Código limpio

Página 3

Serie Robert C. Martin

La misión de esta serie es mejorar el estado del arte de la artesanía del software. Los libros de esta serie son técnicos, pragmáticos y sustanciales. Los autores son artesanos y profesionales altamente experimentados dedicados a escribir sobre lo que realmente funciona en la práctica, a diferencia de lo que podría funcionar en teoría. Leerás sobre lo que ha hecho el autor, no sobre lo que él cree que debería hacer usted. Si el libro es sobre programación, habrá mucho código. Si el libro trata sobre la gestión, habrá muchos casos de estudio de proyectos reales.

Estos son los libros que todos los practicantes serios tendrán en sus estanterías. Estos son los libros que serán recordados por marcar la diferencia y orientar profesionales para convertirse en verdaderos artesanos.

Gestión de proyectos ágiles
Sanjiv Augustine

Estimación y planificación ágiles
Mike Cohn

Trabajar eficazmente con código heredado
Michael C. Plumas

Agile Java ™: elaboración de código con desarrollo basado en pruebas
Jeff Langr

Principios, patrones y prácticas ágiles en C #
Robert C. Martin y Micah Martin

Desarrollo de software ágil: principios, patrones y prácticas
Robert C. Martin

Código limpio: un manual de artesanía de software ágil
Robert C. Martin

UML para programadores de Java ™
Robert C. Martin

Adecuado para el desarrollo de software: marco para pruebas integradas
Rick Mugridge y Ward Cunningham

Desarrollo de software ágil con SCRUM

Ken Schwaber y Mike Beedle
Ingeniería de software extrema: un enfoque práctico
Daniel H. Steinberg y Daniel W. Palmer

Para obtener más información, visite informit.com/martinseries

Página 4

Código limpio

Un manual de ágil Artesanía del software

Los mentores de objetos:

Robert C. Martin

Michael C. Feathers Timothy R. Ottinger
Jeffrey J. Langr Brett L. Schuchert
James W. Grenning Kevin Dean Wampler
Object Mentor Inc.

Escribir código limpio es lo que debe hacer para poder llamarse un profesional.
No hay excusa razonable para hacer algo menos que lo mejor que puedas.

Upper Saddle River, Nueva Jersey • Boston • Indianápolis • San Francisco
Nueva York • Toronto • Montreal • Londres • Múnich • París • Madrid
Ciudad del Cabo • Sídney • Tokio • Singapur • Ciudad de México

Página 5

Incluye referencias bibliográficas e indice.
ISBN 0-13-235088-2 (pbk.: Papel alcalino)
1. Desarrollo de software ágil. 2. Software informático: fiabilidad. I. Título.
QA76.76.D47M3652 2008
005.1 — dc22 2008024750

ISBN-13: 978-0-13-235088-4
ISBN-10: 0-13-235088-2
Texto impreso en los Estados Unidos en papel reciclado en Courier en Stoughton, Massachusetts.
Primera impresión julio de 2008

Página 6

Para Ann Marie: El amor eterno de mi vida.

Página 7

Esta página se dejó en blanco intencionalmente

Página 8

Contenido

Página 9

Página 10

Página 11

Pagina 12

Página 13

Página 14

Página 15

Página 16

Página 17

Página 18

Página 19

Esta página se dejó en blanco intencionalmente

Prefacio

Uno de nuestros dulces favoritos aquí en Dinamarca es Ga-Jol, cuyos fuertes vapores de regaliz son un complemento perfecto para nuestro clima húmedo y, a menudo, frío. Parte del encanto de Ga-Jol para

nosotros daneses son los dichos sabios o ingeniosos impresos en la solapa de la tapa de cada caja. Compré dos paquete del manjar esta mañana y descubrí que llevaba esta vieja sierra danesa:

Ærlighed i små ting er ikke nogen lille ting .

"La honestidad en las cosas pequeñas no es poca cosa". Fue un buen augurio consistente con lo que yo ya quería decir aquí. Las pequeñas cosas importan. Este es un libro sobre preocupaciones humildes cuyo valor, sin embargo, está lejos de ser pequeño.

Dios está en los detalles , dijo el arquitecto Ludwig mies van der Rohe. Esta cita recuerda argumentos contemporáneos sobre el papel de la arquitectura en el desarrollo de software, y particularmente en el mundo ágil. Bob y yo de vez en cuando nos encontramos apasionadamente comprometidos este diálogo. Y sí, mies van der Rohe estuvo atento a la utilidad y a las formas atemporales de edificios que subyacen a la gran arquitectura. Por otro lado, también seleccionó personalmente cada pomo de cada casa que diseñó. ¿Por qué? Porque las pequeñas cosas importan.

En nuestro "debate" en curso sobre TDD, Bob y yo hemos descubierto que estamos de acuerdo en que La arquitectura de software tiene un lugar importante en el desarrollo, aunque es probable que tengamos diferentes visiones de exactamente lo que eso significa. Sin embargo, tales objeciones son relativamente poco importantes, porque podemos dar por sentado que los profesionales responsables dan *algo de* tiempo para pensar- planificación y planificación desde el inicio de un proyecto. Las nociones de diseño de finales de la década de 1990 impulsadas *únicamente* por las pruebas y el código han desaparecido. Sin embargo, la atención a los detalles es un factor aún más crítico. base de profesionalismo que cualquier gran visión. Primero, es a través de la práctica en el pequeños para que los profesionales adquieran competencia y confianza para la práctica en los grandes. Segundo, el el pedacito más pequeño de construcción descuidada, de la puerta que no cierra herméticamente o el leve baldosas torcidas en el suelo, o incluso el escritorio desordenado, disipa por completo el encanto de la conjunto más grande. De eso se trata el código limpio.

Sin embargo, la arquitectura es solo una metáfora del desarrollo de software y, en particular, de esa parte del software que entrega el *producto* inicial en el mismo sentido que un arquitecto entrega un edificio impecable. En estos días de Scrum y Agile, la atención se centra en la rapidez llevar el *producto* al mercado. Queremos que la fábrica funcione a la máxima velocidad para producir software. Se trata de fábricas humanas: codificadores que piensan y sienten que trabajan a partir de la base de un producto. registro o historia de usuario para crear el *producto* . La metáfora de la fabricación cobra cada vez más fuerza en tales pensando. Los aspectos de producción de la fabricación de automóviles japonesa, de una línea de montaje mundo, inspiran gran parte de Scrum.

Página 21

Sin embargo, incluso en la industria automotriz, la mayor parte del trabajo no reside en la fabricación, sino en mantenimiento — o su evitación. En software, el 80% o más de lo que hacemos se llama curiosamente "Mantenimiento": el acto de reparar. En lugar de abrazar el enfoque occidental tpico en la *pro- duciendo un* buen software, deberíamos pensar más como reparadores de viviendas en el edificio industria, o mecánica automotriz en el campo automotriz. ¿Qué tiene la administración japonesa? que decir sobre *eso* ?

Aproximadamente en 1951, surgió un enfoque de calidad llamado Mantenimiento Productivo Total (TPM). en la escena japonesa. Se centra en el mantenimiento más que en la producción. Uno de los Los pilares principales de TPM son el conjunto de los llamados principios 5S. 5S es un conjunto de disciplinas y aquí utilizo el término "disciplina" de manera instructiva. Estos principios de las 5S están de hecho en la base ciones de Lean, otra palabra de moda en la escena occidental, y un cada vez más prominente palabra de moda en los círculos del software. Estos principios no son una opción. Como relata el tío Bob en En primer plano, la buena práctica del software requiere tal disciplina: concentración, presencia de ánimo, y pensando. No siempre se trata solo de hacer, de impulsar el equipo de la fábrica para producir duce a la velocidad óptima. La filosofía de las 5S comprende estos conceptos:

- *Seiri* u organización (piense en "sort" en inglés). Saber dónde están las cosas: usar enfoques como la denominación adecuada, es crucial. ¿Crees que los identificadores de nombres no ¿importante? Siga leyendo en los siguientes capítulos.
- *Seiton* , o *tidiness* (piense en "sistematizar" en inglés). Hay un viejo dicho americano: *Un lugar para todo y todo en su lugar* . Un fragmento de código debe estar donde espera encontrarlo y, si no es así, debe volver a factorizar para llegar allí.
- *Seiso* , o limpieza (piense en "brillar" en inglés): mantenga el lugar de trabajo libre de colgar alambres, grasa, desperdicios y desperdicios. ¿Qué dicen los autores aquí sobre tirar basura en tu

código con comentarios y líneas de código comentadas que capturan el historial o los deseos de ¿el futuro? Deshazte de ellos.

- *Seiketsu* , o estandarización: el grupo acuerda cómo mantener limpio el lugar de trabajo. ¿Crees que este libro dice algo acerca de tener un estilo de codificación consistente y un conjunto de prácticas dentro del grupo? ¿De dónde provienen esos estándares? Sigue leyendo.
- *Shutsuke* , o disciplina (*auto* -disciplina). Esto significa tener la disciplina para seguir las prácticas y reflexionar con frecuencia sobre el trabajo de uno y estar dispuesto a cambiar.

Si acepta el desafío (sí, el desafío) de leer y aplicar este libro, llegará a comprender y apreciar el último punto. Aquí, finalmente nos dirigimos a la raíces del profesionalismo responsable en una profesión que debe preocuparse por la vida ciclo de un producto. Dado que mantenemos automóviles y otras máquinas bajo TPM, el mantenimiento inactivo, a la espera de que aparezcan errores, es la excepción. En cambio, subimos un nivel: inspeccione las máquinas todos los días y repare las piezas de desgaste antes de que se rompan, o haga lo equivalente al proverbial cambio de aceite de 10,000 millas para prevenir el desgaste. En codigo, refactorizar sin piedad. Puede mejorar un nivel más, ya que el movimiento TPM innova. lanzado hace más de 50 años: construya máquinas que sean más fáciles de mantener en primer lugar. Mak- Hacer que su código sea legible es tan importante como hacerlo ejecutable. La máxima práctica, introducido en los círculos de TPM alrededor de 1960, se centra en la introducción de máquinas completamente nuevas o

Página 22

reemplazando los viejos. Como nos advierte Fred Brooks, probablemente deberíamos volver a hacer trozos de vajilla desde cero cada siete años aproximadamente para barrer los residuos que se arrastran. Quizás deberíamos actualizar la constante de tiempo de Brooks a un orden de semanas, días u horas en lugar de años. Ahí es donde reside el detalle.

Hay un gran poder en los detalles, pero hay algo humilde y profundo en este enfoque de la vida, como podríamos esperar estereotipadamente de cualquier enfoque que afirme nese raíces. Pero esta no es solo una visión oriental de la vida; Sabiduría popular inglesa y americana dom están llenos de tales amonestaciones. La cita de Seiton de arriba fluyó de la pluma de un ministro de Ohio que literalmente veía la pulcritud como "un remedio para todos los grados de maldad". ¿Qué hay de Seiso? *La limpieza está al lado de la piedad* . Tan hermosa como es una casa, un desordenado el escritorio le roba su esplendor. ¿Qué hay de Shutsuke en estos pequeños asuntos? *El que es fiel en lo poco es fiel en lo mucho* . ¿Qué tal estar ansioso por volver a factorizar en el momento responsable, fortalecer la posición de uno para las decisiones "grandes" posteriores, en lugar de posponerlo? *A puntada a tiempo ahorra nueve* . *El pájaro temprano atrapa al gusano. No dejes para mañana lo que puedes hacer hoy.* (Tal era el sentido original de la frase "el último responsable momento ”en Lean hasta que cayó en manos de consultores de software .) ¿Qué tal la calibración? ¿ocupar el lugar de los pequeños esfuerzos individuales en un gran conjunto? *Poderosos robles de pequeñas bellotas crecer.* ¿O qué tal integrar el trabajo preventivo simple en la vida cotidiana? *Una onza de la prevención vale una libra de curación. Una manzana al día mantiene alejado al médico.* Código limpio Honra las raíces profundas de la sabiduría debajo de nuestra cultura más amplia, o nuestra cultura como lo fue antes, o debería ser y *puede* ser con atención a los detalles.

Incluso en la gran literatura arquitectónica encontramos sierras que se remontan a estos supuestos detalles planteados. Piense en los pomos de las puertas de mies van der Rohe. Eso es *seiri* . Eso es estar atento a cada nombre de variable. Debe nombrar una variable con el mismo cuidado con el que nombrar un hijo primogénito.

Como todo propietario sabe, tal cuidado y refinamiento continuo nunca llega a su fin. El arquitecto Christopher Alexander, padre de los patrones y los lenguajes de patrones, considera cada acto de diseño en sí mismo como un pequeño acto local de reparación. Y ve la artesanía de fina estructura que será competencia exclusiva del arquitecto; las formas más grandes se pueden dejar a los patrones y su aplicación por parte de los habitantes. El diseño siempre está en curso, no solo cuando agregamos un nuevo habitación a una casa, pero como estamos atentos a repintar, reemplazar alfombras gastadas o actualizar- ing el fregadero de la cocina. La mayoría de las artes se hacen eco de sentimientos análogos. En nuestra búsqueda de otros que atribuir el hogar de Dios a los detalles, nos encontramos en la buena compañía de los El autor francés del siglo XIX Gustav Flaubert. El poeta francés Paul Valéry nos advierte que un El poema nunca se termina y soporta continuas repeticiones, y dejar de trabajar en él es abandono.

Esta preocupación por los detalles es común a todos los esfuerzos de excelencia. Entonces tal vez allí es poco nuevo aquí, pero al leer este libro, se le desafiará a tomar una buena disciplina.
plina que hace mucho tiempo que se rindió a la apatía o al deseo de espontaneidad y
"Responder al cambio".

Desafortunadamente, no solemos ver estas preocupaciones como piedras angulares del arte de programación. Abandonamos nuestro código temprano, no porque esté hecho, sino porque nuestro valor El sistema se centra más en la apariencia exterior que en la sustancia de lo que ofrecemos.

Página 23

Esta falta de atención nos cuesta al final: *siempre aparece un centavo malo* . Investigación, ni en ni en la industria ni en la academia, se humilla a la humilde posición de mantener limpio el código. atrás en mis días trabajando en la organización Bell Labs Software Production Research (*Produc-de* hecho!), tuvimos algunos hallazgos en el reverso del sobre que sugirieron que El estilo de sangría fue uno de los indicadores estadísticamente más significativos de baja densidad de errores. Queremos que sea esa arquitectura o lenguaje de programación o alguna otra noción alta debe ser la causa de la calidad; como personas cuyo supuesto profesionalismo se debe a la dominio de las herramientas y de los elevados métodos de diseño, nos sentimos insultados por el valor que Las máquinas de piso, los codificadores, agregan mediante la aplicación simple y consistente de una muesca. estilo. Para citar mi propio libro de hace 17 años, tal estilo distingue la excelencia de mera competencia. La cosmovisión japonesa comprende el valor crucial de lo cotidiano. trabajador y, más aún, de los sistemas de desarrollo que deben a la simple, cotidiana acciones de esos trabajadores. La calidad es el resultado de un millón de actos desinteresados de cuidado, no solo de cualquier gran método que descienda de los cielos. Que estos actos sean simples no significa que son simplistas, y eso no significa que sean fáciles. Sin embargo, son los tejido de grandeza y, más aún, de belleza, en cualquier empresa humana. Ignorarlos no es aún por ser completamente humano.

Por supuesto, todavía soy partidario de pensar en un alcance más amplio, y en particular de la valor de los enfoques arquitectónicos arraigados en un conocimiento profundo del dominio y la usabilidad del software. El libro no se trata de eso, o, al menos, obviamente no se trata de eso. Este libro tiene un mensaje cuya profundidad no debe ser subestimada. Encaja con la sierra actual de las personas realmente basadas en códigos como Peter Sommerlad, Kevlin Henney y Giovanni Asproni. “El código es el diseño” y “Código simple” son sus mantras. Mientras debemos tenga cuidado de recordar que la interfaz es el programa, y que sus estructuras tienen mucho decir sobre la estructura de nuestro programa, es crucial adoptar continuamente la postura humilde que el diseño vive en el código. Y mientras que la reelaboración en la metáfora de la fabricación conduce a costo, la reelaboración en el diseño genera valor. Deberíamos ver nuestro código como la hermosa articulación de nobles esfuerzos de diseño: el diseño como un proceso, no como un punto final estático. Está en el código que se desarrollan las métricas arquitectónicas del acoplamiento y la cohesión. Si escuchas a Larry Constan- Cuando describe el acoplamiento y la cohesin, habla en trminos de cdigo, no en elevadas condiciones abstractas. conceptos que se pueden encontrar en UML. Richard Gabriel nos aconseja en su ensayo, “Abstracción Descanse ”que la abstracción es mala. El código es anti-malvado y el código limpio es quizás divino.

Volviendo a mi cajita de Ga-Jol, creo que es importante señalar que el danés La sabiduría nos aconseja no solo que prestemos atención a las cosas pequeñas, sino también que seamos *honestos* en las pequeñas. cosas. Esto significa ser honesto con el código, honesto con nuestros colegas sobre el estado de nuestra código y, sobre todo, ser honestos con nosotros mismos sobre nuestro código. Hicimos nuestro mejor esfuerzo para “Dejar el campamento más limpio de lo que lo encontramos”? ¿Refactorizamos nuestro código antes de verificar? entrando? Estas no son preocupaciones periféricas, sino preocupaciones que se encuentran directamente en el centro de Valores ágiles. Es una práctica recomendada en Scrum que la re-factorización sea parte de la excepto de "Hecho". Ni la arquitectura ni el código limpio insisten en la perfección, solo en la honestidad y haciendo lo mejor que podemos. *Errar es humano; perdonar, divino.* En Scrum, hacemos todo cosa visible. Aireamos nuestra ropa sucia. Somos honestos sobre el estado de nuestro código porque

Página 24

el código nunca es perfecto. Nos volvemos más plenamente humanos, más dignos de lo divino y más cercanos a esa grandeza en los detalles.

En nuestra profesión, necesitamos desesperadamente toda la ayuda que podamos obtener. Si un taller limpio reduce los accidentes y las herramientas de taller bien organizadas aumentan la productividad, entonces estoy a favor ellos. En cuanto a este libro, es la mejor aplicación pragmática de los principios Lean al software I he visto alguna vez en forma impresa. No esperaba menos de este pequeño grupo práctico de personas pensantes personas que han estado luchando juntos durante años no solo para mejorar, sino también para regalar su conocimiento a la industria en trabajos como el que ahora encuentra en sus manos. Deja el mundo un poco mejor de lo que lo encontré antes de que el tío Bob me enviara el manuscrito.

Habiendo completado este ejercicio con nobles ideas, me voy a limpiar mi escritorio.

James O. Coplien
Mørdrup, Dinamarca

Página 25

Esta página se dejó en blanco intencionalmente

Introducción

ocus Shift

(c) 2008 F

Reproducido con el amable permiso de Thom Holwerda.
http://www.osnews.com/story/19266/WTFs_m

¿Qué puerta representa tu código? ¿Qué puerta representa a su equipo o su empresa?
¿Por qué estamos en esa habitación? ¿Es esto solo una revisión de código normal o hemos encontrado un flujo de problemas horribles poco después de salir en vivo? ¿Estamos depurando en pánico, estudiando detenidamente el código? que pensamos que funcionó? ¿Los clientes se van en masa y los gerentes están respirando

Página 27

nuestros cuellos? ¿Cómo podemos asegurarnos de que terminamos detrás de la puerta *correcta* cuando las cosas se ponen? ¿difícil? La respuesta es: *artesanía* .

El aprendizaje de la artesanía consta de dos partes: el conocimiento y el trabajo. Debes ganar el conocimiento de los principios, patrones, prácticas y heurísticas que un artesano conoce, y también debe moler ese conocimiento en sus dedos, ojos e intestinos trabajando duro y practicando.

Puedo enseñarte la física de andar en bicicleta. De hecho, la matemática clásica es relativamente sencillo. Gravedad, fricción, momento angular, centro de masa, etc. adelante, se puede demostrar con menos de una página llena de ecuaciones. Dadas esas fórmulas yo podría demostrarle que andar en bicicleta es práctico y brindarle todo el conocimiento que necesita necesario para que funcione. Y todavía te caerías la primera vez que te subiste a esa bicicleta.

La codificación no es diferente. Podríamos escribir todos los principios de "sentirse bien" de la limpieza código y luego confiar en ti para hacer el trabajo (en otras palabras, dejarte caer cuando te subas la bicicleta), pero entonces, ¿qué tipo de profesores nos convertirían y qué tipo de estudiante eso te haría?

No. Esa no es la forma en que este libro va a funcionar.

Aprender a escribir código limpio es *un trabajo duro* . Requiere algo más que el conocimiento de

principios y patrones. Debes *sudar* por eso. Debes practicarlo tú mismo y observar tú mismo fallas. Debes ver a otros practicarlo y fallar. Debes verlos tropezar y vuelve sobre sus pasos. Debe verlos agonizar por las decisiones y ver el precio que pagan tomar esas decisiones de manera incorrecta.

Esté preparado para trabajar duro mientras lee este libro. Este no es un libro para "sentirse bien" que puedes leer en un avión y terminar antes de aterrizar. Este libro te hará trabajar *y trabajar duro* . ¿Qué tipo de trabajo harás? Leerá código, mucho código. Y se le desafiará a pensar en lo que está bien en ese código y lo que está mal con eso. Se le pedirá que lo siga mientras desarmamos los módulos y los volvemos a colocar juntos de nuevo. Esto requerirá tiempo y esfuerzo; pero creemos que valdrá la pena.

Hemos dividido este libro en tres partes. Los primeros capítulos describen los principios Principios, patrones y prácticas de escritura de código limpio. Hay bastante código en estos capítulos, y serán difíciles de leer. Te prepararán para la segunda sección venir. Si dejas el libro después de leer la primera sección, ¡buena suerte!

La segunda parte del libro es el trabajo más duro. Consta de varios estudios de caso de complejidad cada vez mayor. Cada estudio de caso es un ejercicio de limpieza de algún código, de transformar código que tiene algunos problemas en código que tiene menos problemas. El detalle en esta sección es *intensa* . Tendrás que ir y venir entre la narrativa y el listados de código. Deberá analizar y comprender el código con el que estamos trabajando y Repase nuestro razonamiento para hacer cada cambio que hacemos. Reserva algo de tiempo porque *esto debería llevarte días* .

La tercera parte de este libro es la recompensa. Es un solo capítulo que contiene una lista de heurísticas y olores recopilados durante la creación de los estudios de caso. Mientras caminábamos y limpiamos el código en los estudios de caso, documentamos todas las razones de nuestras acciones como

Página 28

heurístico o olfativo. Intentamos entender nuestras propias reacciones al código que estábamos leyendo. y cambiando, y trabajamos duro para capturar por qué sentimos lo que sentimos e hicimos lo que hicimos. El resultado es una base de conocimientos que describe la forma en que pensamos cuando escribimos, leemos y código limpio.

Esta base de conocimientos tiene un valor limitado si no hace el trabajo de leer detenidamente a través de los estudios de caso de la segunda parte de este libro. En esos estudios de caso tenemos cuidadoanotó completamente cada cambio que hicimos con referencias futuras a la heurística. Estos paralas referencias de barrio aparecen entre corchetes como este: [H22]. Esto le permite ver el *contexto* en que esas heurísticas fueron aplicadas y escritas! No son las heurísticas en sí mismas las que tan valiosa, es la *relación entre esas heurísticas y las decisiones discretas que realizado mientras se limpiaba el código en los estudios de caso* .

Para ayudarlo aún más con esas relaciones, hemos colocado una referencia cruzada al final. del libro que muestra el número de página de cada referencia futura. Puedes usarlo para mirar en cada lugar donde se aplicó una determinada heurística.

Si lee la primera y la tercera sección y omite los estudios de caso, entonces He leído otro libro sobre "sentirse bien" sobre cómo escribir un buen software. Pero si tomas el tiempo para trabajar en los estudios de caso, siguiendo cada pequeño paso, cada minuto de decisión, si te pones en nuestro lugar y te obligas a pensar en los mismos caminos que nosotros pensamiento, entonces obtendrá una comprensión mucho más rica de esos principios, patrones, prácticas tices y heurísticas. Ya no serán conocimientos de "sentirse bien". Ellos habrán sido molido en su intestino, dedos y corazón. Se habrán convertido en parte de ti de la misma manera. que una bicicleta se convierta en una extensión de tu voluntad cuando hayas dominado cómo montarla.

Expresiones de gratitud

Obra de arte

Gracias a mis dos artistas, Jeniffer Kohnke y Angela Brooks. Jennifer es responsable por las impresionantes y creativas imágenes al comienzo de cada capítulo y también por los retratos

de Kent Beck, Ward Cunningham, Bjarne Stroustrup, Ron Jeffries, Grady Booch, Dave Thomas, Michael Feathers y yo.

Ángela es la responsable de las ingeniosas imágenes que adornan las entrañas de cada capítulo. Ha hecho bastantes fotografías para mí a lo largo de los años, incluidas muchas de las fotografías interiores. turas en el *desarrollo ágil de software: principios, patrones y prácticas* . Ella también es mi primogénito en quien tengo complacencia.

Página 29

Esta página se dejó en blanco intencionalmente

Página 30

En la portada

La imagen de la portada es M104: The Sombrero Galaxy. M104 se encuentra en Virgo y es
a poco menos de 30 millones de años luz de nosotros. En su núcleo hay un agujero negro supermasivo que pesa
ing en alrededor de mil millones de masas solares.

¿Te recuerda la imagen a la explosión de *Praxis, la* luna de poder klingon ? I
Recuerdo vívidamente la escena en *Star Trek VI* que mostraba un anillo ecuatorial de escombros volando
lejos de esa explosión. Desde esa escena, el anillo ecuatorial ha sido un artefacto común.
en explosiones de películas de ciencia ficción. Incluso se agregó a la explosión de Alderaan en ediciones posteriores.
de la primera película de *Star Wars* .

¿Qué causó que este anillo se formara alrededor de M104? ¿Por qué tiene una central tan enorme?
bulto y un núcleo tan brillante y diminuto? Me parece que el agujero negro central
perdió la calma y abrió un agujero de 30.000 años luz en el medio de la galaxia. Ay de cualquiera
civilizaciones que podrían haber estado en el camino de esa disrupción cósmica.

Los agujeros negros supermasivos se tragan estrellas enteras para el almuerzo, convirtiendo una fracción considerable
conversión de su masa en energía. $E = MC^2$ es suficiente apalancamiento, pero cuando M es una masa estelar:
¡Estar atento! ¿Cuántas estrellas cayeron de cabeza en esas fauces antes de que el monstruo se saciara?
¿Podría ser una pista el tamaño del vacío central?

La imagen de M104 en la portada es una
combinación de la famosa fotografía de luz visible
tograph de Hubble (derecha), y el reciente
imagen infrarroja del Spitzer orbitando
observatorio (abajo, derecha). Es el infrarrojo
imagen que nos muestra claramente la naturaleza del anillo
de la galaxia. En luz visible solo vemos el
borde frontal del anillo en silueta. El cen-
El bulto tral oscurece el resto del anillo.

Pero en el infrarrojo, las partículas calientes en
el anillo brilla a través del abultamiento central. La
dos imágenes combinadas nos dan una vista que hemos
no visto antes e implica que hace mucho tiempo
era un infierno furioso de actividad.

Imagen de portada: © Telescopio espacial Spitzer

Página 31

Esta página se dejó en blanco intencionalmente

Página 32

1

Código limpio

Estás leyendo este libro por dos razones. Primero, eres un programador. Segundo, quieres para ser un mejor programador. Bien. Necesitamos mejores programadores.

Página 33

Este es un libro sobre buena programación. Está lleno de código. Vamos a mirar código de todas las direcciones diferentes. Lo miraremos desde arriba, lo miraremos desde el abajo, y a través de él desde adentro hacia afuera. Cuando terminemos, sabremos un mucho sobre el código. Además, podremos diferenciar entre código bueno y código malo. código. Sabremos cómo escribir un buen código. Y sabremos cómo transformar el código incorrecto en buen código.

Habrá código

Se podría argumentar que un libro sobre código está de alguna manera atrasado, que el código no es más largo el problema; que deberíamos preocuparnos por los modelos y los requisitos. De hecho, algunos han sugerido que estamos cerca del final del código. Que pronto todo el código ser generado en lugar de escrito. Que los programadores simplemente no serán necesarios porque los negocios

la gente generará programas a partir de especificaciones.

¡Disparates! Nunca nos desharemos del código, porque el código representa los detalles del requisitos. En algún nivel, esos detalles no se pueden ignorar ni abstraer; tienen que ser especificado. Y especificar los requisitos con tal detalle que una máquina pueda ejecutarlos *es programación* . Tal especificación *es código* .

Espero que el nivel de abstracción de nuestros idiomas continúe aumentando. I También se espera que la cantidad de idiomas específicos de dominio continúe creciendo. Esto será algo bueno. Pero no eliminará el código. De hecho, todas las especificaciones escritas en estos niveles y de dominio específico de lenguaje de alto va a *ser* de código! Todavía necesitará ser rigurosos, precisos y tan formales y detallados que una máquina pueda comprender y ejecutalo.

Las personas que piensan que el código desaparecerá algún día son como matemáticos que Espero algún día descubrir una matemática que no tiene por qué ser formal. Ellos están esperando que un día descubriremos una forma de crear máquinas que puedan hacer lo que queramos en lugar de de lo que decimos. Estas máquinas tendrán que ser capaces de comprendernos tan bien que puede traducir necesidades vagamente especificadas en programas que se ejecutan perfectamente y que cumplen con precisión esas necesidades.

Esto nunca va a pasar. Ni siquiera los humanos, con toda su intuición y creatividad, han podido crear sistemas exitosos a partir de los vagos sentimientos de sus clientes. De hecho, si algo nos ha enseñado la disciplina de la especificación de requisitos es que Los requisitos bien especificados son tan formales como el código y pueden actuar como pruebas ejecutables de ese ¡código!

Recuerde que el código es realmente el lenguaje en el que finalmente expresamos los requisitos. mentos. Podemos crear lenguajes que se acerquen más a los requisitos. Podemos crear herramientas que nos ayudan a analizar y ensamblar esos requisitos en estructuras formales. Pero lo haremos nunca elimine la precisión necesaria, por lo que siempre habrá código.

Página 34

Código incorrecto

Recientemente leí el prefacio de Kent Beck's *Patrones de implementación del* libro . [1] Él dice: “. . . esto libro se basa en una premisa bastante frágil: que un buen código importa. . . . " ¿Una premisa *frágil* ? Yo dis- ¡estar de acuerdo! Creo que esa premisa es una de las más robusto, compatible y sobrecargado de todos los mises en nuestro oficio (y creo que Kent lo sabe). Nosotros saber que un buen código es importante porque hemos tenido que lidiar durante tanto tiempo con su falta.

Conozco una empresa que, a finales de los 80, escribió una aplicación *asesina* . Fue muy popular y muchos los profesionales lo compraron y usaron. Pero entonces el Los ciclos de liberación comenzaron a estirarse. Los errores no eran reparado de un lanzamiento al siguiente. Tiempos de carga creció y los choques aumentaron. Recuerdo el dia en que apague el producto con frustración y nunca lo usé de nuevo. La empresa quebró poco tiempo después de eso.

Dos décadas después conocí a uno de los primeros empleados de esa empresa y le pregunté Qué ha pasado. La respuesta confirmó mis temores. Habían apresurado el producto a market y había hecho un gran lío en el código. A medida que agregaron más y más funciones, la el código fue de mal en peor hasta que simplemente ya no pudieron manejarlo. *Fue lo malo código que derribó a la empresa.*

¿ *Alguna* vez *se* ha visto obstaculizado significativamente por un código incorrecto? Si eres programador de cualquier experiencia entonces has sentido este impedimento muchas veces. De hecho, tenemos un nombre para eso. Lo llamamos *vadear* . Atravesamos el código incorrecto. Nos esforzamos por atravesar un pantano de enredados zarzas y trampas escondidas. Luchamos por encontrar nuestro camino, esperando alguna pista, alguna pista de lo que está pasando; pero todo lo que vemos es un código cada vez más insensato.

Por supuesto que se ha visto obstaculizado por un código incorrecto. Entonces, ¿por qué lo escribiste?

¿Estabas tratando de ir rápido? ¿Tenías prisa? Probablemente. Quizás sentiste que tu no tuve tiempo para hacer un buen trabajo; que tu jefe se enfadaría contigo si tomabas el es hora de limpiar su código. Quizás estaba cansado de trabajar en este programa y quería que terminara. O tal vez miró la acumulación de otras cosas que había prometido. estaba listo para terminar y se dio cuenta de que necesitaba unir este módulo para poder pasar al siguiente. Todos lo hemos hecho.

Todos hemos mirado el lío que acabamos de hacer y luego hemos decidido dejarlo para otro día. Todos hemos sentido el alivio de ver que nuestro desordenado programa funciona y de decidir que un

1. [Beck07].

Página 35

el desorden de trabajo es mejor que nada. Todos hemos dicho que volveríamos y lo limpiaríamos más tarde. De Por supuesto, en aquellos días no conocíamos la ley de LeBlanc: más *tarde es igual a nunca* .

El costo total de ser dueño de un desastre

Si ha sido programador durante más de dos o tres años, probablemente haya sido significativamente ralentizado por el código desordenado de otra persona. Si has sido programador durante más de dos o tres años, probablemente se haya visto ralentizado por un código desordenado. El grado de desaceleración puede ser significativo. En el lapso de un año o dos, los equipos que se estaban moviendo muy rápido al comienzo de un proyecto pueden encontrarse moviéndose en el camino de un caracol ritmo. Cada cambio que hacen al código rompe otras dos o tres partes del código. No el cambio es trivial. Cada adición o modificación al sistema requiere que los enredos, los giros y los nudos deben “entenderse” para que se puedan agregar más enredos, giros y nudos. Con el tiempo, el desorden se vuelve tan grande, tan profundo y tan alto que no pueden limpiarlo. Allí no hay manera en absoluto.

A medida que aumenta el desorden, la productividad del equipo continúa disminuyendo, asintóticamente acercándose a cero. A medida que disminuye la productividad, la dirección hace lo único que puede; agregan más personal al proyecto con la esperanza de aumentar la productividad. Pero ese nuevo personal es no versado en el diseño del sistema. No saben la diferencia entre un cambio que coincide con la intención del diseño y un cambio que frustra la intención del diseño. Además, ellos, y todos los demás miembros del equipo, están bajo una presión terrible para aumentar la productividad. Entonces todos hacen más y más líos, llevando la productividad cada vez más hacia cero. (Vea la Figura 1-1.)

Figura 1-1

Página 36

El gran rediseño en el cielo

Finalmente, el equipo se rebela. Informan a la gerencia que no pueden continuar desarrollándose en esta odiosa base de código. Exigen un rediseño. La gerencia no quiere gastar los recursos en un rediseño completamente nuevo del proyecto, pero no pueden negar que la productividad es terrible. Finalmente, se someten a las demandas de los desarrolladores y autorizan la gran rediseño en el cielo.

Se selecciona un nuevo equipo de tigres. Todo el mundo quiere estar en este equipo porque es un verde. proyecto de campo. Pueden empezar de nuevo y crear algo realmente hermoso. Pero solo lo mejor y los más brillantes son elegidos para el equipo tigre. Todos los demás deben continuar manteniendo el sistema actual.

Ahora los dos equipos están en carrera. El equipo tigre debe construir un nuevo sistema que todo lo que hace el sistema antiguo. No solo eso, tienen que mantenerse al día con los cambios. que se están haciendo continuamente al sistema antiguo. La administración no reemplazará a la vieja sistema hasta que el nuevo sistema pueda hacer todo lo que hace el sistema anterior.

Esta carrera puede durar mucho tiempo. Lo he visto tardar 10 años. Y para cuando sea hecho, los miembros originales del equipo tigre se han ido hace mucho tiempo, y los miembros actuales son exigiendo que el nuevo sistema sea rediseñado porque es un desastre.

Si ha experimentado incluso una pequeña parte de la historia que acabo de contar, entonces ya sepa que dedicar tiempo a mantener limpio su código no solo es rentable; es un asunto de supervivencia profesional.

Actitud

¿Alguna vez ha atravesado un lío tan grave que le tomó semanas hacer lo que debería haber hecho? tomado horas? ¿Ha visto lo que debería haber sido un cambio de una línea, hecho en su lugar en cientos de módulos diferentes? Estos síntomas son demasiado comunes.

¿Por qué le sucede esto al código? ¿Por qué el buen código se descompone tan rápidamente en código incorrecto? Nosotros tengo muchas explicaciones para ello. Nos quejamos de que los requisitos cambiaron de manera que frustrar el diseño original. Lamentamos los horarios demasiado ajustados para hacer las cosas bien. Hablamos de gerentes estúpidos, clientes intolerantes y tipos de marketing inútiles. y desinfectantes telefónicos. Pero la culpa, querido Dilbert, no está en nuestras estrellas, sino en nosotros mismos. No somos profesionales.

Esta puede ser una píldora amarga de tragar. ¿Cómo pudo ser culpa *nuestra* este lío ? Qué pasa con la requisitos? ¿Y el horario? ¿Qué pasa con los administradores estúpidos y los inútiles? tipos de marketing? ¿No cargan con parte de la culpa?

No. Los gerentes y especialistas en marketing *nos* buscan la información que necesitan para promesas y compromisos; e incluso cuando no nos miran, no debemos ser tímidos sobre decirles lo que pensamos. Los usuarios nos buscan para validar la forma en que los requisitos encajará en el sistema. Los jefes de proyecto nos buscan para que les ayudemos a elaborar el calendario. Nosotros

Página 37

son profundamente cómplices en la planificación del proyecto y comparten gran parte de las responsabilidades
bilidad por cualquier falla; ¡especialmente si esos fallos tienen que ver con un código incorrecto!

"¡Pero espera!" tu dices. "Si no hago lo que dice mi gerente, me despedirán". Probablemente no.
La mayoría de los gerentes quieren la verdad, incluso cuando no actúan como tal. La mayoría de los gerentes quieren el bien
código, incluso cuando están obsesionados con el horario. Pueden defender el horario y
requisitos con pasión; pero ese es su trabajo. Es *tu* trabajo defender el código con igual
pasión.

Para llevar este punto a casa, ¿qué pasaría si fuera médico y tuviera un paciente que exigiera
que deje de lavarse las manos tontamente en preparación para la cirugía porque estaba tomando
¿demasiado tiempo? [2] Claramente, el paciente es el jefe; y, sin embargo, el médico debería negarse absolutamente
para cumplir. ¿Por qué? Debido a que el médico sabe más que el paciente sobre los riesgos de enfermedad
facilidad e infección. Sería poco profesional (mucho menos criminal) que el médico
Cumplir con el paciente.

También es poco profesional que los programadores se dobleguen a la voluntad de los gerentes que no
comprender los riesgos de hacer líos.

El enigma primordial

Los programadores se enfrentan a un enigma de valores básicos. Todos los desarrolladores con más de unos pocos años
La experiencia sabe que los líos anteriores los ralentizan. Y, sin embargo, todos los desarrolladores sienten
la presión de hacer líos para cumplir con los plazos. En resumen, no se toman el tiempo
ir rápido!

Los verdaderos profesionales saben que la segunda parte del enigma es incorrecta. Se quiere *no*
Cumplir con la fecha límite haciendo el lío. De hecho, el desorden lo ralentizará instantáneamente y
le obligará a no cumplir con la fecha límite. La *única* forma de cumplir la fecha límite, la única forma de
ir rápido: es mantener el código lo más limpio posible en todo momento.

¿El arte del código limpio?

Digamos que cree que el código desordenado es un impedimento importante. Digamos que aceptas
que la única forma de hacerlo rápido es mantener limpio el código. Entonces debes preguntarte: "¿Cómo
¿Escribo código limpio? " No sirve de nada intentar escribir un código limpio si no sabes lo que
significa que el código esté limpio!

La mala noticia es que escribir código limpio es muy parecido a pintar una imagen. La mayor parte de nosotros
saber cuándo un cuadro está bien o mal pintado. Pero poder reconocer el buen arte de
malo no significa que sepamos pintar. También ser capaz de reconocer código limpio
de código sucio no significa que sepamos cómo escribir código limpio.

2. Cuando Ignaz Semmelweis recomendó por primera vez el lavado de manos a los médicos en 1847, se rechazó sobre la base de que los médicos estaban demasiado ocupados y no tenían tiempo para lavarse las manos entre las visitas de los pacientes.

Página 38

Escribir código limpio requiere el uso disciplinado de una miríada de pequeñas técnicas aplicadas a través de un sentido de "limpieza" cuidadosamente adquirido. Este "sentido del código" es la clave. Algunos de nosotros nacemos con eso. Algunos tenemos que luchar para adquirirlo. No solo nos deja ver si el código es bueno o malo, pero también nos muestra la estrategia para aplicar nuestra disciplina pline para transformar el código incorrecto en código limpio.

Un programador sin "sentido de código" puede mirar un módulo desordenado y reconocer el lío, pero no tengo idea de qué hacer al respecto. Un programador *con* "sentido de código" se verá en un módulo desordenado y ver opciones y variaciones. El "sentido del código" ayudará a que El gramo elige la mejor variación y lo guía para trazar una secuencia de comportamiento. preservando las transformaciones para llegar de aquí para allá.

En resumen, un programador que escribe código limpio es un artista que puede tomar una pantalla en blanco a través de una serie de transformaciones hasta convertirse en un sistema elegantemente codificado.

¿Qué es el código limpio?

Probablemente haya tantas definiciones como programadores. Así que le pregunté a algunos programadores bien conocidos y profundamente experimentados lo que pensaban.

Bjarne Stroustrup, inventor de C ++
y autor de *The C ++ Programming*
Idioma

> *Me gusta que mi código sea elegante y eficiente. La*
> *La lógica debe ser sencilla para hacerlo difícil.*
> *para que los errores se oculten, las dependencias mínimas para*
> *mantenimiento sencillo, manejo de errores completo*
> *de acuerdo con una estrategia articulada, y*
> *formación cercana a la óptima para no tentar*
> *que la gente ensucie el código con falta de principios*
> *optimizaciones solicitadas. El código limpio hace una cosa*
> *bien.*

Bjarne usa la palabra "elegante". Esa es ¡Qué palabra! El diccionario en mi MacBook ® proporciona las siguientes definiciones: *agradablemente elegante y elegante en apariencia o manera; agradablemente ingenioso y sencillo.* Observe la énfasis en la palabra "agradable". Aparentemente, Bjarne piensa que el código limpio es *agradable* para leer. Leerlo debería hacerte sonreír como una caja de música bien elaborada o bien diseñada coche lo haría.

Bjarne también menciona la eficiencia, *dos veces* . Quizás esto no debería sorprendernos al venir del inventor de C ++; pero creo que hay más que el puro deseo de velocidad. Los ciclos desperdiciados no son elegantes, no agradan. Y ahora fíjate en la palabra que usa Bjarne

para describir la consecuencia de esa falta de elegancia. Utiliza la palabra "tentar". Hay un profundo verdad aquí. ¡El código incorrecto *tienta* al desorden a crecer! Cuando otros cambian un código incorrecto, tienden a empeorarlo.

Los pragmáticos Dave Thomas y Andy Hunt dijeron esto de una manera diferente. Usaron el meta- por ventanas rotas. [3] Un edificio con ventanas rotas parece que a nadie le importa eso. Entonces otras personas dejan de preocuparse. Permiten que se rompan más ventanas. Finalmente los rompen activamente. Despojan la fachada con grafitis y dejan que la basura se colme. lect. Una ventana rota inicia el proceso hacia la descomposición.

Bjarne también menciona que la gestión de errores debe estar completa. Esto va a la disciplina pline de prestar atención a los detalles. El manejo abreviado de errores es solo una forma las gramáticas pasan por alto los detalles. Las pérdidas de memoria son otra, las condiciones de carrera son otra más. Inconsistente nombrar a otro. El resultado es que el código limpio muestra mucha atención a detalle.

Bjarne cierra con la afirmación de que el código limpio hace bien una cosa. No es casualidad que hay tantos principios de diseño de software que se pueden reducir a este simple amonestación. Escritor tras escritor ha intentado comunicar este pensamiento. El código incorrecto intenta hacer demasiado, ha confundido la intención y la ambigüedad de propósito. El código limpio está *enfocado* . Cada función, cada clase, cada módulo expone una actitud resuelta que permanece enteramente sin distracciones y sin contaminación por los detalles circundantes.

Grady Booch, autor de *Object Análisis y Diseño Orientado con Aplicaciones*

> *El código limpio es simple y directo. Código limpio se lee como una prosa bien escrita. Código limpio nunca oscurece la intención del diseñador, sino que está lleno de abstracciones nítidas y líneas sencillas de control.*

Grady hace algunos de los mismos puntos que Bjarne, pero adopta una perspectiva de *legibilidad* . I especialmente como su opinión de que el código limpio debería leer como prosa bien escrita. Piense en un muy buen libro que has leído. Recuerda como las palabras desaparecieron para ser reemplazadas por imágenes! Fue como ver una película, ¿no? ¡Mejor! Viste a los personajes, tu escuchaste los sonidos, experimentaste el patetismo y el humor.

Leer código limpio nunca será como leer El *señor de los anillos* . Aún así, el litigio La metáfora aria no es mala. Como una buena novela, el código limpio debe exponer claramente los diez siones en el problema a resolver. Debería llevar esas tensiones a un clímax y luego dar

3. http://www.pragmaticprogrammer.com/booksellers/2004-12.html

Página 40

el lector que “¡Ajá! ¡Por supuesto!" a medida que los problemas y tensiones se resuelven en la revelación de una solución obvia.

¡Encuentro que el uso de Grady de la frase "abstracción nítida" es un oxímoron fascinante! Después de todo, la palabra "crujiente" es casi un sinónimo de "hormigón". El diccionario de mi MacBook tiene la siguiente definición de "crujiente": *enérgicamente decisivo y práctico, sin dudarlo tación o detalles innecesarios.* A pesar de esta aparente yuxtaposición de significado, las palabras llevar un mensaje poderoso. Nuestro código debe ser práctico en lugar de especulativo. Debe contener solo lo necesario. Nuestros lectores deberían percibir que hemos sido decisivo.

"Big" Dave Thomas, fundador de OTI, padrino de la Estrategia de eclipse

> *El código limpio se puede leer y mejorar con un desarrollador que no sea su autor original. Tiene pruebas unitarias y de aceptación. Tiene significado nombres. Proporciona una forma en lugar de muchas formas de hacer una cosa. Tiene una dependencia mínima deficiencias, que se definen explícitamente, y pro- vides una API clara y mínima. El código debe ser*

alfabetizados ya que dependiendo del idioma, no todos
la información necesaria se puede expresar claramente
solo en código.

Big Dave comparte el deseo de Grady de legibilidad
ity, pero con un giro importante. Dave afirma que
El código limpio facilita que *otras* personas lo mejoren. Esto puede parecer obvio, pero puede
no se exagere. Después de todo, existe una diferencia entre el código que es fácil de leer
y código que es fácil de cambiar.

¡Dave relaciona la limpieza con las pruebas! Hace diez años esto habría levantado muchas cejas.
Pero la disciplina del desarrollo basado en pruebas ha tenido un profundo impacto en nuestra
industria y se ha convertido en una de nuestras disciplinas más fundamentales. Dave tiene razón. Código,
sin pruebas, no está limpio. No importa cuán elegante sea, no importa cuán legible y accesible sea
Sible, si no tiene pruebas, será inmundo.

Dave usa la palabra *mínimo* dos veces. Aparentemente, valora el código que es pequeño, más bien
que el código que es grande. De hecho, este ha sido un estribillo común en toda la literatura de software.
tura desde sus inicios. Cuanto más pequeño, mejor.

Dave también dice que el código debería ser *alfabetizado* . Esta es una referencia suave a la *letrada* de Knuth.
programación. [4] El resultado es que el código debe estar compuesto de tal forma que
es legible por humanos.

4. [Knuth92].

Página 41

Michael Feathers, autor de *Working*
Efectivamente con el código heredado

Podría enumerar todas las cualidades que noto en
código limpio, pero hay una cualidad general
que conduce a todos ellos. Código limpio siempre
parece que fue escrito por alguien a quien le importa.
No hay nada obvio que puedas hacer para
hacerlo mejor. Todas esas cosas fueron pensadas
sobre por el autor del código, y si intenta
imagina mejoras, volverás a
donde estás, sentado en apreciación de la
codificar que alguien te dejó, código dejado por alguien
uno que se preocupa profundamente por el oficio.

Una palabra: cuidado. Ese es realmente el tema de
este libro. Quizás un subtítulo apropiado
sería *Cómo cuidar el código* .

Michael lo golpeó en la cabeza. El código limpio es
código que se ha cuidado. Alguien se ha tomado el tiempo de hacerlo sencillo y ordenado.
Han prestado la debida atención a los detalles. Se han preocupado.

Ron Jeffries, autor de *Extreme Programming*
Programación instalada* y *extrema
Aventuras en C #

Ron comenzó su carrera programando en Fortran en
el Comando Aéreo Estratégico y ha escrito código en
casi todos los idiomas y en casi todos
máquina. Vale la pena considerar sus palabras con detenimiento.

En los últimos años empiezo, y casi acabo, con Beck's
reglas de código simple. En orden de prioridad, código simple:

• Ejecuta todas las pruebas;

• No contiene duplicaciones;

• Expresa todas las ideas de diseño que se encuentran en el sistema;

• Minimiza el número de entidades como clases, métodos, funciones y similares.

De estos, me centro principalmente en la duplicación. Cuando se hace lo mismo una y otra vez, es una señal de que hay una idea en nuestra mente que no está bien representada en el código. Yo intento averigua qué es. Luego trato de expresar esa idea con más claridad.

Para mí, la expresividad incluye nombres significativos, y es probable que cambie el nombres de cosas varias veces antes de instalarme. Con herramientas de codificación modernas como Eclipse, el cambio de nombre es bastante económico, por lo que no me molesta cambiarlo. La expresividad va

Página 42

más allá de los nombres, sin embargo. También miro si un objeto o método está haciendo más de una cosa. Si es un objeto, probablemente deba dividirse en dos o más objetos. Si es un método, siempre usaré el método de extracción para refactorizarlo, lo que da como resultado un método que dice más claramente lo que hace, y algunos submétodos dicen cómo se hace.

La duplicación y la expresividad me llevan muy lejos hacia lo que considero limpio. código, y mejorar el código sucio con solo estas dos cosas en mente puede hacer una gran diferencia ence. Sin embargo, hay otra cosa que soy consciente de hacer, que es un poco más difícil de explicar.

Después de años de hacer este trabajo, me parece que todos los programas se componen de muy elementos similares. Un ejemplo es "buscar cosas en una colección". Si tenemos datos base de registros de empleados, o un mapa hash de claves y valores, o una matriz de elementos de algunos amable, a menudo nos encontramos deseando un artículo en particular de esa colección. Cuando encuentro que suceda, a menudo envolveré la implementación particular en un método más abstracto o clase. Eso me da un par de ventajas interesantes.

Puedo implementar la funcionalidad ahora con algo simple, digamos un mapa hash, pero ya que ahora todas las referencias a esa búsqueda están cubiertas por mi pequeña abstracción, puedo cambiar la implementación cuando quiera. Puedo avanzar rápidamente mientras conservo mi capacidad de cambiar más tarde.

Además, la abstracción de la colección a menudo me llama la atención sobre lo que "realmente" continúa, y me impide correr por el camino de implementar una colección arbitraria comportamiento cuando todo lo que realmente necesito son algunas formas bastante simples de encontrar lo que quiero.

Duplicación reducida, alta expresividad y construcción temprana de abstracciones simples. Eso es lo que hace que el código sea limpio para mí.

Aquí, en unos breves párrafos, Ron ha resumido el contenido de este libro. No duplicación, una cosa, expresividad, diminutas abstracciones. Todo está ahí.

Ward Cunningham, inventor de Wiki, inventor de Fit, coinventor de eXtreme Programación. Fuerza motriz detrás Patrones de diseño. Smalltalk y OO líder de pensamiento. El padrino de todos aquellos que se preocupan por el código.

Sabes que estás trabajando en un código limpio cuando cada La rutina que lees resulta ser más o menos lo que lo esperabas. Puedes llamarlo código hermoso cuando el código también hace que parezca que el lenguaje fue hecho para el problema.

Declaraciones como esta son características de Ward. Lo lee, asiente con la cabeza y luego pasa al siguiente tema. Suena tan razonable, tan obvio, que apenas se registra como algo profundo. Podrías pensar que fue más o menos lo que esperabas. Pero vamos a acercarnos Mira.

Página 43

“. . . más o menos lo que esperabas ". ¿Cuándo fue la última vez que vio un módulo que fue más o menos lo que esperabas? ¿No es más probable que los módulos que mire ser desconcertante, complicado, enredado? ¿No es la mala dirección la regla? ¿No estás acostumbrado a agitarte? acerca de tratar de agarrar y sostener los hilos del razonamiento que brotan de todo el sistema tem y se abren paso a través del módulo que está leyendo? Cuando fue la última vez que leer un código y asentir con la cabeza de la forma en que podría haber asentido con la cabeza en la declaración de Ward?

Ward espera que cuando lea código limpio no se sorprenda en absoluto. De hecho, tu ni siquiera gastará mucho esfuerzo. Lo leerá y será más o menos lo que esperado. Será obvio, simple y convincente. Cada módulo preparará el escenario para el siguiente. Cada uno le dice cómo se escribirá el siguiente. Los programas que son *tan* limpios son tan profundamente bien escrito que ni siquiera lo notas. El diseñador hace que parezca ridículo terriblemente simple como todos los diseños excepcionales.

¿Y qué hay de la noción de belleza de Ward? Todos hemos criticado el hecho de que nuestro lenguaje Los medidores no fueron diseñados para nuestros problemas. Pero la declaración de Ward nos devuelve la responsabilidad. ¡Él dice que el código hermoso *hace que el lenguaje parezca hecho para el problema* ! Entonces ¡Es *nuestra* responsabilidad hacer que el lenguaje parezca simple! Fanáticos del lenguaje en todas partes, ¡tener cuidado! No es el lenguaje lo que hace que los programas parezcan simples. Es el programador que hacen que el lenguaje parezca simple!

Escuelas de pensamiento

¿Qué hay de mí (tío Bob)? Que pienso el código limpio es? Este libro te dirá, en espantoso detalle, lo que yo y mis compatriotas pensamos código limpio. Te diremos lo que creemos que hace un nombre de variable limpio, una función limpia, una limpia clase, etc. Presentaremos estas opiniones como absolutas laúdes, y no nos disculparemos por nuestra estridencia. Para nosotros, en este punto de nuestras carreras, *son* abso- laúdes. Son *nuestra escuela de pensamiento* sobre la limpieza. código.

No todos los artistas marciales están de acuerdo sobre lo mejor arte marcial, o la mejor técnica dentro de un marcial Arte. A menudo, los artistas marciales maestros formarán su propias escuelas de pensamiento y reunir a los estudiantes para Aprende de ellos. Entonces vemos a *Gracie Jiu Jistu* , fundada y enseñada por la familia Gracie en Brasil. Vemos *Hakkoryu Jiu Jistu* , fundado e impartido por Okuyama Ryuho en Tokio. Vemos *Jeet Kune Do* , fundado y enseñado por Bruce Lee en Estados Unidos.

Los estudiantes de estos enfoques se sumergen en las enseñanzas del fundador.
Se dedican a aprender lo que enseña ese maestro en particular, a menudo a los
sión de la enseñanza de cualquier otro maestro. Más tarde, a medida que los estudiantes crezcan en su arte, pueden
conviértase en alumno de un maestro diferente para que pueda ampliar su conocimiento y práctica.
Algunos eventualmente continúan perfeccionando sus habilidades, descubriendo nuevas técnicas y fundando sus
propias escuelas.

Ninguna de estas diferentes escuelas tiene toda la *razón* . Sin embargo, dentro de una escuela en particular
actuar como si las enseñanzas y técnicas *son* derecha. Después de todo, hay una forma correcta de practicar
tice Hakkoryu Jiu Jitsu, o Jeet Kune Do. Pero esta rectitud dentro de una escuela no invalida
Aprenda las enseñanzas de una escuela diferente.

Considere este libro como una descripción de *Object Mentor School of Clean Code* . La
técnicas y enseñanzas son la manera en que *nos* practicamos *nuestro* arte. Estamos dispuestos a
afirmar que si sigue estas enseñanzas, disfrutará de los beneficios que hemos disfrutado,
y aprenderá a escribir código limpio y profesional. Pero no cometas el error
de pensar que de alguna manera tenemos "razón" en un sentido absoluto. Hay otras escuelas y
otros maestros que tienen tanto reclamo de profesionalismo como nosotros. Te conviene
para aprender de ellos también.

De hecho, muchas de las recomendaciones de este libro son controvertidas. Probablemente
bly no estoy de acuerdo con todos ellos. Podrías estar en desacuerdo violentamente con algunos de ellos. Esta bien.
No podemos reclamar la autoridad final. Por otro lado, las recomendaciones de este libro son
cosas en las que hemos pensado mucho. Los hemos aprendido a lo largo de décadas de
experiencia y repetidas pruebas y errores. Entonces, ya sea que esté de acuerdo o en desacuerdo, sería una
Vergüenza si no vieras, y respetas, nuestro punto de vista.

Somos Autores

El campo @author de un Javadoc nos dice quiénes somos. Somos autores. Y una cosa sobre
autores es que tienen lectores. De hecho, los autores son *responsables* de comunicarse bien
con sus lectores. La próxima vez que escriba una línea de código, recuerde que es un autor,
escribiendo para lectores que juzgarán su esfuerzo.

Podría preguntar: ¿Cuánto se lee realmente el código? ¿No va la mayor parte del esfuerzo
escribirlo?

¿Ha reproducido alguna vez una sesión de edición? En los 80 y 90 tuvimos editores como Emacs
que mantuvo un registro de cada pulsación de tecla. Podrías trabajar durante una hora y luego reproducir todo tu
editar sesión como una película de alta velocidad. Cuando hice esto, los resultados fueron fascinantes.

¡La gran mayoría de la reproducción se desplazaba y navegaba a otros módulos!

Bob entra en el módulo.
Se desplaza hacia abajo hasta la función que necesita un cambio.
Hace una pausa, considerando sus opciones.
Oh, se desplaza hasta la parte superior del módulo para comprobar la inicialización de una variable.
Ahora se desplaza hacia abajo y comienza a escribir.

¡Vaya, está borrando lo que escribió!
Lo escribe de nuevo.
¡Lo vuelve a borrar!

Escribe la mitad de otra cosa, ¡pero luego la borra!
Se desplaza hacia abajo a otra función que llama a la función que está cambiando para ver cómo es.
llamada.
Se desplaza hacia atrás y escribe el mismo código que acaba de borrar.
Hace una pausa.
¡Borra ese código de nuevo!
Abre otra ventana y mira una subclase. ¿Se anula esa función?

. . .

Entiendes la deriva. De hecho, la relación entre el tiempo dedicado a leer y escribir es muy superior a 10: 1. Estamos *en constante* lectura de códigos de edad como parte del esfuerzo de escribir código nuevo.

Debido a que esta proporción es tan alta, queremos que la lectura del código sea fácil, incluso si hace la escritura más difícil. Por supuesto, no hay forma de escribir código sin leerlo, así *que hacerlo fácil de leer en realidad hace que sea más fácil de escribir* .

No hay escapatoria a esta lógica. No puede escribir código si no puede leer el sur- código de redondeo. El código que está intentando escribir hoy será difícil o fácil de escribir. dependiendo de qué tan fácil o difícil sea leer el código circundante. Entonces, si quieres ir rápido, si desea terminar rápidamente, si desea que su código sea fácil de escribir, facilítelo leer.

La regla de los Boy Scouts

No es suficiente escribir bien el código. El código debe *mantenerse limpio a lo* largo del tiempo. Tenemos todos visto código pudrirse y degradarse a medida que pasa el tiempo. Por tanto, debemos desempeñar un papel activo en la prevención de esta degradación.

Los Boy Scouts of America tienen una regla simple que podemos aplicar a nuestra profesión.

Deje el campamento más limpio de lo que lo encontró. [5]

Si todos registramos nuestro código un poco más limpio que cuando lo verificamos, el código simplemente no podía pudrirse. La limpieza no tiene por qué ser algo importante. Cambiar una variable nombre para mejor, divida una función que sea demasiado grande, elimine una pequeña parte de duplicación, limpie una declaración `if` compuesta .

¿Te imaginas trabajar en un proyecto donde el código *simplemente mejoró* con el tiempo? ¿aprobado? ¿Crees que alguna otra opción es profesional? De hecho, no es continuo la mejora es una parte intrínseca de la profesionalidad?

5. Esto fue adaptado del mensaje de despedida de Robert Stephenson Smyth Baden-Powell a los Scouts: "Intenta dejar este mundo en un poco mejor de lo que lo encontraste. . . "

Precuela y principios

En muchos sentidos, este libro es una "precuela" de un libro que escribí en 2002 titulado *Agile Software Desarrollo: Principios, Patrones y Prácticas* (PPP). El libro PPP se refiere a sí mismo con los principios del diseño orientado a objetos, y muchas de las prácticas utilizadas por los profesionales desarrolladores nacionales. Si no ha leído PPP, es posible que descubra que continúa la historia. contado por este libro. Si ya lo ha leído, encontrará muchos de los sentimientos de ese libro hizo eco en este a nivel de código.

En este libro encontrará referencias esporádicas a varios principios del diseño. Estas incluir el Principio de Responsabilidad Única (SRP), el Principio Abierto Cerrado (OCP), y el Principio de Inversión de Dependencia (DIP) entre otros. Estos principios se describen en profundidad en PPP.

Conclusión

Los libros de arte no prometen convertirte en un artista. Todo lo que pueden hacer es darte algunos de los herramientas, técnicas y procesos de pensamiento que otros artistas han utilizado. Así también este libro puede No prometo convertirte en un buen programador. No puede prometer darle un "sentido de código". Todo lo que puede hacer es mostrarle los procesos mentales de los buenos programadores y los trucos, la tecnología niques y herramientas que utilizan.

Al igual que un libro de arte, este libro estará lleno de detalles. Habrá mucho código. Verá un buen código y verá un código incorrecto. Verás mal código transformado en bueno código. Verá listas de heurísticas, disciplinas y técnicas. Verás un ejemplo después ejemplo. Después de eso, depende de ti.

Recuerde el viejo chiste sobre el violinista de concierto que se perdió en su camino hacia una actuación. mance? Detuvo a un anciano en la esquina y le preguntó cómo llegar al Carnegie Hall. El anciano miró al violinista y al violín bajo el brazo y dijo: tice, hijo. ¡Práctica!"

Bibliografía

[Beck07]: *Patrones de implementación* , Kent Beck, Addison-Wesley, 2007.

[Knuth92]: *Programación alfabetizada* , Donald E. Knuth, Centro para el estudio del lenguaje e información, Leland Stanford Junior University, 1992.

Página 47

Esta página se dejó en blanco intencionalmente

2

Nombres significativos

por Tim Ottinger

Introducción

Los nombres están en todas partes en el software. Nombramos nuestras variables, nuestras funciones, nuestros argumentos, clases y paquetes. Nombramos nuestros archivos fuente y los directorios que los contienen. Nosotros nombre nuestros archivos jar, war y ear. Nombramos y nombramos y nombramos. Porque lo hacemos

Página 49

tanto, será mejor que lo hagamos bien. Lo que sigue son algunas reglas simples para crear buenos nombres.

Utilice nombres que revelen intenciones

Es fácil decir que los nombres deben revelar la intención. Lo que queremos impresionarte es que nos tomamos en *serio* esto. Elegir buenos nombres lleva tiempo, pero ahorra más de lo necesario. Así que tenga cuidado con sus nombres y cámbielos cuando encuentre otros mejores. Todos quien lee tu código (incluyéndote a ti) será más feliz si lo haces.

El nombre de una variable, función o clase debe responder a todas las preguntas importantes. Eso debería decirle por qué existe, qué hace y cómo se usa. Si un nombre requiere una ment, entonces el nombre no revela su intención.

```
int d; // tiempo transcurrido en días
```

El nombre `d` no revela nada. No evoca una sensación de tiempo transcurrido ni de días. Nosotros debe elegir un nombre que especifique lo que se está midiendo y la unidad de esa medida- ment:

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

Elegir nombres que revelen la intención puede hacer que sea mucho más fácil de entender y cambiar. código. ¿Cuál es el propósito de este código?

```
Lista pública <int []> getThem () {
    List <int []> list1 = new ArrayList <int []> ();
    para (int [] x: theList)
        si (x [0] == 4)
            list1.add (x);
    return list1;
}
```

¿Por qué es difícil saber qué está haciendo este código? No hay expresiones complejas. El espacio y la sangría son razonables. Solo hay tres variables y dos constantes mencionado. Ni siquiera hay clases sofisticadas o métodos polimórficos, solo una lista de matrices (o eso parece).

El problema no es la simplicidad del código, sino la *implicidad* del código (para acuñar un frase): el grado en el que el contexto no es explícito en el código mismo. El código implícito requiere que sepamos las respuestas a preguntas tales como:

1. ¿Qué tipo de cosas hay en la `Lista` ?
2. ¿Cuál es el significado del subíndice cero de un elemento de la `Lista` ?
3. ¿Cuál es el significado del valor `4` ?
4. ¿Cómo utilizaría la lista que se devuelve?

Página 50

Las respuestas a estas preguntas no están presentes en el ejemplo de código, *pero podrían haber sido* . Digamos que estamos trabajando en un juego de barrendero. Encontramos que el tablero es una lista de celdas llamadas theList . Cambiemos el nombre de eso a gameBoard .

Cada celda del tablero está representada por una matriz simple. Además encontramos que el cero subíndice es la ubicación de un valor de estado y que un valor de estado de 4 significa "marcado". Sólo dando nombres a estos conceptos podemos mejorar el código considerablemente:

```
Lista pública <int []> getFlaggedCells () {
    List <int []> flaggedCells = new ArrayList <int []> ();
    para (int [] celda: gameBoard)
        si (celda [STATUS_VALUE] == FLAGGED)
            flaggedCells.add (celda);
    return flaggedCells;
}
```

Tenga en cuenta que la simplicidad del código no ha cambiado. Todavía tiene exactamente el mismo número de operadores y constantes, con exactamente el mismo número de niveles de anidamiento. Pero el codigo se ha vuelto mucho más explícito.

