max (0, _cantidad - 500) * _itemPrice * 0.05 +
          Math.min ( basePrice * 0.1, 100.0);
}
```


Page 103

A continuación tomo el descuento por cantidad:

```
precio doble () {
    // el precio es el precio base - descuento por cantidad + envío
    precio base doble final = _cantidad * _itemPrice;
    cantidad doble finalDiscount = Math.max (0, _quantity - 500) *
_itemPrice * 0.05;
    volver basePrecio - cantidadDiscount +
        Math.min (basePrice * 0.1, 100.0);
}
```

Finalmente, termino con el envío. Mientras hago eso, también puedo eliminar el comentario, porque ahora no dice nada el código no dice:

```
precio doble () {
    precio base doble final = _cantidad * _itemPrice;
    cantidad doble finalDiscount = Math.max (0, _quantity - 500) *
_itemPrice * 0.05;
    envío doble final = Math.min (basePrice * 0.1, 100.0);
    base de devoluciónPrecio - cantidadDiscount + envío;
}
```

Ejemplo con método de extracción

Para este ejemplo, generalmente no habría hecho las explicaciones temporales; Preferiría hacer eso con Extraer método . Empiezo de nuevo con

```
precio doble () {
    // el precio es el precio base - descuento por cantidad + envío
    return _quantity * _itemPrice -
        Math.max (0, _cantidad - 500) * _itemPrice * 0.05 +
        Math.min (_cantidad * _itemPrice * 0.1, 100.0);
}
```

pero esta vez extraigo un método para el precio base:

```
precio doble () {
     // el precio es el precio base - descuento por cantidad + envío
     return basePrice () -
          Math.max (0, _cantidad - 500) * _itemPrice * 0.05 +
          Math.min ( basePrice () * 0.1, 100.0);
```


Página 104

```
}

base doble privada Precio () {
    return _quantity * _itemPrice;
}
```

Continúo uno a la vez. Cuando termine me sale

```
precio doble () {
     return basePrice () - cantidadDiscount () + envío ();
}

Private double amountDiscount () {
     return Math.max (0, _quantity - 500) * _itemPrice * 0.05;
}

envío privado doble () {
     return Math.min (basePrice () * 0.1, 100.0);
}

base doble privada Precio () {
     return _quantity * _itemPrice;
}
```

Prefiero usar el método de extracción , porque ahora estos métodos están disponibles para cualquier otra parte de El objeto que los necesita. Inicialmente los hago privados, pero siempre puedo relajarme si otro El objeto los necesita. Creo que generalmente no es más esfuerzo usar el Método de extracción que usar Introducir explicando la variable .

Entonces, ¿cuándo uso Introducir variable explicativa ? La respuesta es cuando el método de extracción es más esfuerzo. Si estoy en un algoritmo con muchas variables locales, es posible que no pueda usar fácilmente Extraer método . En este caso utilizo Introducir variable explicativa para ayudarme a comprender Que esta pasando. A medida que la lógica se vuelve menos enredada, siempre puedo usar Reemplazar temp con

Consulta luego. La temperatura también es valiosa si termino teniendo que usar Reemplazar Método con Método Objeto .

Variable temporal dividida

Tiene una variable temporal asignada a más de una vez, pero no es una variable de bucle ni un recolectando variable temporal.

Haga una variable temporal separada para cada tarea.

```
doble temp = 2 * (_ altura + _ ancho);
System.out.println (temp);
temp = _height * _width;
System.out.println (temp);
```



```
doble perímetro final = 2 * (_ altura + _ ancho);
System.out.println (perímetro);
área doble final = _ altura * _ ancho;
System.out.println (área);
```

Motivación

Las variables temporales se hacen para varios usos. Algunos de estos usos conducen naturalmente a la temperatura ser asignado a varias veces. Las variables de bucle [Beck] cambian para cada ejecución alrededor de un bucle (como como `i` en `for ( int i = 0; i <10; i ++ )`. Recolectando variables temporales [Beck] recolectan juntas algún valor que se acumula durante el método.

Muchos otros temporales se usan para contener el resultado de un bit de código largo para facilitar referencia más tarde. Este tipo de variables debe establecerse solo una vez. Que se establecen más que una vez es una señal de que tienen más de una responsabilidad dentro del método. Cualquier variable con Se debe reemplazar más de una responsabilidad por una temperatura para cada responsabilidad. Usando una temperatura porque dos cosas diferentes son muy confusas para el lector.

Mecánica

- Cambiar el nombre de un tempo en su declaración y su primera asignación.

  ? *Si las asignaciones posteriores tienen la forma i = i + alguna expresión, eso indica que es una variable temporal de recolección, así que no la divida. los El operador para una variable temporal de recopilación generalmente es suma, cadena concatenación, escribir en una secuencia o agregar a una colección.*

- Declarar la nueva temperatura como final.
- Cambie todas las referencias de la temperatura hasta su segunda asignación.
- Declare la temperatura en su segunda asignación.
- Compilar y probar.

• Repita en etapas, cambiando el nombre de cada etapa en la declaración y cambiando las referencias hasta La próxima tarea.

Ejemplo

Para este ejemplo calculo la distancia recorrida por un haggis. Desde un principio, un haggis
Experimenta una fuerza inicial. Después de un período retrasado, una fuerza secundaria entra en acción para acelerar aún más los haggis Usando las leyes comunes del movimiento, puedo calcular la distancia recorrida de la siguiente manera:

```
  double getDistanceTravelled (int time) {
       doble resultado;
       doble acc = _primaryForce / _mass;
       int primaryTime = Math.min (tiempo, _delay);
       resultado = 0.5 * acc * primaryTime * primaryTime;
       intdaryTime = time - _delay;
```


Page 106

```
       if (secondTime> 0) {
            double primaryVel = acc * _delay;
            acc = (_primaryForce + _secondaryForce) / _mass;
            resultado + = primaryVel *daryTime + 0.5 * acc *
secundariosTiempo * secundariosTiempo;
       }
       resultado de retorno;
  }
```

Una pequeña y bonita función incómoda. Lo interesante para nuestro ejemplo es la forma en que la variable `acc` es Establecer dos veces. Tiene dos responsabilidades: una para mantener la aceleración inicial causada por la primera fuerza y otro más tarde para mantener la aceleración con ambas fuerzas. Esto lo quiero dividir.

Comienzo al principio cambiando el nombre de la temperatura y declarando el nuevo nombre como final. Luego cambio todas las referencias a la temperatura desde ese punto hasta la siguiente tarea. En el proximo asignación lo declaro:

```
    double getDistanceTravelled (int time) {
         doble resultado;
         final     doble primaria Accc     = _primaryForce / _mass;
         int primaryTime = Math.min (tiempo, _delay);
         resultado = 0.5 * primaryAcc * primaryTime * primaryTime;
         intdaryTime = time - _delay;
         if (secondTime> 0) {
              double primaryVel = primaryAcc * _delay;
              doble acc = (_primaryForce + _secondaryForce) / _mass;
              resultado + = primaryVel *daryTime + 0.5 * acc *
secundariosTiempo * secundariosTiempo;
         }
         resultado de retorno;
    }
```

Elijo el nuevo nombre para representar solo el primer uso de la temp. Lo hago definitivo para asegurarme de que sea solo se configura una vez. Entonces puedo declarar la temperatura original en su segunda asignación. Ahora puedo compilar y prueba, y todo debería funcionar.

Continúo en la segunda asignación de la temp. Esto elimina el nombre temporal original completamente, reemplazándolo con una nueva temperatura nombrada para el segundo uso.

```
    double getDistanceTravelled (int time) {
         doble resultado;
         final double primaryAcc = _primaryForce / _mass;
         int primaryTime = Math.min (tiempo, _delay);
         resultado = 0.5 * primaryAcc * primaryTime * primaryTime;
         intdaryTime = time - _delay;
         if (secondTime> 0) {
              double primaryVel = primaryAcc * _delay;
              final de doble secondaryAcc = (_primaryForce +
_secondaryForce) / _mass;
```


Page 107

```
              resultado + = primaryVel *daryTime + 0.5 *
                            daryAcc *daryTime *daryTime;
         }
         resultado de retorno;
    }
```

Estoy seguro de que puedes pensar en muchas más refactorizaciones para hacer aquí. Disfrútala. (Estoy seguro de que es mejor que comiendo los haggis, ¿sabes lo que ponen en esas cosas?)

Eliminar asignaciones a parámetros

El código se asigna a un parámetro.

Use una variable temporal en su lugar.

```
     descuento int (int inputVal, int cantidad, int yearToDate) {
          if (inputVal> 50) inputVal - = 2;
```

```
     descuento int (int inputVal, int cantidad, int yearToDate) {
          int resultado = inputVal;
          if (inputVal> 50) resultado - = 2;
```

Motivación

Primero permítame asegurarme de que tengamos clara la frase "asigna a un parámetro". Esto significa que si pasa un objeto llamado *foo,* en el parámetro, asignar al parámetro significa cambiar *foo* para referirse a un objeto diferente. No tengo problemas para hacerle algo al objeto que estaba aprobada en; Lo hago todo el tiempo. Solo me opongo a cambiar *foo* para referirme a otro objeto por completo:

```
   nula aMethod (Object foo) {
```

```
foo.modifyInSomeWay ();                 // está bien
foo = otro objeto;                      // seguirán problemas y desesperación
```

tú

La razón por la que no me gusta esto se debe a la falta de claridad y a la confusión entre el *pase por valor* y *pasar por referencia.* Java utiliza el paso por valor exclusivamente (ver más adelante), y esta discusión es basado en ese uso.

Con el paso por valor, cualquier cambio en el parámetro no se refleja en la rutina de llamada. Los que haber utilizado el pase por referencia probablemente encontrará esto confuso.

La otra área de confusión está dentro del cuerpo del código mismo. Es mucho más claro si usa solo el parámetro para representar lo que se ha pasado, porque ese es un uso constante.

En Java, no asigne parámetros, y si ve un código que sí, aplique *Eliminar asignaciones a los parámetros.*

Por supuesto, esta regla no se aplica necesariamente a otros idiomas que usan parámetros de salida, aunque incluso con estos idiomas prefiero usar los parámetros de salida lo menos posible.

Mecánica

- Crear una variable temporal para el parámetro.
- Reemplace todas las referencias al parámetro, hechas después de la asignación, a lo temporal variable.
- Cambiar la asignación para asignar a la variable temporal.
- Compilar y probar.

> ? *Si la semántica es llamada por referencia, mire el método de llamada para ver si el parámetro se usa nuevamente después. También vea cuántos llaman por parámetros de referencia se asignan y se utilizan después en este método. Intente devolver un solo valor como valor de retorno. Si hay más de uno, vea si puede convertir el grupo de datos en un objeto, o crear métodos separados.*

Ejemplo

Comienzo con la siguiente rutina simple:

```
descuento int (int inputVal, int cantidad, int yearToDate) {
     if (inputVal> 50) inputVal - = 2;
     if (cantidad> 100) inputVal - = 1;
     if (yearToDate> 10000) inputVal - = 4;
     return inputVal;
}
```

Reemplazar con una temperatura conduce a

```
     descuento int (int inputVal, int cantidad, int yearToDate) {
          int resultado = inputVal;
          if (inputVal> 50) resultado - = 2;
          if (cantidad> 100) resultado - = 1;
          si (YearToDate> 10.000) resultado - = 4;
          resultado de retorno ;
     }
```

Puede aplicar esta convención con la palabra clave final:


```
     int descuento ( última inputVal, int última cantidad int, última int
El año hasta la fecha) {
          int resultado = inputVal;
          if (inputVal> 50) resultado - = 2;
          if (cantidad> 100) resultado - = 1;
          resultado if (yearToDate> 10000) - = 4;
          resultado de retorno;
     }
```

Admito que no uso mucho el `final` , porque no creo que ayude mucho con claridad para abreviar métodos Lo uso con un método largo para ayudarme a ver si algo está cambiando el parámetro.

Pase por valor en Java

El uso de pasar por valor a menudo es una fuente de confusión en Java. Java utiliza estrictamente el paso por valor en todos lugares, por lo tanto, el siguiente programa:

```
  Param de clase {
    public static void main (String [] args) {
          int x = 5;
          triple);
          System.out.println ("x después de triple:" + x);
    }
    vacío estático privado triple (int arg) {
          arg = arg * 3;
          System.out.println ("arg en triple:" + arg);
    }
}
```

produce el siguiente resultado:

```
  arg en triple: 15
  x después del triple: 5
```

La confusión existe con los objetos. Digamos que uso una fecha, luego este programa:

```
Param de clase {

  public static void main (String [] args) {
       Fecha d1 = nueva fecha ("1 abr 98");
       nextDateUpdate (d1);
       System.out.println ("d1 después de nextDay:" + d1);

       Fecha d2 = nueva Fecha ("1 de abril de 98");
```


Page 110

```
       nextDateReplace (d2);
       System.out.println ("d2 después de nextDay:" + d2);
  }

  vacío estático privado nextDateUpdate (Fecha arg) {
       arg.setDate (arg.getDate () + 1);
       System.out.println ("arg en nextDay:" + arg);
  }

  vacío estático privado nextDateReplace (Date arg) {
       arg = nueva Fecha (arg.getYear (), arg.getMonth (), arg.getDate () +
1);
       System.out.println ("arg en nextDay:" + arg);
  }
}
```

Produce esta salida

```
arg en nextDay: Thu Apr 02 00:00:00 EST 1998
d1 después del próximo día: jueves 02 de abril 00:00:00 EST 1998
arg en nextDay: Thu Apr 02 00:00:00 EST 1998
d2 después del próximo Día: Mié Abr 01 00:00:00 EST 1998
```

Esencialmente, la referencia del objeto se pasa por valor. Esto me permite modificar el objeto pero no No tener en cuenta la reasignación del parámetro.

Java 1.1 y versiones posteriores le permiten marcar un parámetro como *final;* esto evita la asignación a La variable. Todavía le permite modificar el objeto al que se refiere la variable. Siempre trato mi parámetros como finales, pero confieso que rara vez los marco así en la lista de parámetros.

Reemplazar método con objeto de método

Tiene un método largo que usa variables locales de tal manera que no puede aplicar Extraer Método .

Convierta el método en su propio objeto para que todas las variables locales se conviertan en campos en ese objeto. Luego puede descomponer el método en otros métodos en el mismo objeto.

```
Orden de clase ...
  precio doble () {
  double primaryBasePrice;
       doble secundarioBasePrice;
       doble terciarioBasePrice;
       // cómputo largo;
       ...
  }
```


Página 111

Motivación

En este libro enfatizo la belleza de los métodos pequeños. Al extraer piezas de un método grande, haces las cosas mucho más comprensibles.

La dificultad para descomponer un método radica en las variables locales. Si son rampantes, descomposición puede ser difícil. El uso de Reemplazar temp con consulta ayuda a reducir esta carga, pero ocasionalmente es posible que no pueda analizar un método que necesita romperse. En este caso llegas a lo profundo en la bolsa de herramientas y saca tu *objeto de método* [Beck].

La aplicación de Método Reemplazar con Objeto Método convierte todas estas variables locales en campos en el método objeto Luego puede utilizar el Método de extracción en este nuevo objeto para crear más métodos que descomponen el método original.

Mecánica

Robado descaradamente de Beck [Beck].

- Cree una nueva clase, asígnele el nombre del método.
- Dele a la nueva clase un campo final para el objeto que alojó el método original (la fuente objeto) y un campo para cada variable temporal y cada parámetro en el método.

- Dele a la nueva clase un constructor que tome el objeto fuente y cada parámetro.
- Dele a la nueva clase un método llamado "computo".
- Copie el cuerpo del método original en el cálculo. Use el campo de objeto fuente para cualquier invocaciones de métodos sobre el objeto original.
- Compilar.
- Reemplace el método anterior con uno que cree el nuevo objeto y llame a compute.

Ahora viene la parte divertida. Debido a que todas las variables locales son ahora campos, puede libremente descomponga el método sin tener que pasar ningún parámetro.

Ejemplo

Un buen ejemplo de esto requiere un capítulo largo, así que estoy mostrando esta refactorización para un método que no lo necesita (No pregunte cuál es la lógica de este método, lo inventé a medida que avanzaba).

```
Cuenta de clase
   int gamma (int inputVal, int cantidad, int yearToDate) {
        int importantValue1 = (inputVal * cantidad) + delta ();
        int importantValue2 = (inputVal * yearToDate) + 100;
        if ((yearToDate - importantValue1)> 100)
             importantValue2 - = 20;
        int importantValue3 = importantValue2 * 7;
        // y así.
        return importantValue3 - 2 * importantValue1;
   }
```

Para convertir esto en un objeto de método, empiezo declarando una nueva clase. Proporciono un campo final para el objeto original y un campo para cada parámetro y variable temporal en el método.

```
clase Gamma ...
   Cuenta final privada _cuenta;
   private int inputVal;
   cantidad int privada;
   privado int yearToDate;
   private int importantValue1;
   private int importantValue2;
   private int importantValue3;
```

Usualmente uso la convención de prefijo de subrayado para marcar campos. Pero para mantener pequeños pasos me iré los nombres como son por el momento.

Añado un constructor:

```
     Gamma (fuente de la cuenta, int inputValArg, int cantidadArg, int
yearToDateArg) {
          _cuenta = fuente;
          inputVal = inputValArg;
          cantidad = cantidadArg;
          yearToDate = yearToDateArg;
     }
```

Ahora puedo mover el método original. Necesito modificar cualquier llamada de características de la cuenta para usar el campo `_cuenta`

112

113

```
int compute () {
     importantValue1 = (inputVal * cantidad) + _account.delta ();
     importantValue2 = (inputVal * yearToDate) + 100;
     if ((yearToDate - importantValue1)> 100)
          importantValue2 - = 20;
     int importantValue3 = importantValue2 * 7;
     // y así.
     return importantValue3 - 2 * importantValue1;
}
```

Luego modifico el método anterior para delegarlo al objeto del método:

```
int gamma (int inputVal, int cantidad, int yearToDate) {
     devuelve un nuevo Gamma (esto, inputVal, cantidad,
yearToDate) .compute ();
}
```

Esa es la refactorización esencial. El beneficio es que ahora puedo usar fácilmente el Método de extracción en calcule el método sin preocuparse por la aprobación del argumento:

```
int compute () {
     importantValue1 = (inputVal * cantidad) + _account.delta ();
     importantValue2 = (inputVal * yearToDate) + 100;
     cosa importante();
     int importantValue3 = importantValue2 * 7;
     // y así.
     return importantValue3 - 2 * importantValue1;
}

void importantThing () {
     if ((yearToDate - importantValue1)> 100)
          importantValue2 - = 20;
}
```

Algoritmo Sustituto

Desea reemplazar un algoritmo con uno que sea más claro.

Reemplace el cuerpo del método con el nuevo algoritmo.

```
String foundPerson (String [] people) {
```

```
for (int i = 0; i <people.length; i ++) {
     if (people [i] .equals ("Don")) {
          devolver "Don";
     }
     if (people [i] .equals ("John")) {
          regresar "John";
```

113

114

```
     }
     if (people [i] .equals ("Kent")) {
          volver "Kent";
     }
}
regreso "";
}
```

```
String foundPerson (String [] people) {
     Lista de candidatos = Arrays.asList (nueva cadena [] {"Don", "John",
"Kent"});
     for (int i = 0; i <people.length; i ++)
          if (candidatos.contiene (personas [i]))
               devolver personas [i];
     regreso "";
}
```

Motivación

Nunca he tratado de pelar un gato. Me dijeron que hay varias formas de hacerlo. Estoy seguro de que algunos son más fáciles que otros. Así es con los algoritmos. Si encuentra una manera más clara de hacer algo, debe reemplazar La forma complicada con la forma más clara. La refactorización puede descomponer algo complejo en piezas más simples, pero a veces solo llegas al punto en el que tienes que eliminar todo algoritmo y reemplazarlo con algo más simple. Esto ocurre a medida que aprende más sobre el problema y darse cuenta de que hay una manera más fácil de hacerlo. También sucede si comienzas a usar una biblioteca que proporciona funciones que duplican su código.

A veces, cuando desea cambiar el algoritmo para hacer algo ligeramente diferente, es más fácil sustituir el algoritmo primero en algo más fácil para el cambio que necesita hacer.

Cuando tenga que dar este paso, asegúrese de haber descompuesto el método tanto como puede. Sustituir un algoritmo grande y complejo es muy difícil; solo haciéndolo simple puedes hacer que la sustitución sea manejable.

Mecánica

- Prepare su algoritmo alternativo. Consíguelo para que se compile.
- Ejecute el nuevo algoritmo contra sus pruebas. Si los resultados son los mismos, ya ha terminado.
- Si los resultados no son los mismos, use el algoritmo anterior para comparar en las pruebas y depuración

? *Ejecute cada caso de prueba con algoritmos antiguos y nuevos y observe ambos resultados. Eso lo ayudará a ver qué casos de prueba están causando problemas, y*

cómo.

Capítulo 7. Mover características entre objetos

Una de las decisiones más fundamentales, si no la fundamental, en el diseño de objetos es decidir dónde para poner responsabilidades. He estado trabajando con objetos durante más de una década, pero todavía no consigo bien la primera vez. Eso solía molestarme, pero ahora me doy cuenta de que puedo usar la refactorización para cambiar mi mente en estos casos.

A menudo puedo resolver estos problemas simplemente usando Move Method y Move Field para mover el comportamiento alrededor. Si necesito usar ambos, prefiero usar Move Field primero y luego Move Method .

A menudo las clases se hinchan con demasiadas responsabilidades. En este caso uso Extract Class para separe algunas de estas responsabilidades. Si una clase se vuelve demasiado irresponsable, uso Inline Class para fusionarlo en otra clase. Si se está utilizando otra clase, a menudo es útil ocultar este hecho con Ocultar delegado . A veces, ocultar la clase delegada da como resultado un cambio constante del propietario interfaz, en cuyo caso debe usar Eliminar Middle Man

Las dos últimas refactorizaciones en este capítulo, Introducir método extranjero y Introducir local La extensión son casos especiales. Solo los uso cuando no puedo acceder al código fuente de un clase, pero quiero trasladar responsabilidades a esta clase inmutable. Si es solo uno o dos métodos, yo uso Introducir método extranjero ; para más de uno o dos métodos, uso Introducir extensión local .

Método de movimiento

Un método es, o será, utilizado o utilizado por más características de otra clase que la clase en la que se define.

Cree un nuevo método con un cuerpo similar en la clase que más utiliza. O convierte el viejo método en una delegación simple, o eliminarla por completo.

Motivación

Mover métodos es el pan y la mantequilla de la refactorización. Muevo métodos cuando las clases también mucha conducta o cuando las clases están colaborando demasiado y están demasiado acopladas. Por

moviendo los métodos, puedo simplificar las clases y terminar siendo más nítidas implementación de un conjunto de responsabilidades.

Normalmente miro a través de los métodos en una clase para encontrar un método que parece hacer referencia a otro objetar más que el objeto sobre el que vive. Un buen momento para hacerlo es después de haber movido algunos campos. Una vez que veo un método probable para moverme, miro los métodos que lo llaman, los métodos que llama, y cualquier método de redefinición en la jerarquía. Evalúo si seguir adelante sobre la base de objeto con el que el método parece tener más interacción.

No siempre es una decisión fácil de tomar. Si no estoy seguro de si mover un método, continúo con mira otros métodos. Mover otros métodos a menudo facilita la decisión. A veces el La decisión aún es difícil de tomar. En realidad, no es gran cosa. Si es difícil tomar la decisión, probablemente no importa tanto. Entonces elijo según el instinto; después de todo, siempre puedo cámbielo nuevamente más tarde.

Mecánica

- Examine todas las características utilizadas por el método de origen que se definen en la clase de origen. Considere si también deben ser trasladados.

  ? rarr; *Si una característica se usa solo por el método que está a punto de mover, usted bien podría moverlo también. Si la función es utilizada por otros métodos, considere moverlos también. A veces es más fácil mover un embrague de métodos que moverlos uno a la vez.*

- Verifique las subclases y superclases de la clase fuente para ver otras declaraciones de método.

  ? rarr; *Si hay otras declaraciones, es posible que no pueda hacer el movimiento, a menos que el polimorfismo también se pueda expresar en el objetivo.*

- Declarar el método en la clase de destino.

  ? rarr; *Puedes elegir usar un nombre diferente, uno que haga más sentido en la clase objetivo.*

- Copie el código del método fuente al destino. Ajuste el método para que funcione Es su nuevo hogar.

  ? rarr; *Si el método usa su fuente, debe determinar cómo hacer referencia al objeto de origen desde el método de destino. Si no hay mecanismo en la clase de destino, pase la referencia del objeto fuente al Nuevo método como parámetro.*

  ? rarr; *Si el método incluye manejadores de excepciones, decida qué clase debería manejar lógicamente la excepción. Si la clase fuente debe ser responsable, deje atrás a los manejadores.*

- Compilar la clase objetivo.
- Determine cómo hacer referencia al objeto de destino correcto desde la fuente.

Página 117

? rarr; *Puede haber un campo o método existente que le dará la objetivo. De lo contrario, vea si puede crear fácilmente un método que lo haga. De lo contrario, debe crear un nuevo campo en la fuente que pueda almacenar el objetivo. Esto puede ser un cambio permanente, pero también puede hacerlo temporalmente hasta que haya refactorizado lo suficiente como para eliminarlo.*

- Convierta el método de origen en un método de delegación.
- Compilar y probar.
- Decidir si eliminar el método de origen o retenerlo como método de delegación.

? rarr; *Dejar la fuente como método de delegación es más fácil si tiene Muchas referencias.*

- Si elimina el método de origen, reemplace todas las referencias con referencias al destino método.

? rarr; *Puede compilar y probar después de cambiar cada referencia, aunque Por lo general, es más fácil cambiar todas las referencias con una sola búsqueda y reemplazo.*

- Compilar y probar.

Ejemplo

Una clase de cuenta ilustra esta refactorización:

```
 Cuenta de clase ...
   sobregiro dobleCarga () {
        if (_type.isPremium ()) {
             doble resultado = 10;
             if (_daysOverdrawn> 7) result + = (_daysOverdrawn - 7) *
0,85;
             resultado de retorno;
        }
        de lo contrario return _daysOverdrawn * 1.75;
   }

   double bankCharge () {
        doble resultado = 4.5;
        if (_daysOverdrawn> 0) result + = overdraftCharge ();
        resultado de retorno;
   }
   Private AccountType _type;
   privado int _daysOverdrawn;
```

Imaginemos que habrá varios tipos de cuentas nuevas, cada una de las cuales tiene su propia cuenta. regla para calcular el cargo por sobregiro. Así que quiero mover el método de cargo por sobregiro a El tipo de cuenta.

El primer paso es observar las características que usa el método overdraftCharge y considerar si vale la pena mover un lote de métodos juntos. En este caso necesito el `_days` Campo `sobredibujado` para permanecer en la clase de cuenta, porque eso variará con cada individuo

cuentas

117

118

Luego copio el cuerpo del método al tipo de cuenta y hago que se ajuste.

```
clase AccountType ...
   Double overdraftCharge (int daysOverdrawn) {
        if ( isPremium () ) {
             doble resultado = 10;
             if ( daysOverdrawn > 7) result + = ( daysOverdrawn - 7) * 0.85;
             resultado de retorno;
        }
        más días de devolución Sobregiro * 1.75;
   }
```

En este caso, ajustar significa eliminar el `_type` de los usos de las características del tipo de cuenta, y haciendo algo sobre las características de la cuenta que todavía necesito. Cuando necesito usar una función de la clase fuente puedo hacer una de cuatro cosas: (1) mover esta característica también a la clase objetivo, (2) cree o use una referencia de la clase de destino a la fuente, (3) pase el objeto 0source como parámetro al método, (4) si la característica es una variable, pásala como parámetro.

En este caso pasé la variable como parámetro.

Una vez que el método se ajusta y compila en la clase de destino, puedo reemplazar el cuerpo del método fuente con una delegación simple:

```
Cuenta de clase ...
   sobregiro dobleCarga () {
        return _type.overdraftCharge (_daysOverdrawn);
   }
```

En este punto puedo compilar y probar.

Puedo dejar cosas como esta, o puedo eliminar el método en la clase fuente. Para eliminar el Necesito encontrar todos los llamadores del método y redirigirlos para llamar al método en cuenta tipo:

```
Cuenta de clase ...
   double bankCharge () {
        doble resultado = 4.5;
        if (_daysOverdrawn> 0) resultado + =
_type.overdraftCharge (_daysOverdrawn);
        resultado de retorno;
   }
```

Una vez que he reemplazado a todas las personas que llaman, puedo eliminar la declaración del método en cuenta. Puedo compilar y probar después de cada eliminación, o hacerlos en un lote. Si el método no es privado, necesito buscar otras clases que usan este método. En un lenguaje fuertemente tipado, la compilación después de la eliminación de la declaración de la fuente encuentra todo lo que me perdí.

En este caso, el método se refería solo a un solo campo, por lo que podría pasar este campo como variable. Si el método llamara a otro método en la cuenta, no habría podido hacerlo ese. En esos casos, necesito pasar el objeto de origen:

```
clase AccountType ...
   sobregiro doble Cargo (cuenta de la cuenta) {
        if (isPremium ()) {
             doble resultado = 10;
             if (account.getDaysOverdrawn ()> 7)
                 resultado + = (account.getDaysOverdrawn () - 7) * 0.85;
             resultado de retorno;
        }
        de lo contrario return account.getDaysOverdrawn () * 1.75;
   }
```

También paso el objeto fuente si necesito varias características de la clase, aunque si también hay muchos, se necesita una mayor refactorización. Por lo general, necesito descomponerme y mover algunas piezas hacia atrás.

Mover campo

Un campo es, o será, usado por otra clase más que la clase en la que está definido.

Cree un nuevo campo en la clase de destino y cambie todos sus usuarios.

Motivación

Mover el estado y el comportamiento entre clases es la esencia misma de la refactorización. Como el sistema desarrolla, encuentra la necesidad de nuevas clases y la necesidad de barajar las responsabilidades. UNA La decisión de diseño que sea razonable y correcta una semana puede volverse incorrecta en otra. Es decir no es un problema; El único problema es no hacer algo al respecto.

Considero mover un campo si veo más métodos en otra clase usando el campo que la clase sí mismo. Este uso puede ser indirecto, a través de métodos de obtención y configuración. Puedo elegir mover el métodos; Esta decisión se basa en la interfaz. Pero si los métodos parecen razonables donde están, yo mover el campo

Otra razón para mover el campo es cuando se hace la clase de extracción . En ese caso, los campos van primero y entonces los métodos.

Mecánica

- Si el campo es público, use Encapsulate Field .

? rarr; *Si es probable que mueva los métodos que acceden con frecuencia o si muchos métodos acceden al campo, puede ser útil usar* Self Campo encapsulado

- Compilar y probar.
- Cree un campo en la clase de destino con métodos de obtención y configuración.
- Compilar la clase objetivo.
- Determine cómo hacer referencia al objeto de destino desde la fuente.

? rarr; *Un campo o método existente puede darle el objetivo. Si no, ver si puede crear fácilmente un método que lo haga. De lo contrario, usted necesita crear un nuevo campo en la fuente que pueda almacenar el destino. Esta puede ser un cambio permanente, pero también puede hacerlo temporalmente hasta que han refactorizado lo suficiente como para eliminarlo.*

- Eliminar el campo en la clase de origen.
- Reemplace todas las referencias al campo de origen con referencias al método apropiado en el objetivo.

? rarr; *Para acceder a la variable, reemplace la referencia con una llamada a el método de obtención del objeto de destino; para asignaciones, reemplace la referencia con una llamada al método de configuración.*

? rarr; *Si el campo no es privado, busque en todas las subclases de la fuente referencias*

- Compilar y probar.

Ejemplo

Aquí es parte de una clase de cuenta:

```
Cuenta de clase ...
   Private AccountType _type;
   privado doble _interestRate;

   double InterestForAmount_days (cantidad doble, int días) {
        return _interestRate * cantidad * días / 365;
   }
```

Quiero mover el campo de tasa de interés al tipo de cuenta. Hay varios métodos con eso referencia, de los cuales es un ejemplo de `interés para` días de montaje. A continuación, creo el campo y accesores en el tipo de cuenta:

```
clase AccountType ...
   privado doble _interestRate;


   void setInterestRate (doble arg) {
        _interestRate = arg;
   }
```

Page 121

```
double getInterestRate () {
     return _interestRate;
}
```

Puedo compilar la nueva clase en este momento.

Ahora redirijo los métodos de la clase de cuenta para usar el tipo de cuenta y elimino el campo de tasa de interés en la cuenta. Debo eliminar el campo para asegurarme de que la redirección sea realmente sucediendo. De esta forma, el compilador ayuda a detectar cualquier método que no haya podido redirigir.

```
privado doble _interestRate;

double InterestForAmount_days (cantidad doble, int días) {
     return _type.getInterestRate () * cantidad * días / 365;
}
```

Ejemplo: uso de autoencapsulación

Si muchos métodos usan el campo de tasa de interés, podría comenzar usando el Campo de autoencapsulación :

```
Cuenta de clase ...
  Private AccountType _type;
  privado doble _interestRate;

  double InterestForAmount_days (cantidad doble, int días) {
       return getInterestRate () * cantidad * días / 365;
  }

  privado void setInterestRate (doble arg) {
       _interestRate = arg;
  }

  privado doble getInterestRate () {
       return _interestRate;
  }
```

De esa manera solo necesito hacer la redirección para los accesos:

```
  double InterestForAmountAndDays (cantidad doble, int días) {
       return getInterestRate () * cantidad * días / 365;
  }

  privado void setInterestRate (doble arg) {
       _type.setInterestRate (arg);
  }

  privado doble getInterestRate () {
       return _type.getInterestRate ();
  }
```

Puedo redirigir a los clientes de los accesores para usar el nuevo objeto más tarde si lo deseo. Usando uno mismo La encapsulación me permite dar un paso más pequeño. Esto es útil si estoy haciendo muchas cosas con el clase. En particular, simplifica el uso del Método Move para mover los métodos a la clase de destino. Si ellos consulte el descriptor de acceso, tales referencias no necesitan cambiar.

Extraer clase

Tienes una clase haciendo un trabajo que deberían hacer dos.

Cree una nueva clase y mueva los campos y métodos relevantes de la clase anterior a la nueva clase.

Motivación

Probablemente hayas escuchado que una clase debería ser una abstracción nítida, maneja algunas claras responsabilidades, o alguna directriz similar. En la práctica, las clases crecen. Agrega algunas operaciones aquí, un poco de datos allí. Agrega una responsabilidad a una clase sintiendo que no vale la pena clase, pero a medida que esa responsabilidad crece y se reproduce, la clase se vuelve demasiado complicada. Pronto tu la clase es tan crujiente como un pato al microondas.

Tal clase es una con muchos métodos y muchos datos. Una clase que es demasiado grande para entender fácilmente Debe considerar dónde se puede dividir, y lo divide. Una buena señal es que un subconjunto de los datos y un subconjunto de los métodos parecen ir de la mano. Otras buenas señales son subconjuntos de datos que generalmente cambian juntos o son particularmente dependientes el uno del otro. Un útil La prueba es preguntarse qué pasaría si eliminara un dato o un método. Qué otro campos y métodos se convertirían en tonterías?

Una señal que a menudo surge más tarde en el desarrollo es la forma en que se subtipa la clase. Puedes encontrar ese subtipo afecta solo algunas características o que algunas características necesitan subtiparse de una manera y Otras características de una manera diferente.

Mecánica

- Decidir cómo dividir las responsabilidades de la clase.
- Crear una nueva clase para expresar las responsabilidades divididas.

  ? rarr; *Si las responsabilidades de la clase anterior ya no coinciden con su nombre, cambiar el nombre de la vieja clase.*

- Haga un enlace de la clase antigua a la nueva.

  ? rarr; *Es posible que necesite un enlace bidireccional. Pero no hagas el enlace de regreso hasta te parece que lo necesitas.*

- Use Move Field en cada campo que desee mover.
- Compilar y probar después de cada movimiento.
- Use Move Method para mover los métodos de lo antiguo a lo nuevo. Comience con el nivel inferior métodos (llamados en lugar de llamar) y construir al nivel superior.
- Compilar y probar después de cada movimiento.
- Revisar y reducir las interfaces de cada clase.

> ? rarr; *Si tenía un enlace bidireccional, examine para ver si puede ser hecho de una manera.*

- Decidir si exponer la nueva clase. Si expone la clase, decida si exponerlo como un objeto de referencia o como un objeto de valor inmutable.

Ejemplo

Comienzo con una clase de persona simple:

```
Persona de clase ...
  public String getName () {
       return _name;
  }
  public String getTelephoneNumber () {
       return ("(" + _officeAreaCode + ")" + _officeNumber);
  }
  String getOfficeAreaCode () {
       return _officeAreaCode;
  }
  void setOfficeAreaCode (String arg) {
       _officeAreaCode = arg;
  }
  String getOfficeNumber () {
       return _officeNumber;
  }
  void setOfficeNumber (String arg) {
       _officeNumber = arg;
  }

  cadena privada _name;
  Private String _officeAreaCode;
  privado String _officeNumber;
```

En este caso, puedo separar el comportamiento del número de teléfono en su propia clase. Empiezo definiendo un clase de número de teléfono:

```
clase TelephoneNumber {
}
```

¡Eso fue fácil! Luego hago un enlace de la persona al número de teléfono:

```
Persona de clase
  Private TelephoneNumber _officeTelephone = new TelephoneNumber ();
```

Page 124

Ahora uso Move Field en uno de los campos:

```
clase TelephoneNumber {
   String getAreaCode () {
        return _areaCode;
   }
   void setAreaCode (String arg) {
        _areaCode = arg;
   }
   Private String _areaCode;
}
Persona de clase ...
   public String getTelephoneNumber () {
        return ("(" + getOfficeAreaCode () + ")" + _officeNumber);
   }
   String getOfficeAreaCode () {
        return _officeTelephone.getAreaCode ();
   }
   void setOfficeAreaCode (String arg) {
        _officeTelephone.setAreaCode (arg);
   }
```

Luego puedo mover el otro campo y usar Move Method en el número de teléfono:

```
Persona de clase ...
   public String getName () {
        return _name;
   }
   public String getTelephoneNumber () {
        return _officeTelephone.getTelephoneNumber ();
   }
   TelephoneNumber getOfficeTelephone () {
        return _officeTelephone;
   }

   cadena privada _name;
   Private TelephoneNumber _officeTelephone = new TelephoneNumber ();
clase TelephoneNumber ...
   public String getTelephoneNumber () {
        return ("(" + _areaCode + ")" + _number);
   }
   String getAreaCode () {
        return _areaCode;
   }
   void setAreaCode (String arg) {
        _areaCode = arg;
   }
   String getNumber () {
        return _number;
   }
   void setNumber (String arg) {
        _number = arg;
   }
   Cadena privada _ número;
```

```
Private String _areaCode;
```

La decisión entonces es cuánto exponer la nueva clase a mis clientes. Puedo ocultarlo por completo proporcionando métodos de delegación para su interfaz, o puedo exponerlo. Puedo elegir exponerlo a algunos clientes (como los de mi paquete) pero no a otros.

Si elijo exponer la clase, necesito considerar los peligros del aliasing. Si expongo el número de teléfono y un cliente cambia el código de área en ese objeto, ¿cómo me siento al respecto? Puede No ser un cliente directo que haga este cambio. Puede ser el cliente de un cliente de un cliente.

Tengo las siguientes opciones:

1. Acepto que cualquier objeto puede cambiar cualquier parte del número de teléfono. Esto hace que el número de teléfono un objeto de referencia, y debería considerar Cambiar valor a Referencia . En este caso, la persona sería el punto de acceso para el número de teléfono.
2. No quiero que nadie cambie el valor del número de teléfono sin pasar por la persona. Puedo hacer que el número de teléfono sea inmutable o puedo proporcionar un interfaz inmutable para el número de teléfono.
3. Otra posibilidad es clonar el número de teléfono antes de distribuirlo. Pero esto puede conducir a la confusión porque las personas piensan que pueden cambiar el valor. También puede conducir a problemas de alias entre clientes si el número de teléfono se pasa mucho.

Extract Class es una técnica común para mejorar la vida de un programa concurrente porque le permite tener bloqueos separados en las dos clases resultantes. Si no necesitas bloquea ambos objetos que no tienes que hacer. Para más información sobre esto, consulte la sección 3.3 en Lea [Lea].

Sin embargo, hay un peligro allí. Si necesita asegurarse de que ambos objetos estén bloqueados, entrar en el área de transacciones y otros tipos de bloqueos compartidos. Como se discutió en Lea por sección 8.1 [Lea], este es un territorio complejo y requiere maquinaria más pesada de lo que normalmente vale.

Las transacciones son muy útiles cuando las usa, pero escribir administradores de transacciones es más que la mayoría de los programadores deberían intentarlo.

Clase en línea

Una clase no está haciendo mucho.

Mueva todas sus características a otra clase y elimínelo.

Motivación

Inline Class es el reverso de Extract Class . Uso *Inline Class* si una clase ya no tira de su peso y ya no debería estar alrededor. A menudo, este es el resultado de la refactorización que mueve a otros responsabilidades fuera de la clase, por lo que queda poco. Entonces quiero doblar esta clase en otra clase, eligiendo uno que parece usar la clase runt más.

Mecánica

- Declarar el protocolo público de la clase fuente en la clase absorbente. Delegar todo estos métodos a la clase fuente.

  ? rarr; *Si una interfaz separada tiene sentido para los métodos de clase fuente, use la* interfaz de extracción *antes de alinear.*

- Cambie todas las referencias de la clase fuente a la clase absorbente.

  ? rarr; *Declarar la clase de origen privada para eliminar fuera del paquete referencias También cambie el nombre de la clase fuente para que el compilador captura cualquier referencia colgante a la clase fuente.*

- Compilar y probar.
- Use *Move Method* y *Move Field* para mover entidades de la clase fuente a absorbiendo clase hasta que no quede nada.
- Organice un servicio funerario corto y sencillo.

Ejemplo

Debido a que hice una clase con el número de teléfono, ahora lo incluyo en persona. Empiezo con clases separadas:

```
Persona de clase ...
  public String getName () {
       return _name;
  }
  public String getTelephoneNumber () {
       return _officeTelephone.getTelephoneNumber ();
  }
  TelephoneNumber getOfficeTelephone () {
       return _officeTelephone;
  }

  cadena privada _name;
  Private TelephoneNumber _officeTelephone = new TelephoneNumber ();

clase TelephoneNumber ...
  public String getTelephoneNumber () {
       return ("(" + _areaCode + ")" + _number);
  }
  String getAreaCode () {
       return _areaCode;
  }
  void setAreaCode (String arg) {
       _areaCode = arg;
  }
  String getNumber () {
       return _number;
  }
  void setNumber (String arg) {
       _number = arg;
```

```
    }
    Cadena privada _ número;
    Private String _areaCode;
```

Comienzo declarando todos los métodos visibles en el número de teléfono en persona:

```
  Persona de clase ...
    String getAreaCode () {
         return _officeTelephone.getAreaCode ();
    }
    void setAreaCode (String arg) {
         _officeTelephone.setAreaCode (arg);
    }
    String getNumber () {
         return _officeTelephone.getNumber ();
    }
    void setNumber (String arg) {
         _officeTelephone.setNumber (arg);
    }
```

Ahora encuentro clientes con número de teléfono y los cambio para usar la interfaz de la persona. Entonces

```
         Persona martin = nueva Persona ();
         martin.getOfficeTelephone (). setAreaCode ("781");
```

se convierte

```
         Persona martin = nueva Persona ();
         martin.setAreaCode ("781");
```

Ahora puedo usar Move Method y Move Field hasta que la clase telefónica ya no esté.

Ocultar delegado

Un cliente está llamando a una clase delegada de un objeto.

Cree métodos en el servidor para ocultar el delegado.

Motivación

Una de las claves, si no *la* clave, de los objetos es la encapsulación. Encapsulación significa que los objetos necesita saber menos sobre otras partes del sistema. Luego, cuando las cosas cambian, se necesitan menos objetos recibir información sobre el cambio, lo que hace que el cambio sea más fácil de realizar.

Cualquier persona involucrada en objetos sabe que debe ocultar sus campos, a pesar de que Java permite que los campos sean públicos. A medida que te vuelves más sofisticado, te das cuenta de que hay más que puedes encapsular.

Si un cliente llama a un método definido en uno de los campos del objeto del servidor, el cliente debe saber sobre este objeto delegado. Si el delegado cambia, el cliente también puede tener que cambiar. Usted puede elimine esta dependencia colocando un método de delegación simple en el servidor, que oculta delegado (Figura 7.1). Los cambios se limitan al servidor y no se propagan al cliente.

Figura 7.1. Delegación simple

Es posible que valga la pena utilizar Extract Class para algunos clientes del servidor o para todos los clientes. Si se oculta de todos los clientes, puede eliminar toda mención del delegado de la interfaz del servidor.

Mecánica

- Para cada método en el delegado, cree un método de delegación simple en el servidor.

- Ajuste el cliente para llamar al servidor.

  ? rarr; *Si el cliente no está en el mismo paquete que el servidor, considere cambiando el acceso del método delegado a la visibilidad del paquete.*

- Compile y pruebe después de ajustar cada método.
- Si ningún cliente necesita acceder más al delegado, elimine el acceso del servidor para delegar.
- Compilar y probar.

Ejemplo

Comienzo con una persona y un departamento:

```
Persona de clase {
   Departamento _departamento;

   departamento público getDepartment () {
        volver _departamento;
   }
   setDepartment público vacío (Departamento arg) {
        _departamento = arg;
   }
}

Departamento de clase {
   Private String _chargeCode;
   Persona privada _gerente;

   Departamento público (Gerente de persona) {
        _manager = manager;
   }

   public Person getManager () {
        return _manager;
   }
...
```

Si un cliente quiere conocer al gerente de una persona, primero necesita obtener el departamento:

```
manager = john.getDepartment (). getManager ();
```

Esto revela al cliente cómo funciona la clase de departamento y que el departamento es responsable para rastrear al gerente. Puedo reducir este acoplamiento ocultando la clase de departamento del cliente. Hago esto creando un método de delegación simple en persona:

```
   public Person getManager () {
        return _department.getManager ();
   }
```

Ahora necesito cambiar todos los clientes de persona para usar este nuevo método:

```
manager = john.getManager ();
```

Una vez que he realizado el cambio para todos los métodos de departamento y para todos los clientes de persona, puedo retire el `descriptor de` acceso `getDepartment` en persona.

Eliminar al hombre del medio

Una clase está haciendo demasiada delegación simple.

Haga que el cliente llame al delegado directamente.

Motivación

En la motivación para *Hide Delegate,* hablé sobre las ventajas de encapsular el uso de un objeto delegado Hay un precio por esto. El precio es que cada vez que el cliente quiere usar un nueva característica del delegado, debe agregar un método de delegación simple al servidor. Después agregando características por un tiempo, se vuelve doloroso. La clase de servidor es solo un intermediario, y tal vez es hora de que el cliente llame al delegado directamente.

Es difícil descubrir cuál es la cantidad correcta de escondite. Afortunadamente, con *Hide Delegate* y Elimina a Middle Man , no importa tanto. Puede ajustar su sistema a medida que pasa el tiempo.

A medida que cambia el sistema, la base de cuánto oculta también cambia. Una buena encapsulación seis hace meses puede ser incómodo ahora. Refactorizar significa que nunca tienes que decir que lo sientes, tú solo arréglalo.

Mecánica

- Crear un descriptor de acceso para el delegado.
- Para cada uso del cliente de un método delegado, elimine el método del servidor y reemplace la llamada en el cliente para llamar al método en el delegado.
- Compilar y probar después de cada método.

Ejemplo

Página 131

Por ejemplo, yo uso persona y departamento volteado al revés. Empiezo con una persona escondiendo el

Departamento:

```
Persona de clase ...
  Departamento _departamento;
  public Person getManager () {
       return _department.getManager ();

Departamento de clase ...
  Persona privada _gerente;
  Departamento público (Gerente de persona) {
       _manager = manager;
  }
```

Para encontrar el gerente de una persona, los clientes preguntan:

```
manager = john.getManager ();
```

Esto es simple de usar y encapsula el departamento. Sin embargo, si muchos métodos están funcionando esto, termino con demasiadas delegaciones simples en la persona. Ahí es cuando es bueno eliminar al intermediario. Primero hago un descriptor de acceso para el delegado:

```
Persona de clase ...
  departamento público getDepartment () {
       volver _departamento;
  }
```

Luego tomo cada método a la vez. Encuentro clientes que usan el método en persona y lo cambio a primero obtenga el delegado. Entonces lo uso:

```
manager = john.getDepartment (). getManager ();
```

Entonces puedo eliminar `getManager` de la persona. Una compilación muestra si me perdí algo.

Es posible que desee mantener algunas de estas delegaciones por conveniencia. También es posible que desee ocultar el delegar de algunos clientes pero mostrarlo a otros. Eso también dejará algunos de los simples delegaciones en el lugar.

Introducir método extranjero

Una clase de servidor que está utilizando necesita un método adicional, pero no puede modificar la clase.

Cree un método en la clase de cliente con una instancia de la clase de servidor como primer argumento.