Podemos ir más allá y escribir una clase simple para celdas en lugar de usar una matriz de int s. Puede incluir una función que revele la intención (llámela isFlagged) para ocultar el número mágico. bers. Da como resultado una nueva versión de la función:

```
Public List <Cell> getFlaggedCells () {
    List <Cell> flaggedCells = new ArrayList <Cell> ();
    para (celda celda: tablero de juegos)
        si (celda.isFlagged ())
            flaggedCells.add (celda);
    return flaggedCells;
}
```

Con estos simples cambios de nombre, no es difícil entender qué está pasando. Esto es el poder de elegir buenos nombres.

Evite la desinformación

Los programadores deben evitar dejar pistas falsas que oscurezcan el significado del código. Deberíamos evite las palabras cuyos significados arraigados difieran de nuestro significado pretendido. Por ejemplo, hp , aix y sco serían nombres de variable deficientes porque son los nombres de la plataforma Unix. formas o variantes. Incluso si está codificando una hipotenusa y hp parece una buena abreviatura ción, podría ser desinformativo.

No se refiera a una agrupación de cuentas como una accountList a menos que en realidad sea una Lista . La lista de palabras significa algo específico para los programadores. Si el recipiente que contiene el cuentas no es en realidad una Lista , puede llevar a conclusiones falsas. [1] Así que accountGroup o bunchOfAccounts o simplemente cuentas simples serían mejores.

1. Como veremos más adelante, incluso si el contenedor *es* una Lista, probablemente sea mejor no codificar el tipo de contenedor en el nombre.

Página 51

Tenga cuidado con el uso de nombres que varíen en pequeñas formas. ¿Cuánto tiempo se tarda en detectar el sutil diferencia entre un XYZControllerForEfficientHandlingOfStrings en un módulo y, en algún lugar un poco más distante, XYZControllerForEfficientStorageOfStrings ? La las palabras tienen formas espantosamente similares.

Deletrear conceptos similares de manera similar es *información* . El uso de grafías inconsistentes se *dis-información* . Con los entornos Java modernos, disfrutamos de la finalización automática del código. Nosotros escriba algunos caracteres de un nombre y presione alguna combinación de teclas de acceso rápido (si es así) y recompensado con una lista de posibles finalizaciones para ese nombre. Es muy útil si los nombres de cosas muy similares se ordenan alfabéticamente y si las diferencias son muy obvias, porque es probable que el desarrollador elija un objeto por su nombre sin ver su abundante comentarios o incluso la lista de métodos proporcionados por esa clase.

Un ejemplo verdaderamente terrible de nombres desinformativos sería el uso de L minúscula o O mayúscula como nombres de variables, especialmente en combinación. El problema, por supuesto, es que se parecen casi por completo a las constantes uno y cero, respectivamente.

```
int a = l;
si (O == l)
    a = O1;
demás
    l = 01;
```

El lector puede pensar que esto es un truco, pero hemos examinado el código donde tal las cosas eran abundantes. En un caso, el autor del código sugirió usar una fuente diferente para que las diferencias fueran más obvias, una solución que tendría que transmitirse a todos los futuros desarrolladores como tradición oral o en un documento escrito. El problema esta vencido con finalidad y sin crear nuevos productos de obra mediante un simple cambio de nombre.

Hacer significativo Distinciones

Los programadores les crean problemas. ellos mismos cuando escriben código únicamente para sat- isfy un compilador o intérprete. Por ejemplo, porque no puedes usar el mismo nombre para referirte a dos cosas diferentes en el mismo ámbito, podría tener la tentación de cambiar un nombre de forma arbitraria. A veces, esto se hace al escribir mal uno, lo que lleva a la sorprendente situación en la que la corrección de errores ortográficos conduce a la imposibilidad de compilar. [2]

No es suficiente agregar series numéricas o palabras irrelevantes, aunque el compilador sea satisfecho. Si los nombres deben ser diferentes, también deben significar algo diferente.

2. Considere, por ejemplo, la práctica verdaderamente espantosa de crear una variable llamada klass solo porque se usó la clase de nombre por otra cosa.

La denominación de series de números (a1, a2, .. aN) es lo opuesto a la denominación intencional. Semejante los nombres no son desinformativos, no son informativos; no proporcionan ninguna pista sobre el intención del autor. Considerar:

```
public static void copyChars (char a1 [], char a2 []) {
    para (int i = 0; i <a1.length; i ++) {
        a2 [i] = a1 [i];
    }
}
```

Esta función se lee mucho mejor cuando se utilizan el origen y el destino para el argumento

nombres.

Las palabras ruidosas son otra distinción sin sentido. Imagina que tienes un Producto clase. Si tiene otro llamado ProductInfo o ProductData , ha hecho los nombres diferentes diferente sin hacer que signifiquen nada diferente. La información y los datos son un ruido indistinto palabras como a , an y the .

Tenga en cuenta que no hay nada malo con el uso de prefijos convenciones como una y la siempre ya que hacen una distinción significativa. Por ejemplo, puede utilizar un para todas las variables locales. y la de todos los argumentos de la función. [3] El problema surge cuando decide llamar a una variante puede theZork porque ya tiene otra variable llamada zork .

Las palabras ruidosas son redundantes. La palabra variable nunca debe aparecer en una variable. nombre. La palabra tabla nunca debe aparecer en el nombre de una tabla. ¿Cómo es NameString mejor que Nombre ? ¿ Alguna vez un nombre sería un número de coma flotante? Si es así, rompe una regla anterior sobre desinformación. Imagínese encontrar una clase llamada Cliente y otra llamada CustomerObject . ¿Qué debes entender como distinción? Cual representara ¿Cuál es el mejor camino hacia el historial de pagos de un cliente?

Hay una aplicación que conocemos donde se ilustra esto. hemos cambiado los nombres para proteger a los culpables, pero aquí está la forma exacta del error:

```
getActiveAccount ();
getActiveAccounts ();
getActiveAccountInfo ();
```

¿Cómo se supone que los programadores de este proyecto deben saber cuál de estas funciones llamar?

En ausencia de convenciones específicas, la variable moneyAmount es indistinguible de dinero , customerInfo es indistinguible de customer , accountData es indistinguible- capaz de la cuenta , y el mensaje es indistinguible del mensaje . Distinguir nombres en de tal manera que el lector sepa lo que ofrecen las diferencias.

Utilice nombres pronunciables

Los humanos somos buenos con las palabras. Una parte importante de nuestro cerebro está dedicada al concepto de palabras. Y las palabras son, por definición, pronunciables. Seria una pena no tomar

3. El tío Bob solía hacer esto en C ++ pero ha abandonado la práctica porque los IDE modernos lo hacen innecesario.

Página 53

ventaja de esa gran parte de nuestro cerebro que ha evolucionado para lidiar con el lenguaje hablado calibre. Así que haz que tus nombres se puedan pronunciar.

Si no puede pronunciarlo, no puede discutirlo sin sonar como un idiota. "Bien, aquí en el bee cee arr tres cee enn tee tenemos un pipí zee kyew int, ¿ves? Esto importa porque la programación es una actividad social.

Una empresa que conozco tiene genymdhms (fecha de generación, año, mes, día, hora, minuto, y segundo) así que caminaron diciendo “gen por qué emm dee aich emm ess”. Yo tengo un molesto hábito de pronunciar todo como está escrito, así que comencé a decir "gen-yah-mudda- él. " Más tarde, una gran cantidad de diseñadores y analistas lo llamaron así, y todavía sonaba tonto. Pero estábamos en la broma, así que fue divertido. Divertido o no, toleramos mala denominación. Los nuevos desarrolladores tenían que tener explicadas las variables, y luego Habló de ello con tontas palabras inventadas en lugar de utilizar los términos adecuados en inglés. Comparar

```
class DtaRcrd102 {
    genymdhms fecha privada;
    fecha privada modymdhms;
    Cadena final privada pszqint = "102";
    / * ... * /
};
```

a

```
clase Cliente {
    Private Date generationTimestamp;
```

```
    fecha privadamodificaciónTimestamp ;;
    Private final String recordId = "102";
    / * ... * /
};
```

Ahora es posible una conversación inteligente: “¡Oye, Mikey, echa un vistazo a este disco! El gen-
La marca de tiempo de la eración se establece en la fecha de mañana ¿Como puede ser?"

Usar nombres que se pueden buscar

Los nombres de una sola letra y las constantes numéricas tienen un problema particular en el sentido de que no fácil de localizar en un cuerpo de texto.

Uno podría fácilmente grep para MAX_CLASSES_PER_STUDENT , pero el número 7 podría ser más molesto. Las búsquedas pueden aumentar el dígito como parte de los nombres de archivo, otras definiciones constantes. ciones, y en varias expresiones donde el valor se utiliza con diferente intención. Es incluso peor cuando una constante es un número largo y alguien podría haber transpuesto dígitos, creando así un error y evitando simultáneamente la búsqueda del programador.

Asimismo, el nombre e es una mala elección para cualquier variable para la que un programador podría Necesito buscar. Es la letra más común en el idioma inglés y es probable que aparezca en cada pasaje de texto en cada programa. En este sentido, los nombres más largos triunfan sobre los más cortos. nombres, y cualquier nombre que se pueda buscar triunfa sobre una constante en el código.

Mi preferencia personal es que los nombres de una sola letra SÓLO se pueden usar como variantes locales. ables dentro de métodos cortos. *La longitud de un nombre debe corresponder al tamaño de su alcance.*

[N5]. Si una variable o constante puede verse o usarse en varios lugares en un cuerpo de código, es imperativo darle un nombre que permita la búsqueda. Una vez más comparar

```
para (int j = 0; j <34; j ++) {
    s + = (t [j] * 4) / 5;
}
```

a

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int suma = 0;
para (int j = 0; j <NÚMERO_DE_TAREAS; j ++) {
    int realTaskDays = taskEstimate [j] * realDaysPerIdealDay;
    int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
    suma + = realTaskWeeks;
}
```

Tenga en cuenta que suma , arriba, no es un nombre particularmente útil, pero al menos se puede buscar. La El código nombrado intencionalmente hace que la función sea más larga, pero considere cuánto más fácil será encontrar WORK_DAYS_PER_WEEK que buscar todos los lugares donde se usó 5 y filtrar la lista hasta solo las instancias con el significado deseado.

Evite las codificaciones

Tenemos suficientes codificaciones con las que lidiar sin agregar más a nuestra carga. Codificación la información de tipo o alcance en los nombres simplemente agrega una carga adicional de descifrado. Eso Difícilmente parece razonable exigir que cada nuevo empleado aprenda otro lenguaje de codificación calibre ”además de aprender el (generalmente considerable) cuerpo de código que trabajarán- Es una carga mental innecesaria cuando se trata de resolver un problema. Nombres codificados rara vez se pronuncian y son fáciles de escribir incorrectamente.

Notación húngara

En los días de antaño, cuando trabajábamos en idiomas que desafiaban la longitud del nombre, violábamos esta descartar por necesidad y con pesar. Fortran forzó las codificaciones haciendo que la primera letra código para el tipo. Las primeras versiones de BASIC permitían solo una letra más un dígito. húngaro

La notación (HN) llevó esto a un nivel completamente nuevo.

Se consideraba que HN era bastante importante en la API de Windows C, cuando todos cosa era un identificador de enteros o un puntero largo o un puntero vacío , o una de varias implementaciones tations de “string” (con diferentes usos y atributos). El compilador no comprobó los tipos en esos días, por lo que los programadores necesitaban una muleta para ayudarlos a recordar los tipos.

En los lenguajes modernos tenemos sistemas de tipos mucho más ricos, y los compiladores recuerdan y hacer cumplir los tipos. Además, hay una tendencia hacia clases más pequeñas y más cortas. funciones para que las personas puedan ver el punto de declaración de cada variable que están utilizando.

Página 55

Los programadores de Java no necesitan codificación de tipos. Los objetos están fuertemente tipados y la edición Los entornos han avanzado de tal manera que detectan un error de tipo mucho antes de que pueda ejecutar un ¡compilar! Así que hoy en día HN y otras formas de codificación de tipos son simplemente impedimentos. Hacen que sea más difícil cambiar el nombre o el tipo de una variable, función o clase. Ellos dificultar la lectura del código. Y crean la posibilidad de que el sistema de codificación engañará al lector.

```
PhoneNumber phoneString;
// ¡el nombre no cambió cuando cambió el tipo!
```

Prefijos de miembros

Tampoco necesita prefijar las variables miembro con m_ nunca más. Tus clases y funciones Las cantidades deben ser lo suficientemente pequeñas como para que no las necesite. Y deberías estar usando una edición entorno que resalta o colorea a los miembros para hacerlos distintos.

```
parte de clase pública {
    private String m_dsc; // La descripción textual
    void setName (nombre de cadena) {
        m_dsc = nombre;
    }
}
```

```
parte de clase pública {
    Descripción de cadena;
    void setDescription (descripción de la cadena) {
        this.description = descripción;
    }
}
```

Además, las personas aprenden rápidamente a ignorar el prefijo (o sufijo) para ver el significado parte del nombre. Cuanto más leemos el código, menos vemos los prefijos. Eventualmente el los prefijos se convierten en un desorden invisible y un marcador de código más antiguo.

Interfaces e implementaciones

A veces se trata de un caso especial para las codificaciones. Por ejemplo, digamos que está construyendo un A ESUMEN F actory para la creación de formas. Esta fábrica será una interfaz y ser implementado por una clase concreta. ¿Cómo debería nombrarlos? IShapeFactory y ShapeFactory ? Prefiero dejar las interfaces sin adornos. El anterior yo , tan común en Los tacos heredados de hoy son una distracción en el mejor de los casos y demasiada información en el peor. Yo no quiero que mis usuarios sepan que les estoy entregando una interfaz. Solo quiero que sepan que es una ShapeFactory . Entonces, si debo codificar la interfaz o la implementación, elijo la implementación. Llamarlo ShapeFactoryImp , o incluso el horrible CShapeFactory , es preferible era posible codificar la interfaz.

Página 56

Evite el mapeo mental

Los lectores no deberían tener que traducir mentalmente sus nombres a otros nombres que ya saber. Este problema generalmente surge de la elección de no utilizar ni los términos del dominio del problema. ni términos de dominio de solución.

Este es un problema con los nombres de variables de una sola letra. Ciertamente, un contador de bucle puede ser llamado i o j o k (¡aunque nunca l !) si su alcance es muy pequeño y ningún otro nombre puede pelear con él. Esto se debe a que los nombres de una sola letra para los contadores de bucles son tradicionales. Sin embargo, en la mayoría de los demás contextos, un nombre de una sola letra es una mala elección; es solo un lugar titular que el lector debe mapear mentalmente el concepto real. No puede haber peor razon hijo para usar el nombre de c que por una y B estaban ocupadas.

En general, los programadores son personas bastante inteligentes. A las personas inteligentes a veces les gusta mostrar fuera de su inteligencia demostrando sus habilidades mentales de malabarismo. Después de todo, si puedes confiar Recuerde hábilmente que r es la versión en minúsculas de la URL con el host y el esquema eliminado, entonces claramente debes ser muy inteligente.

Una diferencia entre un programador inteligente y un programador profesional es que el profesional entiende que la *claridad es el rey* . Los profesionales usan sus poderes para el bien y escribir código que otros puedan entender.

Nombres de clases

Las clases y los objetos deben tener nombres o frases nominales como Cliente , WikiPage , Cuenta y AddressParser . Evite palabras como Administrador , Procesador , Datos o Información en el nombre de una clase. El nombre de una clase no debe ser un verbo.

Nombres de métodos

Los métodos deben tener nombres de verbos o frases verbales como postPayment , deletePage o save . Los descriptores de acceso, mutantes y predicados se deben nombrar por su valor y tener el prefijo get , set , y está de acuerdo con el estándar javabean. [4]

```
nombre de cadena = empleado.getName ();
customer.setName ("mike");
si (cheque de pago.isPosted ()) ...
```

Cuando los constructores están sobrecargados, use métodos de fábrica estáticos con nombres que describe los argumentos. Por ejemplo,

```
FulcrumPoint complejo = Complex.FromRealNumber (23.0);
```

es generalmente mejor que

```
FulcrumPoint complejo = nuevo complejo (23.0);
```

Considere hacer cumplir su uso haciendo que los constructores correspondientes sean privados.

4. http://java.sun.com/products/javabeans/docs/spec.html

Página 57

No seas lindo

Si los nombres son demasiado inteligentes, serán
memorable solo para las personas que comparten el
sentido del humor del autor, y solo mientras
como estas personas recuerdan el chiste. Voluntad
ellos saben como se llama la función
¿ Se supone que HolyHandGrenade debe hacer? Seguro,
es lindo, pero tal vez en este caso
DeleteItems podría ser un nombre mejor.
Elija la claridad sobre el valor del entretenimiento.

La ternura en el código a menudo aparece en forma de coloquialismos o jerga. Por ejemplo, no use el nombre whack () para significar kill () . No cuentes pequeños chistes que dependan de la cultura como eatMyShorts () significa abortar () .

Di lo que quieres decir. Significa lo que dices.

Elija una palabra por concepto

Elija una palabra para un concepto abstracto y apéguese a ella. Por ejemplo, es confuso tienen fetch , retrieve y get como métodos equivalentes de diferentes clases. Cómo ¿Recuerdas qué nombre de método va con qué clase? Lamentablemente, a menudo tienes que recordar qué empresa, grupo o individuo escribió la biblioteca o clase para recordar qué se utilizó el término. De lo contrario, pasará mucho tiempo navegando por los encabezados y muestras de código anteriores.

Los entornos de edición modernos como Eclipse e IntelliJ proporcionan pistas sensibles al contexto, como la lista de métodos a los que puede llamar en un objeto determinado. Pero tenga en cuenta que la lista no suele Ally le dará los comentarios que escribió sobre los nombres de sus funciones y listas de parámetros. Tiene suerte si da los *nombres* de los parámetros de las declaraciones de funciones. La función Los nombres deben ser independientes y deben ser consistentes para que pueda elegir el método rect sin ninguna exploración adicional.

Asimismo, es confuso tener un controlador y un gerente y un conductor en el mismo base de código. ¿Cuál es la diferencia esencial entre un DeviceManager y un protocolo? Controlador ? ¿Por qué ambos no son controladores o ambos no son administradores ? ¿Son ambos conductores? ¿De Verdad? El nombre le lleva a esperar dos objetos que tienen un tipo muy diferente, así como tener diferentes clases.

Un léxico consistente es una gran ayuda para los programadores que deben usar su código.

No hagas juegos de palabras

Evite usar la misma palabra para dos propósitos. Usando el mismo término para dos ideas diferentes es esencialmente un juego de palabras.

Si sigue la regla de "una palabra por concepto", podría terminar con muchas clases que tienen, por ejemplo, un método add . Siempre que las listas de parámetros y los valores de retorno de los diversos métodos de adición son semánticamente equivalentes, todo está bien.

Sin embargo, uno podría decidir usar la palabra agregar para "coherencia" cuando no está de hecho agregando en el mismo sentido. Digamos que tenemos muchas clases donde add creará un nuevo valor agregando o concatenando dos valores existentes. Ahora digamos que estamos escribiendo un nueva clase que tiene un método que coloca su único parámetro en una colección. Deberíamos llamar este método agregar ? Puede parecer coherente porque tenemos muchos otros métodos de adición , pero en este caso, la semántica es diferente, por lo que deberíamos usar un nombre como insertar o agregar en lugar de. Llamar al nuevo método add sería un juego de palabras.

Nuestro objetivo, como autores, es hacer que nuestro código sea lo más fácil de entender posible. Queremos nuestro código debe ser un rápido vistazo, no un estudio intenso. Queremos utilizar el popular libro de bolsillo modelo en el que el autor es responsable de ser claro y no el académico modelo en el que el trabajo del erudito es extraer el significado del papel.

Usar nombres de dominio de solución

Recuerde que las personas que lean su código serán programadores. Así que adelante y usa términos de informática (CS), nombres de algoritmos, nombres de patrones, términos matemáticos, etc. Eso No es prudente extraer todos los nombres del dominio del problema porque no queremos nuestro los compañeros de trabajo tienen que ir y venir al cliente preguntando qué significa cada nombre cuando ya conocen el concepto con un nombre diferente.

El nombre AccountVisitor significa mucho para un programador familiarizado con el patrón V ISITOR . ¿Qué programador no sabría qué es JobQueue ? Existen muchas cosas muy técnicas que los programadores tienen que hacer. Elegir nombres técnicos para esas cosas suele ser el curso más apropiado.

Usar nombres de dominio problemáticos

Cuando no hay un "programador-eese" para lo que está haciendo, use el nombre del problema dominio lem. Al menos el programador que mantiene su código puede preguntarle a un experto en dominios lo que significa.

Separar los conceptos de dominio de solución y problema es parte del trabajo de un buen programmer y diseñador. El código que tiene más que ver con los conceptos de dominio de problemas deben tener nombres extraídos del dominio del problema.

Agregar contexto significativo

Hay algunos nombres que son significativos en sí mismos, la mayoría no lo son. En lugar de, necesita colocar nombres en contexto para su lector encerrándolos en nombres bien nombrados clases, funciones o espacios de nombres. Cuando todo lo demás falla, entonces puede ser necesario prefijar el nombre. essary como último recurso.

Página 59

Imagine que tiene variables llamadas firstName , lastName , street , houseNumber , city , estado y código postal . Tomados en conjunto, está bastante claro que forman una dirección. Pero que si ¿Acaba de ver que la variable de estado se usa sola en un método? ¿Lo harías automáticamente inferir que era parte de una dirección?

Puede agregar contexto usando prefijos: addrFirstName , addrLastName , addrState , etc. en. Al menos los lectores comprenderán que estas variables son parte de una estructura más amplia. De Por supuesto, una mejor solución es crear una clase llamada Dirección . Entonces, incluso el compilador sabe que las variables pertenecen a un concepto más amplio.

Considere el método del Listado 2-1. ¿Necesitan las variables una contexto? El nombre de la función proporciona solo una parte del contexto; el algoritmo proporciona el resto. Una vez que lea la función, verá que las tres variables, número , verbo y pluralModifier , son parte del mensaje de "estadísticas de conjetura". Desafortunadamente, el contexto debe inferirse. Cuando mira por primera vez el método, los significados de las variables son opacos.

Listado 2-1

Variables con contexto poco claro.

```
private void printGuessStatistics (candidato de carácter, recuento de int) {
        Número de cadena;
        Verbo de cadena;
        String pluralModifier;
        si (cuenta == 0) {
            número = "no";
            verbo = "son";
            pluralModifier = "s";
        } más si (recuento == 1) {
            número = "1";
            verbo = "es";
            pluralModifier = "";
        } demás {
            número = Integer.toString (recuento);
            verbo = "son";
            pluralModifier = "s";
        }
        String guessMessage = String.format (
            "Hay% s% s% s% s", verbo, número, candidato, pluralModificador
        );
        imprimir (adivinarMessage);
    }
```

La función es demasiado larga y las variables se utilizan en todo momento. Para dividir la función en trozos más pequeños, necesitamos crear una clase GuessStatisticsMessage y hacer que el tres campos de variables de esta clase. Esto proporciona un contexto claro para las tres variables. Ellos son *definitivamente* parte del GuessStatisticsMessage . La mejora del contexto también permite el algoritmo se hace mucho más limpio dividiéndolo en muchas funciones más pequeñas. (Ver Listado 2-2.)

Página 60

No agregue contexto gratuito 29

Listado 2-2

Las variables tienen un contexto.

```
public class GuessStatisticsMessage {
    número de cadena privada;
    verbo de cadena privado;
    private String pluralModifier;

    public String make (char candidato, int count) {
        createPluralDependentMessageParts (recuento);
        return String.format (
            "Hay% s% s% s% s",
              verbo, número, candidato, modificador plural);
    }

    private void createPluralDependentMessageParts (recuento int) {
        si (cuenta == 0) {
            thereAreNoLetters ();
        } más si (recuento == 1) {
            thereIsOneLetter ();
        } demás {
            thereAreManyLetters (recuento);
        }
    }

    vacío privado thereAreManyLetters (int count) {
        número = Integer.toString (recuento);
        verbo = "son";
        pluralModifier = "s";
    }

    private void thereIsOneLetter () {
        número = "1";
        verbo = "es";
        pluralModifier = "";
    }
```

```
    vacío privado thereAreNoLetters () {
        número = "no";
        verbo = "son";
        pluralModifier = "s";
    }
}
```

No agregue contexto gratuito

En una aplicación imaginaria llamada "Gas Station Deluxe", es una mala idea prefijar cada clase con GSD . Francamente, estás trabajando en contra de tus herramientas. Escribe G y presiona el botón completa y son recompensados con una lista de una milla de cada clase en el sistema. Es eso ¿sabio? ¿Por qué dificultar la ayuda del IDE?

Asimismo, supongamos que inventó una clase MailingAddress en el módulo de contabilidad de GSD y lo llamó GSDAccountAddress . Más tarde, necesitará una dirección postal para el contrato de su cliente. aplicación de tacto. ¿Utiliza GSDAccountAddress ? ¿Suena como el nombre correcto? Diez de 17 caracteres son redundantes o irrelevantes.

Página 61

Los nombres más cortos son generalmente mejores que los más largos, siempre que sean claros. Agregar no más contexto para un nombre de lo necesario.

Los nombres accountAddress y customerAddress son buenos nombres para instancias de class Address, pero podrían ser nombres incorrectos para las clases. La dirección es un buen nombre para una clase. Si yo necesito diferenciar entre direcciones MAC, direcciones de puerto y direcciones web, podría considere PostalAddress , MAC y URI . Los nombres resultantes son más precisos, que es el punto de todos los nombres.

Ultimas palabras

Lo más difícil de elegir buenos nombres es que requiere buenas habilidades descriptivas y un trasfondo cultural compartido. Este es un problema de enseñanza más que técnico, comercial o problema de gestión. Como resultado, muchas personas en este campo no aprenden a hacerlo muy bien.

La gente también tiene miedo de cambiar el nombre de las cosas por temor a que otros desarrolladores objeto. No compartimos ese miedo y descubrimos que en realidad estamos agradecidos cuando los nombres cambian. (para el mejor). La mayoría de las veces no memorizamos realmente los nombres de las clases y los métodos. ods. Usamos las herramientas modernas para tratar detalles como ese para poder enfocarnos en si el El código se lee como párrafos y oraciones, o al menos como tablas y estructura de datos (una frase tence no siempre es la mejor manera de mostrar datos). Probablemente terminarás sorprendiendo a algunos uno cuando cambia el nombre, al igual que lo haría con cualquier otra mejora de código. No lo dejes detenerte en seco.

Siga algunas de estas reglas y vea si no mejora la legibilidad de su código. Si está manteniendo el código de otra persona, use herramientas de refactorización para ayudar a resolver estos problemas. Valdrá la pena a corto plazo y seguirá rindiendo a largo plazo.

Página 62

3

Funciones

En los primeros días de la programación compusimos nuestros sistemas de rutinas y subrutinas. Luego, en la era de Fortran y PL / 1 compusimos nuestros sistemas de programas, subprogramas, y funciones. Hoy en día solo la función sobrevive de aquellos primeros días. Las funciones son la primera línea de organización en cualquier programa. Escribirlos bien es el tema de este capítulo.

Página 63

Considere el código del Listado 3-1. Es difícil encontrar una función larga en FitNesse, [1] pero después de buscar un poco me encontré con este. No solo es largo, sino que se ha duplicado. código, un montón de cadenas extrañas y muchos tipos de datos y API extraños e inofensivos. Ver cómo tiene mucho sentido en los próximos tres minutos.

Listado 3-1

HtmlUtil.java (FitNesse 20070619)

```
cadena estática pública testableHtml (
    PageData pageData,
    boolean includeSuiteSetup
) lanza Exception {
    WikiPage wikiPage = pageData.getWikiPage ();
    Búfer StringBuffer = nuevo StringBuffer ();
    if (pageData.hasAttribute ("Prueba")) {
        if (includeSuiteSetup) {
            WikiPage suiteSetup =
                PageCrawlerImpl.getInheritedPage (
                        SuiteResponder.SUITE_SETUP_NAME, wikiPage
                );
            if (suiteSetup! = null) {
                WikiPagePath pagePath =
                    suiteSetup.getPageCrawler (). getFullPath (suiteSetup);
                String pagePathName = PathParser.render (pagePath);
                buffer.append ("! include -setup.")
                        .append (pagePathName)
                        .append ("\ n");
            }
        }
        Configuración de WikiPage =
            PageCrawlerImpl.getInheritedPage ("Configuración", wikiPage);
        if (configuración! = nulo) {
            WikiPagePath setupPath =
                wikiPage.getPageCrawler (). getFullPath (configuración);
            String setupPathName = PathParser.render (setupPath);
            buffer.append ("! include -setup.")
                    .append (setupPathName)
                    .append ("\ n");
        }
    }
    buffer.append (pageData.getContent ());
    if (pageData.hasAttribute ("Prueba")) {
        Desmontaje de WikiPage =
            PageCrawlerImpl.getInheritedPage ("TearDown", wikiPage);
        if (desmontaje! = nulo) {
            WikiPagePath tearDownPath =
                wikiPage.getPageCrawler (). getFullPath (desmontaje);
            String tearDownPathName = PathParser.render (tearDownPath);
            buffer.append ("\ n")
                    .append ("! include -teardown.")
                    .append (tearDownPathName)
                    .append ("\ n");
        }
```

1. Una herramienta de prueba de código abierto. www.fitnese.org

Página 64

Listado 3-1 (continuación)
HtmlUtil.java (FitNesse 20070619)

```
        if (includeSuiteSetup) {
            Suite WikiPageTeardown =
                PageCrawlerImpl.getInheritedPage (
                            SuiteResponder.SUITE_TEARDOWN_NAME,
                            wikiPage
                );
            if (suiteTeardown! = null) {
                WikiPagePath pagePath =
                    suiteTeardown.getPageCrawler (). getFullPath (suiteTeardown);
                String pagePathName = PathParser.render (pagePath);
                buffer.append ("! include -teardown.")
                        .append (pagePathName)
                        .append ("\ n");
            }
        }
    }
    pageData.setContent (buffer.toString ());
    return pageData.getHtml ();
}
```

¿Entiendes la función después de tres minutos de estudio? Probablemente no. Hay
hay demasiadas cosas sucediendo allí en demasiados niveles diferentes de abstracción. Hay extraños
cadenas y llamadas a funciones impares mezcladas con declaraciones if doblemente anidadas controladas por
banderas.

Sin embargo, con solo algunas extracciones de métodos simples, algunos cambios de nombre y un poco
reestructuración, pude capturar la intención de la función en las nueve líneas del Listado 3-2.
Vea si puede entender *eso* en los próximos 3 minutos.

Listado 3-2
HtmlUtil.java (refactorizado)

```
cadena estática pública renderPageWithSetupsAndTeardowns (
    PageData pageData, booleano isSuite
) lanza Exception {
    boolean isTestPage = pageData.hasAttribute ("Prueba");
    if (isTestPage) {
        WikiPage testPage = pageData.getWikiPage ();
        StringBuffer newPageContent = nuevo StringBuffer ();
        includeSetupPages (testPage, newPageContent, isSuite);
        newPageContent.append (pageData.getContent ());
        includeTeardownPages (testPage, newPageContent, isSuite);
        pageData.setContent (newPageContent.toString ());
    }

    return pageData.getHtml ();
}
```


Página 65

A menos que sea un estudiante de FitNesse, probablemente no comprenda todos los detalles.
Aún así, probablemente comprenda que esta función realiza la inclusión de alguna configuración y
desmontar páginas en una página de prueba y luego renderiza esa página en HTML. Si estas familiarizado
con JUnit, [2] probablemente se dé cuenta de que esta función pertenece a algún tipo de
marco de prueba. Y, por supuesto, eso es correcto. Adivinando esa información del Listado 3-2
es bastante fácil, pero está bastante oculto por el Listado 3-1.

Entonces, ¿qué es lo que hace que una función como el Listado 3-2 sea fácil de leer y comprender? Cómo
¿Podemos hacer que una función comunique su intención? ¿Qué atributos podemos dar a nuestras funciones?
que permitirá a un lector casual intuir el tipo de programa en el que viven?

¡Pequeña!

La primera regla de funciones es que deben ser pequeñas. La segunda regla de funciones es que *deberían ser más pequeños que eso* . Esta no es una afirmación que pueda justificar. No puedo proporcionar cualquier referencia a investigaciones que demuestren que las funciones muy pequeñas son mejores. Lo que puedo decir es que durante casi cuatro décadas he escrito funciones de todos los tamaños diferentes. He escrito Diez varias abominaciones desagradables de 3.000 líneas. He escrito montones de funciones del 100 al 300 rango de línea. Y he escrito funciones que tenían entre 20 y 30 líneas de longitud. Que esta experiencia me ha enseñado, a través de una larga prueba y error, que las funciones deben ser muy pequeñas.

En los años ochenta solíamos decir que una función no debería ser más grande que una pantalla completa. Por supuesto, dijimos que en un momento en que las pantallas VT100 tenían 24 líneas por 80 columnas, y nuestros editores utilizaron 4 líneas para fines administrativos. Hoy en día con una fuente reducida y un monitor grande y agradable, puede colocar 150 caracteres en una línea y 100 líneas o más en una pantalla. Las líneas no deben tener 150 caracteres. Las funciones no deben tener una longitud de 100 líneas. Las funciones casi nunca deberían tener una longitud de 20 líneas.

¿Qué tan corta debe ser una función? En 1999 fui a visitar a Kent Beck a su casa en Oregon. gon. Nos sentamos y programamos juntos. En un momento me mostró un lindo pequeño programa Java / Swing que llamó *Sparkle* . Produjo un efecto visual en la pantalla. muy similar a la varita mágica del hada madrina de la película Cenicienta. Como tu movió el mouse, los destellos gotearían del cursor con un centelleo satisfactorio, cayendo al fondo de la ventana a través de un campo gravitacional simulado. Cuando Kent me mostró el código, me sorprendió lo pequeñas que eran todas las funciones. Estaba acostumbrado a funcionar ciones en programas de swing que ocupaban millas de espacio vertical. Cada función en *este* programa tenía sólo dos, tres o cuatro líneas. Cada uno era claramente obvio. Cada uno dijo una historia. Y cada uno te llevó al siguiente en un orden convincente. *Así de* cortas son tus funciones ¡debiera ser! [3]

2. Una herramienta de prueba unitaria de código abierto para Java. www.junit.org
3. Le pregunté a Kent si todavía tenía una copia, pero no pudo encontrar una. También busqué en todas mis computadoras viejas, pero fue en vano. Todo lo que queda ahora es mi memoria de ese programa.

Página 66

¿Qué tan breves deben ser sus funciones? Por lo general, deberían ser más cortos que el Listado 3-2. De hecho, el Listado 3-2 realmente debería reducirse al Listado 3-3.

Listado 3-3

HtmlUtil.java (refactorizado)

```
cadena estática pública renderPageWithSetupsAndTeardowns (
    PageData pageData, boolean isSuite) lanza Exception {
    si (isTestPage (pageData))
        includeSetupAndTeardownPages (pageData, isSuite);
    return pageData.getHtml ();
}
```

Bloques y sangría

Esto implica que los bloques dentro de declaraciones if , declaraciones else , declaraciones while y así sucesivamente debe tener una línea de longitud. Probablemente esa línea debería ser una llamada a función. No solo lo hace Esto mantiene pequeña la función de encerrar, pero también agrega valor documental porque el La función llamada dentro del bloque puede tener un nombre muy descriptivo.

Esto también implica que las funciones no deben ser lo suficientemente grandes para contener estructuras anidadas. Por lo tanto, el nivel de sangría de una función no debe ser mayor que uno o dos. Esto, de Por supuesto, hace que las funciones sean más fáciles de leer y comprender.

Haz una cosa

Debe quedar muy claro que el Listado 3-1 está haciendo mucho más de una cosa. Está creando búferes, recuperando páginas, búsqueda de páginas heredadas, rutas de renderizado, agregando cadenas arcanas y generando HTML, entre otras cosas. El Listado 3-1 está muy ocupado haciendo muchas cosas diferentes. Por otro lado, el Listado 3-3 está haciendo una cosa simple. Incluye configuraciones y desmonta en páginas de prueba.

El siguiente consejo ha aparecido en una forma u otro por 30 años o más.

> ***F** UNCIONES DEBE hacer una cosa . **T** HEY DEBE hacerlo bien .*
> ***T** HEY debe hacerlo sólo .*

El problema con esta afirmación es que es difícil saber qué es "una cosa". Lo hace Listado 3-3 ¿hace una cosa? Es fácil argumentar que está haciendo tres cosas:

1. Determinar si la página es una página de prueba.
2. Si es así, incluidas las configuraciones y desmontajes.
3. Representación de la página en HTML.

Página 67

Entonces, ¿cuál es? ¿La función está haciendo una o tres cosas? Note que los tres los pasos de la función son un nivel de abstracción por debajo del nombre indicado de la función. Nosotros puede describir la función describiéndola como un breve párrafo *TO* [4] :

> *PARA RenderPageWithSetupsAndTeardowns, verificamos si la página es una página de prueba y si es así, incluimos las configuraciones y desmontajes. En cualquier caso, representamos la página en HTML.*

Si una función realiza solo los pasos que están un nivel por debajo del nombre indicado del función, entonces la función está haciendo una cosa. Después de todo, la razón por la que escribimos funciones es para descomponer un concepto más grande (en otras palabras, el nombre de la función) en un conjunto de pasos en el siguiente nivel de abstracción.

Debe quedar muy claro que el Listado 3-1 contiene pasos en muchos niveles diferentes de abstracción. Por lo que claramente está haciendo más de una cosa. Incluso el Listado 3-2 tiene dos niveles de abstracción, como lo demuestra nuestra capacidad para reducirlo. Pero sería muy difícil de decir ... Reducir completamente el Listado 3-3. Podríamos extraer la declaración `if` en una función llamada `includeSetupsAndTeardownsIfTestPage` , pero eso simplemente reafirma el código sin cambiar el nivel de abstracción.

Por lo tanto, otra forma de saber que una función está haciendo más de "una cosa" es si puede extraer otra función de él con un nombre que no sea simplemente una reafirmación de su implementación mentación [G34].

Secciones dentro de funciones

Mire el Listado 4-7 en la página 71. Observe que la función `generatePrimes` se divide en secciones como *declaraciones* , *inicializaciones* y *tamiz* . Este es un síntoma obvio de haciendo más de una cosa. Las funciones que hacen una cosa no pueden dividirse razonablemente en secciones.

Un nivel de abstracción por función

Para asegurarnos de que nuestras funciones están haciendo "una cosa", debemos asegurarnos de que el las declaraciones dentro de nuestra función están todas en el mismo nivel de abstracción. Es fcil ver cmo El listado 3-1 viola esta regla. Hay conceptos ahí que están en un nivel muy alto de

abstracción, como getHtml () ; otros que se encuentran en un nivel intermedio de abstracción, como como: String pagePathName = PathParser.render (pagePath) ; y otros que se destacan nivel hábilmente bajo, como: .append ("\ n") .

Mezclar niveles de abstracción dentro de una función siempre resulta confuso. Los lectores no pueden Ser capaz de decir si una expresión en particular es un concepto esencial o un detalle. Peor,

4. El lenguaje LOGO usaba la palabra clave "TO" de la misma manera que Ruby y Python usan "def". Entonces cada función comenzaba con la palabra "TO". Esto tuvo un efecto interesante en la forma en que se diseñaron las funciones.

Página 68

como ventanas rotas, una vez que los detalles se mezclan con conceptos esenciales, cada vez más los detalles tienden a acumularse dentro de la función.

Lectura de código de arriba a abajo: *la regla de reducción*

Queremos que el código se lea como una narrativa de arriba hacia abajo. [5] Queremos que se sigan todas las funciones los que están en el siguiente nivel de abstracción para que podamos leer el programa, descendiendo un nivel de abstracción a la vez a medida que leemos la lista de funciones. Yo llamo a esto *El Paso-Regla abajo* .

Para decir esto de otra manera, queremos poder leer el programa como si fuera un conjunto de los párrafos *TO* , cada uno de los cuales describe el nivel actual de abstracción y referencia ferencias posteriores *A* los párrafos en el siguiente nivel hacia abajo.

> *Para incluir las configuraciones y desmontajes, incluimos configuraciones, luego incluimos la página de prueba con carpa, y luego incluimos los desmontajes.*
>
> *Para incluir las configuraciones, incluimos la configuración de la suite si se trata de una suite, luego incluimos la configuración regular.*
>
> *Para incluir la configuración de la suite, buscamos en la jerarquía principal la página "SuiteSetUp" y agregue una declaración de inclusión con la ruta de esa página.*
>
> *Para buscar al padre. . .*

Resulta muy difícil para los programadores aprender a seguir esta regla y escribir funciones que permanecen en un solo nivel de abstracción. Pero aprender este truco también es muy importante. Es la clave para mantener las funciones breves y asegurarse de que hagan "una cosa". Hacer que el código se lea como un conjunto de párrafos *TO* de arriba hacia abajo es una técnica eficaz para manteniendo el nivel de abstracción consistente.

Eche un vistazo al Listado 3-7 al final de este capítulo. Muestra el todo Función testableHtml refactorizada de acuerdo con los principios descritos aquí. darse cuenta cómo cada función introduce la siguiente, y cada función permanece en un nivel consistente de abstracción.

Cambiar declaraciones

Es difícil hacer una pequeña declaración de cambio . [6] Incluso una declaración de cambio con solo dos casos es más grande de lo que me gustaría que fuera un solo bloque o función. También es difícil hacer un cambio de estado. ment que hace una cosa. Por su naturaleza, las declaraciones de cambio siempre hacen *N* cosas. Desafortunadamente damente no siempre podemos evitar el interruptor de declaraciones, pero *podemos* asegurar que cada interruptor La declaración está enterrada en una clase de bajo nivel y nunca se repite. Hacemos esto, por supuesto, con polimorfismo.

5. [KP78], pág. 37.
6. Y, por supuesto, incluyo cadenas if / else en esto.

Página 69

Considere el listado 3-4. Muestra solo una de las operaciones que pueden depender de la tipo de empleado.

Listado 3-4
Payroll.java

```
public Money calculatePay (Empleado e)
lanza InvalidEmployeeType {
        switch (e.type) {
            caso COMISIONADO:
                return calculateCommissionedPay (e);
            caso HORARIO:
                return calculateHourlyPay (e);
            caso SALARIADO:
                return calculateSalariedPay (e);
            defecto:
                lanzar nuevo InvalidEmployeeType (e.type);
        }
    }
```

Hay varios problemas con esta función. Primero, es grande y cuando es nuevo se agregan tipos de empleados, crecerá. En segundo lugar, claramente hace más de una cosa. En tercer lugar, viola el Principio [7] de Responsabilidad Única (SRP) porque hay más de una razón para que cambie. Cuarto, viola el Principio [8] Abierto Cerrado (OCP) porque debe cambiar cada vez que se agregan nuevos tipos. Pero posiblemente el peor problema con esto función es que hay un número ilimitado de otras funciones que tendrán la misma estructura. Por ejemplo, podríamos tener

```
isPayday (Empleado e, Fecha fecha),
```

o

```
deliverPay (Empleado e, Pago de dinero),
```

o una multitud de otros. Todo lo cual tendría la misma estructura deletérea.

La solución a este problema (vea el Listado 3-5) es enterrar la instrucción switch en el Un sótano de un ESUMEN F actory , [9] y nunca deje que nadie lo vea. La fábrica utilizará el instrucción switch para crear instancias apropiadas de las derivadas de Employee , y la variable Las funciones importantes, como calculatePay , isPayday y deliverPay , se enviarán morphically a través de la interfaz de empleado .

Mi regla general para las declaraciones de cambio es que pueden tolerarse si aparecen solo una vez, se utilizan para crear objetos polimórficos y se ocultan detrás de una herencia

7. a. http://en.wikipedia.org/wiki/Single_responsibility_principl e
   B. http://www.objectmentor.com/resources/articles/srp.pdf
8. a. http://en.wikipedia.org/wiki/Open/closed_principle
   B. http://www.objectmentor.com/resources/articles/ocp.pdf
9. [GOF].

Página 70

Listado 3-5

Empleado y fábrica

```
Empleado público de la clase abstracta {
    public abstract boolean isPayday ();
    Resumen público Money calculatePay ();
    public abstract void deliverPay (Pago de dinero);
}
----------
interfaz pública EmployeeFactory {
    El empleado público makeEmployee (EmployeeRecord r) arroja InvalidEmployeeType;
}
----------
La clase pública EmployeeFactoryImpl implementa EmployeeFactory {
    El empleado público makeEmployee (EmployeeRecord r) arroja InvalidEmployeeType {
        switch (r.type) {
            caso COMISIONADO:
                devolver nuevo CommissionedEmployee (r);
            caso HORARIO:
                return new HourlyEmployee (r);
            caso SALARIADO:
                return new SalariedEmploye (r);
            defecto:
                lanzar nuevo InvalidEmployeeType (r.type);
        }
    }
}
```

relación para que el resto del sistema no pueda verlos [G23]. Por supuesto, en todas las circunstancias La postura es única, y hay ocasiones en las que violé una o más partes de esa regla.

Utilice nombres descriptivos

En el Listado 3-7 cambié el nombre de nuestra función de ejemplo de testableHtml a SetupTeardownIncluder.render . Este es un nombre mucho mejor porque describe mejor lo que la función lo hace. También le di a cada uno de los métodos privados un nombre igualmente descriptivo como isTestable o includeSetupAndTeardownPages . Es difícil sobreestimar el valor de buenos nombres. Recuerde el principio de Ward: " *Sabes que estás trabajando en un código limpio cuando cada rutina resulta ser más o menos lo que esperabas.* "La mitad de la batalla para lograr ese principio es elegir buenos nombres para funciones pequeñas que hacen una cosa. Cuanto más pequeña y centrada sea una función, más fácil será elegir una función descriptiva. nombre.

No tenga miedo de hacerse un nombre largo. Un nombre descriptivo largo es mejor que uno corto. nombre enigmático. Un nombre descriptivo largo es mejor que un comentario descriptivo largo. Usar una convención de nomenclatura que permite que se lean fácilmente varias palabras en los nombres de las funciones, y luego hacer uso de esas múltiples palabras para darle a la función un nombre que diga lo que lo hace.

No tenga miedo de perder tiempo eligiendo un nombre. De hecho, deberías probar varias ent nombres y lea el código con cada uno en su lugar. Los IDE modernos como Eclipse o IntelliJ hacen es trivial cambiar de nombre. Utilice uno de esos IDE y experimente con diferentes nombres hasta que encuentre uno que sea lo más descriptivo posible.

La elección de nombres descriptivos aclarará el diseño del módulo en su mente y

ayudarlo a mejorarlo. No es nada infrecuente que la búsqueda de un buen nombre resulte en una reestructuración favorable del código.

Sea coherente con sus nombres. Use las mismas frases, sustantivos y verbos en la función nombres que elija para sus módulos. Considere, por ejemplo, los nombres incluyenSetup-AndTeardownPages , includeSetupPages , includeSuiteSetupPage e includeSetupPage . La una fraseología similar en esos nombres permite que la secuencia cuente una historia. De hecho, si yo te mostraba solo la secuencia anterior, te preguntarías: "¿Qué pasó con includeTeardownPages , includeSuiteTeardownPage e includeTeardownPage ? " Cómo es eso por ser “. . . *más o menos lo que esperabas* ".

Argumentos de función

El número ideal de argumentos para una función es cero (niladico). Luego viene uno (monádico), seguido de cerca por dos (diádico). Tres argumentos (triádica) debe evitarse siempre que sea posible. Mas de tres (poliádico) requiere una justificación muy especial, y entonces no debería usarse de todos modos.

Los argumentos son difíciles. Requieren mucho control. poder ceptual. Por eso me deshice de casi todos ellos del ejemplo. Considere, por ejemplo, el StringBuffer en el ejemplo. Nosotros podríamos tener lo pasó como un argumento en lugar de hacer como una variable de instancia, pero luego nuestros lectores habría tenido que interpretarlo cada vez que vieran eso. Cuando está leyendo la historia contada por el módulo, includeSetupPage () es más fácil de entender que includeSetupPageInto (newPage-Contenido) . El argumento está en un nivel diferente de abstracción que el nombre de la función y te obliga a conocer un detalle (en otras palabras, StringBuffer) que no es particularmente importante en ese punto.

Los argumentos son aún más difíciles desde el punto de vista de las pruebas. Imagina la dificultad de escribir todos los casos de prueba para garantizar que todas las combinaciones de argumentos funcionen adecuadamente. Si no hay argumentos, esto es trivial. Si hay un argumento, no es demasiado difícil. Con dos argumentos, el problema se vuelve un poco más desafiante. Con más de dos argumentos En estos casos, probar cada combinación de valores apropiados puede resultar abrumador.

Página 72

Los argumentos de salida son más difíciles de entender que los argumentos de entrada. Cuando leemos un función, estamos acostumbrados a la idea de la información que va *en* la función a través de argumentos y *a* través del valor de retorno. No solemos esperar que salga información a través de los argumentos. Por lo tanto, los argumentos de salida a menudo nos hacen pensar dos veces.

Un argumento de entrada es la mejor alternativa a ningún argumento. Configuración Desmontaje Includer.render (pageData) es bastante fácil de entender. Claramente vamos a *renderizar* el datos en el objeto pageData .

Formas monádicas comunes

Hay dos razones muy comunes para pasar un solo argumento a una función. Usted puede ser hacer una pregunta sobre ese argumento, como en boolean fileExists ("MyFile") . O puedes ser operando sobre ese argumento, transformándolo en otra cosa y *devolviéndolo* . Para ejemplo, InputStream fileOpen ("MyFile") transforma un nombre de archivo String en un Valor de retorno InputStream . Estos dos usos son los que esperan los lectores cuando ven una función. ción. Debe elegir nombres que hagan clara la distinción y siempre use los dos

formas en un contexto coherente. (Consulte Separación de consultas de comandos a continuación).

Una forma algo menos común, pero aún muy útil para una función de argumento único, es un *evento* . En esta forma hay un argumento de entrada pero no un argumento de salida. El general El programa está destinado a interpretar la llamada a la función como un evento y usar el argumento para alterar el estado del sistema, por ejemplo, `void passwordAttemptFailedNtimes (int intentos)` . Usar este formulario con cuidado. Debe quedar muy claro para el lector que se trata de un evento. Escoger nombres y contextos cuidadosamente.

Trate de evitar cualquier función monádica que no siga estas formas, por ejemplo, `void includeSetupPageInto (StringBuffer pageText)` . Usando un argumento de salida en lugar de un El valor de retorno de una transformación es confuso. Si una función va a transformar su entrada argumento, la transformación debería aparecer como el valor de retorno. De hecho, `StringBuffer transform (StringBuffer in)` es mejor que `void transform- (StringBuffer out)` , incluso si el La implementación en el primer caso simplemente devuelve el argumento de entrada. Al menos todavía sigue la forma de una transformación.

Argumentos de la bandera

Los argumentos de la bandera son feos. Pasar un valor booleano a una función es una práctica realmente terrible. Eso inmediatamente complica la firma del método, proclamando en voz alta que esta función hace más de una cosa. ¡Hace una cosa si la bandera es verdadera y otra si la bandera es falsa!

En el Listado 3-7 no teníamos otra opción porque las personas que llamaban ya estaban pasando esa bandera in, y quería limitar el alcance de la refactorización a la función y más abajo. Aún así, el la llamada al método `render (true)` es simplemente confusa para un lector deficiente. Pasando el mouse sobre la llamada y ver `render (boolean isSuite)` ayuda un poco, pero no tanto. Nosotros deberíamos tener divide la función en dos: `renderForSuite ()` y `renderForSingleTest ()` .

Página 73

Funciones diádicas

Una función con dos argumentos es más difícil de entender que una función monádica. Para examen- Por ejemplo, `writeField (nombre)` es más fácil de entender que `writeField (flujo de salida, nombre)` . [10] Aunque el significado de ambos es claro, el primero se desliza más allá del ojo, depositando fácilmente su significado. El segundo requiere una breve pausa hasta que aprendamos a ignorar el primer parámetro. Y *eso* , por supuesto, eventualmente resulta en problemas porque nunca debemos ignorar parte del código. Las partes que ignoramos son donde se esconden los errores.

Hay momentos, por supuesto, en los que dos argumentos son apropiados. Por ejemplo, `Punto p = nuevo Punto (0,0);` es perfectamente razonable. Los puntos cartesianos naturalmente toman dos argumentos. De hecho, nos sorprendería mucho ver el `nuevo Punto (0)` . Sin embargo, los dos argumentan En este caso , *los componentes son componentes ordenados de un solo valor.* Mientras que `Output-Stream` y `Los nombres` no tienen cohesión natural ni ordenamiento natural.

Incluso las funciones diádicas obvias como `assertEquals (esperado, real)` son problemáticas. ¿Cuántas veces ha puesto lo `real` donde debería estar lo `esperado` ? Los dos discuten los mentos no tienen un orden natural. El pedido `real esperado` es una convención que requiere práctica para aprender.

Las diadas no son malvadas, y seguramente tendrás que escribirlas. Sin embargo, deberías estar conscientes de que tienen un costo y deben aprovechar lo que los mecánicos pueden ser disponible para convertirlas en mónadas. Por ejemplo, puede hacer que el método `writeField` un miembro de `outputStream` para que pueda decir `outputStream. writeField (nombre)` . O puede hacer que `outputStream sea` una variable miembro del actual clase para que no tengas que pasarla. O puede extraer una nueva clase como `FieldWriter` que toma `outputStream` en su constructor y tiene un método de `escritura` .

Tríadas

Las funciones que toman tres argumentos son significativamente más difíciles de entender que las díadas. La

los problemas de ordenar, pausar e ignorar son más del doble. Te sugiero que pienses muy cuidadosamente antes de crear una tríada.

Por ejemplo, considere la sobrecarga común de assertEquals que toma tres argumentos ments: assertEquals (mensaje, esperado, actual) . ¿Cuántas veces has leído el mensaje y pensó que era lo esperado ? Me he tropezado y me he detenido en ese particular tríada muchas veces. De hecho, *cada vez que lo veo, lo* miro dos veces y luego aprendo a ignorar la mensaje.

Por otro lado, aquí hay una tríada que no es tan insidiosa: assertEquals (1.0, cantidad, .001) . Aunque esto todavía requiere una doble toma, vale la pena tomarla. Su Siempre es bueno recordar que la igualdad de los valores de coma flotante es algo relativo.

10. Acabo de terminar de refactorizar un módulo que usaba la forma diádica. Pude hacer que outputStream fuera un campo de la clase y convierte todas las llamadas a writeField a la forma monádica. El resultado fue mucho más limpio.

Página 74

Objetos de argumento

Cuando una función parece necesitar más de dos o tres argumentos, es probable que algunos de esos argumentos deberían incluirse en una clase propia. Considere, por ejemplo, el diferencia entre las dos declaraciones siguientes:

```
Círculo makeCircle (doble x, doble y, doble radio);
Círculo makeCircle (centro del punto, radio doble);
```

Reducir el número de argumentos creando objetos a partir de ellos puede parecer trampa, pero no lo es. Cuando los grupos de variables se pasan juntos, la forma en que x y Si están en el ejemplo anterior, es probable que formen parte de un concepto que merece un nombre de su propio.

Listas de argumentos

A veces queremos pasar un número variable de argumentos a una función. Considerar para ejemplo, el método String.format :

```
String.format ("% s trabajó% .2f horas.", Nombre, horas);
```

Si todos los argumentos de las variables se tratan de forma idéntica, como en el ejemplo anterior, entonces son equivalentes a un solo argumento de tipo List . Por ese razonamiento, String.format es en realidad diádico. De hecho, la declaración de String.format como se muestra a continuación es claramente diádico.

```
formato de cadena pública (formato de cadena, objeto ... argumentos)
```

Entonces se aplican todas las mismas reglas. Las funciones que toman argumentos variables pueden ser mónadas, díadas, o incluso tríadas. Pero sería un error darles más argumentos que que.

```
mónada vacía (Entero ... argumentos);
díada vacía (nombre de cadena, entero ... argumentos);
tríada vacía (nombre de cadena, recuento int, entero ... args);
```

Verbos y palabras clave

Elegir buenos nombres para una función puede ayudar mucho a explicar la intención de la función y el orden y la intención de los argumentos. En el caso de una mónada, el la función y el argumento deben formar un par de verbo / sustantivo muy agradable. Por ejemplo, escribir (nombre) es muy evocador. Sea lo que sea este "nombre", está siendo "escrito". Un un nombre aún mejor podría ser writeField (nombre) , que nos dice que el "nombre" es un "campo."

Este último es un ejemplo de la forma de *palabra clave* de un nombre de función. Usando este formulario codifica los nombres de los argumentos en el nombre de la función. Por ejemplo, assertEquals

podría escribirse mejor como assertExpectedEqualsActual (esperado, real) . Esto fuertemente mitiga el problema de tener que recordar el orden de los argumentos.

Página 75

No tiene efectos secundarios

Los efectos secundarios son mentiras. Su función promete hacer una cosa, pero también hace otras *ocultas.* cosas. A veces, hará cambios inesperados en las variables de su propia clase. A veces los convertirá en los parámetros pasados a la función o al sistema global bals. En cualquier caso, son falsedades tortuosas y dañinas que a menudo resultan en extrañas acoplamientos temporales y dependencias de orden.

Considere, por ejemplo, la función aparentemente inocua del Listado 3-6. Esta función utiliza un algoritmo estándar para que coincida con un nomUsuario a una contraseña . Devuelve verdadero si coinciden y falso si algo sale mal. Pero también tiene efectos secundarios. ¿Puedes distinguirlo?

Listado 3-6

UserValidator.java

```
UserValidator de clase pública {
  criptógrafo criptógrafo privado;

  public boolean checkPassword (String userName, String contraseña) {
    Usuario usuario = UserGateway.findByName (nombre de usuario);
    if (usuario! = Usuario.NULL) {
      String codedPhrase = user.getPhraseEncodedByPassword ();
      Frase de cadena = cryptographer.decrypt (codedPhrase, contraseña);
      if ("Contraseña válida" .equals (frase)) {
        Session.initialize ();
        devuelve verdadero;
      }
    }
    falso retorno;
  }
}
```

El efecto secundario es la llamada a Session.initialize () , por supuesto. La función checkPassword ción, por su nombre, dice que comprueba la contraseña. El nombre no implica que sus iniciales Iiza la sesión. Entonces, una persona que llama y cree lo que dice el nombre de la función corre el riesgo de borrar los datos de la sesión existente cuando decida comprobar la validez de la usuario.

Este efecto secundario crea un acoplamiento temporal. Es decir, checkPassword solo puede ser llamado en ciertos momentos (en otras palabras, cuando es seguro inicializar la sesión). Si esto es llamado fuera de servicio, los datos de la sesión pueden perderse inadvertidamente. Los acoplamientos temporales se fusión, especialmente cuando se oculta como efecto secundario. Si debe tener un acoplamiento temporal, debe dejarlo claro en el nombre de la función. En este caso, podríamos cambiar el nombre del función checkPasswordAndInitializeSession , aunque eso ciertamente viola "Do one cosa."

Página 76

Argumentos de salida

Los argumentos se interpretan más naturalmente como *entradas* a una función. Si ha sido gramática durante más de unos pocos años, estoy seguro de que ha hecho una doble toma de un argumento eso fue en realidad una *salida* más que una entrada. Por ejemplo:

```
appendFooter (s);
```

¿Esta función agrega s como pie de página a algo? ¿O agrega algún pie de página? a s ? ¿Es s una entrada o una salida? No lleva mucho tiempo mirar la firma de la función y ver:

```
public void appendFooter (informe StringBuffer)
```

Esto aclara el problema, pero solo a expensas de verificar la declaración de la función. Cualquier cosa que le obligue a comprobar la firma de la función equivale a una doble toma. Es un descanso cognitivo y debe evitarse.

En los días anteriores a la programación orientada a objetos, a veces era necesario tener argumentos de salida. Sin embargo, gran parte de la necesidad de argumentos de salida desaparece en el lenguaje OO. Guages porque esta es *la intención* de actuar como un argumento de salida. En otras palabras, sería mejor para que appendFooter sea invocado como

```
report.appendFooter ();
```

En general, deben evitarse los argumentos de salida. Si su función debe cambiar el estado de algo, haga que cambie el estado de su objeto propietario.

Separación de consultas de comandos

Las funciones deben hacer algo o responder algo, pero no ambos. O tu La función debería cambiar el estado de un objeto, o debería devolver alguna información sobre ese objeto. Hacer ambas cosas a menudo conduce a la confusión. Considere, por ejemplo, lo siguiente función:

```
conjunto booleano público (atributo de cadena, valor de cadena);
```

Esta función establece el valor de un atributo con nombre y devuelve verdadero si tiene éxito y falso si no existe tal atributo. Esto conduce a declaraciones extrañas como esta:

```
if (set ("nombre de usuario", "unclebob")) ...
```

Imagínese esto desde el punto de vista del lector. ¿Qué significa? ¿Se está preguntando si el atributo " nombre de usuario " se estableció previamente en " unclebob "? ¿O se pregunta si el ¿El atributo " username " se estableció correctamente en " unclebob "? Es difícil inferir el significado de la llamada porque no está claro si la palabra " set " es un verbo o un adjetivo.

El autor pretendía que el conjunto fuera un verbo, pero en el contexto de la declaración if se *siente* como un adjetivo. Por lo tanto, la declaración se lee como "Si el atributo de nombre de usuario se estableció previamente en unclebob ”y no“ establezca el atributo de nombre de usuario en unclebob y si eso funcionó, entonces. . . . " Nosotros

Página 77

podría intentar resolver esto cambiando el nombre de la función set a setAndCheckIfExists , pero eso no ayuda mucho a la legibilidad de la declaración if . La verdadera solución es separar el comando de la consulta para que no se produzca la ambigüedad.

```
if (attributeExists ("nombre de usuario")) {
    setAttribute ("nombre de usuario", "unclebob");
    ...
}
```

Prefiere las excepciones a la devolución de códigos de error

Devolver códigos de error de funciones de comando es una violación sutil de la consulta de comando separación. Promueve el uso de comandos como expresiones en los predicados de if state- mentos.

```
if (deletePage (página) == E_OK)
```

Esto no sufre de confusión verbo / adjetivo, pero conduce a estructuras profundamente anidadas. tures. Cuando devuelve un código de error, crea el problema con el que debe lidiar la persona que llama el error inmediatamente.

```
if (deletePage (página) == E_OK) {
    if (registry.deleteReference (page.name) == E_OK) {
        if (configKeys.deleteKey (page.name.makeKey ()) == E_OK) {
            logger.log ("página eliminada");
        } demás {
            logger.log ("configKey no eliminada");
        }
    } demás {
        logger.log ("DeleteReference del registro falló");
    }
} demás {
    logger.log ("error de eliminación");
    return E_ERROR;
}
```

Por otro lado, si usa excepciones en lugar de códigos de error devueltos, el error El código de procesamiento se puede separar del código de ruta feliz y se puede simplificar:

```
intentar {
    deletePage (página);
    registration.deleteReference (page.name);
    configKeys.deleteKey (page.name.makeKey ());
}
captura (Excepción e) {
    logger.log (e.getMessage ());
}
```

Extraer bloques de prueba / captura

Los bloques Try / catch son feos por derecho propio. Confunden la estructura del código y mezclar el procesamiento de errores con el procesamiento normal. Entonces es mejor extraer los cuerpos del intento. y captura bloques en funciones propias.

Página 78


```
public void delete (página de la página) {
    intentar {
        deletePageAndAllReferences (página);
    }
    captura (Excepción e) {
        logError (e);
    }
}

private void deletePageAndAllReferences (página de página) arroja Exception {
    deletePage (página);
    registration.deleteReference (page.name);
    configKeys.deleteKey (page.name.makeKey ());
}
```

```
    private void logError (Excepción e) {
        logger.log (e.getMessage ());
    }
```

En lo anterior, la función de eliminación tiene que ver con el procesamiento de errores. Es facil de entender y luego ignorar. La función deletePageAndAllReferences tiene que ver con los procesos de eliminar completamente una página . El manejo de errores se puede ignorar. Esto proporciona una buena separación que hace que el código sea más fácil de entender y modificar.

El manejo de errores es una cosa

Las funciones deberían hacer una cosa. La entrega de errores es una cosa. Por lo tanto, una función que maneja los errores no deberían hacer nada más. Esto implica (como en el ejemplo anterior) que si la palabra clave try existe en una función, debería ser la primera palabra en la función y que no no debería haber nada después de los bloques catch / finalmente .

El imán de dependencia Error.java

Devolver códigos de error generalmente implica que hay alguna clase o enumeración en la que todos los Se definen los códigos de error.

```
error de enumeración pública {
    OK,
    INVÁLIDO,
    NO TAL,
    BLOQUEADO,
    OUT_OF_RESOURCES,
    WAITING_FOR_EVENT;
}
```

Clases como esta son un *imán de dependencia;* muchas otras clases deben importar y usar ellos. Por lo tanto, cuando cambia la enumeración Error , todas esas otras clases deben volver a compilarse y redistribuido. [11] Esto ejerce una presión negativa sobre la clase Error . Los programadores no quieren

11. Aquellos que sintieron que podían escapar sin recompilarse y volver a desplegarse, fueron encontrados y tratados.

Página 79

para agregar nuevos errores porque luego tienen que reconstruir y volver a implementar todo. Entonces ellos reutilizan códigos de error antiguos en lugar de agregar nuevos.

Cuando usa excepciones en lugar de códigos de error, las nuevas excepciones son *derivadas* de la clase de excepción. Se pueden agregar sin forzar ninguna recompilación o redespliegue. [12]

No se repita [13]

Vuelva a mirar el Listado 3-1 con cuidado y notará que hay un algoritmo que se repite cuatro veces, una para cada uno de el SetUp , SuiteSetUp , TearDown , y Estuches SuiteTearDown . No es fácil de detectar esta duplicación porque las cuatro instancias están mezclados con otro código y no uniformemente duplicado. Aún así, la duplicación es un problema porque hincha el código y requerirá una modificación cuádruple en caso de que el algoritmo tenga que cambiar. También es un oportunidad cuádruple para un error de omisión.

Esta duplicación se solucionó con el método de inclusión en el Listado 3-7. Leer de parte a parte ese código de nuevo y observe cómo la legibilidad de todo el módulo se ve mejorada por la reducción de esa duplicación.

La duplicación puede ser la raíz de todos los males del software. Muchos principios y prácticas se han

sido creado con el propósito de controlarlo o eliminarlo. Considere, por ejemplo, que
todas las formas normales de la base de datos de Codd sirven para eliminar la duplicación de datos. Considere también
cómo la programación orientada a objetos sirve para concentrar el código en clases base que
de lo contrario, sería redundante. Programación estructurada, Programación orientada a aspectos, Compo-
La programación orientada nent, son todas, en parte, estrategias para eliminar la duplicación. Eso
Parecería que desde la invención de la subrutina, las innovaciones en el desarrollo de software
ha sido un intento continuo de eliminar la duplicación de nuestro código fuente.

Programación estructurada

Algunos programadores siguen las reglas de programación estructurada de Edsger Dijkstra. [14] Dijkstra
dijo que cada función, y cada bloque dentro de una función, debe tener una entrada y una
Salida. Seguir estas reglas significa que solo debe haber una declaración de retorno en una función.
ción, sin ruptura o continuar declaraciones en un bucle, y nunca, *nunca,* ningún Goto declaraciones.

12. Este es un ejemplo del principio abierto cerrado (OCP) [PPP02].
13. El principio DRY. [PRAG].
14. [SP72].

Si bien simpatizamos con los objetivos y disciplinas de la programación estructurada,
esas reglas sirven de poco beneficio cuando las funciones son muy pequeñas. Es solo en funciones más grandes
que tales reglas brindan un beneficio significativo.

Entonces, si mantiene sus funciones pequeñas, entonces el retorno múltiple ocasional , interrupción o
Continuar la declaración no hace daño y, a veces, incluso puede ser más expresiva que el pecado.
regla de entrada única y salida única. Por otro lado, goto solo tiene sentido en funciones grandes, por lo que
debe evitarse.

¿Cómo se escriben funciones como esta?

El software de escritura es como cualquier otro tipo de escritura. Cuando escribes un artículo o un artículo,
primero anota tus pensamientos y luego lo masajeas hasta que se lea bien. El primer borrador
puede ser torpe y desorganizado, por lo que lo redacta, lo reestructura y lo refina hasta
se lee de la forma que usted desea que se lea.

Cuando escribo funciones, resultan largas y complicadas. Tienen muchos
sangría y bucles anidados. Tienen largas listas de argumentos. Los nombres son arbitrarios y
hay código duplicado. Pero también tengo un conjunto de pruebas unitarias que cubren cada una de esas
torpes líneas de código.

Entonces masajeo y refino ese código, dividiendo funciones, cambiando nombres, eliminando
inating duplicación. Reduzco los métodos y los reordeno. A veces me rompo todo
clases, todo el tiempo manteniendo las pruebas pasando.

Al final, termino con funciones que siguen las reglas que establecí en este capítulo.
No los escribo de esa manera para empezar. No creo que nadie pudiera.

Conclusión

Cada sistema está construido a partir de un lenguaje específico de dominio diseñado por los programadores para
describir ese sistema. Las funciones son los verbos de ese idioma y las clases son los sustantivos.
Esto no es un retroceso a la vieja y espantosa noción de que los sustantivos y verbos en un requisito
Los documentos son la primera suposición de las clases y funciones de un sistema. Más bien, esto es
una verdad mucho más antigua. El arte de programar es, y siempre ha sido, el arte del lenguaje.
diseño.

Los programadores expertos piensan en los sistemas como historias para contar en lugar de programas para

ser escrito. Usan las facilidades de su lenguaje de programación elegido para construir un lenguaje mucho más rico y expresivo que se puede utilizar para contar esa historia. Parte de eso El lenguaje específico del dominio es la jerarquía de funciones que describen todas las acciones que tener lugar dentro de ese sistema. En un acto ingenioso de recursividad, esas acciones se escriben en utilizar el lenguaje específico del dominio que definen para contar su propia pequeña parte de la historia.

Este capítulo ha tratado bien la mecánica de escribir funciones. Si tu sigues según las reglas de este documento, sus funciones serán breves, estarán bien nombradas y bien organizadas. Pero

Página 81

Nunca olvides que tu objetivo real es contar la historia del sistema y que las funciones que escribas deben encajar perfectamente en un lenguaje claro y preciso para ayudarte con ese relato.

SetupTeardownIncluder

Listado 3-7

SetupTeardownIncluder.java

```
package fitnesse.html;

import fitnesse.responders.run.SuiteResponder;
importar fitnesse.wiki. *;

SetupTeardownIncluder de clase pública {
   PageData privado pageData;
   isSuite booleano privado;
   página de prueba de WikiPage privada;
   private StringBuffer newPageContent;
   PageCrawler privado pageCrawler;


   El procesamiento de cadenas estáticas públicas (PageData pageData) arroja una excepción {
      return render (pageData, falso);
   }

   Representación pública de cadenas estáticas (PageData pageData, boolean isSuite)
      lanza Exception {
      devolver nuevo SetupTeardownIncluder (pageData) .render (isSuite);
   }

   SetupTeardownIncluder privado (PageData pageData) {
      this.pageData = pageData;
      testPage = pageData.getWikiPage ();
      pageCrawler = testPage.getPageCrawler ();
      newPageContent = new StringBuffer ();
   }

   Private String render (boolean isSuite) arroja Exception {
      this.isSuite = isSuite;
      si (isTestPage ())
         includeSetupAndTeardownPages ();
      return pageData.getHtml ();
   }

   private boolean isTestPage () lanza Exception {
      return pageData.hasAttribute ("Prueba");
   }

   private void includeSetupAndTeardownPages () lanza Exception {
      includeSetupPages ();
      includePageContent ();
      includeTeardownPages ();
      updatePageContent ();
   }
```

Página 82

Listado 3-7 (continuación)

SetupTeardownIncluder.java

```
private void includeSetupPages () lanza Exception {
    si (isSuite)
        includeSuiteSetupPage ();
    includeSetupPage ();
}

private void includeSuiteSetupPage () lanza Exception {
    incluir (SuiteResponder.SUITE_SETUP_NAME, "-setup");
}

private void includeSetupPage () lanza Exception {
    incluir ("Configuración", "-configuración");
}

private void includePageContent () lanza Exception {
    newPageContent.append (pageData.getContent ());
}

private void includeTeardownPages () lanza Exception {
    includeTeardownPage ();
    si (isSuite)
        includeSuiteTeardownPage ();
}

private void includeTeardownPage () lanza Exception {
    incluir ("TearDown", "-teardown");
}

private void includeSuiteTeardownPage () lanza Exception {
    incluir (SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");
}

private void updatePageContent () lanza Exception {
    pageData.setContent (newPageContent.toString ());
}

private void include (String pageName, String arg) lanza Exception {
    WikiPage hereitedPage = findInheritedPage (nombre de la página);
    if (hereitedPage! = null) {
        String pagePathName = getPathNameForPage (hereitedPage);
        buildIncludeDirective (pagePathName, arg);
    }
}

Private WikiPage findInheritedPage (String pageName) arroja Exception {
    return PageCrawlerImpl.getInheritedPage (pageName, testPage);
}

private String getPathNameForPage (página WikiPage) arroja Exception {
    WikiPagePath pagePath = pageCrawler.getFullPath (página);
    return PathParser.render (pagePath);
}

private void buildIncludeDirective (String pagePathName, String arg) {
    newPageContent
        .append ("\ n! incluir")
```


Página 83

Listado 3-7 (continuación)

SetupTeardownIncluder.java

```
            .append (arg)
            .adjuntar(" .")
            .append (pagePathName)
            .append ("\ n");
    }
}
```

Bibliografía

[KP78]: Kernighan y Plaugher, *Los elementos del estilo de programación* , 2d. ed., McGraw-Hill, 1978.

[PPP02]: Robert C. Martin, *Desarrollo de software ágil: principios, patrones y prácticas tices* , Prentice Hall, 2002.

[GOF]: *Patrones de diseño: elementos de software orientado a objetos reutilizables* , Gamma et al., Addison-Wesley, 1996.

[PRAG]: *El programador pragmático* , Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

[SP72]: *Programación estructurada* , O.-J. Dahl, EW Dijkstra, CAR Hoare, académico Prensa, Londres, 1972.

4
Comentarios

"No comente el código incorrecto, reescríbalo".

—Brian W. Kernighan y PJ Plaugher [1]

Nada puede ser tan útil como un comentario bien ubicado. Nada puede desordenar un mod-
ule más que frívolos comentarios dogmáticos. Nada puede ser tan dañino como un viejo
comentario grosero que propaga mentiras y desinformación.

Los comentarios no son como la Lista de Schindler. No son "pura bondad". De hecho, comentarios
son, en el mejor de los casos, un mal necesario. Si nuestros lenguajes de programación fueran lo suficientemente expresivos, o si

1. [KP78], pág. 144.

Página 85

teníamos el talento para manejar sutilmente esos lenguajes para expresar nuestra intención, no necesitaríamos
muchos comentarios, tal vez no en absoluto.

El uso adecuado de los comentarios es para compensar nuestra incapacidad para expresarnos en
código. Tenga en cuenta que usé la palabra *fracaso* . Lo dije en serio. Los comentarios siempre son fallos. Debemos
tenerlos porque no siempre podemos encontrar la manera de expresarnos sin ellos,
pero su uso no es motivo de celebración.

Entonces, cuando se encuentre en una posición en la que necesite escribir un comentario, piénselo
a través y ver si no hay alguna manera de cambiar las tornas y expresarse en
código. Cada vez que se exprese en código, debe darse una palmadita en la espalda. Cada
vez que escribe un comentario, debe hacer una mueca y sentir el fracaso de su capacidad de
expresión.

¿Por qué estoy tan deprimido con los comentarios? Porque mienten. No siempre, y no intencionalmente,
pero con demasiada frecuencia. Cuanto más antiguo es un comentario y más lejos está del código que describe,
lo más probable es que sea simplemente incorrecto. La razón es simple. Los programadores no pueden darse cuenta
mantenerlos ticamente.

El código cambia y evoluciona. Trozos se mueven de aquí para allá. Esos trozos se bifurcan
cate y se reproduzca y vuelva a unirse para formar quimeras. Desafortunadamente, la comunidad

los pensamientos no siempre los siguen , *no* siempre los *pueden* seguir. Y con demasiada frecuencia el los comentarios se separan del código que describen y se convierten en borrones huérfanos de disminución de la precisión. Por ejemplo, mire lo que ha sucedido con este comentario y la línea tenía la intención de describir:

```
Solicitud MockRequest;
Cadena final privada HTTP_DATE_REGEXP =
  "[SMTWF] [az] {2} \\, \\ s [0-9] {2} \\ s [JFMASOND] [az] {2} \\ s" +
  "[0-9] {4} \\ s [0-9] {2} \\: [0-9] {2} \\: [0-9] {2} \\ sGMT";
respuesta de respuesta privada;
contexto privado FitNesseContext;
respondedor FileResponder privado;
private Locale saveLocale;
// Ejemplo: "Martes, 02 de abril de 2003 22:18:49 GMT"
```

Otras variables de instancia que probablemente se agregaron más tarde se interpusieron entre los HTTP_DATE_REGEXP constante y su comentario explicativo.

Es posible señalar que los programadores deben ser lo suficientemente disciplinados para Mantenga los comentarios en un alto estado de conservación, relevancia y precisión. Estoy de acuerdo, deberían hacerlo. Pero preferiría que la energía se destine a hacer que el código sea tan claro y expresivo que no necesita los comentarios en primer lugar.

Los comentarios inexactos son mucho peores que ningún comentario. Engañan y engañan. Establecen expectativas que nunca se cumplirán. Establecen viejas reglas que no necesitan, o no debe seguirse más.

La verdad solo se puede encontrar en un lugar: el código. Solo el código puede realmente decirte lo que lo hace. Es la única fuente de información verdaderamente precisa. Por lo tanto, aunque los comentarios son a veces es necesario, gastaremos una gran cantidad de energía para minimizarlos.

Página 86

Los comentarios no compensan el código incorrecto

Una de las motivaciones más comunes para escribir comentarios es el código incorrecto. Escribimos un mod- ule y sabemos que es confuso y desorganizado. Sabemos que es un desastre. Entonces le decimos a nuestro- yo mismo, "¡Ooh, será mejor que comente eso!" ¡No! ¡Será mejor que lo limpie!

El código claro y expresivo con pocos comentarios es muy superior al desordenado y complejo código con muchos comentarios. En lugar de dedicar su tiempo a escribir los comentarios que Explica el desastre que has hecho, gástalo limpiando ese desastre.

Explíquese en el código

Ciertamente, hay ocasiones en las que el código es un vehículo deficiente para la explicación. Desafortunadamente, Muchos programadores han interpretado que esto significa que el código rara vez, si es que alguna vez, es un buen medio para explicación. Esto es evidentemente falso. ¿Qué preferirías ver? Esto:

```
// Verifique si el empleado es elegible para los beneficios completos
si ((employee.flags & HOURLY_FLAG) &&
      (empleado.Edad> 65))
```

¿O esto?

```
if (employee.isEligibleForFullBenefits ())
```

Solo se necesitan unos segundos de pensamiento para explicar la mayor parte de su intención en el código. En muchos casos, es simplemente una cuestión de crear una función que diga lo mismo que el comentario quieres escribir.

Buenos comentarios

Algunos comentarios son necesarios o beneficiosos. Veremos algunos que considero dignos de los bits que consumen. Tenga en cuenta, sin embargo, que el único comentario verdaderamente bueno es el

Comenta que encontraste una manera de no escribir.

Comentarios legales

A veces, nuestros estándares de codificación corporativos nos obligan a escribir ciertos comentarios para fines legales. razones. Por ejemplo, las declaraciones de derechos de autor y autoría son necesarias y razonables. cosas para poner en un comentario al comienzo de cada archivo fuente.

Aquí, por ejemplo, está el encabezado de comentario estándar que colocamos al principio de cada archivo fuente en FitNesse. Me complace decir que nuestro IDE oculta este comentario de la acción. como desorden al contraerlo automáticamente.

```
// Copyright (C) 2003,2004,2005 de Object Mentor, Inc. Todos los derechos reservados.
// Publicado bajo los términos de la GNU General Public License versión 2 o posterior.
```


Página 87

Comentarios como este no deben ser contratos o tomos legales. Siempre que sea posible, consulte una norma dard licencia u otro documento externo en lugar de poner todos los términos y condiciones en el comentario.

Comentarios informativos

A veces es útil proporcionar información básica con un comentario. Por ejemplo, con- Tenga en cuenta este comentario que explica el valor de retorno de un método abstracto:

```
// Devuelve una instancia del Respondedor que se está probando.
Respondedor abstracto protegido responderInstance ();
```

Un comentario como este a veces puede ser útil, pero es mejor usar el nombre de la función. ción para transmitir la información siempre que sea posible. Por ejemplo, en este caso el comentario podría volverse redundante cambiando el nombre de la función: `respondedorBeingTested` .

Aquí hay un caso que es un poco mejor:

```
// formato coincidente kk: mm: ss EEE, MMM dd, aaaa
Patrón timeMatcher = Pattern.compile (
    "\\ d *: \\ d *: \\ d * \\ w *, \\ w * \\ d *, \\ d *");
```

En este caso, el comentario nos permite saber que la expresión regular está destinada a coincidir con un hora y fecha que fueron formateadas con la función `SimpleDateFormat.format` usando la cadena de formato especificado. Aún así, podría haber sido mejor y más claro si este código hubiera sido se trasladó a una clase especial que convirtió los formatos de fechas y horas. Entonces el comentario probablemente habría sido superfluo.

Explicación de intención

A veces, un comentario va más allá de la mera información útil sobre la implementación y proporciona la intención detrás de una decisión. En el siguiente caso vemos una decisión interesante documentado por un comentario. Al comparar dos objetos, el autor decidió que quería clasificar los objetos de su clase por encima de los objetos de cualquier otro.

```
public int compareTo (Objeto o)
{
    si (o instancia de WikiPagePath)
    {
        WikiPagePath p = (WikiPagePath) o;
        String compressedName = StringUtil.join (nombres, "");
        String compressedArgumentName = StringUtil.join (p.names, "");
        return compressedName.compareTo (compressedArgumentName);
    }
    return 1; // somos más grandes porque somos del tipo correcto.
}
```

Aquí hay un ejemplo aún mejor. Es posible que no esté de acuerdo con la solución del programador para el problema, pero al menos sabes lo que estaba tratando de hacer.

```
public void testConcurrentAddWidgets () lanza Exception {
```

```
WidgetBuilder widgetBuilder =
    new WidgetBuilder (nueva Clase [] {BoldWidget.class});
```


Página 88


```
String text = "" 'texto en negrita' "";
ParentWidget parent =
    new BoldWidget (new MockWidgetRoot (), "" 'texto en negrita' "");
AtomicBoolean failFlag = new AtomicBoolean ();
failFlag.set (falso);

// Este es nuestro mejor intento de conseguir una condición de carrera
// creando una gran cantidad de hilos.
para (int i = 0; i <25000; i ++) {
    WidgetBuilderThread widgetBuilderThread =
        new WidgetBuilderThread (widgetBuilder, text, parent, failFlag);
    Thread thread = nuevo Thread (widgetBuilderThread);
    thread.start ();
}
asertEquals (falso, failFlag.get ());
}
```

Aclaración

A veces es útil traducir el significado de algún argumento oscuro o devolver valor en algo que sea legible. En general, es mejor encontrar una manera de argumentar valor de retorno o valor claro por derecho propio; pero cuando es parte de la biblioteca estándar, o en código que no puede alterar, entonces un comentario aclaratorio útil puede ser útil.

```
public void testCompareTo () arroja una excepción
{
    WikiPagePath a = PathParser.parse ("PáginaA");
    WikiPagePath ab = PathParser.parse ("PageA.PageB");
    WikiPagePath b = PathParser.parse ("PáginaB");
    WikiPagePath aa = PathParser.parse ("PageA.PageA");
    WikiPagePath bb = PathParser.parse ("PáginaB.PáginaB");
    WikiPagePath ba = PathParser.parse ("PageB.PageA");

    asertTrue (a.compareTo (a) == 0); // a == a
    asertTrue (a.compareTo (b)! = 0); // a! = b
    asertTrue (ab.compareTo (ab) == 0); // ab == ab
    asertTrue (a.compareTo (b) == -1); // a <b
    asertTrue (aa.compareTo (ab) == -1); // aa <ab
    asertTrue (ba.compareTo (bb) == -1); // ba <bb
    asertTrue (b.compareTo (a) == 1); // b> a
    asertTrue (ab.compareTo (aa) == 1); // ab> aa
    asertTrue (bb.compareTo (ba) == 1); // bb> ba
}
```

Por supuesto, existe un riesgo sustancial de que un comentario aclaratorio sea incorrecto. Ir a través del ejemplo anterior y vea lo difícil que es verificar que sean correctos. Esto explica tanto por qué la aclaración es necesaria y por qué es arriesgada. Entonces, antes de escribir En este tipo de situaciones, tenga cuidado de que no haya una mejor manera, y luego cuide aún más de que son precisos.

Página 89

Advertencia de consecuencias

A veces es útil advertir a otros programas grammers sobre ciertas consecuencias. Para ejemplo, aquí hay un comentario que explica por qué un caso de prueba en particular está desactivado:

```
// No corras a menos que
// tengo algo de tiempo para matar.
public void _testWithReallyBigFile ()
{
    writeLinesToFile (10000000);

    response.setBody (testFile);
    response.readyToSend (esto);
    String responseString = output.toString ();
    asertSubString ("Content-Length: 1000000000", responseString);
    asertTrue (bytesEnviados> 1000000000);
}
```

Hoy en día, por supuesto, apagaríamos el caso de prueba usando el atributo `@Ignore` con un cadena explicativa apropiada. `@Ignore ("Tarda demasiado en ejecutarse")` . Pero de vuelta en los días antes de JUnit 4, poner un guión bajo delante del nombre del método era una convención. El comentario, aunque frívolo, lo explica bastante bien.

Aquí hay otro ejemplo más conmovedor:

```
public static SimpleDateFormat makeStandardHttpDateFormat ()
{
    // SimpleDateFormat no es seguro para subprocesos,
    // por lo que necesitamos crear cada instancia de forma independiente.
    SimpleDateFormat df = new SimpleDateFormat ("EEE, dd MMM aaaa HH: mm: ss z");
    df.setTimeZone (TimeZone.getTimeZone ("GMT"));
    return df;
}
```

Puede quejarse de que existen mejores formas de resolver este problema. Podría estar de acuerdo con usted. Pero el comentario, como se da aquí, es perfectamente razonable. Evitará algunos excesos programador ansioso por usar un inicializador estático en nombre de la eficiencia.

TODO Comentarios

A veces es razonable dejar notas de "Tareas pendientes" en forma de comentarios `// TODO` . En el siguiente caso, el comentario `TODO` explica por qué la función tiene una implementación degenerada ción y cuál debería ser el futuro de esa función.

```
// TODO-MdM estos no son necesarios
// Esperamos que esto desaparezca cuando hagamos el modelo de pago
protegido VersionInfo makeVersion () arroja una excepción
{
    devolver nulo;
}
```


Página 90

no puedo hacer en este momento. Puede ser un recordatorio para eliminar una función obsoleta o para pedirle a otra persona que vea un problema. Podría ser una solicitud para que otra persona Piense en un nombre mejor o en un recordatorio para hacer un cambio que dependa de un evento planeado. Cualquier otra cosa que un TODO podría ser, es *no* una excusa para salir mal código en el sistema.

Hoy en día, la mayoría de los IDE buenos proporcionan funciones y gestos especiales para localizar todos los TODO comenta, por lo que no es probable que se pierdan. Aún así, no quieres tu código estar lleno de TODO s. Así que escanéalos con regularidad y elimina los que lata.

Amplificación

Se puede usar un comentario para amplificar la importancia de algo que de otra manera podría parecer inconsecuente.

```
String listItemContent = match.group (3) .trim ();
// el recorte es muy importante. Elimina el arranque
// espacios que pueden hacer que se reconozca el artículo
// como otra lista.
new ListItemWidget (esto, listItemContent, this.level + 1);
return buildList (text.substring (match.end ()));
```

Javadocs en API públicas

No hay nada tan útil y satisfactorio como una API pública bien descrita. El java-
Los documentos de la biblioteca estándar de Java son un buen ejemplo. Sería difícil, en el mejor de los casos, escribir Programas Java sin ellos.

Si está escribiendo una API pública, entonces debería escribir buenos javadocs para ella.
Pero tenga en cuenta el resto de los consejos de este capítulo. Los javadocs pueden ser igualmente engañosos, no local y deshonesto como cualquier otro tipo de comentario.

Comentarios malos

La mayoría de los comentarios entran en esta categoría. Por lo general, son muletas o excusas para un código deficiente. o justificaciones de decisiones insuficientes, que ascienden a poco más que el programador hablando consigo mismo.

Masculleo

Incluir un comentario solo porque cree que debe hacerlo o porque el proceso lo requiere, es un truco. Si decide escribir un comentario, dedique el tiempo necesario para asegurarse de que es el mejor comentario que puedes escribir.

Página 91

Aquí, por ejemplo, hay un caso que encontré en FitNesse, donde un comentario podría tener ha sido útil. Pero el autor tenía prisa o simplemente no prestaba mucha atención. Su madre-bling dejó un enigma:

```
loadProperties public void ()
{
    intentar
    {
      Cadena propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;
      FileInputStream propertiesStream = nuevo FileInputStream (propertiesPath);
      loadedProperties.load (propertiesStream);
    }
    captura (IOException e)
    {
        // Ningún archivo de propiedades significa que se cargan todos los valores predeterminados
    }
```

```
}
```

¿Qué significa ese comentario en el bloque de captura ? Claramente significó algo para el autor, pero el significado no llega tan bien. Aparentemente, si obtenemos un IOException , significa que no había ningún archivo de propiedades; y en ese caso todos los valores predeterminados son cargado. Pero, ¿quién carga todos los valores predeterminados? ¿Estaban cargados antes de la llamada a loadProperties.load ? ¿O loadProperties.load capturó la excepción, cargó los valores predeterminados, y luego pasar la excepción para que la ignoremos? ¿O loadProperties.load cargó todos los valores predeterminados antes de intentar cargar el archivo? ¿Estaba el autor tratando de consolarse a sí mismo sobre el hecho de que dejaba vacío el bloque de captura ? O, y esta es la posibilidad aterradora, estaba el autor tratando de decirse a sí mismo que volvería aquí más tarde y escribiría el código que cargar los valores predeterminados?

Nuestro único recurso es examinar el código en otras partes del sistema para averiguar qué es pasando. Cualquier comentario que le obligue a buscar en otro módulo el significado de ese El comentario no se ha podido comunicar con usted y no vale la pena los bits que consume.

Comentarios redundantes

El Listado 4-1 muestra una función simple con un comentario de encabezado que es completamente redundante. Es probable que el comentario tarde más en leer que el código en sí.

Listado 4-1

waitForClose

```
// Método de utilidad que regresa cuando this.closed es verdadero. Lanza una excepción
// si se alcanza el tiempo de espera.
waitForClose vacío sincronizado público (timeoutMillis largo final)
lanza una excepción
{
    si (! cerrado)
    {
        esperar (timeoutMillis);
        si (! cerrado)
            lanzar una nueva excepción ("MockResponseSender no se pudo cerrar");
    }
}
```


Página 92

¿Para qué sirve este comentario? Ciertamente no es más informativo que el código. No justifica el código, ni proporciona una intención o un fundamento. No es más fácil de leer que el código. De hecho, es menos preciso que el código y atrae al lector a aceptar esa falta de precisión en lugar de verdadera comprensión. Es como un vendedor de autos usados alegre asegurándole que no necesita mirar debajo del capó.

Ahora considere la legión de javadocs inútiles y redundantes en el Listado 4-2 tomado de Gato. Estos comentarios solo sirven para desordenar y oscurecer el código. No sirven ningún documento propósito mental en absoluto. Para empeorar las cosas, solo les mostré los primeros. Existen muchos más en este módulo.

Listado 4-2

ContainerBase.java (Tomcat)

```
ContainerBase clase pública abstracta
    implementa contenedor, ciclo de vida, tubería,
    MBeanRegistration, serializable {

    / **
     * La demora del procesador para este componente.
     * /
    protegido int backgroundProcessorDelay = -1;


    / **
     * El soporte de eventos de ciclo de vida para este componente.
     * /
    protected LifecycleSupport ciclo de vida =
        new LifecycleSupport (esto);
```

```
/ **
  * Los oyentes de eventos de contenedor para este contenedor.
  * /
oyentes de ArrayList protegidos = new ArrayList ();


/ **
  * La implementación de Loader con la que se encuentra este contenedor
  * asociado.
  * /
Loader loader protegido = null;


/ **
  * La implementación de Logger con la que está este Container
  * asociado.
  * /
Registrador de registro protegido = nulo;


/ **
  * Nombre del registrador asociado.
  * /
cadena protegida logName = nulo;
```


Página 93

Listado 4-2 (continuación)

ContainerBase.java (Tomcat)

```
/ **
  * La implementación de Manager con la que se encuentra este Container
  * asociado.
  * /
administrador administrador protegido = nulo;


/ **
  * El clúster con el que está asociado este contenedor.
  * /
cluster cluster protegido = nulo;


/ **
  * El nombre legible por humanos de este contenedor.
  * /
Nombre de cadena protegido = nulo;


/ **
  * El contenedor principal del que este contenedor es secundario.
  * /
contenedor protegido parent = null;


/ **
  * El cargador de clases principal que se configurará cuando instalemos un
  * Cargador.
  * /
protegido ClassLoader parentClassLoader = null;


/ **
  * El objeto Pipeline con el que se encuentra este contenedor
  * asociado.
  * /
pipeline protegido = new StandardPipeline (este);


/ **
  * El Reino con el que está asociado este Contenedor.
  * /
reino protegido reino = nulo;
```

```
/ **
 * El objeto de recursos DirContext con el que este contenedor
 * está asociado.
 * /
recursos protegidos de DirContext = nulo;
```


Página 94

Comentarios engañosos

A veces, con las mejores intenciones, un programador hace una declaración en sus comentarios. eso no es lo suficientemente preciso para ser exacto. Considere por otro momento lo mal redundante pero también un comentario sutilmente engañoso que vimos en el Listado 4-1.

¿Descubrió que el comentario era engañoso? El método no regresa *cuando* this.closed se convierte en verdad . Devuelve *si* this.closed es verdadero ; de lo contrario, espera un tiempo de espera ciego y luego lanza una excepción *si* this.closed aún no es cierto .

Esta sutil información errónea, expresada en un comentario que es más difícil de leer que el cuerpo del código, podría hacer que otro programador llamara alegremente a esta función en el expectativa de que regresará tan pronto como this.closed se haga realidad . Ese pobre programador luego se encontraría en una sesión de depuración tratando de averiguar por qué se ejecutó su código tan lentamente.

Comentarios obligatorios

Es simplemente una tontería tener una regla que diga que cada función debe tener un javadoc, o cada variable debe tener un comentario. Comentarios como este simplemente desordenan el código, propagan la puerta miente, y se presta a la confusión y desorganización general.

Por ejemplo, los javadocs requeridos para cada función conducen a abominaciones como Listing 4-3. Este desorden no agrega nada y solo sirve para ofuscar el código y crear el potencial para mentiras y extravíos.

Listado 4-3

```
/ **
 *
 * @param title El título del CD
 * @param author El autor del CD
 * @param tracks El número de pistas del CD
 * @param durationInMinutes La duración del CD en minutos
 * /
public void addCD (título de la cadena, autor de la cadena,
                   int tracks, int durationInMinutes) {
    CD cd = nuevo CD ();
    cd.title = título;
    cd.author = autor;
    cd.tracks = pistas;
    cd.duration = duración;
    cdList.add (cd);
}
```

Comentarios de la revista

A veces, las personas agregan un comentario al inicio de un módulo cada vez que lo editan. Estas Los comentarios se acumulan como una especie de diario, o registro, de cada cambio que se ha hecho. He visto algunos módulos con docenas de páginas de estas entradas continuas del diario.

Página 95


```
* Cambios (desde el 11 de octubre de 2001)
* -------------------------
* 11-Oct-2001: Reorganizó la clase y la movió a un nuevo paquete.
*                      com.jrefinery.date (DG);
* 05-Nov-2001: Se agregó un método getDescription () y se eliminó NotableDate
*                      clase (DG);
* 12 de noviembre de 2001: IBD requiere el método setDescription (), ahora que NotableDate
*                      la clase se ha ido (DG); Se cambió getPreviousDayOfWeek (),
*                      getFollowingDayOfWeek () y getNearestDayOfWeek () para corregir
*                      errores (DG);
* 05-Dec-2001: Se corrigió un error en la clase SpreadsheetDate (DG);
* 29 de mayo de 2002: se movieron las constantes de mes a una interfaz separada
*                      (MonthConstants) (DG);
* 27 de agosto de 2002: Se corrigió un error en el método addMonths (), gracias a N ??? levka Petr (DG);
* 03-Oct-2002: Errores corregidos reportados por Checkstyle (DG);
* 13 de marzo de 2003: Implementado Serializable (DG);
* 29 de mayo de 2003: Se corrigió un error en el método addMonths (DG);
* 04-Sep-2003: Implementado comparable. Se actualizó el isInRange javadocs (DG);
* 05-Jan-2005: Se corrigió un error en el método addYears () (1096282) (DG);
```

Hace mucho tiempo, había una buena razón para crear y mantener estas entradas de registro al principio. de cada módulo. No teníamos sistemas de control de código fuente que lo hicieran por nosotros. Hoy en día, sin embargo, estos largos diarios son más desordenados para ofuscar el módulo. Ellos deberían ser completamente eliminado.

Comentarios de ruido

A veces ves comentarios que no son más que ruido. Reafirman lo obvio y no proporcione información nueva.

```
/ **
  * Constructor predeterminado.
  * /
protected AnnualDateRule () {
}
```

No, de *verdad?* O que tal esto:

```
/ ** El día del mes. * /
      private int dayOfMonth;
```

Y luego está este modelo de redundancia:

```
/ **
  * Devuelve el día del mes.
  *
  * @return el día del mes.
  * /
public int getDayOfMonth () {
    return dayOfMonth;
}
```


Página 96

Estos comentarios son tan ruidosos que aprendemos a ignorarlos. A medida que leemos el código, nuestro los ojos simplemente los pasan por alto. Eventualmente, los comentarios comienzan a mentir a medida que el código que los rodea cambios.

El primer comentario del Listado 4-4 parece apropiado. [2] Explica por qué el bloque de captura está siendo ignorado. Pero el segundo comentario es puro ruido. Aparentemente, el programador fue tan frustrado con la escritura de bloques try / catch en esta función que necesitaba desahogarse.

Listado 4-4
startSending

```
privado void startSending ()
{
    intentar
    {
        doSending ();
    }
    captura (SocketException e)
    {
        // normal. alguien detuvo la solicitud.
    }
    atrapar (Excepción e)
    {
        intentar
        {
            response.add (ErrorResponder.makeExceptionString (e));
            response.closeAll ();
        }
        captura (Excepción e1)
        {
            //¡Dáme un respiro!
        }
    }
}
```

En lugar de desahogarse en un comentario inútil y ruidoso, el programador debería haber reconoció que su frustración podría resolverse mejorando la estructura de su código.
Debería haber redirigido su energía para extraer ese último bloque de intento / captura en un bloque separado. función, como se muestra en el Listado 4-5.

Listado 4-5
startSending (refactorizado)

```
privado void startSending ()
{
    intentar
    {
        doSending ();
    }
```

2. La tendencia actual de los IDE de revisar la ortografía en los comentarios será un bálsamo para aquellos de nosotros que leemos mucho código.

Página 97

Listado 4-5 (continuación)
startSending (refactorizado)

```
    captura (SocketException e)
    {
        // normal. alguien detuvo la solicitud.
    }
    atrapar (Excepción e)
    {
```

```
        addExceptionAndCloseResponse (e);
    }
}

private void addExceptionAndCloseResponse (Excepción e)
{
    intentar
    {
        response.add (ErrorResponder.makeExceptionString (e));
        response.closeAll ();
    }
    captura (Excepción e1)
    {
    }
}
```

Reemplace la tentación de crear ruido con la determinación de limpiar su código. Usted encontrarlo te convierte en un programador mejor y más feliz.

Ruido aterrador

Los javadocs también pueden ser ruidosos. ¿Para qué sirven los siguientes Javadocs (de un conocido biblioteca de código abierto) servir? Respuesta: nada. Son solo comentarios ruidosos redundantes escrito por algún deseo fuera de lugar de proporcionar documentación.

```
/** El nombre. * /
nombre de cadena privada;

/** La versión. * /
versión de cadena privada;

/ ** El nombre de la licencia. * /
private String licenceName;

/** La versión. * /
información de cadena privada;
```

Lea estos comentarios nuevamente con más atención. ¿Ves el error de cortar y pegar? Si los autores no prestan atención cuando los comentarios se escriben (o pegan), ¿por qué los lectores deberían espera sacar provecho de ellos?

Página 98

No use un comentario cuando pueda usar una función o una variable

Considere el siguiente tramo de código:

```
// ¿El módulo de la lista global <mod> depende del
// subsistema del que formamos parte?
si (smodule.getDependSubsystems (). contiene (subSysMod.getSubSystem ()))
```

Esto podría reformularse sin el comentario como

```
ArrayList moduleDependees = smodule.getDependSubsystems ();
String ourSubSystem = subSysMod.getSubSystem ();
si (moduleDependees.contains (ourSubSystem))
```

El autor del código original puede haber escrito el comentario primero (poco probable) y luego escribió el código para cumplir con el comentario. Sin embargo, el autor debería haber refactorizado el código, como hice yo, para que el comentario pueda ser eliminado.

Marcadores de posición

A veces, a los programadores les gusta marcar una posición particular en un archivo fuente. Por ejemplo, yo Recientemente encontré esto en un programa que estaba viendo:

```
// Acciones ///////////////////////////////////
```

Hay raras ocasiones en las que tiene sentido reunir ciertas funciones bajo un banner como este. Pero, en general, son un desorden que debe eliminarse, especialmente el ruidoso tren de barras al final.

Piénsalo de esta manera. Un banner es sorprendente y obvio si no ve los banners muy a menudo. Por lo tanto, utilícelos con moderación y solo cuando el beneficio sea significativo. Si abusa pancartas, caerán en el ruido de fondo y serán ignoradas.

Comentarios de llaves de cierre

A veces, los programadores colocarán comentarios especiales sobre llaves de cierre, como en el Listado 4-6. Aunque esto puede tener sentido para funciones largas con estructuras profundamente anidadas, sirve solo para desordenar el tipo de funciones pequeñas y encapsuladas que preferimos. Así que si encuentras usted mismo desea marcar sus llaves de cierre, intente acortar sus funciones en su lugar.

Listado 4-6

wc.java

```
public class wc {
   public static void main (String [] args) {
      BufferedReader in = new BufferedReader (nuevo InputStreamReader (System.in));
      Línea de cuerda;
      int lineCount = 0;
      int charCount = 0;
      int wordCount = 0;
      intentar {
```


Listado 4-6 (continuación)

wc.java

```
         while ((línea = in.readLine ())! = null) {
            lineCount ++;
            charCount + = line.length ();
            Palabras de cadena [] = line.split ("\\ W");
            wordCount + = words.length;
         } //tiempo
         System.out.println ("wordCount =" + wordCount);
         System.out.println ("lineCount =" + lineCount);
         System.out.println ("charCount =" + charCount);
      } // intentar
      catch (IOException e) {
         System.err.println ("Error:" + e.getMessage ());
      } //captura
   } //principal
}
```

Atribuciones y Bylines

```
/ * Agregado por Rick * /
```

Los sistemas de control de código fuente son muy buenos para recordar quién agregó qué, cuándo. No es necesario contaminar el código con pequeñas firmas. Podrías pensar que tal com- Los comentarios serían útiles para ayudar a otros a saber con quién hablar sobre el código. Pero el La realidad es que tienden a permanecer durante años y años, volviéndose cada vez menos precisos. y relevante.

Nuevamente, el sistema de control de código fuente es un lugar mejor para este tipo de información.

Código comentado

Pocas prácticas son tan odiosas como comentar el código. ¡No hagas esto!

```
         InputStreamResponse respuesta = nuevo InputStreamResponse ();
         response.setBody (formatter.getResultStream (), formatter.getByteCount ());
// InputStream resultsStream = formatter.getResultStream ();
```

```
// Lector StreamReader = new StreamReader (resultStream);
// response.setContent (reader.read (formatter.getByteCount ()));
```

Otros que ven ese código comentado no tendrán el valor de eliminarlo. Ellos pensarán está ahí por una razón y es demasiado importante para eliminarlo. Entonces, el código comentado se reúne como escoria en el fondo de una botella de vino en mal estado.

Considere esto de apache commons:

```
this.bytePos = writeBytes (pngIdBytes, 0);
// hdrPos = bytePos;
writeHeader ();
writeResolution ();
// dataPos = bytePos;
if (writeImageData ()) {
        writeEnd ();
        this.pngBytes = resizeByteArray (this.pngBytes, this.maxPos);
}
```


Página 100


```
demás {
        this.pngBytes = null;
}
return this.pngBytes;
```

¿Por qué se comentan esas dos líneas de código? ¿Son importantes? ¿Se quedaron como recordatorios de algún cambio inminente? ¿O son simplemente cruft que alguien comentó? hace años y simplemente no se ha molestado en limpiar.

Hubo un tiempo, allá por los años sesenta, en el que el código de comentario podría haber sido útil. Pero hemos tenido buenos sistemas de control de código fuente durante mucho tiempo. Esos los sistemas recordarán el código por nosotros. No tenemos que comentarlo más. Sólo eliminar el código. No lo perderemos. Promesa.

Comentarios HTML

HTML en los comentarios del código fuente es una abominación, como puede ver al leer el código debajo. Hace que los comentarios sean difíciles de leer en el único lugar donde deberían ser fáciles de leer. leer — el editor / IDE. Si los comentarios van a ser extraídos por alguna herramienta (como Javadoc) para aparecen en una página web, entonces debería ser responsabilidad de esa herramienta, y no del programamer, para adornar los comentarios con HTML apropiado.

```
/ **
  * Tarea para ejecutar pruebas de ajuste.
  * Esta tarea ejecuta pruebas de aptitud física y publica los resultados.
  * <p />
  * <pre>
  * Uso:
  * & lt; taskdef name = & quot; ejecutar-pruebas-de-ajuste & quot;
  * classname = & quot; fitnesse.ant.ExecuteFitnesseTestsTask & quot;
  * classpathref = & quot; classpath & quot; / & gt;
  * O
  * & lt; taskdef classpathref = & quot; classpath & quot;
  *                         resource = & quot; tasks.properties & quot; / & gt;
  * <p />
  * & lt; ejecutar pruebas de ajuste
  * suitepage = & quot; FitNesse.SuiteAcceptanceTests & quot;
  * fitnesseport = & quot; 8082 & quot;
  * resultsdir = & quot; $ {results.dir} & quot;
  * resultshtmlpage = & quot; fit-results.html & quot;
  * classpathref = & quot; classpath & quot; / & gt;
  * </pre>
  * /
```

Información no local

Si debe escribir un comentario, asegúrese de que describa el código junto al que aparece. No ofrecer información de todo el sistema en el contexto de un comentario local. Considere, por ejemplo, el comentario de javadoc a continuación. Aparte del hecho de que es terriblemente redundante, también ofrece información sobre el puerto predeterminado. Y, sin embargo, la función no tiene absolutamente ningún control sobre cuál es ese valor predeterminado. El comentario no describe la función, sino alguna otra, muy dis-

parte importante del sistema. Por supuesto, no hay garantía de que se cambie este comentario.
cuando se cambia el código que contiene el predeterminado.

Página 101


```
/ **
  * Puerto en el que se ejecutaría fitnesse. El valor predeterminado es <b> 8082 </b>.
  *
  * @param fitnessePort
  * /
public void setFitnessePort (int fitnessePort)
{
    this.fitnessePort = fitnessePort;
}
```

Demasiada información

No coloque discusiones históricas interesantes o descripciones irrelevantes de detalles en su comentarios. El siguiente comentario fue extraído de un módulo diseñado para probar que una función ción podría codificar y decodificar base64. Aparte del número RFC, alguien que lea esto El código no necesita la información arcana contenida en el comentario.

```
/ *
    RFC 2045 - Extensiones multipropósito de correo de Internet (MIME)
    Primera parte: formato de los cuerpos de mensajes de Internet
    sección 6.8. Codificación de transferencia de contenido Base64
    El proceso de codificación representa grupos de bits de entrada de 24 bits como salida
    cadenas de 4 caracteres codificados. Procediendo de izquierda a derecha, una
    El grupo de entrada de 24 bits se forma mediante la concatenación de 3 grupos de entrada de 8 bits.
    Estos 24 bits se tratan como 4 grupos de 6 bits concatenados, cada uno
    de los cuales se traduce a un solo dígito en el alfabeto base64.
    Al codificar un flujo de bits a través de la codificación base64, el flujo de bits
    debe suponerse que se ordena con el bit más significativo primero.
    Es decir, el primer bit de la secuencia será el bit de orden superior en
    el primer byte de 8 bits, y el octavo bit será el bit de orden inferior en
    el primer byte de 8 bits y así sucesivamente.
 * /
```

Conexión obvia

La conexión entre un comentario y el código que describe debería ser obvia. Si usted es si se toma la molestia de escribir un comentario, al menos le gustaría que el lector pudiera mire el comentario y el código y comprenda de qué está hablando el comentario.

Considere, por ejemplo, este comentario extraído de apache commons:

```
/ *
  * comience con una matriz que sea lo suficientemente grande para contener todos los píxeles
  * (más bytes de filtro) y 200 bytes adicionales para la información del encabezado
  * /
this.pngBytes = new byte [((this.width + 1) * this.height * 3) + 200];
```

¿Qué es un byte de filtro? ¿Se relaciona con el +1? ¿O al * 3? ¿Ambas cosas? ¿Es un píxel un byte? Por qué 200? El propósito de un comentario es explicar el código que no se explica a sí mismo. Es una pena cuando un comentario necesita su propia explicación.

Encabezados de funciones

Las funciones cortas no necesitan mucha descripción. Un nombre bien elegido para una pequeña función que una cosa suele ser mejor que un encabezado de comentario.

Página 102

Javadocs en código no público

Tan útiles como son los javadocs para las API públicas, son un anatema para el código que no está destinado para consumo público. Generando páginas javadoc para las clases y funciones dentro de un El sistema generalmente no es útil, y la formalidad adicional de los comentarios de javadoc asciende a poco más que cruft y distracción.

Ejemplo

Escribí el módulo en el Listado 4-7 para la primera *inmersión en XP* . Estaba destinado a ser un ejemplo de mala codificación y estilo de comentarios. Kent Beck luego refactorizó este código en un de forma mucho más agradable frente a varias docenas de estudiantes entusiastas. Más tarde me adapté el ejemplo de mi libro *Desarrollo de software ágil, principios, patrones y prácticas* y el primero de mis artículos de *Craftsman* publicado en *la* revista *Software Development* .

Lo que encuentro fascinante de este módulo es que hubo una época en la que muchos de nosotros lo habría considerado "bien documentado". Ahora lo vemos como un pequeño desastre. Ver cómo muchos problemas de comentarios diferentes que puede encontrar.

Listado 4-7

GeneratePrimes.java

```
/ **
  * Esta clase genera números primos hasta un usuario especificado
  * máximo. El algoritmo utilizado es el Tamiz de Eratóstenes.
  * <p>
  * Eratóstenes de Cirene, 276 a. C., Cirene, Libia -
  * dc 194, Alejandría. El primer hombre en calcular el
  * circunferencia de la Tierra. También conocido por trabajar en
  * calendarios con años bisiestos y dirigía la biblioteca de Alejandría.
  * <p>
  * El algoritmo es bastante simple. Dada una matriz de enteros
  * a partir de 2. Tacha todos los múltiplos de 2. Encuentra el siguiente
  * número entero sin cruzar y tachar todos sus múltiplos.
  * Repita hasta que haya pasado la raíz cuadrada del máximo
  * valor.
  *
  * @autor Alphonse
  * @versión 13 de febrero de 2002 atp
  * /
importar java.util. *;

clase pública GeneratePrimes
{
    / **
     * @param maxValue es el límite de generación.
     * /
    public static int [] generatePrimes (int maxValue)
    {
        if (maxValue> = 2) // el único caso válido
        {
            // declaraciones
            int s = maxValue + 1; // tamaño de la matriz
            booleano [] f = nuevo booleano [s];
            int i;
```


Página 103

Listado 4-7 (continuación)

GeneratePrimes.java

```
        // inicializa la matriz a verdadero.
        para (i = 0; i <s; i ++)
            f [i] = verdadero;

        // deshacerse de los no primos conocidos
        f [0] = f [1] = falso;

        // tamiz
        int j;
        para (i = 2; i <Math.sqrt (s) + 1; i ++)
        {
            if (f [i]) // si i no está cruzado, cruce sus múltiplos.
            {
                para (j = 2 * i; j <s; j + = i)
                    f [j] = falso; // múltiplo no es primo
            }
        }

        // ¿cuántos números primos hay?
        int count = 0;
        para (i = 0; i <s; i ++)
        {
            si (f [i])
                contar ++; // recuento de golpes.
        }

        int [] primes = new int [count];

        // mueve los números primos al resultado
        para (i = 0, j = 0; i <s; i ++)
        {
            si (f [i])                          // si es primo
                primos [j ++] = i;
        }

        devolver primos; // devuelve los primos
    }
    else // maxValue <2
        return new int [0]; // devuelve una matriz nula si la entrada es incorrecta.
  }
}
```

En el Listado 4-8 puede ver una versión refactorizada del mismo módulo. Tenga en cuenta que el uso de comentarios es significativamente restringido. Solo hay dos comentarios en todo el módulo. Ambos comentarios son de naturaleza explicativa.

Listado 4-8

PrimeGenerator.java (refactorizado)

```
/ **
 * Esta clase genera números primos hasta un usuario especificado
 * máximo. El algoritmo utilizado es el Tamiz de Eratóstenes.
 * Dada una matriz de números enteros que comienzan en 2:
 * Encuentra el primer número entero sin cruzar y tacha todos sus
```


Listado 4-8 (continuación)

PrimeGenerator.java (refactorizado)

```
 * múltiplos. Repita hasta que no haya más múltiplos
 * en la matriz.
 * /

PrimeGenerator de clase pública
{
   booleano estático privado [] cruzado;
   resultado privado estático int [];

   public static int [] generatePrimes (int maxValue)
   {
       si (maxValue <2)
           return new int [0];
       demás
       {
```

```
        uncrossIntegersUpTo (maxValue);
        crossOutMultiples ();
        putUncrossedIntegersIntoResult ();
        devolver resultado;
    }
}

vacío estático privado uncrossIntegersUpTo (int maxValue)
{
    CrossOut = new boolean [maxValue + 1];
    para (int i = 2; i <cruzadoOut.length; i ++)
        CrossOut [i] = falso;
}

crossOutMultiples vacío estático privado ()
{
    int limit = determineIterationLimit ();
    para (int i = 2; i <= límite; i ++)
        si (no cruzado (i))
            crossOutMultiplesOf (i);
}

privado estático int determineIterationLimit ()
{
    // Cada múltiplo de la matriz tiene un factor primo que
    // es menor o igual que la raíz del tamaño de la matriz,
    // para que no tengamos que tachar múltiplos de números
    // más grande que esa raíz.
    doble iterationLimit = Math.sqrt (cruzadoOut.length);
    return (int) iterationLimit;
}

vacío estático privado crossOutMultiplesOf (int i)
{
    para (int multiple = 2 * i;
            multiple <crossOut.length;
            múltiple + = i)
        cruzado [múltiple] = verdadero;
}
```


Página 105

Listado 4-8 (continuación)

PrimeGenerator.java (refactorizado)

```
    booleano estático privado notCrossed (int i)
    {
        retorno cruzado [i] == falso;
    }

    vacío estático privado putUncrossedIntegersIntoResult ()
    {
        resultado = nuevo int [numberOfUncrossedIntegers ()];
        para (int j = 0, i = 2; i <cruzadoOut.length; i ++)
            si (no cruzado (i))
                resultado [j ++] = i;
    }

    privado estático int numberOfUncrossedIntegers ()
    {
        int count = 0;
        para (int i = 2; i <cruzadoOut.length; i ++)
            si (no cruzado (i))
                contar ++;

        recuento de devoluciones;
    }
}
```

Es fácil argumentar que el primer comentario es redundante porque se parece mucho a la propia función `generatePrimes` . Aun así, creo que el comentario sirve para facilitar al lector la el algoritmo, así que me inclino a dejarlo.

El segundo argumento es casi con certeza necesario. Explica la razón de ser el uso de la raíz cuadrada como límite de bucle. No pude encontrar un nombre de variable simple, ni ninguna

estructura de codificación diferente que dejó en claro este punto. Por otro lado, el uso de la la raíz cuadrada podría ser una presunción. ¿De verdad estoy ahorrando tanto tiempo al limitar la iteración? a la raíz cuadrada? ¿El cálculo de la raíz cuadrada podría llevar más tiempo del que estoy ahorrando?

Vale la pena pensarlo. El uso de la raíz cuadrada como límite de iteración satisface la antigua C y hacker de lenguaje ensamblador en mí, pero no estoy convencido de que valga la pena el tiempo y el esfuerzo que todos los demás se gastarán en entenderlo.

Bibliografía

[KP78]: Kernighan y Plaugher, *Los elementos del estilo de programación* , 2d. ed., McGraw-Hill, 1978.

Página 106

5

Formateo

Cuando la gente mira bajo el capó, queremos que se impresione con la pulcritud,

coherencia y atención al detalle que perciben. Queremos que se sientan impresionados por la orden. Queremos que sus cejas se eleven mientras se desplazan por los módulos. Queremos que perciban que los profesionales han estado trabajando. Si en cambio ven un revuelto masa de código que parece haber sido escrito por un grupo de marineros borrachos, entonces son probable que concluya que la misma falta de atención al detalle impregna todos los demás aspectos de la proyecto.

Página 107

Debe tener cuidado de que su código esté bien formateado. Deberías elegir un conjunto de reglas simples que gobiernan el formato de su código, y luego debe aplicar consistentemente esas reglas. Si está trabajando en un equipo, entonces el equipo debe acordar un solo conjunto de reglas de formato y todos los miembros deben cumplir. Ayuda tener una herramienta automatizada que puede aplicar esas reglas de formato por usted.

El propósito del formateo

Primero que nada, seamos claros. El formato del código es *importante* . Es demasiado importante para ignorar y es demasiado importante para tratarlo religiosamente. El formato de código tiene que ver con la comunicación y la comunicación es la primera orden del día del desarrollador profesional.

Quizás pensó que "hacer que funcione" era la primera orden del día para un desarrollador profesional. Sin embargo, espero que a estas alturas este libro te haya desengañado de esa ocurrencia. La funcionalidad que crea hoy tiene muchas posibilidades de cambiar en el próximo versión, pero la legibilidad de su código tendrá un efecto profundo en todos los cambios que alguna vez se hará. El estilo de codificación y la legibilidad sientan precedentes que continúan afectar la capacidad de mantenimiento y la extensibilidad mucho después de que se haya cambiado el código original mas allá del reconocimiento. Su estilo y disciplina sobreviven, aunque su código no.

Entonces, ¿cuáles son los problemas de formato que nos ayudan a comunicarnos mejor?

Formato vertical

Comencemos con el tamaño vertical. ¿Qué tamaño debe tener un archivo fuente? En Java, el tamaño del archivo es cercano relacionado con el tamaño de la clase. Hablaremos del tamaño de la clase cuando hablemos de las clases. Para el momento, consideremos el tamaño del archivo.

¿Qué tamaño tienen la mayoría de los archivos fuente de Java? Resulta que existe una amplia gama de tamaños y algunas diferencias notables en el estilo. La figura 5-1 muestra algunas de esas diferencias.

Se representan siete proyectos diferentes. Junit, FitNesse, testNG, Time and Money, JDepend, Ant y Tomcat. Las lneas a travs de las casillas muestran el mnimo y el mximo mum longitudes de archivo en cada proyecto. El cuadro muestra aproximadamente un tercio (un estándar desviación [1]) de los archivos. El medio del cuadro es la media. Entonces, el tamaño de archivo promedio en el El proyecto FitNesse tiene aproximadamente 65 líneas, y aproximadamente un tercio de los archivos tienen entre 40 y Más de 100 líneas. El archivo más grande en FitNesse es de aproximadamente 400 líneas y el más pequeño es de 6 líneas. Tenga en cuenta que esta es una escala logarítmica, por lo que la pequeña diferencia en la posición vertical implica una gran diferencia en tamaño absoluto.

1. El cuadro muestra sigma / 2 por encima y por debajo de la media. Si, sé que la distribución de la longitud del archivo no es normal, por lo que el estándar La desviación de Dard no es matemáticamente precisa. Pero no estamos tratando de ser precisos aquí. Solo estamos tratando de tener una idea.

Página 108

Figura 5-1

Distribuciones de longitud de archivo Escala LOG (altura de caja = sigma)

Junit, FitNesse y Time and Money se componen de archivos relativamente pequeños. Ninguno tienen más de 500 líneas y la mayoría de esos archivos tienen menos de 200 líneas. Tomcat y Ant, en el Por otro lado, algunos archivos tienen varios miles de líneas y cerca de la mitad han terminado. 200 líneas.

¿Qué significa eso para nosotros? Parece posible construir sistemas importantes (FitNesse tiene cerca de 50.000 líneas) de archivos que suelen tener 200 líneas de longitud, con un límite superior de 500. Aunque esta no debería ser una regla estricta, debería considerarse muy deseable. Los archivos pequeños suelen ser más fáciles de entender que los archivos grandes.

La metáfora del periódico

Piense en un artículo de periódico bien escrito. Lo lees verticalmente. En la cima esperas un titular que le dirá de qué se trata la historia y le permitirá decidir si es algo que quieras leer. El primer párrafo le ofrece una sinopsis de toda la historia, ocultando todos los detalles mientras le brinda los conceptos generales. A medida que continúas bajando Ward, los detalles aumentan hasta que tenga todas las fechas, nombres, citas, reclamos y otros minucias.

Nos gustaría que un archivo fuente fuera como un artículo de periódico. El nombre debe ser simple pero explicativo. El nombre, por sí solo, debería ser suficiente para decirnos si estamos en el módulo correcto o no. Las partes superiores del archivo de origen deben proporcionar el alto nivel

Página 109

conceptos y algoritmos. El detalle debe aumentar a medida que nos movemos hacia abajo, hasta que al final encontramos las funciones y detalles de nivel más bajo en el archivo fuente.

Un periódico se compone de muchos artículos; la mayoría son muy pequeñas. Algunos son un poco más grandes. Muy pocos contienen tanto texto como puede contener una página. Esto hace que el periódico sea *utilizable* . Si el periódicos eran sólo una larga historia que contenía una aglomeración desorganizada de hechos, fechas y nombres, entonces simplemente no lo leeríamos.

Apertura vertical entre conceptos

Casi todo el código se lee de izquierda a derecha y de arriba a abajo. Cada línea representa una expresión o una cláusula, y cada grupo de líneas representa un pensamiento completo. Esos pensamientos deben ser separados entre sí con líneas en blanco.

Considere, por ejemplo, el Listado 5-1. Hay líneas en blanco que separan el paquete. declaración, importación (es) y cada una de las funciones. Esta regla extremadamente simple tiene un pro- encontró efecto en el diseño visual del código. Cada línea en blanco es una señal visual que identifica un concepto nuevo y separado. Mientras explora la lista, su atención se dirige a la primera línea que sigue a una línea en blanco.

Listado 5-1

BoldWidget.java

```
paquete fitnesse.wikitext.widgets;

importar java.util.regex. *;

La clase pública BoldWidget extiende ParentWidget {
    Public static final String REGEXP = "" '. +?' "";
    patrón de patrón final estático privado = Pattern.compile ("" '(. +?)' "",
        Patrón.MULTILINE + Patrón.DOTALL
    );

    public BoldWidget (ParentWidget parent, String text) lanza Exception {
        super (padre);
        Coincidencia coincidente = patrón.matcher (texto);
        match.find ();
        addChildWidgets (match.group (1));
    }

    public String render () arroja Exception {
        StringBuffer html = new StringBuffer ("<b>");
        html.append (childHtml ()). append ("</b>");
        return html.toString ();
    }
}
```

Quitar esas líneas en blanco, como en el Listado 5-2, tiene un efecto de oscurecimiento notable en la legibilidad del código.

Listado 5-2

BoldWidget.java

```
paquete fitnesse.wikitext.widgets;
importar java.util.regex. *;
La clase pública BoldWidget extiende ParentWidget {
    Public static final String REGEXP = "" '. +?' "";
    patrón de patrón final estático privado = Pattern.compile ("" '(. +?)' "",
        Patrón.MULTILINE + Patrón.DOTALL);
    public BoldWidget (ParentWidget parent, String text) lanza Exception {
        super (padre);
```

```
        Coincidencia coincidente = patrón.matcher (texto);
        match.find ();
        addChildWidgets (match.group (1));}
    public String render () arroja Exception {
        StringBuffer html = new StringBuffer ("<b>");
        html.append (childHtml ()). append ("</b>");
        return html.toString ();
    }
}
```

Este efecto es aún más pronunciado cuando desenfoca los ojos. En el primer ejemplo las diferentes agrupaciones de líneas aparecen, mientras que el segundo ejemplo parece un confusión. La diferencia entre estos dos listados es un poco de apertura vertical.

Densidad vertical

Si la apertura separa los conceptos, entonces la densidad vertical implica una asociación cercana. Entonces lineas del código que está estrechamente relacionado debe aparecer verticalmente denso. Observa cómo lo inútil los comentarios del Listado 5-3 rompen la estrecha asociación de las dos variables de instancia.

Listado 5-3

```
Public class ReporterConfig {

  / **
    * El nombre de la clase del oyente reportero
    * /
  private String m_className;

  / **
    * Las propiedades del oyente reportero
    * /
  Lista privada <Propiedad> m_properties = new ArrayList <Propiedad> ();

  public void addProperty (propiedad de propiedad) {
      m_properties.add (propiedad);
  }
```

El listado 5-4 es mucho más fácil de leer. Encaja en un "ojo lleno", o al menos lo hace para mí. I puede mirarlo y ver que esta es una clase con dos variables y un método, sin tener que muevo mucho la cabeza o los ojos. La lista anterior me obliga a usar mucho más ojo y movimiento de la cabeza para lograr el mismo nivel de comprensión.

Listado 5-4

```
Public class ReporterConfig {
  private String m_className;
  Lista privada <Propiedad> m_properties = new ArrayList <Propiedad> ();

  public void addProperty (propiedad de propiedad) {
      m_properties.add (propiedad);
  }
```

Distancia vertical

¿Alguna vez te has perseguido en una clase, saltando de una función a la siguiente, desplazarse hacia arriba y hacia abajo en el archivo de origen, tratando de adivinar cómo se relacionan las funciones y operar, sólo para perderse en un nido de ratas de confusión? ¿Alguna vez has cazado en la cadena de herencia para la definición de una variable o función? Esto es frustrante porque eres tratando de entender *lo que hace* el sistema, pero está gastando su tiempo y mental energía en tratar de localizar y recordar *dónde están* las piezas.

Los conceptos que están estrechamente relacionados deben mantenerse verticalmente cerca unos de otros [G10]. Claramente, esta regla no funciona para conceptos que pertenecen a archivos separados. Pero luego de cerca Los conceptos relacionados no deben separarse en archivos diferentes a menos que tenga una muy buena razón. De hecho, esta es una de las razones por las que deben evitarse las variables protegidas.

Para aquellos conceptos que están tan estrechamente relacionados que pertenecen al mismo archivo fuente, su separación vertical debe ser una medida de cuán importante es cada uno para la comprensión capacidad del otro. Queremos evitar obligar a nuestros lectores a recorrer nuestra fuente. archivos y clases.

Declaraciones de variables. Las variables deben declararse lo más cerca posible de su uso. ble. Debido a que nuestras funciones son muy cortas, las variables locales deberían aparecer en la parte superior de cada función, como en esta función larga de Junit4.3.1.

```
private static void readPreferences () {
    InputStream es = nulo;
    intentar {
        es = nuevo FileInputStream (getPreferencesFile ());
        setPreferences (nuevas propiedades (getPreferences ()));
        getPreferences (). load (es);
    } captura (IOException e) {
        intentar {
            si (es! = nulo)
                está cerca();
        } captura (IOException e1) {
        }
    }
}
```

Las variables de control para bucles generalmente deben declararse dentro de la declaración de bucle, como en este linda función de la misma fuente.

Página 112


```
public int countTestCases () {
    int count = 0;
    para ( Probar cada : pruebas)
        count + = each.countTestCases ();
    recuento de devoluciones;
}
```

En casos raros, una variable puede declararse en la parte superior de un bloque o justo antes de un bucle en un función larga. Puede ver una variable de este tipo en este fragmento en medio de una muy larga función en TestNG.

```
...
para (prueba XmlTest: m_suite.getTests ()) {
        TestRunner tr = m_runnerFactory.newTestRunner (esto, prueba);
        tr.addListener (m_textReporter);
        m_testRunners.add (tr);

        invoker = tr.getInvoker ();

        para (ITestNGMethod m: tr.getBeforeSuiteMethods ()) {
            beforeSuiteMethods.put (m.getMethod (), m);
        }

        para (ITestNGMethod m: tr.getAfterSuiteMethods ()) {
            afterSuiteMethods.put (m.getMethod (), m);
        }
    }
...
```

Las variables de instancia, por otro lado, deben declararse en la parte superior de la clase. Esto no debe aumentar la distancia vertical de estas variables, porque en un diseño bien diseñado class, son utilizados por muchos, si no todos, los métodos de la clase.

Ha habido muchos debates sobre dónde deberían ir las variables de instancia. En C ++ nosotros practicaba comúnmente la llamada *regla de las tijeras* , que coloca todas las variables de instancia en el fondo. La convención común en Java, sin embargo, es colocarlos a todos en la parte superior de la clase. No veo ninguna razón para seguir ninguna otra convención. Lo importante es, por ejemplo, la variación capaces de ser declarados en un lugar conocido. Todo el mundo debería saber a dónde ir a ver las declaraciones.

Considere, por ejemplo, el extraño caso de la clase TestSuite en JUnit 4.3.1. tengo atenuó enormemente esta clase para aclarar el punto. Si miras aproximadamente a la mitad de la lista, verá dos variables de instancia declaradas allí. Sería difícil esconderlos de una manera mejor. lugar. Alguien que lea este código tendría que tropezar con las declaraciones por accidente. abolladura (como hice yo).