```
Date newStart = new Date (previousEnd.getYear (),
                      previousEnd.getMonth (), previousEnd.getDate () + 1);
```



```
Fecha newStart = nextDay (previousEnd);
```

```
     fecha estática privada nextDay (fecha arg) {
          return new Date (arg.getYear (), arg.getMonth (), arg.getDate () +
1);
     }
```

Motivación

Sucede con bastante frecuencia. Estás usando esta clase realmente agradable que te da todos estos excelentes servicios. Entonces hay un servicio que no te da pero que debería. Maldices a la clase diciendo: "¿Por qué no haces eso?" Si puede cambiar la fuente, puede agregar el método. Si no puedes cambie la fuente, debe codificar la falta del método en el cliente.

Si usa el método solo una vez en la clase de cliente, la codificación adicional no es gran cosa y probablemente no fue necesario en la clase original de todos modos. Si usa el método varias veces, Sin embargo, debe repetir esta codificación. Debido a que la repetición es la raíz de todo mal del software, este código repetitivo debe factorizarse en un solo método. Cuando haces esta refactorización, tú puede indicar claramente que este método es realmente un método que debería estar en el original al convertirlo en un método extranjero

Si te encuentras creando muchos métodos foráneos en una clase de servidor, o encuentras muchos de tus las clases necesitan el mismo método foráneo, en su lugar debes usar Introducir extensión local .

No olvide que los métodos extranjeros son una solución alternativa. Si puede, intente mover los métodos a sus propios hogares. Si el problema es la propiedad del código, envíe el método externo al propietario del clase de servidor y solicite al propietario que implemente el método por usted.

Mecánica

- Cree un método en la clase de cliente que haga lo que necesita.

  ? rarr; *El método no debe acceder a ninguna de las funciones del cliente. clase. Si necesita un valor, envíelo como parámetro.*

- Haga que una instancia de la clase de servidor sea el primer parámetro.
- Comente el método como "método extraño; debe estar en el *servidor* " .

  ? rarr; *De esta manera, puede utilizar una búsqueda de texto para buscar métodos extraños más adelante si tienes la oportunidad de mover el método.*

Ejemplo

Tengo un código que necesita pasar por un período de facturación. El código original se ve así:

```
Date newStart = new Date (previousEnd.getYear (),
    previousEnd.getMonth (), previousEnd.getDate () + 1);
```

Puedo extraer el código en el lado derecho de la tarea en un método. Este método es un método extranjero para la fecha:


```
Fecha newStart = nextDay (previousEnd);

fecha estática privada nextDay (fecha arg) {
```

```
   // método extranjero, debe estar en fecha
        return new Date (arg.getYear (), arg.getMonth (), arg.getDate () +
1);
   }
```

Introducir extensión local

Una clase de servidor que está utilizando necesita varios métodos adicionales, pero no puede modificar la clase.

Cree una nueva clase que contenga estos métodos adicionales. Convierta esta clase de extensión en una subclase o una envoltorio del original.

Motivación

Lamentablemente, los autores de las clases no son omniscientes y no pueden proporcionarle métodos útiles. Si puede modificar la fuente, a menudo lo mejor es agregar ese método. Sin embargo, a menudo no puedes Modificar la fuente. Si necesita uno o dos métodos, puede usar Introducir método extranjero . Sin embargo, una vez que superas algunos de estos métodos, se salen de control. Entonces necesitas Agrupe los métodos en un lugar adecuado para ellos. Las técnicas estándar orientadas a objetos. de subclases y envolturas son una forma obvia de hacer esto. En estas circunstancias yo llamo el subclase o envoltorio una extensión local.

Una extensión local es una clase separada, pero es un subtipo de la clase que está extendiendo. Eso significa que admite todas las cosas que puede hacer el original, pero también agrega funciones adicionales. En lugar de usar el clase original, crea una instancia de la extensión local y la usa.

Al usar la extensión local, usted mantiene el principio de que los métodos y los datos deben empaquetarse en unidades bien formadas. Si sigue poniendo código en otras clases que deberían estar en la extensión, terminas complicando las otras clases y haciendo que sea más difícil reutilizar estos métodos.

Al elegir entre la subclase y el contenedor, generalmente prefiero la subclase porque es menos trabajo. El mayor obstáculo para una subclase es que debe aplicarse en el momento de la creación del objeto. Si puedo tomar sobre el proceso de creación no hay problema. El problema ocurre si aplica la extensión local más tarde. La subclase me obliga a crear un nuevo objeto de esa subclase. Si otros objetos se refieren a viejo, tengo dos objetos con los datos del original. Si el original es inmutable, no hay

problema; Puedo tomar una copia con seguridad. Pero si el original puede cambiar, hay un problema, porque los cambios en un objeto no cambiarán el otro y tengo que usar un contenedor. Esa forma cambia hecho a través de la extensión local afecta el objeto original y viceversa.

Mecánica

- Cree una clase de extensión como una subclase o un contenedor del original.
- Agregar constructores de conversión a la extensión.

> ? rarr; *Un constructor toma el original como argumento. La subclase la versión llama a un constructor de superclase apropiado; la versión de envoltura establece el campo delegado al argumento.*

- Agregar nuevas funciones a la extensión.
- Reemplace el original con la extensión donde sea necesario.
- Mueva cualquier método extraño definido para esta clase a la extensión.

Ejemplos

Tuve que hacer este tipo de cosas bastante con Java 1.0.1 y la clase de fecha. La clase de calendario en 1.1 me dio el comportamiento que quería, pero antes de que llegara, me dio bastantes Oportunidades para utilizar la extensión. Lo uso como ejemplo aquí.

Lo primero que debe decidir es si usar una subclase o un contenedor. Subclasificar es el más forma obvia:

```
La clase mfDate extiende la fecha {
  public nextDay () ...
  public dayOfYear () ...
```

Un contenedor usa delegación:

```
clase mfDate {
  Fecha privada _original;
```

Ejemplo: uso de una subclase

Primero creo la nueva fecha como una subclase del original:

```
la clase MfDateSub extiende la fecha
```

Luego trato de cambiar entre las fechas y la extensión. Los constructores de la necesidad original. se repetirá con simple delegación:

```
  public MfDateSub (String dateString) {
         super (dateString);
  };
```


Ahora agrego un constructor de conversión, uno que toma un original como argumento:

```
public MfDateSub (Date arg) {
     super (arg.getTime ());
}
```

Ahora puedo agregar nuevas funciones a la extensión y usar Move Method para mover cualquier elemento extraño métodos a la extensión:

```
  clase de cliente ...
    fecha estática privada nextDay (fecha arg) {
    // método extranjero, debe estar en fecha
         return new Date (arg.getYear (), arg.getMonth (), arg.getDate () +
1);
    }
```

se convierte

```
  clase MfDate ...
    Fecha nextDay () {
         return new Date (getYear (), getMonth (), getDate () + 1);
  }
```

Ejemplo: uso de un contenedor

Comienzo declarando la clase de envoltura:

```
  clase mfDate {
    Fecha privada _original;
  }
```

Con el enfoque de ajuste, necesito configurar los constructores de manera diferente. El original Los constructores se implementan con delegación simple:

```
   public MfDateWrap (String dateString) {
        _original = nueva Fecha (dateString);
   };
```

El constructor de conversión ahora solo establece la variable de instancia:

```
   public MfDateWrap (fecha arg) {
        _original = arg;
   }
```

Luego está la tediosa tarea de delegar todos los métodos de la clase original. Solo muestro un Pareja.

Page 136

```
   public int getYear () {
        return _original.getYear ();
   }

   public boolean igual (MfDateWrap arg) {
        return (toDate (). equals (arg.toDate ()));
```

```
    }
```

Una vez que haya hecho esto, puedo usar Move Method para poner un comportamiento específico de la fecha en la nueva clase:

```
   clase de cliente ...
     fecha estática privada nextDay (fecha arg) {
     // método extranjero, debe estar en fecha
          return new Date (arg.getYear (), arg.getMonth (), arg.getDate () +
1);
     }
```

se convierte

```
   clase MfDate ...
     Fecha nextDay () {
          return new Date (getYear (), getMonth (), getDate () + 1);
   }
```

Un problema particular con el uso de envoltorios es cómo lidiar con los métodos que toman un original como argumento, como

```
   público booleano después (fecha arg)
```

Como no puedo alterar el original, solo puedo hacerlo después en una dirección:

```
   aWrapper.after (aDate)                                  // puede hacerse trabajar
   aWrapper.after (otroWrapper)                            // puede hacerse trabajar
   aDate.after (aWrapper)                                  // no trabajará
```

El propósito de este tipo de anulación es ocultar el hecho de que estoy usando un contenedor del usuario del clase. Esta es una buena política porque el usuario del contenedor realmente no debería preocuparse por el contenedor y debería ser capaz de tratar a los dos por igual. Sin embargo, no puedo ocultar completamente esta información. los El problema radica en ciertos métodos del sistema, como `iguales.` Idealmente pensarías que podrías anular `es igual` en `MfDateWrap` como este

```
   public boolean equals (Date arg) // causa problemas
```

Esto es peligroso porque aunque puedo hacer que funcione para mis propios fines, otras partes del El sistema Java asume que igual es simétrico: que si `a.equals (b)` entonces `b.equals (a)` . Si yo violar esta regla me encontraré con un grupo de bichos extraños. La única forma de evitar eso sería modificar Fecha, y si pudiera hacer eso, no estaría usando esta refactorización. Así que en situaciones como esta solo tengo para exponer el hecho de que estoy envolviendo. Para las pruebas de igualdad, esto significa un nuevo nombre de método.

137

```
   public boolean equalsDate (Fecha arg)
```

Puedo evitar probar el tipo de objetos desconocidos proporcionando versiones de este método para ambos Fecha y MfDateWrap.

```
   public boolean equalsDate (MfDateWrap arg)
```

El mismo problema no es un problema con la subclasificación, si no anulo la operación. Si lo hago anular, me confundo completamente con el método de búsqueda. Por lo general no anulo métodos con extensiones; Por lo general, solo agrego métodos.

Page 138

Capítulo 8. Organización de datos

En este capítulo, analizo varias refactorizaciones que facilitan el trabajo con datos. Para mucha gente El campo auto encapsulado parece innecesario. Durante mucho tiempo ha sido un tema de debate afable sobre si un objeto debe acceder a sus propios datos directamente o a través de accesores. A veces necesita los accesores, y luego puede obtenerlos con el campo Self Encapsulate Field . Yo por lo general uso el acceso directo porque me resulta sencillo hacer esta refactorización cuando lo necesito.

Una de las cosas útiles sobre los lenguajes de objetos es que le permiten definir nuevos tipos que van más allá de lo que se puede hacer con los tipos de datos simples de los idiomas tradicionales. Toma un tiempo acostumbrarse a cómo hacer esto, sin embargo. A menudo comienza con un valor de datos simple y luego se da cuenta de que Un objeto sería más útil. Reemplazar valor de datos con objeto le permite volverse tonto datos en objetos articulados. Cuando te das cuenta de que estos objetos son instancias que serán necesarias en muchas partes del programa, puede usar Cambiar valor a referencia para convertirlos en objetos de referencia.

Si ve una matriz que actúa como una estructura de datos, puede aclarar la estructura de datos con Reemplazar matriz con objeto . En todos estos casos, el objeto no es más que el primer paso. El Real La ventaja se obtiene al usar el Método de movimiento para agregar comportamiento a los nuevos objetos.

Los números mágicos, números con un significado especial, han sido durante mucho tiempo un problema. Recuerdo haber sido me dijo en mis primeros días de programación que no los usara. Sin embargo, siguen apareciendo, y yo use Reemplazar número mágico con constante simbólica para deshacerse de los números mágicos siempre que averiguar lo que están haciendo.

Los enlaces entre objetos pueden ser unidireccionales o bidireccionales. Los enlaces unidireccionales son más fáciles, pero a veces usted necesita cambiar la Asociación unidireccional a Bidireccional para admitir una nueva función. Cambiar la Asociación Bidireccional a Unidireccional elimina la complejidad innecesaria si descubres que ya no necesitas el enlace bidireccional.

A menudo me he encontrado con casos en los que las clases de GUI están haciendo lógica empresarial que no deberían. A mover el comportamiento a las clases de dominio adecuadas, debe tener los datos en la clase de dominio y admite la GUI mediante el uso de datos observados duplicados . Normalmente no me gusta duplicar datos, pero esta es una excepción que generalmente es imposible de evitar.

Uno de los principios clave de la programación orientada a objetos es la encapsulación. Si hay datos públicos rayando, puede usar Encapsulate Field para cubrirlo de manera decorativa. Si esos datos son un colección, use Encapsulate Collection en su lugar, porque tiene un protocolo especial. Si todo el registro está desnudo, use Reemplazar registro con clase de datos .

Una forma de datos que requiere un tratamiento particular es el código de tipo: un valor especial que indica algo particular sobre un tipo de instancia. Estos a menudo aparecen como enumeraciones, a menudo implementado como enteros finales estáticos. Si los códigos son para información y no alteran el comportamiento de la clase, puede usar Reemplazar código de tipo con clase , que le brinda una mejor verificación de tipo y una plataforma para mover el comportamiento más tarde. Si el comportamiento de una clase se ve afectado por un código de tipo, use Reemplace el código de tipo con subclases si es posible. Si no puede hacer eso, use más complicado (pero más flexible) Reemplace el código de tipo por estado / estrategia .

Campo auto encapsulado

Está accediendo a un campo directamente, pero el acoplamiento al campo se está volviendo incómodo.

Cree métodos de obtención y configuración para el campo y use solo aquellos para acceder al campo.

```
privado int _low, _high;
boolean incluye (int arg) {
     return arg> = _low && arg <= _high;
}
```

```
privado int _low, _high;
```

```
boolean incluye (int arg) {
     return arg> = getLow () && arg <= getHigh ();
}
int getLow () {return _low;}
int getHigh () {return _high;}
```

Motivación

Cuando se trata de acceder a los campos, hay dos escuelas de pensamiento. Una es que dentro de la clase donde se define la variable, debe acceder a la variable libremente (acceso directo a la variable). los otra escuela es que incluso dentro de la clase, siempre debes usar los accesores (variable indirecta acceso). Los debates entre los dos se pueden calentar. (Véase también la discusión en Auer [Auer] sobre página 413 y Beck [Beck].)

Esencialmente, las ventajas del *acceso variable indirecto* son que permite que una subclase anule cómo obtener esa información con un método y que sea más flexible en la gestión de datos, como la inicialización diferida, que inicializa el valor solo cuando necesita usarlo.

La ventaja del *acceso variable directo* es que el código es más fácil de leer. No necesitas parar y diga: "Esto es solo un método de obtención".

Siempre tengo dos mentes con esta elección. Normalmente estoy feliz de hacer lo que el resto del equipo quiere que hacer. Sin embargo, dejándome a mí mismo, me gusta usar el acceso variable directo como primer recurso, hasta que llegue al camino. Una vez que las cosas comienzan a ponerse incómodas, cambio al acceso variable indirecto. Refactorizar da usted la libertad de cambiar de opinión.

El momento más importante para usar el *campo Self Encapsulate* es cuando está accediendo a un campo en un superclase pero desea anular este acceso variable con un valor calculado en la subclase. Autoencapsular el campo es el primer paso. Después de eso, puede anular la obtención y la configuración métodos como sea necesario.

Mecánica

- Cree un método de obtención y configuración para el campo.
- Encuentre todas las referencias al campo y reemplácelas con un método de obtención o configuración.

  ? rarr; *Reemplace los accesos al campo con una llamada al método de obtención; reemplazar asignaciones con una llamada al método de configuración.*

  ? rarr; *Puede obtener el compilador para ayudarlo a verificar temporalmente renombrando el campo.*

- Hacer el campo privado.
- Verifique que haya captado todas las referencias.
- Compilar y probar.

Ejemplo

Esto parece casi demasiado simple para un ejemplo, pero, oye, al menos es rápido escribir:

```
clase IntRange {
```

```
privado int _low, _high;

boolean incluye (int arg) {
     return arg> = _low && arg <= _high;
}

crecimiento nulo (factor int) {
     _high = _high * factor;
}
IntRange (int bajo, int alto) {
     _low = bajo;
     _high = alto;
}
```

Para autoencapsular defino métodos de obtención y configuración (si aún no existen) y utilizo aquellos:

```
clase IntRange {

   boolean incluye (int arg) {
        return arg> = getLow () && arg <= getHigh ();
   }

   crecimiento nulo (factor int) {
        setHigh (factor getHigh () *);
   }

   privado int _low, _high;

   int getLow () {
        return _low;
   }

   int getHigh () {
        volver _high;
   }

   vacío setLow (int arg) {
        _low = arg;
   }

   void setHigh (int arg) {
        _high = arg;
```



```
}
```

Cuando utiliza la autoencapsulación, debe tener cuidado al utilizar el método de configuración en El constructor. A menudo se supone que usa el método de configuración para los cambios después del objeto se crea, por lo que puede tener un comportamiento diferente en el setter que el que tiene al inicializar. En casos como este prefiero usar el acceso directo desde el constructor o una inicialización separada método:

```
IntRange (int bajo, int alto) {
     inicializar (bajo, alto);
}
```

```
    inicialización de vacío privado (int bajo, int alto) {
         _low = bajo;
         _high = alto;
    }
```

El valor de hacer todo esto viene cuando tienes una subclase, como sigue:

```
clase CappedRange extiende IntRange {

    CappedRange (int low, int high, int cap) {
         super (bajo, alto);
         _cap = cap;
    }

    privado int _cap;

    int getCap () {
         return _cap;
    }

    int getHigh () {
         return Math.min (super.getHigh (), getCap ());
    }
  }
```

Puedo anular todo el comportamiento de `IntRange` para tener en cuenta el límite sin cambiar ninguno de ese comportamiento.

Reemplazar valor de datos con objeto

Tiene un elemento de datos que necesita datos o comportamiento adicionales.

Convierta el elemento de datos en un objeto.

Motivación

A menudo, en las primeras etapas de desarrollo, toma decisiones sobre la representación de hechos simples como Elementos de datos simples. A medida que avanza el desarrollo, te das cuenta de que esos elementos simples no son tan simples nunca más. Un número de teléfono puede ser representado como una cadena por un tiempo, pero luego te das cuenta que el teléfono necesita un comportamiento especial para formatear, extraer el código de área y similares.
Para uno o dos elementos, puede colocar los métodos en el objeto propietario, pero rápidamente el código huele de duplicación y envidia de características. Cuando comienza el olor, convierta el valor de los datos en un objeto.

Mecánica

- Crear la clase para el valor. Déle un campo final del mismo tipo que el valor en el clase fuente Agregue un captador y un constructor que tome el campo como argumento.
- Compilar.
- Cambie el tipo de campo en la clase de origen a la nueva clase.
- Cambie el captador en la clase de origen para llamar al captador en la nueva clase.
- Si el campo se menciona en el constructor de la clase de origen, asigne el campo utilizando el constructor de la nueva clase.
- Cambie el método de obtención para crear una nueva instancia de la nueva clase.
- Compilar y probar.
- Es posible que ahora necesite usar Cambiar valor a referencia en el nuevo objeto.

Ejemplo

Comienzo con una clase de pedido que ha almacenado al cliente del pedido como una cadena y quiere activar El cliente en un objeto. De esta manera, tengo un lugar para almacenar datos, como una dirección o calificación crediticia y comportamiento útil que utiliza esta información.

```
Orden de clase ...
  Orden público (cliente de cadena) {
       _cliente = cliente;
```



```
  }
  public String getCustomer () {
       volver _cliente;
  }
  public void setCustomer (String arg) {
       _customer = arg;
  }
  Cadena privada _cliente;
```

Algunos códigos de clientes que usan esto se parecen a

```
    private static int numberOfOrdersFor (órdenes de colección, cadena
cliente) {
         int resultado = 0;
         Iterador iter = orders.iterator ();
```

```
        while (iter.hasNext ()) {
             Ordenar cada = (Ordenar) iter.next ();
             if (each.getCustomerName (). equals (customer)) result ++;
        }
        resultado de retorno;
    }
```

Primero creo la nueva clase de cliente. Le doy un campo final para un atributo de cadena, porque eso es qué usa actualmente el pedido. Lo llamo *nombre,* porque parece ser para eso que se usa la cadena. También agrego un método de obtención y proporciono un constructor que usa el atributo:

```
clase Cliente {
   Cliente público (nombre de cadena) {
        _name = nombre;
   }
   public String getName () {
        return _name;
   }
   privado final String _name;
 }
```

Ahora cambio el tipo de campo del cliente y los métodos que hacen referencia para usar el referencias apropiadas en la clase de cliente. El captador y el constructor son obvios. Para el setter creo un nuevo cliente:

```
Orden de clase ...
   Orden público (cliente de cadena) {
        _customer = nuevo cliente (cliente);
   }
   public String getCustomer () {
        return _customer.getName ();
   }
   Cliente privado _cliente;

   public void setCustomer (String arg) {
        _customer = nuevo cliente (cliente);
   }
```


Page 144

El instalador crea un nuevo cliente porque el antiguo atributo de cadena era un objeto de valor y, por lo tanto, El cliente actualmente también es un objeto de valor. Esto significa que cada pedido tiene su propio cliente objeto. Como regla, los objetos de valor deben ser inmutables; Esto evita algunos errores de alias desagradables. Más tarde Quiero que el cliente sea un objeto de referencia, pero esa es otra refactorización. En este punto puedo compilar y probar.

Ahora miro los métodos en orden que manipulan al cliente y hago algunos cambios para hacer El nuevo estado de cosas más claro. Con el getter utilizo el método Rename para dejar en claro que es el nombre no el objeto que se devuelve:

```
    public String getCustomerName () {
        return _customer.getName ();
    }
```

En el constructor y el setter, no necesito cambiar la firma, sino el nombre del

los argumentos deberían cambiar:

```
Orden público (String customerName) {
     _customer = nuevo cliente (customerName);
}
public void setCustomer (String customerName) {
     _customer = nuevo cliente (customerName);
}
```

Refactorizar más puede hacer que agregue un nuevo constructor y setter que tome un existente cliente.

Esto termina esta refactorización, pero en este caso, como en muchos otros, hay otro paso. Si yo quiero agregue cosas tales como calificaciones crediticias y direcciones a nuestros clientes, no puedo hacerlo ahora. Esto es porque el cliente es tratado como un objeto de valor. Cada pedido tiene su propio objeto de cliente. A dar a un cliente estos atributos que necesito para aplicar Cambiar valor a referencia al cliente para que todos los pedidos del mismo cliente compartan el mismo objeto de cliente. Encontrarás este ejemplo continuó allí.

Cambiar valor a referencia

Tiene una clase con muchas instancias iguales que desea reemplazar con un solo objeto.

Convierta el objeto en un objeto de referencia.

Motivación

Puede hacer una clasificación útil de objetos en muchos sistemas: objetos de referencia y valor objetos. *Los objetos de referencia* son cosas como cliente o cuenta. Cada objeto representa un objeto en el mundo real, y usas la identidad del objeto para probar si son iguales. *Los objetos de valor* son cosas como fecha o dinero. Se definen completamente a través de sus valores de datos. No te importa eso existen copias; puede tener cientos de objetos "1/1/2000" alrededor de su sistema. Usted necesita diga si dos de los objetos son iguales, por lo que debe anular el método igual (y el método hashCode también).

La decisión entre referencia y valor no siempre es clara. A veces comienzas con un simple valor con una pequeña cantidad de datos inmutables. Entonces quieres darle algunos datos modificables y asegúrese de que los cambios afecten a todos los que se refieren al objeto. En este punto necesitas girarlo en un objeto de referencia.

Mecánica

- Utilice Reemplazar constructor con método de fábrica .
- Compilar y probar.
- Decidir qué objeto es responsable de proporcionar acceso a los objetos.

  ? rarr; *Esto puede ser un diccionario estático o un objeto de registro.*

  ? rarr; *Puede tener más de un objeto que actúa como punto de acceso. para el nuevo objeto.*

- Decidir si los objetos se crean previamente o se crean sobre la marcha.

  ? rarr; *Si los objetos se crean previamente y los está recuperando de memoria, debe asegurarse de que estén cargados antes de que sean necesarios.*

- Modifique el método de fábrica para devolver el objeto de referencia.

  ? rarr; *Si los objetos se calculan previamente, debe decidir cómo manejar errores si alguien solicita un objeto que no existe.*

  ? rarr; *Es posible que desee utilizar el* método de cambio de nombre *en la fábrica para transmitir que devuelve un objeto existente.*

- Compilar y probar.

Ejemplo

Comienzo donde lo dejé en el ejemplo de Reemplazar valor de datos con objeto . Tengo lo siguiente clase de cliente:

```
clase Cliente {
  Cliente público (nombre de cadena) {
       _name = nombre;
  }
```

```
    public String getName () {
         return _name;
    }
    privado final String _name;
}
```

Lo utiliza una clase de orden:

```
Orden de clase ...
    Orden público (String customerName) {
         _customer = nuevo cliente (customerName);
    }
    public void setCustomer (String customerName) {
         _customer = nuevo cliente (customerName);
    }
    public String getCustomerName () {
         return _customer.getName ();
    }
    Cliente privado _cliente;
```

y algo de código de cliente:

```
   private static int numberOfOrdersFor (órdenes de colección, cadena
cliente) {
        int resultado = 0;
        Iterador iter = orders.iterator ();
        while (iter.hasNext ()) {
             Ordenar cada = (Ordenar) iter.next ();
             if (each.getCustomerName (). equals (customer)) result ++;
        }
        resultado de retorno;
   }
```


Page 147

Por el momento es un valor. Cada pedido tiene su propio objeto de cliente, incluso si son para el mismo cliente conceptual Quiero cambiar esto para que si tenemos varios pedidos para el mismo cliente conceptual, comparten un solo objeto de cliente. Para este caso esto significa que hay debe ser solo un objeto de cliente para cada nombre de cliente.

Comienzo usando Reemplazar constructor con el método de fábrica . Esto me permite tomar el control de El proceso de creación, que será importante más adelante. Defino el método de fábrica en el cliente:

```
clase Cliente {
    Public static Customer create (String name) {
         devolver nuevo cliente (nombre);
    }
```

Luego reemplazo las llamadas al constructor con llamadas a la fábrica:

```
Orden de clase {
Orden público (cliente de cadena) {
     _customer = Customer.create (cliente);
```

```
}
```

Luego hago que el constructor sea privado:

```
clase Cliente {
  Cliente privado (nombre de cadena) {
      _name = nombre;
  }
```

Ahora tengo que decidir cómo acceder a los clientes. Mi preferencia es usar otro objeto. Tal una situación funciona bien con algo como las líneas de pedido en un pedido. El pedido es responsable de proporcionar acceso a las líneas de pedido. Sin embargo, en esta situación no hay un objeto tan obvio. En En esta situación, generalmente creo un objeto de registro para ser el punto de acceso. Por simplicidad en esto ejemplo, sin embargo, los almaceno usando un campo estático en el cliente, haciendo que la clase de cliente sea punto de acceso:

```
  Diccionario estático privado _instances = new Hashtable ();
```

Luego decido si crear clientes sobre la marcha cuando se me solicite o crearlos por adelantado. Usaré lo último. En el código de inicio de mi aplicación, cargo a los clientes que están en uso. Estos podrían provienen de una base de datos o de un archivo. Por simplicidad, uso un código explícito. Siempre puedo usar Sustituir Algoritmo para cambiarlo más tarde.

```
clase Cliente ...
  loadCustomers vacío estático () {
       nuevo cliente ("Lemon Car Hire"). store ();
       nuevo cliente ("Máquinas de café asociadas"). store ();
       nuevo cliente ("Bilston Gasworks"). store ();
  }
  tienda privada vacía () {
       _instances.put (this.getName (), this);
```



```
  }
```

Ahora modifico el método de fábrica para devolver al cliente previamente preparado:

```
  Public static Customer create (String name) {
       return (Cliente) _instances.get (nombre);
  }
```

Debido a que el método de creación siempre devuelve un cliente existente, debería aclarar esto al utilizando el método de cambio de nombre .

```
clase Cliente ...
  public static Customer getNamed (String name) {
       return (Cliente) _instances.get (nombre);
  }
```

Cambiar referencia a valor

Tiene un objeto de referencia que es pequeño, inmutable y difícil de administrar.

Conviértalo en un objeto de valor.

Motivación

Al igual que con Cambiar valor a referencia , la decisión entre una referencia y un objeto de valor es No siempre está claro. Es una decisión que a menudo necesita revertirse.

El desencadenante para pasar de una referencia a un valor es trabajar con el objeto de referencia. se vuelve incómodo. Los objetos de referencia deben controlarse de alguna manera. Siempre necesitas solicite al controlador el objeto apropiado. Los enlaces de memoria también pueden ser incómodos. Valor Los objetos son particularmente útiles para sistemas distribuidos y concurrentes.

Una propiedad importante de los objetos de valor es que deben ser inmutables. Cada vez que invocas un consulta en uno, debe obtener el mismo resultado. Si esto es cierto, no hay problema en tener muchos Los objetos representan lo mismo. Si el valor es mutable, debe asegurarse de cambiar cualquier

Page 149

El objeto también actualiza todos los demás objetos que representan la misma cosa. Eso es mucho dolor que lo más fácil de hacer es convertirlo en un objeto de referencia.

Es importante tener claro lo que significa *inmutable* . Si tienes una clase de dinero con una moneda y un valor, que generalmente es un objeto de valor inmutable. Eso no significa que su salario no pueda cambio. Significa que para cambiar su salario, debe reemplazar el objeto de dinero existente con un nuevo objeto de dinero en lugar de cambiar la cantidad en un objeto de dinero existente. Tu la relación puede cambiar, pero el objeto monetario en sí no.

Mecánica

- Compruebe que el objeto candidato es inmutable o puede volverse inmutable.

  ? rarr; *Si el objeto no es inmutable actualmente, use* Eliminar configuración Método *hasta que sea.*

  ? rarr; *Si el candidato no puede volverse inmutable, debe abandonar Esta refactorización.*

- Crear un método igual y un método hash.
- Compilar y probar.

• Considere eliminar cualquier método de fábrica y hacer público un constructor.

Ejemplo

Comienzo con una clase de moneda:

```
Clase Moneda ...
   String privado _code;

   public String getCode () {
        código de retorno;
   }
   Moneda privada (código de cadena) {
        _code = code;
   }
```

Todo lo que hace esta clase es retener y devolver un código. Es un objeto de referencia, así que para obtener una instancia necesito usar

```
        Moneda usd = Currency.get ("USD");
```

La clase de moneda mantiene una lista de instancias. No puedo simplemente usar un constructor (es por eso que es privado).

```
nueva moneda ("USD"). es igual a (nueva moneda ("USD")) // devuelve falso
```

Para convertir esto en un objeto de valor, la clave es verificar que el objeto sea inmutable. Si no es así
No trato de hacer este cambio, ya que un valor mutable no causa un alias doloroso.

Page 150

En este caso, el objeto es inmutable, por lo que el siguiente paso es definir un método igual:

```
    public boolean equals (Objeto arg) {
        if (! (arg instanceof Currency)) devuelve falso;
        Moneda otra = (Moneda) arg;
        return (_code.equals (otro._code));
    }
```

Si defino iguales, también necesito definir hashCode. La manera simple de hacer esto es tomar el hash
códigos de todos los campos utilizados en el método de igualdad y hacer un biwise xor (^) en ellos. Aqui es facil porque solo hay uno:

```
    public int hashCode () {
        return _code.hashCode ();
    }
```

Con ambos métodos reemplazados, puedo compilar y probar. Necesito hacer las dos cosas; de lo contrario cualquier colección que depende de hashing, como Hashtable, HashSet o HashMap, puede actuar de manera extraña.

Ahora puedo crear tantas monedas iguales como quiera. Puedo deshacerme de todo el comportamiento del controlador en la clase y el método de fábrica y solo uso el constructor, que ahora puedo hacer público.

```
nueva moneda ("USD"). es igual a (nueva moneda ("USD")) // ahora devuelve verdadero
```

Reemplazar matriz con objeto

Tiene una matriz en la que ciertos elementos significan cosas diferentes.

Reemplace la matriz con un objeto que tenga un campo para cada elemento.

```
Cadena [] fila = nueva Cadena [3];
fila [0] = "Liverpool";
fila [1] = "15";
```

```
Fila de rendimiento = nuevo rendimiento ();
row.setName ("Liverpool");
row.setWins ("15");
```

Motivación

Las matrices son una estructura común para organizar datos. Sin embargo, deben usarse solo para contener Una colección de objetos similares en algún orden. A veces, sin embargo, ves que solían contener Una serie de cosas diferentes. Convenciones como "el primer elemento en la matriz es la persona nombre "son difíciles de recordar. Con un objeto puede usar nombres de campos y métodos para transmitir esta información para que no tenga que recordarla o esperar que los comentarios estén actualizados. Usted puede también encapsula la información y usa el método Move para agregarle comportamiento.

Mecánica

- Crear una nueva clase para representar la información en la matriz. Dale un campo público para el formación.
- Cambiar todos los usuarios de la matriz para usar la nueva clase.
- Compilar y probar.
- Uno por uno, agregue captadores y establecedores para cada elemento de la matriz. Nombra los accesores después del propósito del elemento de matriz. Cambiar los clientes para usar los accesores. Compilar y prueba después de cada cambio.
- Cuando todos los accesos a la matriz se reemplazan por métodos, haga que la matriz sea privada.
- Compilar.
- Para cada elemento de la matriz, cree un campo en la clase y cambie los accesores a usa el campo.
- Compilar y probar después de cambiar cada elemento.
- Cuando todos los elementos hayan sido reemplazados por campos, elimine la matriz.

Ejemplo

Comienzo con una matriz que se usa para contener el nombre, las victorias y las pérdidas de un equipo deportivo. Podría ser declarado de la siguiente manera:

```
Cadena [] fila = nueva Cadena [3];
```

Se usaría con código como el siguiente:

```
fila [0] = "Liverpool";
fila [1] = "15";

Nombre de cadena = fila [0];
int wins = Integer.parseInt (fila [1]);
```

Para convertir esto en un objeto, empiezo creando una clase:

```
rendimiento de clase {}
```

Para mi primer paso, le doy a la nueva clase un miembro de datos públicos. (Sé que esto es malo y malvado, pero lo haré reforma a su debido tiempo.)

```
Cadena pública [] _data = nueva Cadena [3];
```

Ahora encuentro los puntos que crean y acceden a la matriz. Cuando se crea la matriz, uso

```
Fila de rendimiento = nuevo rendimiento ();
```

Cuando se usa, cambio a

Página 152

```
fila ._data [0] = "Liverpool";
fila ._datos  [1] = "15";

Nombre de cadena = fila ._data [0];
int wins = Integer.parseInt (fila ._datos [1]);
```

Uno por uno, agrego más getters y setters significativos. Empiezo con el nombre:

```
rendimiento de clase ...
  public String getName () {
       return _data [0];
  }
  public void setName (String arg) {
       _data [0] = arg;
  }
```

Altero a los usuarios de esa fila para usar los captadores y establecedores en su lugar:

```
fila. setName ("Liverpool");
row._data [1] = "15";

Nombre de cadena = fila .getName ();
int wins = Integer.parseInt (row._data [1]);
```

Puedo hacer lo mismo con el segundo elemento. Para facilitar las cosas, puedo encapsular los datos
conversión de tipo:

```
rendimiento de clase ...
  public int getWins () {
       return Integer.parseInt (_data [1]);
  }
  public void setWins (String arg) {
       _data [1] = arg;
  }

....
codigo del cliente...
  row.setName ("Liverpool");
  fila. setWins ("15");

  Nombre de cadena = row.getName ();
  int gana = fila. getWins ();
```

Una vez que he hecho esto para cada elemento, puedo hacer que la matriz sea privada.

```
  Private String [] _data = new String [3];
```


Page 153

La parte más importante de esta refactorización, cambiar la interfaz, ya está hecha. También es útil
sin embargo, para reemplazar la matriz internamente. Puedo hacer esto agregando un campo para cada elemento de matriz y
cambiando los accesores para usarlo:

```
rendimiento de clase ...
  public String getName () {
       return _name;
  }
  public void setName (String arg) {
       _name = arg;
  }
  cadena privada _name;
```

Hago esto para cada elemento en la matriz. Cuando los he hecho todos, elimino la matriz.

Datos duplicados observados

Tiene datos de dominio disponibles solo en un control GUI, y los métodos de dominio necesitan acceso.

Copie los datos a un objeto de dominio. Configure un observador para sincronizar las dos piezas de datos.

Motivación

Un sistema bien organizado separa el código que maneja la interfaz de usuario del código que maneja el lógica de negocios. Hace esto por varias razones. Es posible que desee varias interfaces para similares lógica de negocios; la interfaz de usuario se vuelve demasiado complicada si hace ambas cosas; es más fácil de mantener y evolucionar objetos de dominio separados de la GUI; o puede tener diferentes desarrolladores manejando Las diferentes piezas.

Aunque el comportamiento se puede separar fácilmente, los datos a menudo no. Los datos deben ser integrado en el control GUI que tiene el mismo significado que los datos que viven en el modelo de dominio. Usuario Los marcos de interfaz, desde el modelo-vista-controlador (MVC) en adelante, utilizaron un sistema de varios niveles para Proporcionar mecanismos que le permitan proporcionar estos datos y mantener todo sincronizado.

Page 154

Si encuentra un código que se ha desarrollado con un enfoque de dos niveles en el que las empresas la lógica está incrustada en la interfaz de usuario, debe separar los comportamientos. Gran parte de esto es sobre descomposición y métodos de movimiento. Para los datos, sin embargo, no puede simplemente mover los datos, tienes que duplicarlo y proporcionar el mecanismo de sincronización.

Mecánica

- Haga que la clase de presentación sea un observador de la clase de dominio [Banda de cuatro].

  ? rarr; *Si todavía no hay una clase de dominio, cree una.*

  ? rarr; *Si no hay un enlace desde la clase de presentación a la clase de dominio, coloque la clase de dominio en un campo de la clase de presentación.*

- Use el campo Self Encapsulate en los datos del dominio dentro de la clase GUI.
- Compilar y probar.
- Agregue una llamada al método de configuración en el controlador de eventos para actualizar el componente con su valor actual usando acceso directo.

  ? rarr; *Ponga un método en el controlador de eventos que actualice el valor de componente sobre la base de su valor actual. Por supuesto esto es completamente innecesario; solo está configurando el valor a su valor actual, pero por utilizando el método de configuración, permite que se ejecute cualquier comportamiento allí.*

  ? rarr; *Cuando realice este cambio, no utilice el método de obtención para componente; use el acceso directo al componente. Más tarde el conseguir El método extraerá el valor del dominio, que no cambia hasta*

Se ejecuta el método de configuración.

? rarr; *Asegúrese de que la prueba active el mecanismo de manejo de eventos código.*

- Compilar y probar.
- Definir los datos y los métodos de acceso en la clase de dominio.

? rarr; *Asegúrese de que el método de configuración en el dominio active la notificación mecanismo en el patrón observador.*

? rarr; *Use el mismo tipo de datos en el dominio que está en la presentación (generalmente una cadena). Convierta el tipo de datos en una refactorización posterior.*

- Redirigir los accesores para escribir en el campo de dominio.
- Modificar el método de actualización del observador para copiar los datos del campo de dominio a la GUI controlar.
- Compilar y probar.

Ejemplo

Comienzo con la ventana de la Figura 8.1 . El comportamiento es muy simple. Cada vez que cambias el valor en uno de los campos de texto, los otros se actualizan. Si cambia los campos de inicio o fin, el se calcula la longitud; Si cambia el campo de longitud, se calcula el final.

Figura 8.1. Una ventana GUI simple

Todos los métodos están en una sola clase `IntervalWindow` . Los campos están configurados para responder a la pérdida. de enfoque desde el campo.

```
clase pública IntervalWindow extiende Frame ...
  java.awt.TextField _startField;
  java.awt.TextField _endField;
  java.awt.TextField _lengthField;

  clase SymFocus extiende java.awt.event.FocusAdapter
  {
       public void focusLost (evento java.awt.event.FocusEvent)
```

```
        {
             Objeto objeto = event.getSource ();
             if (objeto == _startField)
                  StartField_FocusLost (evento);
             si no (objeto == _endField)
                  EndField_FocusLost (evento);
             si no (objeto == _lengthField)
                  LengthField_FocusLost (evento);
        }
   }
```

El oyente reacciona llamando a `StartField_FocusLost` cuando se pierde el foco en el campo de inicio y `EndField_FocusLost` y `LengthField_FocusLost` para los otros campos. Estos eventos los métodos de manejo se ven así:

```
   nulo StartField_FocusLost (evento java.awt.event.FocusEvent) {
        if (isNotInteger (_startField.getText ()))
             _startField.setText ("0");
        calcularLongitud ();
   }

   nulo EndField_FocusLost (evento java.awt.event.FocusEvent) {
        if (isNotInteger (_endField.getText ()))
             _endField.setText ("0");
```


Page 156

```
        calcularLongitud ();
   }

   nulo LengthField_FocusLost (evento java.awt.event.FocusEvent) {
        if (isNotInteger (_lengthField.getText ()))
             _lengthField.setText ("0");
        CalculateEnd ();
   }
```

Si se pregunta por qué hice la ventana de esta manera, lo hice de la manera más fácil con mi IDE (Cafe) me animó a hacerlo.

Todos los campos insertan un cero si aparecen caracteres no enteros y llaman al cálculo relevante rutina:

```
  void CalculateLength () {
       tratar {
         int start = Integer.parseInt (_startField.getText ());
         int end = Integer.parseInt (_endField.getText ());
         int longitud = fin - inicio;
         _lengthField.setText (String.valueOf (longitud));
       } catch (NumberFormatException e) {
         lanzar nueva RuntimeException ("Error de formato de número inesperado");
       }
}
nulo CalculateEnd () {
     tratar {
       int start = Integer.parseInt (_startField.getText ());
       int longitud = Integer.parseInt (_lengthField.getText ());
       int end = inicio + longitud;
```

```
      endField.setText (String.valueOf (final));
    } catch (NumberFormatException e) {
      lanzar nueva RuntimeException ("Error de formato de número inesperado");
    }
}
```

Mi tarea general, si decido aceptarlo, es separar la lógica no visual de la GUI.
Esencialmente, esto significa mover `CalculateLength` y `CalculateEnd` a un dominio separado
clase. Para hacer esto, necesito consultar los datos de inicio, finalización y longitud *sin hacer* referencia a la ventana
clase. La única forma en que puedo hacer esto es duplicar estos datos en la clase de dominio y sincronizar el
datos con la GUI. Esta tarea se describe mediante datos observados duplicados .

Actualmente no tengo una clase de dominio, así que creo una (vacía):

```
El intervalo de clase se extiende Observable {}
```

La ventana de intervalo necesita un enlace a esta nueva clase de dominio.

```
Intervalo privado _subject;
```


Entonces necesito inicializar correctamente este campo y hacer que la ventana de intervalo sea un observador del intervalo. yo
puede hacer esto poniendo el siguiente código en el constructor de la ventana de intervalo:

```
_subject = new Interval ();
_subject.addObserver (esto);
actualización (_subject, null);
```

Me gusta poner este código al final del proceso de construcción. La llamada a la actualización asegura que a medida que yo
duplicar los datos en la clase de dominio, la GUI se inicializa desde la clase de dominio. Para hacer esto yo
necesita declarar que la ventana de intervalo implementa `Observer`:

```
clase pública IntervalWindow extiende Frame implementa observador
```

Para implementar el observador, necesito crear un método de actualización. Por el momento esto puede estar en blanco:

```
Actualización pública nula (Observable observado, Objeto arg) {
}
```

Puedo compilar y probar en este punto. Todavía no he realizado ningún cambio real, pero puedo cometer errores
en los lugares mas simples.

Ahora puedo centrar mi atención en los campos en movimiento. Como de costumbre, hago los cambios un campo a la vez. A
demostrar mi dominio del idioma inglés, comenzaré con el campo final. La primera tarea es
aplicar el campo auto encapsulado . Los campos de texto se actualizan con los métodos `getText` y `setText` . yo
crear accesores que llaman a estos

```
Cadena getEnd () {
```

```
        return _endField.getText ();
    }

    void setEnd (String arg) {
        _endField.setText (arg);
    }
```

Encuentro todas las referencias a _endField y las reemplazo con los accesores apropiados:

```
    void CalculateLength () {
       tratar {
          int start = Integer.parseInt (_startField.getText ());
          int end = Integer.parseInt ( getEnd ());
          int longitud = fin - inicio;
          _lengthField.setText (String.valueOf (longitud));
       } catch (NumberFormatException e) {
          lanzar nueva RuntimeException ("Error de formato de número inesperado");
       }
    }

    nulo CalculateEnd () {
       tratar {
```


Page 158

```
          int start = Integer.parseInt (_startField.getText ());
          int longitud = Integer.parseInt (_lengthField.getText ());
          int end = inicio + longitud;
          setEnd (String.valueOf (end));
       } catch (NumberFormatException e) {
          lanzar nueva RuntimeException ("Error de formato de número inesperado");
       }
}

    nulo EndField_FocusLost (evento java.awt.event.FocusEvent) {
        if (isNotInteger ( getEnd ()))
            setEnd ("0");
        calcularLongitud ();
     }
```

Ese es el proceso normal para el campo Self Encapsulate Field . Sin embargo, cuando trabajas con un GUI, hay una complicación. El usuario puede cambiar el valor del campo directamente sin llamar a `setEnd`. Entonces necesito hacer una llamada para establecer `End` en el controlador de eventos para la GUI. Esta llamada cambia el valor del campo final al valor actual del campo final. Por supuesto, esto no hace nada por el momento, pero asegura que la entrada del usuario pase por el método de configuración:

```
    nulo EndField_FocusLost (evento java.awt.event.FocusEvent) {
        setEnd (_endField.getText ());
        if (isNotInteger (getEnd ()))
            setEnd ("0");
        calcularLongitud ();
    }
```

En esta llamada no uso `getEnd;` en cambio accedo al campo directamente. Hago esto porque más tarde en el refactorizar `getEnd` obtiene un valor del objeto de dominio, no del campo. En ese punto usándolo significaría que cada vez que el usuario cambiara el valor del campo, este código cámbielo nuevamente, así que aquí debo usar el acceso directo. En este punto puedo compilar y probar el

comportamiento encapsulado

Ahora agrego el campo final a la clase de dominio:

```
intervalo de clases...
   cadena privada _end = "0";
```

Lo inicializo al mismo valor que se inicializa en la GUI. Ahora agrego métodos de obtención y configuración:

```
intervalo de clases...

   Cadena getEnd () {
        volver _end;
   }
   void setEnd (String arg) {
        _end = arg;
        setChanged ();
        notifyObservers ();
   }
```


Page 159

Como estoy usando el patrón de observador, tengo que agregar el código de notificación en la configuración método. Yo uso una cadena, no un número (más lógico). Esto es porque quiero hacer el más pequeño posible cambio Una vez que haya duplicado con éxito los datos, puedo cambiar el tipo de datos internos a un entero.

Ahora puedo hacer una compilación y prueba más antes de realizar la duplicación. Haciendo todo esto trabajo preparatorio, he minimizado el riesgo en este paso complicado.

El primer cambio es actualizar los accesos en `IntervalWindow` para usar `Interval`.

```
clase IntervalWindow ...
   Cadena getEnd () {
        return _subject.getEnd ();
   }
   void setEnd (String arg) {
        _subject.setEnd (arg);
   }
```

También necesito actualizar la `actualización` para asegurarme de que la GUI reacciona a la notificación:

```
clase IntervalWindow ...
   Actualización pública nula (Observable observado, Objeto arg) {
      _endField.setText (_subject.getEnd ());
   }
```

Este es el otro lugar donde tengo que usar el acceso directo. Si tuviera que llamar al método de configuración, yo entraría en una recursión infinita.

Ahora puedo compilar y probar, y los datos se duplican correctamente.

Puedo repetir para los otros dos campos. Una vez hecho esto, puedo aplicar el Método de movimiento para mover

clase que contiene todo el comportamiento y los datos del dominio y lo separa del código GUI.

Si he hecho esto, considero deshacerme de la clase GUI por completo. Si mi clase es una AWT más antigua clase, puedo obtener una mejor apariencia usando Swing, y Swing hace un mejor trabajo con la coordinación. yo puedo construya la GUI de Swing sobre la clase de dominio. Cuando estoy contento, puedo eliminar la vieja clase GUI.

Usar oyentes de eventos

Los datos observados duplicados también se aplican si utiliza escuchas de eventos en lugar de observadores / observables. En este caso, debe crear un detector y un evento en el modelo de dominio (o puede usar clases de AWT si no te importa la dependencia). El objeto de dominio necesita registrar el los oyentes de la misma manera que lo hace el observable y les envía un evento cuando cambia, como en El método de actualización. La ventana de intervalo puede usar una clase interna para implementar el oyente interfaz y llame a los métodos de actualización adecuados.

Cambiar asociación unidireccional a bidireccional

Tiene dos clases que necesitan usar las características de cada uno, pero solo hay un enlace unidireccional.

159

160 de 1189.

Agregue punteros de nuevo y cambie los modificadores para actualizar ambos conjuntos

Motivación

Puede descubrir que inicialmente ha configurado dos clases para que una clase se refiera a la otra. Terminado cada vez que encuentre que un cliente de la clase referida necesita llegar a los objetos que se refieren a ella. Esta efectivamente significa navegar hacia atrás a lo largo del puntero. Los punteros son enlaces unidireccionales, por lo que no puede hacer esto. A menudo puede solucionar este problema buscando otra ruta. Esto puede costar en cómputo pero es razonable, y puede tener un método en la clase referida que use este comportamiento. A veces, sin embargo, esto no es fácil, y debe configurar una referencia bidireccional, a veces llamado un *puntero hacia atrás.* Si no está acostumbrado a retroceder los punteros, es fácil enredarse hasta usarlos. Sin embargo, una vez que te acostumbras al idioma, no es demasiado complicado.

El idioma es lo suficientemente incómodo como para que te hagan pruebas, al menos hasta que te sientas cómodo con el idioma Como generalmente no me molesto en probar los accesores (el riesgo no es lo suficientemente alto), esto es

El raro caso de una refactorización que agrega una prueba.

Esta refactorización utiliza punteros de retroceso para implementar la bidireccionalidad. Otras técnicas, como el enlace. objetos, requieren otras refactorizaciones.

Mecánica

- Agregue un campo para el puntero posterior.
- Decidir qué clase controlará la asociación.
- Cree un método auxiliar en el lado no controlador de la asociación. Nombra este método para indicar claramente su uso restringido.
- Si el modificador existente está en el lado de control, modifíquelo para actualizar los punteros posteriores.
- Si el modificador existente está en el lado controlado, cree un método de control en el lado controlador y llamarlo desde el modificador existente.

Ejemplo

Un programa simple tiene un pedido que se refiere a un cliente:


```
Orden de clase ...
   Cliente getCustomer () {
        volver _cliente;
   }
   void setCustomer (Customer arg) {
        _customer = arg;
   }
   Cliente _cliente;
```

La clase de cliente no tiene referencia al pedido.

Comienzo la refactorización agregando un campo al cliente. Como un cliente puede tener varios pedidos, entonces este campo es una colección. Porque no quiero que un cliente tenga el mismo pedido más de Una vez en su colección, la colección correcta es un conjunto:

```
clase Cliente {
   Private Set _orders = new HashSet ();
```

Ahora necesito decidir qué clase se hará cargo de la asociación. Prefiero dejar que tome una clase cargar porque mantiene toda la lógica para manipular la asociación en un solo lugar. Mi decisión el proceso se ejecuta de la siguiente manera:

1. Si ambos objetos son objetos de referencia y la asociación es de uno a muchos, entonces el objeto que tiene la única referencia es el controlador. (Es decir, si un cliente tiene muchos pedidos, la orden controla la asociación).
2. Si un objeto es un componente del otro, el compuesto debe controlar la asociación.
3. Si ambos objetos son objetos de referencia y la asociación es de muchos a muchos, no importa si el pedido o el cliente controla la asociación.

Debido a que el pedido se hará cargo, necesito agregar un método auxiliar al cliente que permita Acceso directo a la recogida de pedidos. El modificador de la orden usará esto para sincronizar ambos conjuntos de punteros Uso el nombre `friendOrders` para indicar que este método solo se utilizará en este caso especial. También minimizo su visibilidad haciéndola visibilidad del paquete si es posible. Yo tengo para hacerlo público si la otra clase está en otro paquete:

```
clase Cliente ...
   Establecer friendOrders () {
   / ** solo debe ser usado por Order cuando se modifica la asociación * /
        volver _orders;
   }
```

Ahora actualizo el modificador para actualizar los punteros posteriores:

```
Orden de clase ...
   void setCustomer (Customer arg) ...
        if (_customer! = null) _customer.friendOrders (). remove (this);
        _customer = arg;
        if (_customer! = null) _customer.friendOrders (). add (this);
   }
```


El código exacto en el modificador de control varía con la multiplicidad de la asociación. Si el
No se permite que el cliente sea nulo, puedo renunciar a los cheques nulos, pero necesito verificar si hay nulos
argumento. Sin embargo, el patrón básico es siempre el mismo: primero dígale al otro objeto que elimine su
puntero hacia usted, establezca su puntero en el nuevo objeto y luego dígale al nuevo objeto que agregue un puntero a
tú.

Si desea modificar el enlace a través del cliente, deje que llame al método de control:

```
clase Cliente ...
   void addOrder (Order arg) {
        arg.setCustomer (esto);
   }
```

Si un pedido puede tener muchos clientes, tiene un caso de muchos a muchos, y los métodos se ven como
esta:

```
orden de clase ... // métodos de control
   void addCustomer (Customer arg) {
        arg.friendOrders (). add (this);
        _customers.add (arg);
   }
   void removeCustomer (Customer arg) {
        arg.friendOrders (). remove (this);
        _customers.remove (arg);
   }

clase Cliente ...
   void addOrder (Order arg) {
        arg.addCustomer (esto);
   }
   void removeOrder (Order arg) {
        arg.removeCustomer (esto);
   }
```

Cambiar la asociación bidireccional a unidireccional

Tiene una asociación bidireccional, pero una clase ya no necesita características de la otra.

Descarte el final innecesario de la asociación.

Page 163

Motivación

Las asociaciones bidireccionales son útiles, pero tienen un precio. El precio es la complejidad añadida de mantener los enlaces bidireccionales y garantizar que los objetos se creen y eliminen correctamente.
Las asociaciones bidireccionales no son naturales para muchos programadores, por lo que a menudo son una fuente de errores

Muchos enlaces bidireccionales también facilitan que los errores conduzcan a zombis: objetos que deberían ser muerto, pero aún permanece por una referencia que no se borró.

Las asociaciones bidireccionales fuerzan una interdependencia entre las dos clases. Cualquier cambio a uno clase puede causar un cambio a otro. Si las clases están en paquetes separados, obtienes un interdependencia entre los paquetes. Muchas interdependencias conducen a una relación altamente acoplada. sistema, en el que cualquier pequeño cambio conduce a muchas ramificaciones impredecibles.

Debe usar asociaciones bidireccionales cuando lo necesite, pero no cuando no lo necesite. Tan pronto como ves que una asociación bidireccional ya no está tirando de su peso, deja caer el extremo innecesario.

Mecánica

- Examine todos los lectores del campo que contiene el puntero que desea eliminar para ver si la eliminación es factible

  ? rarr; *Mire a los lectores directos y otros métodos que los llaman métodos.*

  ? rarr; *Considere si es posible determinar el otro objeto sin usar el puntero. Si es así, podrá usar* Sustituto Algoritmo *en el captador para permitir a los clientes utilizar el método de obtención incluso si No hay puntero.*

  ? rarr; *Considere agregar el objeto como argumento a todos los métodos que usan el campo.*

Page 164

- Si los clientes necesitan usar el captador, use el Campo de autoencapsulación , realice Sustituto Algoritmo en el captador, compilación y prueba.
- Si los clientes no necesitan el captador, cambie cada usuario del campo para que ingrese el objeto El campo de otra manera. Compile y pruebe después de cada cambio.
- Cuando no quede ningún lector en el campo, elimine todas las actualizaciones del campo y elimine el campo.

  ? rarr; *Si hay muchos lugares que asignan el campo, use* Self Encapsular campo *para que todos usen un solo setter. Compilar y probar.*

  Cambia el setter para tener un cuerpo vacío. Compilar y probar. Si eso funciona, elimina el campo, el setter y todas las llamadas al setter.

- Compilar y probar.

Ejemplo

Comienzo desde donde terminé del ejemplo en Change Unidirectional Association a Bidireccional . Tengo un cliente y un pedido con un enlace bidireccional:

```
Orden de clase ...
  Cliente getCustomer () {
       volver _cliente;
  }
  void setCustomer (Customer arg) {
       if (_customer! = null) _customer.friendOrders (). remove (this);
       _customer = arg;
       if (_customer! = null) _customer.friendOrders (). add (this);
  }
  Cliente privado _cliente;

clase Cliente ...
  void addOrder (Order arg) {
       arg.setCustomer (esto);
  }
  Private Set _orders = new HashSet ();
  Establecer friendOrders () {
       / ** solo debe usarse por orden * /
       volver _orders;
  }
```

Descubrí que en mi solicitud no tengo pedidos a menos que ya tenga un cliente, así que quiero romper el enlace del pedido al cliente

La parte más difícil de esta refactorización es comprobar que puedo hacerlo. Una vez que sé que es seguro hacerlo, es fácil. El problema es si el código depende de que el campo del cliente esté allí. Para eliminar el campo, necesito proporcionar una alternativa.

Mi primer paso es estudiar a todos los lectores del campo y los métodos que usan esos lectores. Puedo encuentra otra forma de proporcionar el objeto de cliente? A menudo esto significa pasar al cliente como argumento para una operación. Aquí hay un ejemplo simplista de esto:

```
Orden de clase ...
       doble getDiscountedPrice () {
```


Page 165

```
            return getGrossPrice () * (1 - _customer.getDiscount ());
      }
```

cambios a

```
Orden de clase ...
      double getDiscountedPrice (Cliente cliente) {
            return getGrossPrice () * (1 - customer.getDiscount ());
      }
```

Esto funciona particularmente bien cuando el cliente llama el comportamiento, porque entonces es fácil de pasar como argumento. Entonces

```
clase Cliente ...
      doble getPriceFor (orden de pedido) {
            Assert.isTrue (_orders.contains (orden)); // ver Introducir
Afirmación (267)
            return order.getDiscountedPrice ();
```

se convierte

```
clase Cliente ...
      doble getPriceFor (orden de pedido) {
            Assert.isTrue (_orders.contains (orden));
            return order.getDiscountedPrice (esto);
      }
```

Otra alternativa que considero es cambiar el captador para que llegue al cliente sin usar el campo. Si lo hace, puedo usar el algoritmo sustituto en el cuerpo de `Order.getCustomer`. Yo podría haz algo como esto:

```
      Cliente getCustomer () {
            Iterator iter = Customer.getInstances (). Iterator ();
            while (iter.hasNext ()) {
                  Cliente cada = (Cliente) iter.next ();
                  if (each.containsOrder (this)) devuelve cada uno;
            }
```

```
        volver nulo;
    }
```

Lento, pero funciona. En un contexto de base de datos, puede que ni siquiera sea tan lento si uso una consulta de base de datos. Si la clase de orden contiene métodos que usan el campo del cliente, puedo cambiarlos para usar `getCustomer` usando el campo Self Encapsulate Field .

Si conservo el descriptor de acceso, la asociación sigue siendo bidireccional en la interfaz pero es unidireccional en implementación. Elimino el backpointer pero mantengo las interdependencias entre los dos clases

Si sustituyo el método de obtención, lo sustituyo y dejo el resto para más tarde. De lo contrario cambio las personas que llaman una a la vez para usar al cliente de otra fuente. Compilo y pruebo después de cada

cambio. En la práctica, este proceso suele ser bastante rápido. Si fuera complicado, me rendiría Esta refactorización.

Una vez que he eliminado a los lectores del campo, puedo trabajar en los escritores del campo. Esto es como simple como eliminar cualquier asignación al campo y luego eliminar el campo. Porque nadie es leerlo más, eso no debería importar.

Reemplazar número mágico con constante simbólica

Tienes un número literal con un significado particular.

Crea una constante, nómbrala después del significado y reemplaza el número con ella.

```
doble potencial Energía (doble masa, doble altura) {
     masa de retorno * 9.81 * altura;
}
```

```
doble potencial Energía (doble masa, doble altura) {
     return mass * GRAVITATIONAL_CONSTANT * height;
}
doble final estático GRAVITATIONAL_CONSTANT = 9.81;
```

Motivación

Los números mágicos son uno de los males más antiguos de la informática. Son números con valores especiales que Por lo general no son obvios. Los números mágicos son realmente desagradables cuando necesitas hacer referencia al mismo Número lógico en más de un lugar. Si los números alguna vez cambian, hacer el cambio es una pesadilla. Incluso si no hace un cambio, tiene la dificultad de descubrir qué está pasando en.

Muchos idiomas le permiten declarar una constante. No hay costo en el rendimiento y hay un Gran mejora en la legibilidad.

Antes de realizar esta refactorización, siempre debe buscar una alternativa. Mira como la magia Se utiliza el número. A menudo puede encontrar una mejor manera de usarlo. Si el número mágico es un código de tipo, considere Reemplazar código de tipo con clase . Si el número mágico es la longitud de una matriz, use `anArray.length` en `su` lugar cuando estás recorriendo la matriz.

Mecánica

- Declare una constante y ajústela al valor del número mágico.
- Encuentra todas las ocurrencias del número mágico.
- Ver si el número mágico coincide con el uso de la constante; si es así, cambie el número mágico para usar la constante.
- Compilar.
- Cuando se cambian todos los números mágicos, compilar y probar. En este punto, todo debería funcionar como si nada ha cambiado

Page 167

? rarr; *Una buena prueba es ver si puede cambiar la constante fácilmente.*
Esto puede significar alterar algunos resultados esperados para que coincidan con el nuevo valor.
Esto no siempre es posible, pero es un buen truco cuando funciona.

Campo encapsulado

Hay un campo público.

Hazlo privado y proporciona accesores.

```
cadena pública _name
```

```
cadena privada _name;
public String getName () {return _name;}
public void setName (String arg) {_name = arg;}
```

Motivación

Uno de los principios principales de la orientación a objetos es la encapsulación o la ocultación de datos. Esto dice que nunca debe hacer públicos sus datos. Cuando haces públicos los datos, otros objetos pueden cambiar y acceder a valores de datos sin que el objeto propietario lo sepa. Esto separa los datos de comportamiento.

Esto se ve como algo malo porque reduce la modularidad del programa. Cuando los datos y el comportamiento que lo usa se agrupan, es más fácil cambiar el código, porque el cambio el código está en un lugar en lugar de estar disperso por todo el programa.

Encapsulate Field comienza el proceso ocultando los datos y agregando accesores. Pero esto es solo el primer paso. Una clase con solo accesos es una clase tonta que realmente no aprovecha las oportunidades de los objetos, y un objeto es algo terrible que desperdiciar. Una vez que termine de encapsular Campo Busco métodos que usan los nuevos métodos para ver si les gusta empacar sus maletas y moviéndose al nuevo objeto con un Método de movimiento rápido .

Mecánica

- Crear métodos de obtención y configuración para el campo.
- Encuentre todos los clientes fuera de la clase que hacen referencia al campo. Si el cliente usa el valor, reemplace la referencia con una llamada al método de obtención. Si el cliente cambia el valor, reemplace la referencia con una llamada al método de configuración.

? rarr; *Si el campo es un objeto y el cliente invoca un modificador en el objeto, eso es un uso. Solo use el método de configuración para reemplazar un asignación.*

- Compilar y probar después de cada cambio.
- Una vez que se cambian todos los clientes, declare el campo como privado.
- Compilar y probar.

Page 168

Colección encapsulada

Un método devuelve una colección.

Haga que devuelva una vista de solo lectura y proporcione métodos de agregar / quitar.

Motivación

A menudo, una clase contiene una colección de instancias. Esta colección puede ser una matriz, lista, conjunto o vector. Tales casos a menudo tienen el captador y colocador habitual para la colección.

Sin embargo, las colecciones deben usar un protocolo ligeramente diferente al de otros tipos de datos. los getter no debe devolver el objeto de colección en sí, porque eso permite a los clientes manipular el contenido de la colección sin que la clase propietaria sepa lo que está sucediendo. También revela también mucho a los clientes sobre las estructuras de datos internos del objeto. Un captador para un atributo multivalor debe devolver algo que evite la manipulación de la colección y oculte innecesariamente Detalles sobre su estructura. La forma de hacerlo varía según la versión de Java que sea. utilizando.

Además, no debería haber un setter para la colección: más bien debería haber operaciones para agregar y eliminar elementos. Esto le da al propietario el control sobre la adición y eliminación de elementos. De la colección.

Con este protocolo, la colección se encapsula correctamente, lo que reduce el acoplamiento de poseer clase a sus clientes.

Mecánica

- Agregue un método de agregar y quitar para la colección.
- Inicialice el campo a una colección vacía.
- Compilar.
- Buscar llamadas del método de configuración. Modifique el método de configuración para usar el complemento y eliminar operaciones o hacer que los clientes llamen a esas operaciones en su lugar.

? rarr; *Los setters se usan en dos casos: cuando la colección está vacía y cuando el setter está reemplazando una colección no vacía.*

? rarr; *Es posible que desee utilizar el* método *de cambio de nombre para cambiar el nombre del configurador. Cámbielo del conjunto para inicializar o reemplazar.*

- Compilar y probar.
- Encuentra todos los usuarios del captador que modifican la colección. Cámbielos para usar el complemento y eliminar métodos Compile y pruebe después de cada cambio.

- Cuando se hayan cambiado todos los usos del captador que modifican, modifique el captador para devolver un vista de solo lectura de la colección.

? rarr; *En Java 2, esta es la vista de colección no modificable adecuada.*

? rarr; *En Java 1.1, debe devolver una copia de la colección.*

Compilar y probar.

- Encuentra los usuarios del captador. Busque el código que debería estar en el objeto host. Usar extracto Método y Método de movimiento para mover el código al objeto host.

Para Java 2, ya ha terminado con eso. Sin embargo, para Java 1.1, los clientes pueden preferir usar un enumeración. Para proporcionar la enumeración:

- Cambie el nombre del captador actual y agregue un captador nuevo para devolver una enumeración. Encuentre usuarios del antiguo captador y cámbielos para usar uno de los nuevos métodos.

? rarr; *Si este es un salto demasiado grande, usa el* método Rename *en el getter anterior, cree un nuevo método que devuelva una enumeración y cambie las llamadas a Usa el nuevo método.*

- Compilar y probar.

Ejemplos

Java 2 agregó un grupo completamente nuevo de clases para manejar colecciones. No solo agregó nuevas clases pero también alteró el estilo de usar colecciones. Como resultado, la forma de encapsular una colección es diferente dependiendo de si utiliza las colecciones Java 2 o las colecciones Java 1.1. yo discuta primero el enfoque de Java 2, porque espero que las colecciones de Java 2 más funcionales desplazar las colecciones Java 1.1 durante la vida de este libro.

Ejemplo: Java 2

Una persona está tomando cursos. Nuestro curso es bastante simple:

```
Curso de clase ...
  Curso público (String name, boolean isAdvanced) {...};
  public boolean isAdvanced () {...};
```

No me voy a molestar con nada más en el curso. La clase interesante es la persona:

```
Persona de clase ...
  setCenter público getCourses () {
       volver _ cursos;
  }
  public void setCourses (Set arg) {
       _courses = arg;
  }
  Set privado _ cursos;
```


Page 170

Con esta interfaz, los clientes agregan cursos con código como

```
Persona kent = nueva Persona ();
Establecer s = new HashSet ();
s.add (nuevo curso ("Programación Smalltalk", falso));
s.add (nuevo curso ("Apreciando maltas individuales", verdadero));
kent.setCourses (s);
Assert.equals (2, kent.getCourses (). Size ());
Curso refact = nuevo curso ("Refactorización", verdadero);
kent.getCourses (). add (refact);
kent.getCourses (). add (nuevo curso ("Brutal Sarcasm",
falso));
Assert.equals (4, kent.getCourses (). Size ());
kent.getCourses (). remove (refact);
Assert.equals (3, kent.getCourses (). Size ());
```

Un cliente que quiere saber sobre cursos avanzados puede hacerlo de esta manera:

```
Iterator iter = person.getCourses (). Iterator ();
int cuenta = 0;
while (iter.hasNext ()) {
    Curso cada = (Curso) iter.next ();
    if (each.isAdvanced ()) count ++;
}
```

Lo primero que quiero hacer es crear los modificadores adecuados para la recopilación y compilación, como sigue:

```
Persona de clase
  public void addCourse (Curso arg) {
       _courses.add (arg);
  }
  public void removeCourse (Curso arg) {
       _courses.remove (arg);
  }
```

La vida será más fácil si inicializo el campo también:

```
privado Set _courses = new HashSet ();
```

Luego miro a los usuarios del setter. Si hay muchos clientes y el setter se usa mucho, yo necesita reemplazar el cuerpo del setter para usar las operaciones de agregar y quitar. La complejidad de Este proceso depende de cómo se utiliza el setter. Hay dos casos. En el caso más simple, el

el cliente usa el setter para inicializar los valores, es decir, no hay cursos antes de que el setter sea aplicado. En este caso, reemplazo el cuerpo del setter para usar el método add.

```
Persona de clase ...
  public void setCourses (Set arg) {
       Assert.isTrue (_courses.isEmpty ());
```


Página 171

```
       Iterador iter = arg.iterator ();
       while (iter.hasNext ()) {
            addCourse ((Curso) iter.next ());
       }
   }
```

Después de cambiar el cuerpo de esta manera, es aconsejable usar el método Rename para hacer la intención más claro

```
public void initializeCourses (Establecer arg) {
     Assert.isTrue (_courses.isEmpty ());
     Iterador iter = arg.iterator ();
     while (iter.hasNext ()) {
          addCourse ((Curso) iter.next ());
     }
}
```

En el caso más general, tengo que usar el método remove para eliminar cada elemento primero y luego agregue los elementos. Pero encuentro que esto ocurre raramente (como suelen ocurrir en casos generales).

Si sé que no tengo ningún comportamiento adicional al agregar elementos al inicializar, puedo elimine el bucle y use `addAll`.

```
public void initializeCourses (Establecer arg) {
     Assert.isTrue (_courses.isEmpty ());
     _courses.addAll (arg);
}
```

No puedo simplemente asignar el conjunto, aunque el conjunto anterior estaba vacío. Si el cliente simplemente crea un establecer y usar el setter, puedo hacer que usen el complemento para modificar el conjunto después de pasarlo, eso violaría la encapsulación. Tengo que hacer una copia.

Si los clientes simplemente crean un conjunto y usan el setter, puedo hacer que usen agregar y quitar métodos directamente y retire el setter por completo. Código como

```
Persona kent = nueva Persona ();
Establecer s = new HashSet ();
s.add (nuevo curso ("Programación Smalltalk", falso));
s.add (nuevo curso ("Apreciando maltas individuales", verdadero));
kent.initializeCourses (s);
```

se convierte

```
Persona kent = nueva Persona ();
```

```
                kent.addCourse (nuevo curso ("Programación Smalltalk",
falso));
                kent.addCourse (nuevo curso ("Apreciando maltas individuales",
cierto));
```


Ahora empiezo a mirar a los usuarios del captador. Mi primera preocupación son los casos en que alguien usa el getter para modificar la colección subyacente, por ejemplo:

```
kent.getCourses (). add (nuevo curso ("Brutal Sarcasm", false));
```

Necesito reemplazar esto con una llamada al nuevo modificador:

```
Kent addCourse (nuevo curso ("Sarcasmo brutal", falso));
```

Una vez que he hecho esto para todos, puedo verificar que nadie esté modificando a través del getter cambiando el cuerpo del captador para devolver una vista inmodificable:

```
setCenter público getCourses () {
     return Collections.unmodifiableSet (_courses);
}
```

En este punto, he encapsulado la colección. Nadie puede cambiar los elementos de la colección. excepto a través de métodos en la persona.

Mover el comportamiento a la clase

Tengo la interfaz correcta. Ahora me gusta mirar a los usuarios del getter para encontrar el código que debería ser en persona Código como

```
                Iterator iter = person.getCourses (). Iterator ();
                int cuenta = 0;
                while (iter.hasNext ()) {
                    Curso cada = (Curso) iter.next ();
                    if (each.isAdvanced ()) count ++;
                }
```

es mejor moverlo a persona porque solo usa los datos de la persona. Primero uso el método de extracción en código:

```
int numberOfAdvancedCourses (Persona persona) {
      Iterator iter = person.getCourses (). Iterator ();
      int cuenta = 0;
      while (iter.hasNext ()) {
           Curso cada = (Curso) iter.next ();
           if (each.isAdvanced ()) count ++;
      }
      cuenta de retorno;
 }
```

Y luego uso Move Method para moverlo a persona:

```
Persona de clase ...
```

172

Page 173

```
int numberOfAdvancedCourses () {
     Iterador iter = getCourses (). Iterator ();
     int cuenta = 0;
     while (iter.hasNext ()) {
          Curso cada = (Curso) iter.next ();
          if (each.isAdvanced ()) count ++;
     }
     cuenta de retorno;
}
```

Un caso común es

```
kent.getCourses (). size ()
```

que se puede cambiar a la más legible

```
kent.numberOfCourses ()

Persona de clase ...
  public int numberOfCourses () {
       return _courses.size ();
  }
```

Hace unos años me preocupaba que trasladar este tipo de comportamiento a una persona llevaría a una clase de persona hinchada En la práctica, he descubierto que generalmente no es un problema.

Ejemplo: Java 1.1

En muchos sentidos, el caso de Java 1.1 es bastante similar al de Java 2. Uso el mismo ejemplo pero con un vector:

```
Persona de clase ...
  public Vector getCourses () {
       volver _ cursos;
  }
  setCourses void público (Vector arg) {
       _courses = arg;
  }
  cursos privados de vectores;
```

Nuevamente empiezo creando modificadores e inicializando el campo de la siguiente manera:

```
Persona de clase
  public void addCourse (Curso arg) {
       _courses.addElement (arg);
  }
```

```
public void removeCourse (Curso arg) {
     _courses.removeElement (arg);
}
vector privado _courses = new Vector ();
```


Puedo modificar los setCourses para inicializar el vector:

```
public void initializeCourses (Vector arg) {
     Assert.isTrue (_courses.isEmpty ());
     Enumeración e = arg.elements ();
     while (e.hasMoreElements ()) {
          addCourse ((Course) e.nextElement ());
     }
}
```

Cambio los usuarios del getter para usar los modificadores, así que

```
          kent.getCourses (). addElement (nuevo curso ("Brutal Sarcasm",
falso));
```

se convierte

```
          kent.addCourse (nuevo curso ("Sarcasmo brutal", falso));
```

Mi último paso cambia porque los vectores no tienen una versión no modificable:

```
  Persona de clase ...
    Vector getCourses () {
         return (Vector) _courses.clone ();
    }
```

En este punto, he encapsulado la colección. Nadie puede cambiar los elementos de la colección.
excepto a través de la persona.

Ejemplo: matrices encapsulantes

Las matrices se usan comúnmente, especialmente por programadores que no están familiarizados con las colecciones. yo
rara vez uso matrices, porque prefiero las colecciones más ricas en comportamiento. A menudo cambio matrices
en colecciones como hago la encapsulación.

Esta vez empiezo con una serie de cuerdas para habilidades:

```
Cadena [] getSkills () {
     return _skills;
}
void setSkills (String [] arg) {
     _skills = arg;
}
String [] _skills;
```

Nuevamente empiezo proporcionando una operación de modificación. Porque es probable que el cliente cambie un valor en un

posición particular, necesito una operación de configuración para un elemento particular:


```
void setSkill (int index, String newSkill) {
     _skills [index] = newSkill;
}
```

Si necesito configurar toda la matriz, puedo hacerlo utilizando la siguiente operación:

```
void setSkills (String [] arg) {
     _skills = new String [longitud de arg.];
     para (int i = 0; i <longitud de arg; i ++)
          setSkill (i, arg [i]);
}
```

Existen numerosas dificultades aquí si hay que hacer algo con los elementos eliminados. los
la situación se complica por lo que sucede cuando la matriz de argumentos es diferente en longitud de la
Matriz original. Esa es otra razón para preferir una colección.

En este punto, puedo comenzar a mirar a los usuarios del getter. puedo cambiar

```
kent.getSkills () [1] = "Refactorización";
```

a

```
kent.setSkill (1, "Refactorización");
```

Cuando haya realizado todos los cambios, puedo modificar el captador para devolver una copia:

```
Cadena [] getSkills () {
     Cadena [] resultado = nueva Cadena [_skills.length];
     System.arraycopy (_skills, 0, result, 0, _skills.length);
     resultado de retorno;
}
```

Este es un buen punto para reemplazar la matriz con una lista:

```
Persona de clase ...
  Cadena [] getSkills () {
       return (String []) _skills.toArray (nueva cadena [0]);
  }
  void setSkill (int index, String newSkill) {
       _skills.set (index, newSkill);
  }
  Lista _skills = new ArrayList ();
```

Reemplazar registro con clase de datos

Necesita interactuar con una estructura de registro en un entorno de programación tradicional.

Haga un objeto de datos tonto para el registro.

Motivación

Las estructuras de registro son una característica común de los entornos de programación. Hay varios razones para incorporarlos a un programa orientado a objetos. Podrías estar copiando un legado programa, o podría estar comunicando un registro estructurado con una API de programación tradicional, o un registro de la base de datos. En estos casos, es útil crear una clase de interfaz para lidiar con esto elemento externo Es más simple hacer que la clase se parezca al registro externo. Te mueves otro campos y métodos en la clase más tarde. Un caso menos obvio pero muy convincente es una matriz en que el elemento en cada índice tiene un significado especial. En este caso, usa Reemplazar matriz con el objeto .

Mecánica

- Crear una clase para representar el registro.
- Dele a la clase un campo privado con un método de obtención y un método de configuración para cada dato articulo.

Ahora tiene un objeto de datos tonto. Todavía no tiene comportamiento, pero una mayor refactorización explorará eso problema.

Reemplazar código de tipo con clase

Una clase tiene un código de tipo numérico que no afecta su comportamiento.

Reemplace el número con una nueva clase.

Motivación

Los códigos de tipo numérico, o enumeraciones, son una característica común de los lenguajes basados en C. Con nombres simbólicos pueden ser bastante legibles. El problema es que el nombre simbólico es solo un alias; el compilador aún ve el número subyacente. El tipo de compilador verifica usando el número

No es el nombre simbólico. Cualquier método que tome el código de tipo como argumento espera un número, y no hay nada que obligue a usar un nombre simbólico. Esto puede reducir la legibilidad y ser un fuente de errores.

Si reemplaza el número con una clase, el compilador puede escribir check en la clase. Proporcionando métodos de fábrica para la clase, puede verificar estáticamente que solo se crean instancias válidas y que esas instancias se pasan a los objetos correctos.

Sin embargo, antes de *reemplazar el código de tipo con clase,* debe considerar el otro código de tipo reemplazos Reemplace el código de tipo con una clase solo si el código de tipo son datos puros, es decir, no causa un comportamiento diferente dentro de una declaración de cambio. Para empezar, Java solo puede activarse un entero, no una clase arbitraria, por lo que el reemplazo fallará. Más importante que eso, cualquier el interruptor debe eliminarse con Reemplazar condicional por polimorfismo . Para eso refactorización, el código de tipo primero debe manejarse con Reemplazar código de tipo con subclases o Reemplazar código de tipo con estado / estrategia .

Incluso si un código de tipo no causa un comportamiento diferente dependiendo de su valor, puede haber comportamiento que se coloca mejor en la clase de código de tipo, así que esté alerta al valor de un Método Move o dos.

Mecánica

• Crear una nueva clase para el código de tipo.

> ? rarr; *La clase necesita un campo de código que coincida con el código de tipo y un método de obtención de este valor. Debe tener variables estáticas para el instancias permitidas de la clase y un método estático que devuelve el instancia apropiada de un argumento basado en el código original.*

• Modificar la implementación de la clase fuente para usar la nueva clase.

> ? rarr; *Mantener la antigua interfaz basada en código, pero cambiar los campos estáticos. usar una nueva clase para generar los códigos. Alterar el otro código métodos para obtener los números de código de la nueva clase.*

• Compilar y probar.

> ? rarr; *En este punto, la nueva clase puede verificar los códigos en tiempo de ejecución.*

• Para cada método en la clase fuente que usa el código, cree un nuevo método que use la nueva clase en su lugar.

> ? rarr; *Los métodos que usan el código como argumento necesitan nuevos métodos que use una instancia de la nueva clase como argumento. Métodos que devuelven un El código necesita un nuevo método que devuelva el código. A menudo es aconsejable usar* Cambiar *el* nombre del método *en un antiguo descriptor de acceso antes de crear uno nuevo para aclare el programa cuando use un código antiguo.*

• Uno por uno, cambie los clientes de la clase fuente para que usen la nueva interfaz.
• Compilar y probar después de actualizar cada cliente.

? rarr; *Es posible que deba modificar varios métodos antes de tener suficiente consistencia para compilar y probar.*

- Elimine la interfaz anterior que usa los códigos y elimine las declaraciones estáticas del códigos
- Compilar y probar.

Ejemplo

Una persona tiene un grupo sanguíneo modelado con un código de tipo:

```
Persona de clase {

   público estático final int O = 0;
   público estático final int A = 1;
   público estático final int B = 2;
   público estático final int AB = 3;

   int privado _bloodGroup;

   Persona pública (int bloodGroup) {
        _bloodGroup = bloodGroup;
   }

   public void setBloodGroup (int arg) {
        _bloodGroup = arg;
   }

   public int getBloodGroup () {
        return _bloodGroup;
   }
}
```

Comienzo creando una nueva clase de grupo sanguíneo con instancias que contienen el número de código de tipo:

```
clase BloodGroup {
   público estático final BloodGroup O = new BloodGroup (0);
   público estático final BloodGroup A = new BloodGroup (1);
   público estático final BloodGroup B = new BloodGroup (2);
   público estático final BloodGroup AB = nuevo BloodGroup (3);
   privado estático final BloodGroup [] _values = {O, A, B, AB};

   privado final int _code;

   BloodGroup privado (código int) {
        _code = code;
   }

   public int getCode () {
        código de retorno;
   }

   código público estático de BloodGroup (int arg) {
```

Page 179

```
        return _values [arg];
    }

}
```

Luego reemplazo el código en Persona con código que usa la nueva clase:

```
Persona de clase {

    public static final int O = BloodGroup.O.getCode ();
    public static final int A = BloodGroup.A.getCode ();
    public static final int B = BloodGroup.B.getCode ();
    public static final int AB = BloodGroup.AB.getCode ();

    BloodGroup privado _bloodGroup;

    Persona pública (int bloodGroup) {
        _bloodGroup = BloodGroup.code (bloodGroup);
    }

    public int getBloodGroup () {
        return _bloodGroup.getCode ();
    }

    public void setBloodGroup (int arg) {
        _bloodGroup = BloodGroup.code (arg);
    }
}
```

En este punto, tengo un control de tiempo de ejecución dentro de la clase de grupo sanguíneo. Para realmente ganar del cambio Tengo que alterar a los usuarios de la clase de persona para usar el grupo sanguíneo en lugar de los enteros.

Para comenzar, uso el método Rename en el descriptor de acceso para el grupo sanguíneo de la persona para aclarar el nuevo estado de cosas:

```
Persona de clase ...
    public int getBloodGroupCode () {
        return _bloodGroup.getCode ();
    }
```

Luego agrego un nuevo método de obtención que utiliza la nueva clase:

```
    público BloodGroup getBloodGroup () {
        return _bloodGroup;
    }
```

También creo un nuevo constructor y método de configuración que usa la clase:

```
Persona pública (BloodGroup bloodGroup) {
    _bloodGroup = bloodGroup;
```

Page 180

```
}

public void setBloodGroup (BloodGroup arg) {
   _bloodGroup = arg;
}
```

Ahora voy a trabajar en los clientes de Person. El arte es trabajar en un cliente a la vez para que usted Puede dar pequeños pasos. Cada cliente puede necesitar varios cambios, y eso lo hace más complicado. Alguna La referencia a las variables estáticas necesita ser cambiada. Entonces

```
Persona thePerson = Persona nueva (Person.A)
```

se convierte

```
   Persona thePerson = Persona nueva ( BloodGroup. A);
```

Las referencias al método de obtención deben usar el nuevo, así que

```
thePerson.getBloodGroupCode ()
```

se convierte

```
thePerson.getBloodGroup (). getCode ()
```

Lo mismo es cierto para los métodos de configuración, entonces

```
thePerson.setBloodGroup (Person.AB)
```

se convierte

```
thePerson.setBloodGroup ( BloodGroup. AB)
```

Una vez hecho esto para todos los clientes de Person, puedo eliminar el método get, constructor, static definiciones y métodos de configuración que usan el entero:

```
Persona de clase ...
   public static final int O = BloodGroup.O.getCode ();
   public static final int A = BloodGroup.A.getCode ();
   public static final int B = BloodGroup.B.getCode ();
   public static final int AB = BloodGroup.AB.getCode ();
   Persona pública (int bloodGroup) {
        _bloodGroup = BloodGroup.code (bloodGroup);
   }
   public int getBloodGroup () {
        return _bloodGroup.getCode ();
   }
```

Página 181

```
public void setBloodGroup (int arg) {
     _bloodGroup = BloodGroup.code (arg);
}
```

También puedo privatizar los métodos sobre el grupo sanguíneo que usan el código:

```
clase BloodGroup ...
  private int getCode () {
       código de retorno;
  }

  código privado de BloodGroup estático (int arg) {
       return _values [arg];
  }
```

Reemplazar código de tipo con subclases

Tiene un código de tipo inmutable que afecta el comportamiento de una clase.

Reemplace el código de tipo con subclases.

Motivación

Si tiene un código de tipo que no afecta el comportamiento, puede usar Reemplazar código de tipo con Clase . Sin embargo, si el código de tipo afecta el comportamiento, lo mejor que puede hacer es usar polimorfismo para manejar el comportamiento variante.

Esta situación generalmente se indica por la presencia de enunciados condicionales similares a mayúsculas y minúsculas. Estos pueden ser interruptores o construcciones if-then-else. En cualquier caso, prueban el valor del código de tipo y luego ejecutar código diferente según el valor del código de tipo. Tales condicionales necesitan ser refactorizado con Reemplazar condicional con polimorfismo . Para que esta refactorización funcione, el tipo el código tiene que ser reemplazado por una estructura de herencia que albergará el comportamiento polimórfico. Dicha estructura de herencia tiene una clase con subclases para cada código de tipo.

La forma más sencilla de establecer esta estructura es *Reemplazar código de tipo con subclases.* Tomas el clase que tiene el código de tipo y crea una subclase para cada código de tipo. Sin embargo, hay casos en el que no puedes hacer esto. En el primero, el valor del código de tipo cambia después de que el objeto es

creado. En el segundo, la clase con el código de tipo ya está subclasificada por otra razón. En en cualquiera de estos casos, debe usar *Reemplazar código de tipo con Estado / Estrategia.*

Reemplazar código de tipo con subclases es principalmente un movimiento de andamios que permite reemplazar Condicional con polimorfismo . El disparador para usar *Reemplazar código de tipo con subclases* es el presencia de declaraciones condicionales. Si no hay declaraciones condicionales, reemplace el código de tipo with Class es el movimiento mejor y menos crítico.

Otra razón para *reemplazar el código de tipo con subclases* es la presencia de características que son relevante solo para objetos con ciertos códigos de tipo. Una vez que haya realizado esta refactorización, puede usar Empuje hacia abajo Método y empujar hacia abajo Campo aclarar que estas características son relevantes sólo en algunos casos.

La ventaja de *Reemplazar código de tipo con subclases* es que mueve el conocimiento de la variante comportamiento de los clientes de la clase a la clase misma. Si agrego nuevas variantes, todo lo que necesito hacer es agregar un subclase Sin polimorfismo tengo que encontrar todos los condicionales y cambiarlos. Así que esto La refactorización es particularmente valiosa cuando las variantes siguen cambiando.

Mecánica

- Autoencapsular el código de tipo.

  ? rarr; *Si el código de tipo se pasa al constructor, debe reemplazar El constructor con un método de fábrica.*

- Para cada valor del código de tipo, cree una subclase. Anular el método de obtención de escriba el código en la subclase para devolver el valor relevante.

  ? rarr; *Este valor está codificado en el retorno (p. Ej.,* `Retorno 1` *). Esta parece desordenado, pero es una medida temporal hasta que todas las declaraciones de casos Ha sido reemplazado.*

- Compile y pruebe después de reemplazar cada valor de código de tipo con una subclase.
- Eliminar el campo de código de tipo de la superclase. Declarar los accesores para el código de tipo como resumen
- Compilar y probar.

Ejemplo

Utilizo el ejemplo aburrido y poco realista del pago de los empleados:

```
Empleado de clase ...
  tipo int privado;
  static final int INGENIERO = 0;
  static final int VENDEDOR = 1;
  static final int MANAGER = 2;

  Empleado (tipo int) {
       _type = type;
  }
```

Página 183

El primer paso es usar el campo Self Encapsulate Field en el código de tipo:

```
int getType () {
     return _type;
}
```

Debido a que el constructor del empleado usa un código de tipo como parámetro, necesito reemplazarlo con un método de fábrica:

```
Empleado crear (tipo int) {
     devolver nuevo empleado (tipo);
}

Empleado privado (tipo int) {
     _type = type;
}
```

Ahora puedo comenzar con el ingeniero como una subclase. Primero creo la subclase y el método de anulación para el código de tipo:

```
ingeniero de clase extiende empleado {
  int getType () {
       volver empleado.ENGINEER;
  }
}
```

También necesito alterar el método de fábrica para crear el objeto apropiado:

```
Empleado de clase
  Empleado estático crear (tipo int) {
       if (type == ENGINEER) return new Engineer ();
       de lo contrario, devuelve nuevo empleado (tipo);
  }
```

Continúo, uno por uno, hasta que todos los códigos se reemplacen con subclases. En este punto puedo deshacerme del campo de código de tipo en el empleado y hacer que `getType sea` un método abstracto. En este punto el El método de fábrica se ve así:

```
resumen int getType ();

Empleado estático crear (tipo int) {
     interruptor (tipo) {
          ingeniero de caso:
              volver nuevo ingeniero ();
          caso VENDEDOR:
              volver nuevo vendedor ();
          administrador de casos:
              volver nuevo Manager ();
          defecto:
```


Page 184

```
                    lanzar nueva IllegalArgumentException ("Código de tipo incorrecto
valor");
            }
        }
```

Por supuesto, este es el tipo de declaración de cambio que preferiría evitar. Pero solo hay uno, y es solo se usa en la creación.

Naturalmente, una vez que haya creado las subclases, debe usar el Método Push Down y Push Campo abajo en cualquier método y campo que sea relevante solo para tipos particulares de empleados.

Reemplazar código de tipo con estado / estrategia

Tiene un código de tipo que afecta el comportamiento de una clase, pero no puede usar subclases.

Reemplace el código de tipo con un objeto de estado.

Motivación

Esto es similar a Reemplazar código de tipo con subclases , pero se puede usar si el código de tipo cambia durante la vida del objeto o si otra razón impide la subclasificación. Utiliza ya sea el patrón de estado o estrategia [Banda de los Cuatro].

El estado y la estrategia son muy similares, por lo que la refactorización es la misma que se use, y Realmente no importa. Elija el patrón que mejor se adapte a las circunstancias específicas. Si usted es tratando de simplificar un algoritmo único con Reemplazar condicional por polimorfismo , la estrategia es El mejor término. Si va a mover datos específicos del estado y piensa en el objeto como estado cambiante, use el patrón de estado.

Mecánica

- Autoencapsular el código de tipo.
- Cree una nueva clase y asígnele el nombre según el propósito del código de tipo. Este es el estado objeto.
- Agregar subclases del objeto de estado, una para cada código de tipo.

  ? rarr; *Es más fácil agregar todas las subclases a la vez, en lugar de una a la vez. hora.*

- Cree una consulta abstracta en el objeto de estado para devolver el código de tipo. Crear anulación consultas de cada subclase de objeto de estado para devolver el código de tipo correcto.
- Compilar.

- Cree un campo en la clase anterior para el nuevo objeto de estado.
- Ajuste la consulta de código de tipo en la clase original para delegar en el objeto de estado.
- Ajuste los métodos de configuración del código de tipo en la clase original para asignar una instancia del subclase de objeto de estado apropiado.
- Compilar y probar.

Ejemplo

De nuevo uso el ejemplo cansado y sin cerebro del pago de los empleados:

```
Empleado de clase {

   tipo int privado;
   static final int INGENIERO = 0;
   static final int VENDEDOR = 1;
   static final int MANAGER = 2;

   Empleado (tipo int) {
        _type = type;
   }
```

Aquí hay un ejemplo del tipo de comportamiento condicional que usaría estos códigos:

```
   int payAmount () {
        interruptor (_tipo) {
             ingeniero de caso:
                 return _monthlySalary;
             caso VENDEDOR:
                 return _monthlySalary + _commission;
             administrador de casos:
                 return _monthlySalary + _bonus;
             defecto:
                 lanzar nueva RuntimeException ("Empleado incorrecto");
        }
   }
```

Supongo que esta es una compañía emocionante y que permite la promoción de gerentes para ingenieros Por lo tanto, el código de tipo es mutable y no puedo usar subclases. Mi primer paso, como siempre, es autoencapsular el código de tipo:

```
   Empleado (tipo int) {
        setType (tipo);
   }

   int getType () {
        return _type;
   }

   void setType (int arg) {
        _type = arg;
   }
```

```
    int payAmount () {
         switch ( getType () ) {
              ingeniero de caso:
                  return _monthlySalary;
              caso VENDEDOR:
                  return _monthlySalary + _commission;
              administrador de casos:
                  return _monthlySalary + _bonus;
              defecto:
                  lanzar nueva RuntimeException ("Empleado incorrecto");
         }
    }
```

Ahora declaro la clase de estado. Declaro esto como una clase abstracta y proporciono un método abstracto para devolver el código de tipo:

```
  clase abstracta EmployeeType {
    resumen int getTypeCode ();
  }
```

Ahora creo las subclases:

```
  El ingeniero de clase amplía EmployeeType {
    int getTypeCode () {
         volver empleado.ENGINEER;
    }
  }
Class Manager extiende EmployeeType {
    int getTypeCode () {
         volver Employee.MANAGER;
    }
  }

vendedor de clase extiende EmployeeType {

    int getTypeCode () {
         volver Empleado. VENDEDOR;
    }
  }
```

He compilado hasta ahora, y todo es tan trivial que incluso para mí se compila fácilmente. Ahora realmente engancho el subclases en el empleado modificando los accesos para el código de tipo:

```
  Empleado de clase ...
    Private EmployeeType _type;

    int getType () {
         return _type.getTypeCode ();
    }

    void setType (int arg) {
```

```
          interruptor (arg) {
               ingeniero de caso:
                   _type = nuevo ingeniero ();
                   rotura;
               caso VENDEDOR:
                   _type = nuevo vendedor ();
                   rotura;
               administrador de casos:
                   _type = new Manager ();
                   rotura;
               defecto:
                   lanzar nueva IllegalArgumentException ("Empleado incorrecto
Código");
          }
     }
```

Esto significa que ahora tengo una declaración de cambio aquí. Una vez que termine de refactorizar, será el único uno en cualquier parte del código, y se ejecutará solo cuando se cambie el tipo. También puedo usar Reemplace el constructor con el método de fábrica para crear métodos de fábrica para diferentes casos. yo puede eliminar todas las otras declaraciones de caso con un impulso rápido de Reemplazar condicional con El polimorfismo .

Me gusta terminar Reemplazar código de tipo con estado / estrategia moviendo todo el conocimiento del tipo códigos y subclases a la nueva clase. Primero copio las definiciones de código de tipo en el tipo de empleado, cree un método de fábrica para tipos de empleados y ajuste el método de configuración en empleado:

```
   Empleado de clase ...
     void setType (int arg) {
          _type = EmployeeType.newType (arg);
     }

clase EmployeeType ...
     estático EmployeeType newType (int code) {
          interruptor (código) {
               ingeniero de caso:
                   volver nuevo ingeniero ();
               caso VENDEDOR:
                   volver nuevo vendedor ();
               administrador de casos:
                   volver nuevo Manager ();
               defecto:
                   lanzar nueva IllegalArgumentException ("Empleado incorrecto
Código");
          }
     }
     static final int INGENIERO = 0;
     static final int VENDEDOR = 1;
     static final int MANAGER = 2;
```

Luego elimino las definiciones de código de tipo del empleado y las reemplazo con referencias a el tipo de empleado:


```
   Empleado de clase ...
```

```
int payAmount () {
     switch (getType ()) {
          caso EmployeeType. INGENIERO:
              return _monthlySalary;
          caso EmployeeType. VENDEDOR:
              return _monthlySalary + _commission;
          caso EmployeeType. GERENTE:
              return _monthlySalary + _bonus;
          defecto:
              lanzar nueva RuntimeException ("Empleado incorrecto");
     }
}
```

Ahora estoy listo para usar Reemplazar condicional con polimorfismo en `payAmount`.

Reemplazar subclase con campos

Tiene subclases que varían solo en los métodos que devuelven datos constantes.

Cambie los métodos a campos de superclase y elimine las subclases.

Motivación

Puede crear subclases para agregar características o permitir que varíe el comportamiento. Una forma de comportamiento variante es El método constante [Beck]. Un método constante es aquel que devuelve un valor codificado. Esto puede Ser muy útil en subclases que devuelven valores diferentes para un descriptor de acceso. Usted define el descriptor de acceso en la superclase e implementarlo con diferentes valores en la subclase.

Aunque los métodos constantes son útiles, una subclase que consiste solo en métodos constantes no es haciendo lo suficiente para que valga la pena existir. Puede eliminar tales subclases completamente poniendo campos en la superclase Al hacerlo, eliminas la complejidad adicional de las subclases.

Mecánica

- Use Reemplazar constructor con método de fábrica en las subclases.
- Si algún código se refiere a las subclases, reemplace la referencia con una a la superclase.

- Declarar campos finales para cada método constante en la superclase.
- Declarar un constructor de superclase protegido para inicializar los campos.
- Agregar o modificar constructores de subclase para llamar al nuevo constructor de superclase.
- Compilar y probar.
- Implemente cada método constante en la superclase para devolver el campo y eliminar el método de las subclases.
- Compilar y probar después de cada extracción.
- Cuando se hayan eliminado todos los métodos de subclase, use el método en línea para constructor en el método de fábrica de la superclase.
- Compilar y probar.
- Eliminar la subclase.
- Compilar y probar.
- Repita alineando el constructor y eliminando cada subclase hasta que se acaben.

Ejemplo

Comienzo con una persona y subclases orientadas al sexo:

```
Clase abstracta Persona {

   booleano abstracto isMale ();
   resumen char getCode ();
...

clase Hombre extiende Persona {
   boolean isMale () {
        volver verdadero;
   }
   char getCode () {
        devuelve 'M';
   }
}

clase Hembra extiende Persona {
   boolean isMale () {
        falso retorno;
   }
   char getCode () {
        devolver 'F';
   }
}
```

Aquí la única diferencia entre las subclases es que tienen implementaciones de resumen métodos que devuelven un método constante codificado [Beck]. Elimino estas subclases perezosas.

Primero necesito usar Reemplazar constructor con el método de fábrica . En este caso quiero una fábrica Método para cada subclase:

```
Persona de clase ...
   Persona estática createMale () {
```



```
        return new Male ();
   }
   Persona estática createFemale () {
```

```
        volver nueva hembra ();
    }
```

Luego reemplazo las llamadas del formulario

```
Persona kent = nuevo hombre ();
```

con

```
Persona kent = Person.createMale ();
```

Una vez que haya reemplazado todas estas llamadas, no debería tener ninguna referencia a las subclases. yo puedo verifique esto con una búsqueda de texto y verifique que al menos nada acceda a ellos fuera del paquete haciendo las clases privadas.

Ahora declaro campos para cada método constante en la superclase:

```
Persona de clase ...
  privado final booleano _isMale;
  privado final char _code;
```

Agrego un constructor protegido en la superclase:

```
Persona de clase ...
  Persona protegida (boolean isMale, código de char) {
       _isMale = isMale;
       _code = code;
  }
```

Agrego constructores que llaman a este nuevo constructor:

```
clase Masculina ...
  Masculino() {
       super (verdadero, 'M');
  }
clase femenina ...
  Mujer () {
       super (falso, 'F');
  }
```

Una vez hecho esto, puedo compilar y probar. Los campos se crean e inicializan, pero hasta ahora no lo son. siendo utilizado. Ahora puedo comenzar a poner los campos en juego colocando los accesos en la superclase y eliminando los métodos de subclase:

Page 191

```
Persona de clase ...
  boolean isMale () {
       return _isMale;
  }
```

```
clase Masculina ...
  boolean isMale () {
       volver verdadero;
  }
```

Puedo hacer este campo y una subclase a la vez o todo de una vez si me siento afortunado.

Después de que todas las subclases estén vacías, elimino el marcador abstracto de la clase de persona y use el método en línea para alinear el constructor de la subclase en la superclase:

```
Persona de clase
  Persona estática createMale () {
       devolver nueva Persona (verdadero, 'M');
  }
```

Después de compilar y probar, elimino la clase masculina y repito el proceso para la clase femenina.

Capítulo 9. Simplificación de expresiones condicionales

La lógica condicional tiene una forma de ponerse difícil, así que aquí hay una serie de refactorizaciones que puede usar para simplificarlo La refactorización central aquí es Descomponer condicional , lo que implica romper un

condicional en pedazos. Es importante porque separa la lógica de conmutación de los detalles de lo que pasa.

Las otras refactorizaciones en este capítulo involucran otros casos importantes. Use Consolidar Expresión condicional cuando tiene varias pruebas y todas tienen el mismo efecto. Utilizar Consolidar fragmentos condicionales duplicados para eliminar cualquier duplicación dentro del código condicional

Si está trabajando con código desarrollado en una mentalidad de un punto de salida, a menudo encuentra indicadores de control que permiten que las condiciones funcionen con esta regla. No sigo la regla sobre un punto de salida de un método. Por lo tanto, uso Reemplazar cláusulas condicionales anidadas con guardia para aclarar casos especiales condicionales y Eliminar bandera de control para deshacerse de las torpes banderas de control.

Los programas orientados a objetos a menudo tienen un comportamiento menos condicional que los programas de procedimientos. porque gran parte del comportamiento condicional es manejado por el polimorfismo. El polimorfismo es mejor porque la persona que llama no necesita saber sobre el comportamiento condicional y, por lo tanto, es más fácil Ampliar las condiciones. Como resultado, los programas orientados a objetos rara vez tienen sentencias de cambio (caso). Cualquiera que aparezca son los principales candidatos para Reemplazar condicional con polimorfismo .

Uno de los usos más útiles, pero menos obvios, del polimorfismo es usar Introducir objeto nulo para eliminar cheques para un valor nulo.

Descomponer Condicional

Tiene una declaración condicional complicada (if-then-else).

Extraiga métodos de la condición, luego parte y otras partes.

```
if (date.before (SUMMER_START) || date.after (SUMMER_END))
     carga = cantidad * _winterRate + _winterServiceCharge;
sino carga = cantidad * _summerRate;
```

```
if (notSummer (date))
     carga = carga de invierno (cantidad);
else charge = summerCharge (cantidad);
```

Motivación

Una de las áreas de complejidad más comunes en un programa radica en la lógica condicional compleja. Como usted escribe código para probar condiciones y hacer varias cosas dependiendo de varias condiciones, usted rápidamente terminan con un método bastante largo. La longitud de un método es en sí mismo un factor que lo hace más difícil de leer, pero las condiciones aumentan la dificultad. El problema generalmente radica en el hecho de que el El código, tanto en las comprobaciones de condición como en las acciones, le dice qué sucede pero puede fácilmente oscuro por qué sucede

Página 193

Al igual que con cualquier bloque de código grande, puede aclarar su intención descomponiéndolo y reemplazar trozos de código con una llamada a un método con el nombre de la intención de ese bloque de código. Con condiciones que puede recibir un beneficio adicional al hacer esto para la parte condicional y cada una de las alternativas. De esta manera, resalta la condición y deja en claro claramente en qué se está ramificando. También destaca el motivo de la ramificación.

Mecánica

- Extraer la condición en su propio método.
- Extraiga la parte then y la parte else en sus propios métodos.

Si encuentro un condicional anidado, generalmente primero miro para ver si debo usar Reemplazar anidado Condicional con cláusulas de guardia . Si eso no tiene sentido, descompongo cada uno de los condicionales

Ejemplo

Supongamos que estoy calculando el cargo por algo que tiene tarifas separadas para invierno y verano:

```
if (date.before (SUMMER_START) || date.after (SUMMER_END))
     carga = cantidad * _winterRate + _winterServiceCharge;
sino carga = cantidad * _summerRate;
```

Extraigo el condicional y cada pata de la siguiente manera:

```
if (notSummer (date))
     carga = carga de invierno (cantidad);
else charge = summerCharge (cantidad);

privado booleano notSummer (fecha fecha) {
     fecha de regreso. antes (SUMMER_START) || fecha.después (SUMMER_END);
}

Private double summerCharge (int cantidad) {
     cantidad de retorno * _summerRate;
}

privado doble winterCharge (int cantidad) {
     cantidad devuelta * _winterRate + _winterServiceCharge;
}
```

Aquí muestro el resultado de la refactorización completa para mayor claridad. En la práctica, sin embargo, hago cada extracción por separado y compilar y probar después de cada uno.

Muchos programadores no extraen las partes de la condición en situaciones como esta. Las condiciones a menudo son bastante cortos, por lo que apenas parece valer la pena. Aunque la condición es a menudo corta, a menudo hay Hay una gran brecha entre la intención del código y su cuerpo. Incluso en este pequeño caso, leyendo `notSummer (date) me` transmite un mensaje más claro que el código original. Con el original tengo que mirar el código y averiguar qué está haciendo. No es difícil hacer eso aquí, pero aun así el método extraído se lee más como un comentario.

Página 194

Consolidar Expresión Condicional

Tienes una secuencia de pruebas condicionales con el mismo resultado.

Combínalos en una sola expresión condicional y extráela.

```
discapacidad doble Cantidad () {
     if (_ superioridad <2) devuelve 0;
     if (_monthsDisabled> 12) devuelve 0;
     if (_isPartTime) devuelve 0;
     // calcular el monto de la discapacidad
```

```
discapacidad doble Cantidad () {
     if (isNotEligableForDisability ()) devuelve 0;
     // calcular el monto de la discapacidad
```

Motivación

A veces se ve una serie de comprobaciones condicionales en las que cada comprobación es diferente pero La acción resultante es la misma. Cuando vea esto, debe usar ands y ors para consolidarlos en una sola verificación condicional con un solo resultado.

La consolidación del código condicional es importante por dos razones. Primero, aclara el cheque al demostrar que realmente está haciendo un solo cheque que está ordenando los otros cheques juntos. los la secuencia tiene el mismo efecto, pero se comunica llevando a cabo una secuencia de comprobaciones separadas que solo se hacen juntos. La segunda razón para esta refactorización es que a menudo establece Estás preparado para el método de extracción . Extraer una condición es una de las cosas más útiles que puede hacer para aclara tu código Reemplaza una declaración de lo que está haciendo con por qué lo está haciendo.

Las razones a favor de la consolidación de los condicionales también apuntan a razones para no hacerlo. Si tu piensa que los cheques son realmente independientes y no deben considerarse como un solo cheque, no lo hagas La refactorización. Su código ya comunica su intención.

Mecánica

- Compruebe que ninguno de los condicionales tiene efectos secundarios.

  ? rarr; *Si hay efectos secundarios, no podrá realizar esta refactorización.*

- Reemplace la cadena de condicionales con una sola declaración condicional usando lógica operadores
- Compilar y probar.
- Considere usar el Método de extracción en la condición.

Ejemplo: Ors

El estado del código es similar al siguiente:

Página 195

```
discapacidad doble Cantidad () {
     if (_ superioridad <2) devuelve 0;
     if (_monthsDisabled> 12) devuelve 0;
     if (_isPartTime) devuelve 0;
     // calcular el monto de la discapacidad
     ...
```

Aquí vemos una secuencia de comprobaciones condicionales que resultan en lo mismo. Con secuencial código como este, las comprobaciones son el equivalente de una declaración o:

```
    discapacidad doble Cantidad () {
         if ((_seniority <2) || (_monthsDisabled> 12) || (_isPartTime))
devuelve 0;
         // calcular el monto de la discapacidad
         ...
```

Ahora puedo ver la condición y usar el Método de extracción para comunicar cuál es la condición buscando:

```
    discapacidad doble Cantidad () {
         if (isNotEligibleForDisability ()) devuelve 0;
         // calcular el monto de la discapacidad
         ...
    }

    boolean isNotEligibleForDisability () {
         return ((_seniority <2) || (_monthsDisabled> 12) ||
(_isPartTime));
    }
```

Ejemplo: Ands

Ese ejemplo mostró ors, pero puedo hacer lo mismo con ands. Aquí la configuración es algo así como siguiendo:

```
    if (onVacation ())
         if (lengthOfService ()> 10)
              retorno 1;
    retorno 0.5;
```

Esto se cambiaría a

```
    if (onVacation () && lengthOfService ()> 10) devuelve 1;
    de lo contrario, devuelve 0.5;
```

Es posible que encuentre una combinación de estos que produce una expresión con ands, ors y Nots. En estos casos, las condiciones pueden ser desordenadas, por lo que trato de usar el Método de extracción en partes del expresión para hacerlo más simple.

Página 196

Si la rutina que estoy viendo prueba solo la condición y devuelve un valor, puedo convertir la rutina en una sola declaración de devolución utilizando el operador ternario. Entonces

```
    if (onVacation () && lengthOfService ()> 10) devuelve 1;
    de lo contrario, devuelve 0.5;
```

se convierte

```
return (onVacation () && lengthOfService ()> 10)? 1: 0,5;
```

Consolidar fragmentos condicionales duplicados

El mismo fragmento de código está en todas las ramas de una expresión condicional.

Muévelo fuera de la expresión.