```
TestSuite de clase pública implementa Test {
    prueba pública estática createTest (Class <? extiende TestCase> theClass,
                                                Nombre de cadena) {
        ...
    }
```


Página 113


```
    constructor estático público <? extiende TestCase>
    getTestConstructor (Class <? extiende TestCase> theClass)
    lanza NoSuchMethodException {
        ...
    }

    Advertencia de prueba estática pública (mensaje de cadena final) {
        ...
    }

    excepción de cadena estática privada a cadena (Throwable t) {
        ...
    }

    Private String fName;

    Private Vector <Test> fTests = new Vector <Test> (10);

    public TestSuite () {
    }

      Public TestSuite (clase final <? extiende TestCase> theClass) {
          ...
    }

    Public TestSuite (Class <? extiende TestCase> theClass, String name) {
        ...
    }
    ... ... ... ... ... ...
}
```

Funciones dependientes. Si una función llama a otra, deberían estar verticalmente cerca, y la persona que llama debe estar por encima de la persona que llama, si es posible. Esto le da al programa un flujo. Si la convención se sigue de manera confiable, los lectores podrán confiar en esa función definida seguirán poco después de su uso. Considere, por ejemplo, el fragmento de FitNesse en el Listado 5-5. Observe cómo la función superior llama a los que están debajo y cómo ellos, a su vez, llame a los que están debajo de ellos. Esto facilita la búsqueda de las funciones llamadas y mejora enormemente la legibilidad de todo el módulo.

Listado 5-5

WikiPageResponder.java

```
WikiPageResponder de clase pública implementa SecureResponder {
    página protegida de WikiPage;
    PageData protegido pageData;
    protected String pageTitle;
    Solicitud de solicitud protegida;
    rastreador PageCrawler protegido;

    makeResponse de respuesta pública (contexto de FitNesseContext, solicitud de solicitud)
        lanza Exception {
        String pageName = getPageNameOrDefault (solicitud, "FrontPage");
```

Página 114

Listado 5-5 (continuación)

WikiPageResponder.java

```
        loadPage (nombre de la página, contexto);
        si (página == nulo)
            return notFoundResponse (contexto, solicitud);
        demás
            return makePageResponse (contexto);
    }

    private String getPageNameOrDefault (solicitud de solicitud, String defaultPageName)
    {
        String pageName = request.getResource ();
        if (StringUtil.isBlank (nombrePágina))
            pageName = defaultPageName;

        return pageName;
    }

    protected void loadPage (recurso String, contexto FitNesseContext)
        lanza Exception {
        WikiPagePath ruta = PathParser.parse (recurso);
        rastreador = context.root.getPageCrawler ();
        crawler.setDeadEndStrategy (new VirtualEnabledPageCrawler ());
        page = crawler.getPage (context.root, ruta);
        si (página! = nulo)
            pageData = page.getData ();
    }

    respuesta privada notFoundResponse (contexto FitNesseContext, solicitud de solicitud)
        lanza Exception {
        devolver nuevo NotFoundResponder (). makeResponse (contexto, solicitud);
    }

    Private SimpleResponse makePageResponse (contexto FitNesseContext)
        lanza Exception {
        pageTitle = PathParser.render (crawler.getFullPath (página));
        String html = makeHtml (contexto);

        Respuesta SimpleResponse = new SimpleResponse ();
        response.setMaxAge (0);
        response.setContent (html);
        respuesta de retorno;
    }
...
```

Como acotación al margen, este fragmento proporciona un buen ejemplo de cómo mantener constantes en el comió nivel [G35]. La constante "FrontPage" podría haberse enterrado en el `getPageNameOrDefault` , pero eso habría ocultado un conocido y esperado constante en una función inapropiadamente de bajo nivel. Era mejor pasar esa constante hacia abajo desde el lugar donde tiene sentido conocerlo hasta el lugar que realmente lo usa.

Página 115

Afinidad conceptual. Ciertos bits de código *quieren* estar cerca de otros bits. Tienen un cierto afinidad conceptual. Cuanto más fuerte sea esa afinidad, menos distancia vertical debe haber entre ellos.

Como hemos visto, esta afinidad puede basarse en una dependencia directa, como una función llamada- ing otra, o una función usando una variable. Pero hay otras posibles causas de afinidad. Afinidad puede deberse a que un grupo de funciones formar una operación similar. Considere este fragmento de código de Junit 4.3.1:

```
public class Assert {
    static public void assertTrue (mensaje de cadena, condición booleana) {
        si (! condición)
            fallar (mensaje);
    }

    static public void assertTrue (condición booleana) {
        asertTrue (nulo, condición);
    }

    static public void assertFalse (mensaje de cadena, condición booleana) {
        asertTrue (mensaje,! condición);
    }

    static public void assertFalse (condición booleana) {
        asertFalse (nulo, condición);
    }
...
```

Estas funciones tienen una fuerte afinidad conceptual porque comparten un nombre común planificar y realizar variaciones de la misma tarea básica. El hecho de que se llamen es secundario. Incluso si no lo hicieran, todavía querrían estar juntos.

Orden vertical

En general, queremos que las dependencias de llamadas a funciones apunten hacia abajo. Es decir, una función que se llama debe estar debajo de una función que realiza la llamada. [2] Esto crea una Buen flujo por el módulo de código fuente de nivel alto a nivel bajo.

Como en los artículos periodísticos, esperamos que los conceptos más importantes sean lo primero, y esperamos que se expresen con la menor cantidad de detalles contaminantes. Esperamos el los detalles de bajo nivel vendrán al final. Esto nos permite hojear los archivos de origen, obteniendo la esencia del

2. Esto es exactamente lo contrario de lenguajes como Pascal, C y C ++ que imponen funciones para ser definidas, o al menos declaradas, *antes de* que se utilicen.

Página 116

primeras funciones, sin tener que sumergirnos en los detalles. El listado 5-5 es organizado de esta manera. Quizás incluso mejores ejemplos son el Listado 15-5 en la página 263, y la Lista- ing 3-7 en la página 50.

Formato horizontal

¿Qué tan ancha debe ser una línea? Para responder a eso, veamos qué tan anchas son las líneas en los programas. Nuevamente, examinamos los siete proyectos diferentes. La figura 5-2 muestra la distribución de longitudes de línea de los siete proyectos. La regularidad es impresionante, especialmente alrededor 45 caracteres. De hecho, cada tamaño de 20 a 60 representa aproximadamente el 1 por ciento del total. número de líneas. ¡Eso es 40 por ciento! Quizás otro 30 por ciento tenga menos de 10 caracteres amplio. Recuerde que esta es una escala logarítmica, por lo que la apariencia lineal de la caída por encima de 80 caracteres acters es realmente muy significativo. Los programadores claramente prefieren las líneas cortas.

Figura 5-2
Distribución del ancho de línea de Java

Esto sugiere que deberíamos esforzarnos por mantener nuestras líneas cortas. El antiguo límite de Hollerith de 80 es un poco arbitrario y no me opongo a que las líneas superen las 100 o incluso las 120. Pero más allá de eso probablemente sea simplemente descuidado.

Solía seguir la regla de que nunca debes desplazarte hacia la derecha. Pero monitores son demasiado anchas para eso hoy en día, y los programadores más jóvenes pueden reducir la fuente tan pequeña

Página 117

que pueden obtener 200 caracteres en la pantalla. No hagas eso. Yo personalmente establezco mi límite en 120.

Apertura y densidad horizontales

Usamos espacios en blanco horizontales para asociar cosas que están fuertemente relacionadas y disociadas cosas que están más débilmente relacionadas. Considere la siguiente función:

```
línea de medida vacía privada (línea de cadena) {
    lineCount ++;
    int lineSize = line.length ();
    totalChars + = lineSize;
    lineWidthHistogram.addLine (lineSize, lineCount);
    recordWidestLine (tamaño de línea);
}
```

Rodeé los operadores de asignación con espacios en blanco para acentuarlos. Asignación
Las declaraciones tienen dos elementos distintos y principales: el lado izquierdo y el lado derecho. La

los espacios hacen que esa separación sea obvia.

Por otro lado, no puse espacios entre los nombres de las funciones y la apertura paréntesis. Esto se debe a que la función y sus argumentos están estrechamente relacionados. Separat- al hacerlos aparecer como separados en lugar de unidos. Yo separo argumentos dentro la función llama a paréntesis para acentuar la coma y mostrar que los argumentos son separar.

Otro uso del espacio en blanco es acentuar la precedencia de los operadores.

```
clase pública cuadrática {
    public static double root1 (doble a, doble b, doble c) {
        determinante doble = determinante (a, b, c);
        return (-b + Math.sqrt (determinante)) / (2 * a);
    }

    public static double root2 (int a, int b, int c) {
        determinante doble = determinante (a, b, c);
        return (-b - Math.sqrt (determinante)) / (2 * a);
    }

    determinante doble estático privado (doble a, doble b, doble c) {
        return b * b - 4 * a * c;
    }
}
```

Observe lo bien que se leen las ecuaciones. Los factores no tienen espacios en blanco entre ellos. porque son de alta precedencia. Los trminos estn separados por espacios en blanco porque ción y resta tienen menor precedencia.

Desafortunadamente, la mayoría de las herramientas para reformatear el código son ciegas a la precedencia de operadores e imponer el mismo espaciado en todas partes. Espacios tan sutiles como esos que se muestran arriba tienden a perderse después de reformatear el código.

Página 118

Alineación horizontal

Cuando era programador en lenguaje ensamblador, ₃ usé la alineación horizontal para acentuar ciertas estructuras. Cuando comencé a codificar en C, C ++ y, finalmente, Java, seguí probando para alinear todos los nombres de variables en un conjunto de declaraciones, o todos los rvalores en un conjunto de asignaciones declaraciones de ment. Mi código podría haberse visto así:

```
La clase pública FitNesseExpediter implementa ResponseSender
{
    toma privada                    enchufe;
    entrada privada InputStream;
    salida OutputStream privada;
    Solicitud privada               pedido;
    respuesta privada               respuesta;
    contexto privado FitNesseContext;
    protegido por mucho tiempo      requestParsingTimeLimit;
    privado largo                   requestProgress;
    privado largo                   requestParsingDeadline;
    booleano privado                hasError;

    public FitNesseExpediter (Socket                    s,
                              FitNesseContext context) arroja una excepción
    {
        this.context =              contexto;
        socket =                    s;
        entrada =                   s.getInputStream ();
        salida =                    s.getOutputStream ();
        requestParsingTimeLimit = 10000;
    }
```

Sin embargo, he descubierto que este tipo de alineación no es útil. La alineación parece enfatizar las cosas incorrectas y desviar mi mirada de la verdadera intención. Por ejemplo, en la lista de declaraciones de arriba, se siente tentado a leer la lista de nombres de variables sin-

mirando a sus tipos. Del mismo modo, en la lista de enunciados de asignación está tentado a Mire la lista de valores r sin ver nunca el operador de asignación. Y al hacer las cosas lo que es peor, las herramientas de reformateo automático suelen eliminar este tipo de alineación.

Entonces, al final, ya no hago este tipo de cosas. Hoy en día prefiero desalineado declaraciones y asignaciones, como se muestra a continuación, porque señalan una defi- eficiencia. Si tengo listas largas que deben alinearse, *el problema es la longitud de las listas* , no la falta de alineación. La longitud de la lista de declaraciones en FitNesseExpediter a continuación sugiere que esta clase debería dividirse.

```
La clase pública FitNesseExpediter implementa ResponseSender
{
    toma de corriente privada;
    entrada privada InputStream;
    salida OutputStream privada;
    solicitud de solicitud privada;
```

3. ¿A quién engaño? Sigo siendo un programador en lenguaje ensamblador. Puedes alejar al chico del metal, pero no puedes saca el metal del chico!

Página 119


```
    respuesta de respuesta privada;
    contexto privado FitNesseContext;
    protected long requestParsingTimeLimit;
    requestProgress largo privado;
    private long requestParsingDeadline;
    hasError booleano privado;

    public FitNesseExpediter (Socket s, contexto FitNesseContext) arroja una excepción
    {
        this.context = context;
        socket = s;
        input = s.getInputStream ();
        salida = s.getOutputStream ();
        requestParsingTimeLimit = 10000;
    }
```

Sangría

Un archivo fuente es una jerarquía más bien como un esquema. Hay información que pertenece al archivo como un todo, a las clases individuales dentro del archivo, a los métodos dentro de las clases, a los bloques dentro de los métodos, y recursivamente a los bloques dentro de los bloques. Cada El nivel de esta jerarquía es un ámbito en el que se pueden declarar nombres y en el que se pueden declarar interpretaciones y sentencias ejecutables.

Para hacer visible esta jerarquía de alcances, sangramos las líneas de código fuente en pro- porción a su posición en la jerarquía. Declaraciones a nivel del archivo, como la mayoría declaraciones de clase, no tienen sangría en absoluto. Los métodos dentro de una clase tienen sangría de un nivel a la derecha de la clase. Las implementaciones de esos métodos se implementan en un nivel para el derecho de la declaración del método. Las implementaciones de bloques se implementan en un nivel a la derecha de su bloque contenedor, y así sucesivamente.

Los programadores dependen en gran medida de este esquema de sangría. Alinean visualmente las líneas en la izquierda para ver en qué ámbito aparecen. Esto les permite saltar rápidamente sobre los ámbitos, como implementaciones de declaraciones `if` o `while` , que no son relevantes para su situación. Exploran la izquierda en busca de nuevas declaraciones de métodos, nuevas variables e incluso nuevas clases. Sin sangría, los programas serían virtualmente ilegibles para los humanos.

Considere los siguientes programas que son sintáctica y semánticamente idénticos:

```
La clase pública FitNesseServer implementa SocketServer {Private FitNesseContext
contexto; public FitNesseServer (contexto FitNesseContext) {this.context =
contexto; } servicio vacío público (Socket s) {servicio (s, 10000); } vacío público
serve (Socket s, requestTimeout largo) {try {FitNesseExpediter sender = new
FitNesseExpediter (s, contexto);
sender.setRequestParsingTimeLimit (requestTimeout); sender.start (); }
catch (Excepción e) {e.printStackTrace (); }}}

-----
```

```
La clase pública FitNesseServer implementa SocketServer {
    contexto privado FitNesseContext;
```


Página 120

Formato horizontal 89

```
    public FitNesseServer (contexto FitNesseContext) {
        this.context = context;
    }

    servicio vacío público (Socket s) {
        servir (s, 10000);
    }

    public void serve (Socket s, requestTimeout largo) {
        intentar {
            FitNesseExpediter remitente = nuevo FitNesseExpediter (s, contexto);
            sender.setRequestParsingTimeLimit (requestTimeout);
            sender.start ();
        }
        captura (Excepción e) {
            e.printStackTrace ();
        }
    }
}
```

Su ojo puede discernir rápidamente la estructura de la lima con sangría. Casi al instante
detectar las variables, constructores, descriptores de acceso y métodos. Solo toma unos segundos darse cuenta
Tenga en cuenta que se trata de una especie de interfaz simple para un enchufe, con un tiempo de espera. Los sin sangría
La versión, sin embargo, es virtualmente impenetrable sin un estudio intenso.

Rompiendo Sangría. A veces es tentador romper brevemente la regla de sangría.
Si las declaraciones, corta , mientras que los bucles o funciones cortos. Siempre que he sucumbido a esto
tentación, casi siempre he vuelto y he vuelto a colocar la muesca. Así que evito col-
traspaso de alcances a una línea como esta:

```
La clase pública CommentWidget extiende TextWidget
{
    Cadena final estática pública REGEXP = "^ # [^ \ r \ n] * (?: (?: \ r \ n) | \ n | \ r)?";

    Public CommentWidget (ParentWidget parent, String text) {super (parent, text);}
    public String render () arroja Exception {return ""; }
}
```

Prefiero expandir y sangrar los ámbitos, así:

```
La clase pública CommentWidget extiende TextWidget {
    Cadena final estática pública REGEXP = "^ # [^ \ r \ n] * (?: (?: \ r \ n) | \ n | \ r)?";

    Public CommentWidget (ParentWidget parent, String text) {
        super (padre, texto);
    }

    public String render () arroja Exception {
        regreso "";
    }
}
```

Página 121

Telescopios ficticios

A veces, el cuerpo de una sentencia while o for es ficticia, como se muestra a continuación. No me gusta este tipo de estructuras y tratar de evitarlas. Cuando no puedo evitarlos, me aseguro de que el cuerpo del maniquí está debidamente sangrado y rodeado de tirantes. No puedo decirte como muchas veces he sido engañado por un punto y coma en silencio sentado en el extremo de un tiempo bucle en la misma línea. A menos que haga *visible* ese punto y coma al sangrarlo en su propia línea, es demasiado difícil de ver.

```
while (dis.read (buf, 0, readBufferSize)! = -1)
    ;
```

Reglas del equipo

El título de esta sección es un juego de palabras. Cada programador tiene el suyo reglas de formato favoritas, pero si funciona en un equipo, entonces el equipo gobierna.

Un equipo de desarrolladores debería estar de acuerdo en un solo estilo de formato, y luego cada miembro de ese equipo debe usar ese estilo. Queremos que el software tenga una estilo consistente. No queremos que parezca haber sido escrito por un grupo de personas en desacuerdo. individuos.

Cuando comencé el proyecto FitNesse en 2002, me senté con el equipo a trabajar un estilo de codificación. Esto tomó unos 10 minutos. Decidimos dónde pondríamos nuestros aparatos ortopédicos, cuál sería nuestro tamaño de sangría, cómo nombraríamos clases, variables y métodos, y así sucesivamente. Luego codificamos esas reglas en el formateador de código de nuestro IDE y nos hemos quedado atascados con ellos desde entonces. Estas no eran las reglas que yo prefiero; eran reglas decididas por el equipo. Como miembro de ese equipo, los seguí al escribir código en FitNesse proyecto.

Recuerde, un buen sistema de software se compone de un conjunto de documentos que leen bien. Necesitan tener un estilo consistente y suave. El lector debe poder confíe en que los gestos de formato que ha visto en un archivo fuente significarán lo mismo cosa en otros. Lo último que queremos hacer es agregar más complejidad al código fuente al escribiéndolo en una mezcla de diferentes estilos individuales.

Reglas de formato del tío Bob

Las reglas que utilizo personalmente son muy simples y están ilustradas por el código del Listado 5-6. Considere esto como un ejemplo de cómo el código hace el mejor documento estándar de codificación.

Página 122

Listado 5-6

CodeAnalyzer.java

```
CodeAnalyzer de clase pública implementa JavaFileAnalysis {
    private int lineCount;
    private int maxLineWidth;
    private int widestLineNumber;
    privado LineWidthHistogram lineWidthHistogram;
    private int totalChars;

    public CodeAnalyzer () {
        lineWidthHistogram = new LineWidthHistogram ();
    }

    Public static List <File> findJavaFiles (File parentDirectory) {
        Lista de archivos <File> = new ArrayList <File> ();
        findJavaFiles (parentDirectory, archivos);
        devolver archivos;
    }

    private static void findJavaFiles (File parentDirectory, List <File> archivos) {
        para (archivo de archivo: parentDirectory.listFiles ()) {
            if (file.getName (). endsWith (". java"))
                files.add (archivo);
            else if (file.isDirectory ())
                findJavaFiles (archivo, archivos);
        }
    }

    public void analysisFile (archivo javaFile) arroja una excepción {
        BufferedReader br = new BufferedReader (nuevo FileReader (javaFile));
        Línea de cuerda;
        while ((línea = br.readLine ())! = null)
            medirLine (línea);
    }

    línea de medida vacía privada (línea de cadena) {
        lineCount ++;
        int lineSize = line.length ();
        totalChars + = lineSize;
        lineWidthHistogram.addLine (lineSize, lineCount);
        recordWidestLine (tamaño de línea);
    }

    private void recordWidestLine (int lineSize) {
        if (lineSize> maxLineWidth) {
            maxLineWidth = lineSize;
            widestLineNumber = lineCount;
        }
    }

    public int getLineCount () {
        return lineCount;
    }

    public int getMaxLineWidth () {
        return maxLineWidth;
    }
```


Listado 5-6 (continuación)

CodeAnalyzer.java

```
    public int getWidestLineNumber () {
        return widestLineNumber;
    }

    public LineWidthHistogram getLineWidthHistogram () {
        return lineWidthHistogram;
    }

    public double getMeanLineWidth () {
```

```
    } return (doble) totalChars / lineCount;

    public int getMedianLineWidth () {
        Integer [] sortedWidths = getSortedWidths ();
        int cumulativeLineCount = 0;
        para (ancho int: sortedWidths) {
            cumulativeLineCount + = lineCountForWidth (ancho);
            si (acumulativoLineCount> lineCount / 2)
                ancho de retorno;
        }
        lanzar nuevo Error ("No se puede llegar aquí");
    }

    private int lineCountForWidth (int ancho) {
        return lineWidthHistogram.getLinesforWidth (ancho) .size ();
    }

    Entero privado [] getSortedWidths () {
        Establecer <Integer> anchos = lineWidthHistogram.getWidths ();
        Integer [] sortedWidths = (widths.toArray (new Integer [0]));
        Arrays.sort (sortedWidths);
        return sortedWidths;
    }
}
```


Página 124

6

Objetos y estructuras de datos

Hay una razón por la que mantenemos nuestras variables privadas. No queremos que nadie más dependa en ellos. Queremos mantener la libertad de cambiar su tipo o implementación por capricho. o un impulso. Entonces, ¿por qué tantos programadores agregan automáticamente getters y setters? a sus objetos, exponiendo sus variables privadas como si fueran públicas?

Abstracción de datos

Considere la diferencia entre el Listado 6-1 y el Listado 6-2. Ambos representan los datos de un punto en el plano cartesiano. Y sin embargo, uno expone su implementación y el otro com- lo oculta completamente.

Página 125

Listado 6-1

Punto de hormigón

```
Punto de clase pública {
    público doble x;
    público doble y;
}
```

Listado 6-2

Punto abstracto

```
Punto de interfaz pública {
    doble getX ();
    doble getY ();
    void setCartesian (doble x, doble y);
    doble getR ();
    doble getTheta ();
    void setPolar (doble r, doble theta);
}
```

Lo hermoso del Listado 6-2 es que no hay forma de saber si el la implementación es en coordenadas rectangulares o polares. ¡Puede que no sea ninguno de los dos! Y sin embargo el La interfaz todavía representa inequívocamente una estructura de datos.

Pero representa más que una simple estructura de datos. Los métodos imponen un acceso política. Puede leer las coordenadas individuales de forma independiente, pero debe establecer las coordenadas nates juntos como una operación atómica.

El listado 6-1, por otro lado, está muy claramente implementado en coordenadas rectangulares, y nos obliga a manipular esas coordenadas de forma independiente. Esto expone la implementación ción. De hecho, expondría la implementación incluso si las variables fueran privadas y estaban usando getters y setters de una sola variable.

Ocultar la implementación no es solo una cuestión de poner una capa de funciones entre las variables. ¡Ocultar la implementación se trata de abstracciones! Una clase no se limita a empujar sus variables a través de getters y setters. Más bien expone interfaces abstractas que permiten a sus usuarios manipular la *esencia* de los datos, sin tener que conocer su implementación.

Considere el Listado 6-3 y el Listado 6-4. El primero utiliza términos concretos para comunicarse. el nivel de combustible de un vehículo, mientras que el segundo lo hace con la abstracción de porcentaje.

En el caso concreto, puede estar bastante seguro de que estos son solo accesos de variables. En el caso abstracto, no tiene ni idea de la forma de los datos.

Listado 6-3

Vehículo de hormigón

```
Vehículo de interfaz pública {
   doble getFuelTankCapacityInGallons ();
   doble getGallonsOfGasoline ();
}
```


Listado 6-4

Vehículo abstracto

```
Vehículo de interfaz pública {
   doble getPercentFuelRemaining ();
}
```

En los dos casos anteriores, es preferible la segunda opción. No queremos exponer los detalles de nuestros datos. Más bien queremos expresar nuestros datos en términos abstractos. Esto no es simplemente se logra mediante el uso de interfaces y / o getters y setters. Necesidades serias de pensamiento para que se coloque de la mejor manera para representar los datos que contiene un objeto. La peor opcion es para agregar alegremente getters y setters.

Antisimetría de datos / objetos

Estos dos ejemplos muestran la diferencia entre objetos y estructuras de datos. Los objetos se esconden sus datos detrás de abstracciones y exponen funciones que operan en esos datos. Estructura de datos ture exponen sus datos y no tienen funciones significativas. Vuelve atrás y vuelve a leerlo. Note la naturaleza complementaria de las dos definiciones. Son opuestos virtuales. Esto La diferencia puede parecer trivial, pero tiene implicaciones de gran alcance.

Considere, por ejemplo, el ejemplo de forma de procedimiento en el Listado 6-5. La geometría La clase opera en las tres clases de formas. Las clases de formas son estructuras de datos simples sin ningún comportamiento. Todo el comportamiento está en la clase de geometría .

Listado 6-5

Forma procedimental

```
plaza de clase pública {
   public Point topLeft;
   doble cara pública;
}

Rectángulo de clase pública {
   public Point topLeft;
   público doble altura;
   doble ancho público;
}

Círculo de clase pública {
   centro de punto público;
   doble radio público;
}

geometría de clase pública {
   PI doble final público = 3,141592653589793;

   el área doble pública (forma de objeto) arroja NoSuchShapeException
   {
      if (instancia de forma de Square) {
         Cuadrado s = forma (cuadrada);
         volver al lado * al lado;
      }
```

Página 127

Listado 6-5 (continuación)

Forma procedimental

```
        else if (instancia de forma de Rectángulo) {
            Rectángulo r = (Rectángulo) forma;
            return r.height * r.width;
        }
        else if (forma instancia del círculo) {
            Círculo c = forma (círculo);
            return PI * c.radius * c.radius;
        }
        lanzar nueva NoSuchShapeException ();
    }
}
```

Los programadores orientados a objetos podrían arrugar sus narices ante esto y quejarse de que es de procedimiento, y estarían en lo cierto. Pero la burla puede no estar justificada. Considere lo que sucedería si se agregara una función perimeter () a Geometry . Las clases de formas no se vea afectado! ¡Cualquier otra clase que dependiera de las formas tampoco se vería afectada! Por otro lado, si agrego una nueva forma, debo cambiar todas las funciones en Geometría a tratar con él. De nuevo, vuelve a leerlo. Observe que las dos condiciones son diametralmente opuesto.

Ahora considere la solución orientada a objetos del Listado 6-6. Aquí el método area () es polimórfico. No es necesaria una clase de geometría . Entonces, si agrego una nueva forma, ninguna de las existentes *Las funciones* se ven afectadas, pero si agrego una nueva función, ¡todas las *formas* deben cambiarse! [1]

Listado 6-6

Formas polimórficas

```
clase pública Square implementa Shape {
    Private Point topLeft;
    lado doble privado;

    área doble pública () {
        lado de retorno * lado;
    }
}

Rectangle de clase pública implementa Shape {
    Private Point topLeft;
    doble altura privada;
    doble ancho privado;

    área doble pública () {
        retorno alto * ancho;
    }
}
```

1. Hay formas de evitar esto que son bien conocidas por los diseñadores experimentados orientados a objetos: V ISITOR , o envío dual, para ejemplo. Pero estas técnicas tienen sus propios costos y generalmente devuelven la estructura a la de un programa de procedimiento.

Página 128

Listado 6-6 (continuación)

Formas polimórficas

```
Circle de clase pública implementa Shape {
    centro de punto privado;
    radio doble privado;
    PI doble final público = 3,141592653589793;

    área doble pública () {
        return PI * radio * radio;
    }
}
```

Nuevamente, vemos la naturaleza complementaria de estas dos definiciones; son virtuales opuestos! Esto expone la dicotomía fundamental entre objetos y estructuras de datos:

> *El código de procedimiento (código que utiliza estructuras de datos) facilita la adición de nuevas funciones sin cambiar las estructuras de datos existentes. El código OO, por otro lado, facilita la adición nuevas clases sin cambiar las funciones existentes.*

El complemento también es cierto:

> *El código de procedimiento dificulta la adición de nuevas estructuras de datos porque todas las funciones deben cambio. El código OO dificulta la adición de nuevas funciones porque todas las clases deben cambiar.*

Entonces, las cosas que son difíciles para OO son fáciles para los procedimientos, y las cosas que son difíciles para los procedimientos son fáciles para OO!

En cualquier sistema complejo, habrá ocasiones en las que queramos agregar nuevos datos. tipos en lugar de nuevas funciones. Para estos casos, los objetos y OO son los más apropiados. En Por otro lado, también habrá ocasiones en las que querremos agregar nuevas funciones en lugar de a los tipos de datos. En ese caso, el código de procedimiento y las estructuras de datos serán más apropiados.

Los programadores maduros saben que la idea de que todo es un objeto *es un mito* . Algunos- tiempos que realmente *haces* quieren estructuras de datos simples con los procedimientos que operan en ellos.

La ley de Demeter

Existe una heurística bien conocida llamada *Ley de Demeter* [2] que dice que un módulo no debe conocer las entrañas de los *objetos* que manipula. Como vimos en la última sección, los objetos ocultar sus datos y exponer operaciones. Esto significa que un objeto no debe exponer su estructura interna a través de accesos porque hacerlo es exponer, en lugar de ocultar, su estructura interna.

Más precisamente, la Ley de Deméter dice que un método *f* de una clase *C* solo debería llamar los métodos de estos:

- *C*
- Un objeto creado por *f*

2. http://en.wikipedia.org/wiki/Law_of_Demete r

- Un objeto pasado como argumento *af*
- Un objeto contenido en una variable de instancia de *C*

El método *no* debe invocar métodos en objetos que sean devueltos por cualquiera de los

funciones permitidas. En otras palabras, hable con amigos, no con extraños.

El siguiente código [3] parece violar la Ley de Demeter (entre otras cosas) porque llama a la función getScratchDir () en el valor de retorno de getOptions () y luego llama a getAbsolutePath () en el valor de retorno de getScratchDir () .

```
cadena final outputDir = ctxt.getOptions (). getScratchDir (). getAbsolutePath ();
```

Ruinas del tren

Este tipo de código a menudo se denomina *accidente de tren* porque parece un montón de trenes acoplados carros. Las cadenas de llamadas como esta generalmente se consideran de estilo descuidado y deben ser evitado [G36]. Por lo general, es mejor dividirlos de la siguiente manera:

```
Opciones opts = ctxt.getOptions ();
Archivo scratchDir = opts.getScratchDir ();
cadena final outputDir = scratchDir.getAbsolutePath ();
```

¿Son estos dos fragmentos de código violatorio? ciones de la ley de Deméter? Definitivamente el módulo contenedor sabe que el El objeto ctxt contiene opciones, que mantener un directorio borrador, que tiene un camino absoluto. Eso es mucho conocimiento para que una función lo sepa. La llamada la función sabe cómo navegar muchos objetos diferentes.

Si esto es una violación de Demeter depende de si ctxt , Options y ScratchDir son objetos o estructuras de datos. Si son objetos, entonces su estructura interna deben ocultarse en lugar de exponerse, por lo que el conocimiento de sus entrañas es una clara violación ción de la Ley de Deméter. Por otro lado, si ctxt , Options y ScratchDir son solo estructuras de datos sin comportamiento, entonces naturalmente exponen su estructura interna, y así Demeter no se aplica.

El uso de funciones de acceso confunde el problema. Si el código se hubiera escrito de la siguiente manera mínimos, entonces probablemente no estaríamos preguntando acerca de las violaciones de Demeter.

```
cadena final outputDir = ctxt.options.scratchDir.absolutePath;
```

Este problema sería mucho menos confuso si las estructuras de datos simplemente tuvieran variables públicas y ninguna función, mientras que los objetos tenían variables privadas y funciones públicas. Sin emabargo,

3. Encontrado en algún lugar del marco de apache.

Página 130

hay marcos y estándares (por ejemplo, "beans") que exigen que incluso los datos simples las estructuras tienen accesores y mutadores.

Híbridos

Esta confusión a veces conduce a estructuras híbridas desafortunadas que son mitad objeto y la mitad de la estructura de datos. Tienen funciones que hacen cosas importantes y también tienen variables públicas o accesores y mutadores públicos que, a todos los efectos, hacen las variables privadas públicas, tentando a otras funciones externas a utilizar esas variables forma en que un programa de procedimiento utilizaría una estructura de datos. [4]

Estos híbridos dificultan la incorporación de nuevas funciones, pero también dificultan la incorporación de nuevos datos. estructuras. Son lo peor de ambos mundos. Evite crearlos. Son indicativos de una diseño confuso cuyos autores no están seguros, o peor aún, ignoran, si necesitan protección de funciones o tipos.

Ocultar estructura

¿Qué pasa si ctxt , options y scratchDir son objetos con comportamiento real? Entonces, porque Se supone que los objetos ocultan su estructura interna, no deberíamos poder navegar a través de ellos. Entonces, ¿cómo obtendríamos la ruta absoluta del directorio temporal?

```
ctxt.getAbsolutePathOfScratchDirectoryOption ();
```

o

```
ctx.getScratchDirectoryOption (). getAbsolutePath ()
```

La primera opción podría conducir a una explosión de métodos en el objeto ctxt. El segundo pre supone que getScratchDirectoryOption () devuelve una estructura de datos, no un objeto. Ninguno de los dos la opción se siente bien.

Si ctxt es un objeto, deberíamos decirle que *haga algo;* no deberíamos estar preguntando sobre sus partes internas. Entonces, ¿por qué queríamos la ruta absoluta del directorio temporal? Qué ¿íbamos a hacer con eso? Considere este código de (muchas líneas más abajo en) el mismo módulo:

```
String outFile = outputDir + "/" + className.replace ('.', '/') + ".Class";
FileOutputStream fout = nuevo FileOutputStream (outFile);
BufferedOutputStream bos = nuevo BufferedOutputStream (fout);
```

La mezcla de diferentes niveles de detalle [G34] [G6] es un poco preocupante. Puntos, barras, extensiones de archivo, y los objetos de archivo no deben mezclarse tan descuidadamente, y mezclarse con el código adjunto. Sin embargo, ignorando eso, vemos que la intención de obtener la absoluta la ruta de acceso del directorio temporal era crear un archivo temporal con un nombre determinado.

4. Esto a veces se denomina Feature Envy de [Refactorización].

Página 131

Entonces, ¿qué pasa si le decimos al objeto ctxt que haga esto?

```
BufferedOutputStream bos = ctxt.createScratchFileStream (classFileName);
```

¡Eso parece algo razonable para que lo haga un objeto! Esto permite que ctxt oculte su internos y evita que la función actual tenga que violar la Ley de Deméter al navegando a través de objetos que no debería conocer.

Objetos de transferencia de datos

La forma por excelencia de una estructura de datos es una clase con variables públicas y sin función ciones. Esto a veces se denomina objeto de transferencia de datos o DTO. Los DTO son estructuras muy útiles tures, especialmente al comunicarse con bases de datos o analizar mensajes de sockets, y así. A menudo se convierten en las primeras de una serie de etapas de traducción que convierten datos sin procesar en una base de datos en objetos en el código de la aplicación.

Algo más común es la forma "frijol" que se muestra en el Listado 6-7. Los frijoles tienen privado variables manipuladas por getters y setters. La cuasi-encapsulación de frijoles parece hacen que algunos puristas de OO se sientan mejor, pero por lo general no proporciona ningún otro beneficio.

Listado 6-7
address.java

```
Dirección de clase pública {
  calle privada de String;
  private String streetExtra;
  ciudad de cadena privada;
  estado de cadena privada;
  cremallera de cadena privada;

  Dirección pública (String street, String streetExtra,
                     Cadena de ciudad, estado de cadena, código postal de cadena) {
    this.street = calle;
```

```
        this.streetExtra = streetExtra;
        this.city = ciudad;
        this.state = estado;
        this.zip = zip;
    }

    public String getStreet () {
        calle de regreso;
    }

    public String getStreetExtra () {
        return streetExtra;
    }

    public String getCity () {
        ciudad de regreso;
    }
```


Página 132

Listado 6-7 (continuación)

address.java

```
    public String getState () {
        estado de retorno;
    }

    public String getZip () {
        cremallera de retorno;
    }
}
```

Registro activo

Los registros activos son formas especiales de DTO. Son estructuras de datos con público (o bean-accedido) variables; pero normalmente tienen métodos de navegación como guardar y buscar . Typi- básicamente, estos registros activos son traducciones directas de tablas de bases de datos u otros datos fuentes.

Desafortunadamente, a menudo encontramos que los desarrolladores intentan tratar estas estructuras de datos como si eran objetos al poner en ellos métodos de reglas comerciales. Esto es incómodo porque crea un híbrido entre una estructura de datos y un objeto.

La solución, por supuesto, es tratar el registro activo como una estructura de datos y crear objetos separados que contienen las reglas de negocio y que ocultan sus datos internos (que son probablemente solo instancias del registro activo).

Conclusión

Los objetos exponen el comportamiento y ocultan datos. Esto facilita la adición de nuevos tipos de objetos. sin cambiar los comportamientos existentes. También dificulta la incorporación de nuevos comportamientos a los objetos. Las estructuras de datos exponen datos y no tienen un comportamiento significativo. Esto hace que sea fácil agregar nuevos comportamientos a las estructuras de datos existentes, pero dificulta la adición de nuevas estructuras de datos a las funciones existentes.

En cualquier sistema dado, a veces querremos la flexibilidad para agregar nuevos tipos de datos, y por lo que preferimos objetos para esa parte del sistema. Otras veces querremos la flexibilidad para agregar nuevos comportamientos, por lo que en esa parte del sistema preferimos tipos de datos y procedimientos. Los buenos desarrolladores de software comprenden estos problemas sin prejuicios y eligen el enfoque que es mejor para el trabajo en cuestión.

Bibliografía

[Refactorización]: *Refactorización: Mejora del diseño del código existente* , Martin Fowler et al., Addison-Wesley, 1999.

Página 133

Esta página se dejó en blanco intencionalmente

Página 134

7

Manejo de errores

por Michael Feathers

Puede parecer extraño tener una sección sobre manejo de errores en un libro sobre código limpio. Error el manejo es solo una de esas cosas que todos tenemos que hacer cuando programamos. La entrada puede ser anormal y los dispositivos pueden fallar. En resumen, las cosas pueden salir mal y, cuando suceden, nosotros, como Los grammers son responsables de asegurarse de que nuestro código haga lo que debe hacer.

Sin embargo, la conexión con el código limpio debería ser clara. Muchas bases de código se componen completamente dominado por el manejo de errores. Cuando digo dominado, no me refiero a ese error en el manejo dling es todo lo que hacen. Quiero decir que es casi imposible ver lo que hace el código. debido a todo el manejo de errores dispersos. El manejo de errores es importante, *pero si oscurece la lógica, está mal* .

En este capítulo describiré una serie de técnicas y consideraciones que puede utilizar escribir código limpio y robusto, código que maneja los errores con elegancia y estilo.

Página 135

Utilice excepciones en lugar de códigos de devolución

En el pasado distante había muchos idiomas que no tenían excepciones. En esos idiomas las técnicas para manejar y reportar errores eran limitadas. O estableces un marca de error o devolvió un código de error que la persona que llama pudo verificar. El código del Listado 7-1 ilustra estos enfoques.

Listado 7-1

DeviceController.java

```
DeviceController de clase pública {
    ...
    public void sendShutDown () {
        DeviceHandle handle = getHandle (DEV1);
        // Verifica el estado del dispositivo
        if (handle! = DeviceHandle.INVALID) {
            // Guarde el estado del dispositivo en el campo de registro
            retrieveDeviceRecord (identificador);
            // Si no está suspendido, apague
            if (record.getStatus ()! = DEVICE_SUSPENDED) {
                pauseDevice (manejar);
                clearDeviceWorkQueue (identificador);
                closeDevice (manejador);
            } demás {
                logger.log ("Dispositivo suspendido. No se puede apagar");
            }
        } demás {
            logger.log ("Identificador no válido para:" + DEV1.toString ());
        }
    }
    ...
}
```

El problema con estos enfoques es que abarrotan a la persona que llama. La persona que llama debe compruebe si hay errores inmediatamente después de la llamada. Desafortunadamente, es fácil de olvidar. Por esta razón hijo, es mejor lanzar una excepción cuando encuentre un error. El código de llamada es limpiador. Su lógica no se ve oscurecida por el manejo de errores.

El Listado 7-2 muestra el código después de haber elegido lanzar excepciones en métodos que puede detectar errores.

Listado 7-2

DeviceController.java (con excepciones)

```
DeviceController de clase pública {
    ...

    public void sendShutDown () {
        intentar {
            tryToShutDown ();
        } captura (DeviceShutDownError e) {
            logger.log (e);
        }
    }
```


Página 136

Listado 7-2 (continuación)

DeviceController.java (con excepciones)

```
    private void tryToShutDown () arroja DeviceShutDownError {
        DeviceHandle handle = getHandle (DEV1);
        Registro de DeviceRecord = retrieveDeviceRecord (identificador);

        pauseDevice (manejar);
        clearDeviceWorkQueue (identificador);
        closeDevice (manejador);
    }

    Private DeviceHandle getHandle (Id. de DeviceID) {
        ...
```

```
            lanzar nuevo DeviceShutDownError ("Identificador no válido para:" + id.toString ());
            ...
        }

        ...
    }
```

Fíjate cuánto más limpio está. Esto no es solo una cuestión de estética. El codigo es mejor
porque dos preocupaciones que estaban enredadas, el algoritmo para el apagado del dispositivo y el manejo de errores
dling, ahora están separados. Puede analizar cada una de esas inquietudes y comprenderlas.
independientemente.

Escriba su declaración Try-Catch-Finalmente primero

Una de las cosas más interesantes sobre las excepciones es que *definen un alcance* dentro de su
programa. Al ejecutar código en el intento porción de un try - catch - por último estado de cuenta,
están indicando que la ejecución puede abortar en cualquier momento y luego reanudarse en la captura .

En cierto modo, los bloques de prueba son como transacciones. Tu captura tiene que salir de tu programa en un
estado consistente, pase lo que pase en el intento . Por esta razón, es una buena práctica
Comience con una declaración try - catch - finalmente cuando esté escribiendo un código que podría arrojar
excepciones. Esto le ayuda a definir qué debe esperar el usuario de ese código, sin importar qué
va mal con el código que se ejecuta en el intento .

Veamos un ejemplo. Necesitamos escribir un código que acceda a un archivo y lea
algunos objetos serializados.

Comenzamos con una prueba unitaria que muestra que obtendremos una excepción cuando el archivo no existe:

```
@Test (esperado = StorageException.class)
public void retrieveSectionShouldThrowOnInvalidFileName () {
    sectionStore.retrieveSection ("archivo inválido");
}
```

La prueba nos impulsa a crear este código auxiliar:

```
public List <RecordedGrip> retrieveSection (String sectionName) {
    // retorno ficticio hasta que tengamos una implementación real
    return new ArrayList <RecordedGrip> ();
}
```


Página 137

Nuestra prueba falla porque no lanza una excepción. A continuación, cambiamos nuestra implementación
ción para que intente acceder a un archivo no válido. Esta operación arroja una excepción:

```
public List <RecordedGrip> retrieveSection (String sectionName) {
    intentar {
        Flujo de FileInputStream = nuevo FileInputStream (nombre de sección)
    } captura (Excepción e) {
        lanzar nueva StorageException ("error de recuperación", e);
    }
    return new ArrayList <RecordedGrip> ();
}
```

Nuestra prueba pasa ahora porque hemos detectado la excepción. En este punto, podemos refaccionar
colina. Podemos reducir el tipo de excepción que detectamos para que coincida con el tipo que es realmente
lanzada desde el FileInputStream constructor: FileNotFoundException :

```
public List <RecordedGrip> retrieveSection (String sectionName) {
    intentar {
        Flujo de FileInputStream = nuevo FileInputStream (nombre de sección);
        stream.close ();
    } captura (FileNotFoundException e) {
        lanzar una nueva StorageException ("error de recuperación", e);
    }
    return new ArrayList <RecordedGrip> ();
}
```

Ahora que hemos definido el alcance con una estructura try - catch , podemos usar TDD para construir
el resto de la lógica que necesitamos. Esa lógica se agregará entre la creación del
FileInputStream y el cierre , y puede fingir que nada sale mal.

Intente escribir pruebas que fuercen excepciones y luego agregue comportamiento a su controlador para isfy tus pruebas. Esto hará que primero cree el alcance de la transacción del bloque try y le ayudará a mantener la naturaleza de transacción de ese alcance.

Usar excepciones sin marcar

Se acabó el debate. Durante años, los programadores de Java han debatido sobre los beneficios y las responsabilidades vínculos de excepciones comprobadas. Cuando se marcó, se introdujeron excepciones en la primera versión de Java, parecían una gran idea. La firma de cada método enumeraría todos los excepciones que podría pasar a su llamador. Además, estas excepciones eran parte del tipo del método. Su código, literalmente, no se compilaría si la firma no coincidiera con su el código podría hacer.

En ese momento, pensamos que las excepciones marcadas eran una gran idea; y si pueden producir *algún* beneficio. Sin embargo, ahora está claro que no son necesarios para la producción de software robusto. C # no tiene excepciones comprobadas y, a pesar de los valientes intentos, C ++ tampoco. Tampoco Python o Ruby. Sin embargo, es posible escribir software robusto en todos de estos idiomas. Debido a que ese es el caso, tenemos que decidir, realmente, si se verificó las excepciones valen su precio.

Página 138

¿Qué precio? El precio de las excepciones marcadas es una violación del Principio 1 de Abierto / Cerrado . Si lanza una excepción marcada de un método en su código y la captura es de tres niveles anterior, *debe declarar esa excepción en la firma de cada método entre usted y la captura* . Esto significa que un cambio en un nivel bajo del software puede forzar la firma. cambios en muchos niveles superiores. Los módulos modificados deben reconstruirse y volver a implementarse, a pesar de que nada de lo que les importa cambió.

Considere la jerarquía de llamadas de un sistema grande. Funciones en las funciones de llamada superior debajo de ellos, que llaman a más funciones debajo de ellos, ad infinitum. Ahora digamos uno de los Las funciones de nivel más bajo se modifican de tal manera que deben lanzar una excepción. Si eso La excepción está marcada, entonces la firma de la función debe agregar una cláusula throws . Pero esto significa que cada función que llama a nuestra función modificada también debe modificarse para capturar la nueva excepción o agregar la cláusula throws apropiada a su firma. Anuncio infinitum. El resultado neto es una cascada de cambios que se abren camino desde los niveles más bajos. del software al más alto! La encapsulación está rota porque todas las funciones en la ruta de un lanzamiento debe conocer los detalles de esa excepción de bajo nivel. Dado que el propósito de excepciones es permitirle manejar errores a distancia, es una pena que marcó excep- de esta forma rompen la encapsulación.

Las excepciones marcadas a veces pueden ser útiles si está escribiendo una biblioteca crítica: debe atraparlos. Pero en el desarrollo de aplicaciones en general, los costos de dependencia superan los beneficios.

Proporcionar contexto con excepciones

Cada excepción que arroje debe proporcionar suficiente contexto para determinar la fuente y ubicación de un error. En Java, puede obtener un seguimiento de la pila de cualquier excepción; sin embargo, una pila trace no puede decirle la intención de la operación que falló.

Cree mensajes de error informativos y páselos junto con sus excepciones. Hombres- mencione la operación que falló y el tipo de falla. Si está iniciando sesión en su aplicación, Transmita suficiente información para poder registrar el error en su captura .

Definir clases de excepción en función de las necesidades de la persona que llama

Hay muchas formas de clasificar los errores. Podemos clasificarlos por su fuente: vienen de un componente u otro? O su tipo: ¿Son fallas de dispositivos, redes fallas o errores de programación? Sin embargo, cuando definimos clases de excepción en una aplicación cación, nuestra preocupación más importante debería ser *cómo se capturan* .

1. [Martín].

Página 139

Veamos un ejemplo de clasificación de excepción deficiente. Aquí hay un intento - atrapar - finalmente declaración para una llamada de biblioteca de terceros. Cubre todas las excepciones que las llamadas pueden lanzar:

```
Puerto ACMEPort = nuevo ACMEPort (12);

intentar {
    port.open ();
} captura (DeviceResponseException e) {
    reportPortError (e);
    logger.log ("Excepción de respuesta del dispositivo", e);
} catch (ATM1212UnlockedException e) {
    reportPortError (e);
    logger.log ("Excepción de desbloqueo", e);
} captura (GMXError e) {
    reportPortError (e);
    logger.log ("Excepción de respuesta del dispositivo");
} finalmente {
    ...
}
```

Esa declaración contiene mucha duplicación y no debería sorprendernos. En la mayoría situaciones de manejo de excepciones, el trabajo que hacemos es relativamente estándar independientemente de la causa real. Tenemos que registrar un error y asegurarnos de que podemos continuar.

En este caso, porque sabemos que el trabajo que estamos haciendo es aproximadamente el mismo independientemente de la excepción, podemos simplificar nuestro código considerablemente envolviendo la API que estamos llamando y asegurándonos de que devuelva un tipo de excepción común:

```
Puerto LocalPort = nuevo LocalPort (12);
intentar {
    port.open ();
} captura (PortDeviceFailure e) {
    reportError (e);
    logger.log (e.getMessage (), e);
} finalmente {
    ...
}
```

Nuestra clase LocalPort es solo un contenedor simple que detecta y traduce excepciones arrojado por la clase ACMEPort :

```
Public class LocalPort {
  private ACMEPort innerPort;

  public LocalPort (int portNumber) {
    innerPort = new ACMEPort (portNumber);
  }

  public void open () {
    intentar {
      innerPort.open ();
    } captura (DeviceResponseException e) {
      lanzar nuevo PortDeviceFailure (e);
    } catch (ATM1212UnlockedException e) {
      lanzar nuevo PortDeviceFailure (e);
    } captura (GMXError e) {
```

```
                lanzar nuevo PortDeviceFailure (e);
            }
        }
        ...
    }
```

Wrappers como el que definimos para ACMEPort pueden ser muy útiles. De hecho, envolviendo API de terceros es una práctica recomendada. Cuando envuelve una API de terceros, minimiza su dependencias de ella: puede elegir mudarse a una biblioteca diferente en el futuro sin mucha pena. La envoltura también hace que sea más fácil simular llamadas de terceros cuando está probando su propio código.

Una última ventaja de la envoltura es que no está atado a la API de un proveedor en particular. opciones de diseño. Puede definir una API con la que se sienta cómodo. En el anterior Por ejemplo, definimos un solo tipo de excepción para la falla del dispositivo de puerto y descubrimos que podría escribir código mucho más limpio.

A menudo, una sola clase de excepción está bien para un área de código en particular. La información enviado con la excepción puede distinguir los errores. Use diferentes clases solo si hay momentos en los que desea capturar una excepción y permitir que la otra pase.

Definir el flujo normal

Si sigue los consejos del anterior secciones, terminarás con una buena cantidad de separación entre su lógica empresarial y su manejo de errores. La mayor parte de tu el código comenzará a verse como limpio algoritmo sin adornos. Sin embargo, el pro- El cese de hacer esto impulsa la detección de errores a los bordes de su programa. Tu envuelves API externas para que pueda lanzar su propias excepciones, y define un controlador por encima de su código para que pueda manejar cualquier cálculo abortado. La mayoría de las veces, este es un gran enfoque, pero hay ocasiones cuando no quieras abortar.

Echemos un vistazo a un ejemplo. Aquí hay un código incómodo que suma los gastos en un solicitud de facturación:

```
intentar {
    Gastos de gastos de comida = gastosReportDAO.getMeals (employee.getID ());
    m_total + = gastos.getTotal ();
} captura (MealExpensesNotFound e) {
    m_total + = getMealPerDiem ();
}
```

En este negocio, si las comidas se contabilizan como gastos, pasan a formar parte del total. Si no lo son, el el empleado recibe una comida *diaria* por ese día. La excepción satura la lógica. ¿No sería mejor si no tuviéramos que ocuparnos del caso especial? Si no lo hicimos, nuestro código parecería mucho más simple. Se vería así:

```
Gastos de gastos de comida = gastosReportDAO.getMeals (employee.getID ());
m_total + = gastos.getTotal ();
```

¿Podemos hacer que el código sea así de simple? Resulta que podemos. Podemos cambiar el ExpenseReportDAO para que siempre devuelva un objeto MealExpense . Si no hay comida gastos, devuelve un objeto MealExpense que devuelve los *viáticos* como su total:

```
PerDiemMealExpenses de clase pública implementa MealExpenses {
    public int getTotal () {
        // devuelve los viáticos predeterminados
    }
}
```

Esto se llama el S PECIAL C ASE P attern [Fowler]. Creas una clase o configuras una objeto para que maneje un caso especial para usted. Cuando lo hace, el código de cliente no tiene para hacer frente a un comportamiento excepcional. Ese comportamiento está encapsulado en el objeto de caso especial.

No devuelva nulo

Creo que cualquier discusión sobre el manejo de errores debería incluir una mención de las cosas que haz eso invitar a errores. El primero de la lista devuelve nulo . No puedo empezar a contar el número de las aplicaciones que he visto en las que casi todas las demás líneas eran un control de nulo . Aquí está algún código de ejemplo:

```
public void registerItem (artículo artículo) {
    if (item! = null) {
        ItemRegistry registro = peristentStore.getItemRegistry ();
        if (registro! = nulo) {
            Elemento existente = registro.getItem (elemento.getID ());
            if (existing.getBillingPeriod (). hasRetailOwner ()) {
                existente.registro (elemento);
            }
        }
    }
}
```

Si trabaja en una base de código con un código como este, puede que no le parezca tan malo, pero es ¡malo! Cuando devolvemos nulo , esencialmente estamos creando trabajo para nosotros mismos e imponiendo problemas a nuestras personas que llaman. Todo lo que se necesita es un cheque nulo faltante para enviar una solicitud girando fuera de control.

¿Notó el hecho de que no había un cheque nulo en la segunda línea de ese anidado? si declaración? ¿Qué hubiera pasado en tiempo de ejecución si persistentStore fuera nulo ? Nosotros habría tenido una NullPointerException en tiempo de ejecución, y alguien está detectando NullPointerException en el nivel superior o no lo son. De cualquier manera es *malo* . Qué exactamente debe hacer en respuesta a una NullPointerException lanzada desde las profundidades de su aplicación ¿catión?

Es fácil decir que el problema con el código anterior es que le falta una verificación nula , pero en realidad, el problema es que tiene *demasiados* . Si tiene la tentación de devolver un valor nulo de un método, considere lanzar una excepción o devolver un objeto C ASE ESPECIAL en su lugar. Si está llamando a un método de retorno nulo desde una API de terceros, considere envolver eso método con un método que arroja una excepción o devuelve un objeto de caso especial.

En muchos casos, los objetos de casos especiales son un remedio fácil. Imagina que tienes un código como esto:

```
Lista <Empleado> empleados = getEmployees ();
if (empleados! = nulo) {
    para (Empleado e: empleados) {
        totalPay + = e.getPay ();
```

```
    }
}
```

En este momento, getEmployees puede devolver un valor nulo , pero ¿tiene que hacerlo? Si cambiamos getEmployee así que devuelve una lista vacía, podemos limpiar el código:

```
Lista <Empleado> empleados = getEmployees ();
para (Empleado e: empleados) {
    totalPay + = e.getPay ();
}
```

Afortunadamente, Java tiene Collections.emptyList () y devuelve una lista inmutable predefinida que podemos usar para este propósito:

```
public List <Employee> getEmployees () {
    si (.. no hay empleados ..)
        return Collections.emptyList ();
}
```

Si codifica de esta manera, minimizará la posibilidad de NullPointerExceptions y su el código será más limpio.

No pase nulo

Devolver nulos de los métodos es malo, pero pasar nulos a los métodos es peor. A menos que usted está trabajando con una API que espera que pase nulo , debe evitar pasar nulo en su código siempre que sea posible.

Veamos un ejemplo para ver por qué. Aquí hay un método simple que calcula una ric por dos puntos:

```
MetricsCalculator de clase pública
{
    public double xProjection (Punto p1, Punto p2) {
        return (p2.x - p1.x) * 1.5;
    }
    ...
}
```

¿Qué sucede cuando alguien pasa nulo como argumento?

```
calculator.xProjection (nulo, nuevo Punto (12, 13));
```

Obtendremos una NullPointerException , por supuesto.

¿Cómo podemos solucionarlo? Podríamos crear un nuevo tipo de excepción y lanzarlo:

```
MetricsCalculator de clase pública
{
```


Página 143


```
    public double xProjection (Punto p1, Punto p2) {
        si (p1 == nulo || p2 == nulo) {
            lanzar InvalidArgumentException (
                "Argumento no válido para MetricsCalculator.xProjection");
        }
        return (p2.x - p1.x) * 1.5;
    }
}
```

¿Es esto mejor? Puede ser un poco mejor que una excepción de puntero nulo , pero recuerde, tiene que definir un controlador para InvalidArgumentException . ¿Qué debe hacer el manejador? Es ¿Hay algún buen curso de acción?

Existe otra alternativa. Podríamos usar un conjunto de afirmaciones:

```
MetricsCalculator de clase pública
{
    public double xProjection (Punto p1, Punto p2) {
        afirmar p1! = null: "p1 no debe ser nulo";
        afirmar p2! = null: "p2 no debe ser nulo";
        return (p2.x - p1.x) * 1.5;
    }
}
```

Es una buena documentación, pero no resuelve el problema. Si alguien pasa nulo , lo haremos todavía tengo un error de tiempo de ejecución.

En la mayoría de los lenguajes de programación no existe una buena forma de lidiar con un nulo que es pasado por una persona que llama accidentalmente. Debido a que este es el caso, el enfoque racional es prohibir pasando nulo por defecto. Cuando lo haga, puede codificar sabiendo que un nulo en un La lista de argumentos es una indicación de un problema y terminan con muchos menos errores por descuido.

Conclusión

El código limpio es legible, pero también debe ser robusto. Estos no son objetivos contradictorios. Podemos escribir código limpio robusto si vemos el manejo de errores como una preocupación separada, algo que es visible independientemente de nuestra lógica principal. En la medida en que podamos hacer eso, podremos razonar sobre ello de forma independiente, y podemos hacer grandes avances en el mantenimiento de nuestro código.

Bibliografía

[Martin]: *Desarrollo de software ágil: principios, patrones y prácticas,* Robert C. Martin, Prentice Hall, 2002.

8

Límites

por James Grenning

Rara vez controlamos todo el software de nuestros sistemas. A veces compramos paquetes de terceros envejece o usa código abierto. Otras veces dependemos de equipos de nuestra propia empresa para producir componentes o subsistemas para nosotros. De alguna manera debemos integrar limpiamente este código externo

Página 145

con los nuestros. En este capítulo examinamos prácticas y técnicas para mantener los límites de nuestro software limpio.

Usar código de terceros

Existe una tensión natural entre el proveedor de una interfaz y el usuario de una interfaz. Los proveedores de paquetes y marcos de terceros se esfuerzan por lograr una amplia aplicabilidad para que Puede funcionar en muchos entornos y atraer a una amplia audiencia. Los usuarios, por otro lado, quieren una interfaz que se centre en sus necesidades particulares. Esta tensión puede causar problemas. en los límites de nuestros sistemas.

Veamos java.util.Map como ejemplo. Como puede ver al examinar la Figura 8-1, Los mapas tienen una interfaz muy amplia con muchas capacidades. Ciertamente este poder y flexibilidad La bilidad es útil, pero también puede ser una carga. Por ejemplo, nuestra aplicación puede generar un Mapa y páselo. Nuestra intención podría ser que ninguno de los destinatarios de nuestro mapa elimine cualquier cosa en el mapa. Pero allí mismo, en la parte superior de la lista, se encuentra el método clear () . Cualquier usuario de el mapa tiene el poder de borrarlo. O tal vez nuestra convención de diseño es que solo en particular Los tipos de objetos se pueden almacenar en el mapa , pero los mapas no restringen de manera confiable los tipos de objetos colocados dentro de ellos. Cualquier usuario determinado puede agregar elementos de cualquier tipo a cualquier mapa .

- clear () void - Mapa
- containsKey (clave de objeto) booleano - Mapa
- containsValue (valor del objeto) booleano - Mapa
- entrySet () Set - Mapa
- es igual a (Objeto o) booleano - Mapa
- obtener (clave de objeto) Objeto - Mapa
- getClass () Class <? extiende Objeto> - Objeto
- hashCode () int - Mapa
- isEmpty () booleano - Mapa
- keySet () Set - Mapa
- notificar () void - Objeto
- notifyAll () void - Objeto
- put (clave de objeto, valor de objeto) Objeto - Mapa
- putAll (Map t) void - Mapa
- eliminar (clave de objeto) Objeto - Mapa
- size () int - Mapa
- toString () String - Objeto
- Colección de valores () - Mapa
- wait () void - Objeto
- esperar (tiempo de espera prolongado) void - Objeto
- esperar (tiempo de espera largo, nanos int) void - Objeto

Figura 8-1
Los métodos de Map

Si nuestra aplicación necesita un mapa de sensores , puede encontrar los sensores configurados así:

```
Sensores de mapa = nuevo HashMap ();
```


Página 146

Luego, cuando alguna otra parte del código necesita acceder al sensor, verá este código:

```
Sensor s = (Sensor) sensores.get (sensorId);
```

No solo lo vemos una vez, sino una y otra vez a lo largo del código. El cliente de este
El código tiene la responsabilidad de obtener un Objeto del Mapa y lanzarlo a la derecha.
tipo. Esto funciona, pero no es un código limpio. Además, este código no cuenta su historia tan bien como
podría. La legibilidad de este código se puede mejorar en gran medida mediante el uso de genéricos, como se muestra
debajo:

```
Map <Sensor> sensores = nuevo HashMap <Sensor> ();
...
Sensor s = sensores.get (sensorId);
```

Sin embargo, esto no resuelve el problema de que Map <Sensor> proporciona más capacidad de la que
necesidad o deseo.

Pasar una instancia de Map <Sensor> generosamente por el sistema significa que habrá
Hay muchos lugares para corregir si la interfaz de Map alguna vez cambia. Podrías pensar en tal cambio
Es poco probable, pero recuerde que cambió cuando se agregó la compatibilidad con genéricos en Java 5.
De hecho, hemos visto sistemas que no pueden usar genéricos debido a la pura
magnitud de los cambios necesarios para compensar el uso liberal de Map s.

Una forma más limpia de usar Map podría verse como la siguiente. A ningún usuario de Sensores le importaría
un poco si se usaron genéricos o no. Esa elección se ha convertido (y siempre debería ser) una
detalle de implementación.

```
Sensores de clase pública {
    sensores de mapas privados = nuevo HashMap ();

    public Sensor getById (String id) {
        return (Sensor) sensors.get (id);
    }

    //recorte
}
```

La interfaz en el límite (mapa) está oculta. Es capaz de evolucionar con muy poco impacto en
el resto de la aplicación. El uso de genéricos ya no es un gran problema porque el casting
y la gestión de tipos se maneja dentro de la clase Sensores .

Esta interfaz también está diseñada y restringida para satisfacer las necesidades de la aplicación. Eso
da como resultado un código que es más fácil de entender y más difícil de usar incorrectamente. La clase Sensores puede
hacer cumplir el diseño y las reglas comerciales.

No estamos sugiriendo que todos los usos de Map se encapsulen de esta forma. Más bien, nosotros
le están aconsejando que no pase Map s (o cualquier otra interfaz en un límite) alrededor de su
sistema. Si usa una interfaz de límites como Map , manténgala dentro de la clase o de la familia cercana
de clases, donde se usa. Evite devolverlo o aceptarlo como argumento para
API públicas.

Página 147

Explorando y aprendiendo límites

El código de terceros nos ayuda a ofrecer más funciones en menos tiempo. Donde empezamos cuando queremos utilizar algún paquete de terceros? No es nuestro trabajo probar el tercero código, pero puede ser de nuestro mejor interés escribir pruebas para el código de terceros que usamos.

Supongamos que no está claro cómo utilizar nuestra biblioteca de terceros. Podríamos pasar uno o dos días (o más) leyendo la documentación y decidiendo cómo la vamos a utilizar. Entonces nosotros Podría escribir nuestro código para usar el código de terceros y ver si hace lo que pensamos. Nosotros No nos sorprendería encontrarnos atascados en largas sesiones de depuración tratando de averigüe si los errores que estamos experimentando están en nuestro código o en el de ellos.

Aprender el código de terceros es difícil. La integración del código de terceros también es difícil. Hacer ambas cosas al mismo tiempo es doblemente difícil. ¿Y si adoptamos un enfoque diferente? En lugar de de experimentar y probar las cosas nuevas en nuestro código de producción, podríamos escribir algunas pruebas para explorar nuestra comprensión del código de terceros. Jim Newkirk llama a tales pruebas *pruebas de aprendizaje.* [1]

En las pruebas de aprendizaje, llamamos a la API de terceros, ya que esperamos usarla en nuestra aplicación. Básicamente, estamos haciendo experimentos controlados que comprueban nuestra comprensión de esa API. Las pruebas se centran en lo que queremos de la API.

Aprendizaje log4j

Digamos que queremos usar el paquete apache log4j en lugar de nuestro propio registro personalizado. ger. Lo descargamos y abrimos la página de documentación introductoria. Sin demasiado leyendo escribimos nuestro primer caso de prueba, esperando que escriba "hola" en la consola.

```
@Prueba
public void testLogCreate () {
    Logger logger = Logger.getLogger ("MyLogger");
    logger.info ("hola");
}
```

Cuando lo ejecutamos, el registrador produce un error que nos dice que necesitamos algo llamado Appender . Después de leer un poco más, encontramos que hay un ConsoleAppender . Entonces creamos un ConsoleAppender y ver si hemos desbloqueado los secretos del inicio de sesión en la consola.