```
if (isSpecialDeal ()) {
     total = precio * 0.95;
     enviar();
}
más {
     total = precio * 0.98;
     enviar();
}
```

```
if (isSpecialDeal ())
     total = precio * 0.95;
más
     total = precio * 0.98;
enviar();
```

Motivación

A veces encuentras el mismo código ejecutado en todas las patas de un condicional. En ese caso deberías mueve el código fuera del condicional. Esto aclara lo que varía y lo que permanece mismo.

Mecánica

- Identifique el código que se ejecuta de la misma manera, independientemente de la condición.
- Si el código común está al principio, muévalo antes del condicional.
- Si el código común está al final, muévalo después del condicional.
- Si el código común está en el medio, observe si el código está antes o después.

  cambia cualquier cosa Si lo hace, puede mover el código común hacia adelante o hacia atrás termina Luego puede moverlo como se describe para el código al final o al principio.
- Si hay más de una sola declaración, debe extraer ese código en un método.

Ejemplo

Encuentra esta situación con código como el siguiente:

```
if (isSpecialDeal ()) {
     total = precio * 0.95;
     enviar();
}
más {
     total = precio * 0.98;
```

```
        enviar();
    }
```

Como el método de envío se ejecuta en cualquier caso, debería sacarlo del condicional:

```
    if (isSpecialDeal ())
        total = precio * 0.95;
    más
        total = precio * 0.98;
    enviar();
```

La misma situación puede aplicarse a las excepciones. Si el código se repite después de una causa de excepción en el bloque try y todos los bloques catch, puedo moverlo al bloque final.

Eliminar bandera de control

Tiene una variable que actúa como un indicador de control para una serie de expresiones booleanas.

Use un descanso o regrese en su lugar.

Motivación

Cuando tiene una serie de expresiones condicionales, a menudo ve un indicador de control utilizado para determinar cuándo dejar de mirar:

```
conjunto hecho a falso
mientras no haya terminado
  si (condición)
       hacer algo
       conjunto hecho a verdadero
  siguiente paso del bucle
```

Tales banderas de control son más problemáticas de lo que valen. Provienen de reglas de estructuración programación que requiere rutinas con una entrada y un punto de salida. Estoy de acuerdo con (y moderno idiomas obligatorios) un punto de entrada, pero la regla de un punto de salida lo lleva a un entorno muy complicado condicionales con estas incómodas banderas en el código. Es por eso que los idiomas tienen descanso y continuar las declaraciones para salir de un condicional complejo. A menudo es sorprendente lo que puedes hacer cuando te deshagas de una bandera de control. El verdadero propósito del condicional se vuelve mucho más claro.

Página 198

Mecánica

La forma obvia de lidiar con los indicadores de control es usar las declaraciones de interrupción o continuación presentes en Java.

- Encuentre el valor de la bandera de control que lo saca de la declaración lógica.
- Reemplace las asignaciones del valor de ruptura con una declaración de ruptura o continuación.
- Compilar y probar después de cada reemplazo.

Otro enfoque, también utilizable en idiomas sin interrupción y continuación, es el siguiente:

- Extraer la lógica en un método.

- Encuentre el valor de la bandera de control que lo saca de la declaración lógica.
- Reemplace las asignaciones del valor de ruptura con un retorno.
- Compilar y probar después de cada reemplazo.

Incluso en idiomas con interrupción o continuación, prefiero usar una extracción y un retorno. El retorno indica claramente que no se ejecuta más código en el método. Si tienes ese tipo de código, a menudo necesita extraer esa pieza de todos modos.

Esté atento a si el indicador de control también indica información de resultados. Si es así, aún necesitas el indicador de control si usa el corte, o puede devolver el valor si ha extraído un método.

Ejemplo: Indicador de control simple reemplazado por descanso

La siguiente función verifica si una lista de personas contiene un par de códigos codificados personajes sospechosos:

```
vacío checkSecurity (String [] people) {
     booleano encontrado = falso;
     for (int i = 0; i <people.length; i ++) {
          si se encuentra) {
              if (people [i] .equals ("Don")) {
                sendAlert ();
                encontrado = verdadero;
              }
              if (people [i] .equals ("John")) {
                sendAlert ();
                encontrado = verdadero;
              }
          }
     }
}
```

En un caso como este, es fácil ver la bandera de control. Es la pieza que establece la variable `encontrada` en cierto. Puedo presentar los descansos de uno en uno:

```
vacío checkSecurity (String [] people) {
     booleano encontrado = falso;
     for (int i = 0; i <people.length; i ++) {
          si se encuentra) {
```



```
              if (people [i] .equals ("Don")) {
                sendAlert ();
                rotura;
              }
              if (people [i] .equals ("John")) {
                sendAlert ();
                encontrado = verdadero;
              }
          }
     }
}
```

hasta que los tenga todos:

```
vacío checkSecurity (String [] people) {
     booleano encontrado = falso;
     for (int i = 0; i <people.length; i ++) {
          si se encuentra) {
              if (people [i] .equals ("Don")) {
                sendAlert ();
                rotura;
              }
              if (people [i] .equals ("John")) {
                sendAlert ();
                rotura;
              }
          }
     }
}
```

Entonces puedo eliminar todas las referencias a la bandera de control:

```
vacío checkSecurity (String [] people) {
     for (int i = 0; i <people.length; i ++) {
          if (people [i] .equals ("Don")) {
              sendAlert ();
              rotura;
          }
          if (people [i] .equals ("John")) {
              sendAlert ();
              rotura;
          }
     }
}
```

Ejemplo: uso de retorno con un resultado de indicador de control

El otro estilo de esta refactorización utiliza un retorno. Ilustra esto con una variante que usa el control marcar como valor de resultado:

```
vacío checkSecurity (String [] people) {
     Cadena encontrada = "";
```


Página 200

```
     for (int i = 0; i <people.length; i ++) {
          if (found.equals ("")) {
              if (people [i] .equals ("Don")) {
                sendAlert ();
                encontrado = "Don";
              }
              if (people [i] .equals ("John")) {
                sendAlert ();
                encontrado = "John";
              }
          }
     }
     someLaterCode (encontrado);
}
```

Aquí se `encuentra` haciendo dos cosas. Indica un resultado y actúa como un indicador de control. Cuando yo veo esto, me gusta extraer el código que determina que se `encuentra` en su propio método:

```
vacío checkSecurity (String [] people) {
     Cadena encontrada = foundMiscreant (personas);
     someLaterCode (encontrado);
}

String foundMiscreant (String [] people) {
     Cadena encontrada = "";
     for (int i = 0; i <people.length; i ++) {
          if (found.equals ("")) {
              if (people [i] .equals ("Don")) {
                sendAlert ();
                encontrado = "Don";
              }
              if (people [i] .equals ("John")) {
                sendAlert ();
                encontrado = "John";
              }
          }
     }
     retorno encontrado;
}
```

Entonces puedo reemplazar la bandera de control con un retorno:

```
String foundMiscreant (String [] people) {
     Cadena encontrada = "";
     for (int i = 0; i <people.length; i ++) {
          if (found.equals ("")) {
              if (people [i] .equals ("Don")) {
                sendAlert ();
                devolver "Don";
              }
              if (people [i] .equals ("John")) {
                sendAlert ();
                encontrado = "John";
              }
```


201

```
          }
     }
     retorno encontrado;
}
```

hasta que haya eliminado la bandera de control:

```
String foundMiscreant (String [] people) {
     for (int i = 0; i <people.length; i ++) {
          if (people [i] .equals ("Don")) {
              sendAlert ();
              devolver "Don";
          }
          if (people [i] .equals ("John")) {
              sendAlert ();
```

```
                regresar "John";
            }
        }
        regreso "";
    }
```

También puede usar el estilo de devolución cuando no devuelve un valor. Solo usa return sin el argumento.

Por supuesto, esto tiene el problema de una función con efectos secundarios. Entonces quiero usar Consulta separada del modificador . Encontrarás este ejemplo continuado allí.

Reemplazar condicional anidado con cláusulas de guardia

Un método tiene un comportamiento condicional que no deja en claro la ruta normal de ejecución.

Utilice cláusulas de guardia para todos los casos especiales.

```
doble getPayAmount () {
   doble resultado;
   if (_isDead) result = deadAmount ();
   más {
        if (_isSeparated) result = separateAmount ();
        más {
             if (_isRetired) result = retiredAmount ();
             más resultado = normalPayAmount ();
        };
   }
resultado de retorno;
};
```

```
doble getPayAmount () {
   if (_isDead) return deadAmount ();
   if (_isSeparated) devuelve separarAmount ();
   if (_isRetired) return retiredAmount ();
```



```
   return normalPayAmount ();
};
```

Motivación

A menudo encuentro que las expresiones condicionales vienen en dos formas. La primera forma es un cheque si cualquiera de los cursos es parte del comportamiento normal. La segunda forma es una situación en la que una respuesta desde el condicional indica un comportamiento normal y el otro indica una condición inusual.

Este tipo de condicionales tienen diferentes intenciones, y estas intenciones deberían aparecer en el código. Si ambos son parte del comportamiento normal, use una condición con una pierna if y otra. Si el La condición es una condición inusual, verifique la condición y regrese si la condición es verdadera. Este tipo de verificación a menudo se llama una *cláusula de guardia* [Beck].

El punto clave sobre *Reemplazar condicional anidado con cláusulas de guardia* es uno de énfasis. Si tu está utilizando una construcción if-then-else que le está dando el mismo peso a la pierna if y a la pierna else. Esta

comunica al lector que las piernas son igualmente probables e importantes. En cambio el guardia
La cláusula dice: "Esto es raro, y si sucede, haga algo y salga".

A menudo encuentro que uso *Reemplazar condicional anidado con cláusulas de protección* cuando estoy trabajando con un programador al que se le ha enseñado a tener solo un punto de entrada y un punto de salida de un método.
Los idiomas modernos imponen un punto de entrada, y un punto de salida realmente no es una regla útil.
La claridad es el principio clave: si el método es más claro con un punto de salida, use un punto de salida;
de lo contrario no lo hagas.

Mecánica

- Por cada cheque puesto en la cláusula de guardia.

  ? rarr; *La cláusula de guardia vuelve o arroja una excepción.*

- Compilar y probar después de cada verificación se reemplaza con una cláusula de protección.

  ? rarr; *Si todas las cláusulas de protección producen el mismo resultado, use* Consolidar Expresiones condicionales.

Ejemplo

Imagine una ejecución de un sistema de nómina en el que tiene reglas especiales para muertos, separados y jubilados. empleados. Tales casos son inusuales, pero ocurren de vez en cuando.

Si veo el código como este

```
doble getPayAmount () {
   doble resultado;
   if (_isDead) result = deadAmount ();
   más {
        if (_isSeparated) result = separateAmount ();
        más {
             if (_isRetired) result = retiredAmount ();
             más resultado = normalPayAmount ();
```



```
        };
   }
resultado de retorno;
};
```

Entonces la verificación está enmascarando el curso normal de acción detrás de la verificación. Entonces en cambio es más claro para usar cláusulas de guardia. Puedo presentar estos uno a la vez. Me gusta comenzar desde arriba:

```
doble getPayAmount () {
   doble resultado;
   if (_isDead) return deadAmount ();
   if (_isSeparated) result = separateAmount ();
   más {
        if (_isRetired) result = retiredAmount ();
        más resultado = normalPayAmount ();
   };
   resultado de retorno;
};
```

Continúo uno a la vez:

```
doble getPayAmount () {
   doble resultado;
   if (_isDead) return deadAmount ();
   if (_isSeparated) devuelve separarAmount ();
   if (_isRetired) result = retiredAmount ();
   más resultado = normalPayAmount ();
   resultado de retorno;
};
```

y entonces

```
doble getPayAmount () {
   doble resultado;
   if (_isDead) return deadAmount ();
   if (_isSeparated) devuelve separarAmount ();
   if (_isRetired) return retiredAmount ();
   resultado = normalPayAmount ();
   resultado de retorno;
};
```

En este punto, la temperatura resultante no está tirando de su peso, así que lo destruyo:

```
doble getPayAmount () {
   if (_isDead) return deadAmount ();
   if (_isSeparated) devuelve separarAmount ();
   if (_isRetired) return retiredAmount ();
   return normalPayAmount ();
};
```


204 de 1189.

El código condicional anidado a menudo lo escriben programadores a los que se les enseña a tener un punto de salida de un método He descubierto que es una regla demasiado simplista. Cuando no tengo más interés en un método, Señalé mi falta de interés al salir. Dirigir al lector a mirar solo un bloque vacío se interpone en el camino de la comprensión.

Ejemplo: inversión de las condiciones

Al revisar el manuscrito de este libro, Joshua Kerievsky señaló que a menudo *reemplaza Cláusulas condicionales anidadas con guardia* invirtiendo las expresiones condicionales. Vino amablemente con un ejemplo para ahorrar más impuestos a mi imaginación:

```
 public double getAdjustedCapital () {
   doble resultado = 0.0;
   si (_capital> 0.0) {
      if (_intRate> 0.0 && _duration> 0.0) {
         resultado = (_income / _duration) * ADJ_FACTOR;
      }
   }
```

```
   resultado de retorno;
}
```

Nuevamente hago los reemplazos uno a la vez, pero esta vez revierto el condicional mientras pongo el cláusula de guardia:

```
 public double getAdjustedCapital () {
      doble resultado = 0.0;
      if (_capital <= 0.0) devuelve el resultado;
      if (_intRate> 0.0 && _duration> 0.0) {
         resultado = (_income / _duration) * ADJ_FACTOR;
      }
      resultado de retorno;
 }
```

Debido a que el siguiente condicional es un poco más complicado, puedo revertirlo en dos pasos. Primero agrego un no:

```
public double getAdjustedCapital () {
     doble resultado = 0.0;
     if (_capital <= 0.0) devuelve el resultado;
     if (! (_ intRate> 0.0 && _duration> 0.0)) devuelve el resultado;
     resultado = (_income / _duration) * ADJ_FACTOR;
     resultado de retorno;
}
```

Dejar notas en un condicional como ese retuerce mi mente en un ángulo doloroso, así que lo simplifico como sigue:

```
public double getAdjustedCapital () {
     doble resultado = 0.0;
     if (_capital <= 0.0) devuelve el resultado;
```

204 204

205 de 1189.

```
     if (_intRate <= 0.0 || _duration <= 0.0) devuelve el resultado;
     resultado = (_income / _duration) * ADJ_FACTOR;
     resultado de retorno;
}
```

En estas situaciones, prefiero poner un valor explícito en los retornos de los guardias. De esa manera tu puede ver fácilmente el resultado de la falla del guardia (también consideraría Reemplazar número mágico con constante simbólica aquí).

```
public double getAdjustedCapital () {
     doble resultado = 0.0;
     if (_capital <= 0.0) devuelve 0.0;
     if (_intRate <= 0.0 || _duration <= 0.0) devuelve 0.0;
     resultado = (_income / _duration) * ADJ_FACTOR;
     resultado de retorno;
}
```

Con eso hecho, también puedo eliminar la temperatura:

```
    public double getAdjustedCapital () {
         if (_capital <= 0.0) devuelve 0.0;
         if (_intRate <= 0.0 || _duration <= 0.0) devuelve 0.0;
         return (_income / _duration) * ADJ_FACTOR;
    }
```

Reemplazar condicional con polimorfismo

Tiene un condicional que elige un comportamiento diferente según el tipo de objeto.

Mueva cada tramo del condicional a un método de anulación en una subclase. Hacer el método original resumen.

```
    doble getSpeed () {
         interruptor (_tipo) {
              caso europeo:
                  return getBaseSpeed ();
              caso AFRICANO:
                  return getBaseSpeed () - getLoadFactor () *
_numberOfCoconuts;
              caso NORWEGIAN_BLUE:
                  volver (_esNailed)? 0: getBaseSpeed (_voltage);
         }
         lanzar una nueva RuntimeException ("Debería ser inalcanzable");
    }
```


Motivación

Una de las palabras que suenan más grandiosas en la jerga de objetos es el *polimorfismo.* La esencia de polymorphsim es que te permite evitar escribir un condicional explícito cuando tienes objetos cuyo comportamiento varía según sus tipos.

Como resultado, encuentra que las declaraciones de cambio que activan los códigos de tipo o las declaraciones if-then-else que activan cadenas de tipo son mucho menos comunes en un programa orientado a objetos.

El polimorfismo te brinda muchas ventajas. La mayor ganancia ocurre cuando este mismo conjunto de Las condiciones aparecen en muchos lugares del programa. Si desea agregar un nuevo tipo, debe Encuentra y actualiza todos los condicionales. Pero con las subclases solo creas una nueva subclase y Proporcionar los métodos apropiados. Los clientes de la clase no necesitan saber sobre las subclases, lo que reduce las dependencias en su sistema y facilita la actualización.

Mecánica

Antes de comenzar con *Reemplazar condicional por polimorfismo* , debe tener el estructura de herencia necesaria. Es posible que ya tenga esta estructura de refactorizaciones anteriores. Si no tiene la estructura, debe crearla.

Para crear la estructura de herencia tiene dos opciones: Reemplazar código de tipo con Subclases y Reemplazar Código de Tipo con Estado / Estrategia . Las subclases son las más simples. opción, por lo que debe usarlos si puede. Si actualiza el código de tipo después de crear el objeto, sin embargo, no puede usar subclases y debe usar el patrón de estado / estrategia. También necesitas usar el patrón de estado / estrategia si ya está subclasificando esta clase por otra razón. Recuerde que si varias declaraciones de caso están activando el mismo código de tipo, solo necesita cree una estructura de herencia para ese código de tipo.

Ahora puedes atacar el condicional. El código al que apunta puede ser una declaración de cambio (caso) o una si declaración.

- Si la declaración condicional es una parte de un método más grande, desarme la condicional declaración y utilizar el método de extracción .
- Si es necesario, use el método Move para colocar el condicional en la parte superior de la herencia estructura.

- Elija una de las subclases. Cree un método de subclase que anule el condicional método de declaración Copie el cuerpo de esa pierna de la declaración condicional en el método de subclase y ajustarlo para que se ajuste.

  ? rarr; *Es posible que deba hacer algunos miembros privados de la superclase protegido para hacer esto.*

- Compilar y probar.
- Eliminar el tramo copiado de la declaración condicional.
- Compilar y probar.
- Repita con cada tramo de la declaración condicional hasta que todos los tramos se conviertan en subclase métodos
- Hacer que el método de la superclase sea abstracto.

Ejemplo

Utilizo el tedioso y simplista ejemplo de pago de empleados. Estoy usando las clases después de usar Reemplace el Código de Tipo con Estado / Estrategia para que los objetos se vean como en la Figura 9.1 (vea el ejemplo en el Capítulo 8 para saber cómo llegamos aquí).

Figura 9.1. _La estructura de herencia

```
Empleado de clase ...
   int payAmount () {
        switch (getType ()) {
             case EmployeeType.ENGINEER:
                 return _monthlySalary;
             case EmployeeType.SALESMAN:
                 return _monthlySalary + _commission;
             case EmployeeType.MANAGER:
                 return _monthlySalary + _bonus;
             defecto:
                 lanzar nueva RuntimeException ("Empleado incorrecto");
        }
   }
```


Page 208

```
   int getType () {
        return _type.getTypeCode ();
   }
   Private EmployeeType _type;

clase abstracta EmployeeType ...
   resumen int getTypeCode ();

ingeniero de clase extiende EmployeeType ...
   int getTypeCode () {
        volver empleado.ENGINEER;
   }

... y otras subclases
```

La declaración del caso ya está bien extraída, por lo que no hay nada que hacer allí. Necesito moverme en el tipo de empleado, porque esa es la clase que se está subclasificando.

```
clase EmployeeType ...
   int payAmount (Empleado emp) {
        switch (getTypeCode ()) {
             ingeniero de caso:
                 return emp.getMonthlySalary ();
             caso VENDEDOR:
```

```
                return emp.getMonthlySalary () + emp.getCommission ();
            administrador de casos:
                return emp.getMonthlySalary () + emp.getBonus ();
            defecto:
                lanzar nueva RuntimeException ("Empleado incorrecto");
        }
    }
```

Debido a que necesito datos del empleado, necesito pasarlo como argumento. Algunos de estos datos podrían moverse al objeto de tipo empleado, pero eso es un problema para otro refactorización.

Cuando esto se compila, cambio el método `payAmount` en Employee para delegar a la nueva clase:

```
  Empleado de clase ...
    int payAmount () {
        return _type.payAmount (esto);
    }
```

Ahora puedo ir a trabajar en la declaración del caso. Es bastante parecido a la forma en que los niños pequeños matan insectos: yo quitar una pierna a la vez. Primero copio el tramo Ingeniero de la declaración del caso en el Ingeniero clase.

```
  Ingeniero de clase ...
    int payAmount (Empleado emp) {
        return emp.getMonthlySalary ();
    }
```


Página 209

Este nuevo método anula toda la declaración del caso para los ingenieros. Porque soy paranoico, yo a veces poner una trampa en la declaración del caso:

```
  clase EmployeeType ...
    int payAmount (Empleado emp) {
        switch (getTypeCode ()) {
            ingeniero de caso:
                lanzar nueva RuntimeException ("Debería estar siendo
anulado ");
            caso VENDEDOR:
                return emp.getMonthlySalary () + emp.getCommission ();
            administrador de casos:
                return emp.getMonthlySalary () + emp.getBonus ();
            defecto:
                lanzar nueva RuntimeException ("Empleado incorrecto");
        }
    }
```

continuar hasta que se quiten todas las piernas:

```
  vendedor de clase ...
    int payAmount (Empleado emp) {
        return emp.getMonthlySalary () + emp.getCommission ();
    }
```

```
Gerente de clase ...
  int payAmount (Empleado emp) {
       return emp.getMonthlySalary () + emp.getBonus ();
  }
```

y luego declarar el resumen del método de superclase:

```
clase EmployeeType ...
  abstract int payAmount (Empleado emp);
```

Introducir objeto nulo

Ha realizado comprobaciones repetidas para un valor nulo.

Reemplace el valor nulo con un objeto nulo.

```
if (cliente == nulo) plan = BillingPlan.basic ();
de lo contrario plan = customer.getPlan ();
```


Página 210

Motivación

La esencia del polimorfismo es que en lugar de preguntarle a un objeto de qué tipo es y luego invocar algunos comportamientos basados en la respuesta, solo invocas el comportamiento. El objeto, dependiendo de su tipo, hace lo correcto. Uno de los lugares menos intuitivos para hacer esto es donde tiene un valor nulo en un campo. Dejaré que Ron Jeffries cuente la historia:

Ron Jeffries

Comenzamos a usar el patrón de objeto nulo cuando Rich Garzaniti encontró que mucho código en el sistema verificaría la presencia de objetos antes enviando un mensaje al objeto. Podríamos preguntarle a un objeto por su persona, luego pregunte el resultado si fue nulo. Si el objeto estaba presente, nosotros le pediría su tarifa. Estábamos haciendo esto en varios lugares, y el el código duplicado resultante se estaba volviendo molesto.

Entonces implementamos un objeto de persona desaparecida que respondió una tasa cero (llamamos a nuestros objetos nulos objetos perdidos). Pronto la persona desaparecida conocía un muchos métodos, como la tasa. Ahora tenemos más de 80 objetos nulos clases

Página 211

Nuestro uso más común de objetos nulos es en la visualización de información. Cuando mostramos, por ejemplo, una persona, el objeto puede o no tener cualquiera de quizás 20 variables de instancia. Si se permitiera que estos fueran nulos, La impresión de una persona sería muy compleja. En cambio, nos conectamos varios objetos nulos, todos los cuales saben cómo mostrarse en un de manera ordenada. Esto eliminó grandes cantidades de código de procedimiento.

Nuestro uso más inteligente del objeto nulo es la sesión de Gemstone que falta. Nosotros usar la base de datos Gemstone para la producción, pero preferimos desarrollar sin él y empuje el nuevo código a Gemstone cada semana más o menos. Ahí Hay varios puntos en el código donde tenemos que iniciar sesión en una Gemstone sesión. Cuando estamos corriendo sin Gemstone, simplemente conectamos un falta la sesión de Gemstone. Se ve igual que la cosa real pero permite nosotros para desarrollar y probar sin darnos cuenta de que la base de datos no está allí.

Otro uso útil del objeto nulo es el contenedor perdido. Un contenedor es una colección. de valores de nómina que a menudo tienen que ser sumados o en bucle. Si un el contenedor particular no existe, respondemos un contenedor perdido, que actúa igual que Un contenedor vacío. El contenedor que falta sabe que tiene saldo cero y ningún valor. Al utilizar este enfoque, eliminamos la creación de decenas de contenedores vacíos. para cada uno de nuestros miles de empleados.

Una característica interesante del uso de objetos nulos es que las cosas casi

nunca explotar Porque el objeto nulo responde de todos modos mensajes como uno real, el sistema generalmente se comporta normalmente. Esta a veces puede hacer que sea difícil detectar o encontrar un problema, porque Nada se rompe nunca. Por supuesto, tan pronto como comience a inspeccionar el objetos, encontrará el objeto nulo en algún lugar donde no debería estar.

Recuerde, los objetos nulos son siempre constantes: nunca hay nada sobre ellos cambios En consecuencia, los implementamos usando el patrón Singleton [Pandilla de cuatro]. Cada vez que pides, por ejemplo, una persona desaparecida, siempre obtienes la única instancia de esa clase.

Puede encontrar más detalles sobre el patrón de objeto nulo en Woolf [Woolf].

Mecánica

- Cree una subclase de la clase fuente para que actúe como una versión nula de la clase. Crear un Operación `isNull` en la clase fuente y la clase nula. Para la clase fuente debería return false, para la clase nula debería devolver true.

  ? rarr; *Puede resultarle útil crear una interfaz explícitamente anulable para la* `isNull` *método.*

  ? rarr; *Como alternativa, puede usar una interfaz de prueba para probar nulidad*

Página 212

- Compilar.
- Encuentre todos los lugares que pueden dar un valor nulo cuando se le solicita un objeto fuente. Reemplazarlos a dar un objeto nulo en su lugar.
- Encuentre todos los lugares que comparen una variable del tipo de fuente con nulo y reemplácelos con una llamada es `nula`.

  ? rarr; *Es posible que pueda hacer esto reemplazando una fuente y sus clientes a la vez y compilando y probando entre trabajar en fuentes.*

  ? rarr; *Algunas afirmaciones que verifican nulo en lugares donde no debe Ya verlo puede ser útil.*

- Compilar y probar.
- Busque casos en los que los clientes invoquen una operación si no es nula y hagan alguna alternativa comportamiento si es nulo.
- Para cada uno de estos casos, anule la operación en la clase nula con la alternativa comportamiento.
- Elimine la verificación de condición para aquellos que usan el comportamiento, compilación y prueba anulados.

Ejemplo

Una empresa de servicios públicos conoce los sitios: las casas y apartamentos que utilizan los servicios de la empresa. A cada vez que un sitio tiene un cliente.

```
Sitio de clase ...
  Cliente getCustomer () {
```

```
        volver _cliente;
    }
    Cliente _cliente;
```

Hay varias características de un cliente. Miro a tres de ellos.

```
  clase Cliente ...
    public String getName () {...}
    public BillingPlan getPlan () {...}
    public PaymentHistory getHistory () {...}
```

El historial de pagos tiene sus propias características:

```
  Clase pública PaymentHistory ...
    int getWeeksDelinquentInLastYear ()
```

Los captadores que muestro permiten a los clientes obtener estos datos. Sin embargo, a veces no tengo un cliente
para un sitio Es posible que alguien se haya mudado y aún no sé quién se mudó. Porque esto
Puede suceder que tengamos que asegurarnos de que cualquier código que use el cliente pueda manejar valores nulos. Aquí hay un
pocos fragmentos de ejemplo:

```
            Cliente cliente = site.getCustomer ();
```


Page 213

```
            Plan de facturación;
            if (cliente == nulo) plan = BillingPlan.basic ();
            de lo contrario plan = customer.getPlan ();
 ...
            Cadena customerName;
            if (customer == null) customerName = "ocupante";
            else customerName = customer.getName ();
 ...
            semanas intDelinquent;
            if (cliente == nulo) semanasDelinquent = 0;
            más semanasDelinquent =
customer.getHistory (). getWeeksDelinquentInLastYear ();
```

En estas situaciones, puedo tener muchos clientes del sitio y clientes, todos los cuales deben verificar
nulos y todos los cuales hacen lo mismo cuando encuentran uno. Parece que es hora de un objeto nulo.

El primer paso es crear la clase de cliente nula y modificar la clase de cliente para admitir un
consulta para una prueba nula:

```
  clase NullCustomer extiende Cliente {
    public boolean isNull () {
        volver verdadero;
    }
  }

  clase Cliente ...
    public boolean isNull () {
        falso retorno;
    }
```

```
    Cliente protegido () {} // que necesita el Cliente Nulo
```

Si no puede modificar la clase Cliente, puede usar una interfaz de prueba.

Si lo desea, puede indicar el uso de un objeto nulo mediante una interfaz:

```
interfaz Nullable {
  boolean isNull ();
}

clase Cliente implementa Nullable
```

Me gusta agregar un método de fábrica para crear clientes nulos. De esa manera, los clientes no tienen que saber sobre la clase nula:

```
clase Cliente ...
  Cliente estático newNull () {
       return new NullCustomer ();
  }
```


Página 214

Ahora viene lo difícil. Ahora tengo que devolver este nuevo objeto nulo cada vez que espero un nulo y reemplace las pruebas de la forma `foo == null` con pruebas de la forma `foo.isNull ()`. Me parece útil para buscar todos los lugares donde solicito un cliente y modificarlos para que devuelvan un valor nulo cliente en lugar de nulo.

```
Sitio de clase ...
  Cliente getCustomer () {
       return (_customer == null)?
            Customer.newNull ():
            _cliente;
  }
```

También tengo que alterar todos los usos de este valor para que prueben con `isNull () en` lugar de `== null`.

```
        Cliente cliente = site.getCustomer ();
        Plan de facturación;
        if (customer. isNull () ) plan = BillingPlan.basic ();
        de lo contrario plan = customer.getPlan ();
...
        Cadena customerName;
        if (customer. isNull () ) customerName = "ocupante";
        else customerName = customer.getName ();
...
        semanas intDelinquent;
        if (customer. isNull () ) weeksDelinquent = 0;
        más semanasDelinquent =
customer.getHistory (). getWeeksDelinquentInLastYear ();
```

No hay duda de que esta es la parte más complicada de esta refactorización. Para cada fuente de un nulo I

reemplazar, tengo que encontrar todas las veces que se prueba la nulidad y reemplazarlas. Si el objeto es ampliamente pasado, estos pueden ser difíciles de rastrear. Tengo que encontrar cada variable de tipo cliente y encontrar en todas partes se usa. Es difícil dividir este proceso en pequeños pasos. A veces encuentro uno fuente que se usa solo en unos pocos lugares, y solo puedo reemplazar esa fuente. Pero la mayor parte del tiempo Sin embargo, tengo que hacer muchos cambios generalizados. Los cambios no son demasiado difíciles de retroceder. de, porque puedo encontrar llamadas de `isNull` sin demasiada dificultad, pero este sigue siendo un paso desordenado.

Una vez hecho este paso, y lo he compilado y probado, puedo sonreír. Ahora comienza la diversión. Como lo no `gano` nada usando `isNull en` lugar de `== null.` La ganancia viene mientras me muevo comportamiento al cliente nulo y eliminar condicionales. Puedo hacer estos movimientos uno a la vez. yo Comience con el nombre. Actualmente tengo un código de cliente que dice

```
Cadena customerName;
if (customer.isNull ()) customerName = "ocupante";
else customerName = customer.getName ();
```

Agrego un método de nombre adecuado para el cliente nulo:

```
clase NullCustomer ...
  public String getName () {
       volver "ocupante";
```



```
  }
```

Ahora puedo hacer que el código condicional desaparezca:

```
Cadena customerName = customer.getName ();
```

Puedo hacer lo mismo con cualquier otro método en el que haya una respuesta general sensible a una consulta. También puedo hacer acciones apropiadas para modificadores. Entonces el código del cliente como

```
if (! customer.isNull ())
     customer.setPlan (BillingPlan.special ());
```

puede ser reemplazado con

```
customer.setPlan (BillingPlan.special ());

clase NullCustomer ...
  public void setPlan (argumento BillingPlan) {}
```

Recuerde que este movimiento de comportamiento solo tiene sentido cuando la mayoría de los clientes quieren lo mismo respuesta. Tenga en cuenta que dije *más,* no *todos.* Cualquier cliente que quiera una respuesta diferente al uno estándar todavía puede probar usando `isNull.` Se beneficia cuando muchos clientes quieren hacer lo mismo cosa; simplemente pueden confiar en el comportamiento nulo predeterminado.

El ejemplo contiene un caso ligeramente diferente: código de cliente que utiliza el resultado de una llamada a cliente:

```
if (customer.isNull ()) weeksDelinquent = 0;
```

```
          más semanasDelinquent =
customer.getHistory (). getWeeksDelinquentInLastYear ();
```

Puedo manejar esto creando un historial de pagos nulo:

```
   class NullPaymentHistory extiende PaymentHistory ...
     int getWeeksDelinquentInLastYear () {
          devuelve 0;
     }
```

Modifico el cliente nulo para devolverlo cuando me preguntan:

```
   clase NullCustomer ...
     public PaymentHistory getHistory () {
          return PaymentHistory.newNull ();
     }
```

Nuevamente puedo eliminar el código condicional:

Page 216

```
          int semanasDelinquent =
customer.getHistory (). getWeeksDelinquentInLastYear ();
```

A menudo encuentra que los objetos nulos devuelven otros objetos nulos.

Ejemplo: interfaz de prueba

La interfaz de prueba es una alternativa a la definición de un método isNull. En este enfoque creo un nulo interfaz sin métodos definidos:

```
   interfaz nula {}
```

Luego implemento nulo en mis objetos nulos:

```
   clase NullCustomer extiende cliente implementa Null ...
```

Luego pruebo la nulidad con el operador instanceof:

```
   Una instancia de cliente de nulo
```

Normalmente huyo gritando del operador de `instancia de` , pero en este caso está bien usar eso. Tiene la ventaja particular de que no necesito cambiar la clase de cliente. Esto me permite usar el objeto nulo incluso cuando no tengo acceso al código fuente del cliente.

Otros casos especiales

Al realizar esta refactorización, puede tener varios tipos de valores nulos. A menudo hay una diferencia entre no hay cliente (nuevo edificio y aún no se ha mudado) y hay un desconocido

cliente (creemos que hay alguien allí, pero no sabemos quién es). Si ese es el caso, usted puede construir clases separadas para los diferentes casos nulos. A veces, los objetos nulos pueden llevar datos, como registros de uso del cliente desconocido, para que podamos facturar a los clientes cuando descubrimos quiénes son.

En esencia, hay un patrón más grande aquí, llamado *caso especial.* Una clase de caso especial es un particular instancia de una clase con comportamiento especial. Entonces, `UnknownCustomer` y `NoCustomer lo` harían Ser casos especiales del `cliente.` A menudo ves casos especiales con números. Puntos flotantes en Java tiene casos especiales para infinito positivo y negativo y no para un número (NaN). El valor de casos especiales es que ayudan a reducir el manejo de errores. Las operaciones de punto flotante no arrojan excepciones Hacer cualquier operación con NaN produce otro NaN de la misma manera que los accesores en Los objetos nulos generalmente resultan en otros objetos nulos.

Introducir aserción

Una sección de código supone algo sobre el estado del programa.

Haz explícita la suposición con una afirmación.

Página 217

```
    doble getExpenseLimit () {
         // debe tener un límite de gastos o un proyecto primario
         return (_expenseLimit! = NULL_EXPENSE)?
              _expenseLimit:
              _primaryProject.getMemberExpenseLimit ();
    }
```

```
    doble getExpenseLimit () {
         Assert.isTrue (_expenseLimit! = NULL_EXPENSE || _primaryProject
! = nulo);
         return (_expenseLimit! = NULL_EXPENSE)?
              _expenseLimit:
              _primaryProject.getMemberExpenseLimit ();
    }
```

Motivación

A menudo, las secciones de código funcionan solo si ciertas condiciones son verdaderas. Esto puede ser tan simple como un cuadrado el cálculo raíz funciona solo en un valor de entrada positivo. Con un objeto se puede suponer que en Al menos uno de un grupo de campos tiene un valor.

Tales suposiciones a menudo no se establecen, pero solo se pueden decodificar mirando a través de un algoritmo. A veces las suposiciones se expresan con un comentario. Una mejor técnica es hacer que supuesto explícito al escribir una afirmación.

Una aserción es una declaración condicional que se supone que es siempre cierta. Fracaso de una afirmación indica error del programador. Como tal, las fallas de aserción siempre deben resultar en desmarcado excepciones Las afirmaciones nunca deben ser utilizadas por otras partes del sistema. De hecho afirmaciones generalmente se eliminan para el código de producción. Por lo tanto, es importante señalar que algo es un afirmación.

Las afirmaciones actúan como ayudas de comunicación y depuración. En comunicación ayudan al lector entender las suposiciones que hace el código. En la depuración, las aserciones pueden ayudar a detectar errores

más cerca de su origen. Me di cuenta de que la ayuda de depuración es menos importante cuando escribo pruebas automáticas código, pero aún aprecio el valor de las afirmaciones en la comunicación.

Mecánica

Debido a que las aserciones no deberían afectar el funcionamiento de un sistema, agregar una siempre es un comportamiento conservación.

• *Cuando vea que se supone que una condición es verdadera, agregue una aserción para indicarla.*

? rarr; *Tenga una clase de aserción que pueda usar para el comportamiento de aserción.*

Tenga cuidado con el uso excesivo de afirmaciones. No use aserciones para verificar todo lo que cree que es verdad Una sección de código. Use aserciones solo para verificar las cosas que *deben* ser ciertas. Uso excesivo de afirmaciones puede conducir a una lógica duplicada que es difícil de mantener. La lógica que cubre una suposición es buena porque te obliga a repensar la sección del código. Si el código funciona sin la aserción, La afirmación es confusa más que útil y puede obstaculizar la modificación en el futuro.

Page 218

Siempre pregunte si el código aún funciona si falla una aserción. Si el código funciona, elimine el afirmación.

Cuidado con el código duplicado en las aserciones. El código duplicado huele tan mal en las comprobaciones de afirmación como Lo hace en cualquier otro lugar. Use el método de extracción generosamente para deshacerse de la duplicación.

Ejemplo

Aquí hay una historia simple de límites de gastos. Los empleados pueden recibir un límite de gasto individual. Si ellos se les asigna un proyecto primario, pueden usar el límite de gastos de ese proyecto primario. Ellos no tienen que tener un límite de gastos o un proyecto primario, pero deben tener uno u otro. Esta la suposición se da por sentado en el código que usa límites de gastos:

```
Empleado de clase ...
  privado estático final doble NULL_EXPENSE = -1.0;
  private double _expenseLimit = NULL_EXPENSE;
  Proyecto privado _Proyecto primario;
 doble getExpenseLimit () {
      return (_expenseLimit! = NULL_EXPENSE)?
           _expenseLimit:
           _primaryProject.getMemberExpenseLimit ();
 }
 boolean dentro de Límite (doble gasto Cantidad) {
      return (chargeAmount <= getExpenseLimit ());
 }
```

Este código contiene una suposición implícita de que el empleado tiene un proyecto o un personal límite de gastos Tal afirmación debe estar claramente establecida en el código:

```
 doble getExpenseLimit () {
      Assert.isTrue (_expenseLimit! = NULL_EXPENSE || _primaryProject
! = nulo);
      return (_expenseLimit! = NULL_EXPENSE)?
           _expenseLimit:
```

```
            _primaryProject.getMemberExpenseLimit ();
    }
```

Esta afirmación no cambia ningún aspecto del comportamiento del programa. De cualquier manera, si el la condición no es verdadera, obtengo una excepción de tiempo de ejecución: una excepción de puntero nulo en `insideLimit` o una excepción de tiempo de ejecución dentro de `Assert.isTrue.` En algunas circunstancias, la afirmación ayuda a encontrar el error, porque está más cerca de donde las cosas salieron mal. Principalmente, sin embargo, la afirmación ayuda a comunicar cómo funciona el código y qué supone.

A menudo encuentro que uso el Método de extracción en el condicional dentro de la aserción. O lo uso en varios lugares y elimine el código duplicado o úselo simplemente para aclarar la intención de la condición.

Una de las complicaciones de las afirmaciones en Java es que no existe un mecanismo simple para poner ellos. Las afirmaciones deben ser fácilmente removibles, para que no afecten el rendimiento en la producción. código. Tener una clase de utilidad, como `Assert,` ciertamente ayuda. Lamentablemente, cualquier expresión dentro del Los parámetros de aserción ejecutan lo que suceda. La única forma de detener eso es usar código como:

Page 219

```
    doble getExpenseLimit () {
         Assert.isTrue (Assert.ON &&
              (_expenseLimit! = NULL_EXPENSE || _primaryProject! = null));
         return (_expenseLimit! = NULL_EXPENSE)?
              _expenseLimit:
              _primaryProject.getMemberExpenseLimit ();
    }
```

o

```
    doble getExpenseLimit () {
         if (Afirmación ON)
              Assert.isTrue (_expenseLimit! = NULL_EXPENSE ||
_primaryProject! = null);
         return (_expenseLimit! = NULL_EXPENSE)?
              _expenseLimit:
              _primaryProject.getMemberExpenseLimit ();
    }
```

Si `Assert.ON` es una constante, el compilador debe detectar y eliminar el código muerto si es falso. Sin embargo, agregar la cláusula es complicado, por lo que muchos programadores prefieren el uso más simple de `Assert` y luego use un filtro para eliminar cualquier línea que use aserción en el momento de la producción (usando perl o similar).

La clase `Assert` debe tener varios métodos que se nombran útilmente. Además de `isTrue,` puede tener `iguales,` y `nunca debe alcanzar aquí.`

219

Page 220

Capítulo 10. Hacer llamadas de método más simples

Los objetos son todo acerca de las interfaces. Proponer interfaces fáciles de entender y usar es
Una habilidad clave en el desarrollo de un buen software orientado a objetos. Este capítulo explora refactorizaciones que
Hacer interfaces más sencillas.

A menudo, lo más simple e importante que puede hacer es cambiar el nombre de un método.
Naming es una herramienta clave en la comunicación. Si comprende lo que hace un programa, debe
No tengas miedo de usar el método Rename para transmitir ese conocimiento. Puedes (y deberías) también
renombrar variables y clases. En general, estos cambios son reemplazos de texto bastante simples,
así que no he agregado refactorizaciones adicionales para ellos.

Los parámetros en sí mismos juegan un papel importante con las interfaces. Agregar parámetro y eliminar
Los parámetros son refactorizaciones comunes. Los programadores nuevos en objetos a menudo usan parámetros largos
listas, que son típicas de otros entornos de desarrollo. Los objetos te permiten mantener el parámetro
las listas son breves, y varias refactorizaciones más involucradas le brindan formas de acortarlas. Si usted es
pasando varios valores de un objeto, use Preserve Whole Object para reducir todos los valores a un
solo objeto Si este objeto no existe, puede crearlo con Introducir objeto de parámetro .
Si puede obtener los datos de un objeto al que el método ya tiene acceso, puede eliminar
parámetros con Reemplazar parámetro con método . Si tiene parámetros que se utilizan para
determinar el comportamiento condicional, puede usar Reemplazar parámetro con métodos explícitos . Tú
puede combinar varios métodos similares agregando un parámetro con el Método de parametrización .

Doug Lea me dio una advertencia sobre las refactorizaciones que reducen las listas de parámetros. Concurrente
La programación a menudo utiliza largas listas de parámetros. Por lo general, esto ocurre para que pueda pasar
parámetros que son inmutables, como suelen ser los objetos integrados y de valor. Por lo general, puede reemplazar
largas listas de parámetros con objetos inmutables, pero de lo contrario, debe tener cuidado con esto
grupo de refactorizaciones.

Una de las convenciones más valiosas que he usado a lo largo de los años es separar claramente los métodos que
cambiar el estado (modificadores) de aquellos que consultan el estado (consultas). No se cuantas veces he
me metí en problemas o vi a otros meterse en problemas al mezclarlos. Entonces cada vez que veo
combinados, yo uso Separate Query from Modifier para deshacerme de ellos.

Las buenas interfaces muestran solo lo que tienen que hacer y nada más. Puede mejorar una interfaz mediante
escondiendo cosas Por supuesto, todos los datos deben estar ocultos (espero no tener que decirte que hagas eso), pero
También cualquier método que pueda debe estar oculto. Cuando refactoriza a menudo necesita hacer cosas

visible por un tiempo y luego cúbralos con Ocultar método y Eliminar método de configuración .

Los constructores son una característica particularmente incómoda de Java y C ++, porque te obligan a saber La clase de un objeto que necesita crear. A menudo no necesitas saber esto. La necesidad de saber se puede eliminar con Reemplazar constructor con método de fábrica .

El casting es otra ruina de la vida del programador Java. En la medida de lo posible, trate de evitar hacer el usuario de una clase hace downcasting si puede contenerlo en otro lugar usando Encapsulate Abatido .

Java, como muchos lenguajes modernos, tiene un mecanismo de manejo de excepciones para cometer errores manejo más fácil. Los programadores que no están acostumbrados a esto a menudo usan códigos de error para señalar problemas. Puede usar Reemplazar código de error con excepción para usar las nuevas características excepcionales. Pero a veces las excepciones no son la respuesta correcta; primero debe probar con Reemplazar excepción con Prueba .

Page 221

Renombrar método

El nombre de un método no revela su propósito.

Cambiar el nombre del método.

Motivación

Una parte importante del estilo de código que estoy recomendando son los pequeños métodos para factorizar complejos procesos. Hecho mal, esto puede llevarte a un baile alegre para descubrir cuáles son todos los pequeños métodos hacer. La clave para evitar este alegre baile es nombrar los métodos. Los métodos deben nombrarse en un manera que comunica su intención. Una buena manera de hacer esto es pensar cuál es el comentario para el sería el método y convertir ese comentario en el nombre del método.

La vida es lo que es, no tendrás tus nombres correctos la primera vez. En esta situación, bien puede ser tentado a dejarlo, después de todo es solo un nombre. Ese es el trabajo del demonio malvado *Obfuscatis;* no lo hagas Escúchalo a él. Si ve un método mal nombrado, es imprescindible que lo cambie. Recuerda Su código es para un humano primero y una computadora en segundo lugar. Los humanos necesitan buenos nombres. Tomar nota de cuando ha pasado años intentando hacer algo que hubiera sido más fácil si un par de los métodos habían sido mejor nombrados. El buen nombramiento es una habilidad que requiere práctica; mejorando esta habilidad es la clave para ser un programador verdaderamente hábil. Lo mismo se aplica a otros aspectos de la firma. Si reordenar los parámetros aclara las cosas, hágalo (consulte Agregar parámetro y Eliminar parámetro).

Mecánica

- Verifique si la firma del método es implementada por una superclase o subclase.
  Si es así, realice estos pasos para cada implementación.
- Declarar un nuevo método con el nuevo nombre. Copie el viejo cuerpo del código al nuevo
  nombrar y hacer modificaciones para encajar.
- Compilar.
- Cambie el cuerpo del método antiguo para que llame al nuevo.

? rarr; *Si solo tiene algunas referencias, puede omitir esto razonablemente paso.*

- Compilar y probar.
- Encuentre todas las referencias al nombre del método anterior y cámbielas para referirse al nuevo.

  Compile y pruebe después de cada cambio.
- Eliminar el método anterior.

  ? rarr; *Si el método anterior es parte de la interfaz y no puede eliminarlo, déjelo en su lugar y márquelo como obsoleto.*

- Compilar y probar.

Página 222

Ejemplo

Tengo un método para obtener el número de teléfono de una persona:

```
public String getTelephoneNumber () {
     return ("(" + _officeAreaCode + ")" + _officeNumber);
}
```

Quiero cambiar el nombre del método a `getOfficeTelephoneNumber`. Empiezo creando el nuevo método y copiando el cuerpo al nuevo método. El antiguo método ahora cambia para llamar al uno nuevo:

```
Persona de clase ...
  public String getTelephoneNumber () {
       return getOfficeTelephoneNumber ();
  }
  public String getOfficeTelephoneNumber () {
       return ("(" + _officeAreaCode + ")" + _officeNumber);
  }
```

Ahora encuentro las personas que llaman del método anterior y las cambio para que llamen al nuevo. Cuando tengo los cambié todos, puedo eliminar el viejo método.

El procedimiento es el mismo si necesito agregar o eliminar un parámetro.

Si no hay muchas personas que llaman, las cambio para llamar al nuevo método sin usar el antiguo método como método de delegación. Si mis pruebas se tambalean, retrocedo y hago los cambios camino lento

Agregar parámetro

Un método necesita más información de su interlocutor.

Agregue un parámetro para un objeto que pueda transmitir esta información.

Motivación

Agregar parámetro es una refactorización muy común, una que seguramente ya ha hecho.
La motivación es simple. Debe cambiar un método, y el cambio requiere información que
no se pasó antes, por lo que agrega un parámetro.

En realidad, la mayor parte de lo que tengo que decir es motivación contra esta refactorización. A menudo tienes
Otras alternativas para agregar un parámetro. Si están disponibles, estas alternativas son mejores porque

Página 223

no conduzca a aumentar la longitud de las listas de parámetros. Las listas de parámetros largas huelen mal porque
son difíciles de recordar y a menudo involucran grupos de datos.

Mira los parámetros existentes. ¿Puedes pedirle a uno de esos objetos la información que necesitas?
Si no, ¿tendría sentido darles un método para proporcionar esa información? Que eres
utilizando la información para? Si ese comportamiento está en otro objeto, el que tiene el
¿información? Mire los parámetros existentes y piense en ellos con el nuevo parámetro.
Tal vez debería considerar Introducir objeto de parámetro .

No estoy diciendo que nunca debas agregar parámetros; Lo hago con frecuencia, pero debes ser consciente
de las alternativas

Mecánica

La mecánica de *Agregar parámetro* es muy similar a la del Método de cambio de nombre .

- Verifique si esta firma de método es implementada por una superclase o subclase.
  Si es así, realice estos pasos para cada implementación.
- Declarar un nuevo método con el parámetro agregado. Copie el antiguo cuerpo de código en el
  Nuevo método.

  ? rarr; *Si necesita agregar más de un parámetro, es más fácil agregar ellos al mismo tiempo.*

- Compilar.
- Cambie el cuerpo del método antiguo para que llame al nuevo.

  ? rarr; *Si solo tiene algunas referencias, puede omitir esto razonablemente paso.*

  ? rarr; *Puede proporcionar cualquier valor para el parámetro, pero generalmente usa nulo para parámetro de objeto y un valor claramente impar para tipos incorporados. Sus a menudo es una buena idea usar algo distinto de cero para los números, de modo que Puede detectar este caso más fácilmente.*

- Compilar y probar.
- Encuentre todas las referencias al método anterior y cámbielas para referirse al nuevo. Compilar
  y prueba después de cada cambio.
- Eliminar el método anterior.

? rarr; *Si el método anterior es parte de la interfaz y no puede eliminarlo, déjelo en su lugar y márquelo como obsoleto.*

• Compilar y probar.

Eliminar parámetro

El cuerpo del método ya no utiliza un parámetro.

Quitarlo

Página 224

Motivación

Los programadores a menudo agregan parámetros, pero son reacios a eliminarlos. Después de todo, un espurio El parámetro no causa ningún problema y es posible que lo necesite nuevamente más tarde.

Este es el demonio que habla *Obfuscatis* ; purgarlo de tu alma! Un parámetro indica información que se necesita; diferentes valores hacen la diferencia. Su interlocutor tiene que preocuparse qué valores pasar. Al no eliminar el parámetro, está trabajando para todos quien usa el método Eso no es una buena compensación, especialmente porque eliminar parámetros es un Refactorización fácil.

El caso a tener en cuenta aquí es un método polimórfico. En este caso, puede encontrar que otros Las implementaciones del método utilizan el parámetro. En este caso, no debe eliminar el parámetro. Puede optar por agregar un método separado que pueda usarse en esos casos, pero usted necesita examinar cómo usan el método las personas que llaman para ver si vale la pena hacerlo. Si algun las personas que llaman ya saben que están lidiando con cierta subclase y haciendo un trabajo extra para encontrar el parámetro o están utilizando el conocimiento de la jerarquía de clases para saber que pueden salirse con la suya, agregue un método adicional sin el parámetro. Si no necesitan saber qué clase tiene qué método, las personas que llaman deben quedar en la ignorancia dichosa.

Mecánica

La mecánica de *Eliminar parámetro* es muy similar a la de Cambiar nombre de método y Agregar Parámetro .

• Verifique si esta firma de método es implementada por una superclase o subclase.

Verifique si la clase o superclase usa el parámetro. Si es así, no hagas esto refactorización.

• Declarar un nuevo método sin el parámetro. Copie el viejo cuerpo del código al nuevo método.

? rarr; *Si necesita eliminar más de un parámetro, es más fácil quítalos juntos.*

• Compilar.

• Cambie el cuerpo del método antiguo para que llame al nuevo.

? rarr; *Si solo tiene algunas referencias, puede omitir esto razonablemente paso.*

• Compilar y probar.
• Encuentre todas las referencias al método anterior y cámbielas para referirse al nuevo. Compilar y prueba después de cada cambio.
• Eliminar el método anterior.

? rarr; *Si el método anterior es parte de la interfaz y no puede eliminarlo, déjelo en su lugar y márquelo como obsoleto.*

• Compilar y probar.

Como me siento bastante cómodo agregando y eliminando parámetros, a menudo hago un lote en uno ir.

Consulta separada del modificador

Tiene un método que devuelve un valor pero también cambia el estado de un objeto.

Cree dos métodos, uno para la consulta y otro para la modificación.

Motivación

Cuando tiene una función que le da un valor y no tiene efectos secundarios observables, tiene un cosa muy valiosa Puede llamar a esta función con la frecuencia que desee. Puedes mover la llamada a otro lugares en el método. En resumen, tiene mucho menos de qué preocuparse.

Es una buena idea señalar claramente la diferencia entre los métodos con efectos secundarios y aquellos sin. Una buena regla a seguir es decir que cualquier método que devuelva un valor no debería tener efectos secundarios observables Algunos programadores tratan esto como una regla absoluta [Meyer]. No tengo 100 % puro en esto (como en cualquier cosa), pero trato de seguirlo la mayor parte del tiempo, y me ha servido bien.

Si encuentra un método que devuelve un valor pero también tiene efectos secundarios, debe intentar separe la consulta del modificador.

Notarás que uso la frase efectos secundarios *observables* . Una optimización común es almacenar en caché el valor de una consulta en un campo para que las llamadas repetidas sean más rápidas. Aunque esto cambia el estado de la objeto con el caché, el cambio no es observable. Cualquier secuencia de consultas siempre devolverá los mismos resultados para cada consulta [Meyer].

Mecánica

• Cree una consulta que devuelva el mismo valor que el método original.

? rarr; *Mire en el método original para ver lo que se devuelve. Si el devuelto el valor es temporal, observe la ubicación de la asignación temporal.*

• Modifique el método original para que devuelva el resultado de una llamada a la consulta.

Página 226

? rarr; *Cada retorno en el método original debería decir* `return newQuery () en` *lugar de devolver cualquier otra cosa.*

? rarr; *Si el método usó una temperatura temporal con una sola asignación para capturar el valor de retorno, debería poder eliminarlo.*

• Compilar y probar.
• Para cada llamada, reemplace la llamada única al método original con una llamada a la consulta. Agrega un llame al método original antes de la línea que llama a la consulta. Compila y prueba después de cada cambiar a un método de llamada.
• Haga que el método original tenga un tipo de retorno nulo y elimine las expresiones de retorno.

Ejemplo

Aquí hay una función que me dice el nombre de un delincuente para un sistema de seguridad y envía una alerta. La regla es que solo se envía una alerta incluso si hay más de un delincuente:

```
String foundMiscreant (String [] people) {
     for (int i = 0; i <people.length; i ++) {
          if (people [i] .equals ("Don")) {
              sendAlert ();
              devolver "Don";
          }
          if (people [i] .equals ("John")) {
              sendAlert ();
              regresar "John";
          }
     }
     regreso "";
}
```

Es llamado por

```
vacío checkSecurity (String [] people) {
     Cadena encontrada = foundMiscreant (personas);
     someLaterCode (encontrado);
}
```

Para separar la consulta del modificador, primero necesito crear una consulta adecuada que devuelva el mismo valor que el modificador pero sin hacer los efectos secundarios.

```
 String foundPerson (String [] people) {
```

```
for (int i = 0; i <people.length; i ++) {
     if (people [i] .equals ("Don")) {
         devolver "Don";
     }
     if (people [i] .equals ("John")) {
         regresar "John";
     }
}
regreso "";
```


Página 227

```
}
```

Luego, reemplazo cada retorno en la función original, uno a la vez, con llamadas a la nueva consulta. yo prueba después de cada reemplazo. Cuando termine, el método original se verá así:

```
String foundMiscreant (String [] people) {
     for (int i = 0; i <people.length; i ++) {
          if (people [i] .equals ("Don")) {
              sendAlert ();
              return foundPerson (personas);
          }
          if (people [i] .equals ("John")) {
              sendAlert ();
              return foundPerson (personas);
          }
     }
     return foundPerson (personas);
}
```

Ahora modifico todos los métodos de llamada para hacer dos llamadas: primero al modificador y luego a la consulta:

```
vacío checkSecurity (String [] people) {
     foundMiscreant (personas);
     Cadena encontrada = foundPerson (personas);
     someLaterCode (encontrado);
}
```

Una vez que he hecho esto para todas las llamadas, puedo alterar el modificador para darle un tipo de retorno nulo:

```
void foundMiscreant (String [] people) {
     for (int i = 0; i <people.length; i ++) {
          if (people [i] .equals ("Don")) {
              sendAlert ();
              regreso;

          }
          if (people [i] .equals ("John")) {
              sendAlert ();
              regreso;
          }
     }
}
```

Ahora parece mejor cambiar el nombre del original:

```
vacío sendAlert (String [] people) {
     for (int i = 0; i <people.length; i ++) {
          if (people [i] .equals ("Don")) {
              sendAlert ();
              regreso;
```


Página 228

```
          }
          if (people [i] .equals ("John")) {
              sendAlert ();
              regreso;
          }
     }
}
```

Por supuesto, en este caso tengo mucha duplicación de código porque el modificador usa el cuerpo del consulta para hacer su trabajo. Ahora puedo usar el algoritmo sustituto en el modificador para aprovechar esta:

```
vacío sendAlert (String [] people) {
     if (! foundPerson (people) .equals (""))
          sendAlert ();
}
```

Problemas de concurrencia

Si está trabajando en un sistema multiproceso, sabrá que realizar las operaciones de prueba y configuración como La acción individual es un idioma importante. ¿Esto entra en conflicto con la consulta separada del modificador ? yo discutió este problema con Doug Lea y concluyó que no es así, pero debe hacer algo cosas adicionales Todavía es valioso tener operaciones separadas de consulta y modificación. Sin embargo, usted necesita retener un tercer método que haga ambas cosas. La operación de consulta y modificación llamará al separe los métodos de consulta y modificación y esté sincronizado. Si las operaciones de consulta y modificación son no sincronizado, también puede restringir su visibilidad a nivel de paquete o privado. De esa manera tu tener una operación segura y sincronizada descompuesta en dos métodos más fáciles de entender. Estas los métodos de nivel inferior están disponibles para otros usos.

Método de parametrización

Varios métodos hacen cosas similares pero con diferentes valores contenidos en el cuerpo del método.

Cree un método que use un parámetro para los diferentes valores.

Motivación

Es posible que vea un par de métodos que hacen cosas similares pero varían según algunos valores. En En este caso, puede simplificar las cosas reemplazando los métodos separados con un método único que maneja las variaciones por parámetros. Tal cambio elimina el código duplicado y aumenta flexibilidad, porque puede manejar otras variaciones agregando parámetros.

Mecánica

- Cree un método parametrizado que se pueda sustituir por cada método repetitivo.
- Compilar.
- Reemplace un método antiguo con una llamada al nuevo método.
- Compilar y probar.
- Repita para todos los métodos, probando después de cada uno.

Puede encontrar que no puede hacer esto para todo el método, pero puede hacerlo por un fragmento de un método. En este caso, primero extraiga el fragmento en un método, luego parametrice ese método.

Ejemplo

El caso más simple son los métodos en las siguientes líneas:

```
Empleado de clase {
   anular tenPercentRaise () {
        salario * = 1.1;
}

   anular fivePercentRaise () {
        salario * = 1.05;
}
```

que puede ser reemplazado por

```
aumento nulo (factor doble) {
   salario * = (1 + factor);
}
```

Por supuesto, eso es tan simple que cualquiera lo detectaría.

Un caso menos obvio es el siguiente:

```
   dólares protegidos baseCharge () {
        resultado doble = Math.min (lastUsage (), 100) * 0.03;
        if (lastUsage ()> 100) {
             resultado + = (Math.min (lastUsage (), 200) - 100) * 0.05;
        };
        if (lastUsage ()> 200) {
             resultado + = (lastUsage () - 200) * 0.07;
        };
        devolver nuevos dólares (resultado);
   }
```

esto puede ser reemplazado con

```
     dólares protegidos baseCharge () {
          resultado doble = usoInRange (0, 100) * 0.03;
          resultado + = useInRange (100,200) * 0.05;
          resultado + = useInRange (200, Integer.MAX_VALUE) * 0.07;
```



```
          devolver nuevos dólares (resultado);
     }

     Protegido int useInRange (int start, int end) {
          if (lastUsage ()> start) return Math.min (lastUsage (), end) -
comienzo;
          de lo contrario, devuelve 0;
     }
```

El truco consiste en detectar el código que es repetitivo en función de algunos valores que se pueden pasar como parámetros

Reemplazar parámetro con métodos explícitos

Tiene un método que ejecuta un código diferente según los valores de una lista parámetro.

Cree un método separado para cada valor del parámetro.

```
void setValue (nombre de cadena, valor int) {
     if (nombre.equals ("altura"))
          _ altura = valor;
     if (name.equals ("ancho"))
          _ ancho = valor;
     Assert.shouldNeverReachHere ();
}
```

```
vacío setHeight (int arg) {
     _height = arg;
}
void setWidth (int arg) {
     _width = arg;
}
```

Motivación

Reemplazar parámetro con métodos explícitos es el reverso del método de parametrización . Lo normal el caso para el primero es que tiene valores discretos de un parámetro, pruebe esos valores en un condicional, y hacer cosas diferentes. La persona que llama tiene que decidir qué quiere hacer configurando el parámetro, por lo que también podría proporcionar diferentes métodos y evitar el condicional. No solo evitar el comportamiento condicional pero también ganar tiempo de compilación comprobando. Además su interfaz También es más claro. Con el parámetro, cualquier programador que use el método no solo necesita mirar los métodos en la clase pero también para determinar un valor de parámetro válido. Este último es a menudo pobre documentado

La claridad de la interfaz explícita puede valer la pena incluso cuando la comprobación del tiempo de compilación no una ventaja. `Switch.beOn ()` es mucho más claro que `Switch.setState (true)`, incluso cuando todos

lo que está haciendo es establecer un campo booleano interno.

Página 231

No debe usar *Reemplazar parámetro con métodos explícitos* cuando los valores de los parámetros sean probables para cambiar mucho Si esto sucede y solo está configurando un campo para el parámetro pasado, use un setter simple Si necesita un comportamiento condicional, necesita Reemplazar condicional por El polimorfismo .

Mecánica

- Cree un método explícito para cada valor del parámetro.
- Para cada tramo del condicional, llame al nuevo método apropiado.
- Compile y pruebe después de cambiar cada pierna.
- Reemplace a cada persona que llama del método condicional con una llamada al nuevo método apropiado.
- Compilar y probar.
- Cuando se cambian todas las llamadas, elimine el método condicional.

Ejemplo

Quiero crear una subclase de empleados sobre la base de un parámetro aprobado, a menudo el resultado de Reemplazar constructor con método de fábrica :

```
    static final int INGENIERO = 0;
    static final int VENDEDOR = 1;
    static final int MANAGER = 2;

    Empleado estático crear (tipo int) {
         interruptor (tipo) {
              ingeniero de caso:
                  volver nuevo ingeniero ();
              caso VENDEDOR:
                  volver nuevo vendedor ();
              administrador de casos:
                  volver nuevo Manager ();
              defecto:
                  lanzar nueva IllegalArgumentException ("Código de tipo incorrecto
valor");
         }
    }
```

Debido a que este es un método de fábrica, no puedo usar Reemplazar condicional con polimorfismo , porque aún no he creado el objeto. No espero demasiadas subclases nuevas, así que explícito La interfaz tiene sentido. Primero creo los nuevos métodos:

```
    Empleado estático createEngineer () {
         volver nuevo ingeniero ();
    }
    Empleado estático createSalesman () {
         volver nuevo vendedor ();
    }
    Empleado estático createManager () {
         volver nuevo Manager ();
```

```
    }
```

Uno por uno, reemplazo los casos en las declaraciones de cambio con llamadas a los métodos explícitos:

231

Página 232

```
    Empleado estático crear (tipo int) {
         interruptor (tipo) {
              ingeniero de caso:
                  return Employee.createEngineer ();
              caso VENDEDOR:
                  volver nuevo vendedor ();
              administrador de casos:
                  volver nuevo Manager ();
              defecto:
                  lanzar nueva IllegalArgumentException ("Código de tipo incorrecto
valor");
         }
    }
```

Compilo y pruebo después de cambiar cada pierna, hasta que las he reemplazado todas:

```
    Empleado estático crear (tipo int) {
         interruptor (tipo) {
              ingeniero de caso:
                  return Employee.createEngineer ();
              caso VENDEDOR:
                  return Employee.createSalesman ();
              administrador de casos:
                  return Employee.createManager ();
              defecto:
                  lanzar nueva IllegalArgumentException ("Código de tipo incorrecto
valor");
         }
    }
```

Ahora paso a las llamadas del antiguo método de creación. Cambio código como

```
   Empleado kent = Employee.create (INGENIERO)
```

a

```
   Empleado kent = Employee.createEngineer ()
```

Una vez que he hecho eso para todas las personas que llaman `crear,` puedo eliminar el método de `creación` . Yo tambien puedo ser capaz de deshacernos de las constantes.

Preservar todo el objeto

Obtiene varios valores de un objeto y pasa estos valores como parámetros en un llamada al método

Envía todo el objeto en su lugar.

```
int low = daysTempRange (). getLow ();
```

232

Página 233

```
int high = daysTempRange (). getHigh ();
insidePlan = plan.withinRange (bajo, alto);
```

```
insidePlan = plan.withinRange (daysTempRange ());
```

Motivación

Este tipo de situación surge cuando un objeto pasa varios valores de datos de un solo objeto como parámetros en una llamada al método. El problema con esto es que si el objeto llamado necesita nuevos datos valores posteriores, debe buscar y cambiar todas las llamadas a este método. Puedes evitar esto si pasando todo el objeto del que provienen los datos. El objeto llamado puede pedir lo que quiera de todo el objeto.

Además de hacer que la lista de parámetros sea más robusta a los cambios, *Preserve Whole Object a* menudo hace que el código sea más legible. Las listas largas de parámetros pueden ser difíciles de trabajar porque ambos La persona que llama y la persona que llama tienen que recordar qué valores estaban allí. También alientan duplicados código porque el objeto llamado no puede aprovechar ningún otro método en todo el objeto para calcular valores intermedios

Hay un inconveniente. Cuando pasa valores, el objeto llamado depende de valores, pero no hay ninguna dependencia del objeto del que se extrajeron los valores. Pasar el objeto requerido provoca una dependencia entre el objeto requerido y el llamado objeto. Si esto va a estropear su estructura de dependencia, no use *Preserve Whole Object.*

Otra razón que he escuchado para no usar *Preserve Whole Object* es que cuando un objeto invocador solo necesita un valor del objeto requerido, es mejor pasar el valor que pasar el objeto completo No me suscribo a esa vista. Un valor y un objeto equivalen a lo mismo cuando los pasa, al menos por razones de claridad (puede haber un costo de rendimiento al pasar por parámetros de valor). La fuerza impulsora es el problema de la dependencia.

Que un método llamado utilice muchos valores de otro objeto es una señal de que el método llamado debería definirse realmente en el objeto del que provienen los valores. Cuando estas considerando *Preserve todo el objeto,* considere el método Move como una alternativa.

Es posible que aún no tenga todo el objeto definido. En este caso necesitas Introducir Parámetro objeto .

Un caso común es que un objeto que llama pasa varios de sus *propios* valores de datos como parámetros. En en este caso, puede realizar la llamada y pasar `esto` en lugar de estos valores, si tiene el métodos de obtención apropiados y no le importa la dependencia.

Mecánica

- Cree un nuevo parámetro para todo el objeto del que provienen los datos.
- Compilar y probar.
- Determine qué parámetros deben obtenerse del objeto completo.
- Tome un parámetro y reemplace las referencias dentro del cuerpo del método invocando un método apropiado en todo el parámetro del objeto.
- Eliminar el parámetro.
- Compilar y probar.

• Repita para cada parámetro que se pueda obtener de todo el objeto.

Página 234

• Elimine el código en el método de llamada que obtiene los parámetros eliminados.

? rarr; *A menos que, por supuesto, el código esté usando estos parámetros en alguna parte más.*

• Compilar y probar.

Ejemplo

Considere un objeto de habitación que registra temperaturas altas y bajas durante el día. Necesita compare esta gama con una gama en un plan de calefacción predefinido:

```
salón de clases...
  boolean dentro del plan (plan del plan de calefacción) {
       int low = daysTempRange (). getLow ();
       int high = daysTempRange (). getHigh ();
       plan de retorno dentro de Rango (bajo, alto);
  }
Clase de Calefacción ...
  boolean insideRange (int bajo, int alto) {
       return (bajo> = _range.getLow () && alto <= _range.getHigh ());
  }
  TempRange privado _range;
```

En lugar de descomprimir la información del rango cuando la paso, puedo pasar todo el objeto de rango. En En este caso simple, puedo hacer esto en un solo paso. Cuando hay más parámetros involucrados, puedo hacerlo en pasos más pequeños Primero agrego todo el objeto a la lista de parámetros:

```
Clase de Calefacción ...
  boolean insideRange ( TempRange roomRange, int low, int high) {
       return (bajo> = _range.getLow () && alto <= _range.getHigh ());
  }

salón de clases...
  boolean dentro del plan (plan del plan de calefacción) {
       int low = daysTempRange (). getLow ();
       int high = daysTempRange (). getHigh ();
       plan de devolución.withinRange ( daysTempRange (), bajo, alto);
  }
```

Luego uso un método en todo el objeto en lugar de uno de los parámetros:

```
Clase de Calefacción ...
  boolean insideRange (TempRange roomRange, int high) {
       return ( roomRange.getLow () > = _range.getLow () && high <=
_range.getHigh ());
  }

salón de clases...
  boolean dentro del plan (plan del plan de calefacción) {
       int low = daysTempRange (). getLow ();
```

Página 235

```
          int high = daysTempRange (). getHigh ();
          plan de devolución.withinRange (daysTempRange (), alto);
     }
```

Continúo hasta que he cambiado todo lo que necesito:

```
   Clase de Calefacción ...
     boolean dentro de Rango (TempRange roomRange) {
          return (roomRange.getLow ()> = _range.getLow () &&
roomRange.getHigh () <= _range.getHigh ());
     }
   salón de clases...
     boolean dentro del plan (plan del plan de calefacción) {
          int low = daysTempRange (). getLow ();
          int high = daysTempRange (). getHigh ();
          plan de devolución.withinRange (daysTempRange ());
     }
```

Ahora ya no necesito las temperaturas:

```
   salón de clases...
     boolean dentro del plan (plan del plan de calefacción) {
          int low = daysTempRange (). getLow ();
          int high = daysTempRange (). getHigh ();
          plan de devolución.withinRange (daysTempRange ());
     }
```

El uso de objetos completos de esta manera pronto lo lleva a darse cuenta de que puede mover útilmente el comportamiento a todo el objeto para que sea más fácil trabajar con él.

```
   Clase de Calefacción ...
     boolean dentro de Rango (TempRange roomRange) {
          return ( _range.includes (roomRange) );
     }
   clase TempRange ...
     boolean incluye (TempRange arg) {
          return arg.getLow ()> = this.getLow () && arg.getHigh () <=
this.getHigh ();
     }
```

Reemplazar parámetro con método

Un objeto invoca un método, luego pasa el resultado como un parámetro para un método. El receptor También puede invocar este método.

Elimine el parámetro y deje que el receptor invoque el método.

```
        int basePrice = _quantity * _itemPrice;
        discountLevel = getDiscountLevel ();
        double finalPrice = precio descontado (basePrice, discountLevel);
```

Página 236

```
int basePrice = _quantity * _itemPrice;
double finalPrice = precio con descuento (basePrice);
```

Motivación

Si un método puede obtener un valor que se pasa como parámetro por otro medio, debería hacerlo. Largo
Las listas de parámetros son difíciles de entender y debemos reducirlas lo más posible.

Una forma de reducir las listas de parámetros es observar si el método de recepción puede hacer que
mismo cálculo Si un objeto está llamando a un método en sí mismo, y el cálculo del parámetro
no hace referencia a ninguno de los parámetros del método de llamada, debe poder eliminar el
parámetro convirtiendo el cálculo en su propio método. Esto también es cierto si está llamando a un
método en un objeto diferente que tiene una referencia al objeto que llama.

No puede eliminar el parámetro si el cálculo se basa en un parámetro del método de llamada,
porque ese parámetro puede cambiar con cada llamada (a menos que, por supuesto, ese parámetro pueda ser
reemplazado por un método). Tampoco puede eliminar el parámetro si el receptor no tiene un
referencia al remitente, y no desea darle uno.

En algunos casos, el parámetro puede estar allí para una parametrización futura del método. En esto
caso todavía me desharía de él. Maneje la parametrización cuando la necesite; puedes descubrir
que no tienes el parámetro correcto de todos modos. Haría una excepción a esta regla solo cuando
el cambio resultante en la interfaz tendría consecuencias dolorosas en todo el conjunto
programa, como una compilación larga o el cambio de una gran cantidad de código incrustado. Si esto te preocupa, mira
cuán doloroso sería realmente ese cambio. También debe mirar para ver si puede reducir
Las dependencias que hacen que el cambio sea tan doloroso. Las interfaces estables son buenas, pero congeladas
Una mala interfaz es un problema.

Mecánica

- Si es necesario, extraiga el cálculo del parámetro en un método.
- Reemplace las referencias al parámetro en los cuerpos de los métodos con referencias al método.
- Compilar y probar después de cada reemplazo.
- Use Eliminar parámetro en el parámetro.

Ejemplo

Otra variación poco probable en las órdenes de descuento es la siguiente:

```
public double getPrice () {
     int basePrice = _quantity * _itemPrice;
     int discountLevel;
     if (_quantity> 100) discountLevel = 2;
     de lo contrario discountLevel = 1;
     double finalPrice = precio descontado (basePrice, discountLevel);
     volver finalPrecio;
}

Precio privado doble con descuento (int basePrice, int discountLevel) {
```

Página 237

```
     if (discountLevel == 2) return basePrice * 0.1;
     de lo contrario, volver basePrice * 0.05;
}
```

Puedo comenzar extrayendo el cálculo del nivel de descuento:

```
public double getPrice () {
     int basePrice = _quantity * _itemPrice;
     int discountLevel = getDiscountLevel ();
     double finalPrice = precio descontado (basePrice, discountLevel);
     volver finalPrecio;
}

private int getDiscountLevel () {
     if (_quantity> 100) devuelve 2;
     de lo contrario, devuelve 1;
}
```

Luego reemplazo las referencias al parámetro en descontadoPrecio:

```
Precio privado doble con descuento (int basePrice, int discountLevel) {
     if ( getDiscountLevel () == 2) return basePrice * 0.1;
     de lo contrario, volver basePrice * 0.05;
}
```

Entonces puedo usar Eliminar parámetro :

```
public double getPrice () {
     int basePrice = _quantity * _itemPrice;
     int discountLevel = getDiscountLevel ();
     double finalPrice = precio con descuento (basePrice);
     volver finalPrecio;
}

Precio privado doble con descuento (int basePrice) {
     if (getDiscountLevel () == 2) return basePrice * 0.1;
     de lo contrario, volver basePrice * 0.05;
}
```

Ahora puedo deshacerme de la temperatura:

```
public double getPrice () {
     int basePrice = _quantity * _itemPrice;
     double finalPrice = precio con descuento (basePrice);
     volver finalPrecio;
}
```

Entonces es hora de deshacerse del otro parámetro y su temperatura. Me queda con

Página 238

```
public double getPrice () {
     volver con descuentoPrecio ();
}

precio doble con descuento privado () {
     if (getDiscountLevel () == 2) return getBasePrice () * 0.1;
     de lo contrario, devuelve getBasePrice () * 0.05;
}

privado doble getBasePrice () {
     return _quantity * _itemPrice;
}
```

así que también podría usar el método en línea con `descuento`

```
privado doble getPrice () {
     if (getDiscountLevel () == 2) return getBasePrice () * 0.1;
     de lo contrario, devuelve getBasePrice () * 0.05;
}
```

Introducir objeto de parámetro

Tienes un grupo de parámetros que naturalmente van de la mano.

Reemplácelos con un objeto.

Motivación

A menudo ve un grupo particular de parámetros que tienden a pasarse juntos. Varios métodos puede usar este grupo, ya sea en una clase o en varias clases. Tal grupo de clases es un dato grupo y puede ser reemplazado con un objeto que lleva todos estos datos. Vale la pena convertir estos parámetros en objetos solo para agrupar los datos juntos. Esta refactorización es útil porque reduce el tamaño de las listas de parámetros, y las largas listas de parámetros son difíciles de entender. los los accesores definidos en el nuevo objeto también hacen que el código sea más consistente, lo que nuevamente lo hace Más fácil de entender y modificar.

Sin embargo, obtiene un beneficio más profundo porque una vez que ha agrupado los parámetros, pronto verá un comportamiento que también puede pasar a la nueva clase. A menudo los cuerpos de los Los métodos tienen manipulaciones comunes de los valores de los parámetros. Al trasladar este comportamiento al nuevo objeto, puede eliminar una gran cantidad de código duplicado.

Mecánica

Página 239

- Cree una nueva clase para representar el grupo de parámetros que está reemplazando. Hacer el clase inmutable
- Compilar.
- Use Agregar parámetro para el nuevo grupo de datos. Utilice un valor nulo para este parámetro en todos los personas que llaman

  ? rarr; *Si tiene muchas personas que llaman, puede conservar la firma anterior y dejarla llama al nuevo método. Aplique primero la refactorización en el método anterior. Tú luego puede mover las llamadas una por una y eliminar el método anterior cuando termines.*

- Para cada parámetro en el grupo de datos, elimine el parámetro de la firma. Modificar los llamadores y el cuerpo del método para usar el objeto de parámetro para ese valor.
- Compile y pruebe después de eliminar cada parámetro.
- Cuando haya eliminado los parámetros, busque el comportamiento que puede mover al objeto de parámetro con Move Method .

  ? rarr; *Esto puede ser un método completo o parte de un método. Si es parte de un método, use primero el* Método de extracción *y luego mueva el nuevo método.*

Ejemplo

Comienzo con una cuenta y entradas. Las entradas son titulares de datos simples.

```
Entrada de clase ...
  Entrada (valor doble, Fecha cargo Fecha) {
       _value = value;
       _chargeDate = chargeDate;
  }
  Fecha getDate () {
       return _chargeDate;
  }
  doble getValue () {
       return _value;
  }
  fecha privada _chargeDate;
  valor privado doble;
```

Me centro en la cuenta, que contiene una colección de entradas y tiene un método para determinar El flujo de la cuenta entre dos fechas:

```
Cuenta de clase ...
  double getFlowBetween (Fecha de inicio, Fecha de finalización) {
       doble resultado = 0;
       Enumeración e = _entries.elements ();
       while (e.hasMoreElements ()) {
            Entrada cada = (Entrada) e.nextElement ();
            if (each.getDate (). es igual a (inicio) ||
                 each.getDate (). equals (end) ||
                   (each.getDate (). after (start) &&
each.getDate (). before (end)))
            {
```

```
                    resultado + = each.getValue ();
               }
          }
          resultado de retorno;
     }
     entradas privadas de vectores = nuevo vector ();

   codigo del cliente...
     doble flujo = anAccount.getFlowBetween (startDate, endDate);
```

No sé cuántas veces me encuentro con pares de valores que muestran un rango, como inicio y fechas de finalización y números superiores e inferiores. Puedo entender por qué sucede esto, después de todo lo hice todo el tiempo yo mismo. Pero desde que vi el patrón de rango [Fowler, AP] siempre trato de usar rangos en su lugar. Mi primer paso es declarar un titular de datos simple para el rango:

```
   class DateRange {
     DateRange (Fecha de inicio, Fecha de finalización) {
          _start = inicio;
          _end = fin;
     }
     Fecha getStart () {
          return _start;
     }
     Fecha getEnd () {
          volver _end;
     }
     Fecha final privada _start;
     Fecha final privada _end;
   }
```

He hecho que la clase de rango de fechas sea inmutable; es decir, todos los valores para el rango de fechas son finales y establecido en el constructor, por lo tanto, no hay métodos para modificar los valores. Este es un movimiento sabio para evitar errores de alias. Como Java tiene parámetros de paso por valor, hace que la clase sea inmutable imita la forma en que funcionan los parámetros de Java, por lo que esta es la suposición correcta para esta refactorización.

A continuación, agrego el rango de fechas en la lista de parámetros para el método `getFlowBetween` :

```
   Cuenta de clase ...
     doble getFlowBetween (Fecha de inicio, Fecha de finalización, Rango de Rango de fechas ) {
          doble resultado = 0;
          Enumeración e = _entries.elements ();
          while (e.hasMoreElements ()) {
               Entrada cada = (Entrada) e.nextElement ();
               if (each.getDate (). es igual a (inicio) ||
                    each.getDate (). equals (end) ||
                    (each.getDate (). after (start) &&
each.getDate (). before (end)))
               {
                    resultado + = each.getValue ();
               }
          }
          resultado de retorno;
     }
```

Página 241

```
   codigo del cliente...
     flujo doble = anAccount.getFlowBetween (startDate, endDate, null);
```

En este punto solo necesito compilar, porque todavía no he alterado ningún comportamiento.

El siguiente paso es eliminar uno de los parámetros y utilizar el nuevo objeto. Para hacer esto yo elimine el parámetro de inicio y modifique el método y sus llamadores para usar el nuevo objeto en su lugar:

```
   Cuenta de clase ...
     double getFlowBetween (Fecha de finalización, rango de Rango de fechas) {
          doble resultado = 0;
          Enumeración e = _entries.elements ();
          while (e.hasMoreElements ()) {
               Entrada cada = (Entrada) e.nextElement ();
               if (each.getDate (). equals ( range.getStart () ) ||
                    each.getDate (). equals (end) ||
                    (each.getDate (). after ( range.getStart () ) &&
each.getDate (). before (end)))
               {
                    resultado + = each.getValue ();
               }
          }
          resultado de retorno;
     }

   codigo del cliente...
     doble flujo = anAccount.getFlowBetween (endDate, new DateRange
(startDate, null) );
```

Luego elimino la fecha de finalización:

```
   Cuenta de clase ...
     doble getFlowBetween (rango de DateRange) {
          doble resultado = 0;
          Enumeración e = _entries.elements ();
          while (e.hasMoreElements ()) {
               Entrada cada = (Entrada) e.nextElement ();
               if (each.getDate (). equals (range.getStart ()) ||
                    each.getDate (). equals ( range.getEnd () ) ||
                    (each.getDate (). after (range.getStart ()) &&
each.getDate (). before ( range.getEnd () ))
               {
                    resultado + = each.getValue ();
               }
          }
          resultado de retorno;
     }

   codigo del cliente...
          doble flujo = anAccount.getFlowBetween (nuevo DateRange
(startDate, endDate ));
```


Página 242

He introducido el objeto de parámetro; sin embargo, puedo obtener más valor de esta refactorización al Mover el comportamiento de otros métodos al nuevo objeto. En este caso puedo tomar el código en el condición y uso del método de extracción y el método de movimiento para obtener

```
Cuenta de clase ...
   doble getFlowBetween (rango de DateRange) {
        doble resultado = 0;
        Enumeración e = _entries.elements ();
        while (e.hasMoreElements ()) {
             Entrada cada = (Entrada) e.nextElement ();
             if ( range.includes (each.getDate ()) ) {
                  resultado + = each.getValue ();
             }
        }
        resultado de retorno;
   }

clase DateRange ...
   boolean incluye (fecha arg) {
        return (arg.equals (_start) ||
                  arg.equals (_end) ||
                    (arg.after (_start) && arg.before (_end)));
   }
```

Usualmente hago extractos y movimientos simples como este en un solo paso. Si me encuentro con un error, puedo retroceder y da los dos pasos más pequeños.

Eliminar método de configuración

Un campo debe establecerse en el momento de la creación y nunca modificarse.

Elimine cualquier método de configuración para ese campo.

Motivación

Proporcionar un método de configuración indica que se puede cambiar un campo. Si no quieres que ese campo cambie una vez que se crea el objeto, luego no proporcione un método de configuración (y haga que el campo sea final). De esa forma, su intención es clara y, a menudo, elimina la posibilidad de que el campo cambio.

Esta situación a menudo ocurre cuando los programadores usan ciegamente acceso variable indirecto [Beck]. Tal los programadores luego usan setters incluso en un constructor. Supongo que hay un argumento para la consistencia. pero no se compara con la confusión que el método de configuración causará más adelante.

Mecánica

- Compilar y probar.
- Compruebe que el método de configuración solo se llama en el constructor o en un método llamado por El constructor.
- Modificar el constructor para acceder a las variables directamente.

> ? rarr; *No puede hacer esto si tiene una subclase que configura los campos privados de una superclase En este caso, debe intentar proporcionar una protección Método de superclase (idealmente un constructor) para establecer estos valores. Lo que sea no le des al método de la superclase un nombre que lo confunda con Un método de configuración.*

- Compilar y probar.
- Elimine el método de configuración y finalice el campo.
- Compilar.

Ejemplo

Un ejemplo simple es el siguiente:

```
Cuenta de clase {

   string privada _id;

   Cuenta (ID de cadena) {
        setId (id);
   }

   void setId (String arg) {
       _id = arg;
   }
```

que puede ser reemplazado por

```
Cuenta de clase {

   Cadena privada final _id;

   Cuenta (ID de cadena) {
       _id = id;
   }
```

Los problemas vienen en algunas variaciones. Primero es el caso en el que estás haciendo cálculos en el argumento:

```
Cuenta de clase {

   string privada _id;

   Cuenta (ID de cadena) {
        setId (id);
   }
```

```
   void setId (String arg) {
       _id = "ZZ" + arg;
   }
```

Si el cambio es simple (como aquí) y solo hay un constructor, puedo hacer el cambio en el constructor. Si el cambio es complejo o necesito llamarlo desde métodos separados, debo proporcionar un método. En ese caso, necesito nombrar el método para aclarar su intención:

```
Cuenta de clase {
   Cadena privada final _id;
   Cuenta (ID de cadena) {
        initializeId (id);
   }

   void initializeId (String arg) {
       _id = "ZZ" + arg;
   }
```

Un caso incómodo radica en las subclases que inicializan variables de superclase privadas:

```
clase InterestAccount extiende la cuenta ...

   privado doble _interestRate;

   InterestAccount (ID de cadena, tasa doble) {
        setId (id);
        _interestRate = tasa;
   }
```

El problema es que no puedo acceder a la `identificación` directamente para configurarlo. La mejor solución es usar una superclase constructor:

```
clase Cuenta de Interés ...

   InterestAccount (ID de cadena, tasa doble) {
        super (id);
        _interestRate = tasa;
   }
```

Si eso no es posible, un método bien nombrado es lo mejor para usar:

```
clase Cuenta de Interés ...

   InterestAccount (ID de cadena, tasa doble) {
        initializeId (id);
        _interestRate = tasa;
   }
```

Otro caso a considerar es establecer el valor de una colección:

```
Persona de clase {
  Vector getCourses () {
       volver _ cursos;
  }
  void setCourses (Vector arg) {
      _courses = arg;
  }
  cursos privados de vectores;
```

Aquí quiero reemplazar el setter con operaciones de agregar y quitar. Hablo de esto en
Colección encapsulada .

Método oculto

Un método no es utilizado por ninguna otra clase.

Haz que el método sea privado.

Motivación

La refactorización a menudo hace que cambie las decisiones sobre la visibilidad de los métodos. Es fácil
detectar casos en los que necesita hacer que un método sea más visible: otra clase lo necesita y usted
así relajar la visibilidad. Es algo más difícil saber cuándo un método es demasiado visible. Idealmente un
La herramienta debe verificar todos los métodos para ver si se pueden ocultar. Si no es así, deberías hacer
esta comprobación a intervalos regulares

Un caso particularmente común es ocultar los métodos de obtención y configuración a medida que trabaja en un entorno más rico.
interfaz que proporciona más comportamiento. Este caso es más común cuando comienza con un
clase que es poco más que un titular de datos encapsulado. A medida que se incorpora más comportamiento a la clase,
es posible que muchos de los métodos de obtención y configuración ya no sean necesarios públicamente, en
en cuyo caso se pueden ocultar. Si hace que un método de obtención o configuración sea privado y está utilizando
acceso variable directo, puede eliminar el método.

Mecánica

• Verifique regularmente las oportunidades para hacer un método más privado.

> ? rarr; *Use una herramienta estilo pelusa, haga verificaciones manuales de vez en cuando y verifique cuando elimina una llamada a un método en otra clase.*
>
> ? rarr; *En particular, busque casos como este con los métodos de configuración.*

• Haga que cada método sea lo más privado posible.
• Compilar después de hacer un grupo de escondites.

Página 246

? rarr; *El compilador verifica esto naturalmente, por lo que no necesita compilar*

con cada cambio Si uno sale mal, es fácil de detectar.

Reemplazar constructor con método de fábrica

Desea hacer algo más que una simple construcción cuando crea un objeto.

Reemplace el constructor con un método de fábrica.

```
Empleado (tipo int) {
     _type = type;
}
```

```
Empleado estático crear (tipo int) {
      devolver nuevo empleado (tipo);
}
```

Motivación

La motivación más obvia para *reemplazar el constructor con el método de fábrica* viene con el reemplazo Un código de tipo con subclases. Tiene un objeto que a menudo se crea con un código de tipo pero ahora Necesita subclases. La subclase exacta se basa en el código de tipo. Sin embargo, los constructores pueden solo devuelve una instancia del objeto que se solicita. Por lo tanto, debe reemplazar el constructor con Un método de fábrica [Gang of Four].

Puede usar métodos de fábrica para otras situaciones en las que los constructores son demasiado limitados. Fábrica Los métodos son esenciales para cambiar el valor de referencia . También se pueden usar para señalar comportamiento de creación diferente que va más allá del número y los tipos de parámetros.

Mecánica

- Crear un método de fábrica. Haga que su cuerpo llame al constructor actual.
- Reemplace todas las llamadas al constructor con llamadas al método de fábrica.
- Compilar y probar después de cada reemplazo.
- Declarar el constructor privado.
- Compilar.

Ejemplo

Un ejemplo rápido pero fastidioso y complicado es el sistema de pago de los empleados. Tengo siguiente empleado:

```
Empleado de clase {

   tipo int privado;
   static final int INGENIERO = 0;
   static final int VENDEDOR = 1;
   static final int MANAGER = 2;
```



```
   Empleado (tipo int) {
```

```
    }     _type = type;
```

Quiero hacer subclases de Empleado para que correspondan con los códigos de tipo. Entonces necesito crear un método de fábrica:

```
    Empleado estático crear (tipo int) {
        devolver nuevo empleado (tipo);
    }
```

Luego cambio todas las llamadas del constructor para usar este nuevo método y hacer que el constructor privado:

```
 codigo del cliente...
    Employee eng = Employee.create (Employee.ENGINEER);

 Empleado de clase ...
    Empleado privado (tipo int) {
        _type = type;
    }
```

Ejemplo: crear subclases con una cadena

Hasta ahora no tengo una gran ganancia; la ventaja viene del hecho de que separé el receptor de la llamada de creación de la clase de objeto creado. Si luego aplico *Reemplazar código de tipo con Subclases* (Capítulo 8) para convertir los códigos en subclases de empleados, puedo ocultar estos subclases de clientes utilizando el método de fábrica:

```
    Empleado estático crear (tipo int) {
         interruptor (tipo) {
              ingeniero de caso:
                  volver nuevo ingeniero ();
              caso VENDEDOR:
                  volver nuevo vendedor ();
              administrador de casos:
                  volver nuevo Manager ();
              defecto:
                  lanzar nueva IllegalArgumentException ("Código de tipo incorrecto
valor");
         }
    }
```

Lo triste de esto es que tengo un interruptor. Debo agregar una nueva subclase, tengo que recordar para actualizar esta declaración de cambio, y tiendo al olvido.

Una buena forma de evitar esto es usar `Class.forName`. Lo primero que debe hacer es cambiar el tipo de El parámetro, esencialmente una variación del método Rename . Primero creo un nuevo método que toma una cadena como argumento:

Página 248

```
    Empleado estático crear (nombre de cadena) {
         tratar {
              return (Empleado) Class.forName (nombre) .newInstance ();
```

```
        } catch (Excepción e) {
            lanzar una nueva IllegalArgumentException ("No se puede crear una instancia"
+ nombre);
        }
    }
```

Luego convierto el entero creado para usar este nuevo método:

```
 Empleado de clase {
   Empleado estático crear (tipo int) {
        interruptor (tipo) {
            ingeniero de caso:
                return create ("Ingeniero");
            caso VENDEDOR:
                return create ("Vendedor");
            administrador de casos:
                return create ("Administrador");
            defecto:
                lanzar nueva IllegalArgumentException ("Código de tipo incorrecto
valor");
        }
    }
```

Entonces puedo trabajar en las personas que llaman de create para cambiar declaraciones como

```
 Employee.create (INGENIERO)
```

a

```
 Employee.create ("Ingeniero")
```

Cuando termine, puedo eliminar la versión del parámetro entero del método.

Este enfoque es bueno ya que elimina la necesidad de actualizar el método de creación a medida que agrego nuevo subclases de empleado. Sin embargo, el enfoque carece de tiempo de compilación: un error ortográfico conduce a un error de tiempo de ejecución. Si esto es importante, uso un método explícito (ver más abajo), pero luego tengo para agregar un nuevo método cada vez que agrego una subclase. Es una compensación entre flexibilidad y tipo la seguridad. Afortunadamente, si tomo la decisión equivocada, puedo usar el Método de parametrización o Reemplace el parámetro con métodos explícitos para revertir la decisión.

Otra razón para desconfiar de `Class.forName` es que expone los nombres de las subclases a los clientes. Esta no es tan malo porque puede usar otras cadenas y llevar a cabo otro comportamiento con la fábrica método. Esa es una buena razón para no usar el método en línea para eliminar la fábrica.

Ejemplo: crear subclases con métodos explícitos

Página 249

Puedo usar un enfoque diferente para ocultar subclases con métodos explícitos. Esto es útil si tienes solo unas pocas subclases que no cambian. Entonces puedo tener una clase de persona abstracta con subclases para hombres y mujeres. Comienzo definiendo un método de fábrica para cada subclase en la superclase:

```
Persona de clase ...
   Persona estática createMale () {
        return new Male ();
   }
   Persona estática createFemale () {
        volver nueva hembra ();
   }
```

Entonces puedo reemplazar las llamadas del formulario

```
Persona kent = nuevo hombre ();
```

con

```
Persona kent = Person.createMale ();
```

Esto deja a la superclase sabiendo sobre las subclases. Si quieres evitar esto, necesitas un esquema más complejo, como un comerciante de productos [Bäumer y Riehle]. La mayoría de las veces, sin embargo, esa complejidad no es necesaria, y este enfoque funciona muy bien.

Encapsular abatido

Un método devuelve un objeto que sus emisores deben descartar.

Mueva el downcast dentro del método.

```
Object lastReading () {
    return readings.lastElement ();
}
```

```
Leyendo lastReading () {
    return (Lectura) lecturas.lastElement ();
}
```

Motivación

Downcasting es una de las cosas más molestas que tienes que hacer con OO fuertemente tipado idiomas Es molesto porque se siente innecesario; le estás diciendo algo al compilador debería ser capaz de descubrirse a sí mismo. Pero debido a que descubrir a menudo es bastante complicado, usted a menudo tienes que hacerlo tú mismo. Esto es particularmente frecuente en Java, en el que la falta de plantillas significa que debe abatir cada vez que saca un objeto de una colección.

La decadencia puede ser un mal necesario, pero debes hacerlo lo menos posible. Si devuelves un valor de un método, y usted sabe que el tipo de lo que se devuelve es más especializado que lo que la firma del método dice que está poniendo trabajo innecesario en sus clientes. En lugar de forzar para que hagan el downcasting, siempre debe proporcionarles el tipo más específico que pueda.

A menudo encuentra esta situación con métodos que devuelven un iterador o una colección. Mira en su lugar para ver

para qué utiliza la gente el iterador y proporciona el método para ello.

Mecánica

- Busque casos en los que tenga que rechazar el resultado de llamar a un método.

  ? rarr; *Estos casos a menudo aparecen con métodos que devuelven una colección o iterador*

- Mueva el downcast al método.

  ? rarr; Con métodos que devuelven colecciones, use Encapsulate Colección .

Ejemplo

Tengo un método llamado `lastReading`, que devuelve la última lectura de un vector de lecturas:

```
Object lastReading () {
   return readings.lastElement ();
}
```

Debería reemplazar esto con

```
Leyendo lastReading () {
    return (Lectura) lecturas.lastElement ();
}
```

Un buen comienzo para hacer esto es donde tengo clases de colección. Digamos que esta colección de lecturas es en una clase de sitio y veo un código como este:

```
Lectura lastReading = (Lectura) theSite.readings (). LastElement ()
```

Puedo evitar el abatido y ocultar con qué colección se está utilizando

```
Lectura lastReading = theSite.lastReading ();

Sitio de clase ...
   Leyendo lastReading () {
        return (Lectura) lecturas (). lastElement ();
   }
```


La alteración de un método para devolver una subclase altera la firma del método pero no interrumpe código existente porque el compilador sabe que puede sustituir una subclase por la superclase. De Por supuesto, debe asegurarse de que la subclase no haga nada que rompa el contrato del superclase

Reemplazar código de error con excepción

Un método devuelve un código especial para indicar un error.

Lanza una excepción en su lugar.

```
 int retiro (int cantidad) {
      if (cantidad> _balance)
           volver -1;
      más {
           _balance - = cantidad;
           devuelve 0;
      }
 }
```

```
nulo retiro (cantidad int) arroja BalanceException {
      if (cantidad> _balance) arroja una nueva BalanceException ();
      _balance - = cantidad;
 }
```

Motivación

En las computadoras, como en la vida, las cosas salen mal ocasionalmente. Cuando las cosas van mal, debes hacer algo al respecto. En el caso más simple, puede detener el programa con un código de error. Este es el software equivalente a suicidarse porque pierde un vuelo. (Si hiciera eso no estaría vivo incluso si fuera un gato.) A pesar de mi intento simplista de humor, el suicidio del software tiene mérito opción. Si el costo de un bloqueo del programa es pequeño y el usuario es tolerante, detener el programa es multa. Sin embargo, los programas más importantes necesitan medidas más importantes.

El problema es que la parte de un programa que detecta un error no siempre es la parte que puede calcular qué hacer al respecto. Cuando una rutina de este tipo encuentra un error, debe avisarle a su interlocutor y la persona que llama puede pasar el error por la cadena. En muchos idiomas se usa una salida especial para indicar error. Los sistemas basados en Unix y C utilizan tradicionalmente un código de retorno para indicar el éxito o el fracaso de un rutina.

Java tiene una mejor manera: excepciones. Las excepciones son mejores porque separan claramente lo normal procesamiento de procesamiento de error. Esto hace que los programas sean más fáciles de entender, y como espero ahora cree, la comprensibilidad está al lado de la piedad.

Mecánica

• Decidir si la excepción se debe marcar o no.

? rarr; *Si la persona que llama es responsable de probar la condición antes de llamar, haga la excepción sin marcar.*

? rarr; *Si la excepción está marcada, cree una nueva excepción o use uno existente.*

• Busque todas las personas que llaman y ajústelas para usar la excepción.

? rarr; *Si la excepción no está marcada, ajuste las llamadas para que Verificación adecuada antes de llamar al método. Compila y prueba después de cada Tal cambio.*

? rarr; *Si se marca la excepción, ajuste las llamadas para llamar al método en Un bloque de prueba.*

- Cambie la firma del método para reflejar el nuevo uso.

Si tiene muchas personas que llaman, esto puede ser un cambio demasiado grande. Puedes hacerlo más gradual con el siguientes pasos:

- Decidir si la excepción se debe marcar o no.
- Cree un nuevo método que use la excepción.
- Modifique el cuerpo del método anterior para llamar al método nuevo.
- Compilar y probar.
- Ajuste cada llamada del método anterior para llamar al método nuevo. Compila y prueba después de cada cambio.
- Eliminar el método anterior.

Ejemplo

¿No es extraño que los libros de texto de computadora a menudo asuman que no puedes retirar más de tu saldo? de una cuenta, aunque en la vida real a menudo puedes?

```
Cuenta de clase ...
   int retiro (int cantidad) {
        if (cantidad> _balance)
             volver -1;
        más {
             _balance - = cantidad;
             devuelve 0;
        }
   }
   privado int _balance;
```

Para cambiar este código para usar una excepción, primero necesito decidir si usar un marcado o excepción no verificada La decisión depende de si es responsabilidad de la persona que realiza la prueba el saldo antes de retirar o si es responsabilidad de la rutina de retiro hacer el cheque Si probar la balanza es responsabilidad de la persona que llama, es un error de programación llamar retirar con una cantidad mayor que el saldo. Porque es un error de programación, es decir, un error: debo usar una excepción no marcada. Si probar el saldo es la rutina de retiro responsabilidad, debo declarar la excepción en la interfaz. De esa manera señalo a la persona que llama que espere la excepción y tomar las medidas apropiadas.

Ejemplo: excepción no verificada

Tomemos el caso sin verificar primero. Aquí espero que la persona que llama haga la verificación. Primero miro el personas que llaman En este caso, nadie debería usar el código de retorno porque es un error del programador hazlo Si veo código como

```
if (account.withdraw (cantidad) == -1)
     handleOverdrawn ();
más doTheUsualThing ();
```

Necesito reemplazarlo con código como

```
if (! account.canWithdraw (cantidad))
     handleOverdrawn ();
más {
     cuenta.traer (cantidad);
     doTheUsualThing ();
}
```

Puedo compilar y probar después de cada cambio.

Ahora necesito eliminar el código de error y lanzar una excepción para el caso de error. Porque el
el comportamiento es (por definición) excepcional, debería usar una cláusula de protección para la verificación de condición:

```
nulo retiro (cantidad int) {
     if (cantidad> _balance)
          lanzar nueva IllegalArgumentException ("Cantidad demasiado grande");
     _balance - = cantidad;
}
```

Debido a que es un error del programador, debo señalar aún más claramente usando una aserción:

```
Cuenta de clase ...
   nulo retiro (cantidad int) {
        Assert.isTrue ("cantidad demasiado grande", cantidad> _balance);
        _balance - = cantidad;
   }

clase Assert ...
   static void isTrue (comentario de cadena, prueba booleana) {
        if (! test) {
             lanzar nueva RuntimeException ("Error en la aserción:" + comentario);
        }
   }
```

Ejemplo: Excepción marcada

Manejo el caso de excepción verificado de manera ligeramente diferente. Primero creo (o uso) un nuevo apropiado
excepción:

```
La clase BalanceException extiende la excepción {}
```


Luego ajusto las llamadas para que se vean como

```
tratar {
     cuenta.traer (cantidad);
     doTheUsualThing ();
} catch (BalanceException e) {
     handleOverdrawn ();
}
```

Ahora cambio el método de retiro para usar la excepción:

```
    nulo retiro (cantidad int) arroja BalanceException {
         if (cantidad> _balance) arroja una nueva BalanceException ();
         _balance - = cantidad;
    }
```

Lo incómodo de este procedimiento es que tengo que cambiar todas las llamadas y las llamadas
rutina de una vez. De lo contrario, el compilador nos pega. Si hay muchas personas que llaman, esto también puede ser
grande un cambio sin el paso de compilación y prueba.

Para estos casos, puedo usar un método intermedio temporal. Comienzo con el mismo caso que
antes de:

```
  if (account.withdraw (cantidad) == -1)
       handleOverdrawn ();
  más doTheUsualThing ();

Cuenta de clase ...
  int retiro (int cantidad) {
       if (cantidad> _balance)
            volver -1;
       más {
            _balance - = cantidad;
            devuelve 0;
       }
   }
```

El primer paso es crear un nuevo método de retiro que use la excepción:

```
    void newWithdraw (cantidad int) arroja BalanceException {
         if (cantidad> _balance) arroja una nueva BalanceException ();
         _balance - = cantidad;
    }
```

A continuación, ajusto el método de extracción actual para usar el nuevo:

```
    int retiro (int cantidad) {
         tratar {
              newWithdraw (cantidad);
```



```
              devuelve 0;
         } catch (BalanceException e) {
              volver -1;
         }
    }
```

Una vez hecho esto, puedo compilar y probar. Ahora puedo reemplazar cada una de las llamadas al método anterior con
una llamada al nuevo:

```
         tratar {
              account.newWithdraw (cantidad);
```

```
            doTheUsualThing ();
        } catch (BalanceException e) {
            handleOverdrawn ();
        }
```

Con los métodos antiguos y nuevos en su lugar, puedo compilar y probar después de cada cambio. Cuando estoy terminado, puedo eliminar el método anterior y usar Rename Method para dar al nuevo método el antiguo nombre.

Reemplazar excepción con prueba

Está lanzando una excepción marcada en una condición que la persona que llamó podría haber verificado primero.

Cambie la persona que llama para hacer la prueba primero.