```
@Prueba
public void testLogAddAppender () {
    Logger logger = Logger.getLogger ("MyLogger");
    ConsoleAppender appender = new ConsoleAppender ();
    logger.addAppender (appender);
    logger.info ("hola");
}
```

1. [BeckTDD], págs. 136-137.

Página 148

Esta vez nos encontramos con que el Appender no tiene flujo de salida. Extraño, parece lógico que Toma uno. Después de un poco de ayuda de Google, probamos lo siguiente:

```
@Prueba
public void testLogAddAppender () {
    Logger logger = Logger.getLogger ("MyLogger");
    logger.removeAllAppenders ();
    logger.addAppender (nuevo ConsoleAppender (
            nuevo PatternLayout ("% p% t% m% n"),
            ConsoleAppender.SYSTEM_OUT));
    logger.info ("hola");
}
```

Eso funciono; ¡Un mensaje de registro que incluye "hola" apareció en la consola! Parece extraño que tenemos que decirle al ConsoleAppender que escribe en la consola.

Curiosamente, cuando eliminamos el argumento ConsoleAppender.SystemOut , ver que "hola" todavía está impreso. Pero cuando sacamos el PatternLayout , una vez más llanuras sobre la falta de un flujo de salida. Este es un comportamiento muy extraño.

Mirando un poco más detenidamente la documentación, vemos que el valor predeterminado El constructor ConsoleAppender no está configurado, lo que no parece demasiado obvio o útil. Esto se siente como un error, o al menos una inconsistencia, en log4j .

Un poco más de búsqueda en Google, lectura y pruebas, y finalmente terminamos con el Listado 8-1. Hemos descubierto mucho sobre la forma en que funciona log4j y lo hemos codificado conocimiento en un conjunto de pruebas unitarias simples.

Listado 8-1

LogTest.java

```
public class LogTest {
    registrador registrador privado;

    @Antes
    public void initialize () {
        logger = Logger.getLogger ("registrador");
        logger.removeAllAppenders ();
        Logger.getRootLogger (). RemoveAllAppenders ();
    }
    @Prueba
    public void basicLogger () {
        BasicConfigurator.configure ();
        logger.info ("basicLogger");
    }

    @Prueba
    public void addAppenderWithStream () {
     logger.addAppender (nuevo ConsoleAppender (
            nuevo PatternLayout ("% p% t% m% n"),
            ConsoleAppender.SYSTEM_OUT));
        logger.info ("addAppenderWithStream");
    }
```


Página 149

Listado 8-1 (continuación)

LogTest.java

```
    @Prueba
    public void addAppenderWithoutStream () {
     logger.addAppender (nuevo ConsoleAppender (
            nuevo PatternLayout ("% p% t% m% n")));
        logger.info ("addAppenderWithoutStream");
    }
}
```

Ahora sabemos cómo inicializar un registrador de consola simple y podemos encapsular ese conocimiento en nuestra propia clase de registrador para que el resto de nuestra aplicación esté aislado de la interfaz de límite log4j .

Las pruebas de aprendizaje son mejores que las gratuitas

Las pruebas de aprendizaje terminan por no costar nada. De todos modos, teníamos que aprender la API y escribir esas pruebas fueron una forma fácil y aislada de obtener ese conocimiento. Las pruebas de aprendizaje fueron experimentos precisos que ayudaron a aumentar nuestra comprensión.

Las pruebas de aprendizaje no solo son gratuitas, sino que también tienen un retorno de la inversión positivo. Cuando ahí son nuevas versiones del paquete de terceros, realizamos las pruebas de aprendizaje para ver si son diferencias de comportamiento.

Las pruebas de aprendizaje verifican que los paquetes de terceros que utilizamos funcionan de la manera que esperar que lo hagan. Una vez integrado, no hay garantías de que el código de terceros permanecerá compatible con nuestras necesidades. Los autores originales se verán presionados a cambiar su código a satisfacer nuevas necesidades propias. Corregirán errores y agregarán nuevas capacidades. Con cada la liberación viene con un nuevo riesgo. Si el paquete de terceros cambia de alguna manera incompatible con nuestras pruebas, lo averiguaremos de inmediato.

Ya sea que necesite el aprendizaje proporcionado por las pruebas de aprendizaje o no, un límite limpio debe estar respaldado por un conjunto de pruebas de salida que ejercen la interfaz de la misma manera que el el código de producción lo hace. Sin estas *pruebas de límites* para facilitar la migración, podríamos estar tentados a quedarnos con la versión anterior más tiempo del que deberíamos.

Usar código que aún no existe

Hay otro tipo de límite, uno que separa lo conocido de lo desconocido. Allí A menudo hay lugares en el código donde nuestro conocimiento parece caer al límite. Algunas veces lo que está al otro lado de la frontera es incognoscible (al menos en este momento). Algunas veces elegimos no mirar más allá del límite.

Hace algunos años, formé parte de un equipo que desarrollaba software para un equipo de radio. sistema de comunicaciones. Había un subsistema, el "Transmisor", del que sabíamos poco sobre, y las personas responsables del subsistema no habían llegado al punto de definir su interfaz. No queríamos ser bloqueados, así que comenzamos nuestro trabajo lejos del parte desconocida del código.

Página 150

Teníamos una idea bastante clara de dónde terminaba nuestro mundo y comenzaba el nuevo mundo. Como nosotros funcionó, a veces tropezamos con este límite. Aunque nieblas y nubes de la ignorancia oscureció nuestra vista más allá de la frontera, nuestro trabajo nos hizo conscientes de lo que *quería que fuera* la interfaz de límites. Queríamos decirle al transmisor algo como esto:

> *Introduzca el transmisor en la frecuencia proporcionada y emita una representación analógica de la datos provenientes de esta secuencia.*

No teníamos idea de cómo se haría eso porque la API aún no se había diseñado. Así que decidimos trabajar en los detalles más tarde.

Para evitar ser bloqueados, definimos nuestra propia interfaz. Lo llamamos algo pegadizo, como `transmisor` . Le dimos un método llamado `transmisión` que tomaba una frecuencia y una flujo de datos. Esta era la interfaz que nos *hubiera* gustado tener.

Una cosa buena de escribir la interfaz que desearíamos tener es que está debajo de nuestra control. Esto ayuda a mantener el código del cliente más legible y enfocado en lo que está intentando realizar.

En la Figura 8-2, puede ver que aislamos las clases `CommunicationsController` de la API del transmisor (que estaba fuera de nuestro control y no estaba definida). Utilizando el nuestro interfaz específica de la aplicación, mantuvimos nuestro código de `CommunicationsController` limpio y expresivo. Una vez definida la API del transmisor, escribimos el `TransmitterAdapter` para cubre la brecha. El A DAPTER [2] encapsuló la interacción con la API y proporciona una único lugar para cambiar cuando la API evoluciona.

Figura 8-2
Predecir el transmisor

Este diseño también nos da una costura [3] muy conveniente en el código para la prueba. Usando un FakeTransmitter adecuado , podemos probar las clases CommunicationsController . También podemos crear pruebas de límites una vez que tengamos el TransmitterAPI que aseguren que estamos usando el API correctamente.

2. Vea el patrón del adaptador en [GOF].
3. Vea más sobre costuras en [WELC].

Página 151

Límites limpios

Suceden cosas interesantes en los límites. El cambio es una de esas cosas. Buen software los diseños se adaptan al cambio sin grandes inversiones ni retrabajos. Cuando usamos código que está fuera de nuestro control, se debe tener especial cuidado para proteger nuestra inversión y hacer seguro que el cambio futuro no es demasiado costoso.

El código en los límites necesita una separación clara y pruebas que definan las expectativas. Nosotros debe evitar que gran parte de nuestro código sepa sobre los detalles de terceros. Es mejor depender de algo *que* controlas que de algo que no controlas, para que no acabe controlándote.

Gestionamos los límites de terceros al tener muy pocos lugares en el código que se refieren a ellos. Podemos envolverlos como lo hicimos con Map , o podemos usar un A DAPTER para convertir de nuestra interfaz perfecta para la interfaz proporcionada. De cualquier forma, nuestro código nos habla mejor, promueve el uso internamente consistente a través del límite y tiene menos mantenimiento puntos cuando cambia el código de terceros.

Bibliografía

[BeckTDD]: *Desarrollo* **basado en** *pruebas,* Kent Beck, Addison-Wesley, 2003.

[GOF]: *Patrones de diseño: elementos de software orientado a objetos reutilizables,* Gamma et al., Addison-Wesley, 1996.

[WELC]: *Trabajar* **eficazmente** *con el código heredado,* Addison-Wesley, 2004.

9

Pruebas unitarias

Nuestra profesión ha avanzado mucho en los últimos diez años. En 1997 nadie había oído hablar de Desarrollo basado en pruebas. Para la gran mayoría de nosotros, las pruebas unitarias eran pequeños fragmentos de código que escribimos para asegurarnos de que nuestros programas "funcionaran". Nosotros concienzudamente escribimos nuestras clases y métodos, y luego inventamos un código ad hoc para probar ellos. Normalmente, esto implicaría algún tipo de programa controlador simple que permitiría para interactuar manualmente con el programa que habíamos escrito.

Recuerdo haber escrito un programa en C ++ para un sistema integrado en tiempo real en el mediados de los 90. El programa era un temporizador simple con la siguiente firma:

```
void Timer :: ScheduleCommand (comando * theCommand, int milisegundos)
```

La idea era sencilla; el método de ejecución del comando se ejecutará en una nueva subproceso después del número especificado de milisegundos. El problema era cómo probarlo.

Página 153

Hice improvisar un programa de controlador simple que escuchaba el teclado. Cada vez que un se escribió un carácter, programaría un comando que escribiría el mismo carácter cinco segundos más tarde. Luego toqué una melodía rítmica en el teclado y esperé a que melodía para reproducir en la pantalla cinco segundos después.

"I . . . quiero-una-chica. . . sólo . . . como-la-chica-que-se-casó. . . ied. . . querido . . . viejo . . . padre."

De hecho, canté esa melodía mientras escribía el "." clave, y luego la canté de nuevo como el aparecieron puntos en la pantalla.

¡Esa fue mi prueba! Una vez que lo vi funcionar y se lo demostré a mis colegas, arrojé el código de prueba de distancia.

Como dije, nuestra profesión ha avanzado mucho. Hoy en día escribiría una prueba que hiciera seguro de que cada rincón y grieta de ese código funcionó como esperaba. Yo aislaría mi código del sistema operativo en lugar de simplemente llamar a las funciones de temporización estándar. I simulaba esas funciones de sincronización para que yo tuviera un control absoluto sobre el tiempo. me gustaría programar comandos que establezcan indicadores booleanos, y luego adelantaría el tiempo, mirando esas banderas y asegurándome de que pasaran de falso a verdadero justo cuando cambié la hora a la valor correcto.

Una vez que obtuviera un conjunto de pruebas para aprobar, me aseguraría de que esas pruebas fueran convenientes para ejecutarlo para cualquier otra persona que necesite trabajar con el código. Me aseguraría de que las pruebas y el código se registró en conjunto en el mismo paquete fuente.

Sí, hemos recorrido un largo camino; pero tenemos que ir más lejos. El movimiento ágil y TDD mentos han animado a muchos programadores a escribir pruebas unitarias automatizadas, y hay más uniéndose a sus filas todos los días. Pero en la loca carrera por agregar pruebas a nuestra disciplina, muchos Los programadores han pasado por alto algunos de los puntos más sutiles e importantes de la escritura. buenas pruebas.

Las tres leyes de TDD

A estas alturas, todo el mundo sabe que TDD nos pide que escribamos pruebas unitarias primero, antes de escribir produc- código de ción. Pero esa regla es solo la punta del iceberg. Considere las siguientes tres leyes: [1]

Primera ley No puede escribir código de producción hasta que haya escrito una prueba unitaria reprobatoria.

Segunda ley No puede escribir más de una prueba unitaria de lo que es suficiente para fallar, y no el apilamiento está fallando.

Tercera ley No puede escribir más código de producción del que es suficiente para pasar la prueba que falla constantemente.

1. *Profesionalismo y desarrollo basado en pruebas* , Robert C. Martin, Mentor de objetos, IEEE Software, mayo / junio de 2007 (Vol. 24, Núm. 3) págs. 32–36 http://doi.ieeecomputersociety.org/10.1109/MS.2007.85

Página 154

Estas tres leyes te encierran en un ciclo que puede durar unos treinta segundos. Los exámenes y el código de producción se escriben *juntos,* con las pruebas solo unos segundos antes de la Codigo de producción.

Si trabajamos de esta manera, escribiremos docenas de pruebas todos los días, cientos de pruebas cada mes y miles de pruebas cada año. Si trabajamos de esta manera, esas pruebas cubrirán virtudes alia todo nuestro código de producción. La gran mayoría de esas pruebas, que pueden rivalizar con el tamaño del código de producción en sí mismo, puede presentar un problema de gestión desalentador.

Mantener limpias las pruebas

Hace algunos años me pidieron que entrenara a un equipo que había decidido explícitamente que su prueba El código *no* debe mantenerse con los mismos estándares de calidad que su código de producción. Se dieron licencia mutuamente para romper las reglas en sus pruebas unitarias. "Rápido y sucio" fue la consigna. Sus variables no tenían que estar bien nombradas, sus funciones de prueba no debe ser breve y descriptivo. Su código de prueba no necesitaba estar bien diseñado y cuidadosamente dividido. Siempre que el código de prueba funcione y cubra el pro- código de producción, era lo suficientemente bueno.

Algunos de los que lean esto pueden simpatizar con esa decisión. Quizás, mucho tiempo en el pasado, escribiste pruebas del tipo que escribí para esa clase de Timer . Es un gran paso desde escribir ese tipo de prueba desechable, hasta escribir un conjunto de pruebas unitarias automatizadas. Entonces, como el equipo que estaba entrenando, podría decidir que tener pruebas sucias es mejor que no tener pruebas.

Lo que este equipo no se dio cuenta fue que tener pruebas sucias es equivalente, si no peor que, no tener pruebas. El problema es que las pruebas deben cambiar a medida que el código de producción evoluciona. Cuanto más sucias sean las pruebas, más difícil será cambiarlas. Cuanto más enredado es el código de prueba, lo más probable es que dedique más tiempo a meter nuevas pruebas en la suite que a tarda en escribir el nuevo código de producción. A medida que modifica el código de producción, comienzan las pruebas antiguas fallar, y el desorden en el código de prueba hace que sea difícil hacer que esas pruebas vuelvan a pasar. Entonces el las pruebas se ven como una responsabilidad cada vez mayor.

Desde el lanzamiento hasta el lanzamiento, el costo de mantener el conjunto de pruebas de mi equipo aumentó. Eventualmente se convirtió en la queja más grande entre los desarrolladores. Cuando los gerentes preguntaron por qué sus estimaciones eran tan grandes que los desarrolladores culparon a las pruebas. Al final fueron obligado a descartar la suite de pruebas por completo.

Pero, sin un conjunto de pruebas, perdieron la capacidad de asegurarse de que los cambios en su código la base funcionó como se esperaba. Sin un conjunto de pruebas, no podían garantizar que los cambios en uno parte de su sistema no rompió otras partes de su sistema. Entonces su tasa de defectos comenzó a aumento. A medida que aumentaba el número de defectos no deseados, empezaron a temer hacer cambios. Ellos dejaron de limpiar su código de producción porque temían que los cambios hicieran más mal que bien. Su código de producción comenzó a pudrirse. Al final se quedaron sin pruebas, código de producción enredado y plagado de errores, clientes frustrados y la sensación de que su el esfuerzo de prueba les había fallado.

En cierto modo tenían razón. Su esfuerzo de prueba les *había* fallado. Pero fue su decisión Permitir que las pruebas fueran desordenadas, esa fue la semilla de ese fracaso. Si hubieran mantenido sus pruebas limpio, su esfuerzo de prueba no habría fallado. Puedo decir esto con cierta certeza porque He participado y entrenado en muchos equipos que han tenido éxito con la unidad *limpia.* pruebas.

La moraleja de la historia es simple: *el código de prueba es tan importante como el código de producción.* Eso

no es un ciudadano de segunda clase. Requiere pensamiento, diseño y cuidado. Debe mantenerse tan limpio como código de producción.

Las pruebas habilitan las -ilidades

Si no mantiene limpias las pruebas, las perderá. Y sin ellos, pierdes el mismísimo algo que mantiene su código de producción flexible. Sí, lo leíste correctamente. Son *pruebas unitarias* que mantienen nuestro código flexible, mantenible y reutilizable. La razón es simple. Si usted tiene pruebas, ¡no temes hacer cambios en el código! Sin pruebas, cada cambio es posible bicho. No importa qué tan flexible sea su arquitectura, no importa qué tan bien particionó su diseño, sin pruebas serás reacio a realizar cambios por temor a que introducirá errores no detectados.

Pero *con las* pruebas ese miedo prácticamente desaparece. Cuanto mayor sea la cobertura de su prueba, menos tu miedo. Puede realizar cambios con casi impunidad en el código que tiene un menos que estelar arquitectura y un diseño enredado y opaco. De hecho, puedes *mejorar* esa arquitectura y diseña sin miedo!

Por lo tanto, tener un conjunto automatizado de pruebas unitarias que cubra el código de producción es la clave para manteniendo su diseño y arquitectura lo más limpios posible. Las pruebas permiten todas las -ilidades, porque las pruebas permiten el *cambio* .

Entonces, si sus pruebas están sucias, entonces su capacidad para cambiar su código se ve obstaculizada y comienzan a perder la capacidad de mejorar la estructura de ese código. Cuanto más sucias sean tus pruebas, más sucio se vuelve su código. Eventualmente pierde las pruebas y su código se pudre.

Pruebas limpias

¿Qué hace que una prueba sea limpia? Tres cosas. Legibilidad, legibilidad y legibilidad. Leer- La capacidad es quizás incluso más importante en las pruebas unitarias que en el código de producción. Qué hace que las pruebas sean legibles? Lo mismo que hace que todo el código sea legible: claridad, simplicidad, y densidad de expresión. En una prueba quieres decir mucho con tan pocas expresiones como posible.

Considere el código de FitNesse en el Listado 9-1. Estas tres pruebas son difíciles de comprender y ciertamente se puede mejorar. Primero, hay una terrible cantidad de duplicados código [G5] en las llamadas repetidas a addPage y assertSubString . Más importante aún, esto El código simplemente está cargado de detalles que interfieren con la expresividad de la prueba.

Listado 9-1

SerializedPageResponderTest.java

```
public void testGetPageHieratchyAsXml () arroja una excepción
{
    crawler.addPage (root, PathParser.parse ("PageOne"));
    crawler.addPage (root, PathParser.parse ("PageOne.ChildOne"));
    crawler.addPage (root, PathParser.parse ("PageTwo"));

    request.setResource ("raíz");
    request.addInput ("tipo", "páginas");
    Responder respondedor = new SerializedPageResponder ();
    Respuesta SimpleResponse =
      (Respuesta simple) respondedor.makeResponse (
            new FitNesseContext (raíz), solicitud);
    String xml = response.getContent ();

    assertEquals ("texto / xml", response.getContentType ());
    asertSubString ("<nombre> PageOne </nombre>", xml);
    assertSubString ("<nombre> PageTwo </nombre>", xml);
    asertSubString ("<nombre> ChildOne </name>", xml);
}

prueba de vacío públicoGetPageHieratchyAsXmlDoesntContainSymbolicLinks ()
```

```
lanza una excepción
{
    WikiPage pageOne = crawler.addPage (root, PathParser.parse ("PageOne"));
    crawler.addPage (root, PathParser.parse ("PageOne.ChildOne"));
    crawler.addPage (root, PathParser.parse ("PageTwo"));

    PageData datos = pageOne.getData ();
    Propiedades de WikiPageProperties = data.getProperties ();
    WikiPageProperty symLinks = properties.set (SymbolicPage.PROPERTY_NAME);
    symLinks.set ("SymPage", "PageTwo");
    pageOne.commit (datos);

    request.setResource ("raíz");
    request.addInput ("tipo", "páginas");
    Responder respondedor = new SerializedPageResponder ();
    Respuesta SimpleResponse =
      (Respuesta simple) respondedor.makeResponse (
            new FitNesseContext (raíz), solicitud);
    String xml = response.getContent ();

    assertEquals ("texto / xml", response.getContentType ());
    asertSubString ("<nombre> PageOne </nombre>", xml);
    assertSubString ("<nombre> PageTwo </nombre>", xml);
    asertSubString ("<nombre> ChildOne </name>", xml);
    asertNotSubString ("SymPage", xml);
}

public void testGetDataAsHtml () arroja una excepción
{
    crawler.addPage (root, PathParser.parse ("TestPageOne"), "página de prueba");

    request.setResource ("TestPageOne");
    request.addInput ("tipo", "datos");
```


Página 157

Listado 9-1 (continuación)

SerializedPageResponderTest.java

```
    Responder respondedor = new SerializedPageResponder ();
    Respuesta SimpleResponse =
        (Respuesta simple) respondedor.makeResponse (
            new FitNesseContext (raíz), solicitud);
    String xml = response.getContent ();

    assertEquals ("texto / xml", response.getContentType ());
    asertSubString ("página de prueba", xml);
    asertSubString ("<Prueba", xml);
}
```

Por ejemplo, mire las llamadas `PathParser` . Transforman cadenas en `PagePath` instancias utilizadas por los rastreadores. Esta transformación es completamente irrelevante para la prueba en mano y sirve sólo para ofuscar la intención. Los detalles que rodean la creación del `respondedor` y la recopilación y emisión de la `respuesta` también son solo ruido. Luego está el forma torpe en la que la URL de la solicitud se construye a partir de un `recurso` y un argumento. (Ayudé escribir este código, así que me siento libre de criticarlo rotundamente).

Al final, este código no fue diseñado para ser leído. El pobre lector se inunda con un enjambre de detalles que deben entenderse antes de que las pruebas tengan algún sentido real.

Ahora considere las pruebas mejoradas del Listado 9-2. Estas pruebas hacen exactamente lo mismo, pero se han refactorizado en una forma mucho más limpia y explicativa.

Listado 9-2

SerializedPageResponderTest.java (refactorizado)

```
public void testGetPageHierarchyAsXml () lanza Exception {
    makePages ("PageOne", "PageOne.ChildOne", "PageTwo");

    submitRequest ("raíz", "tipo: páginas");

    asertResponseIsXML ();
    asertResponseContains (
        "<nombre> PageOne </name>", "<nombre> PageTwo </name>", "<nombre> ChildOne </name>"
```

```
    );
}

public void testSymbolicLinksAreNotInXmlPageHierarchy () lanza Exception {
    Página WikiPage = makePage ("PageOne");
    makePages ("PageOne.ChildOne", "PageTwo");

    addLinkTo (página, "PageTwo", "SymPage");

    submitRequest ("raíz", "tipo: páginas");

    asertResponseIsXML ();
    asertResponseContains (
        "<nombre> PageOne </name>", "<nombre> PageTwo </name>", "<nombre> ChildOne </name>"
    );
    asertResponseDoesNotContain ("SymPage");
}
```


Página 158

Listado 9-2 (continuación)
SerializedPageResponderTest.java (refactorizado)

```
public void testGetDataAsXml () lanza Exception {
    makePageWithContent ("TestPageOne", "página de prueba");

    submitRequest ("TestPageOne", "tipo: datos");

    asertResponseIsXML ();
    asertResponseContains ("página de prueba", "<Prueba");
}
```

El patrón B UILD -O PERATE -C HECK ₂ se hace evidente por la estructura de estas pruebas. Cada una de las pruebas se divide claramente en tres partes. La primera parte construye los datos de prueba, el La segunda parte opera con esos datos de prueba, y la tercera parte verifica que la operación rindió los resultados esperados.

Tenga en cuenta que se ha eliminado la gran mayoría de los detalles molestos. Las pruebas consiguen directamente al grano y use solo los tipos de datos y las funciones que realmente necesitan. Alguien Quien lee estas pruebas debe poder averiguar lo que hacen muy rápidamente, sin ser engañado o abrumado por los detalles.

Lenguaje de prueba específico del dominio

Las pruebas del Listado 9-2 demuestran la técnica de construir un lenguaje específico de dominio. para sus pruebas. En lugar de utilizar las API que los programadores utilizan para manipular el sistema tem, creamos un conjunto de funciones y utilidades que hacen uso de esas API y que hacen que las pruebas sean más cómodas de escribir y más fáciles de leer. Estas funciones y utilidades conviértase en una API especializada utilizada por las pruebas. Son un *lenguaje de* prueba que programa Los usuarios utilizan para ayudarse a sí mismos a escribir sus pruebas y para ayudar a los que deben leerlos. pruebas más adelante.

Esta API de prueba no está diseñada por adelantado; más bien, evoluciona a partir de la continua refactoring de código de prueba que se ha corrompido demasiado por la confusión de detalles. Tal como me viste refactorizarán el Listado 9-1 en el Listado 9-2, así también los desarrolladores disciplinados refactorizarán su prueba codifique en formas más sucintas y expresivas.

Un doble estándar

En cierto sentido, el equipo que mencioné al principio de este capítulo tenía las cosas bien. La código dentro de la API de prueba *no* tiene un conjunto diferente de estándares de ingeniería de productividad código de ción. Debe ser simple, conciso y expresivo, pero no es necesario que sea tan eficiente como Codigo de producción. Después de todo, se ejecuta en un entorno de prueba, no en un entorno de producción, y esos dos entornos tienen necesidades muy diferentes.

2. http://fitnesse.org/FitNesse.AcceptanceTestPatterns

Página 159

Considere la prueba del Listado 9-3. Escribí esta prueba como parte de un sistema de control de entorno tem que estaba haciendo prototipos. Sin entrar en detalles, puede decir que esta prueba verifica que la alarma de baja temperatura, el calentador y el ventilador se encienden cuando la temperatura La temperatura es "demasiado fría".

Listado 9-3
EnvironmentControllerTest.java

```
@Prueba
  public void turnOnLoTempAlarmAtThreashold () lanza Exception {
    hw.setTemp (WAY_TOO_COLD);
    controller.tic ();
    asertTrue (hw.heaterState ());
    asertTrue (hw.blowerState ());
    asertFalse (hw.coolerState ());
    asertFalse (hw.hiTempAlarm ());
    asertTrue (hw.loTempAlarm ());
  }
```

Por supuesto, aquí hay muchos detalles. Por ejemplo, ¿cuál es esa función `tic` todo ¿acerca de? De hecho, prefiero que no se preocupe por eso mientras lee esta prueba. Prefiero que solo preocuparse acerca de si está de acuerdo en que el estado final del sistema es consistente con la temperatura peratura siendo "demasiado fría".

Observe, mientras lee la prueba, que su ojo necesita rebotar de un lado a otro entre el nombre del estado que se está comprobando y el *sentido* del estado que se está comprobando. Verás `calentadorState` , y luego tus ojos brillan hacia la izquierda para `afirmar Verdad` . Ves `CoolerState` y tu los ojos deben seguir a la izquierda para `afirmar Falso` . Esto es tedioso y poco confiable. Hace que la prueba sea difícil leer.

Mejoré enormemente la lectura de esta prueba transformándola en el Listado 9-4.

Listado 9-4
EnvironmentControllerTest.java (refactorizado)

```
@Prueba
  public void turnOnLoTempAlarmAtThreshold () lanza Exception {
    wayTooCold ();
    asertEquals ("HBchL", hw.getState ());
  }
```

Por supuesto, oculté el detalle de la función `tic` creando una función `wayTooCold` . Pero el Lo que hay que tener en cuenta es la extraña cadena en `assertEquals` . Mayúsculas significa "activado", minúsculas significa "apagado" y las letras están siempre en el siguiente orden: `{calentador, ventilador, enfriador, alarma de alta temperatura, alarma de baja temperatura}` .

Aunque esto está cerca de una violación de la regla sobre el mapeo mental, [3] parece apropiado en este caso. Fíjate, una vez que conoces el significado, tus ojos se deslizan

3. “Evite el mapeo mental” en la página 25.

Página 160

esa cadena y puede interpretar rápidamente los resultados. Leer el examen se vuelve casi un Placer. Solo eche un vistazo al Listado 9-5 y compruebe lo fácil que es comprender estas pruebas.

Listado 9-5
EnvironmentControllerTest.java (selección más grande)

```
@Prueba
  public void turnOnCoolerAndBlowerIfTooHot () lanza Exception {
      demasiado caliente();
      asertEquals ("hBChl", hw.getState ());
  }

  @Prueba
  public void turnOnHeaterAndBlowerIfTooCold () lanza Exception {
      muy frío();
      asertEquals ("HBchl", hw.getState ());
  }

  @Prueba
  public void turnOnHiTempAlarmAtThreshold () lanza Exception {
      wayTooHot ();
      asertEquals ("hBCHl", hw.getState ());
  }

  @Prueba
  public void turnOnLoTempAlarmAtThreshold () lanza Exception {
      wayTooCold ();
      asertEquals ("HBchL", hw.getState ());
  }
```

La función getState se muestra en el Listado 9-6. Tenga en cuenta que esto no es muy eficiente código. Para hacerlo eficiente, probablemente debería haber usado un StringBuffer .

Listado 9-6
MockControlHardware.java

```
public String getState () {
      Estado de cadena = "";
      estado + = calentador? "S.S";
      estado + = soplador? "B": "b";
      estado + = más fresco? "C": "c";
      estado + = hiTempAlarm? "S.S";
      estado + = loTempAlarm? "L": "l";
      estado de retorno;
  }
```

Los StringBuffer s son un poco feos. Incluso en el código de producción, los evitaré si el costo es pequeña; y podría argumentar que el costo del código del Listado 9-6 es muy pequeño. Sin emabargo, esta aplicación es claramente un sistema integrado en tiempo real, y es probable que la computadora y los recursos de memoria están muy limitados. La *prueba de* medio ambiente, sin embargo, no es probable que sea restringido en absoluto.

Página 161

Esa es la naturaleza del estándar dual. Hay cosas que tal vez nunca hagas en un

entorno de producción que están perfectamente bien en un entorno de prueba. Por lo general involucran problemas de memoria o eficiencia de la CPU. Pero *nunca* involucran problemas de limpieza.

Una afirmación por prueba

Hay una escuela de pensamiento [4] que dice que cada función de prueba en una prueba JUnit debe tener una y solo una declaración de aserción. Esta regla puede parecer draconiana, pero se puede ver la ventaja. en el Listado 9-5. Esas pruebas llegan a una única conclusión que es rápida y fácil de entender.

Pero, ¿qué pasa con el Listado 9-2? Parece irrazonable que de alguna manera podamos fusionar la afirmación de que la salida es XML y que contiene ciertas subcadenas. Cómo-Nunca, podemos dividir la prueba en dos pruebas separadas, cada una con su propia aserción particular, como se muestra en el Listado 9-7.

Listado 9-7
SerializedPageResponderTest.java (afirmación única)

```
public void testGetPageHierarchyAsXml () lanza Exception {
      givenPages ("PageOne", "PageOne.ChildOne", "PageTwo");

      whenRequestIsIssued ("raíz", "tipo: páginas");

      thenResponseShouldBeXML ();
   }

   public void testGetPageHierarchyHasRightTags () lanza Exception {
      givenPages ("PageOne", "PageOne.ChildOne", "PageTwo");

      whenRequestIsIssued ("raíz", "tipo: páginas");

      thenResponseShouldContain (
         "<nombre> PageOne </name>", "<nombre> PageTwo </name>", "<nombre> ChildOne </name>"
      );
   }
```

Observe que he cambiado los nombres de las funciones para usar el común dado-cuando-luego [5] convención. Esto hace que las pruebas sean aún más fáciles de leer. Desafortunadamente, dividir las pruebas como se muestra da como resultado una gran cantidad de código duplicado.

Podemos eliminar la duplicación utilizando la T emplatar M ÉTODO [6] patrón y poner el *dado / cuando* partes en la clase base, y el *entonces* partes en diferentes derivadas. O podríamos crear una clase de prueba completamente separada y poner las partes *dadas* y *when* en la función @Before ción, y las partes *when* en cada función @Test . Pero esto parece demasiado mecanismo para un problema tan menor. Al final, prefiero las afirmaciones múltiples del Listado 9-2.

4. Consulte la entrada del blog de Dave Astel: http://www.artima.com/weblogs/viewpost.jsp?thread=35578
5. [RSpec].
6. [GOF].

Creo que la regla de aserción única es una buena guía. [7] Normalmente trato de crear un dominio lenguaje de prueba específico que lo admita, como en el Listado 9-5. Pero no tengo miedo de poner más de una afirmación en una prueba. Creo que lo mejor que podemos decir es que el número de afirma en una prueba debe minimizarse.

Concepto único por prueba

Quizás una regla mejor es que queremos probar un solo concepto en cada función de prueba. Nosotros no quieren funciones de prueba largas que vayan probando una cosa miscelánea tras otra. Listado 9-8 es un ejemplo de tal prueba. Esta prueba debe dividirse en tres pruebas independientes porque prueba tres cosas independientes. Fusionándolos todos juntos en la misma función obliga al lector a averiguar por qué cada sección está allí y qué está siendo probado por esa sección.

Listado 9-8

```
/ **
  * Pruebas varias para el método addMonths ().
  * /
public void testAddMonths () {
        SerialDate d1 = SerialDate.createInstance (31, 5, 2004);

        SerialDate d2 = SerialDate.addMonths (1, d1);
        asertEquals (30, d2.getDayOfMonth ());
        asertEquals (6, d2.getMonth ());
        asertEquals (2004, d2.getYYYY ());

        SerialDate d3 = SerialDate.addMonths (2, d1);
        asertEquals (31, d3.getDayOfMonth ());
        asertEquals (7, d3.getMonth ());
        asertEquals (2004, d3.getYYYY ());

        SerialDate d4 = SerialDate.addMonths (1, SerialDate.addMonths (1, d1));
        asertEquals (30, d4.getDayOfMonth ());
        asertEquals (7, d4.getMonth ());
        asertEquals (2004, d4.getYYYY ());
}
```

Las tres funciones de prueba probablemente deberían ser así:

• *Dado* el último día de un mes con 31 días (como mayo):

1. *Cuando* agrega un mes, de modo que el último día de ese mes sea el 30 (como junio), *entonces* la fecha debe ser el 30 de ese mes, no el 31.
2. *Cuando* agrega dos meses a esa fecha, de modo que el último mes tenga 31 días, *entonces* la fecha debería ser el 31.

7. "¡Cumpla con el código!"

Página 163

• *Dado* el último día de un mes con 30 días (como junio):

1. *Cuando* agrega un mes de modo que el último día de ese mes tenga 31 días, *entonces* el la fecha debe ser el 30, no el 31.

Dicho así, puede ver que hay una regla general que se esconde en medio de la miscelánea. pruebas neous. Cuando incrementa el mes, la fecha no puede ser mayor que el último día de el mes. Esto implica que incrementar el mes el 28 de febrero debería producir marzo 28. Falta *esa* prueba y sería útil escribirla.

Entonces, no son las múltiples afirmaciones en cada sección del Listado 9-8 las que causan el problema. Más bien es el hecho de que se está probando más de un concepto. Entonces probablemente el mejor La regla es que debe minimizar el número de afirmaciones por concepto y probar solo una con- excepto por función de prueba.

PRIMEROS [8]

Las pruebas limpias siguen otras cinco reglas que forman el acrónimo anterior:

Las pruebas **rápidas** deben ser rápidas. Deben correr rápido. Cuando las pruebas son lentas, no querrás para ejecutarlos con frecuencia. Si no los ejecuta con frecuencia, no encontrará problemas temprano suficiente para arreglarlos fácilmente. No se sentirá tan libre para limpiar el código. Eventualmente el código comenzará a pudrirse.

Las pruebas **independientes** no deben depender unas de otras. Una prueba no debe configurar la condición ciones para la siguiente prueba. Debería poder ejecutar cada prueba de forma independiente y ejecutar las pruebas en

cualquier orden que te guste. Cuando las pruebas dependen unas de otras, la primera en fallar provoca un cascadencia de fallas aguas abajo, lo que dificulta el diagnóstico y oculta los defectos aguas abajo.

Las pruebas **repetibles** deben poder repetirse en cualquier entorno. Debería poder ejecutar el pruebas en el entorno de producción, en el entorno de control de calidad y en su computadora portátil mientras volver a casa en el tren sin una red. Si sus pruebas no se pueden repetir en ningún entorno ment, entonces siempre tendrá una excusa de por qué fallan. También te encontrarás incapaz para ejecutar las pruebas cuando el entorno no está disponible.

Autovalidables Las pruebas deben tener una salida booleana. O pasan o no. Tú No debería tener que leer un archivo de registro para saber si las pruebas pasan. No deberías tener para comparar manualmente dos archivos de texto diferentes para ver si las pruebas pasan. Si las pruebas no son autovalidables, el fallo puede volverse subjetivo y ejecutar las pruebas puede requerir un largo evaluación manual.

8. Materiales de capacitación para mentores de objetos.

Página 164

Oportunamente Las pruebas deben redactarse de manera oportuna. Las pruebas unitarias deben escribirse *solo antes* del código de producción que los hace pasar. Si escribe pruebas después de la producción código, entonces puede encontrar que el código de producción sea difícil de probar. Puede decidir que algunos El código de producción es demasiado difícil de probar. No puede diseñar el código de producción para que sea comprobable.

Conclusión

Apenas hemos arañado la superficie de este tema. De hecho, creo que un libro completo podría escrito sobre *pruebas limpias* . Las pruebas son tan importantes para la salud de un proyecto como la producción el código es. Quizás sean aún más importantes, porque las pruebas preservan y mejoran la flexibilidad, mantenibilidad y reutilización del código de producción. Así que mantén tus pruebas con permanentemente limpio. Trabaja para que sean expresivos y concisos. Invente API de prueba que actúen como Lenguaje específico del dominio que le ayuda a redactar las pruebas.

Si deja que las pruebas se pudran, su código también se pudrirá. Mantenga limpias sus pruebas.

Bibliografía

[RSpec]: *RSpec: Desarrollo basado en comportamiento para programadores Ruby* , Aslak Hellesøy, David Chelimsky, Estantería pragmática, 2008.

[GOF]: *Patrones de diseño: elementos de software orientado a objetos reutilizables* , Gamma et al., Addison-Wesley, 1996.

Página 165

Esta página se dejó en blanco intencionalmente

Página 166

10

Clases

con Jeff Langr

Hasta ahora en este libro nos hemos centrado en cómo escribir bien líneas y bloques de código. Tenemos
profundizó en la composición adecuada de las funciones y cómo se interrelacionan. Pero para todos los asistentes
relación a la expresividad de las declaraciones de código y las funciones que comprenden, todavía
no tengamos un código limpio hasta que hayamos prestado atención a los niveles más altos de organización del código. Vamos
hablar de clases limpias.

Página 167

Organización de la clase

Siguiendo la convención estándar de Java, una clase debe comenzar con una lista de variables. Pub-
Las constantes estáticas lic, si las hay, deben ser lo primero. Luego, las variables estáticas privadas, seguidas de
variables de instancia vate. Rara vez hay una buena razón para tener una variable pública.

Las funciones públicas deben seguir la lista de variables. Nos gusta poner los servicios privados
llamado por una función pública inmediatamente después de la función pública en sí. Esto sigue al stepdown
regla y ayuda al programa a leerse como un artículo de periódico.

Encapsulamiento

Nos gusta mantener nuestras variables y funciones de utilidad privadas, pero no somos fanáticos de eso.
A veces necesitamos proteger una variable o función de utilidad para que pueda ser
accedido por una prueba. Para nosotros, las pruebas gobiernan. Si una prueba en el mismo paquete necesita llamar a una función
o acceder a una variable, la haremos protegida o con el alcance del paquete. Sin embargo, primero buscaremos
una forma de mantener la privacidad. Aflojar la encapsulación es siempre el último recurso.

¡Las clases deben ser pequeñas!

La primera regla de las clases es que deben ser pequeñas. La segunda regla de las clases es que
debería ser más pequeño que eso. No, no vamos a repetir exactamente el mismo texto del
Capítulo de *funciones* . Pero al igual que con las funciones, más pequeño es la regla principal cuando se trata de
diseñar clases. Al igual que con las funciones, nuestra pregunta inmediata es siempre "¿Qué tan pequeño?"

Con las funciones medimos el tamaño contando líneas físicas. Con las clases usamos un
medida diferente. Contamos *responsabilidades*. [1]

El Listado 10-1 describe una clase, SuperDashboard , que expone alrededor de 70 métodos públicos.
La mayoría de los desarrolladores estarían de acuerdo en que es demasiado grande. Algunos desarrolladores pueden referirse
a SuperDashboard como una "clase de Dios".

Listado 10-1
Demasiadas responsabilidades

```
SuperDashboard de clase pública extiende JFrame implementa MetaDataUser
    public String getCustomizerLanguagePath ()
    public void setSystemConfigPath (String systemConfigPath)
    public String getSystemConfigDocument ()
    public void setSystemConfigDocument (String systemConfigDocument)
    public boolean getGuruState ()
    public boolean getNoviceState ()
    public boolean getOpenSourceState ()
    showObject public void (objeto MetaObject)
    public void showProgress (String s)
```

1. [RDD].

Página 168

Listado 10-1 (continuación)
Demasiadas responsabilidades

```
    public boolean isMetadataDirty ()
    public void setIsMetadataDirty (booleano isMetadataDirty)
    componente público getLastFocusedComponent ()
    public void setLastFocused (componente lastFocused)
    public void setMouseSelectState (booleano isMouseSelected)
    public boolean isMouseSelected ()
    public LanguageManager getLanguageManager ()
```

```
    proyecto público getProject ()
    proyecto público getFirstProject ()
    proyecto público getLastProject ()
    public String getNewProjectName ()
    public void setComponentSizes (Dimensión atenuada)
    public String getCurrentDir ()
    public void setCurrentDir (String newDir)
    public void updateStatus (int dotPos, int markPos)
    clase pública [] getDataBaseClasses ()
    public MetadataFeeder getMetadataFeeder ()
    public void addProject (proyecto de proyecto)
    public boolean setCurrentProject (proyecto de proyecto)
    public boolean removeProject (proyecto de proyecto)
    public MetaProjectHeader getProgramMetadata ()
    resetDashboard vacío público ()
    proyecto público loadProject (String fileName, String projectName)
    public void setCanSaveMetadata (boolean canSave)
    public MetaObject getSelectedObject ()
    public void deselectObjects ()
    public void setProject (Proyecto de proyecto)
    public void editorAction (String actionName, evento ActionEvent)
    setMode public void (modo int)
    public FileManager getFileManager ()
    public void setFileManager (FileManager fileManager)
    public ConfigManager getConfigManager ()
    public void setConfigManager (ConfigManager configManager)
    public ClassLoader getClassLoader ()
    public void setClassLoader (ClassLoader classLoader)
    propiedades públicas getProps ()
    public String getUserHome ()
    public String getBaseDir ()
    public int getMajorVersionNumber ()
    public int getMinorVersionNumber ()
    public int getBuildNumber ()
    pegado público de MetaObject (
        Destino MetaObject, MetaObject pegado, proyecto MetaProject)
    public void processMenuItems (MetaObject metaObject)
    public void processMenuSeparators (MetaObject metaObject)
    public void processTabPages (MetaObject metaObject)
    processPlacement vacío público (objeto MetaObject)
    public void processCreateLayout (objeto MetaObject)
    public void updateDisplayLayer (objeto MetaObject, int layerIndex)
    public void propertyEditedRepaint (objeto MetaObject)
    public void processDeleteObject (objeto MetaObject)
    public boolean getAttachedToDesigner ()
    public void processProjectChangedState (booleano hasProjectChanged)
    public void processObjectNameChanged (objeto MetaObject)
    public void runProject ()
```


Página 169

Listado 10-1 (continuación)

Demasiadas responsabilidades

```
    public void setAçowDragging (booleano allowDragging)
    public boolean allowDragging ()
    public boolean isCustomizing ()
    public void setTitle (título de cadena)
    public IdeMenuBar getIdeMenuBar ()
    public void showHelper (MetaObject metaObject, String propertyName)
    // ... siguen muchos métodos no públicos ...
}
```

Pero, ¿y si SuperDashboard contuviera solo los métodos que se muestran en el Listado 10-2?

Listado 10-2

¿Suficientemente pequeño?

```
SuperDashboard de clase pública extiende JFrame implementa MetaDataUser
    componente público getLastFocusedComponent ()
    public void setLastFocused (componente lastFocused)
    public int getMajorVersionNumber ()
    public int getMinorVersionNumber ()
    public int getBuildNumber ()
}
```

Cinco métodos no es demasiado, ¿verdad? En este caso es porque a pesar de su reducido número de métodos, SuperDashboard tiene demasiadas *responsabilidades* .

El nombre de una clase debe describir qué responsabilidades cumple. De hecho, nombrar es probablemente la primera forma de ayudar a determinar el tamaño de la clase. Si no podemos derivar un conciso nombre de una clase, es probable que sea demasiado grande. Cuanto más ambiguo sea el nombre de la clase, más probablemente tenga demasiadas responsabilidades. Por ejemplo, nombres de clases que incluyen palabras de comadreja como Processor o Manager o Super a menudo insinúan una desafortunada agregación de responsabilidades.

También deberíamos poder escribir una breve descripción de la clase en unas 25 palabras, sin usar las palabras "si", "y", "o" o "pero". ¿Cómo describiríamos el SuperDashboard ? "El SuperDashboard proporciona acceso al último componente que contenía el focus, y también nos permite realizar un seguimiento de la versión y los números de compilación ". El primer "y" es un insinúe que SuperDashboard tiene demasiadas responsabilidades.

El principio de responsabilidad única

El Principio de Responsabilidad Única (SRP) [2] establece que una clase o módulo debe tener uno, y solo una, *razón para cambiar* . Este principio nos da una definición de responsabilidad, y pautas para el tamaño de la clase. Las clases deben tener una responsabilidad, una razón para cambio.

2. Puede leer mucho más sobre este principio en [PPP].

Página 170

La aparentemente pequeña clase SuperDashboard del Listado 10-2 tiene dos razones para cambiar. Primero, rastrea la información de la versión que aparentemente necesitaría ser actualizada cada vez que el se envía el software. En segundo lugar, gestiona los componentes de Java Swing (es un derivado de JFrame , la representación Swing de una ventana GUI de nivel superior). Sin duda vamos a querer Actualice el número de versión si cambiamos alguno de los códigos de Swing, pero lo contrario no es necesario esencialmente cierto: podríamos cambiar la información de la versión en función de los cambios en otro código en el sistema.

Tratar de identificar responsabilidades (razones para cambiar) a menudo nos ayuda a reconocer y crear mejores abstracciones en nuestro código. Podemos extraer fácilmente los tres SuperDashboard métodos que se ocupan de la información de la versión en una clase separada denominada Versión. (Ver Listado 10-3.) La clase Version es una construcción que tiene un alto potencial de reutilización en otros aplicaciones!

Listado 10-3
Una clase de responsabilidad única

```
Versión de clase pública {
    public int getMajorVersionNumber ()
    public int getMinorVersionNumber ()
    public int getBuildNumber ()
}
```

SRP es uno de los conceptos más importantes en el diseño OO. También es uno de los más simples conceptos para comprender y cumplir. Sin embargo, curiosamente, SRP es a menudo el diseño de clase más abusado principio. Regularmente encontramos clases que hacen demasiadas cosas. ¿Por qué?

Hacer que el software funcione y limpiar el software son dos actividades muy diferentes. La mayoría de nosotros tenemos un espacio limitado en la cabeza, por lo que nos enfocamos en hacer que nuestro código funcione más que la organización y la limpieza. Esto es totalmente apropiado. Mantener una separación de Las preocupaciones son tan importantes en nuestras *actividades de* programación como en nuestros programas.

El problema es que muchos de nosotros pensamos que hemos terminado una vez que el programa funciona. No logramos cambiar a la *otra* preocupación de la organización y la limpieza. Pasamos al siguiente problema en lugar de volver atrás y dividir las clases sobrecargadas en desacopladas Unidades con responsabilidades únicas.

Al mismo tiempo, muchos desarrolladores temen que una gran cantidad de pequeños proyectos de un solo propósito Las clases hacen que sea más difícil comprender el panorama general. Les preocupa que Deben navegar de una clase a otra para descubrir cómo se obtiene un trabajo más grande. logrado.

Sin embargo, un sistema con muchas clases pequeñas no tiene más partes móviles que un sistema con algunas clases grandes. Hay tanto que aprender en el sistema con unos pocos clases. Entonces la pregunta es: ¿Quiere que sus herramientas estén organizadas en cajas de herramientas con muchos cajones pequeños, cada uno con componentes bien definidos y bien etiquetados? O quieres algunos cajones en los que simplemente arrojas todo?

Cada sistema de tamaño considerable contendrá una gran cantidad de lógica y complejidad. El PRI- El objetivo principal de la gestión de tal complejidad es *organizarlo de* modo que un desarrollador sepa dónde

Página 171

buscar para encontrar cosas y solo necesita comprender la complejidad directamente afectada en cualquier tiempo dado. Por el contrario, un sistema con clases polivalentes más grandes siempre nos obstaculiza insistiendo en que analicemos muchas cosas que no necesitamos saber en este momento.

Para reafirmar los puntos anteriores para enfatizar: Queremos que nuestros sistemas estén compuestos de muchas clases pequeñas, no pocas grandes. Cada pequeña clase encapsula una sola responsabilidad bilidad, tiene una sola razón para cambiar y colabora con algunos otros para lograr el comportamientos deseados del sistema.

Cohesión

Las clases deben tener una pequeña cantidad de variables de instancia. Cada uno de los métodos de una clase debe manipular una o más de esas variables. En general, cuantas más variables tenga un método manipula cuanto más cohesivo es ese método con su clase. Una clase en la que cada variable es utilizado por cada método es máximamente cohesivo.

En general, no es aconsejable ni posible crear tales cohesiones máximas clases; por otro lado, nos gustaría que la cohesión fuera alta. Cuando la cohesión es alta, significa que los métodos y variables de la clase son co-dependientes y se cuelgan juntos como un todo lógico.

Considere la implementación de una `pila` en el Listado 10-4. Esta es una clase muy cohesionada. De los tres métodos, solo `size ()` no usa ambas variables.

Listado 10-4

Stack.java Una clase cohesionada.

```
Pila de clase pública {
   privado int topOfStack = 0;
   Lista <Integer> elementos = new LinkedList <Integer> ();

   public int size () {
      return topOfStack;
   }

   push public void (elemento int) {
      topOfStack ++;
      elementos.add (elemento);
   }

   public int pop () lanza PoppedWhenEmpty {
      si (topOfStack == 0)
         lanzar nuevo PoppedWhenEmpty ();
      int elemento = elementos.get (- topOfStack);
      elements.remove (topOfStack);
      elemento de retorno;
   }
}
```

La estrategia de mantener las funciones pequeñas y las listas de parámetros breves puede, en algunos casos, los tiempos conducen a una proliferación de variables de instancia que son utilizadas por un subconjunto de métodos. Cuando esto sucede, casi siempre significa que hay al menos otra clase tratando de

Página 172

sal de la clase más grande. Debe intentar separar las variables y los métodos en dos o más clases de modo que las nuevas clases sean más cohesivas.

Mantener los resultados de la cohesión en muchas clases pequeñas

El simple hecho de dividir funciones grandes en funciones más pequeñas provoca una proliferación de clases. Considere una función grande con muchas variables declaradas dentro de ella. Digamos tu desea extraer una pequeña parte de esa función en una función separada. Sin embargo, el código que desea extraer utiliza cuatro de las variables declaradas en la función. Debes pasar todo cuatro de esas variables en la nueva función como argumentos?

¡Para nada! Si promocionamos esas cuatro variables a variables de instancia de la clase, entonces podríamos extraer el código sin pasar *ninguna* variable. Sería *fácil* de romper la función en pedazos pequeños.

Desafortunadamente, esto también significa que nuestras clases pierden cohesión porque acumulan cada vez más variables de instancia que existen únicamente para permitir que algunas funciones las compartan. ¡Pero espera! Si hay algunas funciones que quieren compartir ciertas variables, ¿no es así? convertirlos en una clase por derecho propio? Claro que lo hace. Cuando las clases pierdan cohesión, divídanse ¡ellos!

Por lo tanto, dividir una función grande en muchas funciones más pequeñas a menudo nos da la oportunidad nity para dividir varias clases más pequeñas también. Esto le da a nuestro programa una organización mucho mejor nización y una estructura más transparente.

Como demostración de lo que quiero decir, usemos un ejemplo consagrado tomado de El maravilloso libro *Literate Programming de* Knuth . [3] Listado 10-5 muestra una traducción a Java del programa PrintPrimes de Knuth . Para ser justos con Knuth, este no es el programa como lo escribió. sino más bien como fue generado por su herramienta WEB. Lo estoy usando porque es un gran comienzo. lugar para dividir una función grande en muchas funciones y clases más pequeñas.

Listado 10-5

PrintPrimes.java

```
paquete literatePrimes;

PrintPrimes de clase pública {
  public static void main (String [] args) {
    int final M = 1000;
    int final RR = 50;
    int final CC = 4;
    final int WW = 10;
    final int ORDMAX = 30;
    int P [] = nuevo int [M + 1];
    int PAGENUMBER;
    int PAGEOFFSET;
    int ROWOFFSET;
    int C;
```

3. [Knuth92].

Página 173

Listado 10-5 (continuación)

PrintPrimes.java

```
        int J;
        int K;
        booleano JPRIME;
        int ORD;
        int CUADRADO;
        int N;
        int MULT [] = nuevo int [ORDMAX + 1];

        J = 1;
        K = 1;
        P [1] = 2;
        ORD = 2;
        CUADRADO = 9;

        mientras que (K <M) {
            hacer {
                J = J + 2;
                si (J == CUADRADO) {
                    ORD = ORD + 1;
                    CUADRADO = P [ORD] * P [ORD];
                    MULT [ORD - 1] = J;
                }
                N = 2;
                JPRIME = verdadero;
                mientras que (N <ORD && JPRIME) {
                    mientras que (MULT [N] <J)
                        MULT [N] = MULT [N] + P [N] + P [N];
                    si (MULT [N] == J)
                        JPRIME = falso;
                    N = N + 1;
                }
            } mientras (! JPRIME);
            K = K + 1;
            P [K] = J;
        }
        {
            PAGENUMBER = 1;
            PAGEOFFSET = 1;
            mientras que (PAGEOFFSET <= M) {
                System.out.println ("El primero" + M +
                                                "Números primos --- Página" + PAGENUMBER);
                System.out.println ("");
                para (ROWOFFSET = PAGEOFFSET; ROWOFFSET <PAGEOFFSET + RR; ROWOFFSET ++) {
                    para (C = 0; C <CC; C ++)
                        si (ROWOFFSET + C * RR <= M)
                            System.out.format ("% 10d", P [ROWOFFSET + C * RR]);
                    System.out.println ("");
                }
                System.out.println ("\ f");
                PAGENUMBER = PAGENUMBER + 1;
                PAGEOFFSET = PAGEOFFSET + RR * CC;
            }
        }
    }
}
```


Página 174

Este programa, escrito como una sola función, es un desastre. Tiene una estructura profundamente tura, una plétora de variables extrañas y una estructura estrechamente acoplada. Por lo menos, el

la función grande debe dividirse en algunas funciones más pequeñas.

El Listado 10-6 al Listado 10-8 muestra el resultado de dividir el código en el Listado 10-5 en clases y funciones más pequeñas, y eligiendo nombres significativos para esas clases, ciones y variables.

Listado 10-6

PrimePrinter.java (refactorizado)

```
paquete literatePrimes;

PrimePrinter de clase pública {
   public static void main (String [] args) {
      final int NUMBER_OF_PRIMES = 1000;
      int [] primes = PrimeGenerator.generate (NUMBER_OF_PRIMES);

      final int ROWS_PER_PAGE = 50;
      final int COLUMNS_PER_PAGE = 4;
      RowColumnPagePrinter tablePrinter =
         nueva RowColumnPagePrinter (ROWS_PER_PAGE,
                                     COLUMNS_PER_PAGE,
                                     "El primero" + NUMBER_OF_PRIMES +
                                        " Números primos");

      tablePrinter.print (primos);
   }

}
```

Listado 10-7

RowColumnPagePrinter.java

```
paquete literatePrimes;

import java.io.PrintStream;

public class RowColumnPagePrinter {
   private int rowsPerPage;
   private int columnPerPage;
   private int numbersPerPage;
   private String pageHeader;
   PrintStream privado printStream;

   Public RowColumnPagePrinter (int rowsPerPage,
                                int columnPerPage,
                                String pageHeader) {
      this.rowsPerPage = rowsPerPage;
      this.columnsPerPage = columnasPerPage;
      this.pageHeader = pageHeader;
      númerosPerPágina = filasPerPágina * columnasPerPágina;
      printStream = System.out;
   }
```


Página 175

Listado 10-7 (continuación)

RowColumnPagePrinter.java

```
   public void print (int data []) {
      int pageNumber = 1;
      para (int firstIndexOnPage = 0;
               firstIndexOnPage <data.length;
               firstIndexOnPage + = numbersPerPage) {
         int lastIndexOnPage =
            Math.min (firstIndexOnPage + numbersPerPage - 1,
                      longitud.datos - 1);
         printPageHeader (pageHeader, pageNumber);
         printPage (firstIndexOnPage, lastIndexOnPage, datos);
         printStream.println ("\ f");
         pageNumber ++;
      }
   }

   printPage vacío privado (int firstIndexOnPage,
```

```
                                        int lastIndexOnPage,
                                        int [] datos) {
        int firstIndexOfLastRowOnPage =
            firstIndexOnPage + rowsPerPage - 1;
        for (int firstIndexInRow = firstIndexOnPage;
                firstIndexInRow <= firstIndexOfLastRowOnPage;
                firstIndexInRow ++) {
            printRow (firstIndexInRow, lastIndexOnPage, datos);
            printStream.println ("");
        }
    }

    printRow vacío privado (int firstIndexInRow,
                                        int lastIndexOnPage,
                                        int [] datos) {
        para (int columna = 0; columna <columnasPágina; columna ++) {
            int índice = firstIndexInRow + columna * rowsPerPage;
            si (índice <= lastIndexOnPage)
                printStream.format ("% 10d", datos [índice]);
        }
    }

    printPageHeader vacío privado (String pageHeader,
                                                int pageNumber) {
        printStream.println (pageHeader + "--- Page" + pageNumber);
        printStream.println ("");
    }

    public void setOutput (PrintStream printStream) {
        this.printStream = printStream;
    }
}
```


Página 176

Listado 10-8

PrimeGenerator.java

```
paquete literatePrimes;

import java.util.ArrayList;

PrimeGenerator de clase pública {
    primos privados estáticos int [];
    private static ArrayList <Integer> multiplesOfPrimeFactors;

    protegido estático int [] generate (int n) {
        primos = nuevo int [n];
        multiplesOfPrimeFactors = new ArrayList <Integer> ();
        set2AsFirstPrime ();
        checkOddNumbersForSubsequentPrimes ();
        devolver primos;
    }

    vacío estático privado set2AsFirstPrime () {
        primos [0] = 2;
        multiplesOfPrimeFactors.add (2);
    }

    cheque vacío estático privadoOddNumbersForSubsequentPrimes () {
        int primeIndex = 1;
        para (int candidato = 3;
                primeIndex <primes.length;
                candidato + = 2) {
            if (isPrime (candidato))
                primos [primeIndex ++] = candidato;
        }
    }

    isPrime (int candidato) booleano estático privado {
```

```
        if (isLeastRelevantMultipleOfNextLargerPrimeFactor (candidato)) {
            multiplesOfPrimeFactors.add (candidato);
            falso retorno;
        }
        return isNotMultipleOfAnyPreviousPrimeFactor (candidato);
    }

    booleano estático privado
    isLeastRelevantMultipleOfNextLargerPrimeFactor (int candidato) {
        int nextLargerPrimeFactor = primes [multiplesOfPrimeFactors.size ()];
        int minimumRelevantMultiple = nextLargerPrimeFactor * nextLargerPrimeFactor;
        return candidato == LessRelevantMultiple;
    }

    booleano estático privado
    isNotMultipleOfAnyPreviousPrimeFactor (int candidato) {
        para (int n = 1; n <multiplesOfPrimeFactors.size (); n ++) {
            if (isMultipleOfNthPrimeFactor (candidato, n))
                falso retorno;
        }
```


Página 177

Listado 10-8 (continuación)

PrimeGenerator.java

```
        devuelve verdadero;
    }

    booleano estático privado
    isMultipleOfNthPrimeFactor (int candidato, int n) {
        regreso
            candidato == smallestOddNthMultipleNotLessThanCandidate (candidato, n);
    }

    int estático privado
    smallestOddNthMultipleNotLessThanCandidate (int candidato, int n) {
        int multiple = multiplesOfPrimeFactors.get (n);
        while (multiple <candidato)
            múltiples + = 2 * primos [n];
        multiplesOfPrimeFactors.set (n, multiple);
        devolver múltiples;
    }
}
```

Lo primero que notará es que el programa se alargó mucho más. Pasó de un poco más de una página a casi tres páginas de extensión. Hay varias razones para esto crecimiento. Primero, el programa refactorizado usa nombres de variables más largos y descriptivos. En segundo lugar, el programa refactorizado utiliza declaraciones de funciones y clases como una forma de agregar comentario al código. En tercer lugar, utilizamos espacios en blanco y técnicas de formato para mantener el programa es legible.

Observe cómo el programa se ha dividido en tres responsabilidades principales. El principal El programa está contenido en la clase PrimePrinter por sí mismo. Su responsabilidad es manejar el entorno de ejecución. Cambiará si cambia el método de invocación. Para Por ejemplo, si este programa se convirtiera en un servicio SOAP, esta es la clase que Ser afectado.

El RowColumnPagePrinter sabe todo acerca de cómo dar formato a una lista de números en páginas con un cierto número de filas y columnas. Si es necesario formatear la salida cambiando, entonces esta es la clase que se vería afectada.

La clase PrimeGenerator sabe cómo generar una lista de números primos. Note que es no está destinado a ser instanciado como un objeto. La clase es solo un ámbito útil en el que sus variables pueden declararse y mantenerse ocultas. Esta clase cambiará si el algoritmo para calcular cambios en los números primos.

¡Esto no fue una reescritura! No empezamos de cero y escribimos el programa de nuevo. de nuevo. De hecho, si observa detenidamente los dos programas diferentes, verá que utilizan la

mismo algoritmo y mecánica para realizar su trabajo.

El cambio se realizó escribiendo un conjunto de pruebas que verificaba el comportamiento *preciso* del primer programa. Luego se hicieron una gran cantidad de pequeños cambios, uno a la vez. Después de cada cambiar el programa se ejecutó para asegurarse de que el comportamiento no había cambiado. Uno diminuto paso tras otro, el primer programa se limpió y se transformó en el segundo.

Página 178

Organizándose para el cambio

Para la mayoría de los sistemas, el cambio es continuo. Todo cambio nos somete al riesgo de que el el resto del sistema ya no funciona como se esperaba. En un sistema limpio organizamos nuestras clases para reducir el riesgo de cambio.

La clase Sql en el Listado 10-9 se usa para generar cadenas SQL formadas correctamente dadas metadatos apropiados. Es un trabajo en progreso y, como tal, aún no es compatible con la función SQL. funcionalidad como declaraciones de actualización . Cuando llegue el momento de que la clase Sql admita un declaración de actualización , tendremos que "abrir" esta clase para hacer modificaciones. El problema con la apertura de una clase es que introduce riesgo. Cualquier modificación a la clase tiene la potencial de romper otro código en la clase. Debe volver a probarse por completo.

Listado 10-9

Una clase que debe abrirse al cambio.

```
public class Sql {
    public Sql (tabla de cadenas, columnas de columna [])
    cadena pública create ()
    inserción de cadena pública (campos Objeto [])
    public String selectAll ()
    public String findByKey (String keyColumn, String keyValue)
    selección de cadena pública (columna de columna, patrón de cadena)
    selección de cadena pública (criterios de criterios)
    public String preparadoInsert ()
    Private String columnList (Columna [] columnas)
    Private String valuesList (campos Objeto [], Columna final [] columnas)
    private String selectWithCriteria (criterios de cadena)
    Private String placeholderList (Columna [] columnas)
}
```

La clase Sql debe cambiar cuando agregamos un nuevo tipo de declaración. También debe cambiar cuando modificamos los detalles de un solo tipo de declaración, por ejemplo, si necesitamos modificar la funcionalidad de selección para admitir subselecciones. Estas dos razones para cambiar significan que el La clase sql viola el SRP.

Podemos detectar esta violación de SRP desde un punto de vista organizativo simple. El método El esquema de Sql muestra que existen métodos privados, como selectWithCriteria , que parecen relacionarse solo con declaraciones seleccionadas .

El comportamiento del método privado que se aplica solo a un pequeño subconjunto de una clase puede ser útil heurística para detectar áreas potenciales de mejora. Sin embargo, el estímulo principal para tomar La acción debe ser el cambio de sistema en sí mismo. Si la clase Sql se considera lógicamente completa, entonces no debemos preocuparnos por separar las responsabilidades. Si no necesitaremos una actualización funcionalidad para el futuro previsible, entonces deberíamos dejar Sql solo. Pero tan pronto como nos encontramos abriendo una clase, deberíamos considerar arreglar nuestro diseño.

¿Qué pasa si consideramos una solución como esa en el Listado 10-10? Cada interfaz pública El método definido en el Sql anterior del Listado 10-9 se refactoriza a su propia derivada de la clase Sql . Tenga en cuenta que los métodos privados, como valuesList , se mueven directamente donde

Página 179

son necesarios. El comportamiento privado común se aísla a un par de clases de servicios públicos, donde y ColumnList .

Listado 10-10

Un conjunto de clases cerradas

```
clase pública abstracta Sql {
    public Sql (tabla de cadenas, columnas de columna [])
    Cadena pública abstracta generate ();
}

La clase pública CreateSql extiende Sql {
    Public CreateSql (tabla de cadenas, columnas [] columnas)
    @Override public String generate ()
}

La clase pública SelectSql extiende Sql {
    Public SelectSql (tabla de cadenas, columnas Column [])
    @Override public String generate ()
}

La clase pública InsertSql extiende Sql {
    Public InsertSql (tabla de cadenas, columnas de columna [], campos de objeto [])
    @Override public String generate ()
    Private String valuesList (campos Objeto [], Columna final [] columnas)
}

La clase pública SelectWithCriteriaSql extiende Sql {
    public SelectWithCriteriaSql (
        Tabla de cadenas, Columna [] columnas, Criterios de criterios)
    @Override public String generate ()
}

La clase pública SelectWithMatchSql extiende Sql {
    public SelectWithMatchSql (
        Tabla de cadenas, Columna [] columnas, Columna de columna, Patrón de cadena)
    @Override public String generate ()
}

la clase pública FindByKeySql extiende Sql
    public FindByKeySql (
        Tabla de cadenas, Columna [] columnas, Cadena keyColumn, Cadena keyValue)
    @Override public String generate ()
}

La clase pública PreparedInsertSql extiende Sql {
    public PreparedInsertSql (tabla de cadenas, columnas Column [])
    @Override public String generate () {
    Private String placeholderList (Columna [] columnas)
}

clase pública Donde {
    public Where (criterios de cadena)
    Cadena pública generate ()
}
```


Página 180

Listado 10-10 (continuación)

Un conjunto de clases cerradas

```
public class ColumnList {
    Public ColumnList (Columna [] columnas)
    Cadena pública generate ()
}
```

El código de cada clase se vuelve insoportablemente simple. Nuestra comprensión requerida el tiempo para comprender cualquier clase se reduce a casi nada. El riesgo de que una función romper otro se vuelve cada vez más pequeño. Desde el punto de vista de la prueba, se vuelve más fácil tarea para probar todos los bits de lógica en esta solución, ya que las clases están todas aisladas de una otro.

De igual importancia, cuando llega el momento de agregar las declaraciones de actualización , ninguna de las ¡las clases necesitan un cambio! Codificamos la lógica para construir declaraciones de actualización en una nueva subclase de Sql llamado UpdateSql . Ningún otro código del sistema se romperá debido a este cambio.

Nuestra lógica Sql reestructurada representa lo mejor de todos los mundos. Es compatible con el SRP. También admite otro principio clave de diseño de la clase OO conocido como el principio abierto-cerrado, o OCP: [4] Las clases deben estar abiertas para extensión pero cerradas para modificaciones. Nuestro reestructurado La clase SQL está abierta para permitir nuevas funciones a través de subclases, pero podemos hacer este cambio. manteniendo todas las demás clases cerradas. Simplemente colocamos nuestra clase UpdateSql en su lugar.

Queremos estructurar nuestros sistemas de tal manera que hagamos lo mínimo posible cuando actualícelos con funciones nuevas o modificadas. En un sistema ideal, incorporamos nuevas características tures extendiendo el sistema, no haciendo modificaciones al código existente.

Aislamiento del cambio

Las necesidades cambiarán, por lo tanto, el código cambiará. Aprendimos en OO 101 que hay con- clases de creta, que contienen detalles de implementación (código) y clases abstractas, que representan conceptos solamente. Una clase de cliente que depende de detalles concretos está en riesgo cuando esos detalles cambian. Podemos introducir interfaces y clases abstractas para ayudar a aislar el impacto de esos detalles.

Las dependencias de detalles concretos crean desafíos para probar nuestro sistema. Si eran crear una clase de cartera y depende de una API externa de TokyoStockExchange para derivar el valor de la cartera, nuestros casos de prueba se ven afectados por la volatilidad de dicha búsqueda. ¡Es difícil escribir una prueba cuando obtenemos una respuesta diferente cada cinco minutos!

En lugar de diseñar Portfolio para que dependa directamente de TokyoStockExchange , creamos una interfaz, StockExchange , que declara un solo método:

```
interfaz pública StockExchange {
    Money currentPrice (símbolo de cadena);
}
```

4. [PPP].

Página 181

Diseñamos TokyoStockExchange para implementar esta interfaz. También nos aseguramos de que El constructor de Portfolio toma una referencia de StockExchange como argumento:

```
cartera pública {
    Bolsa de valores privada;
    Cartera pública (bolsa de valores) {
        this.exchange = intercambio;
    }
    // ...
}
```

Ahora nuestra prueba puede crear una implementación comprobable de la interfaz de StockExchange que emula el TokyoStockExchange. Esta implementación de prueba fijará el valor actual para cualquier símbolo que usemos en las pruebas. Si nuestra prueba demuestra la compra de cinco acciones de Microsoft para nuestra cartera, codificamos la implementación de prueba para que siempre devuelva $ 100 por acción de Microsoft. Nuestra implementación de prueba de la interfaz de StockExchange se reduce a un simple búsqueda de tabla. Luego, podemos escribir una prueba que espere $ 500 por el valor total de nuestra cartera.