```
 double getValueForPeriod (int periodNumber) {
      tratar {
           return _values [periodNumber];
      } catch (ArrayIndexOutOfBoundsException e) {
           devuelve 0;
      }
 }
```

```
double getValueForPeriod (int periodNumber) {
      if (periodNumber> = _values.length) devuelve 0;
      return _values [periodNumber];
 }
```

Motivación

Las excepciones son un avance importante en los lenguajes de programación. Nos permiten evitar complejos códigos mediante el uso de Reemplazar código de error con excepción . Como tantos placeres, las excepciones pueden se usan en exceso y dejan de ser placenteros. (Incluso puedo cansarme de Aventinus [Jackson]).
Las excepciones deben usarse para un comportamiento excepcional, comportamiento que es un error inesperado. Ellos No debe actuar como un sustituto de las pruebas condicionales. Si puede esperar razonablemente que la persona que llama verifique la condición antes de llamar a la operación, debe proporcionar una prueba y la persona que llama debe úsalo.

Mecánica

- Ponga una prueba por adelantado y copie el código del bloque de captura en el tramo apropiado del si declaración.
- Agregue una aserción al bloque catch para notificarle si se ejecuta el bloque catch.
- Compilar y probar.
- Retire el bloque de captura y el bloque de prueba si no hay otros bloques de captura.
- Compilar y probar.

Ejemplo

Para este ejemplo, uso un objeto que gestiona recursos que son caros de crear pero que pueden ser

reutilizado Las conexiones a bases de datos son un buen ejemplo de esto. Tal gerente tiene dos grupos de recursos, uno que está disponible para su uso y otro que está asignado. Cuando un cliente quiere un recurso, el grupo lo entrega y lo transfiere del grupo disponible al grupo asignado. Cuando un cliente libera un recurso, el administrador lo devuelve. Si un cliente solicita un recurso y ninguno es disponible, el gerente crea uno nuevo.

El método para distribuir recursos podría verse así:

```
clase ResourcePool
   Recurso getResource () {
        Resultado del recurso;
        tratar {
             resultado = (Recurso) _available.pop ();
             _allocated.push (resultado);
             resultado de retorno;
        } catch (EmptyStackException e) {
             resultado = nuevo recurso ();
             _allocated.push (resultado);
             resultado de retorno;
        }
}
Pila _disponible;
Pila _ asignada;
```

En este caso, quedarse sin recursos no es una ocurrencia inesperada, por lo que no debería usar un excepción.

Para eliminar la excepción, primero agrego una prueba inicial apropiada y hago el comportamiento vacío allí:

```
   Recurso getResource () {
        Resultado del recurso;
        if (_available.isEmpty ()) {
            resultado = nuevo recurso ();
            _allocated.push (resultado);
            resultado de retorno;
       }
        más {
             tratar {
                 resultado = (Recurso) _available.pop ();
                 _allocated.push (resultado);
```


Página 257

```
                 resultado de retorno;
             } catch (EmptyStackException e) {
                 resultado = nuevo recurso ();
                 _allocated.push (resultado);
                 resultado de retorno;
             }
        }
    }
```

Con esto, la excepción nunca debería ocurrir. Puedo agregar una afirmación para verificar esto:

```
   Recurso getResource () {
        Resultado del recurso;
```

```
        if (_available.isEmpty ()) {
             resultado = nuevo recurso ();
             _allocated.push (resultado);
             resultado de retorno;
        }
        más {
             tratar {
                  resultado = (Recurso) _available.pop ();
                  _allocated.push (resultado);
                  resultado de retorno;
             } catch (EmptyStackException e) {
                Assert.shouldNeverReachHere ("disponible estaba vacío en pop");
                 resultado = nuevo recurso ();
                 _allocated.push (resultado);
                 resultado de retorno;
             }
        }
   }

 clase Assert ...
   static void shouldNeverReachHere (Mensaje de cadena) {
        lanzar nueva RuntimeException (mensaje);
   }
```

Ahora puedo compilar y probar. Si todo va bien, puedo eliminar completamente el bloque de prueba:

```
   Recurso getResource () {
        Resultado del recurso;
        if (_available.isEmpty ()) {
             resultado = nuevo recurso ();
             _allocated.push (resultado);
             resultado de retorno;
        }
        más {
             resultado = (Recurso) _available.pop ();
         _allocated.push (resultado);
         resultado de retorno;
      }
   }
```


Page 258

Después de esto, generalmente encuentro que puedo limpiar el código condicional. Aquí puedo usar Consolidar Fragmentos condicionales duplicados :

```
   Recurso getResource () {
        Resultado del recurso;
        if (_available.isEmpty ())
             resultado = nuevo recurso ();
        más
             resultado = (Recurso) _available.pop ();
        _allocated.push (resultado);
        resultado de retorno;
   }
```

Page 259

Capítulo 11. Manejo de la generalización

La generalización produce su propio lote de refactorizaciones, principalmente con métodos de movimiento alrededor de una jerarquía de herencia. Pull Up Field y Pull Up Method promueven la función una jerarquía, y el Método Push Down y la función Push Down Field push hacia abajo.

Los constructores son un poco más incómodos de levantar, por lo que Pull Up Constructor Body se ocupa de Esos problemas. En lugar de presionar a un constructor, a menudo es útil usar Reemplazar Constructor con Método de Fábrica .

Si tiene métodos que tienen un cuerpo de esquema similar pero varían en detalles, puede usar Formulario Método de plantilla para separar las diferencias de las similitudes.

Además de mover la función alrededor de una jerarquía, puede cambiar la jerarquía creando nuevos clases Extract Subclass , Extract Superclass y Extract Interface hacen esto formando

Nuevos elementos de varios puntos. La interfaz de extracción es particularmente importante cuando quieres marque una pequeña cantidad de función para el sistema de tipos. Si te encuentras con clases innecesarias en su jerarquía, puede usar Contraer Jerarquía para eliminarlos.

A veces descubres que la herencia no es la mejor manera de manejar una situación y que necesitas delegación en su lugar. Reemplazar herencia con delegación ayuda a realizar este cambio. A veces la vida es al revés y tienes que usar Reemplazar delegación con herencia .

Pull Up Field

Dos subclases tienen el mismo campo.

Mueve el campo a la superclase.

Motivación

Si las subclases se desarrollan de forma independiente o se combinan mediante la refactorización, a menudo encontrará que duplican características. En particular, ciertos campos pueden ser duplicados. Tales campos a veces tienen nombres similares pero no siempre. La única forma de determinar qué está pasando es mirar los campos. y ver cómo son utilizados por otros métodos. Si se utilizan de manera similar, puede generalizarlos.

Hacer esto reduce la duplicación de dos maneras. Elimina la declaración de datos duplicados y permite para pasar de las subclases al comportamiento de superclase que usa el campo.

Page 260

Mecánica

- Inspeccione todos los usos de los campos candidatos para asegurarse de que se usan de la misma manera.
- Si los campos no tienen el mismo nombre, cambie el nombre de los campos para que tengan el nombre desea usar para el campo de superclase.
- Compilar y probar.
- Crear un nuevo campo en la superclase.

> ? rarr; *Si los campos son privados, deberá proteger el campo de superclase para que las subclases puedan referirse a ella.*

- Eliminar los campos de subclase.
- Compilar y probar.
- Considere utilizar el campo de autoencapsulación en el nuevo campo.

Método Pull Up

Tiene métodos con resultados idénticos en subclases.

Muévelos a la superclase.

Motivación

Eliminar el comportamiento duplicado es importante. Aunque dos métodos duplicados funcionan bien ya que son, no son más que un caldo de cultivo para los insectos en el futuro. Cuando hay duplicación, usted corre el riesgo de que no se realice una modificación en uno. Por lo general es Difícil de encontrar los duplicados.

El caso más fácil de usar el *Método Pull Up* ocurre cuando los métodos tienen el mismo cuerpo, lo que implica que ha habido una copia y pega. Por supuesto, no siempre es tan obvio como eso. Tú podrías simplemente realice la refactorización y vea si las pruebas no funcionan bien, pero eso confía mucho en sus pruebas. yo generalmente les resulta valioso buscar las diferencias; a menudo muestran un comportamiento que olvidé probar para.

A menudo, el *Método Pull Up* viene después de otros pasos. Ves dos métodos en diferentes clases que pueden ser parametrizados de tal manera que terminen siendo esencialmente el mismo método. En ese caso el paso más pequeño es parametrizar cada método por separado y luego generalizarlos. Hazlo en una vez si te sientes lo suficientemente seguro.

Un caso especial de la necesidad del *Método Pull Up* ocurre cuando tiene un método de subclase que anula un método de superclase pero hace lo mismo.

El elemento más incómodo del *Método Pull Up* es que el cuerpo de los métodos puede referirse a características que están en la subclase pero no en la superclase. Si la función es un método, puede generalice el otro método o cree un método abstracto en la superclase. Tu puedes necesitar para cambiar la firma de un método o crear un método de delegación para que esto funcione.

Si tiene dos métodos que son similares pero no iguales, puede usar Formulario Método de plantilla .

Mecánica

- Inspeccione los métodos para asegurarse de que sean idénticos.

  ? rarr; *Si parece que los métodos hacen lo mismo pero no son idéntico, use la sustitución del algoritmo en uno de ellos para hacerlos idéntico.*

• Si los métodos tienen firmas diferentes, cambie las firmas a la que desea
uso en la superclase.
• Cree un nuevo método en la superclase, copie el cuerpo de uno de los métodos, ajuste,
y compilar.

? rarr; *Si está en un idioma fuertemente tipado y el método llama otro método que está presente en ambas subclases pero no en el superclase, declare un método abstracto en la superclase.*

? rarr; *Si el método usa un campo de subclase, use* Pull Up Field *o* Self Encapsular campo *y declarar y usar un método de obtención abstracto.*

• Eliminar un método de subclase.
• Compilar y probar.
• Continúe eliminando métodos y pruebas de subclase hasta que solo quede el método de superclase.
• Eche un vistazo a las personas que llaman de este método para ver si puede cambiar un tipo requerido
a la superclase

Ejemplo

Considere un cliente con dos subclases: cliente habitual y cliente preferido.

Página 262

El método `createBill` es idéntico para cada clase:

```
anular createBill (fecha Fecha) {
   double chargeAmount = charge (lastBillDate, date);
   addBill (fecha, cargo);
}
```

No puedo mover el método a la superclase, porque `chargeFor` es diferente en cada subclase Primero tengo que declararlo en la superclase como abstracto:

```
clase Cliente ...
   Resumen doble carga Para (fecha de inicio, fecha de finalización)
```

Entonces puedo copiar `createBill` de una de las subclases. Compilo con eso en su lugar y luego elimine el método `createBill` de una de las subclases, compile y pruebe. Luego lo quito del otro, compilar y probar:

Página 263

Levante el cuerpo del constructor

Tienes constructores en subclases con cuerpos en su mayoría idénticos.

Crea un constructor de superclase; llame a esto desde los métodos de subclase.

```
Gerente de clase extiende Empleado ...
  Administrador público (nombre de cadena, ID de cadena, grado int) {
       _name = nombre;
       _id = id;
       _grade = grado;
  }
```

```
  Administrador público (nombre de cadena, ID de cadena, grado int) {
       super (nombre, id);
       _grade = grado;
  }
```

Motivación

263

Página 264

Los constructores son cosas difíciles. No son métodos bastante normales, por lo que está más restringido en qué puede hacer con ellos de lo que es cuando usa métodos normales.

Si ve métodos de subclase con comportamiento común, su primer pensamiento es extraer el común comportamiento en un método y subirlo a la superclase. Con un constructor, sin embargo, el Comportamiento común es a menudo la construcción. En este caso, necesita un constructor de superclase que sea llamado por subclases. En muchos casos, este es el cuerpo completo del constructor. No puedes usar Pull Arriba Método aquí, porque no puedes heredar constructores (¿no odias eso?).

Si la refactorización se vuelve compleja, es posible que desee considerar Reemplazar constructor con Método de fábrica en su lugar.

Mecánica

- Definir un constructor de superclase.
- Mueva el código común al principio de la subclase al constructor de la superclase.

? rarr; *Este puede ser todo el código.*

? rarr; *Intenta mover el código común al comienzo del constructor.*

• Llame al constructor de la superclase como el primer paso en el constructor de la subclase.

? rarr; *Si todo el código es común, esta será la única línea de la subclase constructor.*

• Compilar y probar.

? rarr; *Si hay algún código común más tarde, use el* método de extracción *para factorizar extraiga el código común y use el* Método Pull Up para *extraerlo.*

Ejemplo

Aquí hay un gerente y un empleado:

```
Empleado de clase ...
   String _name protegido;
   String protegido _id;

Gerente de clase extiende Empleado ...
   Administrador público (nombre de cadena, ID de cadena, grado int) {
        _name = nombre;
        _id = id;
        _grade = grado;
   }

   privado int _grade;
```


Página 265

Los campos de `Empleado` deben establecerse en un constructor para `Empleado`. Yo defino uno y lo hago protegido para indicar que las subclases deberían llamarlo:

```
Empleado de clase
   Empleado protegido (nombre de cadena, ID de cadena) {
        _name = nombre;
        _id = id;
   }
```

Entonces lo llamo de la subclase:

```
   Administrador público (nombre de cadena, ID de cadena, grado int) {
        super (nombre, id);
        _grade = grado;
   }
```

Una variación ocurre cuando hay un código común más tarde. Digamos que tengo el siguiente código:

```
Empleado de clase ...
  boolean isPriviliged () {..}
  void asignarCar () {..}
Gerente de clase ...
  Administrador público (nombre de cadena, ID de cadena, grado int) {
       super (nombre, id);
       _grade = grado;
       if (isPriviliged ()) asignarCar (); // cada subclase hace esto
  }
  boolean isPriviliged () {
       return _grade> 4;
  }
```

No puedo mover el comportamiento `asignarCar` al constructor de superclase porque debe ser ejecutado después de que la calificación haya sido asignada al campo. Así que necesito el método de extracción y pull up Método .

```
Empleado de clase ...
  vacío inicializar () {
       if (isPriviliged ()) asignarCar ();
  }
Gerente de clase ...
  Administrador público (nombre de cadena, ID de cadena, grado int) {
       super (nombre, id);
       _grade = grado;
       inicializar();
  }
```


Page 266

Método de empuje

El comportamiento en una superclase es relevante solo para algunas de sus subclases.

Muévelo a esas subclases.

Motivación

El método Pull Down es lo opuesto al método Pull Up . Lo uso cuando necesito mover el comportamiento de una superclase a una subclase específica, generalmente porque solo tiene sentido allí. A menudo lo haces esto cuando usas Extract Subclass .

Mecánica

• Declare un método en todas las subclases y copie el cuerpo en cada subclase.

> ? rarr; *Es posible que deba declarar los campos como protegidos para que el método acceder a ellos Por lo general, hace esto si tiene la intención de empujar hacia abajo el campo más tarde. De lo contrario, use un descriptor de acceso en la superclase. Si este descriptor de acceso es no público, debe declararlo como protegido.*

• Eliminar el método de la superclase.

> ? rarr; *Puede que tenga que cambiar las llamadas para usar la subclase en variable y declaraciones de parámetros.*

> ? rarr; *Si tiene sentido acceder al método a través de una superclase variable, no tiene la intención de eliminar el método de ninguna subclase, y la superclase es abstracta, puedes declarar el método como abstracto, en la superclase*

• Compilar y probar.
• Elimine el método de cada subclase que no lo necesita.
• Compilar y probar.

Campo de empuje

Un campo solo lo usan algunas subclases.

Página 267

Mueva el campo a esas subclases.

Motivación

Push Down Field es lo opuesto a Pull Up Field . Úselo cuando no necesite un campo en el superclase pero solo en una subclase.

Mecánica

• Declarar el campo en todas las subclases.
• Eliminar el campo de la superclase.
• Compilar y probar.
• Elimine el campo de todas las subclases que no lo necesitan.

• Compilar y probar.

Extraer subclase

Una clase tiene características que se usan solo en algunos casos.

Cree una subclase para ese subconjunto de características.

Motivación

Página 268

El principal desencadenante para el uso de la *subclase Extract* es darse cuenta de que una clase tiene un comportamiento utilizado algunas instancias de la clase y no para otras. A veces esto se indica mediante un código de tipo, en en cuyo caso puede usar Reemplazar código de tipo con subclases o Reemplazar código de tipo con Estado / Estrategia . Pero no es necesario tener un código de tipo para sugerir el uso de una subclase.

La principal alternativa a la *subclase* Extract es Extract Class . Esta es una elección entre delegación y herencia. Extraer subclase suele ser más sencillo de hacer, pero tiene limitaciones. No puedes cambiar el comportamiento basado en clases de un objeto una vez que se crea el objeto. Puedes cambiar el comportamiento basado en clases con Extract Class simplemente conectando diferentes componentes. Usted puede También use solo subclases para representar un conjunto de variaciones. Si quieres que la clase varíe De varias maneras diferentes, debe usar la delegación para todos menos uno de ellos.

Mecánica

• Definir una nueva subclase de la clase fuente.
• Proporcionar constructores para la nueva subclase.

? rarr; *En los casos simples, copie los argumentos de la superclase y llame el constructor de superclase con* `super` .

? rarr; *Si desea ocultar el uso de la subclase de los clientes, puede use* Reemplazar constructor con método de fábrica .

• Encuentra todas las llamadas a constructores de la superclase. Si necesitan la subclase, reemplace con un Llamada al nuevo constructor.

? rarr; *Si el constructor de la subclase necesita argumentos diferentes, use* Cambiar el nombre del método *para cambiarlo. Si algunos de los parámetros del constructor de la superclase ya no es necesaria, use el* método Rename también *en eso.*

? rarr; *Si la superclase ya no puede ser instanciada directamente, declararla resumen.*

• Una a una utilización empuje hacia abajo Método y empuje hacia abajo Campo de moverse características en el subclase

? rarr; *A diferencia de* Extract Class , *generalmente es más fácil trabajar con los métodos primero y los datos al final.*

? rarr; *Cuando se empuja un método público, es posible que deba redefinir un variable de la persona que llama o tipo de parámetro para llamar al nuevo método. El compilador atrapará estos casos.*

• Busque cualquier campo que designe información ahora indicada por la jerarquía (generalmente un código booleano o de tipo). Elimínelo usando el campo de autoencapsulación y reemplazando el getter con métodos constantes polimórficos. Todos los usuarios de este campo deben ser refactorizados con Reemplazar condicional con polimorfismo .

? rarr; *Para cualquier método fuera de la clase que use un descriptor de acceso, considere usando Move Method para mover el método a esta clase; luego usa* Reemplazar condicional con polimorfismo .

Página 269

• Compile y pruebe después de cada empuje hacia abajo.

Ejemplo

Comenzaré con una clase de artículos de trabajo que determina los precios de los artículos de trabajo en un garaje local:

```
   clase JobItem ...
      Public JobItem (int unitPrice, int cantidad, boolean isLabor,
Empleado empleado) {
           _unitPrice = unitPrice;
           _cantidad = cantidad;
           _isLabor = isLabor;
           _empleado = empleado;
      }
      public int getTotalPrice () {
           return getUnitPrice () * _quantity;
      }
      public int getUnitPrice () {
           volver (_isLabor)?
                 _employee.getRate ():
                 _precio unitario;
      }
      public int getQuantity () {
           retorno _cantidad;
      }
      empleado público getEmployee () {
           volver _empleado;
```

```
    }
    privado int _unitPrice;
    int _cantidad privada;
    Empleado privado _empleado;
    _isLabor privado booleano;

  Empleado de clase ...
    Empleado público (tasa int) {
         _rate = rate;
    }
    public int getRate () {
         tasa de retorno;
    }
    int intrate privado;
```

`Extraigo` una subclase `LaborItem` de esta clase porque parte del comportamiento y los datos son solo se necesita en ese caso. Comienzo creando la nueva clase:

```
  clase LaborItem extiende JobItem {}
```

Lo primero que necesito es un constructor para el elemento de trabajo porque el elemento de trabajo no tiene ningún argumento constructor. Para esto copio la firma del constructor padre:

Page 270

```
    Public LaborItem (int unitPrice, int cantidad, boolean isLabor,
Empleado empleado) {
         super (unitPrice, cantidad, isLabor, empleado);
    }
```

Esto es suficiente para que se compile la nueva subclase. Sin embargo, el constructor es desordenado; algunos el elemento laboral necesita argumentos, y algunos no. Me encargo de eso más tarde.

El siguiente paso es buscar llamadas al constructor del elemento de trabajo y buscar casos en los que el constructor del elemento laboral debería llamarse en su lugar. Entonces declaraciones como

```
         JobItem j1 = nuevo JobItem (0, 5, verdadero, kent);
```

volverse

```
         JobItem j1 = new LaborItem (0, 5, true, kent);
```

En esta etapa no he cambiado el tipo de la variable; Solo he cambiado el tipo de constructor. Esto se debe a que quiero usar el nuevo tipo solo donde tengo que hacerlo. En este punto tengo no hay una interfaz específica para la subclase, por lo que aún no quiero declarar ninguna variación.

Ahora es un buen momento para limpiar las listas de parámetros del constructor. Yo uso Rename Method en cada de ellos. Yo trabajo con la superclase primero. Creo el nuevo constructor y hago el viejo protegido (la subclase todavía lo necesita):

```
   clase JobItem ...
     JobItem protegido (int unitPrice, int cantidad, boolean isLabor,
Empleado empleado) {
          _unitPrice = unitPrice;
          _cantidad = cantidad;
          _isLabor = isLabor;
          _empleado = empleado;
     }
     Public JobItem (int unitPrice, int cantidad) {
          esto (precio unitario, cantidad, falso, nulo)
     }
```

Las llamadas desde afuera ahora usan el nuevo constructor:

```
          JobItem j2 = nuevo JobItem (10, 15);
```


Página 271

Una vez que he compilado y probado, uso el método Rename en el constructor de subclase:

```
   clase LaborItem
     Public LaborItem (int cantidad, empleado empleado) {
          super (0, cantidad, verdadero, empleado);
     }
```

Por el momento, todavía uso el constructor de superclase protegido.

Ahora puedo comenzar a presionar las características del elemento de trabajo. Comienzo con los métodos. Empiezo con usando el Método Push Down en `getEmployee`:

```
   clase LaborItem ...
     empleado público getEmployee () {
          volver _empleado;
     }
   clase JobItem ...
     Empleado protegido _empleado;
```

Debido a que el campo del empleado se eliminará más tarde, lo declaro protegido por el momento.

Una vez que el campo del empleado está protegido, puedo limpiar los constructores para que el empleado solo esté configurado

en la subclase en la que se está empujando hacia abajo:

```
clase JobItem ...
   JobItem protegido (int unitPrice, int cantidad, boolean isLabor) {
        _unitPrice = unitPrice;
        _cantidad = cantidad;
        _isLabor = isLabor;
   }
clase LaborItem ...
   Public LaborItem (int cantidad, empleado empleado) {
        super (0, cantidad, verdadero);
         _empleado = empleado;
   }
```

El campo `_isLabor` se usa para indicar información que ahora es inherente a la jerarquía. Así que puedo eliminar el campo La mejor manera de hacer esto es usar primero el campo de autoencapsulación y luego cambiar el descriptor de acceso utiliza un método constante polimórfico. Un método constante polimórfico es uno por el cual cada implementación devuelve un valor fijo (diferente):

```
clase JobItem ...
```

271

Página 272

```
   isLabor booleano protegido () {
        falso retorno;
   }
clase LaborItem ...
   isLabor booleano protegido () {
        volver verdadero;
   }
```

Entonces puedo deshacerme del campo `isLabor` .

Ahora puedo mirar a los usuarios de los métodos `isLabor` . Estos deben ser refactorizados con Reemplazar Condicional con polimorfismo . Tomo el metodo

```
clase JobItem ...
   public int getUnitPrice () {
        return (isLabor ())?
              _employee.getRate ():
              _precio unitario;
   }
```

y reemplazarlo con

```
clase JobItem ...
   public int getUnitPrice () {
        return _unitPrice;
   }
```

```
clase LaborItem ...
  public int getUnitPrice () {
       return _employee.getRate ();
  }
```

Una vez que un grupo de métodos que usan algunos datos han sido empujados hacia abajo, puedo usar Push Down Campo en los datos. Que no puedo usarlo porque un método usa los datos es una señal para trabajar más en los métodos, ya sea con el método Push Down o Reemplazar condicional con polimorfismo .

Debido a que el precio unitario es utilizado solo por artículos que no son de trabajo (artículos de piezas de trabajo), puedo usar Extraer Subclase en artículo de trabajo nuevamente para crear una clase de artículo de piezas. Cuando he hecho eso, la clase de elemento de trabajo Será abstracto.

Extraer Superclase

Tienes dos clases con características similares.

Cree una superclase y mueva las características comunes a la superclase.

Página 273

Motivación

El código duplicado es una de las principales cosas malas en los sistemas. Si dices cosas en varios lugares, entonces cuando llega el momento de cambiar lo que dices, tienes más cosas que cambiar que tú debería.

Una forma de código duplicado son dos clases que hacen cosas similares de la misma manera o cosas similares En maneras diferentes. Los objetos proporcionan un mecanismo incorporado para simplificar esta situación con la herencia. Sin embargo, a menudo no notas los puntos en común hasta que hayas creado algunas clases, en las que en caso de que necesite crear la estructura de herencia más tarde.

Una alternativa es la clase de extracto . La elección es esencialmente entre herencia y delegación. La herencia es la opción más simple si las dos clases comparten la interfaz y el comportamiento. Si tu tome la decisión incorrecta, siempre puede usar Reemplazar herencia con delegación más adelante.

Mecánica

- Crear una superclase abstracta en blanco; hacer las clases originales subclases de este superclase
- Una a una, utilizar Pull Up campo , Pull Up Método y Pull Up Constructor del cuerpo de mover elementos comunes a la superclase.

  ? rarr; *Por lo general, es más fácil mover los campos primero.*

  ? rarr; *Si tiene métodos de subclase que tienen firmas diferentes pero el mismo propósito, use el* método Rename *para obtener el mismo nombre y luego use el* Método Pull Up .

  ? rarr; *Si tiene métodos con la misma firma pero diferentes cuerpos, declarar la firma común como método abstracto en la superclase.*

  ? rarr; *Si tiene métodos con diferentes cuerpos que hacen lo mismo, puede intentar usar* Algoritmo sustituto *para copiar un cuerpo en el otro. Si esto funciona, puede usar el* Método Pull Up .

- Compilar y probar después de cada tirón.

Página 274

- Examinar los métodos que quedan en las subclases. Vea si hay partes comunes, si hay puede utilizar el método Extracto seguido de Pull Up Método en las partes comunes. Si el el flujo general es similar, puede usar el Método de plantilla de formulario .
- Después de extraer todos los elementos comunes, verifique cada cliente de las subclases. Si usan solo la interfaz común puede cambiar el tipo requerido a la superclase.

Ejemplo

Para este caso tengo un empleado y un departamento:

```
Empleado de clase ...
  Empleado público (String name, String id, int annualCost) {
       _name = nombre;
       _id = id;
       _annualCost = annualCost;
  }
  public int getAnnualCost () {
       return _annualCost;
  }
  public String getId () {
       return _id;
  }
  public String getName () {
       return _name;
  }
  cadena privada _name;
  privado int _annualCost;
  string privada _id;

Departamento de clase pública ...
  Departamento público (nombre de cadena) {
       _name = nombre;
  }
```

```
public int getTotalAnnualCost () {
     Enumeración e = getStaff ();
     int resultado = 0;
     while (e.hasMoreElements ()) {
          Empleado cada = (Empleado) e.nextElement ();
          resultado + = each.getAnnualCost ();
     }
     resultado de retorno;
}
public int getHeadCount () {
      return _staff.size ();
}
Enumeración pública getStaff () {
     return _staff.elements ();
}
public void addStaff (Employee arg) {
     _staff.addElement (arg);
}
public String getName () {
     return _name;
}
```


Página 275

```
cadena privada _name;
Private Vector _staff = nuevo Vector ();
```

Aquí hay un par de áreas comunes. Primero, tanto los empleados como los departamentos tienen nombres Segundo, ambos tienen costos anuales, aunque los métodos para calcularlos son ligeramente diferente. Extraigo una superclase para ambas características. El primer paso es crear el nueva superclase y defina las superclases existentes para que sean subclases de esta superclase:

```
fiesta de clase abstracta {}
clase Empleado extiende Fiesta ...
Departamento de clase extiende Fiesta ...
```

Ahora empiezo a mostrar características de la superclase. Por lo general, es más fácil usar Pull Up Field primero:

```
Fiesta de clase ...
  String _name protegido;
```

Entonces puedo usar el Método Pull Up en los captadores:

```
fiesta de clase {

  public String getName () {
       return _name;
  }
```

Me gusta hacer que el campo sea privado. Para hacer esto, necesito usar Pull Up Constructor Body para asignar el nombre:

```
Fiesta de clase ...
   Parte protegida (nombre de cadena) {
        _name = nombre;
   }
   cadena privada _name;

Empleado de clase ...
   Empleado público (String name, String id, int annualCost) {
        super (nombre);
        _id = id;
        _annualCost = annualCost;
   }

Departamento de clase ...
   Departamento público (nombre de cadena) {
```


Página 276

```
        super (nombre);
   }
```

Los métodos `Department.getTotalAnnualCost` y `Employee.getAnnualCost`, hacen llevar a cabo la misma intención, por lo que deben tener el mismo nombre. Primero uso Rename Method para llévalos al mismo nombre:

```
El departamento de clase extiende la fiesta {
   public int getAnnualCost () {
        Enumeración e = getStaff ();
        int resultado = 0;
        while (e.hasMoreElements ()) {
             Empleado cada = (Empleado) e.nextElement ();
             resultado + = each.getAnnualCost ();
        }
        resultado de retorno;
   }
```

Sus cuerpos siguen siendo diferentes, por lo que no puedo usar el Método Pull Up ; sin embargo, puedo declarar un método abstracto en la superclase:

```
   abstract public int getAnnualCost ()
```

Una vez que he hecho los cambios obvios, miro a los clientes de las dos clases para ver si puedo cambiar cualquiera de ellos para usar la nueva superclase. Un cliente de estas clases es el departamento. clase en sí, que tiene una colección de empleados. El método `getAnnualCost` usa solo el método de costo anual, que ahora se declara en la fiesta:

```
Departamento de clase ...
  public int getAnnualCost () {
       Enumeración e = getStaff ();
       int resultado = 0;
       while (e.hasMoreElements ()) {
            Party each = (Party) e.nextElement ();
            resultado + = each.getAnnualCost ();
       }
       resultado de retorno;
  }
```

Este comportamiento sugiere una nueva posibilidad. Podría tratar al departamento y al empleado como un *compuesto* [Pandilla de cuatro]. Esto me permitiría dejar que un departamento incluya otro departamento. Esto sería Ser una nueva funcionalidad, por lo que no es estrictamente una refactorización. Si se quisiera un compuesto, lo obtendría cambiando el nombre del campo de personal para representar mejor la imagen. Este cambio implicaría

Página 277

haciendo un cambio correspondiente en el nombre `addStaff` y alterando el parámetro para ser parte. El cambio final sería hacer que el método `headCount sea` recursivo. Podría hacer esto creando un método de recuento para empleados que solo devuelve 1 y utiliza Algoritmo sustituto en el recuento del departamento para sumar los recuentos de los componentes.

Interfaz de extracción

Varios clientes usan el mismo subconjunto de la interfaz de una clase, o dos clases tienen parte de su Interfaces en común.

Extraiga el subconjunto en una interfaz.

Motivación

Las clases se usan entre sí de varias maneras. El uso de una clase a menudo significa abarcar toda el área de responsabilidades de una clase. Otro caso es el uso de solo un subconjunto particular de una clase responsabilidades de un grupo de clientes. Otra es que una clase necesita trabajar con cualquier clase que pueda manejar ciertas solicitudes.

Para los segundos dos casos, a menudo es útil hacer que el subconjunto de responsabilidades sea algo en sí mismo. correcto, para que quede claro en el uso del sistema. De esa manera es más fácil ver cómo Las responsabilidades se dividen. Si se necesitan nuevas clases para admitir el subconjunto, es más fácil ver exactamente lo que cabe en el subconjunto.

En muchos lenguajes orientados a objetos, esta capacidad es compatible con la herencia múltiple. Tú cree una clase para cada segmento de comportamiento y combínelos en una implementación. Java tiene herencia única, pero le permite establecer e implementar este tipo de requisitos utilizando interfaces. Las interfaces han tenido una gran influencia en la forma en que los programadores diseñan programas Java. Incluso ¡Los programadores de Smalltalk piensan que las interfaces son un paso adelante!

Página 278

Existe cierta similitud entre Extract Superclass y Extract Interface . Extraer La interfaz solo puede mostrar interfaces comunes, no códigos comunes. Usando la interfaz de extracción puede conducir a código duplicado maloliente. Puede reducir este problema utilizando Extract Class para poner el comportamiento en un componente y delegar en él. Si hay un comportamiento común sustancial Extraer Superclase es más simple, pero solo tienes una superclase.

Las interfaces son buenas para usar siempre que una clase tenga roles distintos en diferentes situaciones. Usar extracto Interfaz para cada rol. Otro caso útil es aquel en el que desea describir la salida interfaz de una clase, es decir, las operaciones que la clase realiza en su servidor. Si quieres permitir En otros tipos de servidores en el futuro, todo lo que necesitan hacer es implementar la interfaz.

Mecánica

- Crear una interfaz vacía.
- Declarar las operaciones comunes en la interfaz.
- Declarar las clases relevantes como implementando la interfaz.
- Ajuste las declaraciones de tipo de cliente para usar la interfaz.

Ejemplo

Una clase de hoja de tiempo genera cargos para los empleados. Para hacer esto, la hoja de tiempo necesita saber la tasa del empleado y si el empleado tiene una habilidad especial:

```
cargo doble (empleado empleado, días int) {
     int base = emp.getRate () * días;
     if (emp.hasSpecialSkill ())
          base de retorno * 1.05;
     otra base de retorno;
}
```

`El empleado` tiene muchos otros aspectos además de la tarifa y la información de habilidades especiales, pero esas son las únicas piezas que necesita esta aplicación. Puedo resaltar el hecho de que solo necesito esto subconjunto definiendo una interfaz para él:

```
interfaz facturable {
   public int getRate ();
   public boolean hasSpecialSkill ();
}
```

Luego declaro que el empleado implementa la interfaz:

```
empleado de clase implementa facturable ...
```


Página 279

Una vez hecho esto, puedo cambiar la declaración de cargo para mostrar solo esta parte del empleado se usa el comportamiento:

```
carga doble ( Emp . facturable , días int) {
     int base = emp.getRate () * días;
     if (emp.hasSpecialSkill ())
          base de retorno * 1.05;
     otra base de retorno;
}
```

Por el momento, la ganancia es una ganancia modesta en documentabilidad. Tal ganancia no valdría la pena para un método, pero si varias clases usaran la interfaz facturable en persona, eso sería útil. La gran ganancia aparece cuando quiero facturar a las computadoras también. Para hacerlos facturables, sé que todo lo que tengo que hacer es implementar la interfaz facturable y puedo poner las computadoras en hojas de horas.

Contraer Jerarquía

Una superclase y una subclase no son muy diferentes.

Fusionarlos juntos.

Motivación

Si ha estado trabajando durante un tiempo con una jerarquía de clases, puede enredarse fácilmente para
Su propio bien. Refactorizar la jerarquía a menudo implica empujar métodos y campos hacia arriba y hacia abajo
La jerarquía. Después de hacer esto, puede encontrar que tiene una subclase que no agrega ninguna
valor, por lo que debe fusionar las clases juntas.

Mecánica

- Elija qué clase se va a eliminar: la superclase o las subclases.

Page 280

- Uso Pull Up campo y Pull Up Método o Empuje hacia abajo Método y empuje hacia abajo Campo para mover todo el comportamiento y los datos de la clase eliminada a la clase con la que se está fusionado
- Compilar y probar con cada movimiento.
- Ajuste las referencias a la clase que se eliminará para usar la clase fusionada. Esta voluntad afectar declaraciones de variables, tipos de parámetros y constructores.
- Eliminar la clase vacía.
- Compilar y probar.

Método de plantilla de formulario

Tiene dos métodos en subclases que realizan pasos similares en el mismo orden, pero los pasos
son diferentes.

Obtenga los pasos de los métodos con la misma firma, para que los métodos originales se conviertan en mismo. Entonces puedes levantarlos.

Motivación

La herencia es una herramienta poderosa para eliminar el comportamiento duplicado. Cada vez que vemos dos similares métodos en una subclase, queremos reunirlos en una superclase. Pero, ¿y si no lo son? ¿exactamente lo mismo? ¿Qué hacemos entonces? Todavía necesitamos eliminar toda la duplicación que podamos pero mantener las diferencias esenciales

Un caso común son dos métodos que parecen llevar a cabo pasos ampliamente similares en el mismo secuencia, pero los pasos no son los mismos. En este caso podemos mover la secuencia al superclase y permitir que el polimorfismo desempeñe su papel para garantizar que los diferentes pasos hagan lo suyo diferentemente. Este tipo de método se llama *método de plantilla* [Gang of Four].

Mecánica

- Descomponga los métodos para que todos los métodos extraídos sean idénticos o completamente diferente.
- Utilice el método Pull Up para extraer los métodos idénticos en la superclase.
- Para los diferentes métodos, use Cambiar nombre de método para que las firmas de todos los métodos en Cada paso es el mismo.

  ? rarr; *Esto hace que los métodos originales sean los mismos, ya que todos emiten el mismo conjunto de llamadas a métodos, pero las subclases manejan las llamadas de manera diferente.*

- Compilar y probar después de cada cambio de firma.
- Utilice el Método Pull Up en uno de los métodos originales. Definir las firmas de diferentes métodos como métodos abstractos en la superclase.
- Compilar y probar.
- Elimine los otros métodos, compile y pruebe después de cada eliminación.

Ejemplo

Termino donde lo dejé en el Capítulo 1 . Tuve una clase de cliente con dos métodos para imprimir declaraciones. El método de `declaración` imprime declaraciones en ASCII:

```
declaración de cadena pública () {
    Enumeración alquileres = _rentals.elements ();
     String result = "Registro de alquiler para" + getName () + "\ n";
     while (rentals.hasMoreElements ()) {
          Alquiler de cada = (Alquiler) rentals.nextElement ();
          // muestra las cifras de este alquiler
```

```
            resultado + = "\ t" + each.getMovie (). getTitle () + "\ t" +
                String.valueOf (each.getCharge ()) + "\ n";
        }
        // agrega líneas de pie de página
        resultado + = "El monto adeudado es" + String.valueOf (getTotalCharge ())
+ "\ n";
        resultado + = "Ganaste" +
String.valueOf (getTotalFrequentRenterPoints ()) +
             "puntos frecuentes de arrendamiento";
        resultado de retorno;
    }
```


Página 282

mientras que `htmlStatement los` hace en HTML:

```
    public String htmlStatement () {
        Enumeración alquileres = _rentals.elements ();
        String result = "<H1> Alquileres para <EM>" + getName () +
"</EM> </H1> <P> \ n";
        while (rentals.hasMoreElements ()) {
            Alquiler de cada = (Alquiler) rentals.nextElement ();
            // muestra las cifras de cada alquiler
            resultado + = each.getMovie (). getTitle () + ":" +
                String.valueOf (each.getCharge ()) + "<BR> \ n";
        }
        // agrega líneas de pie de página
        resultado + = "<P> Debe <EM>" + String.valueOf (getTotalCharge ())
+ "</EM> <P> \ n";
        resultado + = "En este alquiler ganó <EM>" +
            String.valueOf (getTotalFrequentRenterPoints ()) +
            "</EM> puntos frecuentes de alquiler <P>";
        resultado de retorno;
    }
```

Antes de poder usar el Método de plantilla de formulario , necesito organizar las cosas de modo que los dos métodos sean subclases de alguna superclase común. Hago esto usando un objeto de método [Beck] para crear un jerarquía de estrategia separada para imprimir las declaraciones (Figura 11.1).

Figura 11.1. Usando una estrategia para declaraciones

```
Declaración de clase {}
class TextStatement extiende la declaración {}
class HtmlStatement extiende la declaración {}
```


Page 283

Ahora uso Move Method para mover los dos métodos de declaración a las subclases:

```
clase Cliente ...
declaración de cadena pública () {
     return new TextStatement (). value (this);
}
public String htmlStatement () {
     return new HtmlStatement (). value (this);
}
```

```
class TextStatement {
   Valor de cadena pública (Cliente a Cliente) {
           Enumeración alquileres = aCustomer.getRentals ();
           String result = "Registro de alquiler para" + aCustomer.getName () +
"\norte";
           while (rentals.hasMoreElements ()) {
                Alquiler de cada = (Alquiler) rentals.nextElement ();

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

           // agrega líneas de pie de página
           resultado + = "El monto adeudado es" +
String.valueOf (aCustomer.getTotalCharge ()) + "\ n";
           resultado + = "Ganaste" +
String.valueOf (aCustomer.getTotalFrequentRenterPoints ()) +
              "puntos frecuentes de arrendamiento";
           resultado de retorno;
     }
 clase HtmlStatement {
   Valor de cadena pública (Cliente a Cliente) {
           Enumeración alquileres = aCustomer.getRentals ();
           String result = "<H1> Alquileres para <EM>" + aCustomer.getName () +
"</EM> </H1> <P> \ n";
           while (rentals.hasMoreElements ()) {
                Alquiler de cada = (Alquiler) rentals.nextElement ();
                // muestra las cifras de cada alquiler
                resultado + = each.getMovie (). getTitle () + ":" +
                              String.valueOf (each.getCharge ()) + "<BR> \ n";
           }
           // agrega líneas de pie de página
```

```
                resultado + = "<P> Debe <EM>" +
String.valueOf (aCustomer.getTotalCharge ()) +
                  "</EM> <P> \ n";
           resultado + = "En este alquiler ganó <EM>"

String.valueOf (aCustomer.getTotalFrequentRenterPoints ()) +
                     "</EM> puntos frecuentes de alquiler <P>";
           resultado de retorno;
 }
```


Page 284

A medida que los movía, cambié el nombre de los métodos de declaración para adaptarlos mejor a la estrategia. Les di el mismo nombre porque la diferencia entre los dos ahora radica en la clase en lugar del método.
(Para aquellos que intentan esto con el ejemplo, también tuve que agregar un método `getRentals` al cliente y relaje la visibilidad de `getTotalCharge` y `getTotalFrequentRenterPoints`.

Con dos métodos similares en subclases, puedo comenzar a usar el Método de plantilla de formulario . La clave para esta refactorización consiste en separar el código variable del código similar mediante el uso del método de extracción para extraer las piezas que son diferentes entre los dos métodos. Cada vez que extraigo creo métodos con diferentes cuerpos pero la misma firma.

El primer ejemplo es la impresión del encabezado. Ambos métodos utilizan al cliente para obtener información, pero la cadena resultante tiene un formato diferente. Puedo extraer el formato de este cadena en métodos separados con la misma firma:

```
 class TextStatement ...
    String headerString (Cliente a Cliente) {
       devuelve "Registro de alquiler para" + aCustomer.getName () + "\ n";
    }
    Valor de cadena pública (Cliente a Cliente) {
           Enumeración alquileres = aCustomer.getRentals ();
           Resultado de cadena = headerString (aCustomer);
           while (rentals.hasMoreElements ()) {
                Alquiler de cada = (Alquiler) rentals.nextElement ();

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

           // agrega líneas de pie de página
           resultado + = "El monto adeudado es" +
String.valueOf (aCustomer.getTotalCharge ()) + "\ n";
           resultado + = "Ganaste" +
String.valueOf (aCustomer.getTotalFrequentRenterPoints ()) +
                 "puntos frecuentes de arrendamiento";
           resultado de retorno;
     }

 clase HtmlStatement ...
    String headerString (Cliente a Cliente) {
           devuelve "<H1> Alquileres para <EM>" + aCustomer.getName () +
"</EM> </H1> <P> \ n";
 }
 Valor de cadena pública (Cliente a Cliente) {
```

```
        Enumeración alquileres = aCustomer.getRentals ();
        Resultado de cadena = headerString (aCustomer);
        while (rentals.hasMoreElements ()) {
             Alquiler de cada = (Alquiler) rentals.nextElement ();
             // muestra las cifras de cada alquiler
             resultado + = each.getMovie (). getTitle () + ":" +
                           String.valueOf (each.getCharge ()) + "<BR> \ n";
        }
```


Page 285

```
        // agrega líneas de pie de página
      resultado + = "<P> Debe <EM>" +
String.valueOf (aCustomer.getTotalCharge ()) + "</ EM> <P> \ n";
       resultado + = "En este alquiler ganó <EM>" +
            String.valueOf (aCustomer.getTotalFrequentRenterPoints ()) +
            "</EM> puntos frecuentes de alquiler <P>";
       resultado de retorno;
  }
```

Compilo y pruebo y luego continúo con los otros elementos. Hice los pasos uno a la vez. aquí es el resultado:

```
 class TextStatement ...
   Valor de cadena pública (Cliente a Cliente) {
        Enumeración alquileres = aCustomer.getRentals ();
        Resultado de cadena = headerString (aCustomer);
        while (rentals.hasMoreElements ()) {
             Alquiler de cada = (Alquiler) rentals.nextElement ();
             resultado + = eachRentalString (cada uno);
        }
        resultado + = footerString (aCustomer);
        resultado de retorno;
    }
 Cadena eachRentalString (Rental aRental) {
        return "\ t" + aRental.getMovie (). getTitle () + "\ t" +
             String.valueOf (aRental.getCharge ()) + "\ n";
     }
 String footerString (Cliente a Cliente) {
          devolver "El monto adeudado es" +
String.valueOf (aCustomer.getTotalCharge ()) + "\ n" +
              "Ganaste" +
String.valueOf (aCustomer.getTotalFrequentRenterPoints ()) +
              "puntos frecuentes de arrendamiento";
     }
 clase HtmlStatement ...
   Valor de cadena pública (Cliente a Cliente) {
          Enumeración alquileres = aCustomer.getRentals ();
          Resultado de cadena = headerString (aCustomer);
          while (rentals.hasMoreElements ()) {
               Alquiler de cada = (Alquiler) rentals.nextElement ();
               resultado + = eachRentalString (cada uno);
          }
          resultado + = footerString (aCustomer);
          resultado de retorno;
   }
   Cadena eachRentalString (Rental aRental) {
```

```
        devuelve aRental.getMovie (). getTitle () + ":" +
             String.valueOf (aRental.getCharge ()) + "<BR> \ n";
    }
    String footerString (Cliente a Cliente) {
           devuelve "<P> Debes <EM>" +
String.valueOf (aCustomer.getTotalCharge ()) +
           "</EM> <P>
```


Page 286

```
"+" En este alquiler ganó <EM> "+
           String.valueOf (aCustomer.getTotalFrequentRenterPoints ()) +
             "</EM> puntos frecuentes de alquiler <P>";
    }
```

Una vez que se han realizado estos cambios, los dos métodos de valor se ven notablemente similares. Entonces uso Método Pull Up en uno de ellos, eligiendo la versión de texto al azar. Cuando me detengo, necesito declare los métodos de la subclase como abstractos:

```
 Declaración de clase ...
   Valor de cadena pública (Cliente a Cliente) {
           Enumeración alquileres = aCustomer.getRentals ();
           Resultado de cadena = headerString (aCustomer);
           while (rentals.hasMoreElements ()) {
                Alquiler de cada = (Alquiler) rentals.nextElement ();
                resultado + = eachRentalString (cada uno);
           }
           resultado + = footerString (aCustomer);
           resultado de retorno;
      }
   cadena abstracta headerString (Cliente a Cliente);
   cadena abstracta eachRentalString (Rental aRental);
   abstract String footerString (Cliente a Cliente);
```

Elimino el método de valor de la declaración de texto, la compilación y la prueba. Cuando eso funciona, elimino el método de valor de la declaración HTML, compilar y probar de nuevo. El resultado se muestra en la Figura 11,2

Figura 11.2. Clases después de formar el método de plantilla

Página 287

Después de esta refactorización, es fácil agregar nuevos tipos de declaraciones. Todo lo que tienes que hacer es crear un subclase de enunciado que anula los tres métodos abstractos.

Reemplazar herencia con delegación

Una subclase usa solo parte de una interfaz de superclases o no desea heredar datos.

Cree un campo para la superclase, ajuste los métodos para delegar a la superclase y elimine el subclases

Motivación

La herencia es algo maravilloso, pero a veces no es lo que quieres. A menudo comienzas a heredar de una clase, pero luego descubren que muchas de las operaciones de superclase no son realmente ciertas subclase En este caso, tiene una interfaz que no es un verdadero reflejo de lo que hace la clase. O

puede descubrir que está heredando una carga completa de datos que no es apropiada para la subclase. O puede encontrar que hay métodos de superclase protegidos que no tienen mucho sentido con la subclase

Puedes vivir con la situación y usar la convención para decir que aunque es una subclase, está usando solo parte de la función de superclase. Pero eso da como resultado un código que dice una cosa cuando su la intención es otra cosa: una confusión que debes eliminar.

Al utilizar delegación, deja en claro que solo está haciendo un uso parcial de clase delegada Usted controla qué aspectos de la interfaz tomar y cuáles ignorar. El costo Hay métodos de delegación adicionales que son aburridos de escribir pero que son demasiado simples para equivocarse.

Mecánica

- Cree un campo en la subclase que se refiera a una instancia de la superclase. Inicialízalo a `esta.`
- Cambie cada método definido en la subclase para usar el campo delegado. Compilar y probar Después de cambiar cada método.

  ? rarr; *No podrá reemplazar ningún método que invoque un método en* `super` *que se define en la subclase, o pueden entrar en un infinito recurse Estos métodos se pueden reemplazar solo después de haber roto el herencia.*

- Eliminar la declaración de subclase y reemplazar la asignación de delegado con un asignación a un nuevo objeto.
- Para cada método de superclase utilizado por un cliente, agregue un método de delegación simple.
- Compilar y probar.

Ejemplo

Uno de los ejemplos clásicos de herencia inapropiada es hacer de una pila una subclase de vector. Java 1.1 hace esto en sus utilidades (¡muchachos traviesos!), Pero en este caso uso una forma simplificada de pila:

```
clase MyStack extiende Vector {

    public void push (elemento Object) {
         insertElementAt (elemento, 0);
    }

    public Object pop () {
         Resultado del objeto = firstElement ();
         removeElementAt (0);
         resultado de retorno;
    }
}
```

Al observar a los usuarios de la clase, me doy cuenta de que los clientes solo hacen cuatro cosas con stack: push, pop, tamaño, y está `vacío.` Los dos últimos se heredan de Vector.

Page 289

Comienzo la delegación creando un campo para el vector delegado. Enlace este campo a `este` para que yo puedo mezclar delegación y herencia mientras realizo la refactorización:

```
vector privado _vector = this;
```

Ahora empiezo a reemplazar los métodos para que usen la delegación. Empiezo con push:

```
   public void push (elemento Object) {
      _vector. insertElementAt (elemento, 0);
}
```

Puedo compilar y probar aquí, y todo seguirá funcionando. Ahora pop:

```
   public Object pop () {
      Resultado del objeto = _vector. firstElement ();
      _vector. removeElementAt (0);
      resultado de retorno;
}
```

Una vez que haya completado estos métodos de subclase, necesito romper el enlace a la superclase:

```
clase MyStack extiende Vector

   vector privado _vector = nuevo vector ();
```

Luego agrego métodos de delegación simples para los métodos de superclase utilizados por los clientes:

```
public int size () {
     volver _vector. Talla();
}
public boolean isEmpty () {
     volver _vector. esta vacio();
}
```

Ahora puedo compilar y probar. Si olvidé agregar un método de delegación, la compilación me lo dirá.

Reemplazar delegación con herencia

Page 290

Estás utilizando delegación y a menudo escribes muchas delegaciones simples para toda la interfaz.

Convierta la clase delegante en una subclase del delegado.

Motivación

Este es el otro lado de Reemplazar delegación con herencia . Si te encuentras usando todos los métodos del delegado y están cansados de escribir todos esos métodos simples de delegación, puede volver a la herencia con bastante facilidad.

Hay un par de advertencias a tener en cuenta. Si no está utilizando todos los métodos de la clase para que está delegando, no debe usar *Reemplazar delegación con herencia,* porque un La subclase siempre debe seguir la interfaz de la superclase. Si los métodos de delegación son cansado, tienes otras opciones. Puede dejar que los clientes llamen al delegado ellos mismos con Eliminar al hombre del medio . Puede usar Extract Superclass para separar la interfaz común y luego heredar de la nueva clase. Puede usar la interfaz de extracción de manera similar.

Otra situación a tener en cuenta es aquella en la que el delegado es compartido por más de un objeto y es mutable En este caso, no puede reemplazar el delegado con herencia porque ya no podrá compartir los datos El intercambio de datos es una responsabilidad que no puede transferirse nuevamente a la herencia. Cuando el objeto es inmutable, el intercambio de datos no es un problema, ya que solo puede copiar y Nadie puede decirlo.

Mecánica

- Convierta el objeto delegante en una subclase del delegado.
- Compilar.

> ? rarr; *Puede obtener algunos conflictos de métodos en este punto; los métodos pueden tienen el mismo nombre pero varían en tipo de devolución, excepciones o visibilidad. Utilizar* Cambie el nombre del método *para solucionarlos.*

- Configure el campo delegado para que sea el objeto mismo.
- Eliminar los métodos simples de delegación.
- Compilar y probar.
- Reemplace todas las demás delegaciones con llamadas al objeto mismo.
- Eliminar el campo delegado.

Ejemplo

Un empleado simple delega a una persona simple:

```
Empleado de clase {
   Person _person = new Person ();

   public String getName () {
        return _person.getName ();
   }
   public void setName (String arg) {
         _person.setName (arg);
   }
   public String toString () {
        return "Emp:" + _person.getLastName ();
   }
}

Persona de clase {
   String _name;

   public String getName () {
        return _name;
   }
   public void setName (String arg) {
        _name = arg;
   }
   public String getLastName () {
        return _name.substring (_name.lastIndexOf ('') +1);
   }
}
```

El primer paso es declarar la subclase:

```
clase Empleado extiende Persona
```

La compilación en este punto me alerta sobre cualquier conflicto de métodos. Estos ocurren si los métodos con el nombre tener diferentes tipos de retorno o lanzar diferentes excepciones. Cualquier problema de este tipo debe solucionarse con Renombrar método . Este simple ejemplo está libre de tales gravámenes.

El siguiente paso es hacer que el campo delegado se refiera al objeto mismo. Debo eliminar todo simple métodos de delegación como `getName` y `setName`. Si dejo alguno, obtendré un desbordamiento de pila error causado por recursividad infinita. En este caso, esto significa eliminar `getName` y `setName` de `Empleado`.

Una vez que tengo la clase funcionando, puedo cambiar los métodos que usan los métodos delegados. yo cámbielos para usar llamadas directamente:

```
   public String toString () {
         return "Emp:" + getLastName ();
```

Page 292