```
Public class PortfolioTest {
    intercambio privado FixedStockExchangeStub;
    cartera de cartera privada;

    @Antes
    protegido vacío setUp () arroja Excepción {
        exchange = new FixedStockExchangeStub ();
        exchange.fix ("MSFT", 100);
        cartera = nueva cartera (intercambio);
    }

    @Prueba
    public void GivenFiveMSFTTotalShouldBe500 () lanza Exception {
        portfolio.add (5, "MSFT");
        Assert.assertEquals (500, portfolio.value ());
    }
}
```

Si un sistema está lo suficientemente desacoplado para ser probado de esta manera, también será más flexible y promover una mayor reutilización. La falta de acoplamiento significa que los elementos de nuestro sistema son mejor aislados unos de otros y del cambio. Este aislamiento facilita la comprensión soportar cada elemento del sistema.

Al minimizar el acoplamiento de esta manera, nuestras clases se adhieren a otro principio de diseño de clases. ple conocido como el Principio de Inversión de Dependencia (DIP). [5] En esencia, el DIP dice que nuestro las clases deberían depender de abstracciones, no de detalles concretos.

En lugar de depender de los detalles de implementación del TokyoStock-

Clase Exchange , nuestra clase Portfolio ahora depende de la interfaz StockExchange . La interfaz de StockExchange representa el concepto abstracto de preguntar por el precio actual de un símbolo. Esta abstracción aísla todos los detalles específicos de la obtención de dicho precio, incluso de dónde se obtiene ese precio.

5. [PPP].

Página 182

Bibliografía

[RDD]: *Diseño de objetos: roles, responsabilidades y colaboraciones* , Rebecca Wirfs-Brock y col., Addison-Wesley, 2002.

[PPP]: *Desarrollo de software ágil: principios, patrones y prácticas* , Robert C. Martin, Prentice Hall, 2002.

[Knuth92]: *Programación alfabetizada,* Donald E. Knuth, Centro para el estudio del lenguaje e información, Leland Stanford Junior University, 1992.

Página 183

Esta página se dejó en blanco intencionalmente

11

Sistemas

por el Dr. Kevin Dean Wampler

"La complejidad mata. Les quita la vida a los desarrolladores,
hace que los productos sean difíciles de planificar, construir y probar ".

—Ray Ozzie, director de tecnología de Microsoft Corporation

¿Cómo construirías una ciudad?

¿Podrías gestionar todos los detalles tú mismo? Probablemente no. Incluso gestionando una ciudad existente es demasiado para una persona. Sin embargo, las ciudades funcionan (la mayor parte del tiempo). Funcionan porque las ciudades tener equipos de personas que administran partes particulares de la ciudad, los sistemas de agua, la energía sistemas, tráfico, aplicación de la ley, códigos de construcción, etc. Algunas de esas personas son responsable del *panorama general* , mientras que otros se centran en los detalles.

Las ciudades también funcionan porque han desarrollado niveles apropiados de abstracción y modernización. ularidad que hacen posible que las personas y los "componentes" que logran funcionen efectivamente, incluso sin comprender el panorama general.

Aunque los equipos de software a menudo también se organizan así, los sistemas en los que trabajan a menudo no tienen la misma separación de preocupaciones y niveles de abstracción. Código limpio nos ayuda a lograr esto en los niveles más bajos de abstracción. En este capítulo, consideremos cómo para mantenerse limpio en niveles más altos de abstracción, el nivel del *sistema* .

Separe la construcción de un sistema de su uso

Primero, considere que la *construcción* es un proceso muy diferente al *uso* . Mientras escribo esto, Hay un nuevo hotel en construcción que veo por mi ventana en Chicago. Hoy es una caja de hormigón desnudo con una grúa de construcción y un ascensor atornillado al exterior. La la gente ocupada lleva cascos y ropa de trabajo. En un año más o menos, el hotel estará terminado. La grúa y el ascensor se habrán ido. El edificio estará limpio, encerrado en Paredes de ventana de vidrio y pintura atractiva. La gente que trabaja y se queda allí buscará muy diferente también.

> *Los sistemas de software deben separar el proceso de inicio, cuando los objetos de la aplicación están construido y las dependencias están "cableadas" juntas, a partir de la lógica de tiempo de ejecución que toma después del inicio.*

El proceso de inicio es una *preocupación* que cualquier aplicación debe abordar. Es el primer *con- preocupación* que examinaremos en este capítulo. La *separación de preocupaciones* es una de las más antiguas y técnicas de diseño más importantes en nuestro oficio.

Desafortunadamente, la mayoría de las aplicaciones no separan esta preocupación. El código para la puesta en marcha El proceso es ad hoc y está mezclado con la lógica de ejecución. Aquí está un ejemplo típico:

```
public Service getService () {
    si (servicio == nulo)
        service = new MyServiceImpl (...); // ¿Lo suficientemente bueno por defecto para la mayoría de los casos?
    servicio de devolución;
}
```

Este es el modismo LAZY INITIALIZATION / EVALUATION , y tiene varios méritos. Nosotros no incurra en los gastos generales de construcción a menos que realmente usemos el objeto, y nuestra puesta en marcha los tiempos pueden ser más rápidos como resultado. También nos aseguramos de que nunca se devuelva un valor nulo .

Sin embargo, ahora tenemos una dependencia codificada en MyServiceImpl y todo lo que constructor requiere (que he elidido). No podemos compilar sin resolver estos dependencias, incluso si nunca usamos un objeto de este tipo en tiempo de ejecución.

Las pruebas pueden ser un problema. Si MyServiceImpl es un objeto pesado, necesitaremos Asegúrese de que una adecuada T EST D OBLE 1 o M OCK O bject se asigna a la service campo antes de que se llame a este método durante la prueba unitaria. Porque tenemos construccion lógica mezclada con el procesamiento normal en tiempo de ejecución, deberíamos probar todas las rutas de ejecución (para ejemplo, la prueba nula y su bloque). Tener ambas responsabilidades significa que el El método está haciendo más de una cosa, por lo que estamos rompiendo el *principio de responsabilidad única* a pequeña escala.

Quizás lo peor de todo es que no sabemos si MyServiceImpl es el objeto correcto en todos casos. Lo insinué tanto en el comentario. ¿Por qué la clase con este método tiene que ¿Conoce el contexto global? Podemos *nunca* saber realmente el objeto derecho a utilizar aquí? Es incluso ¿Es posible que un tipo sea el adecuado para todos los contextos posibles?

Una aparición de LAZY - INICIALIZACIÓN no es un problema grave, por supuesto. Sin emabargo, normalmente hay muchos casos de pequeños modismos de configuración como este en las aplicaciones. Por eso, la *estrategia de* configuración global (si hay una) está *dispersa* en la aplicación, con poca modularidad y, a menudo, una duplicación significativa.

Si somos *diligentes* en la construcción de sistemas robustos y bien formados, nunca debemos permitir los modismos pequeños y *convenientes* conducen a la ruptura de la modularidad. El proceso de inicio de la con- La estructura y el cableado no son una excepción. Debemos modularizar este proceso por separado de la lógica de tiempo de ejecución normal y debemos asegurarnos de tener una estrategia global y coherente egy para resolver nuestras principales dependencias.

Separación de Main

Una forma de separar la construcción del uso es simplemente trasladar todos los aspectos de la construcción a main , o módulos llamados por main , y diseñar el resto del sistema asumiendo que todos Los objetos se han construido y cableado de forma adecuada. (Vea la Figura 11-1.)

El flujo de control es fácil de seguir. La función principal construye los objetos necesarios. para el sistema, luego los pasa a la aplicación, que simplemente los usa. Observe la dirección de las flechas de dependencia que cruzan la barrera entre la principal y la aplicación. Todos van en una dirección, apuntando en dirección opuesta a la principal. Esto significa que la aplicación no tiene conocimiento del principal o del proceso de construcción. Simplemente espera que todo tenga ha sido construido correctamente.

Suerte

A veces, por supuesto, necesitamos hacer que la aplicación sea responsable de *cuando* un objeto se creado. Por ejemplo, en un sistema de procesamiento de pedidos, la aplicación debe crear el

1. [Mezzaros07].

Página 187

Figura 11-1

Separación de la construcción en main ()

Instancias de LineItem para agregar a un pedido . En este caso podemos utilizar la A ESUMEN F actory [2] patrón para dar a la aplicación el control de *cuándo* construir los elementos de línea , pero mantener los detalles de esa construcción separada del código de la aplicación. (Vea la Figura 11-2.)

Figura 11-2

Construcción de separación con fábrica

Una vez más, observe que todas las dependencias apuntan desde la principal hacia el Procesamiento de pedidos. solicitud. Esto significa que la aplicación está desvinculada de los detalles de cómo construir un LineItem . Esa capacidad se mantiene en LineItemFactoryImplementation , que está en el lado principal de la línea. Y, sin embargo, la aplicación tiene el control total de cuándo las instancias de LineItem se compilan e incluso pueden proporcionar un constructor específico de la aplicación argumentos.

2. [GOF].

Página 188

Inyección de dependencia

Un poderoso mecanismo para separar la construcción del uso es la *inyección de dependencia* (DI), la aplicación de *Inversión de Control* (IoC) a la gestión de dependencias. [3] Inversión de El control traslada las responsabilidades secundarias de un objeto a otros objetos que están dedicados al propósito, apoyando así el *Principio de Responsabilidad Única.* En el contexto de gestión de dependencias, un objeto no debe asumir la responsabilidad de instanciar dependencias dencies sí mismo. En cambio, debería pasar esta responsabilidad a otro mecanismo "autorizado" nismo, invirtiendo así el control. Debido a que la configuración es una preocupación global, esta autoridad Por lo general, el mecanismo será la rutina "principal" o un *contenedor de* propósito especial .

Las búsquedas JNDI son una implementación "parcial" de DI, donde un objeto pregunta a un directorio servidor para proporcionar un "servicio" que coincida con un nombre en particular.

```
MyService myService = (MyService) (jndiContext.lookup (“NameOfMyService”));
```

El objeto que invoca no controla qué tipo de objeto se devuelve realmente (siempre que

implementa la interfaz apropiada, por supuesto), pero el objeto que invoca todavía resuelve la dependencia.

True Dependency Injection va un paso más allá. La clase no toma pasos directos para resolver sus dependencias; es completamente pasivo. En su lugar, proporciona métodos de establecimiento o argumentos del constructor (o ambos) que se utilizan para *inyectar* las dependencias. Durante la conferencia proceso de construcción, el contenedor DI crea una instancia de los objetos requeridos (generalmente a pedido) y utiliza los argumentos del constructor o los métodos de establecimiento proporcionados para conectar las dependencias dencies. Los objetos dependientes que se utilizan realmente se especifican mediante una configuración. archivo o mediante programación en un módulo de construcción de propósito especial.

Spring Framework proporciona el contenedor DI más conocido para Java. [4] Tú defines qué objetos conectar juntos en un archivo de configuración XML, luego solicita una objetos por nombre en código Java. Veremos un ejemplo en breve.

Pero, ¿qué hay de las virtudes de LAZY - INICIALIZACIÓN ? Este modismo todavía es a veces útil con DI. Primero, la mayoría de los contenedores DI no construirán un objeto hasta que se necesiten. Segundo, muchos de estos contenedores proporcionan mecanismos para invocar fábricas o para construir proxies, que podrían usarse para LAZY - EVALUATION y *optimizaciones* similares . [5]

Ampliar

Las ciudades crecen a partir de pueblos, que crecen a partir de asentamientos. Al principio, las carreteras son estrechas y prácticamente inexistentes, luego se pavimentan, luego se ensanchan con el tiempo. Pequeños edificios y

3. Ver, por ejemplo, [Fowler].
4. Ver [Primavera]. También hay un marco Spring.NET.
5. ¡No olvide que la instanciación / evaluación perezosa es solo una optimización y quizás prematura!

Página 189

las parcelas vacías están llenas de edificios más grandes, algunos de los cuales eventualmente serán reemplazados con rascacielos.

Al principio no hay servicios como luz, agua, alcantarillado e Internet (¡jadeo!). Estas Los servicios también se agregan a medida que aumenta la población y la densidad de edificios.

Este crecimiento no está exento de dolor. ¿Cuántas veces has conducido de parachoques a parachoques? a través de un proyecto de "mejora" de carreteras y se preguntó: "¿Por qué no lo construyeron ¿¡Suficiente la primera vez !? "

Pero no podría haber sucedido de otra manera. ¿Quién puede justificar el gasto de seis ¿Carretera de carril a través del medio de una pequeña ciudad que anticipa crecimiento? Quién podría *¿Quieres* un camino así a través de su ciudad?

Es un mito que podemos hacer que los sistemas "sean correctos a la primera". En cambio, deberíamos implementar Mente solo las *historias* de hoy , luego refactorice y expanda el sistema para implementar nuevas historias. mañana. Esta es la esencia de la agilidad iterativa e incremental. Desarrollo impulsado por pruebas ment, la refactorización y el código limpio que producen hacen que esto funcione a nivel de código.

Pero, ¿qué pasa a nivel del sistema? ¿No requiere la arquitectura del sistema una planificación previa? ning? Ciertamente, *no* puede crecer gradualmente de simple a complejo , ¿verdad?

> *Los sistemas de software son únicos en comparación con los sistemas físicos. Sus arquitecturas pueden crecer incrementalmente,* ***si*** *mantenemos la adecuada separación de preocupaciones.*

La naturaleza efímera de los sistemas de software lo hace posible, como veremos. Vamos primero considere un contraejemplo de una arquitectura que no separa las preocupaciones de manera adecuada.

Las arquitecturas EJB1 y EJB2 originales no separaron las preocupaciones de manera adecuada y impuso así barreras innecesarias al crecimiento orgánico. Considere un *Entity Bean* para un clase de banco persistente . Un bean de entidad es una representación en memoria de datos relacionales, en otras palabras, una fila de la tabla.

Primero, tenía que definir una interfaz local (en proceso) o remota (JVM separada), que los clientes usarían. El Listado 11-1 muestra una posible interfaz local:

Listado 11-1

Una interfaz local EJB2 para un banco EJB

```
paquete com.example.banking;
import java.util.Collections;
importar javax.ejb. *;

La interfaz pública BankLocal extiende java.ejb.EJBLocalObject {
    String getStreetAddr1 () lanza EJBException;
    String getStreetAddr2 () lanza EJBException;
    String getCity () lanza EJBException;
    String getState () lanza EJBException;
    String getZipCode () lanza EJBException;
    void setStreetAddr1 (String street1) lanza EJBException;
    void setStreetAddr2 (String street2) lanza EJBException;
    void setCity (String city) lanza EJBException;
    void setState (estado de cadena) lanza EJBException;
```


Página 190

Listado 11-1 (continuación)

Una interfaz local EJB2 para un banco EJB

```
    void setZipCode (String zip) lanza EJBException;
    La colección getAccounts () lanza EJBException;
    void setAccounts (cuentas de colección) lanza EJBException;
    void addAccount (AccountDTO accountDTO) lanza EJBException;
}
```

He mostrado varios atributos para la dirección del banco y una colección de cuentas que el banco es propietario, cada uno de los cuales tendría sus datos manejados por una cuenta separada EJB. El Listado 11-2 muestra la clase de implementación correspondiente para el bean Bank .

Listado 11-2

La implementación de EJB2 Entity Bean correspondiente

```
paquete com.example.banking;
import java.util.Collections;
importar javax.ejb. *;

El banco público de clases abstractas implementa javax.ejb.EntityBean {
    // Lógica de negocios...
    public abstract String getStreetAddr1 ();
    public abstract String getStreetAddr2 ();
    public abstract String getCity ();
    public abstract String getState ();
    public abstract String getZipCode ();
    public abstract void setStreetAddr1 (String street1);
    public abstract void setStreetAddr2 (String street2);
    public abstract void setCity (ciudad de cadenas);
    public abstract void setState (estado de cadena);
    public abstract void setZipCode (String zip);
    getAccounts () de la colección pública abstracta;
    public abstract void setAccounts (cuentas de colección);
    public void addAccount (AccountDTO accountDTO) {
        Contexto InitialContext = nuevo InitialContext ();
        AccountHomeLocal accountHome = context.lookup ("AccountHomeLocal");
        AccountLocal account = accountHome.create (accountDTO);
        Cuentas de cobro = getAccounts ();
        cuentas.add (cuenta);
    }
    // Lógica del contenedor EJB
    public abstract void setId (ID de número entero);
    public abstract Integer getId ();
    public Integer ejbCreate (ID de entero) {...}
    public void ejbPostCreate (ID de número entero) {...}
    // El resto tuvo que implementarse pero generalmente estaban vacíos:
    public void setEntityContext (EntityContext ctx) {}
    public void unsetEntityContext () {}
    public void ejbActivate () {}
    public void ejbPassivate () {}
```

```
    public void ejbLoad () {}
    public void ejbStore () {}
    public void ejbRemove () {}
}
```


Página 191

No he mostrado la interfaz *LocalHome* correspondiente , esencialmente una fábrica utilizada para crear objetos, ni ninguno de los posibles métodos de búsqueda de bancos (consulta) que pueda agregar.

Finalmente, tuvo que escribir uno o más descriptores de implementación XML que especifiquen el detalles de mapeo relacional de objetos a un almacén de persistencia, el comportamiento transaccional deseado, restricciones de seguridad, y así sucesivamente .

La lógica empresarial está estrechamente acoplada al "contenedor" de la aplicación EJB2. Debes tipos de contenedores de subclase y debe proporcionar muchos métodos de ciclo de vida que son necesarios por el contenedor.

Debido a este acoplamiento al contenedor de peso pesado, las pruebas de unidades aisladas son difíciles. Es necesario burlarse del contenedor, que es duro, o perder mucho tiempo desplegándolo EJB y pruebas a un servidor real. La reutilización fuera de la arquitectura EJB2 es efectivamente imposible. sible, debido al acoplamiento apretado.

Finalmente, incluso la programación orientada a objetos se ve socavada. Un frijol no puede heredar de otro frijol. Observe la lógica para agregar una nueva cuenta. Es común en los beans EJB2. para definir "objetos de transferencia de datos" (DTO) que son esencialmente "estructuras" sin comportamiento. Esto generalmente conduce a tipos redundantes que contienen esencialmente los mismos datos, y requiere código repetitivo para copiar datos de un objeto a otro.

Preocupaciones transversales

La arquitectura EJB2 se acerca a la verdadera separación de preocupaciones en algunas áreas. Para Por ejemplo, los comportamientos transaccionales, de seguridad y de persistencia deseados son declarado en los descriptores de despliegue, independientemente del código fuente.

Tenga en cuenta que *preocupaciones* como la persistencia tienden a traspasar los límites de los objetos naturales de un dominio. Desea conservar todos sus objetos utilizando generalmente la misma estrategia, por ejemplo ple , usando un DBMS [6] particular versus archivos planos, siguiendo ciertas convenciones de nomenclatura para tablas y columnas, utilizando la semántica transaccional consistentes, y así sucesivamente .

En principio, puede razonar sobre su estrategia de persistencia en un módulo encapsulado camino. Sin embargo, en la práctica, debe difundir esencialmente el mismo código que implementa la persistencia Tence estrategia a través de muchos objetos. Usamos el término *preocupaciones transversales* para preocupaciones como estas. Una vez más, el marco de persistencia podría ser modular y nuestra lógica de dominio, en aislamiento ción, podría ser modular. El problema es la *intersección* detallada de estos dominios.

De hecho, la forma en que la arquitectura EJB manejó la persistencia, la seguridad y las transacciones , *Programación orientada a aspectos* (AOP) "anticipada" , [7] que es un enfoque de propósito general a restaurar la modularidad para las preocupaciones transversales.

En AOP, las construcciones modulares llamadas *aspectos* especifican qué puntos del sistema deben modificar su comportamiento de alguna manera coherente para apoyar una preocupación en particular. Esto La especificación se realiza mediante un mecanismo sucinto declarativo o programático.

6. Sistema de gestión de bases de datos.
7. Consulte [AOSD] para obtener información general sobre aspectos y [AspectJ]] y [Colyer] para obtener información específica de AspectJ.

Página 192

Usando la persistencia como ejemplo, declararía qué objetos y atributos (o *patrones de los* mismos) deben persistir y luego delegar las tareas de persistencia a su persistencia marco tence. Las modificaciones de comportamiento se realizan de forma *no invasiva* [8] en el código de destino por el marco AOP. Veamos tres aspectos o mecanismos de aspecto en Java.

Proxies de Java

Los proxies de Java son adecuados para situaciones simples, como encapsular llamadas a métodos en objetos o clases. Sin embargo, los proxies dinámicos proporcionados en el JDK solo funcionan con interfaces. Para las clases de proxy, debe usar una biblioteca de manipulación de código de bytes, como CGLIB, ASM o Javassist. [9]

El Listado 11-3 muestra el esqueleto de un proxy JDK para proporcionar soporte de persistencia para nuestra aplicación Bancaria , que cubre solo los métodos para obtener y configurar la lista de cuentas.

Listado 11-3

Ejemplo de proxy JDK

```
// Bank.java (suprimiendo los nombres de los paquetes ...)
importar java.utils. *;

// La abstracción de un banco.
Banco de interfaz pública {
    Colección <Cuenta> getAccounts ();
    void setAccounts (cuentas de colección <Cuenta>);
}

// BankImpl.java
importar java.utils. *;

// El "Objeto simple de Java antiguo" (POJO) implementando la abstracción.
BankImpl de clase pública implementa Bank {
    Lista privada de cuentas <Cuenta>;

    Colección pública <Cuenta> getAccounts () {
        cuentas de devolución;
    }
    public void setAccounts (Collection <Ccount> cuentas) {
        this.accounts = new ArrayList <Cuenta> ();
        para (Cuenta cuenta: cuentas) {
            this.accounts.add (cuenta);
        }
    }
}

// BankProxyHandler.java
import java.lang.reflect. *;
importar java.util. *;
```

8. Lo que significa que no se requiere edición manual del código fuente de destino.
9. Consulte [CGLIB], [ASM] y [Javassist].

Página 193

Listado 11-3 (continuación)

Ejemplo de proxy JDK

```
// "InvocationHandler" requerido por la API de proxy.
La clase pública BankProxyHandler implementa InvocationHandler {
    banco de banco privado;

    public BankHandler (banco bancario) {
        this.bank = banco;
    }

    // Método definido en InvocationHandler
    invocación de objeto público (proxy de objeto, método de método, argumentos de objeto [])
            lanza Throwable {
        String nombreMétodo = método.getName ();
        if (methodName.equals ("getAccounts")) {
            bank.setAccounts (getAccountsFromDatabase ());
            return bank.getAccounts ();
        } else if (methodName.equals ("setAccounts")) {
            bank.setAccounts ((Colección <Cuenta>) args [0]);
            setAccountsToDatabase (bank.getAccounts ());
            devolver nulo;
        } demás {
            ...
        }
    }

    // Muchos detalles aquí:
    Colección protegida <Cuenta> getAccountsFromDatabase () {...}
    protected void setAccountsToDatabase (Collection <Account> cuentas) {...}
}

// En algún otro lugar...

Banco banco = (Banco) Proxy.newProxyInstance (
    Bank.class.getClassLoader (),
    nueva clase [] {Bank.class},
    nuevo BankProxyHandler (nuevo BankImpl ()));
```

Definimos una interfaz Bank , que será *envuelta* por el proxy, y una *Plain-Old Objeto Java* (POJO), BankImpl , que implementa la lógica empresarial. (Volveremos a visitar los POJO dentro de poco.)

La API de proxy requiere un objeto InvocationHandler al que llama para implementar cualquier Llamadas al método bancario realizadas al proxy. Nuestro BankProxyHandler usa la reflexión de Java API para mapear las invocaciones de métodos genéricos a los métodos correspondientes en BankImpl , y así.

Hay *mucho* código aquí y es relativamente complicado, incluso para este simple caso. [10] El uso de una de las bibliotecas de manipulación de bytes es igualmente desafiante. Este código "volumen"

10. Para obtener ejemplos más detallados de la API de proxy y ejemplos de su uso, consulte, por ejemplo, [Goetz].

y la complejidad son dos de los inconvenientes de los proxies. Hacen que sea difícil crear limpio ¡código! Además, los proxies no proporcionan un mecanismo para especificar la ejecución en todo el sistema. "Puntos" de interés, que son necesarios para una verdadera solución AOP. [11]

Frameworks de Java puro AOP

Afortunadamente, la mayor parte del texto estándar de proxy se puede manejar automáticamente mediante herramientas. Proxies se utilizan internamente en varios marcos de Java, por ejemplo, Spring AOP y JBoss AOP, para implementar aspectos en Java puro. [12] En Spring, escribe su lógica de negocios como *Plain-Old*

Objetos Java . Los POJO se centran exclusivamente en su dominio. No tienen dependencias de marcos empresariales (o cualquier otro dominio). Por lo tanto, son conceptualmente más simples y más fácil de probar. La relativa simplicidad hace que sea más fácil asegurarse de que está implementando Mentar correctamente las historias de usuario correspondientes y mantener y desarrollar el código para historias futuras.

Incorpora la infraestructura de aplicaciones necesaria, incluidas las conexiones transversales. cerns como persistencia, transacciones, seguridad, almacenamiento en caché, conmutación por error, etc., utilizando declaraciones archivos de configuración activa o API. En muchos casos, en realidad está especificando Spring o Aspectos de la biblioteca JBoss, donde el marco maneja la mecánica del uso de proxies Java o bibliotecas de códigos de bytes de forma transparente para el usuario. Estas declaraciones impulsan la dependencia contenedor de inyección (DI), que instancia los objetos principales y los conecta en demanda .

El Listado 11-4 muestra un fragmento típico de un archivo de configuración Spring V2.5, app.xml [13] :

Listado 11-4

Archivo de configuración Spring 2.X

```
<frijoles>
    ...
    <bean id = "appDataSource"
     class = "org.apache.commons.dbcp.BasicDataSource"
     método-destruir = "cerrar"
     p: driverClassName = "com.mysql.jdbc.Driver"
     p: url = "jdbc: mysql: // localhost: 3306 / mydb"
     p: username = "yo" />

    <bean id = "bankDataAccessObject"
     class = "com.example.banking.persistence.BankDataAccessObject"
     p: dataSource-ref = "appDataSource" />

    <bean id = "banco"
```

11. AOP se confunde a veces con las técnicas que se utilizan para implementarlo, como la interceptación de métodos y la apoderados. El valor real de un sistema AOP es la capacidad de especificar comportamientos sistémicos de manera concisa y modular.
12. Consulte [Spring] y [JBoss]. "Java puro" significa sin el uso de AspectJ.
13. Adaptado de http : //www.theserverside.com/tt/articles/article.tss? L = IntrotoSpring25

Página 195

Listado 11-4 (continuación)

Archivo de configuración Spring 2.X

```
     class = "com.example.banking.model.Bank"
     p: dataAccessObject-ref = "bankDataAccessObject" />
    ...
</beans>
```

Cada "frijol" es como una parte de una "muñeca rusa" anidada, con un objeto de dominio para un Banco proxy (envuelto) por un objeto de acceso a datos (DAO), que a su vez es proxy Fuente de datos del controlador JDBC. (Vea la Figura 11-3.)

Figura 11-3
La "muñeca rusa" de los decoradores

El cliente cree que está invocando getAccounts () en un objeto Bank , pero en realidad es una conversación. al más exterior de un conjunto de objetos D ECORATOR [14] anidados que amplían el comportamiento básico del Banco POJO. Podríamos añadir otros decoradores para transacciones, almacenamiento en caché, y así sucesivamente .

En la aplicación, se necesitan algunas líneas para pedirle al contenedor DI el nivel superior objetos en el sistema, como se especifica en el archivo XML.

```
XmlBeanFactory bf =
    new XmlBeanFactory (new ClassPathResource ("app.xml", getClass ()));
Banco banco = (Banco) bf.getBean ("banco");
```

Debido a que se requieren tan pocas líneas de código Java específico de Spring, *la aplicación es casi completamente desacoplado de Spring* , eliminando todos los problemas de acoplamiento estrecho de los sistemas como EJB2.

Aunque XML puede ser detallado y difícil de leer, [15] la "política" especificada en estos Los archivos de figuración son más simples que la complicada lógica de proxy y aspecto que se oculta a ver y crear automáticamente. Este tipo de arquitectura es tan convincente que el marco funciona como Spring llevó a una revisión completa del estándar EJB para la versión 3. EJB3

14. [GOF].
15. El ejemplo se puede simplificar utilizando mecanismos que aprovechan la *convención sobre la configuración* y las anotaciones de Java 5 para reducir la cantidad de lógica de "cableado" explícita requerida.

Página 196

sigue en gran medida el modelo Spring de apoyar declarativamente preocupaciones transversales utilizando Archivos de configuración XML y / o anotaciones Java 5.

El Listado 11-5 muestra nuestro objeto Bank reescrito en EJB3 [16] .

Listado 11-5

Un EJB de EBJ3 Bank

```
paquete com.example.banking.model;
importar javax.persistence. *;
import java.util.ArrayList;
import java.util.Collection;

@Entidad
@Table (nombre = "BANCOS")
public class Bank implementa java.io.Serializable {
    @Id @GeneratedValue (estrategia = GenerationType.AUTO)
    identificación int privada;

    @Embeddable // Un objeto "en línea" en la fila de la base de datos del banco
    Dirección de clase pública {
        protected String streetAddr1;
        protected String streetAddr2;
        ciudad de cadena protegida;
        estado de cadena protegido;
        protected String zipCode;
    }

    @Incorporado
    dirección de dirección privada;

    @OneToMany (cascade = CascadeType.ALL, fetch = FetchType.EAGER,
                mappedBy = "banco")
    Colección privada <Cuenta> cuentas = new ArrayList <Cuenta> ();

    public int getId () {
        id de retorno;
    }

    public void setId (int id) {
        this.id = id;
    }

    public void addAccount (cuenta de cuenta) {
        account.setBank (esto);
```

```
        cuentas.add (cuenta);
    }

    Colección pública <Cuenta> getAccounts () {
        cuentas de devolución;
    }
```

16. Adaptado de http://www.onjava.com/pub/a/onjava/2006/05/17/standardizing-with-ejb3-java-persistence-api.html

Página 197

Listado 11-5 (continuación)
Un EJB de EBJ3 Bank

```
    public void setAccounts (Collection <Ccount> cuentas) {
        this.accounts = cuentas;
    }
}
```

Este código es mucho más limpio que el código EJB2 original. Algunos de los detalles de la entidad son todavía aquí, contenido en las anotaciones. Sin embargo, debido a que ninguna de esa información se lado de las anotaciones, el código es limpio, claro y, por lo tanto, fácil de probar, mantener y pronto.

Parte o toda la información de persistencia en las anotaciones se puede mover a XML descriptores de implementación, si se desea, dejando un POJO verdaderamente puro. Si el mapeo de persistencia los detalles no cambiarán con frecuencia, muchos equipos pueden optar por mantener las anotaciones, pero con muchos menos inconvenientes dañinos en comparación con la invasividad EJB2.

AspectJ Aspectos

Finalmente, la herramienta más completa para separar preocupaciones a través de aspectos es AspectJ lenguaje, [17] una extensión de Java que proporciona soporte de "primera clase" para aspectos como modular-construcciones de ity. Los enfoques de Java puro proporcionados por Spring AOP y JBoss AOP son suficientes suficiente para el 80-90 por ciento de los casos en los que los aspectos son más útiles. Sin embargo, AspectJ proporciona un conjunto de herramientas muy rico y poderoso para separar preocupaciones. El inconveniente de AspectJ es la necesidad de adoptar varias herramientas nuevas y aprender nuevas construcciones del lenguaje y modismos de uso.

Los problemas de adopción se han mitigado parcialmente con una “anotación "forma de" de AspectJ, donde las anotaciones de Java 5 se utilizan para definir aspectos utilizando Java puro código. Además, Spring Framework tiene una serie de características que hacen que la incorporación de Los aspectos basados en anotaciones son mucho más fáciles para un equipo con experiencia limitada en AspectJ.

Una discusión completa de AspectJ está más allá del alcance de este libro. Consulte [AspectJ], [Colyer], y [Primavera] para obtener más información.

Pruebe la arquitectura del sistema

No se puede exagerar el poder de separar las preocupaciones a través de enfoques similares a aspectos. Si puede escribir la lógica de dominio de su aplicación utilizando POJO, desacoplado de cualquier arquitectura ture preocupaciones a nivel de código, entonces es posible *probar* realmente su arquitectura. Tú puede evolucionarlo de simple a sofisticado, según sea necesario, mediante la adopción de nuevas tecnologías en

17. Consulte [AspectJ] y [Colyer].

Página 198

demanda. No es necesario hacer un *Big Design Up Front* [18] (BDUF). De hecho, BDUF es incluso dañino porque inhibe la adaptación al cambio, debido a la resistencia psicológica a la discardando el esfuerzo previo y debido a la forma en que las elecciones de arquitectura influyen pensando en el diseño.

Los arquitectos de edificios tienen que hacer BDUF porque no es factible hacer archi- Cambios tecturales en una gran estructura física una vez que la construcción está bien avanzada. [19] Aunque el software tiene su propia *física* , [20] es económicamente factible hacer radicales cambiar, *si* la estructura del software separa sus preocupaciones de manera efectiva.

Esto significa que podemos iniciar un proyecto de software con un "ingenuamente simple" pero muy bien decou- arquitectura pled, entregando historias de usuarios que trabajan rápidamente, luego agregando más infraestructura a medida que escalamos. Algunos de los sitios web más grandes del mundo han logrado una disponibilidad muy alta y rendimiento, utilizando sofisticado almacenamiento en caché de datos, seguridad, virtualización, etc. todo hecho de manera eficiente y flexible porque los diseños mínimamente acoplados están apropiadamente *simple* en cada nivel de abstracción y alcance.

Por supuesto, esto no significa que entremos en un proyecto "sin timón". Tenemos algo expectativas del alcance general, los objetivos y el cronograma del proyecto, así como el estructura general del sistema resultante. Sin embargo, debemos mantener la capacidad de cambiar curso en respuesta a circunstancias cambiantes.

La arquitectura EJB temprana es solo una de las muchas API conocidas que están sobrecargadas de ingeniería. necesario y que comprometen la separación de preocupaciones. Incluso las API bien diseñadas pueden superarse matar cuando no son realmente necesarios. Una buena API debería *desaparecer en* gran medida de la vista del tiempo, por lo que el equipo dedica la mayor parte de sus esfuerzos creativos a que el usuario Riesgos que se están implementando. De lo contrario, las restricciones arquitectónicas inhibirán la eficiencia entrega de valor óptimo al cliente.

Para recapitular esta larga discusión,

> *Una arquitectura de sistema óptima consta de dominios de interés modularizados, cada uno de los cuales se implementa con Plain Old Java (u otros) Objetos. Los diferentes dominios están integrados rallado junto con Aspectos mínimamente invasivos o herramientas similares a Aspectos. Esta arquitectura puede ser impulsado por pruebas, al igual que el código.*

Optimizar la toma de decisiones

La modularidad y la separación de preocupaciones hacen que la gestión y la toma de decisiones sean descentralizadas. haciendo posible. En un sistema suficientemente grande, ya sea una ciudad o un proyecto de software, no una persona puede tomar todas las decisiones.

18. No debe confundirse con la buena práctica del diseño inicial, BDUF es la práctica de diseñar *todo* por adelantado antes implementar cualquier cosa en absoluto.
19. Todavía hay una cantidad significativa de exploración iterativa y discusión de detalles, incluso después de que comienza la construcción.
20. El término *física del software* fue utilizado por primera vez por [Kolence].

Página 199

Todos sabemos que lo mejor es dar responsabilidades a las personas más calificadas. Nosotros a menudo olvídate de que también es mejor *posponer las decisiones hasta el último momento posible* . Esto no es perezoso o irresponsable; nos permite tomar decisiones informadas con la mejor información posible. Una decisión prematura es una decisión que se toma con un conocimiento subóptimo. Tendremos eso mucho menos comentarios de los clientes, reflexión mental sobre el proyecto y experiencia con nuestro opciones de implementación si decidimos demasiado pronto.

La agilidad que proporciona un sistema POJO con preocupaciones modularizadas nos permite optimizar decisiones erróneas, justo a tiempo, basadas en los conocimientos más recientes. La complejidad de estos las decisiones también se reducen.

Utilice los estándares con prudencia cuando *añadan un* valor *demostrable*

La construcción de edificios es una maravilla de ver debido al ritmo al que se están construyendo nuevos edificios. construido (incluso en pleno invierno) y debido a los extraordinarios diseños que son posibles ble con la tecnología actual. La construcción es una industria madura con piezas altamente optimizadas, métodos y estándares que han evolucionado bajo presión durante siglos.

Muchos equipos utilizaron la arquitectura EJB2 porque era un estándar, incluso cuando era más ligero. el peso y los diseños más sencillos habrían sido suficientes. He visto equipos obsesionarse con varios estándares *fuertemente promocionados* y perder el enfoque en la implementación valor para sus clientes.

Los estándares facilitan la reutilización de ideas y componentes, la contratación de personas con experiencia relevante experiencia, encapsular buenas ideas y conectar componentes. Sin embargo, el proceso de La creación de estándares a veces puede llevar demasiado tiempo para que la industria espere, y algunos estándares perder el contacto con las necesidades reales de los adoptantes a los que están destinados a servir.

Los sistemas necesitan lenguajes específicos de dominio

La construcción de edificios, como la mayoría de los dominios, ha desarrollado un lenguaje rico con un vocabulario lary, modismos y patrones [21] que transmiten información esencial de forma clara y concisa. En suave ware, ha habido un renovado interés recientemente en la creación *de lenguajes específicos de dominio* (DSL), [22] que son pequeños lenguajes de secuencias de comandos o API independientes en lenguajes estándar que Permitir que el código se escriba de modo que se lea como una forma estructurada de prosa que un dominio el experto podría escribir.

Un buen DSL minimiza la "brecha de comunicación" entre un concepto de dominio y el código que lo implementa, al igual que las prácticas ágiles optimizan las comunicaciones dentro de un equipo y con los stakeholders del proyecto. Si está implementando lógica de dominio en el

21. El trabajo de [Alexander] ha sido particularmente influyente en la comunidad del software.
22. Ver, por ejemplo, [DSL]. [JMock] es un buen ejemplo de una API de Java que crea un DSL.

Página 200

mismo lenguaje que usa un experto en el dominio, hay menos riesgo de que trans- tarde el dominio en la implementación.

Los DSL, cuando se utilizan de forma eficaz, elevan el nivel de abstracción por encima de los modismos y el diseño del código. patrones. Permiten al desarrollador revelar la intención del código en el nivel apropiado.

de abstracción.

> *Los lenguajes específicos de dominio permiten todos los niveles de abstracción y todos los dominios de la aplicación. que se exprese como POJO, desde la política de alto nivel hasta los detalles de bajo nivel.*

Conclusión

Los sistemas también deben estar limpios. Una arquitectura invasiva abruma la lógica del dominio y Impacta la agilidad. Cuando la lógica del dominio se oscurece, la calidad sufre porque los errores la encuentran más fácil de ocultar y las historias se vuelven más difíciles de implementar. Si la agilidad se ve comprometida, la produc- La actividad sufre y los beneficios de TDD se pierden.

En todos los niveles de abstracción, la intención debe ser clara. Esto solo sucederá si escribe POJO y usa mecanismos de aspecto para incorporar otra implementación preocupaciones de forma no invasiva.

Ya sea que esté diseñando sistemas o módulos individuales, nunca olvide *utilizar el cosa más simple que pueda funcionar* .

Bibliografía

[Alexander]: Christopher Alexander, *A Timeless Way of Building,* Universidad de Oxford Press, Nueva York, 1979.

[AOSD]: puerto de desarrollo de software orientado a aspectos, http://aosd.net

[ASM]: Página de inicio de ASM, http : //asm.objectweb.org/

[AspectJ]: http://eclipse.org/aspectj

[CGLIB]: Biblioteca de generación de código, http://cglib.sourceforge.net/

[Colyer]: Adrian Colyer, Andy Clement, George Hurley, Mathew Webster, *Eclipse AspectJ,* Person Education, Inc., Upper Saddle River, Nueva Jersey, 2005.

[DSL]: lenguaje de programación específico del dominio, http : //en.wikipedia.org/wiki/Domain-idioma_programación_específico

[Fowler]: Inversión de contenedores de control y el patrón de inyección de dependencia, http://martinfowler.com/articles/injection.html

Página 201

[Goetz]: Brian Goetz, *Teoría y práctica de Java: decoración con proxies dinámicos* , http://www.ibm.com/developerworks/java/library/j-jtp08305.html

[Javassist]: Página de inicio de Javassist, http://www.csg.is.titech.ac.jp/~chiba/javassist/

[JBoss]: Página de inicio de JBoss, http://jboss.org

[JMock]: JMock **: una** biblioteca ligera de objetos simulados para Java, http://jmock.org

[Kolence]: Kenneth W. Kolence, Física de software y medición del rendimiento informático mentos, *Actas de la conferencia anual de ACM — Volumen 2* , Boston, Massachusetts, págs. 1024-1040, 1972.

[Primavera]: *The Spring Framework* , http://www.springframework.org

[Mezzaros07]: *XUnit Patterns* , Gerard Mezzaros, Addison-Wesley, 2007.

[GOF]: *Patrones de diseño: elementos de software orientado a objetos reutilizables* , Gamma et al., Addison-Wesley, 1996.

12

Aparición

por Jeff Langr

Limpiar a través del diseño emergente

¿Qué pasaría si hubiera cuatro reglas simples que pudiera seguir y que lo ayudarían a crear buenas diseños como trabajaste? ¿Qué pasaría si siguiendo estas reglas obtuvieras conocimientos sobre la estructura? tura y diseño de su código, facilitando la aplicación de principios como SRP y DIP? ¿Y si estas cuatro reglas facilitaran la *aparición* de buenos diseños?

Muchos de nosotros creemos que las cuatro reglas del *diseño simple* [1] de Kent Beck son de gran ayuda para creando software bien diseñado.

1. [XPE].

Página 203

Según Kent, un diseño es "simple" si sigue estas reglas:

- Ejecuta todas las pruebas
- No contiene duplicaciones
- Expresa la intención del programador
- Minimiza el número de clases y métodos

Las reglas se dan por orden de importancia.

Regla de diseño simple 1: ejecuta todas las pruebas

En primer lugar, un diseño debe producir un sistema que actúe según lo previsto. Un sistema podría tienen un diseño perfecto en papel, pero si no hay una forma sencilla de verificar que el sistema funciona Ally funciona según lo previsto, entonces todo el esfuerzo en papel es cuestionable.

Un sistema que se prueba exhaustivamente y pasa todas sus pruebas todo el tiempo es una prueba: sistema capaz. Esa es una declaración obvia, pero importante. Sistemas que no se pueden probar no son verificables. Podría decirse que un sistema que no se puede verificar nunca debería implementarse.

Afortunadamente, hacer que nuestros sistemas sean probables nos empuja hacia un diseño donde nuestras clases son pequeñas y de un solo propósito. Es más fácil probar clases que se ajusten al SRP. La Cuantas más pruebas escribamos, más seguiremos avanzando hacia cosas que sean más sencillas de probar. Por lo tanto, asegurarnos de que nuestro sistema sea completamente comprobable nos ayuda a crear mejores diseños.

El acoplamiento estrecho dificulta la escritura de pruebas. De manera similar, cuantas más pruebas escribimos, cuanto más usamos principios como DIP y herramientas como inyección de dependencia, interfaces y abstracción para minimizar el acoplamiento. Nuestros diseños mejoran aún más.

Sorprendentemente, siguiendo una regla simple y obvia que dice que necesitamos hacernos pruebas y ejecutarlos afecta continuamente la adherencia de nuestro sistema a los objetivos primarios de OO de baja Acoplamiento y alta cohesión. Escribir pruebas conduce a mejores diseños.

Reglas de diseño simple 2-4: Refactorización

Una vez que tenemos las pruebas, estamos autorizados a mantener limpio nuestro código y nuestras clases. Hacemos esto por refactorizar incrementalmente el código. Por cada pocas líneas de código que agregamos, hacemos una pausa y reflexionamos sobre el nuevo diseño. ¿Lo acabamos de degradar? Si es así, lo limpiamos y ejecutamos nuestras pruebas para demostrar Esté seguro de que no hemos roto nada. *El hecho de que tengamos estas pruebas elimina el miedo que limpiar el código lo romperá!*

Durante este paso de refactorización, podemos aplicar cualquier cosa del conjunto de conocimientos sobre un buen diseño de software. Podemos aumentar la cohesión, disminuir el acoplamiento, separar el con-

cerns, modularizar las preocupaciones del sistema, reducir nuestras funciones y clases, elegir mejores nombres, y así. Aquí es también donde aplicamos las tres reglas finales del diseño simple: Eliminar duplicación, aseguran la expresividad y minimizan el número de clases y métodos.

Página 204

Sin duplicación

La duplicación es el enemigo principal de un sistema bien diseñado. Representa adicional trabajo, riesgo adicional y complejidad adicional innecesaria. Manifiestos de duplicación sí mismo en muchas formas. Las líneas de código que se ven exactamente iguales son, por supuesto, una duplicación. Las líneas de código que son similares a menudo se pueden modificar para que se parezcan aún más, de modo que se pueden refactorizar más fácilmente. Y la duplicación puede existir en otras formas como duplicación de la implementación. Por ejemplo, podríamos tener dos métodos en una colección. clase:

```
int tamaño () {}
boolean isEmpty () {}
```

Podríamos tener implementaciones separadas para cada método. El método `isEmpty` podría rastrear un booleano, mientras que el `tamaño` podría rastrear un contador. O podemos eliminar esta duplicación vinculando `isEmpty` para la definición de `tamaño` :

```
boolean isEmpty () {
    return 0 == tamaño ();
}
```

Crear un sistema limpio requiere la voluntad de eliminar la duplicación, incluso en unos pocos líneas de código. Por ejemplo, considere el siguiente código:

```
public void scaleToOneDimension (
        float deseadaDimensión, flotante imageDimension) {
    if (Math.abs (dimensión deseada - dimensión de la imagen) <errorThreshold)
        regreso;
    float scalingFactor = deseadaDimensión / imagenDimensión;
    scalingFactor = (float) (Math.floor (scalingFactor * 100) * 0.01f);

    RenderedOp newImage = ImageUtilities.getScaledImage (
        image, scalingFactor, scalingFactor);
    image.dispose ();
    System.gc ();
    imagen = nuevaImagen;
}
rotar vacío sincronizado público (grados int) {
    RenderedOp newImage = ImageUtilities.getRotatedImage (
        imagen, grados);
    image.dispose ();
    System.gc ();
    imagen = nuevaImagen;
}
```

Para mantener limpio este sistema, debemos eliminar la pequeña cantidad de duplicación entre los métodos `scaleToOneDimension` y `rotar` :

```
public void scaleToOneDimension (
        float deseadaDimensión, flotante imageDimension) {
    if (Math.abs (dimensión deseada - dimensión de la imagen) <errorThreshold)
        regreso;
    float scalingFactor = deseadaDimensión / imagenDimensión;
    scalingFactor = (float) (Math.floor (scalingFactor * 100) * 0.01f);
```

Página 205


```
        reemplazarImage (ImageUtilities.getScaledImage (
            image, scalingFactor, scalingFactor));
    }

    rotar vacío sincronizado público (grados int) {
        replaceImage (ImageUtilities.getRotatedImage (imagen, grados));
    }

    private void replaceImage (RenderedOp newImage) {
        image.dispose ();
        System.gc ();
        imagen = nuevaImagen;
    }
```

A medida que extraemos elementos en común en este nivel muy pequeño, comenzamos a reconocer las violaciones de SRP. Entonces, podríamos mover un método recién extraído a otra clase. Eso eleva su visibilidad. Alguien más en el equipo puede reconocer la oportunidad de abstraer aún más el nuevo método y reutilizarlo en un contexto diferente. Esta "reutilización en lo pequeño" puede causar que el sistema se complique. flexibilidad para encogerse dramáticamente. Entender cómo lograr la reutilización en lo pequeño es fundamental para lograr la reutilización a lo grande.

El T Emplate M ÉTODO [2] patrón es una técnica común para la eliminación de mayor nivel duplicación. Por ejemplo:

```
VacationPolicy de clase pública {
    public void accrueUSDivisionVacation () {
        // código para calcular las vacaciones en función de las horas trabajadas hasta la fecha
        // ...
        // código para garantizar que las vacaciones cumplan con los mínimos de EE. UU.
        // ...
        // código para aplicar la vacación al registro de nómina
        // ...
    }

    public void accrueEUDivisionVacation () {
        // código para calcular las vacaciones en función de las horas trabajadas hasta la fecha
        // ...
        // código para garantizar que las vacaciones cumplan con los mínimos de la UE
        // ...
        // código para aplicar la vacación al registro de nómina
        // ...
    }
}
```

El código entre accrueUSDivisionVacation y accrueEuropeanDivisionVacation es en gran medida lo mismo, a excepción del cálculo de mínimos legales. Esa parte del algoritmo cambios basados en el tipo de empleado.

Podemos eliminar la duplicación obvia mediante la aplicación de la T emplatar M ÉTODO patrón.

```
política de vacaciones de clase pública abstracta {
    public void accrueVacation () {
        calcularBaseVacationHours ();
```

2. [GOF].

Página 206

```
            alterForLegalMinimums ();
            applyToPayroll ();
        }

        private void calculateBaseVacationHours () {/ * ... * /};
        alterForLegalMinimums () vacío protegido abstracto;
        private void applyToPayroll () {/ * ... * /};
    }

    public class USVacationPolicy extiende VacationPolicy {
        @Override protected void alterForLegalMinimums () {
            // Lógica específica de EE. UU.
        }
    }

    La clase pública EUVacationPolicy extiende VacationPolicy {
        @Override protected void alterForLegalMinimums () {
            // Lógica específica de la UE
        }
    }
```

Las subclases llenan el "hueco" en el algoritmo accrueVacation , proporcionando los únicos bits de información que no esté duplicada.

Expresivo

La mayoría de nosotros hemos tenido la experiencia de trabajar en código intrincado. Muchos de nosotros hemos Hemos creado un código complicado nosotros mismos. Es fácil escribir código que *nos* entendemos, porque en el momento en que lo escribimos, tenemos un profundo conocimiento del problema que estamos tratando de resolver. Otros mantenedores del código no van a tener una comprensión tan profunda.

La mayor parte del costo de un proyecto de software está en el mantenimiento a largo plazo. Con el fin de minimizar el potencial de defectos a medida que introducimos cambios, es fundamental para nosotros poder entender lo que hace un sistema. A medida que los sistemas se vuelven más complejos, requieren más y más tiempo para que un desarrollador comprenda, y existe una oportunidad cada vez mayor para un error comprensión. Por lo tanto, el código debe expresar claramente la intención de su autor. El mas claro el autor puede hacer el código, menos tiempo tendrán que dedicar otros a comprenderlo. Esto reducirá los defectos y reducirá el costo de mantenimiento.

Puedes expresarte eligiendo buenos nombres. Queremos poder escuchar una clase o nombre de la función y no se sorprenda cuando descubramos sus responsabilidades.

También puede expresarse manteniendo sus funciones y clases pequeñas. Pequeña las clases y funciones suelen ser fáciles de nombrar, fáciles de escribir y fáciles de entender.

También puede expresarse utilizando una nomenclatura estándar. Patrones de diseño, para Por ejemplo, tienen que ver en gran medida con la comunicación y la expresividad. Usando el estándar nombres de patrones, como C OMMAND o V ISITOR , en los nombres de las clases que implementan Mente esos patrones, puede describir sucintamente su diseño a otros desarrolladores.

Las pruebas unitarias bien escritas también son expresivas. Un objetivo principal de las pruebas es actuar como mentación con el ejemplo. Alguien que lea nuestras pruebas debería poder comprender rápidamente de qué se trata una clase.

Página 207

Pero la forma más importante de ser expresivo es *intentarlo* . Con demasiada frecuencia obtenemos nuestro código trabajando y luego pasar al siguiente problema sin pensar lo suficiente en hacer ese código es fácil de leer para la próxima persona. Recuerde, la próxima persona más probable que lea el código serás tú.

Así que siéntete un poco orgulloso de tu mano de obra. Dedique un poco de tiempo a cada una de sus funciones. ciones y clases. Elija mejores nombres, divida las funciones grandes en funciones más pequeñas y en general, solo cuide lo que ha creado. El cuidado es un recurso precioso.

Clases y métodos mínimos

Incluso conceptos tan fundamentales como la eliminación de la duplicación, la expresividad del código y la

SRP puede llevarse demasiado lejos. En un esfuerzo por hacer que nuestras clases y métodos sean pequeños, podríamos crear demasiadas clases y métodos diminutos. Entonces esta regla sugiere que también mantenemos nuestra función ción y clase cuenta baja.

Los conteos de clase alta y método son a veces el resultado de un dogmatismo inútil. Estafasider, por ejemplo, un estándar de codificación que insiste en crear una interfaz para cada cada clase. O considere a los desarrolladores que insisten en que los campos y el comportamiento siempre deben estar separados clasificados en clases de datos y clases de comportamiento. Se debe oponer resistencia a tal dogma y Adopción de un enfoque pragmático.

Nuestro objetivo es mantener nuestro sistema en general pequeño mientras también mantenemos nuestras funciones y clases pequeñas. Sin embargo, recuerde que esta regla es la prioridad más baja de las cuatro reglas. de Diseño Simple. Entonces, aunque es importante mantener un recuento bajo de clases y funciones, es más importante es hacerse pruebas, eliminar la duplicación y expresarse.

Conclusión

¿Existe un conjunto de prácticas simples que puedan reemplazar la experiencia? Claramente no. En el otro Por otro lado, las prácticas descritas en este capítulo y en este libro son una forma cristalizada de la muchas décadas de experiencia disfrutada por los autores. Siguiendo la práctica de simple El diseño puede y anima y permite a los desarrolladores adherirse a buenos principios y patrones que de otro modo llevarían años aprender.

Bibliografía

[XPE]: *Explicación de la programación extrema: Embrace Change,* Kent Beck, Addison-Wesley, 1999.

[GOF]: *Patrones de diseño: elementos de software orientado a objetos reutilizables,* Gamma et al., Addison-Wesley, 1996.

13

Concurrencia

por Brett L. Schuchert

"Los objetos son abstracciones del procesamiento. Los hilos son abstracciones del horario ".

—James O. Coplien [1]

1. Correspondencia privada.

Página 209

Escribir programas simultáneos limpios es difícil, muy difícil. Es mucho más fácil escribir código que se ejecuta en un solo hilo. También es fácil escribir código multiproceso que se ve bien en el superficie pero se rompe a un nivel más profundo. Dicho código funciona bien hasta que se coloca el sistema bajo estrés.

En este capítulo discutimos la necesidad de programación concurrente y las dificultades presenta. A continuación, presentamos varias recomendaciones para abordar esas dificultades, y escribir código concurrente limpio. Finalmente, concluimos con los problemas relacionados con las pruebas. código concurrente.

Clean Concurrency es un tema complejo, digno de un libro en sí mismo. Nuestra estrategia en *este* El libro es presentar una descripción general aquí y proporcionar un tutorial más detallado en "Concurrencia II" en la página 317. Si solo siente curiosidad por la simultaneidad, este capítulo será suficiente para usted ahora. Si necesita comprender la simultaneidad a un nivel más profundo, debería leer a través del tutorial también.

¿Por qué la concurrencia?

La concurrencia es una estrategia de desacoplamiento. Nos ayuda desacoplar *lo que* se hace a partir de *cuando* se se hace. En aplicaciones de un solo subproceso, *qué* y *cuándo* están tan fuertemente acoplados que el El estado de toda la aplicación a menudo se puede determinar observando el seguimiento de la pila. A El programador que depura un sistema de este tipo puede establecer un punto de interrupción, o una secuencia de puntos de interrupción, y *conocer* el estado del sistema mediante el cual se alcanzan los puntos de interrupción.

Desacoplar *qué* y *cuándo* puede mejorar drásticamente tanto el rendimiento como la estructura. tures de una aplicación. Desde un punto de vista estructural, la aplicación se parece a muchos textos tle computadoras colaboradoras en lugar de un gran bucle principal. Esto puede facilitar el sistema para comprender y ofrece algunas formas poderosas de separar las preocupaciones.

Considere, por ejemplo, el modelo estándar de "Servlet" de aplicaciones web. Estos sistemas Los elementos se ejecutan bajo el paraguas de un contenedor Web o EJB que gestiona *parcialmente la* concurrencia. rencia para ti. Los servlets se ejecutan de forma asincrónica cada vez que ingresan solicitudes web. El programador de servlets no tiene que gestionar todas las solicitudes entrantes. *En principio,* Cada ejecución de servlet vive en su propio pequeño mundo y está desacoplada de todos los demás servicios. Dejemos ejecuciones.

Por supuesto, si fuera tan fácil, este capítulo no sería necesario. De hecho, el decou- El pling proporcionado por los contenedores web es mucho menos que perfecto. Los programadores de servlets deben ser

muy consciente y muy cuidadoso para asegurarse de que sus programas simultáneos sean correctos. Aún así, el Los beneficios estructurales del modelo de servlet son significativos.

Pero la estructura no es el único motivo para adoptar la concurrencia. Algunos sistemas tienen Restricciones de tiempo de respuesta y rendimiento que requieren soluciones simultáneas codificadas a mano. Por ejemplo, considere un agregador de información de un solo subproceso que adquiere información de muchos sitios web diferentes y fusiona esa información en un resumen diario. Porque

Página 210

este sistema es de un solo subproceso, llega a cada sitio web por turno, siempre terminando uno antes comenzando el siguiente. La ejecución diaria debe ejecutarse en menos de 24 horas. Sin embargo, como más y se agregan más sitios web, el tiempo aumenta hasta que se necesitan más de 24 horas para recopilar todos los datos. El hilo único implica mucha espera en los sockets web para que se complete la E / S. Podríamos mejorar el rendimiento mediante el uso de un algoritmo multiproceso que alcanza más de un sitio web a la vez.

O considere un sistema que maneja un usuario a la vez y requiere solo un segundo de tiempo por usuario. Este sistema es bastante receptivo para algunos usuarios, pero como el número de usuarios aumenta, el tiempo de respuesta del sistema aumenta. Ningún usuario quiere hacer cola detrás de otros 150! Podríamos mejorar el tiempo de respuesta de este sistema manejando muchos usuarios al mismo tiempo.

O considere un sistema que interpreta grandes conjuntos de datos pero que solo puede brindar una solución completa. ción después de procesarlos todos. Quizás cada conjunto de datos podría procesarse en un computadora, por lo que muchos conjuntos de datos se procesan en paralelo.

Mitos y conceptos erróneos

Por tanto, existen razones de peso para adoptar la simultaneidad. Sin embargo, como dijimos antes, la concurrencia es *difícil* . Si no tiene mucho cuidado, puede crear situaciones muy desagradables. Considere estos mitos y conceptos erróneos comunes:

- La *concurrencia siempre mejora el rendimiento.*
  La simultaneidad a *veces* puede mejorar el rendimiento, pero solo cuando hay mucha espera tiempo que se puede compartir entre varios subprocesos o varios procesadores. Ninguna situación ación es trivial.
- El *diseño no cambia al escribir programas concurrentes.*
  De hecho, el diseño de un algoritmo concurrente puede ser notablemente diferente del diseño de un sistema de un solo hilo. El desacoplamiento de *qué* y *cuándo* generalmente tiene un gran efecto en la estructura del sistema.
- *Comprender los problemas de simultaneidad no es importante cuando se trabaja con un contenedor. como un contenedor Web o EJB.*
  De hecho, es mejor que sepa qué está haciendo su contenedor y cómo protegerse contra los problemas de actualización simultánea y bloqueo que se describen más adelante en este capítulo.

Aquí hay algunos fragmentos de sonido más equilibrados con respecto a la escritura de software concurrente:

- *La simultaneidad genera algunos gastos generales,* tanto en el rendimiento como en la escritura adicional. código.
- *La concurrencia correcta es compleja,* incluso para problemas simples.

Página 211

- Los *errores de simultaneidad no suelen ser repetibles,* por lo que a menudo se ignoran como excepcionales [2] en lugar de los verdaderos defectos que son.
- La *concurrencia a menudo requiere un cambio fundamental en la estrategia de diseño* .

Desafíos

¿Qué hace que la programación concurrente sea tan difícil? Considere la siguiente clase trivial:

```
clase pública X {
    private int lastIdUsed;

    public int getNextId () {
            return ++ lastIdUsed;
      }
}
```

Digamos que creamos una instancia de `X` , establecemos el campo `lastIdUsed` en 42 y luego compartimos el instancia entre dos subprocesos. Ahora suponga que ambos subprocesos llaman al método `getNextId ()` ; hay tres posibles resultados:

- El hilo uno obtiene el valor 43, el hilo dos obtiene el valor 44, `lastIdUsed` es 44.
- El hilo uno obtiene el valor 44, el hilo dos obtiene el valor 43, `lastIdUsed` es 44.
- El hilo uno obtiene el valor 43, el hilo dos obtiene el valor 43, `lastIdUsed` es 43.

El sorprendente tercer resultado [3] produce cuando los dos hilos se pisan entre sí. Este suceso bolígrafos porque hay muchos caminos posibles que los dos hilos pueden tomar a través de ese línea de código Java, y algunas de esas rutas generan resultados incorrectos. Cuantos diferentes caminos hay? Para responder realmente a esa pregunta, debemos comprender qué es lo Time Compiler hace con el código de bytes generado y comprende lo que la memoria Java El modelo considera que es atómico.

Una respuesta rápida, trabajando solo con el código de bytes generado, es que hay 12,870 diferentes posibles rutas de ejecución [4] para esos dos subprocesos que se ejecutan dentro de `getNextId` método. Si el tipo de `lastIdUsed` se cambia de `int` a `long` , el número de posibles caminos aumenta a 2.704.156. Por supuesto, la mayoría de esos caminos generan resultados válidos. La El problema es que *algunos de ellos no lo hacen* .

Principios de defensa de la concurrencia

Lo que sigue es una serie de principios y técnicas para defender sus sistemas de la problemas de código concurrente.

2. Rayos cósmicos, fallas, etc.
3. Consulte "Profundizar más" en la página 323.
4. Consulte “Posibles rutas de ejecución” en la página 321.

Página 212

Principio de responsabilidad única

El SRP [5] establece que un método / clase / componente dado debe tener una única razón para cambio. El diseño de simultaneidad es lo suficientemente complejo como para ser una razón para cambiar por derecho propio y por lo tanto merece ser separado del resto del código. Desafortunadamente, todo es demasiado Es común que los detalles de implementación de la concurrencia se incrusten directamente en otros programas. código de producción. Aquí hay algunas cosas para considerar:

- El *código relacionado con la simultaneidad tiene su propio ciclo de vida de desarrollo,* cambio y ajuste.
- El *código relacionado con la simultaneidad tiene sus propios desafíos,* que son diferentes y a menudo más difícil que el código no relacionado con la moneda corriente.
- La cantidad de formas en que el código basado en concurrencia mal escrito puede fallar hace que lo suficientemente desafiante sin la carga adicional del código de aplicación circundante.

Recomendación : *mantenga su código relacionado con la simultaneidad separado de otros códigos* . [6]

Corolario: limitar el alcance de los datos

Como vimos, dos subprocesos que modifican el mismo campo de un objeto compartido pueden interferir con cada otro, provocando un comportamiento inesperado. Una solución es utilizar la palabra clave sincronizada para proteger una *sección crítica* en el código que utiliza el objeto compartido. Es importante restringir el número de esas secciones críticas. Cuantos más lugares puedan actualizarse los datos compartidos, más como:

- Olvidarás proteger uno o más de esos lugares, rompiendo efectivamente todo el código. que modifica esos datos compartidos.
- Se requerirá una duplicación de esfuerzos para asegurarse de que todo sea eficaz vigilado (violación de DRY [7]).
- Será difícil determinar la fuente de fallas, que ya son lo suficientemente difíciles encontrar.

Recomendación : *tome en serio la encapsulación de datos; limitar severamente el acceso de cualquier datos que pueden ser compartidos.*

Corolario: use copias de datos

Una buena forma de evitar los datos compartidos es evitar compartir los datos en primer lugar. En alguna situación En ocasiones, es posible copiar objetos y tratarlos como de solo lectura. En otros casos, podría ser posible copiar objetos, recopilar resultados de varios subprocesos en estas copias y luego fusionar los resultados en un solo hilo.

5. [PPP]
6. Consulte “Ejemplo de cliente / servidor” en la página 317.
7. [PRAG].

Página 213

Si hay una manera fácil de evitar compartir objetos, el código resultante será mucho menos probable causar problemas. Es posible que le preocupe el costo de toda la creación de objetos adicionales. Es Vale la pena experimentar para averiguar si esto es realmente un problema. Sin embargo, si usa copias de Los objetos permiten que el código evite la sincronización, el ahorro al evitar el bloqueo intrínseco probablemente compensará la sobrecarga adicional de creación y recolección de basura.

Corolario: los hilos deben ser lo más independientes posible

Considere escribir su código de subprocesos de modo que cada subproceso exista en su propio mundo, compartiendo no hay datos con ningún otro hilo. Cada hilo procesa una solicitud de cliente, con todos sus datos requeridos provenientes de una fuente no compartida y almacenados como variables locales. Esto hace cada uno de esos hilos se comporta como si fuera el único hilo del mundo y no hubiera requisitos de sincronización.

Por ejemplo, las clases de esa subclase de HttpServlet reciben toda su información como parámetros pasados a los métodos doGet y doPost . Esto hace que cada Servlet actúe como si tuviera su propia máquina. Siempre que el código del Servlet use solo variables locales, no hay posibilidad de que el Servlet cause problemas de sincronización. Por supuesto, la mayoría de las aplicaciones que utilizan Servlets eventualmente se ejecutan en recursos compartidos como la base de datos conexiones.

Recomendación : *intente dividir los datos en subconjuntos independientes de los que se pueden operado por subprocesos independientes, posiblemente en diferentes procesadores.*

Conozca su biblioteca

Java 5 ofrece muchas mejoras para el desarrollo simultáneo con respecto a versiones anteriores. Allí Hay varias cosas a considerar al escribir código enhebrado en Java 5:

- Utilice las colecciones seguras para subprocesos proporcionadas.
- Utilizar el marco ejecutor para ejecutar tareas no relacionadas.
- Utilice soluciones que no bloqueen cuando sea posible.
- Varias clases de biblioteca no son seguras para subprocesos.

Colecciones seguras para subprocesos

Cuando Java era joven, Doug Lea escribió el libro fundamental [8] *Programación concurrente en Java* . Junto con el libro, desarrolló varias colecciones seguras para subprocesos, que más tarde se convirtió en parte del JDK en el paquete java.util.concurrent . Las colecciones de ese paquete age son seguras para situaciones multiproceso y funcionan bien. De hecho, el

8. [Lea99].

La implementación de ConcurrentHashMap funciona mejor que HashMap en casi todas las situaciones. Eso también permite lecturas y escrituras simultáneas simultáneas, y tiene métodos que admiten operaciones compuestas comunes que de otro modo no son seguras para subprocesos. Si Java 5 es la implementación entorno de ment, comience con ConcurrentHashMap .

Se han agregado varios otros tipos de clases para admitir la concurrencia avanzada. diseño. Aquí están algunos ejemplos:

ReentrantLock	Un candado que puede adquirirse con un método y liberarse con otro.
Semáforo	Una implementación del semáforo clásico, un candado con cuenta.
CountDownLatch	Un candado que espera una serie de eventos antes de liberar todos hilos esperando en él. Esto permite que todos los subprocesos tengan una oportunidad justa de comenzar aproximadamente al mismo tiempo.

Recomendación : *revise las clases disponibles para usted. En el caso de Java, conviértase familiarizado con java.util.concurrent, java.util.concurrent.atomic, java.util.concurrent.locks.*

Conozca sus modelos de ejecución

Hay varias formas diferentes de particionar el comportamiento en una aplicación concurrente. Para dis-
Maldita sea, necesitamos entender algunas definiciones básicas.

Recursos vinculados	Recursos de un tamaño o número fijo utilizados en un entorno concurrente ment. Los ejemplos incluyen conexiones de base de datos y lectura / escribir búferes.
Exclusión mutua	Solo un hilo puede acceder a datos compartidos o un recurso compartido en un hora.
Inanición	Se prohíbe el avance de un hilo o un grupo de hilos durante un tiempo excesivamente largo o para siempre. Por ejemplo, siempre deje Pasar primero los subprocesos de ejecución rápida podría privar a los crear subprocesos si no hay fin para los subprocesos de ejecución rápida.
Punto muerto	Dos o más subprocesos esperando que el otro termine. Cada hilo tiene un recurso que el otro hilo requiere y ninguno puede terminar hasta que obtenga el otro recurso.
Livelock	Hilos en sincronía, cada uno tratando de hacer un trabajo pero encontrando otro "en la forma." Debido a la resonancia, los hilos continúan intentando progresar pero no puede hacerlo durante un tiempo excesivamente largo— o para siempre.

Dadas estas definiciones, ahora podemos discutir los diversos modelos de ejecución utilizados en programación concurrente.

Página 215

Productor-Consumidor [9]

Uno o más subprocesos de productores crean algo de trabajo y lo colocan en un búfer o cola. Uno o más subprocesos de consumidores adquieren ese trabajo de la cola y lo completan. La cola entre los productores y los consumidores es un *recurso ligado* . Esto significa que los productores deben esperar espacio libre en la cola antes de escribir y los consumidores deben esperar hasta que haya algo en la cola para consumir. Coordinación entre productores y consumidores a través de la cola involucra a productores y consumidores que se señalan entre sí. Los productores escriben a la cola y señalar que la cola ya no está vacía. Los consumidores leen de la cola y señalar que la cola ya no está llena. Ambos potencialmente esperan ser notificados cuando Puede continuar.

Lectores-Escritores [10]

Cuando tiene un recurso compartido que sirve principalmente como fuente de información para leer, ers, pero que ocasionalmente es actualizado por escritores, el rendimiento es un problema. Enfatizando el rendimiento puede causar inanición y la acumulación de información obsoleta. Permitiendo las actualizaciones pueden afectar el rendimiento. Coordinar a los lectores para que no lean algo El escritor se actualiza y viceversa es un duro acto de equilibrio. Los escritores tienden a bloquear muchas lecturas ers durante un largo período de tiempo, lo que provoca problemas de rendimiento.

El desafío es equilibrar las necesidades de lectores y escritores para satisfacer las operación, proporcionar un rendimiento razonable y evitar el hambre. Una estrategia simple hace que los escritores esperen hasta que no haya lectores antes de permitir que el escritor realice una actualizar. Sin embargo, si hay lectores continuos, los escritores se morirán de hambre. En el otro Por otro lado, si hay escritores frecuentes y se les da prioridad, el rendimiento se verá afectado. Encontrar- Lograr ese equilibrio y evitar problemas de actualización simultáneos es lo que aborda el problema.

Filósofos gastronómicos [11]

Imagínese a varios filósofos sentados alrededor de una mesa circular. Se coloca un tenedor en el

a la izquierda de cada filósofo. Hay un gran plato de espaguetis en el centro de la mesa. La
los filósofos pasan su tiempo pensando a menos que tengan hambre. Una vez hambrientos, escogen
sube los tenedores a cada lado de ellos y come. Un filósofo no puede comer a menos que esté sosteniendo
dos tenedores. Si el filósofo a su derecha o izquierda ya está usando una de las bifurcaciones,
necesidades, debe esperar hasta que el filósofo termine de comer y vuelva a bajar los tenedores.
Una vez que un filósofo come, vuelve a poner ambos tenedores sobre la mesa y espera hasta
tiene hambre de nuevo.

Reemplace a los filósofos con hilos y bifurcaciones con recursos y este problema es similar.
lar a muchas aplicaciones empresariales en las que los procesos compiten por los recursos. A menos que te importe
completamente diseñados, los sistemas que compiten de esta manera pueden experimentar interbloqueo, bloqueo activo,
rendimiento y degradación de la eficiencia.

9. http://en.wikipedia.org/wiki/Producer-consumer
10. http://en.wikipedia.org/wiki/Readers-writers_problem
11. http://en.wikipedia.org/wiki/Dining_philosophers_problem

Página 216

La mayoría de los problemas simultáneos que probablemente encontrará serán alguna variación de estos
tres problemas. Estudie estos algoritmos y escriba soluciones utilizándolos por su cuenta para
que cuando se encuentre con problemas concurrentes, estará más preparado para resolver el problema
problema.

Recomendación : *aprenda estos algoritmos básicos y comprenda sus soluciones.*

Tenga cuidado con las dependencias entre métodos sincronizados

Las dependencias entre métodos sincronizados provocan errores sutiles en el código concurrente. La
El lenguaje Java tiene la noción de sincronizado , que protege un método individual. Cómo-
alguna vez, si hay más de un método sincronizado en la misma clase compartida, entonces su
El sistema puede estar escrito incorrectamente. [12]

Recomendación : *evite utilizar más de un método en un objeto compartido.*

Habrá ocasiones en las que deberá utilizar más de un método en un objeto compartido.
Cuando este es el caso, hay tres formas de corregir el código:

- **Bloqueo basado en** el cliente: pida al cliente que bloquee el servidor antes de llamar al primer
  y asegúrese de que la extensión del bloqueo incluya código que llame al último método.
- **Bloqueo** basado en servidor: dentro del servidor, cree un método que bloquee el servidor, las llamadas
  todos los métodos, y luego desbloquea. Haga que el cliente llame al nuevo método.
- **Servidor adaptado:** cree un intermediario que realice el bloqueo. Este es un examen
  ple de bloqueo basado en servidor, donde el servidor original no se puede cambiar.

Mantenga pequeñas las secciones sincronizadas

La palabra clave sincronizada introduce un candado. Todas las secciones de código protegidas por el
Se garantiza que el mismo bloqueo tenga solo un hilo ejecutándose a través de ellos en cualquier momento dado.
hora. Los bloqueos son costosos porque crean retrasos y agregan gastos generales. Entonces no lo hacemos
queremos ensuciar nuestro código con declaraciones sincronizadas . Por otro lado, los sectores críticos
tions [13] deben ser resguardados. Así que queremos diseñar nuestro código con tan pocas secciones críticas como
posible.

Algunos programadores ingenuos intentan lograr esto haciendo que sus secciones críticas sean muy
grande. Sin embargo, extender la sincronización más allá de la sección crítica mínima aumenta
contención y degrada el rendimiento. [14]

Recomendación : *Mantenga sus secciones sincronizadas lo más pequeñas posible.*

12. Consulte “Las dependencias entre métodos pueden romper el código concurrente” en la página 329.
13. Una sección crítica es cualquier sección de código que debe protegerse del uso simultáneo para que el programa sea correcto.
14. Consulte "Aumento del rendimiento" en la página 333.

Página 217

Escribir un código de apagado correcto es difícil

Escribir un sistema que está destinado a permanecer en vivo y en funcionamiento para siempre es diferente a escribir algo. cosa que funciona por un tiempo y luego se apaga con gracia.

El cierre ordenado puede ser difícil de realizar correctamente. Los problemas comunes implican un punto muerto, [15] con hilos esperando una señal para continuar que nunca llega.

Por ejemplo, imagine un sistema con un subproceso principal que genera varios subprocesos secundarios y luego espera a que todos terminen antes de que libere sus recursos y se apague. Y si uno de los subprocesos generados está bloqueado? El padre esperará para siempre y el sistema nunca se apagará.

O considere un sistema similar al que se le haya *ordenado* que se apague. El padre lo dice todo los niños engendrados abandonan sus tareas y terminan. Pero y si dos de los niños operaban como un par productor / consumidor. Supongamos que el productor recibe la señal del padre y se apaga rápidamente. El consumidor podría haber estado esperando un mensaje salvia del productor y ser bloqueado en un estado en el que no pueda recibir la señal de apagado nal. Podría quedarse atascado esperando al productor y nunca terminar, evitando que el padre terminando también.

Situaciones como esta no son nada infrecuentes. Entonces, si debe escribir código concurrente que implica cerrar con gracia, espere pasar gran parte de su tiempo haciendo el cierre para que suceda correctamente.

Recomendación : *Piense en cerrar temprano y hacer que funcione temprano. Va hacia tardará más de lo esperado. Revise los algoritmos existentes porque probablemente esto sea más difícil de lo que piensas.*

Prueba de código enhebrado

Demostrar que el código es correcto no es práctico. Las pruebas no garantizan la corrección. Cómo- siempre, una buena prueba puede minimizar el riesgo. Todo esto es cierto en una solución de un solo subproceso. Tan pronto como hay dos o más subprocesos que usan el mismo código y trabajan con datos compartidos, las cosas se vuelven sustancialmente más complejos.

Recomendación : *escriba pruebas que tengan el potencial de exponer problemas y luego ejecutarlos con frecuencia, con diferentes configuraciones programáticas y configuraciones del sistema y cargar. Si alguna vez las pruebas fallan, localice la falla. No ignore un fracaso solo porque el las pruebas pasan en una ejecución posterior.*

Eso es mucho para tener en cuenta. Aquí hay algunos más detallados recomendaciones:

- Trate las fallas falsas como posibles problemas de subprocesos.
- Primero haga funcionar su código sin hilos.

15. Consulte "Interbloqueo" en la página 335.

Página 218

- Haga que su código enhebrado sea enchufable.
- Haga que su código enhebrado sea sintonizable.
- Ejecutar con más subprocesos que procesadores.
- Ejecutar en diferentes plataformas.
- Instrumente su código para intentar forzar fallas.

Trate las fallas falsas como problemas de subprocesos candidatos

El código enhebrado hace que fallen cosas que "simplemente no pueden fallar". La mayoría de los desarrolladores no tienen una sensación intuitiva de cómo el subproceso interactúa con otro código (autores incluidos). Errores en el código enhebrado puede mostrar sus síntomas una vez en mil o un millón de ejecuciones.
Los intentos de repetir los sistemas pueden resultar frustrantes. Esto a menudo lleva a los desarrolladores a cancelar el fallo como un rayo cósmico, un fallo de hardware o algún otro tipo de "única". Es mejor suponga que las excepciones no existen. Cuanto más tiempo se ignoren estas "excepciones", más código se basa en un enfoque potencialmente defectuoso.

Recomendación : *No ignore los fallos del sistema como únicos.*

Primero, haga funcionar su código sin subprocesos

Esto puede parecer obvio, pero no está de más reforzarlo. Asegúrese de que el código funcione en el exterior de su uso en hilos. Generalmente, esto significa crear POJO a los que llaman sus hilos.
Los POJO no son conscientes de los subprocesos y, por lo tanto, se pueden probar fuera del entorno de subprocesos. ambiente. Cuanto más de su sistema pueda colocar en dichos POJO, mejor.

Recomendación : *no intente perseguir errores que no son subprocesos ni errores de subprocesos al mismo tiempo. Asegúrese de que su código funcione fuera de los subprocesos .*

Haga que su código enhebrado se pueda conectar

Escriba el código compatible con la simultaneidad de modo que se pueda ejecutar en varias configuraciones:

- Un hilo, varios hilos, variado a medida que se ejecuta
- El código enhebrado interactúa con algo que puede ser tanto real como un doble de prueba.
- Ejecutar con dobles de prueba que se ejecutan de forma rápida, lenta, variable.
- Configure las pruebas para que se puedan ejecutar durante varias iteraciones.

Recomendación : *haga que su código basado en subprocesos sea especialmente conectable para que puede ejecutarlo en varias configuraciones.*

Haga que su código de subprocesos se pueda ajustar

Lograr el equilibrio correcto de subprocesos generalmente requiere probar un error. Al principio, encuentre formas de cronometra el rendimiento de su sistema en diferentes configuraciones. Permita el número de

Página 219

hilos para afinarlos fácilmente. Considere permitir que cambie mientras el sistema está en funcionamiento.
Considere la posibilidad de permitir el autoajuste en función del rendimiento y la utilización del sistema.

Ejecutar con más subprocesos que procesadores

Suceden cosas cuando el sistema cambia entre tareas. Para fomentar el intercambio de tareas, ejecute con más subprocesos que procesadores o núcleos. Cuanto más frecuentemente cambien sus tareas, más es probable que encuentre un código al que le falta una sección crítica o que cause un punto muerto.

Ejecutar en diferentes plataformas

A mediados de 2007 desarrollamos un curso sobre programación concurrente. El curso el desarrollo se produjo principalmente en OS X. La clase se presentó utilizando Windows XP corriendo bajo una VM. Las pruebas escritas para demostrar las condiciones de falla no fallaron tan frecuentemente con frecuencia en un entorno XP como lo hacían con OS X.

En todos los casos, se sabía que el código bajo prueba era incorrecto. Esto solo reforzó el hecho que los diferentes sistemas operativos tienen diferentes políticas de subprocesos, cada una de las cuales tiene un impacto la ejecución del código. El código multiproceso se comporta de manera diferente en diferentes entornos. [dieciséis] Debe ejecutar sus pruebas en todos los entornos de implementación potenciales.

Recomendación : *ejecute su código enhebrado en todas las plataformas de destino de forma temprana y frecuente.*

Instrumente su código para intentar y forzar fallas

Es normal que se oculten las fallas en el código concurrente. Las pruebas simples a menudo no los exponen. De hecho, a menudo se esconden durante el procesamiento normal. Pueden aparecer una vez cada pocos horas, días o semanas!

La razón por la que los errores de enhebrado pueden ser poco frecuentes, esporádicos y difíciles de repetir es que sólo unas pocas vías de los muchos miles de posibles vías a través de una vul-
sección nerable en realidad falla. Entonces, la probabilidad de que se tome una ruta defectuosa puede ser hormigueo bajo. Esto dificulta mucho la detección y la depuración.

¿Cómo podría aumentar sus posibilidades de contraer ocurrencias tan raras? Usted puede Instrumentar su código y obligarlo a ejecutarse en diferentes órdenes agregando llamadas a métodos como Object.wait () , Object.sleep () , Object.yield () y Object.priority () .

Cada uno de estos métodos puede afectar el orden de ejecución, aumentando así las probabilidades de detectar un defecto. Es mejor cuando el código roto falla lo antes y con la mayor frecuencia posible.

Hay dos opciones para la instrumentación de código:

- Codificado a mano
- Automatizado

16. ¿Sabía que el modelo de subprocesamiento en Java no garantiza el subprocesamiento preventivo? Soporte preventivo del sistema operativo moderno subprocesos, por lo que obtienes eso "gratis". Aun así, no está garantizado por la JVM.

Página 220

Codificado a mano

Puede insertar manualmente llamadas a wait () , sleep () , yield () y priority () en su código. Eso podría ser justo lo que debe hacer cuando está probando un fragmento de código particularmente espinoso.

Aquí hay un ejemplo de cómo hacer precisamente eso:

```
cadena pública sincronizada nextUrlOrNull () {
        if (hasNext ()) {
                String url = urlGenerator.next ();
                Thread.yield (); // insertado para probar.
                updateHasNext ();
                return url;
        }
        devolver nulo;
}
```

La llamada insertada a yield () cambiará las rutas de ejecución tomadas por el código y posiblemente haga que el código falle donde antes no fallaba. Si el código se rompe, fue no porque agregaste una llamada a yield () . [17] Más bien, su código se rompió y esto simplemente hizo evidente el fracaso.

Hay muchos problemas con este enfoque:

- Tiene que encontrar manualmente los lugares apropiados para hacer esto.
- ¿Cómo sabe dónde poner la llamada y qué tipo de llamada utilizar?
- Dejar dicho código en un entorno de producción ralentiza innecesariamente el código.
- Es un enfoque de escopeta. Puede o no encontrar fallas. De hecho, las probabilidades no están contigo.

Lo que necesitamos es una forma de hacer esto durante las pruebas, pero no en la producción. También necesitamos Mezclar fácilmente configuraciones entre diferentes ejecuciones, lo que resulta en mayores posibilidades de encontrar errores en el agregado.

Claramente, si dividimos nuestro sistema en POJO que no saben nada de subprocesamiento y clases que controlan el subproceso, será más fácil encontrar lugares apropiados para instrumentar el código. Además, podríamos crear muchas plantillas de prueba diferentes que invoquen los POJO en diferentes regímenes de llamadas a dormir , ceder , etc.

Automatizado

Puede utilizar herramientas como un marco orientado a aspectos, CGLIB o ASM para programar Instrumentar icamente su código. Por ejemplo, podría usar una clase con un solo método:

```
public class ThreadJigglePoint {
        public static void jiggle () {
        }
}
```

17. Este no es estrictamente el caso. Dado que la JVM no garantiza el subproceso preventivo, un algoritmo particular siempre trabajar en un sistema operativo que no se adelanta a los subprocesos. Lo contrario también es posible, pero por diferentes razones.

Página 221

Puede agregar llamadas a esto en varios lugares dentro de su código:

```
cadena pública sincronizada nextUrlOrNull () {
        if (hasNext ()) {
                ThreadJiglePoint.jiggle ();
                String url = urlGenerator.next ();
                ThreadJiglePoint.jiggle ();
                updateHasNext ();
                ThreadJiglePoint.jiggle ();
                return url;
        }
        devolver nulo;
}
```

Ahora usa un aspecto simple que selecciona aleatoriamente entre no hacer nada, dormir o flexible.

O imagine que la clase ThreadJigglePoint tiene dos implementaciones. El primer imple- mentos se mueven para no hacer nada y se utiliza en la producción. El segundo genera un azar número para elegir entre dormir, ceder o simplemente caer. Si ejecuta sus pruebas mil veces con movimientos aleatorios, es posible que elimine algunos defectos. Si las pruebas pasan, en al menos puede decir que ha realizado la debida diligencia. Aunque un poco simplista, esto podría ser una razón opción sonable en lugar de una herramienta más sofisticada.

Existe una herramienta llamada ConTest, [18] desarrollada por IBM que hace algo similar, pero lo hace con un poco más de sofisticación.

El punto es mover el código para que los subprocesos se ejecuten en diferentes órdenes en diferentes veces. La combinación de pruebas bien escritas y jiggling puede aumentar drásticamente la posibilidad de encontrar errores.

Recomendación : *utilice estrategias de jiggling para descubrir errores.*

Conclusión

El código concurrente es difícil de acertar. El código que es simple de seguir puede volverse nocturno se estropea cuando varios hilos y datos compartidos entran en la mezcla. Si se enfrenta a un escrito ing código concurrente, necesita escribir código limpio con rigor o de lo contrario enfrentar sutil y fallas poco frecuentes.

En primer lugar, siga el principio de responsabilidad única. Divide tu sistema en POJO que separan el código que admite subprocesos del código que ignora subprocesos. Asegúrate de que cuando está probando su código compatible con subprocesos, solo lo está probando y nada más. Esta sugerencia que su código consciente de subprocesos debe ser pequeño y enfocado.

Conozca las posibles fuentes de problemas de simultaneidad: múltiples subprocesos que operan en datos compartidos o utilizando un grupo de recursos común. Casos límite, como el cierre limpiamente o terminar la iteración de un bucle, puede ser especialmente espinoso.

18. http://www.alphaworks.ibm.com/tech/contest

Conozca su biblioteca y conozca los algoritmos fundamentales. Entender cómo algunos de las funciones que ofrece la biblioteca apoyan la resolución de problemas similares a los fundamentales algoritmos.

Aprenda a buscar regiones de código que se deben bloquear y bloquearlas. No bloquear regiones de código que no necesitan bloquearse. Evite llamar a una sección bloqueada desde otro. Esto requiere una comprensión profunda de si algo se comparte o no. Mantenerse la cantidad de objetos compartidos y el alcance del intercambio lo más reducido posible. Cambio Diseños de los objetos con datos compartidos para acomodar a los clientes en lugar de forzarlos. para gestionar el estado compartido.

Surgirán problemas. Los que no surgen temprano a menudo se descartan como un ocurrencia de tiempo. Estas llamadas excepciones suelen ocurrir solo bajo carga o al parecer. veces al azar. Por lo tanto, debe poder ejecutar su código relacionado con el hilo en muchas configuraciones en muchas plataformas de forma repetida y continua. Testabilidad, que proviene naturalmente de seguir las Tres Leyes de TDD, implica cierto nivel de capacidad de conexión, que ofrece el soporte necesario para ejecutar código en una gama más amplia de configuraciones.

Mejorará en gran medida sus posibilidades de encontrar un código erróneo si se toma el tiempo para instrumentar su código. Puede hacerlo a mano o mediante algún tipo de tecnología. Invierta en esto temprano. Desea ejecutar su código basado en subprocesos siempre que posible antes de ponerlo en producción.

Si adopta un enfoque limpio, sus posibilidades de hacerlo bien aumentan drásticamente.

Bibliografía

[Lea99]: *Programación concurrente en Java: Principios y patrones de diseño* , 2d. ed., Doug Lea, Prentice Hall, 1999.

[PPP]: *Desarrollo de software ágil: principios, patrones y prácticas* , Robert C. Martin, Prentice Hall, 2002.

[PRAG]: *El programador pragmático* , Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

Página 223

Esta página se dejó en blanco intencionalmente

Página 224

14

Refinamiento sucesivo

Estudio de caso de un analizador de argumentos de línea de comandos

Este capítulo es un estudio de caso en sucesivas mejoras. Verá un módulo que comenzó bien, pero no escala. Luego verá cómo se refactorizó y limpió el módulo.

La mayoría de nosotros hemos tenido que analizar los argumentos de la línea de comandos de vez en cuando. Si nosotros no tenemos una utilidad conveniente, entonces simplemente recorremos la matriz de cadenas que se pasa en la función principal . Hay varias buenas utilidades disponibles de varias fuentes,

Página 225

pero ninguno de ellos hace exactamente lo que quiero. Entonces, por supuesto, decidí escribir el mío. Yo lo llamo eso: Args .

Args es muy sencillo de utilizar. Simplemente construye la clase Args con el argumento de entrada mentos y una cadena de formato, y luego consultar la instancia de Args para los valores de los argumentos mentos. Considere el siguiente ejemplo sencillo:

Listado 14-1

Uso simple de Args

```
public static void main (String [] args) {
    intentar {
        Args arg = new Args ("l, p #, d *", args);
        registro booleano = arg.getBoolean ('l');
        puerto int = arg.getInt ('p');
        Directorio de cadenas = arg.getString ('d');
        executeApplication (registro, puerto, directorio);
    } captura (ArgsException e) {
        System.out.printf ("Error de argumento:% s \ n", e.errorMessage ());
    }
}
```

Puedes ver lo simple que es esto. Simplemente creamos una instancia de la clase Args con dos parámetros. El primer parámetro es el formato, o *esquema,* cadena: "l, p #, d * " . Define tres argumentos de la línea de comandos. El primero, –l , es un argumento booleano. El segundo, -p , es un número entero argumento. El tercero, -d , es un argumento de cadena. El segundo parámetro del constructor Args es simplemente la matriz de argumentos de la línea de comandos que se pasan a main .

Si el constructor regresa sin lanzar una ArgsException , entonces la entrada Se analizó la línea de comandos y la instancia de Args está lista para ser consultada. Métodos como getBoolean , getInteger y getString nos permiten acceder a los valores de los argumentos mediante sus nombres.

Si hay un problema, ya sea en la cadena de formato o en los argumentos de la línea de comandos ellos mismos, se lanzará una ArgsException . Una descripción conveniente de lo que sucedió incorrecto se puede recuperar del método errorMessage de la excepción.

Implementación de Args

El listado 14-2 es la implementación de la clase Args . Por favor, léalo con mucha atención. Yo trabajé duro con el estilo y la estructura y espero que valga la pena emularlo.

Listado 14-2

Args.java

```
paquete com.objectmentor.utilities.args;

importar com.objectmentor.utilities.args.ArgsException.ErrorCode. * estático;
importar java.util. *;

public class Args {
    marshalers privados de Map <Character, ArgumentMarshaler>;
```


Listado 14-2 (continuación)

Args.java

```
    private Set <Carácter> argsFound;
    Private ListIterator <String> currentArgument;

    public Args (esquema de cadena, String [] args) lanza ArgsException {
        marshalers = new HashMap <Carácter, ArgumentoMarshaler> ();
        argsFound = new HashSet <Carácter> ();
```

```
    parseSchema (esquema);
    parseArgumentStrings (Arrays.asList (args));
}

private void parseSchema (esquema de cadena) lanza ArgsException {
    para (elemento de cadena: esquema.split (","))
        if (element.length ()> 0)
            parseSchemaElement (element.trim ());
}

private void parseSchemaElement (elemento String) lanza ArgsException {
    char elementId = element.charAt (0);
    Cadena elementTail = element.substring (1);
    validateSchemaElementId (elementId);
    if (elementTail.length () == 0)
        marshalers.put (elementId, new BooleanArgumentMarshaler ());
    else if (elementTail.equals ("*"))
        marshalers.put (elementId, nuevo StringArgumentMarshaler ());
    else if (elementTail.equals ("#"))
        marshalers.put (elementId, nuevo IntegerArgumentMarshaler ());
    else if (elementTail.equals ("##"))
        marshalers.put (elementId, nuevo DoubleArgumentMarshaler ());
    else if (elementTail.equals ("[*]"))
        marshalers.put (elementId, nuevo StringArrayArgumentMarshaler ());
    demás
        lanzar nueva ArgsException (INVALID_ARGUMENT_FORMAT, elementId, elementTail);
}

private void validateSchemaElementId (char elementId) lanza ArgsException {
    if (! Character.isLetter (elementId))
        lanzar nueva ArgsException (INVALID_ARGUMENT_NAME, elementId, null);
}

private void parseArgumentStrings (List <String> argsList) lanza ArgsException
{
    para (argumentoActual = listaArgs.listIterador (); argumento actual.hasNext ();)
    {
        String argString = currentArgument.next ();
        if (argString.startsWith ("-")) {
            parseArgumentCharacters (argString.substring (1));
        } demás {
            currentArgument.previous ();
            rotura;
        }
    }
}
```


Página 227

Listado 14-2 (continuación)

Args.java

```
private void parseArgumentCharacters (String argChars) lanza ArgsException {
    para (int i = 0; i <argChars.length (); i ++)
        parseArgumentCharacter (argChars.charAt (i));
}

parseArgumentCharacter vacío privado (char argChar) lanza ArgsException {
    ArgumentMarshaler m = marshalers.get (argChar);
    si (m == nulo) {
        lanzar una nueva ArgsException (UNEXPECTED_ARGUMENT, argChar, null);
    } demás {
        argsFound.add (argChar);
        intentar {
            m.set (argumento actual);
        } captura (ArgsException e) {
            e.setErrorArgumentId (argChar);
            tirar e;
        }
    }
}

public boolean tiene (char arg) {
    return argsFound.contains (arg);
```

```
    }

    public int nextArgument () {
        return currentArgument.nextIndex ();
    }

    public boolean getBoolean (char arg) {
        return BooleanArgumentMarshaler.getValue (marshalers.get (arg));
    }

    public String getString (char arg) {
        return StringArgumentMarshaler.getValue (marshalers.get (arg));
    }

    public int getInt (char arg) {
        return IntegerArgumentMarshaler.getValue (marshalers.get (arg));
    }

    public double getDouble (char arg) {
        return DoubleArgumentMarshaler.getValue (marshalers.get (arg));
    }

    public String [] getStringArray (char arg) {
        return StringArrayArgumentMarshaler.getValue (marshalers.get (arg));
    }
}
```

Tenga en cuenta que puede leer este código de arriba a abajo sin muchos saltos alrededor o mirando hacia adelante. Lo único que puede haber tenido que esperar es la definición ción de ArgumentMarshaler , que omití intencionalmente. Habiendo leído este código detenidamente,

Página 228

debe comprender qué es la interfaz ArgumentMarshaler y qué hacen sus derivados. Le mostraré algunos de ellos ahora (del Listado 14-3 al Listado 14-6).

Listado 14-3

ArgumentoMarshaler.java

```
interfaz pública ArgumentMarshaler {
    void set (Iterator <String> currentArgument) lanza ArgsException;
}
```

Listado 14-4

BooleanArgumentMarshaler.java

```
public class BooleanArgumentMarshaler implementa ArgumentMarshaler {
    private boolean booleanValue = false;

    public void set (Iterator <String> currentArgument) lanza ArgsException {
        booleanValue = verdadero;
    }

    public static boolean getValue (ArgumentMarshaler am) {
        if (am! = null && am instancia de BooleanArgumentMarshaler)
            return ((BooleanArgumentMarshaler) am) .booleanValue;
        demás
            falso retorno;
    }
}
```

Listado 14-5

StringArgumentMarshaler.java

```
importar com.objectmentor.utilities.args.ArgsException.ErrorCode. * estático;

La clase pública StringArgumentMarshaler implementa ArgumentMarshaler {
    private String stringValue = "";

    public void set (Iterator <String> currentArgument) lanza ArgsException {
```

```
        intentar {
            stringValue = currentArgument.next ();
        } captura (NoSuchElementException e) {
            lanzar una nueva ArgsException (MISSING_STRING);
        }
    }

    public static String getValue (ArgumentMarshaler am) {
        si (soy! = nulo && soy una instancia de StringArgumentMarshaler)
            return ((StringArgumentMarshaler) am) .stringValue;
        demás
            regreso "";
    }
}
```


Página 229

Listado 14-6

IntegerArgumentMarshaler.java

```
importar com.objectmentor.utilities.args.ArgsException.ErrorCode. * estático;

La clase pública IntegerArgumentMarshaler implementa ArgumentMarshaler {
    privado int intValue = 0;

    public void set (Iterator <String> currentArgument) lanza ArgsException {
        Parámetro de cadena = nulo;
        intentar {
            parámetro = currentArgument.next ();
            intValue = Integer.parseInt (parámetro);
        } captura (NoSuchElementException e) {
            lanzar una nueva ArgsException (MISSING_INTEGER);
        } catch (NumberFormatException e) {
            lanzar nueva ArgsException (INVALID_INTEGER, parámetro);
        }
    }

    public static int getValue (ArgumentMarshaler am) {
        if (am! = null && am instancia de IntegerArgumentMarshaler)
            return ((IntegerArgumentMarshaler) am) .intValue;
        demás
            return 0;
    }
}
```

Los otros derivados de ArgumentMarshaler simplemente replican este patrón para dobles y Matrices de cadenas y servirían para abarrotar este capítulo. Te los dejo como ejercicio.

Otro dato más podría estar preocupado: la definición del código de error. constantes. Están en la clase ArgsException (Listado 14-7).

Listado 14-7

ArgsException.java

```
importar com.objectmentor.utilities.args.ArgsException.ErrorCode. * estático;

La clase pública ArgsException extiende la excepción {
    carácter privado errorArgumentId = '\ 0';
    Private String errorParameter = null;
    errorCode privado errorCode = OK;

    public ArgsException () {}

    public ArgsException (mensaje de cadena) {super (mensaje);}

    public ArgsException (ErrorCode errorCode) {
        this.errorCode = errorCode;
    }

    public ArgsException (ErrorCode errorCode, String errorParameter) {
        this.errorCode = errorCode;
        this.errorParameter = errorParameter;
    }
```

Página 230

Listado 14-7 (continuación)

ArgsException.java

```
public ArgsException (ErrorCode errorCode,
                      char errorArgumentId, String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
    this.errorArgumentId = errorArgumentId;
}

public char getErrorArgumentId () {
    return errorArgumentId;
}

public void setErrorArgumentId (char errorArgumentId) {
    this.errorArgumentId = errorArgumentId;
}

public String getErrorParameter () {
    return errorParameter;
}

public void setErrorParameter (String errorParameter) {
    this.errorParameter = errorParameter;
}

public ErrorCode getErrorCode () {
    return errorCode;
}

public void setErrorCode (ErrorCode errorCode) {
    this.errorCode = errorCode;
}

public String errorMessage () {
    switch (errorCode) {
        caso OK:
            return "TILT: No debería llegar aquí";
        caso UNEXPECTED_ARGUMENT:
            return String.format ("Argumento -% c inesperado.", errorArgumentId);
        caso MISSING_STRING:
            return String.format ("No se pudo encontrar el parámetro de cadena para -% c.",
                                  errorArgumentId);
        caso INVALID_INTEGER:
            return String.format ("Argumento -% c espera un entero pero era '% s'.",
                                  errorArgumentId, errorParameter);
        caso MISSING_INTEGER:
            return String.format ("No se pudo encontrar el parámetro entero para -% c.",
                                  errorArgumentId);
        caso INVALID_DOUBLE:
            return String.format ("Argumento -% c espera un doble pero era '% s'.",
                                  errorArgumentId, errorParameter);
        caso MISSING_DOUBLE:
            return String.format ("No se pudo encontrar el parámetro doble para -% c.",
                                  errorArgumentId);
        caso INVALID_ARGUMENT_NAME:
            return String.format ("'% c' no es un nombre de argumento válido.",
                                  errorArgumentId);
```

Página 231

Listado 14-7 (continuación)

ArgsException.java

```
            caso INVALID_ARGUMENT_FORMAT:
                return String.format ("'% s' no es un formato de argumento válido.",
                                      errorParameter);
        }
        regreso "";
    }

    public enum ErrorCode {
        OK, INVALID_ARGUMENT_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
        MISSING_STRING,
        MISSING_INTEGER, INVALID_INTEGER,
        MISSING_DOUBLE, INVALID_DOUBLE}
}
```

Es notable la cantidad de código que se requiere para desarrollar los detalles de este simple concepto. cept. Una de las razones de esto es que estamos usando un lenguaje particularmente prolijo. Java, al ser un lenguaje de tipo estático, requiere muchas palabras para satisfacer el sistema de tipos tem. En un lenguaje como Ruby, Python o Smalltalk, este programa es mucho más pequeño. [1]

Lea el código una vez más. Presta especial atención a cómo están las cosas. nombrado, el tamaño de las funciones y el formato del código. Si eres un experimentado programador, es posible que tenga algunas objeciones aquí y allá con varias partes del estilo o estructura. En general, sin embargo, espero que concluya que este programa está bien escrito y tiene una estructura limpia.

Por ejemplo, debería ser obvio cómo agregaría un nuevo tipo de argumento, como un argumento de fecha o un argumento de nmero complejo, y que tal adicin requerir un cantidad de esfuerzo trivial. En resumen, simplemente requeriría una nueva derivada de `Argument-Marshaler` , una nueva función `getXXX` y una nueva declaración de caso en `parseSchemaElement` función. Probablemente también habría un nuevo `ArgsException.ErrorCode` y un nuevo error mensaje.

¿Cómo hice esto?

Déjame descansar tu mente. No escribí simplemente este programa de principio a fin en su forma actual. Más importante aún, no espero que pueda escribir limpio y programas elegantes en una sola pasada. Si hemos aprendido algo durante las últimas dos décadas, es que la programación es un oficio más que una ciencia. Para escribir código limpio, debe primero escriba el código sucio *y luego límpielo* .

Esto no debería ser una sorpresa para ti. Aprendimos esta verdad en la escuela primaria cuando nuestro los profesores intentaron (normalmente en vano) que escribiéramos borradores de nuestras composiciones. La proceso, nos dijeron, era que deberíamos escribir un borrador, luego un segundo borrador, luego varios borradores posteriores hasta que tuvimos nuestra versión final. Escribiendo composiciones limpias, trató de decirnos, es una cuestión de refinamiento sucesivo.

1. Recientemente reescribí este módulo en Ruby. Era 1/7 del tamaño y tenía una estructura sutilmente mejor.

Página 232

La mayoría de los programadores de primer año (como la mayoría de los estudiantes de primaria) no siguen este consejo

particularmente bien. Creen que el objetivo principal es hacer que el programa funcione. Una vez que es "Trabajando", pasan a la siguiente tarea, dejando el programa "funcionando" en cualquier estado finalmente consiguieron que "funcionara". La mayoría de los programadores experimentados saben que esto es profesional. suicidio.

Args: El borrador

El listado 14-8 muestra una versión anterior de la clase Args . Funciona." Y es desordenado.

Listado 14-8

Args.java (primer borrador)

```
import java.text.ParseException;
importar java.util. *;

public class Args {
    esquema de cadena privada;
    Private String [] args;
    privado booleano válido = verdadero;
    conjunto privado <Carácter> Argumentos inesperados = nuevo Conjunto de árbol <Carácter> ();
    Mapa privado <Carácter, Booleano> booleanArgs =
        new HashMap <Carácter, Booleano> ();
    Mapa privado <Carácter, Cadena> stringArgs = new HashMap <Carácter, Cadena> ();
    Private Map <Carácter, Entero> intArgs = new HashMap <Carácter, Entero> ();
    private Set <Character> argsFound = new HashSet <Character> ();
    private int currentArgument;
    carácter privado errorArgumentId = '\ 0';
    private String errorParameter = "TILT";
    errorCode privado errorCode = ErrorCode.OK;

    Private enum ErrorCode {
        OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}

    public Args (esquema de cadena, String [] args) lanza ParseException {
        this.schema = esquema;
        this.args = args;
        válido = analizar ();
    }

    parse () booleano privado lanza ParseException {
        if (schema.length () == 0 && args.length == 0)
            devuelve verdadero;
        parseSchema ();
        intentar {
            parseArguments ();
        } captura (ArgsException e) {
        }
        retorno válido;
    }

    parseSchema booleano privado () lanza ParseException {
        para (Elemento de cadena: esquema.split (",")) {
```


Listado 14-8 (continuación)

Args.java (primer borrador)

```
            if (element.length ()> 0) {
                String trimmedElement = element.trim ();
                parseSchemaElement (trimmedElement);
            }
        }
        devuelve verdadero;
    }

    private void parseSchemaElement (elemento String) lanza ParseException {
        char elementId = element.charAt (0);
        Cadena elementTail = element.substring (1);
        validateSchemaElementId (elementId);
        if (isBooleanSchemaElement (elementTail))
            parseBooleanSchemaElement (elementId);
        else if (isStringSchemaElement (elementTail))
```

```
        parseStringSchemaElement (elementId);
    else if (isIntegerSchemaElement (elementTail)) {
        parseIntegerSchemaElement (elementId);
    } demás {
        lanzar nueva ParseException (
            String.format ("Argumento:% c tiene un formato no válido:% s.",
                                    elementId, elementTail), 0);
    }
}

private void validateSchemaElementId (char elementId) lanza ParseException {
    if (! Character.isLetter (elementId)) {
        lanzar nueva ParseException (
            "Carácter incorrecto:" + elementId + "en formato Args:" + esquema, 0);
    }
}

private void parseBooleanSchemaElement (char elementId) {
    booleanArgs.put (elementId, falso);
}

private void parseIntegerSchemaElement (char elementId) {
    intArgs.put (elementId, 0);
}

private void parseStringSchemaElement (char elementId) {
    stringArgs.put (elementId, "");
}

isStringSchemaElement booleano privado (String elementTail) {
    return elementTail.equals ("*");
}

private boolean isBooleanSchemaElement (String elementTail) {
    return elementTail.length () == 0;
}

private boolean isIntegerSchemaElement (String elementTail) {
    return elementTail.equals ("#");
}
```


Página 234

Listado 14-8 (continuación)

Args.java (primer borrador)

```
parseArguments () booleano privado lanza ArgsException {
    para (currentArgument = 0; currentArgument <args.length; currentArgument ++)
    {
        String arg = args [currentArgument];
        parseArgument (arg);
    }
    devuelve verdadero;
}

private void parseArgument (String arg) lanza ArgsException {
    if (arg.startsWith ("-"))
        parseElements (arg);
}

private void parseElements (String arg) lanza ArgsException {
    para (int i = 1; i <arg.length (); i ++)
        parseElement (arg.charAt (i));
}

parseElement vacío privado (char argChar) lanza ArgsException {
    si (setArgument (argChar))
        argsFound.add (argChar);
    demás {
        inesperadosArgumentos.add (argChar);
        errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
        válido = falso;
    }
}

setArgument booleano privado (char argChar) lanza ArgsException {
    si (isBooleanArg (argChar))
```

```
            setBooleanArg (argChar, verdadero);
        más si (isStringArg (argChar))
            setStringArg (argChar);
        más si (isIntArg (argChar))
            setIntArg (argChar);
        demás
            falso retorno;

        devuelve verdadero;
    }

    isIntArg booleano privado (char argChar) {return intArgs.containsKey (argChar);}

    setIntArg vacío privado (char argChar) lanza ArgsException {
        currentArgument ++;
        Parámetro de cadena = nulo;
        intentar {
            parámetro = args [currentArgument];
            intArgs.put (argChar, nuevo Integer (parámetro));
        } catch (ArrayIndexOutOfBoundsException e) {
            válido = falso;
            errorArgumentId = argChar;
            errorCode = ErrorCode.MISSING_INTEGER;
```


Página 235

Listado 14-8 (continuación)

Args.java (primer borrador)

```
            lanzar nueva ArgsException ();
        } catch (NumberFormatException e) {
            válido = falso;
            errorArgumentId = argChar;
            errorParameter = parámetro;
            errorCode = ErrorCode.INVALID_INTEGER;
            lanzar nueva ArgsException ();
        }
    }

    private void setStringArg (char argChar) lanza ArgsException {
        currentArgument ++;
        intentar {
            stringArgs.put (argChar, args [currentArgument]);
        } catch (ArrayIndexOutOfBoundsException e) {
            válido = falso;
            errorArgumentId = argChar;
            errorCode = ErrorCode.MISSING_STRING;
            lanzar nueva ArgsException ();
        }
    }

    isStringArg booleano privado (char argChar) {
        return stringArgs.containsKey (argChar);
    }

    private void setBooleanArg (char argChar, valor booleano) {
        booleanArgs.put (argChar, valor);
    }

    isBooleanArg booleano privado (char argChar) {
        return booleanArgs.containsKey (argChar);
    }

    public int cardinality () {
        return argsFound.size ();
    }

    uso de cadena pública () {
        if (schema.length ()> 0)
            return "- [" + esquema + "]";
        demás
            regreso "";
    }

    public String errorMessage () lanza Exception {
        switch (errorCode) {
```

```
            caso OK:
                lanzar una nueva excepción ("TILT: no debería llegar aquí");
            caso UNEXPECTED_ARGUMENT:
                return inesperadoArgumentMessage ();
            caso MISSING_STRING:
                return String.format ("No se pudo encontrar el parámetro de cadena para -% c.",
                                      errorArgumentId);
```


Página 236

Listado 14-8 (continuación)

Args.java (primer borrador)

```
            caso INVALID_INTEGER:
                return String.format ("Argumento -% c espera un entero pero era '% s'.",
                                      errorArgumentId, errorParameter);
            caso MISSING_INTEGER:
                return String.format ("No se pudo encontrar el parámetro entero para -% c.",
                                      errorArgumentId);
        }
        regreso "";
    }

    private String unknownArgumentMessage () {
        StringBuffer message = new StringBuffer ("Argumento (s) -");
        para (char c: unknownArguments) {
            message.append (c);
        }
        message.append ("inesperado");

        return message.toString ();
    }

    private boolean falseIfNull (Boolean b) {
        return b! = null && b;
    }

    private int zeroIfNull (Integer i) {
        return i == null? 0: i;
    }

    private String blankIfNull (String s) {
        return s == null? "" : s;
    }

    public String getString (char arg) {
        return blankIfNull (stringArgs.get (arg));
    }

    public int getInt (char arg) {
        return zeroIfNull (intArgs.get (arg));
    }

    public boolean getBoolean (char arg) {
        return falseIfNull (booleanArgs.get (arg));
    }

    public boolean tiene (char arg) {
        return argsFound.contains (arg);
    }

    public boolean isValid () {
        retorno válido;
    }

    La clase privada ArgsException extiende la excepción {
    }
}
```

Página 237

Espero que su reacción inicial a esta masa de código sea "Ciertamente me alegra que no lo haya dejado ¡como eso!" Si te sientes así, recuerda que así es como se sentirán otras personas. sobre el código que deja en forma de borrador.

En realidad, "borrador" es probablemente lo más amable que puede decir sobre este código. Es claramente un trabajo en progreso. La gran cantidad de variables de instancia es abrumadora. El extraño cadenas como "TILT , " los HashSets y TreeSets , y los try-catch-catch bloques se suman a una pila supurante.

No había querido escribir una pila enconada. De hecho, estaba tratando de mantener las cosas razonables. Hábilmente bien organizado. Probablemente pueda decirlo por mi elección de función y variable nombres y el hecho de que el programa tiene una estructura burda. Pero, claramente, había dejado que el problema aléjate de mí.

El desorden se acumuló gradualmente. Las versiones anteriores no habían sido tan desagradables. Por ejemplo, El Listado 14-9 muestra una versión anterior en la que solo funcionaban los argumentos booleanos .

Listado 14-9

Args.java (solo booleano)

```
paquete com.objectmentor.utilities.getopts;

importar java.util. *;

public class Args {
    esquema de cadena privada;
    Private String [] args;
    booleano privado válido;
    conjunto privado <Carácter> Argumentos inesperados = nuevo Conjunto de árbol <Carácter> ();
    Mapa privado <Carácter, Booleano> booleanArgs =
        new HashMap <Carácter, Booleano> ();
    private int numberOfArguments = 0;

    public Args (esquema de cadena, String [] args) {
        this.schema = esquema;
        this.args = args;
        válido = analizar ();
    }

    public boolean isValid () {
        retorno válido;
    }

    análisis booleano privado () {
        if (schema.length () == 0 && args.length == 0)
            devuelve verdadero;
        parseSchema ();
        parseArguments ();
        return unknownArguments.size () == 0;
    }

    parseSchema booleano privado () {
        para (Elemento de cadena: esquema.split (",")) {
            parseSchemaElement (elemento);
        }
```


Página 238

Listado 14-9 (continuación)

Args.java (solo booleano)

```
        devuelve verdadero;
    }

    private void parseSchemaElement (elemento String) {
        if (element.length () == 1) {
            parseBooleanSchemaElement (elemento);
        }
    }

    private void parseBooleanSchemaElement (elemento String) {
        char c = element.charAt (0);
        if (Character.isLetter (c)) {
            booleanArgs.put (c, falso);
        }
    }

    parseArguments booleanos privados () {
        para (String arg: args)
            parseArgument (arg);
        devuelve verdadero;
    }

    parseArgument vacío privado (String arg) {
        if (arg.startsWith ("-"))
            parseElements (arg);
    }

    private void parseElements (String arg) {
        para (int i = 1; i <arg.length (); i ++)
            parseElement (arg.charAt (i));
    }

    parseElement vacío privado (char argChar) {
        if (isBoolean (argChar)) {
            numberOfArguments ++;
            setBooleanArg (argChar, verdadero);
        } demás
            inesperadosArgumentos.add (argChar);
    }

    private void setBooleanArg (char argChar, valor booleano) {
        booleanArgs.put (argChar, valor);
    }

    isBoolean booleano privado (char argChar) {
        return booleanArgs.containsKey (argChar);
    }

    public int cardinality () {
        return numberOfArguments;
    }

    uso de cadena pública () {
        if (schema.length ()> 0)
            return "- [" + esquema + "]";
```


Página 239

Listado 14-9 (continuación)

Args.java (solo booleano)

```
        demás
            regreso "";
    }

    public String errorMessage () {
        if (unknownArguments.size ()> 0) {
            return inesperadoArgumentMessage ();
        } demás
```

```
            regreso "";
        }

        private String unknownArgumentMessage () {
            StringBuffer message = new StringBuffer ("Argumento (s) -");
            para (char c: unknownArguments) {
                message.append (c);
            }
            message.append ("inesperado");

            return message.toString ();
        }

        public boolean getBoolean (char arg) {
            return booleanArgs.get (arg);
        }
    }
```

Aunque puede encontrar mucho de qué quejarse en este código, en realidad no es tan malo. Es compacto, simple y fácil de entender. Sin embargo, dentro de este código es fácil de ver las semillas de la pila supurante posterior. Está bastante claro cómo esto se convirtió en el último lío.

Tenga en cuenta que el último lío tiene solo dos tipos de argumentos más que este: String y entero . La adición de solo dos tipos de argumentos más tuvo un impacto enormemente negativo en el código. Lo convirtió de algo que habría sido razonablemente mantenible en algo que esperaría que estuviera plagado de bichos y verrugas.

Agregué los dos tipos de argumentos de forma incremental. Primero, agregué el argumento String , que produjo esto:

Listado 14-10

Args.java (booleano y cadena)

```
paquete com.objectmentor.utilities.getopts;

import java.text.ParseException;
importar java.util. *;

public class Args {
    esquema de cadena privada;
    Private String [] args;
    privado booleano válido = verdadero;
    conjunto privado <Carácter> Argumentos inesperados = nuevo Conjunto de árbol <Carácter> ();
    Mapa privado <Carácter, Booleano> booleanArgs =
        new HashMap <Carácter, Booleano> ();
```


Página 240

Listado 14-10 (continuación)

Args.java (booleano y cadena)

```
    Mapa privado <Carácter, Cadena> stringArgs =
        new HashMap <Carácter, Cadena> ();
    private Set <Character> argsFound = new HashSet <Character> ();
    private int currentArgument;
    carácter privado errorArgument = '\ 0';

    enum ErrorCode {
        OK, MISSING_STRING}

    errorCode privado errorCode = ErrorCode.OK;

    public Args (esquema de cadena, String [] args) lanza ParseException {
        this.schema = esquema;
        this.args = args;
        válido = analizar ();
    }

    parse () booleano privado lanza ParseException {
        if (schema.length () == 0 && args.length == 0)
            devuelve verdadero;
        parseSchema ();
        parseArguments ();
        retorno válido;
```

```
}
parseSchema booleano privado () lanza ParseException {
    para (Elemento de cadena: esquema.split (",")) {
        if (element.length ()> 0) {
            String trimmedElement = element.trim ();
            parseSchemaElement (trimmedElement);
        }
    }
    devuelve verdadero;
}

private void parseSchemaElement (elemento String) lanza ParseException {
    char elementId = element.charAt (0);
    Cadena elementTail = element.substring (1);
    validateSchemaElementId (elementId);
    if (isBooleanSchemaElement (elementTail))
        parseBooleanSchemaElement (elementId);
    else if (isStringSchemaElement (elementTail))
        parseStringSchemaElement (elementId);
}

private void validateSchemaElementId (char elementId) lanza ParseException {
    if (! Character.isLetter (elementId)) {
        lanzar nueva ParseException (
            "Carácter incorrecto:" + elementId + "en formato Args:" + esquema, 0);
    }

}

private void parseStringSchemaElement (char elementId) {
    stringArgs.put (elementId, "");
}
```


Página 241

Listado 14-10 (continuación)

Args.java (booleano y cadena)

```
isStringSchemaElement booleano privado (String elementTail) {
    return elementTail.equals ("*");
}

private boolean isBooleanSchemaElement (String elementTail) {
    return elementTail.length () == 0;
}

private void parseBooleanSchemaElement (char elementId) {
    booleanArgs.put (elementId, falso);
}

parseArguments booleanos privados () {
    para (currentArgument = 0; currentArgument <args.length; currentArgument ++)
    {
        String arg = args [currentArgument];
        parseArgument (arg);
    }
    devuelve verdadero;
}

parseArgument vacío privado (String arg) {
    if (arg.startsWith ("-"))
        parseElements (arg);
}

private void parseElements (String arg) {
    para (int i = 1; i <arg.length (); i ++)
        parseElement (arg.charAt (i));
}

parseElement vacío privado (char argChar) {
    si (setArgument (argChar))
        argsFound.add (argChar);
    demás {
        inesperadosArgumentos.add (argChar);
        válido = falso;
    }
}
```

```
setArgument booleano privado (char argChar) {
    conjunto booleano = verdadero;
    si (isBoolean (argChar))
        setBooleanArg (argChar, verdadero);
    más si (isString (argChar))
        setStringArg (argChar, "");
    demás
        set = falso;

    conjunto de retorno;
}

private void setStringArg (char argChar, String s) {
    currentArgument ++;
    intentar {
```


Página 242

Listado 14-10 (continuación)

Args.java (booleano y cadena)

```
            stringArgs.put (argChar, args [currentArgument]);
        } catch (ArrayIndexOutOfBoundsException e) {
            válido = falso;
            errorArgument = argChar;
            errorCode = ErrorCode.MISSING_STRING;
        }
    }

    isString booleano privado (char argChar) {
        return stringArgs.containsKey (argChar);
    }

    private void setBooleanArg (char argChar, valor booleano) {
        booleanArgs.put (argChar, valor);
    }

    isBoolean booleano privado (char argChar) {
        return booleanArgs.containsKey (argChar);
    }

    public int cardinality () {
        return argsFound.size ();
    }

    uso de cadena pública () {
        if (schema.length ()> 0)
            return "- [" + esquema + "]";
        demás
            regreso "";
    }

    public String errorMessage () lanza Exception {
        if (unknownArguments.size ()> 0) {
            return inesperadoArgumentMessage ();
        } demás
            switch (errorCode) {
                caso MISSING_STRING:
                    return String.format ("No se pudo encontrar el parámetro de cadena para -% c.",
                                          errorArgument);
                caso OK:
                    lanzar una nueva excepción ("TILT: no debería llegar aquí");
            }
        regreso "";
    }

    private String unknownArgumentMessage () {
        StringBuffer message = new StringBuffer ("Argumento (s) -");
        para (char c: unknownArguments) {
            message.append (c);
        }
        message.append ("inesperado");

        return message.toString ();
    }
```

Página 243

Listado 14-10 (continuación)

Args.java (booleano y cadena)

```
    public boolean getBoolean (char arg) {
        return falseIfNull (booleanArgs.get (arg));
    }

    private boolean falseIfNull (Boolean b) {
        return b == null? falso: b;
    }

    public String getString (char arg) {
        return blankIfNull (stringArgs.get (arg));
    }

    private String blankIfNull (String s) {
        return s == null? "" : s;
    }

    public boolean tiene (char arg) {
        return argsFound.contains (arg);
    }

    public boolean isValid () {
        retorno válido;
    }
}
```

Puede ver que esto está empezando a salirse de control. Todavía no es horrible, pero el lío es sin duda está empezando a crecer. Es una pila, pero todavía no está enconada. Se necesitó la adición de el tipo de argumento entero para que esta pila realmente fermente y supura.

Así que me detuve

Tenía al menos dos tipos de argumentos más para agregar, y podía decir que harían las cosas mucho peor. Si arrastro mi camino hacia adelante, probablemente podría hacer que funcionen, pero dejar atrás un lío que era demasiado grande para arreglarlo. Si la estructura de este código alguna vez ser mantenible, ahora era el momento de arreglarlo.

Así que dejé de agregar funciones y comencé a refactorizar. Habiendo agregado la Cadena y argumentos enteros , sabía que cada tipo de argumento requería un nuevo código en tres lugares. Primero, cada tipo de argumento requería alguna forma de analizar su elemento de esquema en orden para seleccionar el HashMap para ese tipo. A continuación, cada tipo de argumento debe analizarse en el cadenas de línea de comandos y convertidas a su tipo verdadero. Finalmente, cada tipo de argumento necesitaba un getXXX para que pueda devolverse a la persona que llama como su tipo verdadero.

Muchos tipos diferentes, todos con métodos similares, eso me suena como una clase. Y entonces la ArgumentMarshaler concepto nació.

Sobre el incrementalismo

Una de las mejores formas de arruinar un programa es realizar cambios masivos en su estructura en nombre de mejora. Algunos programas nunca se recuperan de tales "mejoras". El problema es ese Es muy difícil hacer que el programa funcione de la misma manera que antes de la "mejora".

Página 244

Para evitar esto, utilizo la disciplina de Desarrollo basado en pruebas (TDD). Uno de los cen-
La doctrina tradicional de este enfoque es mantener el sistema en funcionamiento en todo momento. En otras palabras, usando TDD, no se me permite hacer un cambio en el sistema que rompe ese sistema.
Cada cambio que haga debe mantener el sistema funcionando como antes.

Para lograr esto, necesito un conjunto de pruebas automatizadas que pueda ejecutar por capricho y que ver-
indica que el comportamiento del sistema no ha cambiado. Para la clase de Args , había creado una suite.
de pruebas unitarias y de aceptación mientras estaba construyendo la pila supurante. Las pruebas unitarias fueron escrito en Java y administrado por JUnit . Las pruebas de aceptación se escribieron como páginas wiki.
en FitNesse . Podía ejecutar estas pruebas en cualquier momento que quisiera, y si pasaban, tenía confianza que el sistema estaba funcionando como lo especifiqué.

Entonces procedí a hacer una gran cantidad de cambios muy pequeños. Cada cambio movía el estructura del sistema hacia el concepto ArgumentMarshaler . Y sin embargo, cada cambio se mantuvo el sistema funcionando. El primer cambio que hice fue agregar el esqueleto del ArgumentMarshaller hasta el final de la pila supurante (Listado 14-11).

Listado 14-11
ArgumentMarshaller agregado a Args.java

```
clase privada ArgumentMarshaler {
        private boolean booleanValue = false;

        public void setBoolean (valor booleano) {
            booleanValue = valor;
        }

        public boolean getBoolean () {return booleanValue;}
    }

    clase privada BooleanArgumentMarshaler extiende ArgumentMarshaler {
    }

    la clase privada StringArgumentMarshaler extiende ArgumentMarshaler {
    }

    clase privada IntegerArgumentMarshaler extiende ArgumentMarshaler {
    }
}
```

Claramente, esto no iba a romper nada. Entonces hice la modificación más simple
Podría, uno que se rompa lo menos posible. Cambié el HashMap por el booleano argumentos para tomar un ArgumentMarshaler .

```
mapa privado <Character, ArgumentMarshaler > booleanArgs =
    new HashMap <Carácter, ArgumentMarshaler > ();
```

Esto rompió algunas declaraciones, que arreglé rápidamente.

```
...
    private void parseBooleanSchemaElement (char elementId) {
        booleanArgs.put (elementId, new BooleanArgumentMarshaler () );
    }
..
```


Página 245


```
    private void setBooleanArg (char argChar, valor booleano) {
        booleanArgs. obtener (argChar) .setBoolean (valor);
    }
...
```

```
public boolean getBoolean (char arg) {
    return falseIfNull (booleanArgs.get (arg). getBoolean () );
}
```

Observe cómo estos cambios están exactamente en las áreas que mencioné antes: el análisis , set y get para el tipo de argumento. Desafortunadamente, por pequeño que sea este cambio, algunos de los las pruebas empezaron a fallar. Si observa con atención getBoolean , verá que si lo llama con 'Y , ' pero no hay y argumento, entonces booleanArgs.get ('y') volverá nula , y la función arrojará una NullPointerException . La función falseIfNull se ha utilizado para proteger contra esto, pero el cambio que hice hizo que esa función se volviera irrelevante.

El incrementalismo exigía que esto funcionara rápidamente antes de hacer cualquier otro cambios. De hecho, la solución no fue demasiado difícil. Solo tenía que mover el cheque por nulo . Era ya no el booleano es nulo que necesitaba verificar; era el ArgumentMarshaller .

Primero, eliminé la llamada falseIfNull en la función getBoolean . Era inútil ahora, así que También eliminé la función en sí. Las pruebas todavía fallaron de la misma manera, así que estaba confiado Dent que no había introducido ningún error nuevo.

```
public boolean getBoolean (char arg) {
    return booleanArgs.get (arg) .getBoolean ();
}
```

A continuación, divido la función en dos líneas y pongo ArgumentMarshaller en su propia variante. Argumento nombrado capazMarshaller . No me importó el nombre largo de la variable; estuvo mal redundante y desordenado la función. Así lo acorté a la mañana [N5].

```
public boolean getBoolean (char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get (arg);
    return am .getBoolean ();
}
```

Y luego puse la lógica de detección nula.

```
public boolean getBoolean (char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get (arg);
    return am! = null && am.getBoolean ();
}
```

Argumentos de cadena

Agregar argumentos de cadena fue muy similar a agregar argumentos booleanos . Tuve que cambiar el HashMap y hacer que las funciones parse , set y get funcionen. No debería haber ninguna sorpresa premios en lo que sigue excepto, tal vez, que parece que estoy poniendo todos los implementos mentation en la clase base ArgumentMarshaller en lugar de distribuirla a las derivadas.

```
Mapa privado <Carácter , ArgumentoMarshaler > stringArgs =
        new HashMap <Carácter , ArgumentMarshaler > ();
...
```



```
private void parseStringSchemaElement (char elementId) {
    stringArgs.put (elementId , nuevo StringArgumentMarshaler () );
}
...
private void setStringArg (char argChar) lanza ArgsException {
    currentArgument ++;
    intentar {
        stringArgs .get (argChar) .setString (args [currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
        válido = falso;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_STRING;
        lanzar nueva ArgsException ();
    }
}
...
public String getString (char arg) {
    Args.ArgumentMarshaler am = stringArgs.get (arg);
    return am == null? "" : am.getString ();
```

```
... }
    clase privada ArgumentMarshaler {
        private boolean booleanValue = false;
        private String stringValue;

        public void setBoolean (valor booleano) {
            booleanValue = valor;
        }

        public boolean getBoolean () {
            return booleanValue;
        }

        public void setString (String s) {
            stringValue = s;
        }

        public String getString () {
            return stringValue == null? "" : valor de cadena;
        }
    }
```

Nuevamente, estos cambios se realizaron uno a la vez y de tal manera que las pruebas se mantuvieron corriendo, si no pasando. Cuando se rompió una prueba, me aseguré de que pasara nuevamente antes de Continuando con el próximo cambio.

A estas alturas deberías poder ver mi intención. Una vez que obtenga toda la clasificación actual comportamiento en la clase base ArgumentMarshaler , voy a empezar a impulsar ese comportamiento hacia abajo en las derivadas. Esto me permitirá mantener todo funcionando mientras gradualmente cambiar la forma de este programa.

El siguiente paso obvio fue mover la funcionalidad del argumento int a la ArgumentoMarshaler . Nuevamente, no hubo sorpresas.