```
}
```

Una vez que me deshago de todos los métodos que usan métodos delegados, puedo deshacerme del campo `_person` .

Capítulo 12. Grandes refactorizaciones

por Kent Beck y Martin Fowler

Los capítulos anteriores presentan los "movimientos" individuales de refactorización. Lo que falta es un sentido de todo el "juego". Está refactorizando para algún propósito, no solo para evitar progresos (en por lo general, estás refactorizando para algún propósito). ¿Cómo se ve todo el juego?

La naturaleza del juego

Una cosa que seguramente notará en lo que sigue es que los pasos no se detallan tan cuidadosamente como en las refactorizaciones anteriores. Eso es porque las situaciones cambian mucho en el gran refactorizaciones. No podemos decirte exactamente qué hacer, porque no sabemos exactamente qué serás viendo cuando lo haces. Cuando agrega un parámetro a un método, la mecánica es clara porque el alcance es claro. Cuando estás desenredando un desastre de herencia, cada desastre es diferente.

Otra cosa a tener en cuenta sobre estas refactorizaciones es que toman tiempo. Todas las refactorizaciones en Los capítulos 6 al 11 se pueden lograr en unos minutos o una hora como máximo. Tenemos Trabajé en algunas de las grandes refactorizaciones durante meses o años en sistemas en ejecución. Cuando usted tiene una sistema y está en producción y necesita agregar funcionalidad, tendrá dificultades persuadir a los gerentes de que deben detener el progreso durante un par de meses mientras usted ordena. En cambio, tienes que hacer como Hansel y Gretel y mordisquear los bordes, un poco hoy, un Un poco más mañana.

Al hacerlo, debe guiarse por su necesidad de hacer otra cosa. Haz las refactorizaciones como necesita agregar funciones y corregir errores. No tiene que completar la refactorización cuando empezar. Haz todo lo que necesites para lograr tu tarea real. Siempre puedes volver mañana.

Esta filosofía se refleja en los ejemplos. Para mostrarle cada una de las refactorizaciones en este libro tomaría fácilmente cien páginas cada una. Lo sabemos porque Martin lo intentó. Entonces hemos comprimió los ejemplos en unos pocos diagramas incompletos.

Debido a que pueden tomar tanto tiempo, las grandes refactorizaciones tampoco tienen el instante gratificación de las refactorizaciones en los otros capítulos. Tendrás que tener un poco de fe en que están haciendo que el mundo sea un poco más seguro para su programa cada día.

Las grandes refactorizaciones requieren un grado de acuerdo entre todo el equipo de programación que no necesario con las refactorizaciones más pequeñas. Las grandes refactorizaciones marcan la dirección de muchos, muchos cambios Todo el equipo tiene que reconocer que una de las grandes refactorizaciones está "en juego" y hacer sus movimientos en consecuencia. No quieres meterte en la situación de los dos tipos cuyo auto se detiene cerca de la cima de una colina Salen para empujar, uno en cada extremo del automóvil. Después de una media hora infructuosa el chico de enfrente dice: "Nunca pensé que empujar un auto cuesta abajo sería tan difícil". A lo que el otro chico responde: "¿Qué quieres decir con" cuesta abajo "?"

Por qué son importantes las grandes refactorizaciones

Si las grandes refactorizaciones carecen de tantas de las cualidades que hacen que las pequeñas refactorizaciones sean valiosas (previsibilidad, progreso visible, satisfacción instantánea), ¿por qué son lo suficientemente importantes como para que podamos quería ponerlos en este libro? Porque sin ellos corres el riesgo de invertir tiempo y esfuerzo para aprender a refactorizar y luego refactorizar y no obtener el beneficio. Eso podría reflexiona mal sobre nosotros. No podemos soportar eso.

En serio, refactorizas no porque sea divertido sino porque hay cosas que esperas poder
hazlo con tus programas si refactorizas lo que simplemente no puedes hacer si no refactorizas.

La acumulación de decisiones de diseño medio entendidas eventualmente ahoga un programa como una maleza de agua
ahoga un canal Al refactorizar puede asegurarse de que comprende completamente cómo funciona el programa
debe diseñarse siempre se refleja en el programa. A medida que una hierba de agua se extiende rápidamente
zarcillos, decisiones de diseño parcialmente entendidas propagan rápidamente sus efectos a lo largo de su
programa. Ninguna o dos o incluso diez acciones individuales serán suficientes para erradicar el problema.

Cuatro grandes refactorizaciones

En este capítulo describimos cuatro ejemplos de grandes refactorizaciones. Estos son ejemplos del tipo de
cosa, en lugar de cualquier intento de cubrir todo el terreno. La mayor parte de la investigación y práctica sobre
La refactorización hasta ahora se ha concentrado en las refactorizaciones más pequeñas. Hablando de grandes refactorizaciones en
de esta manera es muy nuevo y ha surgido principalmente de la experiencia de Kent, que es mayor que
cualquiera está haciendo esto a gran escala.

Tease Apart Inheritance trata con una jerarquía de herencia enredada que parece combinar
Varias variaciones de una manera confusa. Convertir diseño de procedimiento en objetos ayuda a resolver el
Problema clásico de qué hacer con el código de procedimiento. Muchos programadores usan orientado a objetos
idiomas sin saber realmente acerca de los objetos, por lo que esta es una refactorización que a menudo tiene que hacer. Si
ve código escrito con el clásico enfoque de dos niveles para interfaces de usuario y bases de datos, verá
encuentra que necesita separar el dominio de la presentación cuando quiere aislar la lógica de negocios
del código de la interfaz de usuario. Desarrolladores orientados a objetos experimentados han aprendido que esto
La separación es vital para un sistema de larga vida y prosperidad. Extraer jerarquía simplifica un
clase demasiado compleja convirtiéndola en un grupo de subclases.

Tease Apart Herencia

Tiene una jerarquía de herencia que realiza dos trabajos a la vez.

Cree dos jerarquías y use la delegación para invocar una de la otra.

Page 295

Motivación

La herencia es genial. Le ayuda a escribir dramáticamente código "comprimido" en subclases. Un solo El método puede tomar importancia fuera de proporción con su tamaño debido a dónde se encuentra en el jerarquía.

No es sorprendente que para un mecanismo tan poderoso, es fácil hacer un mal uso de la herencia. Y el mal uso puede arrastrarte fácilmente sobre ti. Un día estás agregando una pequeña subclase para hacer un pequeño trabajo. El siguiente día que agrega otras subclases para hacer el mismo trabajo en otras partes de la jerarquía. Una semana (o mes o año) después estás nadando en espagueti. Sin paleta.

La herencia enredada es un problema porque conduce a la duplicación de código, la ruina de La existencia del programador. Hace los cambios más difíciles, porque las estrategias para resolver un cierto tipo de problema se extiende por todas partes. Finalmente, el código resultante es difícil de entender. Tú no puede simplemente decir: "Esta jerarquía aquí, calcula resultados". Tienes que decir: "Bueno, calcula resultados, y hay subclases para las versiones tabulares, y cada una de ellas tiene subclases para cada uno de los países ".

Puede detectar fácilmente una única jerarquía de herencia que está haciendo dos trabajos. Si cada clase en un cierto nivel en la jerarquía tiene subclases que comienzan con el mismo adjetivo, probablemente estás haciendo Dos trabajos con una jerarquía.

Mecánica

- Identificar los diferentes trabajos que realiza la jerarquía. Cree una cuadrícula bidimensional (o tridimensional o cuatridimensional, si su jerarquía es un verdadero desastre y tiene algo realmente genial papel cuadriculado) y etiquete los ejes con los diferentes trabajos. Asumimos dos o más las dimensiones requieren aplicaciones repetidas de esta refactorización (una a la vez, por supuesto).
- Decidir qué trabajo es más importante y se debe retener en la jerarquía actual y que se moverá a otra jerarquía.
- Use *Extract Class* (Capítulo 6) en la superclase común para crear un objeto para el trabajo subsidiario y agregue una variable de instancia para contener este objeto.
- Crear subclases del objeto extraído para cada una de las subclases en el original jerarquía. Inicialice la variable de instancia creada en el paso anterior a una instancia de Esta subclase.
- Utilice el *Método de movimiento* (Capítulo 7) en cada una de las subclases para mover el comportamiento en el subclase al objeto extraído.
- Cuando la subclase no tenga más código, elimínelo.
- Continúe hasta que todas las subclases hayan desaparecido. Mira la nueva jerarquía para posibles más refactorizaciones como el *Método Pull Up* o el *Campo Pull Up* (Capítulo 11).

Ejemplos

Tomemos el ejemplo de una jerarquía enredada (Figura 12.1).

Figura 12.1. Una jerarquía enredada

Página 297

Esta jerarquía se hizo de la manera en que lo hizo porque Deal se usaba originalmente solo para mostrar un solo acuerdo. Entonces alguien tuvo la brillante idea de mostrar una tabla de ofertas. Un pequeño experimento con el La subclase rápida Active Deal muestra que, de hecho, puede mostrar una tabla con poco trabajo. Oh quieres tablas de ofertas pasivas, también? No hay problema, otra pequeña subclase y nos vamos.

Dos meses después, el código de la tabla se ha complicado pero no hay un lugar simple para ponerlo, el tiempo apremia, la historia habitual. Ahora es difícil agregar un nuevo tipo de acuerdo, porque la lógica del acuerdo es enredado con la lógica de presentación.

Siguiendo la receta, el primer paso es identificar los trabajos que realiza la jerarquía. Un trabajo es capturando la variación según el tipo de acuerdo. Otro trabajo es capturar la variación según Estilo de presentación. Así que aquí está nuestra cuadrícula:

Acuerdo	Acuerdo activo	Acuerdo pasivo
Acuerdo tabular		

El siguiente paso nos dice que decidamos qué trabajo es más importante. El trato del objeto está lejos más importante que el estilo de presentación, así que dejamos solo Deal y extraemos la presentación estilo a su propia jerarquía. Hablando en términos prácticos, probablemente deberíamos dejar solo el trabajo que tiene la mayor cantidad de código asociado, por lo que hay menos código para mover.

El siguiente paso nos dice que usemos la clase de extracto para crear un estilo de presentación (Figura 12.2).

Figura 12.2. Agregar un estilo de presentación

El siguiente paso nos dice que creemos subclases de la clase extraída o para cada una de las subclases en la jerarquía original (Figura 12.3) y para inicializar la variable de instancia a la apropiada subclase:

Figura 12.3. Agregar subclases de estilo de presentación

```
Constructor ActiveDeal
   ... presentación = nuevo SingleActivePresentationStyle (); ...
```

Bien puede estar diciendo: "¿No tenemos más clases ahora que antes? ¿Cómo es esto? se supone que debe mejorar mi vida? "Es cierto que a veces hay que dar un paso atrás antes de que puedas dar dos pasos hacia adelante. En casos como esta jerarquía enredada, la jerarquía de el objeto extraído casi siempre se puede simplificar dramáticamente una vez que el objeto ha sido extraído. Sin embargo, es más seguro dar un paso a la refactorización que saltar diez pasos por delante del diseño ya simplificado.

Ahora usamos Move Method y Move Field para mover los métodos relacionados con la presentación y variables de las subclases de acuerdo a las subclases de estilo de presentación. No tenemos un buen camino de simular esto con el ejemplo como se dibuja, por lo que le pedimos que se imagine que sucede. Cuando estamos hecho, sin embargo, no debería quedar ningún código en las clases Tabular Active Deal y Tabular Reparto pasivo, por lo que los eliminamos (Figura 12.4).

Figura 12.4. Se han eliminado las subclases tabulares de Deal.

Ahora que hemos separado los dos trabajos, podemos trabajar para simplificar cada uno por separado. Cuando hemos realizada esta refactorización, siempre hemos podido simplificar drásticamente la clase extraída y a menudo simplifican aún más el objeto original. El siguiente movimiento eliminará la distinción activo-pasivo. en el estilo de presentación en la figura 12.6 .

Figura 12.6. Las diferencias de presentación se pueden manejar con un par de variables

Incluso la distinción entre simple y tabular puede ser capturada por los valores de algunas variables. No necesita subclases en absoluto (Figura 12.6).

Figura 12.5. Las jerarquías ahora están separadas

Convertir diseño de procedimiento a objetos

Tiene un código escrito en un estilo de procedimiento.

Convierta los registros de datos en objetos, divida el comportamiento y mueva el comportamiento a los objetos.

Motivación

Un cliente nuestro comenzó una vez un proyecto con dos principios absolutos que los desarrolladores tuvieron que seguir:
(1) debe usar Java, (2) no debe usar objetos.

Podemos reírnos, pero aunque Java es un lenguaje orientado a objetos, hay más para usar objetos
que llamar a un constructor. Usar bien los objetos lleva tiempo para aprender. A menudo te enfrentas a la
problema de código de procedimiento que tiene que estar más orientado a objetos. La situación típica es larga.
métodos de procedimiento en una clase con pocos datos y objetos de datos tontos con nada más que
accesorios Si está convirtiendo desde un programa puramente de procedimiento, es posible que ni siquiera tenga esto,
Pero es un buen lugar para comenzar.

No estamos diciendo que nunca debe tener objetos con comportamiento y poca o ninguna información. Nosotros
A menudo utilizamos pequeños objetos de estrategia cuando necesitamos variar el comportamiento. Sin embargo, tal procedimiento
Los objetos generalmente son pequeños y se usan cuando tenemos una necesidad particular de flexibilidad.

Mecánica

- Tome cada tipo de registro y conviértalo en un objeto de datos tonto con accesores.

  ? rarr; *Si tiene una base de datos relacional, tome cada tabla y conviértala en*
  Objeto de datos tontos.

Página 302

• Tome todo el código de procedimiento y póngalo en una sola clase.

? rarr; *Puede hacer que la clase sea un singleton (para facilitar reinicialización) o hacer que los métodos sean estáticos.*

• Tome cada procedimiento largo y aplique el Método de extracción y las refactorizaciones relacionadas a descomponerlo. A medida que desglosa los procedimientos, use el Método de movimiento para mover cada uno a la clase de datos tontos apropiada.
• Continúe hasta que haya eliminado todo el comportamiento de la clase original. Si el original La clase era una clase puramente de procedimiento, es muy gratificante eliminarla.

Ejemplo

El ejemplo en el Capítulo 1 es un buen ejemplo de la necesidad de *Convertir Diseño Procesal a Objetos,* particularmente la primera etapa, en la que el método de declaración se divide y distribuye. Cuando haya terminado, puede trabajar en objetos de datos ahora inteligentes con otras refactorizaciones.

Dominio separado de la presentación

Tiene clases de GUI que contienen lógica de dominio.

Separe la lógica del dominio en clases de dominio separadas

Motivación

Cada vez que escuchas personas hablando sobre objetos, escuchas sobre el modelo-vista-controlador (MVC). Esta idea apuntala la relación entre la interfaz gráfica de usuario (GUI) y el dominio. objetos en Smalltalk-80.

El oro en el corazón de MVC es la separación entre el código de interfaz de usuario (la *vista*, estos días a menudo llamado la *presentación*) y la lógica de dominio (el *modelo*). Las clases de presentacion contienen solo la lógica necesaria para tratar con la interfaz de usuario. Los objetos de dominio no contienen visual código pero toda la lógica de negocios. Esto separa dos partes complicadas del programa en partes que son más fáciles de modificar También permite múltiples presentaciones de la misma lógica de negocios. Aquellos Con experiencia en el trabajo con objetos, utilice esta separación instintivamente, y ha demostrado su valía.

Pero no es así como la mayoría de las personas que trabajan con GUI hacen su diseño. La mayoría de los entornos con Las GUI cliente-servidor usan un diseño lógico de dos niveles: los datos se encuentran en la base de datos y la lógica se encuentra en Las clases de presentación. El entorno a menudo lo obliga a este estilo de diseño, lo que lo convierte en difícil para usted poner la lógica en otro lugar.

Java es un entorno orientado a objetos adecuado, por lo que puede crear objetos de dominio no visual que contener lógica de negocios. Sin embargo, a menudo encontrará código escrito en el estilo de dos niveles.

Mecánica

- Crear una clase de dominio para cada ventana.
- Si tiene una cuadrícula, cree una clase para representar las filas en la cuadrícula. Use una colección en la clase de dominio de la ventana para contener los objetos de dominio de fila.
- Examinar los datos en la ventana. Si se usa solo para fines de interfaz de usuario, déjelo encendido la ventana. Si se usa dentro de la lógica de dominio pero no se muestra realmente en el ventana, use Move Method para moverlo al objeto de dominio. Si es utilizado tanto por el usuario interfaz y la lógica de dominio, use Duplicar datos observados para que esté en ambos lugares y se mantienen sincronizados.
- Examinar la lógica en la clase de presentación. Utilice el método de extracción para separar la lógica sobre La presentación desde la lógica del dominio. A medida que aísla la lógica del dominio, use el método Move para moverlo al objeto de dominio.
- Cuando haya terminado, tendrá clases de presentación que manejan la GUI y objetos de dominio que contienen toda la lógica de negocios. Los objetos de dominio no estarán bien factorizado, pero más refactorizaciones se ocuparán de eso.

Ejemplo

Un programa que permite a los usuarios ingresar y cotizar pedidos. La GUI se parece a la Figura 12.7 . los La clase de presentación interactúa con una base de datos relacional presentada como en la Figura 12.8 .

Figura 12.7. La interfaz de usuario para un programa de inicio

Figura 12.8. La base de datos para el programa de pedidos.

Todo el comportamiento, tanto para la GUI como para fijar el precio de los pedidos, está en una sola clase de ventana de pedido.

Comenzamos creando una clase de orden adecuada. Vinculamos esto a la ventana de pedido como en la Figura 12.9 .
Debido a que la ventana contiene una cuadrícula para mostrar las líneas de orden, también creamos una clase de línea de orden para las filas de la cuadrícula.

Figura 12.9. Ventana de pedido y orden

Trabajamos desde la ventana en lugar de la base de datos. Basando un modelo de dominio inicial en el
La base de datos es una estrategia razonable, pero nuestro mayor riesgo es mezclar la presentación y la lógica de dominio.
Así que los separamos en función de las ventanas y refactorizamos el resto más adelante.

Con este tipo de programa es útil observar las declaraciones del lenguaje de consulta estructurado (SQL)
incrustado en la ventana. Los datos retirados de las declaraciones SQL son datos de dominio.

Los datos de dominio más fáciles de manejar son aquellos que no se muestran directamente en la GUI. En el
ejemplo, la base de datos tiene un campo de códigos en la tabla de clientes. El código no se muestra directamente
en la GUI; se convierte en una frase más legible para los humanos. Como tal, el campo es una clase simple,
como una cadena, en lugar de un componente AWT. Podemos usar Mover campo de forma segura para mover ese campo
a la clase de dominio.

No tenemos tanta suerte con los otros campos. Contienen componentes AWT que se muestran en el
ventana y utilizado en los objetos de dominio. Para estos, necesitamos utilizar datos observados duplicados .
Esto coloca un campo de dominio en la clase de orden con un campo AWT correspondiente en la ventana de orden.

Este es un proceso lento, pero al final podemos obtener todos los campos lógicos de dominio en la clase de dominio.
Una buena forma de impulsar este proceso es intentar mover todas las llamadas SQL a la clase de dominio. Usted puede
haga esto para mover la lógica de la base de datos y los datos del dominio a la clase de dominio juntos. Usted puede
obtenga una buena sensación de finalización eliminando la importación de java.sql de la ventana de pedido. Esta
significa que haces mucho Método de extracción y Método de movimiento .

Las clases resultantes, como en la Figura 12.10 , están muy lejos de estar bien factorizadas. Pero esto
modelo es suficiente para separar la lógica del dominio. A medida que realiza esta refactorización, debe pagar
atención a dónde está su riesgo. Si la mezcla de presentación y lógica de dominio es la mayor
riesgo, separarlos completamente antes de hacer mucho más. Si otras cosas son más importantes,
como las estrategias de fijación de precios para los productos, obtenga la lógica de la parte importante fuera de la ventana
y refactorizar esa lógica para crear una estructura adecuada para el área de alto riesgo. Lo más probable es

Página 306

que la mayor parte de la lógica de dominio tendrá que ser sacada de la ventana de pedido. Si puedes refactorizar
y deje un poco de lógica en la ventana, hágalo para abordar primero su mayor riesgo.

Figura 12.10. Distribuir los datos a las clases de dominio.

Extraer Jerarquía

Tienes una clase que está haciendo demasiado trabajo, al menos en parte a través de muchos condicionales declaraciones.

Cree una jerarquía de clases en la que cada subclase represente un caso especial.

Página 307

Motivación

En el diseño evolutivo, es común pensar que una clase implementa una idea y llegar a
más tarde nos damos cuenta de que realmente está implementando dos, tres o diez. Creas la clase simplemente al principio. UNA
unos días o semanas después, verá que si solo agrega una bandera y un par de pruebas, puede usarla en
Un nuevo caso. Un mes después tienes otra oportunidad. Un año después tienes un verdadero desastre:
banderas y expresiones condicionales por todo el lugar.

Cuando te encuentres con una clase de navaja suiza que ha crecido para abrir latas, corta pequeñas
árboles, brillen un punto láser en artículos de bala de presentación reacios, y, oh sí, supongo que cortarán cosas,
necesitas una estrategia para separar los distintos hilos. La estrategia aquí solo funciona si su
La lógica condicional permanece estática durante la vida del objeto. Si no, puede que tenga que usar Extraer
Clasifique antes de que pueda comenzar a separar los casos entre sí.

No se desanime si *Extract Hierarchy* es una refactorización que no puede terminar en un día. Puede tomar
semanas o meses para desenredar un diseño que se ha enredado. Haz los pasos que sean fáciles y
obvio, entonces tómate un descanso. Haga un trabajo visiblemente productivo durante unos días. Cuando has aprendido
algo, regrese y haga algunos pasos más fáciles y obvios.

Mecánica

Página 308

Hemos puesto dos juegos de mecánica. En el primer caso, no está seguro de cuáles deberían ser las variaciones
ser. En este caso, desea dar un paso a la vez, de la siguiente manera:

- Identificar una variación.

? rarr; *Si las variaciones pueden cambiar durante la vida del objeto, use*

Extraiga Clase *para llevar ese aspecto a una clase separada.*

- Cree una subclase para ese caso especial y use Reemplazar constructor con fábrica Método sobre el original. Modifique el método de fábrica para devolver una instancia de la subclase donde corresponda.
- Uno a la vez, copie métodos que contengan lógica condicional a la subclase, luego simplifique los métodos dados lo que puede decir con certeza sobre las instancias de la subclase que No puedo decir sobre instancias de la superclase.

  ? rarr; *Utilice el* método de extracción *en la superclase si es necesario para aislar partes condicionales de métodos de las partes incondicionales.*

- Continúe aislando casos especiales hasta que pueda declarar el resumen de la superclase.
- Eliminar los cuerpos de los métodos en la superclase que se anulan en todas las subclases y Hacer que las declaraciones de la superclase sean abstractas.

Cuando las variaciones son muy claras desde el principio, puede usar una estrategia diferente, de la siguiente manera:

- Crear una subclase para cada variación.
- Use Reemplazar constructor con método de fábrica para devolver la subclase apropiada para cada variación

  ? rarr; *Si las variaciones están marcadas con un código de tipo, use* Reemplazar tipo Código con subclases . *Si las variaciones pueden cambiar dentro de la vida de la clase, use* Reemplazar código de tipo con Estado / Estrategia .

- Tome métodos que tengan lógica condicional y aplique Reemplazar condicional con El polimorfismo . Si el método completo no varía, aísle la parte variable con Extracto Método .

Ejemplo

El ejemplo es un caso no obvio. Puede seguir las refactorizaciones para Reemplazar código de tipo con subclases , reemplazar código de tipo con estado / estrategia y reemplazar condicional con Polimorfismo para ver cómo funciona el caso obvio.

Comenzamos con un programa que calcula una factura de electricidad. Los objetos iniciales se parecen a la figura 12.11 .

Figura 12.11. Cliente y esquema de facturación

El esquema de facturación contiene mucha lógica condicional para la facturación en diferentes circunstancias. Se usan diferentes cargos para el verano y el invierno, y se usan diferentes planes de facturación para residencial, pequeña empresa, clientes que reciben Seguridad Social (línea de vida) y aquellos con un discapacidad. La lógica compleja resultante hace que la clase Esquema de facturación sea bastante compleja.

Nuestro primer paso es elegir un aspecto variante que siga apareciendo en la lógica condicional. Esto podría Existen varias condiciones que dependen de si el cliente tiene un plan de discapacidad. Esto puede ser un marca en Cliente, Esquema de facturación o en otro lugar.

Creamos una subclase para la variación. Para usar la subclase, debemos asegurarnos de que se cree y usado Así que miramos al constructor para el esquema de facturación. Primero usamos Reemplazar constructor con el método de fábrica . Luego observamos el método de fábrica y vemos cómo la lógica depende de discapacidad. Luego creamos una cláusula que devuelve un esquema de facturación por discapacidad cuando corresponde.

Observamos los diversos métodos en el esquema de facturación y buscamos aquellos que contienen condiciones lógica que varía según la discapacidad. CreateBill es uno de esos métodos, por lo que lo copiamos al subclase (Figura 12.12).

Figura 12.12. Agregar una subclase por discapacidad

Ahora examinamos la copia de la subclase de createBill y la simplificamos sobre la base de que sabemos que es ahora dentro del contexto de un esquema de discapacidad. Entonces el código que dice

```
if (disabledScheme ()) doSomething
```


Page 310

puede ser reemplazado con

```
hacer algo
```

Si las discapacidades son exclusivas del esquema comercial, podemos eliminar cualquier código que sea condicional en el esquema de negocios.

Mientras hacemos esto, nos gusta asegurarnos de que el código variable se separe del código que permanece igual. Utilizamos el método de extracción y la descomposición condicional para hacer eso. Seguimos haciendo esto por varios métodos de esquema de facturación hasta que sentimos que hemos lidiado con la mayor parte de la discapacidad condicionales Luego elegimos otra variación, digamos línea de vida, y hacemos lo mismo para eso.

Sin embargo, a medida que hacemos la segunda variación, observamos cómo las variaciones de la línea de vida se comparan con los de discapacidad. Queremos identificar casos en los que podamos tener métodos que tengan el mismo intención, pero llevarlo a cabo de manera diferente en los dos casos separados. Podríamos tener variación en el cálculo de impuestos para los dos casos. Queremos asegurarnos de que tenemos dos métodos en el subclases que tienen la misma firma. Esto puede significar alterar la discapacidad para que podamos alinear el subclases Por lo general, encontramos que a medida que hacemos más variaciones, el patrón de similar y variable Los métodos tienden a estabilizarse, lo que facilita las variaciones adicionales.

Página 311

Capítulo 13. Refactorización, reutilización y realidad

por William Opdyke

Martin Fowler y yo nos conocimos en Vancouver durante OOPSLA 92. Unos meses antes, tuve Completé mi tesis doctoral sobre la refactorización de marcos orientados a objetos [1] en la Universidad de Illinois Mientras estaba considerando continuar mi investigación sobre la refactorización, también estaba explorando otras opciones, como la informática médica. Martin estaba trabajando en una informática médica. aplicación en ese momento, que es lo que nos unió para conversar durante el desayuno en Vancouver. Como

Martin relata anteriormente en este libro, pasamos unos minutos discutiendo mi investigación de refactorización. Él tenía un interés limitado en el tema en ese momento, pero como ya sabe, su interés en el tema ha crecido.

A primera vista, podría parecer que la refactorización comenzó en los laboratorios de investigación académica. En realidad, es comenzó en las trincheras de desarrollo de software, donde los programadores orientados a objetos, luego usaban Smalltalk, encontró situaciones en las que se necesitaban técnicas para apoyar mejor el proceso de desarrollo de marcos o, más generalmente, para apoyar el proceso de cambio. Esto generó investigación que ha madurado hasta el punto en que sentimos que está "lista para el horario estelar", el punto en que un conjunto más amplio de profesionales de software puede experimentar los beneficios de la refactorización.

Cuando Martin me ofreció la oportunidad de escribir un capítulo en este libro, surgieron varias ideas mente. Podría describir la investigación de refactorización temprana, la era en la que Ralph Johnson y yo vinimos juntos de muy diversos antecedentes técnicos para centrarse en el apoyo al cambio de objeto software orientado Podría discutir cómo proporcionar soporte automatizado para la refactorización, un área de mi investigación es bastante diferente del enfoque de este libro. Podría compartir algunas de las lecciones que tengo. aprendió cómo la refactorización se relaciona con las preocupaciones cotidianas de los profesionales del software, especialmente aquellos que trabajan en grandes proyectos en la industria.

Muchas de las ideas que obtuve durante mi investigación de refactorización han sido útiles en una amplia gama de áreas: en la evaluación de tecnologías de software y en la formulación de estrategias de evolución de productos, en desarrollando prototipos y productos en la industria de las telecomunicaciones, y en capacitación y consultoría con grupos de desarrollo de productos.

Decidí centrarme brevemente en muchos de estos temas. Como el título de este capítulo implica, muchas de las Los conocimientos sobre la refactorización se aplican de manera más general a cuestiones como la reutilización de software, el producto evolución y selección de plataforma. Aunque partes de este capítulo tocan brevemente algunos de los más Aspectos teóricos interesantes de la refactorización, el enfoque principal es práctico, en el mundo real preocupaciones y cómo se pueden abordar.

Si desea explorar más la refactorización, consulte Recursos y referencias para la refactorización más adelante en Este capítulo.

Un control de realidad

Trabajé en Bell Labs durante varios años antes de decidir continuar mis estudios de doctorado. La mayoría de ese tiempo lo pasó trabajando en una parte de la compañía que desarrolló la conmutación electrónica sistemas. Dichos productos tienen limitaciones muy estrictas con respecto a la fiabilidad y la velocidad. con el que manejan llamadas telefónicas. Se han invertido miles de años de personal en el desarrollo y evolucionando tales sistemas. La vida útil de los productos ha abarcado décadas. La mayor parte del costo de desarrollar estos sistemas no viene en desarrollar la versión inicial sino en cambiar y adaptando los sistemas a lo largo del tiempo. Las formas de hacer tales cambios más fáciles y menos costosos resultarían en una gran victoria para la empresa.

Página 312

Debido a que Bell Labs estaba financiando mis estudios de doctorado, quería un campo de investigación que no fuera solo técnicamente interesante pero también relacionado con una necesidad práctica de negocios. A fines de la década de 1980, el objeto La tecnología orientada apenas comenzaba a emerger de los laboratorios de investigación. Cuando Ralph Johnson propuso un tema de investigación que se centró tanto en la tecnología orientada a objetos como en el apoyo a la proceso de cambio y evolución del software, lo agarré.

Me han dicho que cuando las personas terminan sus estudios de doctorado, rara vez son neutrales con respecto a su tema. Algunos están cansados del tema y rápidamente pasan a otra cosa. Otros permanecen Entusiasta sobre el tema. Estaba en el último campamento.

Cuando volví a Bell Labs después de recibir mi título, sucedió algo extraño. La gente

a mi alrededor no estaban tan entusiasmados con la refactorización como yo.

Puedo recordar vívidamente presentar una charla a principios de 1993 en un foro de intercambio de tecnología para el personal de AT&T Bell Labs y NCR (todos éramos parte de la misma compañía en ese momento). Me dieron 45 minutos para hablar sobre refactorización. Al principio la conversación pareció ir bien. Mi entusiasmo por el tema se encontró con Pero al final de la charla, había muy pocas preguntas. Uno de los asistentes vino después para aprender más; estaba comenzando su trabajo de posgrado y estaba pescando Un tema de investigación. Tenía la esperanza de ver a algunos miembros de proyectos de desarrollo mostrar entusiasmo en aplicando refactorización a sus trabajos. Si estaban ansiosos, no lo expresaron en ese momento.

La gente simplemente no parecía entenderlo.

Ralph Johnson me enseñó una importante lección sobre investigación: si alguien (un revisor de un artículo, un asistente en una charla) comenta: "No entiendo" o simplemente no lo entiende, es *nuestra* culpa. Es *nuestro* responsabilidad de trabajar duro para desarrollar y comunicar nuestras ideas.

Durante los siguientes dos años, tuve numerosas oportunidades para hablar sobre la refactorización en AT&T Bell Foros internos de laboratorios y en conferencias y talleres externos. Como hablé más con desarrolladores en las trincheras, comencé a entender por qué mis mensajes anteriores no aparecían claramente. La desconexión fue causada en parte por la novedad de la tecnología orientada a objetos. Aquellos quienes habían trabajado con él rara vez habían progresado más allá de la versión inicial y, por lo tanto, todavía no enfrentado los difíciles problemas de evolución que la refactorización puede ayudar a resolver. Este era el típico investigador dilema: el estado del arte estaba más allá del estado de la práctica común. Sin embargo, hubo otra causa preocupante para la desconexión. Hubo varias razones de sentido común los desarrolladores, incluso si adquirieron los beneficios de la refactorización, se mostraron reacios a refactorizar *sus* programas Estas preocupaciones tuvieron que ser abordadas antes de que la refactorización pudiera ser adoptada por el comunidad de desarrollo.

¿Por qué los desarrolladores son reacios a refactorizar sus programas?

Supongamos que eres un desarrollador de software. Si su proyecto es un nuevo comienzo (sin retroceso) problemas de compatibilidad) *y* si comprende el problema que su sistema está destinado a resolver *y* si su financiador está dispuesto a pagar hasta *que* esté satisfecho con los resultados, considérese muy afortunado. Aunque tal escenario puede ser ideal para aplicar técnicas orientadas a objetos, para La mayoría de nosotros tal escenario es solo un sueño.

Más a menudo se le pide que extienda una pieza de software existente. Tienes un poco menos que completo comprensión de lo que estás haciendo. Está bajo presión programada para producir. Que puedes ¿hacer?

Página 313

Puedes reescribir el programa. Puede aprovechar su experiencia de diseño y corregir los males del pasado y ser creativo y divertirse. ¿Quién pagará la factura? ¿Cómo puedes estar seguro de que el nuevo Qué sistema hace todo el sistema anterior?

Puede copiar y modificar partes del sistema existente para ampliar sus capacidades. Esto puede parecer conveniente e incluso puede verse como una forma de demostrar la reutilización; ni siquiera tienes que entiende lo que estás reutilizando. Sin embargo, con el tiempo, los errores se propagan, los programas hincharse, el diseño del programa se corrompe y el costo incremental del cambio se intensifica

La refactorización es un punto medio entre los dos extremos. Es una forma de reestructurar el software para hacer que las ideas de diseño sean más explícitas, para desarrollar marcos y extraer componentes reutilizables, para

aclarar la arquitectura del software y prepararse para facilitar las adiciones. Refactorizar puede ayudar usted aprovecha su inversión pasada, reduce la duplicación y simplifica un programa.

Supongamos que usted, como desarrollador, acepta estas ventajas. Estás de acuerdo con Fred Brooks en que tratar con el cambio es una de las "complejidades esenciales" del desarrollo de software. [2] Usted acepta que en La refactorización abstracta puede proporcionar las ventajas indicadas.

¿Por qué todavía no puedes refactorizar *tus* programas? Aquí hay cuatro posibles razones:

1. Es posible que no entiendas cómo refactorizar.
2. Si los beneficios son a largo plazo, ¿por qué hacer el esfuerzo ahora? A largo plazo, es posible que no seas con el proyecto para cosechar los beneficios.
3. El código de refactorización es una actividad general; Te pagan para escribir *nuevas* funciones.
4. La refactorización puede romper el programa existente.

Estas son todas las preocupaciones válidas. Los he escuchado expresados por el personal de telecomunicaciones y en Empresas de alta tecnología. Algunos de estos son problemas técnicos; otros son gerenciales preocupaciones Todo debe abordarse antes de que los desarrolladores consideren refactorizar su software. Vamos lidiar con cada uno de estos problemas a su vez.

Comprender cómo y dónde refactorizar

¿Cómo puedes aprender a refactorizar? ¿Cuáles son las herramientas y técnicas? Como pueden ser combinado para lograr algo útil? ¿Cuándo debemos aplicarlos? Este libro define varias docenas de refactorizaciones que Martin encontró útiles en su trabajo. Presenta ejemplos de cómo el las refactorizaciones se pueden aplicar para respaldar cambios significativos en los programas.

En el proyecto Software Refactory de la Universidad de Illinois, elegimos un enfoque minimalista. Nosotros definió un conjunto más pequeño de refactorizaciones [1] , [3] y mostró cómo podrían aplicarse. Nosotros basamos nuestra colección de refactorizaciones en nuestras propias experiencias de programación. Evaluamos el estructural evolución de varios marcos orientados a objetos, principalmente en C ++, y habló y leyó el retrospectivas de varios desarrolladores experimentados de Smalltalk. La mayoría de nuestras refactorizaciones son bajas nivel, como crear o eliminar una clase, variable o función; cambio de atributos de variables y funciones, como sus permisos de acceso (p. ej., públicos o protegidos) y funciones argumentos; o mover variables y funciones entre clases. Un conjunto más pequeño de alto nivel las refactorizaciones se utilizan para operaciones como crear una superclase abstracta, simplificar una clase mediante subclases y simplificando condicionales, o dividiendo parte de una clase existente para crear una nueva clase de componente reutilizable (a menudo convirtiendo entre herencia y delegación o agregación). Las refactorizaciones más complejas se definen en términos de las refactorizaciones de bajo nivel. Nuestro enfoque fue motivado por la preocupación por el soporte automatizado y la seguridad, que discuto más adelante.

Página 314

Dado un programa existente, ¿qué refactorizaciones debe aplicar? Eso depende, por supuesto, de su metas. Una razón común, que es el enfoque de este libro, es reestructurar un programa para hacerlo Es más fácil agregar (a corto plazo) una nueva característica. Discuto esto en la siguiente sección. Hay, sin embargo, otras razones por las cuales puede aplicar refactorizaciones.

Programadores experimentados orientados a objetos y aquellos que han sido entrenados en patrones de diseño. y buenas técnicas de diseño han aprendido que varias cualidades estructurales deseables y Se ha demostrado que las características de los programas son compatibles con la extensibilidad y la reutilización. [4] , [5] , [6] Las técnicas de diseño orientado a objetos, como CRC [7], se centran en definir clases y sus protocolos. Aunque la atención se centra en el diseño inicial, hay formas de evaluar los programas existentes contra tales pautas.

Se puede utilizar una herramienta automatizada para identificar debilidades estructurales en un programa, como funciones

que tienen un número excesivamente grande de argumentos o son excesivamente largos. Estos son candidatos para refactorización. Una herramienta automatizada también puede identificar similitudes estructurales que pueden indicar despidos. Por ejemplo, si dos funciones son casi idénticas (como sucede a menudo cuando un proceso de copiar y modificar se aplica a una primera función para producir una segunda), tales similitudes se pueden detectar y se sugieren refactorizaciones que pueden mover el código común a un lugar. Si dos Las variables en diferentes partes de un programa tienen el mismo nombre, a veces se pueden reemplazar con una sola variable que se hereda en ambos lugares. Estos son algunos ejemplos muy simples. Muchos otros casos más complejos se pueden detectar y corregir con una herramienta automatizada. Estas anormalidades estructurales o similitudes estructurales no siempre significan que desearía aplicar un refactorización, pero a menudo lo hacen.

Gran parte del trabajo sobre patrones de diseño se ha centrado en un buen estilo de programación y en útiles patrones de interacciones entre partes de un programa que pueden mapearse en estructuras características y en refactorización. Por ejemplo, la sección de aplicabilidad del método de plantilla El patrón [8] se refiere a nuestra refactorización de superclase abstracta. [9]

He enumerado [1] algunas de las heurísticas que pueden ayudar a identificar candidatos para refactorizar en un C ++ programa. John Brant y Don Roberts [10] , [11] han creado una herramienta que aplica un extenso conjunto de heurísticas para analizar automáticamente los programas Smalltalk. Sugieren refactorizaciones que podría mejorar el diseño del programa y dónde aplicarlos.

Aplicar una herramienta de este tipo para analizar su programa es algo análogo a aplicar pelusa a una C o Programa C ++. La herramienta no es lo suficientemente inteligente como para comprender el significado del programa. Sólo algunos de las sugerencias que hace sobre la base del análisis estructural del programa pueden ser cambios que usted Realmente quiero hacer. Como programador, haces la llamada. Tú decides qué recomendaciones aplicar a su programa. Esos cambios deberían mejorar la estructura de su programa y mejorar Apoyar los cambios en el camino.

Antes de que los programadores puedan convencerse de que deberían refactorizar su código, deben Comprender cómo y dónde refactorizar. No hay sustituto para la experiencia. Aprovechamos el ideas de desarrolladores experimentados orientados a objetos en nuestra investigación para obtener un conjunto de útiles refactorizaciones e ideas sobre dónde deben aplicarse. Las herramientas automatizadas pueden analizar el estructura de un programa y sugerir refactorizaciones que podrían mejorar esa estructura. Como con la mayoría Las disciplinas, herramientas y técnicas pueden ayudar, *pero solo si las usa.* A medida que los programadores refactorizan sus código, su comprensión crece.

Refactorización de programas C ++

Bill Opdyke

Página 315

Cuando Ralph Johnson y yo comenzamos nuestra investigación de refactorización en 1989, el El lenguaje de programación C ++ estaba evolucionando y se estaba volviendo muy popular dentro de la comunidad orientada a objetos. La importancia de La refactorización se había reconocido por primera vez dentro de la comunidad Smalltalk. Nosotros consideró que demostrar su aplicabilidad a los programas de C ++ interesaría Comunidad más amplia de desarrolladores orientados a objetos.

C ++ tiene características de lenguaje, más notablemente su verificación de tipo estático, que simplificar algunas tareas de análisis y refactorización de programas. En el otro Por otro lado, el lenguaje C ++ es complejo, en gran parte debido a su historia y evolución desde el lenguaje de programación C. Algunos estilos de programación permitido en C ++ hace que sea difícil refactorizar y desarrollar un programa.

Características del lenguaje y estilos de programación que admiten Refactorización

Las características de escritura estática de C ++ hacen que sea relativamente fácil reducir posibles referencias a la parte del programa que tal vez desee refactorizador Para elegir un caso simple pero frecuente, suponga que desea cambiar el nombre de una función miembro de una clase C ++. Para aplicar correctamente el al cambiar el nombre, debe cambiar la declaración de la función y todas las referencias a esa función Encontrar y cambiar las referencias puede ser difícil si el El programa es grande.

En comparación con Smalltalk, C ++ tiene herencia y protección de clase. características de modo de control de acceso (público, protegido, privado) que lo hacen Es más fácil determinar dónde se cambia el nombre de las referencias a la función puede ser. Si la función a renombrar se declara privada para la clase, entonces las referencias a esa función solo pueden ocurrir dentro de la propia clase o en clases que se declaran como amigos de esa clase. Si la función es declarado protegido, las referencias solo se pueden encontrar en la clase, su subclases (y sus descendientes), y en amigos de la clase. Si el la función se declara pública (el modo de protección menos restrictivo), el el análisis todavía se limita a las clases enumeradas para funciones "protegidas" y operaciones en instancias de la clase que contiene la función, su subclases y sus descendientes.

En algunos programas muy grandes, las funciones con el mismo nombre pueden ser declarado en diferentes lugares del programa. En algunos casos, dos o más. las funciones con el mismo nombre podrían reemplazarse mejor con una sola función; hay refactorizaciones que a menudo se pueden aplicar para hacer esto cambio. Por otro lado, a veces sucede que uno de los las funciones deben renombrarse mientras que la otra función se mantiene igual. En un proyecto de varias personas, dos o más programadores podrían haber dado mismo nombre para funciones que son independientes entre sí. En C ++, al cambiar el nombre de una de estas funciones, casi siempre es

Página 316

fácil de determinar qué referencias se refieren a la función que se renombra y que se refieren a la otra función. En Smalltalk, ese análisis es más difícil.

Debido a que C ++ usa subclases para implementar subtipos, el alcance a un variable o función generalmente puede generalizarse o especializarse moviendo hacia arriba o hacia abajo de una jerarquía de herencia. Los esfuerzos para analizar un programa y realizar la refactorización son bastante sencillos.

Varios buenos principios de diseño aplicados durante el desarrollo inicial y a lo largo del proceso de desarrollo de software facilita el proceso de refactorizar y facilitar la evolución del software. Miembro definitorio variables y la mayoría de las funciones miembro como privadas o protegidas es un

técnica de abstracción que a menudo facilita la refactorización de las partes internas de una clase mientras se minimizan los cambios realizados en otras partes de un programa. Uso de la herencia para modelar jerarquías de generalización y especialización (como es natural en C ++) hace que sea bastante sencillo generalizar o especialice el alcance de las funciones o variables miembro utilizando refactorizaciones para mover estos miembros dentro de las jerarquías de herencia.

Las características en entornos C ++ admiten refactorización. Si mientras refactoriza un programar un programador introduce un error, a menudo los indicadores del compilador de C ++ el error. Muchos entornos de desarrollo de software C ++ proporcionan potentes capacidades para referencias cruzadas y navegación de código.

Características del lenguaje y estilos de programación

Esa complicada refactorización

La compatibilidad de C ++ con C es, como la mayoría de ustedes saben, un doble espada afilada Muchos programas se han escrito en C y muchos los programadores han estado entrenando en C, que (en la superficie, al menos) facilita la migración a C ++ que a otras orientadas a objetos idiomas Sin embargo, C ++ admite muchos estilos de programación, algunos de que violan los principios del diseño de sonido.

Programas que usan características del lenguaje C ++ como punteros, cast Las operaciones y sizeof (objeto) son difíciles de refactorizar. Punteros y elenco Las operaciones introducen alias, lo que dificulta determinar todas referencias a un objeto que quizás desee refactorizar. Cada uno de estos características expone la representación interna de un objeto, que viola principios de abstracción.

Por ejemplo, C ++ usa un mecanismo de tabla en V para representar miembros variables en el programa ejecutable. El conjunto de miembros heredados Las variables aparecen primero seguidas de las variables miembro definidas localmente.

316

Página 317

Una refactorización generalmente segura es mover una variable a una superclase. Debido a que la variable ahora se hereda en lugar de definirse localmente en el subclase, es probable que la ubicación física de la variable en el ejecutable haber cambiado como resultado de la refactorización. Si todas las referencias variables en el programa son a través de la interfaz de clase, como un reordenamiento físico las ubicaciones de la variable no cambiarán el comportamiento del programa.

Por otro lado, si la variable fue referenciada a través del puntero aritmética (por ejemplo, un programador tenía un puntero a un objeto, usado saber que la variable estaba en el quinto byte y le asignó un valor a el quinto byte del objeto usando la aritmética del puntero), luego moviendo el variable a la superclase probablemente cambiará el comportamiento del programa. Del mismo modo, si un programador escribió un condicional del formulario si (sizeof (object) == 15)), y refactorizó el programa para eliminar un

variable sin referencia de una clase, el tamaño de las instancias de esa clase cambiaría, y un condicional que una vez probado verdadero probaría falso.

Puede parecer absurdo que alguien escriba programas condicional pruebas basadas en el tamaño de los objetos o para usar aritmética de puntero cuando C ++ Proporciona una interfaz mucho más limpia para las variables de clase. Mi punto es que Estas características (y otras que dependen del diseño físico de un objeto) se proporcionan en C ++, y hay programadores con experiencia en utilizarlos. La migración de C a C ++ no está orientada a objetos programador o diseñador de marca.

Porque C ++ es un lenguaje tan complicado (en comparación con Smalltalk y, en menor medida, Java), es mucho más difícil crear los tipos de representaciones de la estructura del programa que son útiles para automatizar soporte para verificar si una refactorización es segura y, si es así, para realizando la refactorización.

Como C ++ resuelve la mayoría de las referencias en tiempo de compilación, refactoriza un el programa generalmente requiere recompilar al menos parte de un programa y vincular el ejecutable antes de probar los efectos de un cambio. Por contraste, Smalltalk y CLOS (Common Lisp Object System) proporcionan entornos para interpretación y compilación incremental. Mientras aplicando (y posiblemente retrocediendo) una serie de refactorizaciones incrementales es bastante natural para Smalltalk y CLOS, el costo por iteración (en términos de recompilación y reevaluación) es mayor para los programas C ++; así los programadores tienden a estar menos dispuestos a hacer estos pequeños cambios.

Muchas aplicaciones usan una base de datos. Cambios en la estructura de los objetos en un programa C ++ puede requerir que los cambios esquemáticos correspondientes sean hecho a la base de datos. (Muchas de las ideas que apliqué en mi refactorización el trabajo provino de la investigación en un esquema de base de datos orientado a objetos

Página 318

evolución.)

Otra limitación, que puede interesar a los investigadores de software más que muchos profesionales de software, es que C ++ no proporciona soporte para Análisis y cambio del programa metalevel. No hay nada análogo a El protocolo metaobjeto disponible para CLOS. El protocolo metaobjeto de CLOS, por ejemplo, admite una refactorización a veces útil para cambiar instancias seleccionadas de una clase en instancias de una clase diferente mientras que todas las referencias a los objetos antiguos apuntan automáticamente a nuevos objetos Afortunadamente, los casos en los que quería o necesitaba estos Las características eran pocas y distantes entre sí.

Comentarios de cierre

Las técnicas de refactorización pueden y han sido aplicadas a programas C ++ en un

variedad de contextos A menudo se espera que los programas C ++ evolucionen
varios años. Es durante ese proceso de evolución del software que el
Los beneficios de la refactorización pueden verse más fácilmente. El lenguaje proporciona
Algunas características que simplifican la refactorización, mientras que otras características del lenguaje
Si se aplica, la refactorización será más difícil. Afortunadamente, es ampliamente
reconoció que el uso de características del lenguaje como la aritmética de punteros es una
mala idea, por lo que la mayoría de los buenos programadores orientados a objetos evitan usarlos.

Muchas gracias a Ralph Johnson, Mick Murphy, James Roskind y
otros por presentarme algo del poder y la complejidad de C ++
con respecto a la refactorización.

Refactorización para lograr beneficios a corto plazo

Es relativamente fácil describir los beneficios de rango medio a largo de la refactorización. Muchas organizaciones,
sin embargo, son juzgados cada vez más por la comunidad inversora y por otros a corto plazo
actuación. ¿Puede la refactorización hacer una diferencia en el corto plazo?

La refactorización se ha aplicado con éxito durante más de diez años por objetos experimentados.
desarrolladores orientados Muchos de estos programadores se cortaron los dientes en una cultura Smalltalk que valoraba
claridad y simplicidad de código y reutilización aceptada. En tal cultura, los programadores invertirían
tiempo de refactorizar porque era lo correcto. El lenguaje Smalltalk y sus
Las implementaciones hicieron posible la refactorización de formas que no habían sido ciertas para la mayoría de los idiomas anteriores
y entornos de desarrollo de software. Gran parte de la programación inicial de Smalltalk se realizó en
grupos de investigación como Xerox, PARC o pequeños equipos de programación en empresas de vanguardia.
y firmas consultoras. Los valores de estos grupos eran algo diferentes de los valores de
muchos grupos de software industrial. Martin y yo somos conscientes de que para abrazar la refactorización
por la comunidad de desarrollo de software convencional, al menos algunos de sus beneficios deben estar cerca
término.

Nuestro equipo de investigación [3] , [9] , [12] , [13] , [14] , [15] ha descrito varios ejemplos de cómo
las refactorizaciones se pueden intercalar con extensiones a un programa de una manera que logre tanto
beneficios a largo y largo plazo. Uno de nuestros ejemplos es el marco del sistema de archivos Choices. Inicialmente
el marco implementó el formato de sistema de archivos Unix BSD (Berkeley Software Distribution).

Página 319

Más tarde se amplió para admitir archivos UNIX System V, MS-DOS, persistentes y distribuidos
sistemas. Los sistemas de archivos System V tienen muchas similitudes con los sistemas de archivos BSD UNIX. El enfoque
tomado por el desarrollador del framework fue el primero en clonar partes de la implementación de BSD Unix y luego
modificar el clon para admitir el Sistema V. La implementación resultante funcionó, pero hubo muchas
código duplicado Después de agregar el nuevo código, el desarrollador del marco refactorizó el código, creando
Superclases abstractas para contener el comportamiento común a los dos sistemas de archivos Unix
implementaciones Las variables y funciones comunes se trasladaron a superclases. En casos en
qué funciones correspondientes eran casi, pero no completamente idénticas, para los dos sistemas de archivos
implementaciones, se definieron nuevas funciones en cada subclase para contener las diferencias. En el
funciones originales esos segmentos de código fueron reemplazados con llamadas a las nuevas funciones. El código era
incrementalmente más similar en las dos subclases. Cuando las funciones eran idénticas, ellas
fueron trasladados a una superclase común.

Estas refactorizaciones brindan varios beneficios a corto y mediano plazo. En el corto plazo, errores
que se encuentra en la base de código común durante las pruebas, solo debía modificarse *en un lugar.* los
El tamaño general del código fue menor. El comportamiento específico de un formato de sistema de archivos particular fue limpio
separado del código común a los dos formatos del sistema de archivos. Esto facilitó el seguimiento
abajo y corregir comportamientos específicos de ese formato de sistema de archivos. A medio plazo, las abstracciones que
Los resultados de la refactorización a menudo fueron útiles para definir los sistemas de archivos posteriores. Por supuesto, el
El comportamiento común a los dos formatos de sistema de archivos existentes podría no ser completamente común para un tercero

formato, pero la base existente de código común fue un valioso punto de partida. Subsecuente
Se podrían aplicar refactorizaciones para aclarar lo que era realmente común. El desarrollo del marco
El equipo descubrió que, con el tiempo, se requería menos esfuerzo para agregar soporte incremental para un nuevo sistema de archivos
formato. Aunque los formatos más nuevos eran más complejos, el desarrollo se realizó por menos
personal experimentado.

Podría citar otros ejemplos de beneficios a corto y largo plazo de la refactorización, pero Martin tiene
Ya hecho esto. En lugar de agregar a su lista, permítanme hacer una analogía con algo que está cerca
y querido por muchos de nosotros, nuestra salud física.

En muchos sentidos, refactorizar es como hacer ejercicio y comer una dieta adecuada. Muchos de nosotros sabemos que nosotros
Debería hacer más ejercicio y comer una dieta más equilibrada. Algunos de nosotros vivimos en culturas que altamente
Fomentar estos hábitos. Algunos de nosotros podemos sobrevivir por un tiempo sin practicar estos buenos hábitos,
incluso sin efectos visibles. Siempre podemos poner excusas, pero solo nos estamos engañando a nosotros mismos si
Seguimos ignorando el buen comportamiento.

Algunos de nosotros estamos motivados por los beneficios a corto plazo del ejercicio y una dieta adecuada, como
altos niveles de energía, mayor flexibilidad, mayor autoestima y otros beneficios. Casi todos sabemos
que estos beneficios a corto plazo son *muy* reales. Muchos, pero no todos, hacemos al menos esfuerzos esporádicos en
estas areas. Otros, sin embargo, no están lo suficientemente motivados para hacer algo hasta que alcanzan un
punto de crisis

Sí, hay precauciones que deben aplicarse; las personas deben consultar con un experto antes
embarcarse en un programa. En el caso de ejercicio y dieta, deben consultar con su
médico. En el caso de refactorización, deben buscar recursos como este libro y
documentos citados en otra parte de este capítulo. El personal con experiencia en refactorización puede proporcionar más enfoque
asistencia.

Varias personas que he conocido son modelos a seguir con respecto a la aptitud física y la refactorización. Admiro su
energía y su productividad. Los modelos negativos muestran signos visibles de abandono. Su futuro y
El futuro de los sistemas de software que producen puede no ser optimista.

La refactorización puede lograr beneficios a corto plazo y hacer que el software sea más fácil de modificar y mantener.
Refactorizar es un medio más que un fin. Es parte de un contexto más amplio de cómo los programadores o
Los equipos de programación desarrollan y mantienen su software. [3]

Reducción de los gastos generales de refactorización

"La refactorización es una actividad general. Me pagan para escribir nuevas características que generan ingresos". Mi
La respuesta, en resumen, es esta:

- Λασ herramientas y tecnologías están disponibles para permitir que la refactorización se realice de manera rápida y relativamente sin dolor
- Λασ experiencias reportadas por algunos programadores orientados a objetos sugieren que la sobrecarga de refactorización está más que compensado por la reducción de esfuerzos e intervalos en otras fases de desarrollo de programas.
- Aunque la refactorización puede parecer un poco incómoda y un elemento general al principio, ya que se convierte en parte de un régimen de desarrollo de software, deja de sentirse como una sobrecarga y comienza a sentirse como un imprescindible.

Quizás la herramienta más madura para la refactorización automática ha sido desarrollada para Smalltalk por el
Equipo de Software Refactory en la Universidad de Illinois (ver Capítulo 14). Está disponible gratuitamente en

su sitio web (http://st-www.cs.vivc.edu). Aunque las herramientas de refactorización para otros idiomas son no tan fácilmente disponible, muchas de las técnicas descritas en nuestros documentos y en este libro pueden ser aplicado de una manera relativamente sencilla con un editor de texto o, mejor aún, un navegador. Software Los entornos de desarrollo y los navegadores han progresado sustancialmente en los últimos años. Esperamos para ver un conjunto creciente de herramientas de refactorización disponibles en el futuro.

Kent Beck y Ward Cunningham, ambos programadores experimentados de Smalltalk, han informado en Las conferencias de OOPSLA y otros foros que la refactorización les ha permitido desarrollar software rápidamente en dominios como el comercio de bonos. He escuchado testimonios similares de C ++ y CLOS desarrolladores En este libro, Martin describe los beneficios de refactorizar con respecto a Java programas Esperamos escuchar más testimonios de quienes leen este libro y aplican estos principios

Mi experiencia sugiere que a medida que la refactorización se convierte en parte de una rutina, deja de sentirse como gastos generales. Esta declaración es fácil de hacer pero difícil de justificar. Para los escépticos entre ustedes, mi consejo es simplemente hacerlo, luego decide por ti mismo. Sin embargo, dale tiempo.

Refactorización segura

La seguridad es una preocupación, especialmente para las organizaciones que desarrollan y desarrollan sistemas grandes. En muchos aplicaciones, existen importantes consideraciones financieras, legales y éticas para proporcionar Servicio continuo, confiable y sin errores. Muchas organizaciones brindan una amplia capacitación y intentan aplicar procesos de desarrollo disciplinados para ayudar a garantizar la seguridad de sus productos.

Sin embargo, para muchos programadores, la seguridad a menudo parece ser menos preocupante. Es mas que un poco Es irónico que muchos de nosotros prediquemos la seguridad primero a nuestros hijos, sobrinas y sobrinos, pero en nuestro papel de los programadores gritan por la libertad, un híbrido del pistolero y conductor adolescente del Salvaje Oeste. Dar libertad, danos los recursos y míranos volar. Después de todo, ¿realmente queremos que nuestra organización perder los frutos de nuestra creatividad simplemente por el bien de la repetibilidad y la conformidad?

En esta sección, analizo los enfoques para la refactorización segura. Me enfoco en un enfoque que compara con lo que Martin describe anteriormente en este libro es algo más estructurado y riguroso pero eso puede eliminar muchos errores que podrían introducirse en la refactorización.

Página 321

La seguridad es un concepto difícil de definir. Una definición intuitiva es que una refactorización segura es aquella que no rompe un programa Porque una refactorización está destinada a reestructurar un programa sin cambiando su comportamiento, un programa debe funcionar de la misma manera después de una refactorización como lo hace antes de.

¿Cómo se refactoriza de manera segura? Hay varias opciones:

- Confíe en sus habilidades de codificación.
- Confíe en que su compilador detectará los errores que omite.
- Confíe en que su conjunto de pruebas detectará los errores que usted y su compilador pierden.
- Confíe en que la revisión del código detectará los errores que usted, su compilador y su conjunto de pruebas omiten.

Martin se enfoca en las primeras tres opciones en su refactorización. Organizaciones medianas a grandes a menudo Complemente estos pasos con revisiones de código.

Mientras que los compiladores, los conjuntos de pruebas, las revisiones de código y los estilos de codificación disciplinados son valiosos, existe son límites a todos estos enfoques, como sigue:

- Λοσ programadores son falibles, incluso tú (sé que lo soy).
- Hay errores sutiles y no tan sutiles que los compiladores no pueden detectar, especialmente el alcance errores relacionados con la herencia. [1]

- Perry y Kaiser [16] y otros han demostrado que aunque es o al menos solía ser
  Es sabido que la tarea de prueba se simplifica cuando la herencia se utiliza como
  técnica de implementación, en realidad a menudo se necesita un amplio conjunto de pruebas para cubrir todo
  los casos en que las operaciones que solían solicitarse en una instancia de clase ahora son
  solicitado en instancias de sus subclases. A menos que su diseñador de prueba sea omnisciente o pague
  gran atención al detalle, es probable que haya casos que su conjunto de pruebas no cubrirá. Probar todo
  Las posibles rutas de ejecución en un programa son un problema computacionalmente indecidible. En otra
  Es decir, no se puede garantizar que haya detectado todos los casos con su conjunto de pruebas.
- Λοσ revisores de códigos, como los programadores, son falibles. Además, los revisores pueden estar demasiado ocupados
  con su trabajo principal para revisar a fondo el código de otra persona.

Otro enfoque, que tomé en mi investigación, es definir y crear un prototipo de una herramienta de refactorización para compruebe si una refactorización se puede aplicar de forma segura a un programa y, si es así, refactorice el programa. Esto evita muchos de los errores que pueden introducirse por error humano.

Aquí proporciono una descripción de alto nivel de mi enfoque para la refactorización segura. Esto puede ser lo más parte valiosa de este capítulo. Para más detalles, vea mi disertación [1] y otras referencias en el final de este capitulo; Ver también el Capítulo 14 . Si considera que esta sección es demasiado técnica, lea adelante a los últimos párrafos de esta sección.

Parte de mi herramienta de refactorización es un analizador de programas, que es un programa que analiza la estructura. de otro programa (en este caso, un programa C ++ al que se podría aplicar una refactorización). los La herramienta puede responder una serie de preguntas sobre el alcance, la tipificación y la semántica del programa (el significado u operaciones previstas de un programa). Los problemas de alcance relacionados con la herencia hacen que esto análisis más complejo que con muchos programas no orientados a objetos, pero para C ++, lenguaje características como la escritura estática hacen que el análisis sea más fácil que, por ejemplo, Smalltalk.

Considere, por ejemplo, la refactorización para eliminar una variable de un programa. Una herramienta puede determinar qué otras partes de un programa (si las hay) hacen referencia a la variable. Si hay alguna referencia, eliminar la variable dejaría referencias colgantes; Por lo tanto, esta refactorización no sería segura. UNA El usuario que solicite a la herramienta que refactorice el programa recibirá un indicador de error. El usuario podría entonces Decidir que la refactorización es una mala idea después de todo o cambiar las partes del programa que se refieren

Página 322

a esa variable y aplique la refactorización para eliminar la variable. Hay muchos otros controles, tan simple como esto, algunos más complejos.

En mi investigación, definí la seguridad en términos de propiedades del programa (relacionadas con actividades tales como alcance y mecanografía) que deben continuar reteniéndose después de refactorizar. Muchos de estos programas las propiedades son similares a las restricciones de integridad que deben mantenerse cuando los esquemas de la base de datos cambio. [17] Cada refactorización tiene asociada un conjunto de condiciones previas necesarias que si es cierto aseguraría que se preservan las propiedades del programa. Solo si la herramienta determinara que todo está seguro si la herramienta realizara la refactorización.

Afortunadamente, determinar si una refactorización es segura a menudo es trivial, especialmente para los de bajo nivel. refactorizaciones que constituyen la mayor parte de nuestra refactorización. Para garantizar que el nivel superior, más las refactorizaciones complicadas son seguras, las definimos en términos de refactorizaciones de bajo nivel. por Por ejemplo, la refactorización para crear una superclase abstracta se define en términos de pasos, que son Refactorizaciones más simples como crear y mover variables y métodos. Al mostrar que cada el paso de una refactorización más complicada es seguro, podemos saber por construcción que la refactorización es seguro.

Hay algunos casos (relativamente raros) en los que una refactorización podría ser segura de aplicar a un programa pero una herramienta no puede estar seguro. En estos casos, la herramienta toma la ruta segura y no permite refactorización. Por ejemplo, considere nuevamente el caso en el que desea eliminar una variable de un programa, pero hay una referencia a él en otro lugar del programa. Quizás la referencia es contenido en un segmento de código que nunca se ejecutará. Por ejemplo, la referencia puede aparecer

dentro de un condicional, como un bucle if-then, que nunca será verdadero. Si puede estar seguro de que el condicional nunca probaría verdadero, podría eliminar la prueba condicional, incluido el código refiriéndose a la variable o función que desea eliminar. A continuación, puede eliminar con seguridad el variable o función. En general, no es posible saber con certeza si la afección siempre se falso. (Suponga que heredó el código que fue desarrollado por otra persona. Cómo ¿seguro de que eliminarías este código?)

Una herramienta de refactorización puede marcar la referencia y alertar al usuario. El usuario puede decidir dejar el código solo Si o cuando el usuario se aseguró de que el código de referencia nunca se ejecutaría, él o ella podría eliminar el código y aplicar la refactorización. La herramienta informa al usuario de implicaciones de la referencia en lugar de aplicar ciegamente el cambio.

Esto puede sonar como algo complicado. Está bien para una disertación doctoral (la audiencia principal, el comité de tesis, quiere ver un poco de atención a los problemas teóricos), pero ¿es práctico para refactorización?

Toda la verificación de seguridad se puede implementar bajo el capó de una herramienta de refactorización. Un programador quien quiera refactorizar un programa simplemente necesita pedirle a la herramienta que verifique el código y, si es seguro, realizar la refactorización. Mi herramienta fue un prototipo de investigación. Don Roberts, John Brant, Ralph Johnson y yo [10] hemos implementado una herramienta mucho más robusta y destacada (ver Capítulo 14) como parte de nuestra investigación sobre la refactorización de los programas Smalltalk.

Se pueden aplicar muchos niveles de seguridad a la refactorización. Algunos son fáciles de aplicar pero no garantizan un Alto nivel de seguridad. El uso de una herramienta de refactorización puede proporcionar muchos beneficios. Puede hacer muchos simples pero cheques tediosos y señalan de antemano problemas que si no se controlan causarían el programa para romper como resultado de la refactorización.

Aunque la aplicación de una herramienta de refactorización evita la introducción de muchos de los errores que de otra manera la esperanza se marcará durante la compilación, las pruebas y la revisión del código, las últimas técnicas aún son de valor, particularmente en el desarrollo o evolución de sistemas en tiempo real. Los programas a menudo no ejecutar de forma aislada; son parte de una red más grande de sistemas de comunicación. Algunos

Página 323

Las refactorizaciones no solo limpian el código sino que también hacen que un programa se ejecute más rápidamente. Acelerando uno El programa podría generar cuellos de botella en el rendimiento en otros lugares. Esto es similar a los efectos de Actualización de microprocesadores que aceleran partes de un sistema y requieren enfoques similares a ajustar y probar el rendimiento general del sistema. Por el contrario, algunas refactorizaciones pueden disminuir en general rendimiento un poco, pero en general tales efectos sobre el rendimiento son mínimos.

Los enfoques de seguridad están destinados a garantizar que la refactorización no introduzca *nuevos* errores en un programa. Estos enfoques no detectan ni corrigen errores que estaban en el programa antes de que fuera refactorizado. Sin embargo, la refactorización puede hacer que sea más fácil detectar esos errores y corregirlos.

Un control de realidad (revisitado)

Hacer que la refactorización sea real requiere abordar las preocupaciones del mundo real de los profesionales del software. Cuatro preocupaciones comúnmente expresadas son las siguientes:

- Los programadores pueden no entender cómo refactorizar.
- Si los beneficios son a largo plazo, ¿por qué hacer el esfuerzo ahora? A largo plazo, es posible que no seas con el proyecto para cosechar los beneficios.
- Ελ código de refactorización es una actividad general; a los programadores se les paga para escribir *nuevas* funciones.
- Λα refactorización puede romper el programa existente.

En este capítulo, abordo brevemente cada una de estas preocupaciones y proporciono consejos para aquellos que desean profundizar más en estos temas.

Los siguientes problemas son de interés para algunos proyectos:

- ¿Qué sucede si el código a refactorizar es propiedad colectiva de varios programadores? En algunos
  En algunos casos, muchos de los mecanismos tradicionales de gestión del cambio son relevantes. En otra casos, si el software ha sido bien diseñado y refactorizado, los subsistemas serán suficientemente desacoplado para que muchas refactorizaciones afecten solo un pequeño subconjunto del código base.
- ¿Qué sucede si hay varias versiones o líneas de código de una base de código? En algunos casos,
  la refactorización puede ser relevante para todas las versiones, en cuyo caso todas deben ser verificadas por seguridad antes de aplicar la refactorización. En otros casos, las refactorizaciones pueden ser relevante solo para algunas versiones, lo que simplifica el proceso de verificación y refactorización el código. La gestión de cambios en varias versiones a menudo requiere la aplicación de muchas de las Técnicas tradicionales de gestión de versiones. La refactorización puede ser útil para fusionar variantes o versiones en una base de código actualizada, lo que puede simplificar la gestión de versiones río abajo.

En resumen, convencer a los profesionales de software del valor práctico de la refactorización es bastante diferente de persuadir a un comité de doctorado de que la refactorización de la investigación es digna de un Ph.D. Eso Me tomó un tiempo después de completar mis estudios de posgrado para apreciar plenamente estas diferencias.

Recursos y referencias para refactorización

En este punto del libro, espero que esté planeando aplicar técnicas de refactorización en su trabajo y están alentando a otros en su organización a hacerlo. Si aún no está decidido, es posible que desee consulte las referencias que he proporcionado o comuníquese con Martin (Fowler@acm.org), yo u otras personas que tienen experiencia en refactorización.

Página 324

Si desea explorar más la refactorización, aquí hay algunas referencias que puede consultar fuera. Como Martin ha señalado, este libro no es el primer trabajo escrito sobre refactorización, pero (espero) lo hará exponer a un público cada vez más amplio a los conceptos y beneficios de la refactorización. Aunque mi doctorado disertación fue el primer trabajo escrito importante sobre el tema, la mayoría de los lectores interesados en explorar el Los primeros trabajos fundamentales sobre la refactorización probablemente deberían examinar primero varios documentos. [3] , [9] , [12] , [13] La refactorización fue un tema tutorial en OOPSLA 95 y OOPSLA 96. [14] , [15] Para aquellos con interés en los patrones de diseño y la refactorización, el documento "Ciclo de vida y patrones de refactorización That Support Evolution and Reuse ", [3] que Brian Foote y yo presentamos en PLoP '94 y que aparece en el primer volumen de Addison-Wesley Pattern Languages of Program Design serie, es un buen lugar para comenzar. Mi investigación de refactorización se basó principalmente en el trabajo de Ralph Johnson y Brian sobre marcos de aplicaciones orientadas a objetos y el diseño de reutilizables clases [4] Investigación de refactorización posterior realizada por John Brant, Don Roberts y Ralph Johnson en La Universidad de Illinois se ha centrado en la refactorización de los programas Smalltalk. [10] , [11] Su web El sitio (http://st-www.cs.uiuc.edu) incluye algunos de sus trabajos más recientes. Interés en La refactorización ha crecido dentro de la comunidad de investigación orientada a objetos. Varios documentos relacionados fueron presentados en OOPSLA 96 en una sesión titulada Refactoring and Reuse [18] .

Implicaciones sobre la reutilización de software y la transferencia de tecnología

Las preocupaciones del mundo real abordadas anteriormente no se aplican solo a la refactorización. Se aplican más en términos generales a la evolución y reutilización de software.

Durante gran parte de los últimos años, me he centrado en cuestiones relacionadas con la reutilización de software, plataformas, marcos, patrones y la evolución de sistemas heredados, a menudo involucrando software que no era

orientado a objetos. Además de trabajar con proyectos dentro de Lucent y Bell Labs, tengo participó en foros con personal de otras organizaciones que han estado lidiando con similares cuestiones. [19] , [20] , [21] , [22]

Las preocupaciones del mundo real con respecto a un programa de reutilización son similares a las relacionadas con la refactorización.

- El personal técnico puede no entender qué reutilizar o cómo reutilizarlo.
- El personal técnico puede no estar motivado para aplicar un enfoque de reutilización a menos que sea a corto plazo
  Se pueden lograr beneficios.
- Los problemas de gastos generales, la curva de aprendizaje y los costos de descubrimiento deben abordarse para su reutilización
  enfoque para ser adoptado con éxito.
- Adoptar un enfoque de reutilización no debe ser perjudicial para un proyecto; puede haber fuerte
  presiones para aprovechar los activos existentes o la implementación, aunque con restricciones heredadas.
  Las nuevas implementaciones deberían interfuncionar o ser compatibles con los sistemas existentes.

Geoffrey Moore [23] describió el proceso de adopción de tecnología en términos de una curva en forma de campana en el que la cola delantera incluye innovadores y primeros usuarios, la gran joroba media incluye mayoría temprana y mayoría tardía, y la cola posterior incluye rezagados. Para una idea y producto para Para tener éxito, deben ser adoptados por las mayorías temprana y tardía. Dicho de otra manera, muchos las ideas que atraen a los innovadores y a los primeros usuarios finalmente fracasan porque nunca lo logran a través del abismo a las mayorías tempranas y tardías. La desconexión radica principalmente en los diferentes motivadores de estos grupos de clientes. Los innovadores y los primeros usuarios se sienten atraídos por los nuevos tecnologías, visiones de cambios de paradigma y avances. Las mayorías temprana y tardía son preocupado principalmente por la madurez, el costo, el soporte y ver si la nueva idea o producto tiene sido aplicado con éxito por otros con necesidades similares a las de ellos.

Los profesionales de desarrollo de software están impresionados y convencidos de maneras muy diferentes a las que están investigadores de software. Los investigadores de software son a menudo lo que Moore se refiere como innovadores. Los desarrolladores de software y especialmente los gerentes de software a menudo son parte de los principios y tardíos

Page 325

mayorías Reconocer estas diferencias es importante para llegar a cada uno de estos grupos. Con Reutilización del software, al igual que con la refactorización, es importante llegar a los profesionales de desarrollo de software. en sus términos

Dentro de Lucent / Bell Labs descubrí que se requería una aplicación alentadora de reutilización y plataformas llegando a una variedad de partes interesadas. Se requería formular una estrategia con los ejecutivos, organizar reuniones del equipo de liderazgo entre gerentes intermedios, consultoría con proyectos de desarrollo y Dar a conocer los beneficios de estas tecnologías a un público amplio de investigación y desarrollo. a través de seminarios y publicaciones. En todo momento fue importante capacitar al personal en los principios, abordar los beneficios a corto plazo, proporcionar formas de reducir los gastos generales y abordar cómo estos Las técnicas podrían introducirse de forma segura. Obtuve estos conocimientos de mi investigación de refactorización.

Como Ralph Johnson, quien era mi asesor de tesis, señaló al revisar un borrador de este capítulo: estos principios no se aplican solo a la refactorización y la reutilización de software; son cuestiones genéricas de Transferencia tecnológica. Si te encuentras tratando de persuadir a otras personas para que refactoricen (o adopten otra tecnología o práctica), asegúrese de enfocarse en estos temas y llegar a las personas Dónde están. La transferencia de tecnología es difícil, pero se puede hacer.

Una nota final

Gracias por tomarse el tiempo de leer este capítulo. He tratado de abordar muchas de las preocupaciones que es posible que tenga sobre la refactorización y trató de demostrar que muchas de las preocupaciones del mundo real con respecto a la refactorización se aplica más ampliamente a la evolución y reutilización del software. Espero que hayas venido Le entusiasma la aplicación de estas ideas en su trabajo. Los mejores deseos a medida que avanzas sus tareas de desarrollo de software

Notas finales

1. Opdyke, William F. "Refactorización de marcos orientados a objetos". Ph.D. diss., Universidad de Illinois en Urbana-Champaign. También disponible como Informe técnico UIUCDCS-R-92-1759, Departamento de Ciencias de la Computación, Universidad de Illinois en Urbana-Champaign.

2. Brooks, Fred. "No Silver Bullet: esencia y accidentes de la ingeniería de software". *Procesamiento de información 1986: Actas de la Décima Computación Mundial IFIP Conferencia* , editada por H.-L.Kugler. Amsterdam: Elsevier, 1986.

3. Foote, Brian y William F. Opdyke. "Ciclo de vida y patrones de refactorización que admiten Evolution and Reuse ". En *Pattern Languages of Program Design* , editado por J. Coplien y D. Schmidt. Lectura, Misa: Addison-Wesley, 1995.

4. Johnson, Ralph E. y Brian Foote. "Diseño de clases reutilizables". *Journal of Object-Programación orientada* 1 (1988): 22-35.

5. Rochat, Roxanna. "En busca del buen estilo de programación de Smalltalk". Informe técnico CR-86-19, Tektronix, 1986.

6. Lieberherr, Karl J. e Ian M. Holland. "Asegurar un buen estilo para orientado a objetos Programas ". *IEEE Software* (septiembre de 1989) 38-48.

7. Wirfs-Brock, Rebecca, Brian Wilkerson y Luaren Wiener. *Diseño orientado a objetos Software* . Upper Saddle River, Nueva Jersey: Prentice Hall, 1990.

Página 326

8. Gamma, Erich, Richard Helm, Ralph Johnson y John Vlissides. *Patrones de diseño: Elementos del software orientado a objetos reutilizables* . Lectura, misa .: Addison-Wesley, 1985.

9. Opdyke, William F. y Ralph E. Johnson. "Creación de superclases abstractas por Refactorización. ”En *Actas de CSC '93: La Conferencia de Ciencias de la Computación ACM 1993* . 1993.

10. Roberts, Don, John Brant, Ralph Johnson y William Opdyke. Refactoring Tool. ”En *Actas de ICAST 96: 12ª Conferencia Internacional sobre Ciencia y tecnología avanzada* . 1996.

11. Roberts, Don, John Brant y Ralph E. Johnson. "Una herramienta de refactorización para Smalltalk". *TAPOS* 3 (1997) 39-42.

12. Opdyke, William F. y Ralph E. Johnson. "Refactorización: una ayuda en el diseño de aplicaciones Marcos y sistemas orientados a objetos en evolución ”. En las *actas de SOOPPA '90: Simposio sobre programación orientada a objetos que enfatiza aplicaciones prácticas* . 1990

13. Johnson, Ralph E. y William F. Opdyke. "Refactorización y agregación". En *Actas de ISOTAS '93: Simposio internacional sobre tecnologías de objetos para El software avanzado* . 1993.

14. Opdyke, William y Don Roberts. "Refactorización". Tutorial presentado en OOPSLA 95: 10th Conferencia anual sobre sistemas de programas orientados a objetos, lenguajes y aplicaciones, Austin, Texas, octubre de 1995.

15. Opdyke, William y Don Roberts. "Refactorización de software orientado a objetos para soportar Evolution and Reuse. ”Tutorial presentado en OOPSLA 96: 11ª Conferencia Anual sobre Sistemas de programas orientados a objetos, lenguajes y aplicaciones, San José, California, Octubre de 1996.

16. Perry, Dewayne E. y Gail E. Kaiser. "Pruebas adecuadas y orientadas a objetos Programación ". *Revista de Programación Orientada a Objetos* (1990).

17. Banerjee, Jay y Won Kim. "Semántica e implementación de la evolución del esquema en Bases de datos orientadas a objetos ". En *Actas de la Conferencia ACM SIGMOD* , 1987.

18. Actas de OOPSLA 96: Conferencia sobre sistemas de programación orientada a objetos, Idiomas y aplicaciones, San José, California, octubre de 1996.

19. Informe sobre WISR '97: Octavo taller anual sobre reutilización de software, Columbus, Ohio, Marzo de 1997. *Notas de ingeniería de software de ACM* . (1997)

20. Beck, Kent, Grady Booch, Jim Coplien, Ralph Johnson y Bill Opdyke. "Más allá de Exageración: ¿los patrones y marcos reducen los costos de descubrimiento? OOPSLA 97: 12ª Conferencia Anual sobre Sistemas de Programas Orientados a Objetos, Idiomas y Aplicaciones, Atlanta, Georgia, octubre de 1997.

21. Kane, David, William Opdyke y David Dikel. "Gestión del cambio a reutilizable Software. ”Documento presentado en PLoP 97: 4ta Conferencia Anual sobre el Patrón Idiomas de los programas, Monticello, Illinois, septiembre de 1997.

Página 327

22. Davis, Maggie, Martin L. Griss, Luke Hohmann, Ian Hopper, Rebecca Joos y William F. Opdyke. Sesión de panel “Reutilización de software: ¿Némesis o Nirvana?” En OOPSLA 98: 13 Conferencia anual sobre sistemas de programas orientados a objetos, lenguajes y aplicaciones, Vancouver, Columbia Británica, Canadá, octubre de 1998.

23. Geoffrey A.Moore, *Cross the Chasm: Marketing y venta de productos tecnológicos para Clientes principales* . Nueva York: HarperBusiness, 1991.

Página 328

Capítulo 14. Herramientas de refactorización

por Don Roberts y John Brant

Una de las mayores barreras para la refactorización del código ha sido la lamentable falta de soporte de herramientas. Los idiomas en los que la refactorización es parte de la cultura, como Smalltalk, generalmente tienen poderosos entornos que admiten muchas de las características necesarias para refactorizar el código. Incluso allí, el el proceso solo se ha respaldado parcialmente hasta hace poco, y la mayor parte del trabajo todavía se está realizando a mano.

Refactorizando con una herramienta

La refactorización con soporte automatizado de herramientas se siente diferente de la refactorización manual. Incluso con el red de seguridad de un conjunto de pruebas en su lugar, la refactorización manual lleva mucho tiempo. Este simple hecho evita que los programadores realicen refactorizaciones que saben que deberían, simplemente porque refactorizar cuesta demasiado. Al hacer que la refactorización sea tan económica como ajustar el formato del código, El trabajo de limpieza se puede hacer de manera similar a limpiar el aspecto del código. Sin embargo, este tipo de limpieza puede tener un profundo efecto positivo en la mantenibilidad, la reutilización y Comprensibilidad del código. Kent Beck dice:

Kent Beck

[El navegador de refactorización] cambia por completo tu forma de pensar programación. Todos esos pequeños y molestos "bueno, debería cambiar este nombre pero ... "los pensamientos desaparecen, porque solo cambias el nombre porque hay siempre es un elemento de menú único para simplemente cambiar el nombre.

Cuando comencé a usar esta herramienta, pasé unas dos horas refactorizando mi viejo ritmo Haría una refactorización, luego simplemente miraría al espacio durante los cinco minutos me hubiera llevado a hacer la refactorización a mano, luego haz otra y mira al espacio otra vez. Después de un rato, me sorprendí y me di cuenta de que tenía que aprender a pensar pensamientos refactorizadores más grandes y Piense en ellos más rápido. Ahora uso probablemente la mitad de refactorización y la mitad de entrada nuevo código, todo a la misma velocidad.

Con este nivel de soporte de herramientas para la refactorización, se convierte cada vez menos en una actividad separada de programación. Muy raramente decimos: "Ahora estoy programando" y "Ahora estoy refactorizando". Fueron es más probable que diga: "Extraiga esta parte del método, empújela hacia la superclase y luego agregue una llamada al nuevo método en la nueva subclase en la que estoy trabajando ". Porque no tengo que probar después de refactorizaciones automatizadas, las actividades fluyen entre sí y el proceso de cambio de sombreros se vuelve mucho menos evidente, aunque todavía está ocurriendo.

Considere el método de extracción , una refactorización importante. Tienes muchas cosas para verificar cuando lo haces a mano Con el navegador de refactorización, simplemente selecciona el texto para extraer y encontrar el elemento de menú denominado Método de extracción. La herramienta determina si el texto seleccionado es legal para extraer. Hay varias razones por las cuales la selección podría ser ilegal. Podría contener solo una parte de un identificador, o podría contener asignaciones a una variable sin contener todas las referencias. No tiene que preocuparse por todos los casos, porque la herramienta se ocupa de ellos. La herramienta entonces calcula cuántos parámetros se deben pasar al nuevo método. Luego te pide un nombre para el nuevo método y le permite especificar el orden de los parámetros en la llamada al

Page 329

nueva función Una vez hecho esto, la herramienta extrae el código del método original y reemplaza con una llamada Luego crea el nuevo método en la misma clase que el método original y con el nombre que especificó el usuario. Todo el proceso dura unos 15 segundos. Compare esto con El tiempo necesario para realizar los pasos del Método de extracción .

A medida que la refactorización se vuelve menos costosa, los errores de diseño se vuelven menos costosos. Porque es menos costoso de corregir errores de diseño, se necesita hacer menos diseño por adelantado. El diseño inicial es un actividad predictiva porque los requisitos serán incompletos. Porque el código no es disponible, la forma correcta de diseñar para simplificar el código no es obvia. En el pasado, tuvimos que vivir con cualquier diseño que creamos inicialmente porque el costo de cambiar el diseño era demasiado alto. Con las herramientas de refactorización automática, podemos permitir que el diseño sea más fluido porque cambiarlo es Mucho menos costoso. Dado este nuevo conjunto de costos, podemos diseñar al nivel del problema actual sabiendo que podemos ampliar el diseño a bajo costo para agregar flexibilidad adicional en el futuro. No ya no tenemos que intentar predecir todas las formas posibles en que el sistema podría cambiar en el futuro. Si encontramos que el diseño actual hace que el código sea incómodo con los olores descritos en Capítulo 3 , podemos cambiar rápidamente el diseño para que el código sea limpio y fácil de mantener.

La refactorización asistida por herramientas afecta las pruebas. Mucho menos pruebas tienen que ocurrir porque muchas de las Las refactorizaciones se realizan automáticamente. Siempre habrá refactorizaciones que no pueden ser automatizado, por lo que el paso de prueba nunca se eliminará. La observación empírica ha sido que las pruebas se ejecutan la misma cantidad de veces por día que en entornos sin pruebas automáticas herramientas pero que se realiza una mayor refactorización.

Como Martin ha señalado, Java necesita herramientas para soportar este tipo de comportamiento de los programadores. Nosotros quiero señalar algunos de los criterios que dicha herramienta debe tener para tener éxito. Aunque nosotros Hemos incluido los criterios técnicos, creemos que los criterios prácticos son mucho más importante.

Criterios técnicos para una herramienta de refactorización

El objetivo principal de una herramienta de refactorización es permitir que el programador refactorice el código sin tener volver a probar el programa. Las pruebas requieren mucho tiempo incluso cuando están automatizadas, y eliminarlas puede acelerar el proceso de refactorización por un factor significativo. Esta sección discute brevemente el requisitos técnicos para una herramienta de refactorización que son necesarios para permitirle transformar un programa mientras se preserva el comportamiento del programa.

Base de datos del programa

Uno de los primeros requisitos reconocidos fue la capacidad de buscar varias entidades del programa. en todo el programa, como con un método particular, encontrar todas las llamadas que potencialmente pueden consulte el método en cuestión, o con una variable de instancia particular, para encontrar todos los métodos que lo lean o lo escriban. En entornos estrechamente integrados como Smalltalk, esta información es mantenido constantemente en una forma de búsqueda. Esta no es una base de datos como se entiende tradicionalmente, pero *es* un repositorio de búsqueda. El programador puede realizar una búsqueda para encontrar referencias cruzadas a cualquier elemento del programa, principalmente debido a la compilación dinámica del código. Tan pronto āsa el cambio se realiza en cualquier clase, el cambio se compila inmediatamente en bytecodes y el "base de datos" se actualiza. En entornos más estáticos como Java, los programadores ingresan el código en archivos de texto Las actualizaciones de la base de datos deben realizarse ejecutando un programa para procesarlas. archivos y extraer la información relevante. Estas actualizaciones son similares a la compilación de Java código en sí mismo. Algunos de los entornos más modernos, como IBM VisualAge para Java, imitan Actualización dinámica de Smalltalk de la base de datos del programa.

Un enfoque ingenuo es utilizar herramientas textuales como grep para hacer la búsqueda. Este enfoque se rompe hacia abajo rápidamente porque no puede diferenciar entre una variable llamada *foo* y una función llamada *foo* La creación de una base de datos requiere el uso de análisis semántico (análisis) para determinar la "parte de discurso "de cada token en el programa. Esto debe hacerse tanto en el nivel de definición de clase, para determinar definiciones de variables y métodos de instancia, y a nivel de método, para determinar Variables de instancia y referencias de métodos.

Árboles Parse

La mayoría de las refactorizaciones tienen que manipular partes del sistema por debajo del nivel del método. Estos son usualmente referencias a elementos del programa que están siendo cambiados. Por ejemplo, si una instancia se cambia el nombre de la variable (simplemente un cambio de definición), todas las referencias dentro de los métodos de esa clase y sus subclases deben actualizarse. Otras refactorizaciones están completamente por debajo del nivel del método, como como extraer una parte de un método en su propio método independiente. Cualquier actualización de un método tiene para poder manipular la estructura del método. Para hacer esto se requieren árboles de análisis. Un árbol de análisis es una estructura de datos que representa la estructura interna del método en sí. Como un simple ejemplo, considere el siguiente método:

```
public void hello () {
   System.out.println ("Hola Mundo");
}
```

El árbol de análisis correspondiente a esto se vería como en la Figura 14.1 .

Figura 14.1. Árbol de Parse para hola mundo

Exactitud

Página 331

Las refactorizaciones implementadas por una herramienta deben preservar razonablemente el comportamiento de los programas. La preservación total del comportamiento es imposible de lograr. Por ejemplo, ¿qué pasa si una refactorización hace Es un programa unos milisegundos más rápido o más lento? Esto generalmente no afectaría un programa, pero si el los requisitos del programa incluyen restricciones duras en tiempo real, esto podría causar que un programa sea incorrecto.

Incluso los programas más tradicionales pueden romperse. Por ejemplo, si su programa construye una cadena y utiliza la API de Java Reflection para ejecutar el método que los nombres de cadena, renombrando El método hará que el programa arroje una excepción que el original no hizo.

Sin embargo, las refactorizaciones pueden hacerse razonablemente precisas para la mayoría de los programas. Siempre y cuando el se identifican casos que romperán una refactorización, los programadores que usan esas técnicas pueden evite la refactorización o arregle manualmente las partes del programa que la herramienta de refactorización No se puede arreglar.

Criterios prácticos para una herramienta de refactorización

Las herramientas se crean para apoyar a un humano en una tarea particular. Si una herramienta no se ajusta a la forma en que una persona funciona, la persona no lo usará. Los criterios más importantes son los que integran el proceso de refactorización con otras herramientas.

Velocidad

El análisis y las transformaciones necesarios para realizar refactorizaciones pueden llevar mucho tiempo si Son muy sofisticados. Los costos relativos de tiempo y precisión siempre deben considerarse. Si un la refactorización lleva demasiado tiempo, un programador nunca usará la refactorización automática, sino que simplemente realizarlo a mano y vivir con las consecuencias. La velocidad siempre debe ser considerada. En el proceso de desarrollo del navegador de refactorización, tuvimos algunas refactorizaciones que no tuvimos implementar simplemente porque no pudimos implementarlos de manera segura en un período de tiempo razonable. Sin embargo, hicimos un trabajo decente, y la mayoría de las refactorizaciones son extremadamente rápidas y muy precisas. Los informáticos tienden a centrarse en todos los casos límite que un enfoque particular no encargarse de. El hecho es que la mayoría de los programas no son casos límite y que son más simples y rápidos Los enfoques funcionan sorprendentemente bien.

Un enfoque para considerar si un análisis sería demasiado lento es simplemente pedirle al programador que proporcionar la información Esto pone la responsabilidad de la precisión nuevamente en manos del programador mientras permite que el análisis se realice rápidamente. Muy a menudo el programador conoce la información que se requiere. Aunque este enfoque no es demostrablemente seguro porque el programador puede cometer errores, la responsabilidad del error recae en los programadores. Irónicamente, esto hace que los programadores sean más propensos a usar la herramienta porque no están obligados a confíe en la heurística de un programa para encontrar información.

Deshacer

La refactorización automática permite un enfoque exploratorio del diseño. Puedes empujar el código y vea cómo se ve bajo el nuevo diseño. Porque se supone que una refactorización es comportamiento preservando, la refactorización inversa, que deshace el original, también es una refactorización y es Preservar el comportamiento. Las versiones anteriores del navegador de refactorización no incorporaban el deshacer característica. Esto hizo que la refactorización sea un poco más tentativa porque deshacer algunas refactorizaciones, aunque preservar el comportamiento, fue difícil. Muy a menudo tendríamos que encontrar una versión antigua de el programa y comenzar de nuevo. Esto fue molesto. Con la adición de deshacer, otro grillete fue arrojado. Ahora podemos explorar con impunidad, sabiendo que podemos volver a cualquier versión anterior.

Página 332

Podemos crear clases, mover métodos a ellas para ver cómo se verá el código y cambiar nuestro mentes e ir en una dirección completamente diferente, todo muy rápido.

Integrado con herramientas

En la última década, el entorno de desarrollo integrado (IDE) ha sido el núcleo de la mayoría proyectos de desarrollo. El IDE integra el editor, compilador, enlazador, depurador y cualquier otro herramienta necesaria para desarrollar programas. Una implementación temprana del navegador de refactorización para Smalltalk era una herramienta separada de las herramientas de desarrollo estándar de Smalltalk. Lo que encontramos fue que nadie lo usó Ni siquiera lo usamos nosotros mismos. Una vez que integramos las refactorizaciones directamente en el navegador Smalltalk, los usamos ampliamente. Simplemente tenerlos a nuestro alcance. Toda la diferencia.

Envolver

Llevamos varios años desarrollando y utilizando el navegador de refactorización. Es bastante común para que lo usemos para refactorizar su propio código. Una de las razones de su éxito es que somos programadores y hemos tratado constantemente de que se ajuste a nuestra forma de trabajar. Si nos topamos con un refactorizando que teníamos que realizar a mano y sentimos que era general, lo implementaríamos y agrégalo Si algo llevara demasiado tiempo, lo haríamos más rápido. Si algo no fuera lo suficientemente preciso, Lo mejoraríamos.

Creemos que las herramientas de refactorización automática son la mejor manera de gestionar la complejidad que surge a medida que evoluciona un proyecto de software. Sin herramientas para hacer frente a esta complejidad, el software se convierte en hinchado, con errores y quebradizo. Porque Java es mucho más simple que el lenguaje con el que comparte sintaxis, es mucho más fácil desarrollar herramientas para refactorizarlo. Esperamos que esto ocurra y que podamos evitar los pecados de C ++.

Página 333

Capítulo 15. Poniendo todo junto

por Kent Beck

Ahora tienes todas las piezas del rompecabezas. Has aprendido las refactorizaciones. Has estudiado el catalogar. Has practicado todas las listas de verificación. Te has vuelto bueno en las pruebas, así que no tienes miedo. Ahora puede pensar que sabe cómo refactorizar. Aún no.

La lista de técnicas es solo el comienzo. Es la puerta por la que debes pasar. Sin el técnicas, no se puede manipular el diseño de los programas en ejecución. Con ellos, todavía no puedes, pero en Al menos puedes empezar.

¿Por qué todas estas maravillosas técnicas son solo el comienzo? Porque aún no lo sabes cuándo usarlos y cuándo no, cuándo comenzar y cuándo parar, cuándo ir y cuándo esperar. Es el ritmo lo que hace la refactorización, no las notas individuales.

¿Cómo sabrá cuándo realmente lo está recibiendo? Lo sabrás cuando comiences a calmarte. Cuando usted siente absoluta confianza de que no importa cuán jodido alguien lo haya dejado, puede hacer que el codificar mejor, lo suficientemente mejor para seguir progresando.

Sin embargo, en su mayoría, sabrá que lo está recibiendo cuando pueda detenerse con confianza. Parar es el movimiento más fuerte en el repertorio del refactorizador. Ves un gran objetivo: una gran cantidad de subclases pueden ser eliminado Empiezas a moverte hacia esa meta, cada paso es pequeño y seguro, cada paso retrocede manteniendo todas las pruebas en ejecución. Te estás acercando. Solo tienes dos métodos para unificar en cada de las subclases, y luego pueden desaparecer.

Ahí es cuando sucede. Te quedas sin gasolina. Tal vez se está haciendo tarde y te estás convirtiendo fatigado Tal vez te equivocaste en primer lugar y realmente no puedes deshacerte de todos esos subclases Tal vez no tienes las pruebas para respaldarte. Cualquiera sea la causa, tu confianza se ha ido. No puedes dar el siguiente paso con certeza. No crees que arruinarás nada, Pero no estás seguro.

Ahí es cuando te detienes. Si el código ya es mejor, integre y libere lo que ha hecho. Si se no es mejor, vete. Tirar de la cadena. Me alegro de haber aprendido una lección, lástima que no haya funcionado. Que hay en

¿para mañana?

Mañana o al día siguiente o al mes siguiente o tal vez incluso el año que viene (mi récord personal es nueve años esperando la segunda mitad de una refactorización), llega la idea. O entiendes por qué estabas equivocado o entiendes por qué tenías razón. En cualquier caso, el siguiente paso es claro. Tú da el paso con la confianza que tenías cuando empezaste. Tal vez incluso estás un poco avergonzado por lo estúpido que podrías haber sido de no haberlo visto todo el tiempo. No se Le pasa a todo el mundo.

Es un poco como caminar por un sendero estrecho sobre una caída de mil pies. Mientras la luz espera, puede avanzar con cautela pero con confianza. Sin embargo, tan pronto como se pone el sol, Será mejor que te detengas. Te acuestas por la noche, seguro que el sol volverá a salir por la mañana.

Esto puede sonar místico y vago. En cierto sentido lo es, porque es un nuevo tipo de relación con su programa Cuando realmente entiendes la refactorización, el diseño del sistema es tan fluido y plástico y moldeable para usted como los caracteres individuales en un archivo de código fuente. Puedes sentir el Todo el diseño a la vez. Puedes ver cómo podría flexionarse y cambiar, un poco de esta manera y esto es posible, un poco de esa manera y eso es posible.

Página 334

En otro sentido, sin embargo, no es para nada místico o vago. La refactorización es una habilidad que se puede aprender, el componentes sobre los que has leído en este libro y sobre los que has comenzado a aprender. Obtienes esos Pequeñas habilidades juntas y pulidas. Entonces comienzas a ver el desarrollo bajo una nueva luz.

Dije que esto era una habilidad que se puede aprender. Como lo aprendes

Acostúmbrate a elegir un gol.

En algún lugar tu código huele mal. Resuelva deshacerse del problema. Entonces marche hacia eso Gol. No estás refactorizando para buscar la verdad y la belleza (al menos eso no es todo). Usted está tratando de hacer que su mundo sea más fácil de entender, para recuperar el control de un programa que está aleteando suelto.

Detente cuando no estés seguro.

A medida que avanza hacia su objetivo, puede llegar un momento en que no pueda probarse exactamente a sí mismo y otros que lo que está haciendo preservará la semántica de su programa. Detener. Si el código es ya mejor, adelante y libera tu progreso. Si no es así, deseche sus cambios.

Retractarse.

La disciplina de refactorización es difícil de aprender y fácil de perder de vista, aunque solo sea por un momento. yo Todavía pierdo de vista con más frecuencia de lo que me gustaría admitir. Haré dos o tres o cuatro refactorizaciones seguidas sin volver a ejecutar los casos de prueba. Por supuesto que puedo salirse con la suya. Soy confidente. He practicado ¡Auge! Una prueba falla y no puedo ver cuál de mis cambios causó el problema.

En este momento, estarás muy tentado a depurarte para salir de los problemas. Después de todo, tu consiguió esas pruebas para ejecutar en primer lugar. ¿Qué tan difícil puede ser hacer que vuelvan a funcionar? Detener. Tú están fuera de control y no tienes idea de lo que se necesitará para recuperar el control al avanzar. Regrese a su última buena configuración conocida. Repita sus cambios uno por uno. Ejecuta las pruebas despues de cada uno.

Esto puede sonar obvio aquí en la comodidad de su sillón reclinable. Cuando estás pirateando y puedes huele una gran simplificación a centímetros de distancia, es lo más difícil de hacer para parar y retroceder. Pero piénsalo ahora, mientras tienes la cabeza despejada. Si ha refactorizado durante una hora, solo tomará unos diez minutos para repetir lo que hiciste. Por lo tanto, puede estar seguro de que volverá a la normalidad en diez

minutos. Sin embargo, si intenta avanzar, puede estar depurando durante cinco segundos o durante dos horas

Es fácil para mí decirte qué hacer ahora. Es brutalmente difícil hacerlo realmente. Creo que mi personal El registro por no seguir mi propio consejo es de cuatro horas y tres intentos separados. Salí de control, retrocedió, avanzó lentamente al principio, perdió el control una y otra vez, durante cuatro Horas dolorosas. No es divertido Por eso necesitas ayuda.

Duetos

Por amor de Dios, refactorizar con alguien. Hay muchas ventajas para trabajar en parejas para todos Tipos de desarrollo. Las ventajas funcionan en espadas para la refactorización. En la refactorización hay un prima por trabajar con cuidado y metódicamente. Tu pareja está ahí para mantenerte en movimiento paso a paso paso, y estás ahí para él o ella. En la refactorización hay una ventaja en ver posiblemente lejos consecuencias variadas. Su pareja está allí para ver cosas que no ve y sabe cosas que usted no lo se En la refactorización, es importante saber cuándo dejar de fumar. Cuando tu pareja no entiende lo que está haciendo, es una señal segura de que usted tampoco. Sobre todo, en

refactorizar hay una prima absoluta en la confianza silenciosa. Tu pareja está ahí para gentilmente animarle cuando de lo contrario podría detenerse.

Otro aspecto de trabajar con un compañero es hablar. Quieres hablar de lo que piensas para que suceda, entonces ustedes dos apuntan en la misma dirección. Quieres hablar de lo que pensar está sucediendo, por lo que puede detectar problemas lo antes posible. Quieres hablar de lo que solo sucedió, así sabrás mejor la próxima vez. Todos esos cementos parlantes en tu mente exactamente donde refactorizaciones individuales se ajustan al ritmo de refactorización.

Es probable que vea nuevas posibilidades en su código, incluso si ha trabajado con él durante años, una vez que conozca los olores y las refactorizaciones que pueden esterilizarlos. Puede que incluso quieras para saltar y limpiar cada problema a la vista. No lo hagas Ningún gerente quiere escuchar al equipo decirlo tiene que detenerse durante tres meses para limpiar el desorden que ha creado. Y, bueno, no deberían. Un gran La refactorización es una receta para el desastre.

Tan feo como se ve el desorden ahora, disciplínese para mordisquear el problema. Cuando estás Para agregar alguna funcionalidad nueva a un área, primero tome unos minutos para limpiarla. Si usted tiene para agregar algunas pruebas antes de que pueda limpiar con confianza, agréguelas. Estaras contento de haberlo hecho. Refactorizar primero es menos peligroso que agregar un nuevo código. Tocar el código te recordará cómo funciona. Terminará más rápido y tendrá la satisfacción de saber que la próxima vez que De esta manera, el código se verá mejor que esta vez.

Nunca olvides los dos sombreros. Cuando refactorice, inevitablemente descubrirá casos en los que el código no funciona bien Estarás absolutamente seguro de ello. Resistir la tentación. Cuando refactorizas, su objetivo es dejar que el código computee exactamente las mismas respuestas que cuando lo encontró. Nada más y nada menos. Mantenga una lista (siempre tengo una tarjeta de índice al lado de mi computadora) de cosas para cambiar más tarde: casos de prueba para agregar o cambiar, refactorizaciones no relacionadas, documentos para escribir, diagramas para dibujar. De esa manera no perderás esos pensamientos, pero no dejarás que estropeen lo que que estas haciendo ahora

Referencias

[Auer] Ken. Auer *"Reusabilidad mediante autoencapsulación".* En lenguajes de patrón del programa Diseño 1, Coplien JO Schmidt. DC Reading, Mass .: Addison-Wesley, 1995. Documento de patrones sobre El concepto de autoencapsulación.

[Bäumer y Riehle] Bäumer, Riehle y Riehle. Dirk *"comerciante de productos".* En lenguajes de patrones de Diseño del programa 3, R. Martin F. Buschmann D. Riehle. Reading, Mass .: Addison-Wesley, 1998. A patrón para crear objetos de manera flexible sin saber en qué clase deberían ser.

[Beck] Kent. Beck *Smalltalk Patrones de mejores prácticas.* Upper Saddle River, Nueva Jersey: Prentice Hall, 1997a. Un libro esencial para cualquier Smalltalker, y un libro muy útil para cualquier orientado a objetos. desarrollador. Los rumores de una versión de Java abundan.

[Beck, hanoi] Kent.Beck *"Hazlo funcionar,* hazlo bien: diseña a *través de la refactorización".* The Smalltalk Informe, 6: (1997b): 19-24. La primera escritura publicada que realmente tiene la sensación de cómo funciona el proceso de refactorización de obras. La fuente de muchas ideas para el Capítulo 1.

[Beck, XP] Kent.Beck *eXtreme Programming eXplained: Embrace Change.* Lectura, misa .: Addison-Wesley, 2000.

[Fowler, UML] FowlerM.Scott.K. *UML destilado, segunda edición: una breve guía de la norma Lenguaje de modelado de objetos.* Lectura, Misa: Addison-Wesley, 2000.

Una guía concisa del lenguaje de modelado unificado utilizado para varios diagramas en este libro.

[Fowler, AP] Patrones de *análisis de* M.Fowler *: modelos de objetos reutilizables.* Lectura, Misa .: Addison-Wesley, 1997. Un libro de patrones de modelos de dominio. Incluye una discusión del patrón de rango.

[Banda de cuatro] E. Gamma, R. Helm, R. Johnsonand J. Vlissides. *Patrones de diseño: elementos de Software orientado a objetos reutilizables.* Reading, Mass .: Addison-Wesley, 1995. Probablemente el single. libro más valioso sobre diseño orientado a objetos. Ahora es imposible parecer que sabes algo sobre objetos si no puede hablar inteligentemente sobre estrategia, singleton y cadena de responsabilidad.

[Jackson, 1993] Michael. Jackson *Michael Jackson's Beer Guide,* Mitchell Beazley, 1993. A

guía útil para un tema que recompensa un estudio práctico considerable.

[Java Spec] Gosling, James, Bill Joy y Guy Steele. *La especificación del lenguaje Java,* segundo Edición. Boston, Massachusetts: Addison-Wesley, 2000. La respuesta autorizada a las preguntas de Java.

[JUnit] Beck, Kent y Erich Gamma. *Marco de prueba JUnit de código abierto.* Disponible en el Web (http://www.junit.org). Herramienta esencial para trabajar en Java. Un marco simple que ayuda usted escribe, organiza y ejecuta pruebas unitarias. Marcos similares están disponibles para Smalltalk y C ++.

[Lea] Doug.Lea, *Programación concurrente en Java: principios y patrones de diseño,* lectura, Mass .: Addison-Wesley, 1997. El compilador debería detener a cualquiera que implemente Runnable que No ha leído este libro.

Page 337

[McConnell] Steve. McConnell, *Code Complete: A Practical Handbook of Software Construction.* Redmond, Washington: Microsoft Press, 1993. Una excelente guía para el estilo y el software de programación construcción. Escrito antes de Java, pero se aplican casi todos sus consejos.

[Meyer] Bertrand. Meyer, *construcción de software orientado a objetos.* 2 ed. Upper Saddle River, Nueva Jersey: Prentice Hall, 1997. Un libro muy bueno, aunque muy grande, sobre diseño orientado a objetos. Incluye una discusión exhaustiva del diseño por contrato.

[Opdyke] William F. Opdyke, Ph.D. diss., *"Refactorización de marcos orientados a objetos".* Universidad de Illinois en Urbana-Champaign, 1992. Ver ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z . La primera longitud decente escribiendo sobre refactorización. Lo aborda desde un ángulo algo académico y orientado a las herramientas (después de todo, es una disertación), pero vale la pena leer para aquellos que quieren más sobre la teoría de la refactorización.

[Navegador de refactorización] Brant, John y Don Roberts. *Refactorización de la herramienta del navegador,* http: // st-www.cs.uiuc.edu/~brant/RefactoringBrowser . El futuro de las herramientas de desarrollo de software.

[Woolf] Bobby. Woolf, *"Objeto nulo". En Lenguajes de patrones de diseño de programas 3,* Martin, R. Riehle D. Buschmann F. Reading, Mass .: Addison-Wesley, 1998. Una discusión sobre el objeto nulo modelo.

337