```
Mapa privado <Carácter , ArgumentoMarshaler > intArgs =
        new HashMap <Carácter , ArgumentMarshaler > ();
...
```


Página 247


```
    private void parseIntegerSchemaElement (char elementId) {
        intArgs.put (elementId, nuevo IntegerArgumentMarshaler () );
    }
...
    setIntArg vacío privado (char argChar) lanza ArgsException {
        currentArgument ++;
        Parámetro de cadena = nulo;
        intentar {
            parámetro = args [currentArgument];
            intArgs .get (argChar) .setInteger (Integer.parseInt (parámetro));
        } catch (ArrayIndexOutOfBoundsException e) {
            válido = falso;
            errorArgumentId = argChar;
            errorCode = ErrorCode.MISSING_INTEGER;
            lanzar nueva ArgsException ();
        } catch (NumberFormatException e) {
            válido = falso;
            errorArgumentId = argChar;
            errorParameter = parámetro;
            errorCode = ErrorCode.INVALID_INTEGER;
            lanzar nueva ArgsException ();
        }
    }
...
    public int getInt (char arg) {
        Args.ArgumentMarshaler am = intArgs.get (arg);
        return am == null? 0 : am.getInteger ();
    }
...
    clase privada ArgumentMarshaler {
        private boolean booleanValue = false;
        private String stringValue;
        private int integerValue;

        public void setBoolean (valor booleano) {
```

```
    } booleanValue = valor;

    public boolean getBoolean () {
        return booleanValue;
    }

    public void setString (String s) {
        stringValue = s;
    }

    public String getString () {
        return stringValue == null? "" : valor de cadena;
    }

    public void setInteger (int i) {
        integerValue = i;
    }

    public int getInteger () {
        return integerValue;
    }
}
```


Página 248

Con toda la clasificación trasladada al ArgumentMarshaler , comencé a presionar funcionalidad en los derivados. El primer paso fue mover la función setBoolean a la BooleanArgumentMarshaller y asegúrese de que se haya llamado correctamente. Entonces creé un resumen establecer el método.

```
clase abstracta privada ArgumentMarshaler {
    protected boolean booleanValue = false;
    private String stringValue;
    private int integerValue;

    public void setBoolean (valor booleano) {
        booleanValue = valor;
    }

    public boolean getBoolean () {
        return booleanValue;
    }

    public void setString (String s) {
        stringValue = s;
    }

    public String getString () {
        return stringValue == null? "" : valor de cadena;
    }

    public void setInteger (int i) {
        integerValue = i;
    }

    public int getInteger () {
        return integerValue;
    }

    conjunto vacío abstracto público (String s);
}
```

Luego implementé el método set en BooleanArgumentMarshaller .

```
clase privada BooleanArgumentMarshaler extiende ArgumentMarshaler {
    conjunto vacío público (String s) {
        booleanValue = verdadero;
    }
}
```

Y finalmente reemplacé la llamada a setBoolean con una llamada a set .

```
private void setBooleanArg (char argChar, valor booleano) {
    booleanArgs.get (argChar) .set ("verdadero");
}
```

Todas las pruebas aún pasaron. Debido a que este cambio hizo que el conjunto se implementara en el booleano-ArgumentMarshaler , eliminé el método setBoolean de la base ArgumentMarshaler clase.

Observe que la función de conjunto abstracto toma un argumento String , pero la implementación en el BooleanArgumentMarshaller no lo usa. Puse ese argumento ahí porque yo sabía que StringArgumentMarshaller y IntegerArgumentMarshaller *lo* usarían.

Página 249

A continuación, quería implementar el método get en BooleanArgumentMarshaler . Despliegue obtener funciones siempre es feo porque el tipo de retorno tiene que ser Object , y en este caso debe convertirse en un booleano .

```
public boolean getBoolean (char arg) {
        Args.ArgumentMarshaler am = booleanArgs.get (arg);
        devuelve am! = null && (booleano) am. obtener ();
    }
```

Solo para que esto se compile, agregué la función get al ArgumentMarshaler .

```
clase abstracta privada ArgumentMarshaler {
        ...

        public Object get () {
            devolver nulo;
        }
    }
```

Esto compiló y obviamente falló las pruebas. Conseguir que las pruebas funcionen de nuevo fue simplemente una cuestión de hacer que se vuelva abstracto e implementarlo en BooleanAgumentMarshaler .

```
clase abstracta privada ArgumentMarshaler {
        protected boolean booleanValue = false;
        ...

        Objeto abstracto público get ();
    }

    clase privada BooleanArgumentMarshaler extiende ArgumentMarshaler {
        conjunto vacío público (String s) {
            booleanValue = verdadero;
        }

        public Object get () {
            return booleanValue;
        }
    }
```

Una vez más pasaron las pruebas. ¡Así que obtienen y configuran la implementación en BooleanArgumentMarshaler ! Esto me permitió eliminar la antigua función getBoolean de ArgumentMarshaler , mover el protected booleanValue hasta BooleanArgumentMarshaler y convertirla en privada .

Hice el mismo patrón de cambios para Strings . Implementé set y get , eliminé el funciones no utilizadas, y movió las variables.

```
private void setStringArg (char argChar) lanza ArgsException {
        currentArgument ++;
        intentar {
            stringArgs.get (argChar). set (args [currentArgument]);
        } catch (ArrayIndexOutOfBoundsException e) {
            válido = falso;
            errorArgumentId = argChar;
            errorCode = ErrorCode.MISSING_STRING;
            lanzar nueva ArgsException ();
        }
    }
```

Página 250


```
...
    public String getString (char arg) {
        Args.ArgumentMarshaler am = stringArgs.get (arg);
        return am == null? "": (Cadena) am. obtener ();
    }
...
    clase abstracta privada ArgumentMarshaler {
        private int integerValue;

        public void setInteger (int i) {
            integerValue = i;
        }

        public int getInteger () {
            return integerValue;
        }

        conjunto vacío abstracto público (String s);

        Objeto abstracto público get ();
    }

    clase privada BooleanArgumentMarshaler extiende ArgumentMarshaler {
        private boolean booleanValue = false;

        conjunto vacío público (String s) {
            booleanValue = verdadero;
        }

        public Object get () {
            return booleanValue;
        }
    }

    la clase privada StringArgumentMarshaler extiende ArgumentMarshaler {
        private String stringValue = "";

        conjunto vacío público (String s) {
            stringValue = s;
        }

        public Object get () {
            return stringValue;
        }
    }

    clase privada IntegerArgumentMarshaler extiende ArgumentMarshaler {

        conjunto vacío público (String s) {

        }

        public Object get () {
            devolver nulo;
        }
    }
}
```


Página 251

Finalmente, repetí el proceso para números enteros . Esto fue solo un poco más complicado porque los enteros necesitaban ser analizados, y la operación de análisis puede generar una excepción. Pero el resultado es mejor porque todo el concepto de NumberFormatException quedó enterrado en el IntegerArgumentMarshaler .

```
isIntArg booleano privado (char argChar) {return intArgs.containsKey (argChar);}

    setIntArg vacío privado (char argChar) lanza ArgsException {
        currentArgument ++;
        Parámetro de cadena = nulo;
        intentar {
            parámetro = args [currentArgument];
            intArgs.get (argChar). set (parámetro);
        } catch (ArrayIndexOutOfBoundsException e) {
            válido = falso;
            errorArgumentId = argChar;
            errorCode = ErrorCode.MISSING_INTEGER;
            lanzar nueva ArgsException ();
        } captura ( ArgsException e) {
            válido = falso;
            errorArgumentId = argChar;
            errorParameter = parámetro;
            errorCode = ErrorCode.INVALID_INTEGER;
            tirar e ;
        }
    }
...
    private void setBooleanArg (char argChar) {
        intentar {
            booleanArgs.get (argChar) .set ("verdadero");
        } captura (ArgsException e) {
        }
    }
...
    public int getInt (char arg) {
        Args.ArgumentMarshaler am = intArgs.get (arg);
        return am == null? 0: (Entero) am. obtener ();
    }
...
    clase abstracta privada ArgumentMarshaler {
        el conjunto vacío abstracto público (String s) lanza ArgsException;
        Objeto abstracto público get ();
    }
...
    clase privada IntegerArgumentMarshaler extiende ArgumentMarshaler {
        privado int intValue = 0;

        public void set (String s) lanza ArgsException {
            intentar {
                intValue = Integer.parseInt (s);
            } catch (NumberFormatException e) {
                lanzar nueva ArgsException ();
            }
        }

        public Object get () {
            return intValue;
        }
    }
```


Página 252

Por supuesto, las pruebas continuaron pasando. A continuación, me deshice de los tres mapas diferentes en la parte superior del algoritmo. Esto hizo que todo el sistema fuera mucho más genérico. Sin embargo, yo no podía deshacerse de ellos simplemente eliminándolos porque eso rompería el sistema.
En su lugar, agregué un nuevo mapa para ArgumentMarshaler y luego, uno por uno, cambié el métodos para usarlo en lugar de los tres mapas originales.

```
public class Args {
...
    mapa privado <Character, ArgumentMarshaler> booleanArgs =
        new HashMap <Carácter, ArgumentMarshaler> ();
    Mapa privado <Carácter, ArgumentoMarshaler> stringArgs =
```

```
    new HashMap <Carácter, ArgumentMarshaler> ();
Mapa privado <Carácter, ArgumentoMarshaler> intArgs =
    new HashMap <Carácter, ArgumentMarshaler> ();
mapa privado <Personaje, ArgumentoMarshaler> marshalers =
    new HashMap <Carácter, ArgumentMarshaler> ();
...
private void parseBooleanSchemaElement (char elementId) {
    ArgumentMarshaler m = new BooleanArgumentMarshaler ();
    booleanArgs.put (elementId, m);
    marshalers.put (elementId, m);
}

private void parseIntegerSchemaElement (char elementId) {
    ArgumentMarshaler m = new IntegerArgumentMarshaler ();
    intArgs.put (elementId, m);
    marshalers.put (elementId, m);
}

private void parseStringSchemaElement (char elementId) {
    ArgumentMarshaler m = nuevo StringArgumentMarshaler ();
    stringArgs.put (elementId, m);
    marshalers.put (elementId, m);
}
```

Por supuesto, todas las pruebas todavía pasaron. A continuación, cambié isBooleanArg de esto:

```
isBooleanArg booleano privado (char argChar) {
    return booleanArgs.containsKey (argChar);
}
```

a esto:

```
isBooleanArg booleano privado (char argChar) {
    ArgumentMarshaler m = marshalers.get (argChar);
    return m instancia de BooleanArgumentMarshaler;
}
```

Las pruebas aún pasaron. Así que hice el mismo cambio en isIntArg e isStringArg .

```
isIntArg booleano privado (char argChar) {
    ArgumentMarshaler m = marshalers.get (argChar);
    return m instancia de IntegerArgumentMarshaler;
}

isStringArg booleano privado (char argChar) {
    ArgumentMarshaler m = marshalers.get (argChar);
    return m instancia de StringArgumentMarshaler;
}
```


Página 253

Las pruebas aún pasaron. Así que eliminé todas las llamadas duplicadas a marshalers.get de la siguiente manera:

```
setArgument booleano privado (char argChar) lanza ArgsException {
    ArgumentMarshaler m = marshalers.get (argChar);
    si (isBooleanArg ( m ))
        setBooleanArg (argChar);
    más si (isStringArg ( m ))
        setStringArg (argChar);
    más si (isIntArg ( m ))
        setIntArg (argChar);
    demás
        falso retorno;

    devuelve verdadero;
}

isIntArg booleano privado ( ArgumentMarshaler m ) {
    return m instancia de IntegerArgumentMarshaler;
}

isStringArg booleano privado ( ArgumentMarshaler m ) {
    return m instancia de StringArgumentMarshaler;
}

private boolean isBooleanArg ( ArgumentMarshaler m ) {
    return m instancia de BooleanArgumentMarshaler;
}
```

Esto no dejó una buena razón para los tres métodos isxxxArg . Así que los incluí:

```
setArgument booleano privado (char argChar) lanza ArgsException {
  ArgumentMarshaler m = marshalers.get (argChar);
  if ( m instancia de BooleanArgumentMarshaler )
    setBooleanArg (argChar);
  else if ( m instancia de StringArgumentMarshaler )
    setStringArg (argChar);
  else if ( m instancia de IntegerArgumentMarshaler )
    setIntArg (argChar);
  demás
    falso retorno;

  devuelve verdadero;
}
```

A continuación, comencé a usar el mapa de marshalers en las funciones establecidas , rompiendo el uso de las otras tres mapas. Empecé con los booleanos .

```
setArgument booleano privado (char argChar) lanza ArgsException {
  ArgumentMarshaler m = marshalers.get (argChar);
  if (m instancia de BooleanArgumentMarshaler)
    setBooleanArg ( m );
  else if (m instancia de StringArgumentMarshaler)
    setStringArg (argChar);
  else if (m instancia de IntegerArgumentMarshaler)
    setIntArg (argChar);
  demás
    falso retorno;
```


Página 254


```
    devuelve verdadero;
  }
...
  private void setBooleanArg ( ArgumentMarshaler m ) {
    intentar {
      m .set ("verdadero"); // era: booleanArgs.get (argChar) .set ("true");
    } captura (ArgsException e) {
    }
  }
```

Las pruebas aún pasaron, así que hice lo mismo con Strings e Integers . Esto me permitió inte-
Grabe algunos de los feos códigos de gestión de excepciones en la función setArgument .

```
setArgument booleano privado (char argChar) lanza ArgsException {
  ArgumentMarshaler m = marshalers.get (argChar);
  intentar {
    if (m instancia de BooleanArgumentMarshaler)
      setBooleanArg (m);
    else if (m instancia de StringArgumentMarshaler)
      setStringArg ( m );
    else if (m instancia de IntegerArgumentMarshaler)
      setIntArg ( m );
    demás
      falso retorno;
  } captura (ArgsException e) {
    válido = falso;
    errorArgumentId = argChar;
    tirar e;
  }
  devuelve verdadero;
}

private void setIntArg ( ArgumentMarshaler m ) lanza ArgsException {
  currentArgument ++;
  Parámetro de cadena = nulo;
  intentar {
    parámetro = args [currentArgument];
    m .set (parámetro);
  } catch (ArrayIndexOutOfBoundsException e) {
    errorCode = ErrorCode.MISSING_INTEGER;
    lanzar nueva ArgsException ();
  } captura (ArgsException e) {
    errorParameter = parámetro;
    errorCode = ErrorCode.INVALID_INTEGER;
```

```
            tirar e;
        }
    }

    private void setStringArg ( ArgumentMarshaler m ) lanza ArgsException {
        currentArgument ++;
        intentar {
            m .set (args [currentArgument]);
        } catch (ArrayIndexOutOfBoundsException e) {
            errorCode = ErrorCode.MISSING_STRING;
            lanzar nueva ArgsException ();
        }
    }
```


Página 255

Estuve cerca de poder eliminar los tres mapas antiguos. Primero, necesitaba cambiar el getBoolean función de esto:

```
    public boolean getBoolean (char arg) {
        Args.ArgumentMarshaler am = booleanArgs.get (arg);
        return am! = null && (Boolean) am.get ();
    }
```

a esto:

```
public boolean getBoolean (char arg) {
    Args.ArgumentMarshaler am = marshalers.get (arg);
    booleano b = falso;
    intentar {
        b = am! = null && (booleano) am.get ();
    } captura (ClassCastException e) {
        b = falso;
    }
    volver b;
}
```

Este último cambio podría haber sido una sorpresa. ¿Por qué de repente decidí lidiar con la ClassCastException ? La razón es que tengo un conjunto de pruebas unitarias y un conjunto separado de pruebas de aceptación escritas en FitNesse. Resulta que las pruebas de FitNesse aseguraron que si usted llamó a getBoolean con un argumento no booleano, obtuvo un falso . Las pruebas unitarias no lo hicieron. Hasta este punto, solo había estado ejecutando las pruebas unitarias. [2]

Este último cambio me permitió sacar otro uso del mapa booleano :

```
    private void parseBooleanSchemaElement (char elementId) {
        ArgumentMarshaler m = new BooleanArgumentMarshaler ();
        booleanArgs.put (elementId, m);
        marshalers.put (elementId, m);
    }
```

Y ahora podemos eliminar el mapa booleano .

```
public class Args {
...
    mapa privado <Character, ArgumentMarshaler> booleanArgs =
     new HashMap <Carácter, ArgumentMarshaler> ();
     Mapa privado <Carácter, ArgumentoMarshaler> stringArgs =
     new HashMap <Carácter, ArgumentMarshaler> ();
    Mapa privado <Carácter, ArgumentoMarshaler> intArgs =
     new HashMap <Carácter, ArgumentMarshaler> ();
    mapa privado <Personaje, ArgumentoMarshaler> marshalers =
     new HashMap <Carácter, ArgumentMarshaler> ();
...
```

A continuación, migré los argumentos String e Integer de la misma manera e hice un poco limpieza con los booleanos .

```
    private void parseBooleanSchemaElement (char elementId) {
        marshalers.put (elementId, new BooleanArgumentMarshaler () );
    }
```

2. Para evitar más sorpresas de este tipo, agregué una nueva prueba unitaria que invocaba todas las pruebas de FitNesse.

Página 256


```
    private void parseIntegerSchemaElement (char elementId) {
        marshalers.put (elementId, nuevo IntegerArgumentMarshaler () );
    }

    private void parseStringSchemaElement (char elementId) {
        marshalers.put (elementId, nuevo StringArgumentMarshaler () );
    }
...
    public String getString (char arg) {
        Args.ArgumentMarshaler am = marshalers .get (arg);
        intentar {
            return am == null? "": (Cadena) am.get ();
        } captura (ClassCastException e) {
            regreso "";
        }
    }

    public int getInt (char arg) {
        Args.ArgumentMarshaler am = marshalers .get (arg);
        intentar {
            return am == null? 0: (Entero) am.get ();
        } captura (Excepción e) {
            return 0;
        }
    }
...
public class Args {
...
    Mapa privado <Carácter, ArgumentoMarshaler> stringArgs =
     new HashMap <Carácter, ArgumentMarshaler> ();
    Mapa privado <Carácter, ArgumentoMarshaler> intArgs =
     new HashMap <Carácter, ArgumentMarshaler> ();
     mapa privado <Personaje, ArgumentoMarshaler> marshalers =
     new HashMap <Carácter, ArgumentMarshaler> ();
...
```

A continuación, incluí los tres métodos de análisis porque ya no hacían mucho:

```
 private void parseSchemaElement (elemento String) lanza ParseException {
    char elementId = element.charAt (0);
    Cadena elementTail = element.substring (1);
    validateSchemaElementId (elementId);
    if (isBooleanSchemaElement (elementTail))
        marshalers.put (elementId, new BooleanArgumentMarshaler ());
    else if (isStringSchemaElement (elementTail))
        marshalers.put (elementId, nuevo StringArgumentMarshaler ());
    else if (isIntegerSchemaElement (elementTail)) {
        marshalers.put (elementId, nuevo IntegerArgumentMarshaler ());
    } demás {
        lanzar una nueva ParseException (String.format (
         "Argumento:% c tiene un formato no válido:% s.", ElementId, elementTail), 0);
    }
}
```

Bien, ahora veamos el panorama completo nuevamente. El Listado 14-12 muestra la corriente forma de la clase Args .

Página 257

Listado 14-12

Args.java (después de la primera refactorización)

```
paquete com.objectmentor.utilities.getopts;

import java.text.ParseException;
importar java.util. *;

public class Args {
    esquema de cadena privada;
    Private String [] args;
    privado booleano válido = verdadero;
    conjunto privado <Carácter> Argumentos inesperados = nuevo Conjunto de árbol <Carácter> ();
    mapa privado <Personaje, ArgumentoMarshaler> marshalers =
     new HashMap <Carácter, ArgumentMarshaler> ();
    private Set <Character> argsFound = new HashSet <Character> ();
    private int currentArgument;
    carácter privado errorArgumentId = '\ 0';
    private String errorParameter = "TILT";
    errorCode privado errorCode = ErrorCode.OK;

    Private enum ErrorCode {
        OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}

    public Args (esquema de cadena, String [] args) lanza ParseException {
        this.schema = esquema;
        this.args = args;
        válido = analizar ();
    }

    parse () booleano privado lanza ParseException {
        if (schema.length () == 0 && args.length == 0)
            devuelve verdadero;
        parseSchema ();
        intentar {
            parseArguments ();
        } captura (ArgsException e) {
        }
        retorno válido;
    }

    parseSchema booleano privado () lanza ParseException {
        para (Elemento de cadena: esquema.split (",")) {
            if (element.length ()> 0) {
                String trimmedElement = element.trim ();
                parseSchemaElement (trimmedElement);
            }
        }
        devuelve verdadero;
    }

    private void parseSchemaElement (elemento String) lanza ParseException {
        char elementId = element.charAt (0);
        Cadena elementTail = element.substring (1);
        validateSchemaElementId (elementId);
        if (isBooleanSchemaElement (elementTail))
            marshalers.put (elementId, new BooleanArgumentMarshaler ());
        else if (isStringSchemaElement (elementTail))
            marshalers.put (elementId, nuevo StringArgumentMarshaler ());
```


Página 258

Listado 14-12 (continuación)

Args.java (después de la primera refactorización)

```
        else if (isIntegerSchemaElement (elementTail)) {
            marshalers.put (elementId, nuevo IntegerArgumentMarshaler ());
```

```
    } demás {
        lanzar una nueva ParseException (String.format (
         "Argumento:% c tiene un formato no válido:% s.", ElementId, elementTail), 0);
    }
}

private void validateSchemaElementId (char elementId) lanza ParseException {
    if (! Character.isLetter (elementId)) {
        lanzar nueva ParseException (
         "Carácter incorrecto:" + elementId + "en formato Args:" + esquema, 0);
    }
}

isStringSchemaElement booleano privado (String elementTail) {
    return elementTail.equals ("*");
}

private boolean isBooleanSchemaElement (String elementTail) {
    return elementTail.length () == 0;
}

private boolean isIntegerSchemaElement (String elementTail) {
    return elementTail.equals ("#");
}

parseArguments () booleano privado lanza ArgsException {
    for (currentArgument = 0; currentArgument <args.length; currentArgument ++) {
        String arg = args [currentArgument];
        parseArgument (arg);
    }
    devuelve verdadero;
}

private void parseArgument (String arg) lanza ArgsException {
    if (arg.startsWith ("-"))
        parseElements (arg);
}

private void parseElements (String arg) lanza ArgsException {
    para (int i = 1; i <arg.length (); i ++)
        parseElement (arg.charAt (i));
}

parseElement vacío privado (char argChar) lanza ArgsException {
    si (setArgument (argChar))
        argsFound.add (argChar);
    demás {
        inesperadosArgumentos.add (argChar);
        errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
        válido = falso;
    }
}
```


Página 259

Listado 14-12 (continuación)

Args.java (después de la primera refactorización)

```
setArgument booleano privado (char argChar) lanza ArgsException {
    ArgumentMarshaler m = marshalers.get (argChar);
    intentar {
        if (m instancia de BooleanArgumentMarshaler)
            setBooleanArg (m);
        else if (m instancia de StringArgumentMarshaler)
            setStringArg (m);
        else if (m instancia de IntegerArgumentMarshaler)
            setIntArg (m);
        demás
            falso retorno;
    } captura (ArgsException e) {
        válido = falso;
        errorArgumentId = argChar;
        tirar e;
    }
    devuelve verdadero;
}
```

```
private void setIntArg (ArgumentMarshaler m) lanza ArgsException {
    currentArgument ++;
    Parámetro de cadena = nulo;
    intentar {
        parámetro = args [currentArgument];
        m.set (parámetro);
    } catch (ArrayIndexOutOfBoundsException e) {
        errorCode = ErrorCode.MISSING_INTEGER;
        lanzar nueva ArgsException ();
    } captura (ArgsException e) {
        errorParameter = parámetro;
        errorCode = ErrorCode.INVALID_INTEGER;
        tirar e;
    }
}

private void setStringArg (ArgumentMarshaler m) lanza ArgsException {
    currentArgument ++;
    intentar {
        m.set (args [currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
        errorCode = ErrorCode.MISSING_STRING;
        lanzar nueva ArgsException ();
    }
}

private void setBooleanArg (ArgumentMarshaler m) {
    intentar {
        m.set ("verdadero");
    } captura (ArgsException e) {
    }
}

public int cardinality () {
    return argsFound.size ();
}
```


Página 260

Listado 14-12 (continuación)

Args.java (después de la primera refactorización)

```
uso de cadena pública () {
    if (schema.length ()> 0)
        return "- [" + esquema + "]";
    demás
        regreso "";
}

public String errorMessage () lanza Exception {
    switch (errorCode) {
        caso OK:
            lanzar una nueva excepción ("TILT: no debería llegar aquí");
        caso UNEXPECTED_ARGUMENT:
            return inesperadoArgumentMessage ();
        caso MISSING_STRING:
            return String.format ("No se pudo encontrar el parámetro de cadena para -% c.",
                                  errorArgumentId);
        caso INVALID_INTEGER:
            return String.format ("Argumento -% c espera un entero pero era '% s'.",
                                  errorArgumentId, errorParameter);
        caso MISSING_INTEGER:
            return String.format ("No se pudo encontrar el parámetro entero para -% c.",
                                  errorArgumentId);
    }
    regreso "";
}

private String unknownArgumentMessage () {
    StringBuffer message = new StringBuffer ("Argumento (s) -");
    para (char c: unknownArguments) {
        message.append (c);
    }
    message.append ("inesperado");

    return message.toString ();
```

```
}

public boolean getBoolean (char arg) {
    Args.ArgumentMarshaler am = marshalers.get (arg);
    booleano b = falso;
    intentar {
        b = am! = null && (booleano) am.get ();
    } captura (ClassCastException e) {
        b = falso;
    }
    volver b;
}

public String getString (char arg) {
    Args.ArgumentMarshaler am = marshalers.get (arg);
    intentar {
        return am == null? "": (Cadena) am.get ();
    } captura (ClassCastException e) {
        regreso "";
    }
}
```


Página 261

Listado 14-12 (continuación)

Args.java (después de la primera refactorización)

```
public int getInt (char arg) {
    Args.ArgumentMarshaler am = marshalers.get (arg);
    intentar {
        return am == null? 0: (Entero) am.get ();
    } captura (Excepción e) {
        return 0;
    }
}

public boolean tiene (char arg) {
    return argsFound.contains (arg);
}

public boolean isValid () {
    retorno válido;
}

La clase privada ArgsException extiende la excepción {
}

clase abstracta privada ArgumentMarshaler {
    el conjunto vacío abstracto público (String s) lanza ArgsException;
    Objeto abstracto público get ();
}

clase privada BooleanArgumentMarshaler extiende ArgumentMarshaler {
    private boolean booleanValue = false;

    conjunto vacío público (String s) {
        booleanValue = verdadero;
    }

    public Object get () {
        return booleanValue;
    }
}

la clase privada StringArgumentMarshaler extiende ArgumentMarshaler {
    private String stringValue = "";

    conjunto vacío público (String s) {
        stringValue = s;
    }

    public Object get () {
        return stringValue;
    }
}

clase privada IntegerArgumentMarshaler extiende ArgumentMarshaler {
```

```
privado int intValue = 0;

public void set (String s) lanza ArgsException {
    intentar {
        intValue = Integer.parseInt (s);
```


Página 262

Listado 14-12 (continuación)
Args.java (después de la primera refactorización)

```
        } catch (NumberFormatException e) {
            lanzar nueva ArgsException ();
        }
    }

    public Object get () {
        return intValue;
    }
}
}
```

Después de todo ese trabajo, esto es un poco decepcionante. La estructura es un poco mejor, pero todavía tener todas esas variables en la parte superior; todavía hay un tipo de caso horrible en setArgument ; y todas esas funciones establecidas son realmente feas. Sin mencionar todo el procesamiento de errores. Aún tenemos mucho trabajo por delante.

Realmente me gustaría deshacerme de ese tipo de caso en setArgument [G23]. Lo que me gustaría en setArgument es una única llamada a ArgumentMarshaler.set . Esto significa que necesito empujar setIntArg , setStringArg y setBooleanArg en el ArgumentMarshaler apropiado derivados. Pero hay un problema.

Si observa de cerca setIntArg , notará que usa dos variables de instancia: args y currentArg . Para mover setIntArg hacia abajo en BooleanArgumentMarshaler , tendré que pasar tanto args como currentArgs como argumentos de función. Eso es sucio [F1]. Prefiero pasar uno argumento en lugar de dos. Afortunadamente, existe una solución sencilla. Podemos convertir los argumentos matriz en una lista y pasar un iterador a las funciones establecidas . Lo siguiente me llevó diez pasos, pasando todas las pruebas después de cada uno. Pero solo te mostraré el resultado. Usted debería ser capaz de averiguar cuáles eran la mayoría de los pequeños pasos.

```
public class Args {
    esquema de cadena privada;
    Private String [] args;
    privado booleano válido = verdadero;
    conjunto privado <Carácter> Argumentos inesperados = nuevo Conjunto de árbol <Carácter> ();
    mapa privado <Personaje, ArgumentoMarshaler> marshalers =
     new HashMap <Carácter, ArgumentMarshaler> ();
    private Set <Character> argsFound = new HashSet <Character> ();
    private Iterator <String> currentArgument;
    carácter privado errorArgumentId = '\ 0';
    private String errorParameter = "TILT";
    errorCode privado errorCode = ErrorCode.OK;
    Private List <String> argsList;

    Private enum ErrorCode {
        OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}

    public Args (esquema de cadena, String [] args) lanza ParseException {
        this.schema = esquema;
        argsList = Arrays.asList (args);
        válido = analizar ();
    }
```

Página 263


```
    parse () booleano privado lanza ParseException {
        if (schema.length () == 0 && argsList.size () == 0)
            devuelve verdadero;
        parseSchema ();
        intentar {
            parseArguments ();
        } captura (ArgsException e) {
        }
        retorno válido;
    }
---
    parseArguments () booleano privado lanza ArgsException {
        for (currentArgument = argsList.iterator () ; currentArgument. hasNext () ;) {
            Cadena arg = argumento actual. siguiente () ;
            parseArgument (arg);
        }

        devuelve verdadero;
    }
---
    private void setIntArg (ArgumentMarshaler m) lanza ArgsException {
        Parámetro de cadena = nulo;
        intentar {
            parámetro = currentArgument. siguiente () ;
            m.set (parámetro);
        } captura ( NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_INTEGER;
            lanzar nueva ArgsException ();
        } captura (ArgsException e) {
            errorParameter = parámetro;
            errorCode = ErrorCode.INVALID_INTEGER;
            tirar e;
        }
    }

    private void setStringArg (ArgumentMarshaler m) lanza ArgsException {
        intentar {
            m.set (currentArgument .next () );
        } captura ( NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_STRING;
            lanzar nueva ArgsException ();
        }
    }
```

Estos fueron cambios simples que mantuvieron aprobadas todas las pruebas. Ahora podemos empezar a mover el set funciona en las derivadas apropiadas. Primero, necesito hacer el siguiente cambio en setArgument :

```
    setArgument booleano privado (char argChar) lanza ArgsException {
        ArgumentMarshaler m = marshalers.get (argChar);
        si (m == nulo)
            falso retorno;
        intentar {
            if (m instancia de BooleanArgumentMarshaler)
                setBooleanArg (m);
            else if (m instancia de StringArgumentMarshaler)
                setStringArg (m);
            else if (m instancia de IntegerArgumentMarshaler)
                setIntArg (m);
```


Página 264

```
            demás
                falso retorno;
        } captura (ArgsException e) {
            válido = falso;
            errorArgumentId = argChar;
            tirar e;
        }
        devuelve verdadero;
    }
```

Este cambio es importante porque queremos eliminar por completo la cadena if-else .
Por lo tanto, necesitábamos eliminar la condición de error.

Ahora podemos empezar a mover las funciones establecidas . La función setBooleanArg es trivial, por lo que lo prepararemos primero. Nuestro objetivo es cambiar la función setBooleanArg para simplemente forward al BooleanArgumentMarshaler .

```
setArgument booleano privado (char argChar) lanza ArgsException {
        ArgumentMarshaler m = marshalers.get (argChar);
        si (m == nulo)
            falso retorno;
        intentar {
            if (m instancia de BooleanArgumentMarshaler)
                setBooleanArg (m, argumento actual );
            else if (m instancia de StringArgumentMarshaler)
                setStringArg (m);
            else if (m instancia de IntegerArgumentMarshaler)
                setIntArg (m);

        } captura (ArgsException e) {
            válido = falso;
            errorArgumentId = argChar;
            tirar e;
        }
        devuelve verdadero;
    }
---
    private void setBooleanArg (ArgumentMarshaler m,
                                Iterador <String> currentArgument)
                                lanza ArgsException {
    intentar {
            m.set ("verdadero");
    catch (ArgsException e) {
     }
    }
```

¿No acabamos de poner ese procesamiento de excepción? Poniendo cosas para que puedas tomar sacarlos de nuevo es bastante común en la refactorización. La pequeñez de los pasos y la necesidad de Mantener las pruebas en ejecución significa que mueves mucho las cosas. Refactorizar es muy parecido a resolviendo un cubo de Rubik. Se requieren muchos pequeños pasos para lograr un gran objetivo. Cada paso habilita el siguiente.

¿Por qué pasamos ese iterador cuando setBooleanArg ciertamente no lo necesita? Porque setIntArg y setStringArg lo harán! Y porque quiero implementar las tres funciones a través de un método abstracto en ArgumentMarshaller , necesito pasarlo a setBooleanArg .

Página 265

Así que ahora setBooleanArg es inútil. Si hubiera una función establecida en ArgumentMarshaler , podría llamarlo directamente. ¡Así que es hora de hacer esa función! El primer paso es agregar el nuevo método abstracto para ArgumentMarshaler .

```
clase abstracta privada ArgumentMarshaler {
    conjunto vacío abstracto público (Iterator <String> currentArgument)
                                    lanza ArgsException;
    el conjunto vacío abstracto público (String s) lanza ArgsException;
    Objeto abstracto público get ();
}
```

Por supuesto, esto rompe todas las derivadas. Así que implementemos el nuevo método en cada uno.

```
clase privada BooleanArgumentMarshaler extiende ArgumentMarshaler {
```

```
    private boolean booleanValue = false;

    public void set (Iterator <String> currentArgument) lanza ArgsException {
      booleanValue = verdadero;
    }

    conjunto vacío público (String s) {
      booleanValue = verdadero;
    }

    public Object get () {
      return booleanValue;
    }
  }

  la clase privada StringArgumentMarshaler extiende ArgumentMarshaler {
    private String stringValue = "";

    public void set (Iterator <String> currentArgument) lanza ArgsException {
    }

    conjunto vacío público (String s) {
      stringValue = s;
    }

    public Object get () {
      return stringValue;
    }
  }

  clase privada IntegerArgumentMarshaler extiende ArgumentMarshaler {
    privado int intValue = 0;

    public void set (Iterator <String> currentArgument) lanza ArgsException {
    }

    public void set (String s) lanza ArgsException {
      intentar {
        intValue = Integer.parseInt (s);
      } catch (NumberFormatException e) {
        lanzar nueva ArgsException ();
      }
    }
```


Página 266


```
    public Object get () {
      return intValue;
    }
  }
```

¡Y ahora podemos eliminar setBooleanArg !

```
setArgument booleano privado (char argChar) lanza ArgsException {
    ArgumentMarshaler m = marshalers.get (argChar);
    si (m == nulo)
      falso retorno;
    intentar {
      if (m instancia de BooleanArgumentMarshaler)
        m.set (argumento actual);
      else if (m instancia de StringArgumentMarshaler)
        setStringArg (m);
      else if (m instancia de IntegerArgumentMarshaler)
        setIntArg (m);

    } captura (ArgsException e) {
      válido = falso;
      errorArgumentId = argChar;
      tirar e;
    }
    devuelve verdadero;
  }
```

Todas las pruebas pasan y la función set se implementa en BooleanArgumentMarshaler .

Ahora podemos hacer lo mismo con Strings e Integers .

```
setArgument booleano privado (char argChar) lanza ArgsException {
    ArgumentMarshaler m = marshalers.get (argChar);
    si (m == nulo)
        falso retorno;
    intentar {
        if (m instancia de BooleanArgumentMarshaler)
            m.set (argumento actual);
        else if (m instancia de StringArgumentMarshaler)
            m.set (argumento actual);
        else if (m instancia de IntegerArgumentMarshaler)
            m.set (argumento actual);

    } captura (ArgsException e) {
        válido = falso;
        errorArgumentId = argChar;
        tirar e;
    }
    devuelve verdadero;
}
---
la clase privada StringArgumentMarshaler extiende ArgumentMarshaler {
    private String stringValue = "";

    public void set (Iterator <String> currentArgument) lanza ArgsException {
        intentar {
            stringValue = currentArgument.next ();
        } captura (NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_STRING;
```


Página 267


```
            lanzar nueva ArgsException ();
        }
    }

    conjunto vacío público (String s) {
    }

    public Object get () {
        return stringValue;
    }
}

clase privada IntegerArgumentMarshaler extiende ArgumentMarshaler {
    privado int intValue = 0;

public void set (Iterator <String> currentArgument) lanza ArgsException {
    Parámetro de cadena = nulo;
    intentar {
        parámetro = currentArgument.next ();
        set (parámetro);
    } captura (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_INTEGER;
        lanzar nueva ArgsException ();
    } captura (ArgsException e) {
        errorParameter = parámetro;
        errorCode = ErrorCode.INVALID_INTEGER;
        tirar e;
    }
}

    public void set (String s) lanza ArgsException {
        intentar {
            intValue = Integer.parseInt (s);
        } catch (NumberFormatException e) {
            lanzar nueva ArgsException ();
        }
    }

    public Object get () {
        return intValue;
    }
}
```

Y así el *golpe de gracia* : ¡el tipo de caja se puede quitar! ¡Touche!

```
setArgument booleano privado (char argChar) lanza ArgsException {
```

```
    ArgumentMarshaler m = marshalers.get (argChar);
    si (m == nulo)
        falso retorno;
    intentar {
        m.set (argumento actual);
        devuelve verdadero;
    } captura (ArgsException e) {
        válido = falso;
        errorArgumentId = argChar;
        tirar e;
    }
}
```


Página 268

Ahora podemos deshacernos de algunas funciones crudas en IntegerArgumentMarshaler y limpiarlo un poco.

```
clase privada IntegerArgumentMarshaler extiende ArgumentMarshaler {
    privado int intValue = 0

    public void set (Iterator <String> currentArgument) lanza ArgsException {
        Parámetro de cadena = nulo;
        intentar {
            parámetro = currentArgument.next ();
            intValue = Integer.parseInt (parámetro);
        } captura (NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_INTEGER;
            lanzar nueva ArgsException ();
        } catch ( NumberFormatException e) {
            errorParameter = parámetro;
            errorCode = ErrorCode.INVALID_INTEGER;
            lanzar nueva ArgsException ();
        }
    }

    public Object get () {
        return intValue;
    }
}
```

También podemos convertir **ArgumentMarshaler** en una interfaz.

```
interfaz privada ArgumentMarshaler {
    void set (Iterator <String> currentArgument) lanza ArgsException;
    Objeto get ();
}
```

Así que ahora veamos qué tan fácil es agregar un nuevo tipo de argumento a nuestra estructura. Debería requieren muy pocos cambios, y esos cambios deben aislarse. Primero, comenzamos agregando un nuevo caso de prueba para comprobar que el argumento doble funciona correctamente.

```
public void testSimpleDoublePresent () lanza Exception {
    Args args = new Args ("x ##", new String [] {"-x", "42.3"});
    asertTrue (args.isValid ());
    asertEquals (1, args.cardinality ());
    asertTrue (args.has ('x'));
    asertEquals (42.3, args.getDouble ('x'), .001);
}
```

Ahora limpiamos el código de análisis del esquema y agregamos la detección ## para el doble tipo de argumento.

```
private void parseSchemaElement (elemento String) lanza ParseException {
    char elementId = element.charAt (0);
    Cadena elementTail = element.substring (1);
    validateSchemaElementId (elementId);
    if (elementTail. length () == 0 )
        marshalers.put (elementId, new BooleanArgumentMarshaler ());
    else if (elementTail. es igual a ("*") )
        marshalers.put (elementId, nuevo StringArgumentMarshaler ());
    else if (elementTail. es igual a ("#") )
        marshalers.put (elementId, nuevo IntegerArgumentMarshaler ());
```

Página 269


```
    else if (elementTail.equals ("##"))
        marshalers.put (elementId, nuevo DoubleArgumentMarshaler ());
    demás
        lanzar una nueva ParseException (String.format (
            "Argumento:% c tiene un formato no válido:% s.", ElementId, elementTail), 0);
}
```

A continuación, escribimos la clase DoubleArgumentMarshaler .

```
la clase privada DoubleArgumentMarshaler implementa ArgumentMarshaler {
    private double doubleValue = 0;

    public void set (Iterator <String> currentArgument) lanza ArgsException {
        Parámetro de cadena = nulo;
        intentar {
            parámetro = currentArgument.next ();
            doubleValue = Double.parseDouble (parámetro);
        } captura (NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_DOUBLE;
            lanzar nueva ArgsException ();
        } catch (NumberFormatException e) {
            errorParameter = parámetro;
            errorCode = ErrorCode.INVALID_DOUBLE;
            lanzar nueva ArgsException ();
        }
    }

    public Object get () {
        return doubleValue;
    }
}
```

Esto nos obliga a agregar un nuevo ErrorCode .

```
Private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,
    MISSING_DOUBLE, INVALID_DOUBLE }
```

Y necesitamos una función getDouble .

```
public double getDouble (char arg) {
    Args.ArgumentMarshaler am = marshalers.get (arg);
    intentar {
        return am == null? 0: (Doble) am.get ();
    } captura (Excepción e) {
        return 0.0;
    }
}
```

¡Y todas las pruebas pasan! Eso fue bastante indoloro. Así que ahora asegurémonos de que todo el error el procesamiento funciona correctamente. El siguiente caso de prueba verifica que se declare un error si un La cadena no analizable se alimenta a un argumento ## .

```
public void testInvalidDouble () lanza Exception {
    Args args = new Args ("x ##", nueva Cadena [] {"-x", "Cuarenta y dos"});
    asertFalse (args.isValid ());
    asertEquals (0, args.cardinality ());
    aseverarFalso (args.has ('x'));
    asertEquals (0, args.getInt ('x'));
```


Página 270

```
        assertEquals ("El argumento -x espera un doble pero era 'Cuarenta y dos'.",
                      args.errorMessage ());
    }
---
    public String errorMessage () lanza Exception {
        switch (errorCode) {
            caso OK:
                lanzar una nueva excepción ("TILT: no debería llegar aquí");
            caso UNEXPECTED_ARGUMENT:
                return inesperadoArgumentMessage ();
            caso MISSING_STRING:
                return String.format ("No se pudo encontrar el parámetro de cadena para -% c.",
                                      errorArgumentId);
            caso INVALID_INTEGER:
                return String.format ("Argumento -% c espera un entero pero era '% s'.",
                                      errorArgumentId, errorParameter);
            caso MISSING_INTEGER:
                return String.format ("No se pudo encontrar el parámetro entero para -% c.",
                                      errorArgumentId);
            caso INVALID_DOUBLE:
                return String.format ("Argumento -% c espera un doble pero era '% s'.",
                                      errorArgumentId, errorParameter);
            caso MISSING_DOUBLE:
                return String.format ("No se pudo encontrar el parámetro doble para -% c.",
                                      errorArgumentId);
        }
        regreso "";
    }
```

Y pasan las pruebas. La siguiente prueba asegura que detectemos correctamente un argumento doble faltante .

```
    public void testMissingDouble () lanza Exception {
        Args args = new Args ("x ##", nueva cadena [] {"- x"});
        asertFalse (args.isValid ());
        asertEquals (0, args.cardinality ());
        aseverarFalso (args.has ('x'));
        asertEquals (0.0, args.getDouble ('x'), 0.01);
        assertEquals ("No se pudo encontrar el parámetro doble para -x.",
                      args.errorMessage ());
    }
```

Esto pasa como se esperaba. Lo escribimos simplemente para completarlo.

El código de excepción es bastante feo y realmente no pertenece a la clase Args . Somos también arrojando ParseException , que realmente no nos pertenece. Así que fusionemos todos los excepciones en una sola clase ArgsException y muévala a su propio módulo.

```
La clase pública ArgsException extiende la excepción {
    carácter privado errorArgumentId = '\ 0';
    private String errorParameter = "TILT";
    errorCode privado errorCode = ErrorCode.OK;

    public ArgsException () {}

    public ArgsException (mensaje de cadena) {super (mensaje);}

    public enum ErrorCode {
        OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,
        MISSING_DOUBLE, INVALID_DOUBLE}
}
---
```


Página 271


```
public class Args {
    ...
    carácter privado errorArgumentId = '\ 0';
    private String errorParameter = "TILT";
    private ArgsException .ErrorCode errorCode = ArgsException .ErrorCode.OK;
    Private List <String> argsList;
```

```
public Args (esquema de cadena, String [] args) lanza ArgsException {
    this.schema = esquema;
    argsList = Arrays.asList (args);
    válido = analizar ();
}

parse () booleano privado lanza ArgsException {
    if (schema.length () == 0 && argsList.size () == 0)
        devuelve verdadero;
    parseSchema ();
    intentar {
        parseArguments ();
    } captura ( ArgsException e) {
    }
    retorno válido;
}

parseSchema () booleano privado lanza ArgsException {
    ...
}

private void parseSchemaElement (elemento String) lanza ArgsException {
    ...
    demás
        lanzar una nueva ArgsException (
            String.format ("Argumento:% c tiene un formato no válido:% s.",
                                    elementId, elementTail));
}

private void validateSchemaElementId (char elementId) lanza ArgsException {
    if (! Character.isLetter (elementId)) {
        lanzar una nueva ArgsException (
            "Carácter incorrecto:" + elementId + "en formato Args:" + esquema);
    }
}

...

parseElement vacío privado (char argChar) lanza ArgsException {
    si (setArgument (argChar))
        argsFound.add (argChar);
    demás {
        inesperadosArgumentos.add (argChar);
        errorCode = ArgsException .ErrorCode.UNEXPECTED_ARGUMENT;
        válido = falso;
    }
}

...
```


Página 272


```
la clase privada StringArgumentMarshaler implementa ArgumentMarshaler {
    private String stringValue = "";

    public void set (Iterator <String> currentArgument) lanza ArgsException {
        intentar {
            stringValue = currentArgument.next ();
        } captura (NoSuchElementException e) {
            errorCode = ArgsException .ErrorCode.MISSING_STRING;
            lanzar nueva ArgsException ();
        }
    }

    public Object get () {
        return stringValue;
    }
}

la clase privada IntegerArgumentMarshaler implementa ArgumentMarshaler {
    privado int intValue = 0;

    public void set (Iterator <String> currentArgument) lanza ArgsException {
        Parámetro de cadena = nulo;
        intentar {
```

```
            parámetro = currentArgument.next ();
            intValue = Integer.parseInt (parámetro);
        } captura (NoSuchElementException e) {
            errorCode = ArgsException.ErrorCode.MISSING_INTEGER;
            lanzar nueva ArgsException ();
        } catch (NumberFormatException e) {
            errorParameter = parámetro;
            errorCode = ArgsException .ErrorCode.INVALID_INTEGER;
            lanzar nueva ArgsException ();
        }
    }

    public Object get () {
        return intValue;
    }
}

la clase privada DoubleArgumentMarshaler implementa ArgumentMarshaler {
    private double doubleValue = 0;

    public void set (Iterator <String> currentArgument) lanza ArgsException {
        Parámetro de cadena = nulo;
        intentar {
            parámetro = currentArgument.next ();
            doubleValue = Double.parseDouble (parámetro);
        } captura (NoSuchElementException e) {
            errorCode = ArgsException .ErrorCode.MISSING_DOUBLE;
            lanzar nueva ArgsException ();
        } catch (NumberFormatException e) {
            errorParameter = parámetro;
            errorCode = ArgsException .ErrorCode.INVALID_DOUBLE;
            lanzar nueva ArgsException ();
        }
    }
```


Página 273


```
    public Object get () {
        return doubleValue;
    }
  }
}
```

Esto es bonito. Ahora, la única excepción lanzada por Args es ArgsException . Moviente ArgsException en su propio módulo significa que podemos mover muchos de los código de soporte de error en ese módulo y fuera del módulo Args . Aporta un efecto natural y lugar obvio para poner todo ese código y realmente nos ayudará a limpiar el módulo Args en marcha hacia adelante.

Así que ahora hemos separado completamente el código de excepción y error de Args. módulo. (Consulte el Listado 14-13 hasta el Listado 14-16). Esto se logró mediante una serie de unos 30 pequeños pasos, manteniendo las pruebas pasando entre cada paso.

Listado 14-13

ArgsTest.java

```
paquete com.objectmentor.utilities.args;

import junit.framework.TestCase;

La clase pública ArgsTest extiende TestCase {
  public void testCreateWithNoSchemaOrArguments () lanza Exception {
    Args args = new Args ("", nueva cadena [0]);
    asertEquals (0, args.cardinality ());
  }

  public void testWithNoSchemaButWithOneArgument () lanza Exception {
    intentar {
      new Args ("", nueva cadena [] {"- x"});
      fallar();
    } captura (ArgsException e) {
      assertEquals (ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                    e.getErrorCode ());
      asertEquals ('x', e.getErrorArgumentId ());
```

```
    }
}

public void testWithNoSchemaButWithMultipleArguments () lanza Exception {
    intentar {
        new Args ("", nueva cadena [] {"- x", "-y"});
        fallar();
    } captura (ArgsException e) {
        assertEquals (ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                      e.getErrorCode ());
        asertEquals ('x', e.getErrorArgumentId ());
    }

}

public void testNonLetterSchema () lanza Exception {
    intentar {
        new Args ("*", nueva cadena [] {});
        fail ("El constructor de Args debería haber lanzado una excepción");
    } captura (ArgsException e) {
```


Página 274

Listado 14-13 (continuación)

ArgsTest.java

```
        assertEquals (ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
                      e.getErrorCode ());
        asertEquals ('*', e.getErrorArgumentId ());
    }
}

public void testInvalidArgumentFormat () lanza Exception {
    intentar {
        new Args ("f~", nueva cadena [] {});
        fail ("El constructor de Args debería tener una excepción de lanzamiento");
    } captura (ArgsException e) {
        assertEquals (ArgsException.ErrorCode.INVALID_FORMAT, e.getErrorCode ());
        asertEquals ('f', e.getErrorArgumentId ());
    }
}

public void testSimpleBooleanPresent () lanza Exception {
    Args args = new Args ("x", nueva cadena [] {"- x"});
    asertEquals (1, args.cardinality ());
    asertEquals (verdadero, args.getBoolean ('x'));
}

public void testSimpleStringPresent () lanza Exception {
    Args args = new Args ("x *", new String [] {"- x", "param"});
    asertEquals (1, args.cardinality ());
    asertTrue (args.has ('x'));
    assertEquals ("param", args.getString ('x'));
}

public void testMissingStringArgument () lanza Exception {
    intentar {
        new Args ("x *", new String [] {"- x"});
        fallar();
    } captura (ArgsException e) {
        assertEquals (ArgsException.ErrorCode.MISSING_STRING, e.getErrorCode ());
        asertEquals ('x', e.getErrorArgumentId ());
    }
}

public void testSpacesInFormat () lanza Exception {
    Args args = new Args ("x, y", nueva cadena [] {"- xy"});
    asertEquals (2, args.cardinality ());
    asertTrue (args.has ('x'));
    asertTrue (args.has ('y'));
}

public void testSimpleIntPresent () lanza Exception {
    Args args = new Args ("x #", nueva cadena [] {"- x", "42"});
    asertEquals (1, args.cardinality ());
    asertTrue (args.has ('x'));
    asertEquals (42, args.getInt ('x'));
```

```
    }
    public void testInvalidInteger () lanza Exception {
        intentar {
            new Args ("x #", new String [] {"- x", "Cuarenta y dos"});
```


Página 275

Listado 14-13 (continuación)

ArgsTest.java

```
            fallar();
        } captura (ArgsException e) {
            assertEquals (ArgsException.ErrorCode.INVALID_INTEGER, e.getErrorCode ());
            asertEquals ('x', e.getErrorArgumentId ());
            asertEquals ("Cuarenta y dos", e.getErrorParameter ());
        }

    }

    public void testMissingInteger () lanza Exception {
        intentar {
            new Args ("x #", new String [] {"- x"});
            fallar();
        } captura (ArgsException e) {
            assertEquals (ArgsException.ErrorCode.MISSING_INTEGER, e.getErrorCode ());
            asertEquals ('x', e.getErrorArgumentId ());
        }
    }

    public void testSimpleDoublePresent () lanza Exception {
        Args args = new Args ("x ##", new String [] {"- x", "42.3"});
        asertEquals (1, args.cardinality ());
        asertTrue (args.has ('x'));
        asertEquals (42.3, args.getDouble ('x'), .001);
    }

    public void testInvalidDouble () lanza Exception {
        intentar {
            new Args ("x ##", new String [] {"- x", "Cuarenta y dos"});
            fallar();
        } captura (ArgsException e) {
            assertEquals (ArgsException.ErrorCode.INVALID_DOUBLE, e.getErrorCode ());
            asertEquals ('x', e.getErrorArgumentId ());
            asertEquals ("Cuarenta y dos", e.getErrorParameter ());
        }
    }

    public void testMissingDouble () lanza Exception {
        intentar {
            new Args ("x ##", new String [] {"- x"});
            fallar();
        } captura (ArgsException e) {
            assertEquals (ArgsException.ErrorCode.MISSING_DOUBLE, e.getErrorCode ());
            asertEquals ('x', e.getErrorArgumentId ());
        }
    }
}
```

Listado 14-14

ArgsExceptionTest.java

```
La clase pública ArgsExceptionTest extiende TestCase {
    public void testUnexpectedMessage () lanza Exception {
        ArgsException e =
```

Página 276

Listado 14-14 (continuación)

ArgsExceptionTest.java

```
            nueva ArgsException (ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                                    'x', nulo);
        assertEquals ("Argumento -x inesperado.", e.errorMessage ());
    }

    public void testMissingStringMessage () lanza Exception {
        ArgsException e = nueva ArgsException (ArgsException.ErrorCode.MISSING_STRING,
                                                'x', nulo);
        assertEquals ("No se pudo encontrar el parámetro de cadena para -x.", e.errorMessage ());
    }

    public void testInvalidIntegerMessage () lanza Exception {
        ArgsException e =
            nueva ArgsException (ArgsException.ErrorCode.INVALID_INTEGER,
                                    'x', "Cuarenta y dos");
        assertEquals ("El argumento -x espera un número entero pero era 'Cuarenta y dos'.",
                      e.errorMessage ());
    }

    public void testMissingIntegerMessage () lanza Exception {
        ArgsException e =
            new ArgsException (ArgsException.ErrorCode.MISSING_INTEGER, 'x', nulo);
        assertEquals ("No se pudo encontrar el parámetro entero para -x.", e.errorMessage ());
    }

    public void testInvalidDoubleMessage () lanza Exception {
        ArgsException e = new ArgsException (ArgsException.ErrorCode.INVALID_DOUBLE,
                                             'x', "Cuarenta y dos");
        assertEquals ("El argumento -x espera un doble pero era 'Cuarenta y dos'.",
                      e.errorMessage ());
    }

    public void testMissingDoubleMessage () lanza Exception {
        ArgsException e = new ArgsException (ArgsException.ErrorCode.MISSING_DOUBLE,
                                             'x', nulo);
        assertEquals ("No se pudo encontrar el parámetro doble para -x.", e.errorMessage ());
    }
}
```

Listado 14-15

ArgsException.java

```
La clase pública ArgsException extiende la excepción {
    carácter privado errorArgumentId = '\ 0';
    private String errorParameter = "TILT";
    errorCode privado errorCode = ErrorCode.OK;

    public ArgsException () {}

    public ArgsException (mensaje de cadena) {super (mensaje);}

    public ArgsException (ErrorCode errorCode) {
        this.errorCode = errorCode;
    }
```


Página 277

Listado 14-15 (continuación)

ArgsException.java

```
public ArgsException (ErrorCode errorCode, String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
}

public ArgsException (ErrorCode errorCode, char errorArgumentId,
                      String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
    this.errorArgumentId = errorArgumentId;
}

public char getErrorArgumentId () {
    return errorArgumentId;
}

public void setErrorArgumentId (char errorArgumentId) {
    this.errorArgumentId = errorArgumentId;
}

public String getErrorParameter () {
    return errorParameter;
}

public void setErrorParameter (String errorParameter) {
    this.errorParameter = errorParameter;
}

public ErrorCode getErrorCode () {
    return errorCode;
}

public void setErrorCode (ErrorCode errorCode) {
    this.errorCode = errorCode;
}

public String errorMessage () lanza Exception {
    switch (errorCode) {
        caso OK:
            lanzar una nueva excepción ("TILT: no debería llegar aquí");
        caso UNEXPECTED_ARGUMENT:
            return String.format ("Argumento -% c inesperado.", errorArgumentId);
        caso MISSING_STRING:
            return String.format ("No se pudo encontrar el parámetro de cadena para -% c.",
                                  errorArgumentId);
        caso INVALID_INTEGER:
            return String.format ("Argumento -% c espera un entero pero era '% s'.",
                                  errorArgumentId, errorParameter);
        caso MISSING_INTEGER:
            return String.format ("No se pudo encontrar el parámetro entero para -% c.",
                                  errorArgumentId);
        caso INVALID_DOUBLE:
            return String.format ("Argumento -% c espera un doble pero era '% s'.",
                                  errorArgumentId, errorParameter);
```


Página 278

Listado 14-15 (continuación)

ArgsException.java

```
        caso MISSING_DOUBLE:
            return String.format ("No se pudo encontrar el parámetro doble para -% c.",
                                  errorArgumentId);
    }
    regreso "";
}

public enum ErrorCode {
    OK, INVALID_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
    MISSING_STRING,
    MISSING_INTEGER, INVALID_INTEGER,
    MISSING_DOUBLE, INVALID_DOUBLE}
```

```
}
```

Listado 14-16

Args.java

```
public class Args {
    esquema de cadena privada;
    mapa privado <Personaje, ArgumentoMarshaler> marshalers =
        new HashMap <Carácter, ArgumentMarshaler> ();
    private Set <Character> argsFound = new HashSet <Character> ();
    private Iterator <String> currentArgument;
    Private List <String> argsList;

    public Args (esquema de cadena, String [] args) lanza ArgsException {
        this.schema = esquema;
        argsList = Arrays.asList (args);
        analizar gramaticalmente();
    }

    private void parse () lanza ArgsException {
        parseSchema ();
        parseArguments ();
    }

    parseSchema () booleano privado lanza ArgsException {
        para (Elemento de cadena: esquema.split (",")) {
            if (element.length ()> 0) {
                parseSchemaElement (element.trim ());
            }
        }
        devuelve verdadero;
    }

    private void parseSchemaElement (elemento String) lanza ArgsException {
        char elementId = element.charAt (0);
        Cadena elementTail = element.substring (1);
        validateSchemaElementId (elementId);
        if (elementTail.length () == 0)
            marshalers.put (elementId, new BooleanArgumentMarshaler ());
        else if (elementTail.equals ("*"))
            marshalers.put (elementId, nuevo StringArgumentMarshaler ());
```


Página 279

Listado 14-16 (continuación)

Args.java

```
        else if (elementTail.equals ("#"))
            marshalers.put (elementId, nuevo IntegerArgumentMarshaler ());
        else if (elementTail.equals ("##"))
            marshalers.put (elementId, nuevo DoubleArgumentMarshaler ());
        demás
            lanzar una nueva ArgsException (ArgsException.ErrorCode.INVALID_FORMAT,
                                            elementId, elementTail);
    }

    private void validateSchemaElementId (char elementId) lanza ArgsException {
        if (! Character.isLetter (elementId)) {
            lanzar una nueva ArgsException (ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
                                            elementId, nulo);
        }
    }

    private void parseArguments () lanza ArgsException {
        for (currentArgument = argsList.iterator (); currentArgument.hasNext ();) {
            String arg = currentArgument.next ();
            parseArgument (arg);
        }
    }

    private void parseArgument (String arg) lanza ArgsException {
        if (arg.startsWith ("-"))
            parseElements (arg);
    }
```

```
private void parseElements (String arg) lanza ArgsException {
    para (int i = 1; i <arg.length (); i ++)
        parseElement (arg.charAt (i));
}

parseElement vacío privado (char argChar) lanza ArgsException {
    si (setArgument (argChar))
        argsFound.add (argChar);
    demás {
        lanzar una nueva ArgsException (ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                                        argChar, nulo);
    }
}

setArgument booleano privado (char argChar) lanza ArgsException {
    ArgumentMarshaler m = marshalers.get (argChar);
    si (m == nulo)
        falso retorno;
    intentar {
        m.set (argumento actual);
        devuelve verdadero;
    } captura (ArgsException e) {
        e.setErrorArgumentId (argChar);
        tirar e;
    }
}
```


Página 280

Listado 14-16 (continuación)

Args.java

```
public int cardinality () {
    return argsFound.size ();
}

uso de cadena pública () {
    if (schema.length ()> 0)
        return "- [" + esquema + "]";
    demás
        regreso "";
}

public boolean getBoolean (char arg) {
    ArgumentMarshaler am = marshalers.get (arg);
    booleano b = falso;
    intentar {
        b = am! = null && (booleano) am.get ();
    } captura (ClassCastException e) {
        b = falso;
    }
    volver b;
}

public String getString (char arg) {
    ArgumentMarshaler am = marshalers.get (arg);
    intentar {
        return am == null? "": (Cadena) am.get ();
    } captura (ClassCastException e) {
        regreso "";
    }
}

public int getInt (char arg) {
    ArgumentMarshaler am = marshalers.get (arg);
    intentar {
        return am == null? 0: (Entero) am.get ();
    } captura (Excepción e) {
        return 0;
    }
}


public double getDouble (char arg) {
    ArgumentMarshaler am = marshalers.get (arg);
    intentar {
```

```
            return am == null? 0: (Doble) am.get ();
        } captura (Excepción e) {
            return 0.0;
        }
    }

    public boolean tiene (char arg) {
        return argsFound.contains (arg);
    }
}
```


Página 281

La mayoría de los cambios en la clase Args fueron eliminaciones. Se acaba de obtener una gran cantidad de código se movió fuera de Args y se puso en ArgsException . Lindo. También movimos todos los ArgumentMarshaller s en sus propios archivos. ¡Mejor!

Gran parte del buen diseño de software se trata simplemente de particionar: crear lugares para poner diferentes tipos de código. Esta separación de preocupaciones hace que el código sea mucho más simple de entender y mantener.

De especial interés es la errorMessage método de ArgsException . Claramente fue una violación del SRP para poner el formato del mensaje de error en Args . Args debe ser sobre el procesamiento de argumentos, no sobre el formato de los mensajes de error. Sin embargo, lo hace ¿Realmente tiene sentido poner el código de formato del mensaje de error en ArgsException ?

Francamente, es un compromiso. Usuarios a los que no les gustan los mensajes de error proporcionados por ArgsException tendrá que escribir el suyo propio. Pero la conveniencia de tener un error enlatado los mensajes ya preparados para ti no son insignificantes.

A estas alturas debería estar claro que estamos a una distancia sorprendente de la solución final que apareció al comienzo de este capítulo. Te dejo las transformaciones finales como ejercicio.

Conclusión

No es suficiente que el código funcione. El código que funciona a menudo está muy mal. Programadores quienes se satisfacen con un código que simplemente funciona se están comportando de manera poco profesional. Ellos puede temer que no tengan tiempo para mejorar la estructura y el diseño de su código, pero yo discrepar. Nada tiene un efecto degradante más profundo y a largo plazo sobre un desarrollo proyecto de ment que el código incorrecto. Los malos horarios se pueden rehacer, los malos requisitos se pueden redefinir. multado. Se pueden reparar las malas dinámicas de equipo. Pero el código incorrecto se pudre y fermenta, convirtiéndose en un peso inexorable que arrastra al equipo hacia abajo. Una y otra vez he visto a equipos rechinar a gatear porque, en su prisa, crearon un maligno pantano de código que para siempre a partir de entonces dominó su destino.

Por supuesto, el código incorrecto se puede limpiar. Pero es muy caro. A medida que el código se pudre, el modules se insinúan entre sí, creando muchas dependencias ocultas y enredadas cies. Encontrar y romper viejas dependencias es una tarea larga y ardua. Por otro lado, mantener el código limpio es relativamente fácil. Si hiciste un lío en un módulo por la mañana, es fácil de limpiar por la tarde. Mejor aún, si hiciste un desastre hace cinco minutos, es muy fácil de limpiar ahora mismo.

Entonces, la solución es mantener continuamente su código tan limpio y simple como sea posible. Nunca dejes que comience la podredumbre.

Página 282

15

Internos de JUnit

JUnit es uno de los frameworks Java más famosos. A medida que avanzan los marcos, es simple en concepción, precisa en la definición y elegante en la implementación. Pero que hace el codigo ¿parece? En este capítulo analizaremos un ejemplo extraído del marco JUnit.

Página 283

El marco JUnit

JUnit ha tenido muchos autores, pero comenzó con Kent Beck y Eric Gamma juntos en un avión a Atlanta. Kent quería aprender Java, y Eric quería aprender acerca de Small-hablar sobre el marco de pruebas. "¿Qué podría ser más natural para un par de geeks en trimestres que sacar nuestras computadoras portátiles y comenzar a codificar? " [1] Después de tres horas de gran altitud trabajo, habían escrito los conceptos básicos de JUnit.

El módulo que veremos es el código inteligente que ayuda a identificar la comparación de cadenas. errores de hijo. Este módulo se llama `ComparisonCompactor` . Dadas dos cadenas que difieren, como `ABCDE` y `ABXDE`, expondrá la diferencia generando una cadena como `<... B [X] D ...>` .

Podría explicarlo más, pero los casos de prueba hacen un mejor trabajo. Así que eche un vistazo al Listado 15-1. y comprenderá en profundidad los requisitos de este módulo. Mientras estás en eso, criticar la estructura de las pruebas. ¿Podrían ser más simples o más obvios?

Listado 15-1

ComparisonCompactorTest.java

```
paquete junit.tests.framework;

import junit.framework.ComparisonCompactor;
import junit.framework.TestCase;

La clase pública ComparisonCompactorTest extiende TestCase {

    public void testMessage () {
        Error de cadena = nuevo ComparisonCompactor (0, "b", "c"). Compact ("a");
        asertTrue ("a esperado: <[b]> pero era: <[c]>". es igual a (falla));
    }

    public void testStartSame () {
        Error de cadena = nuevo ComparisonCompactor (1, "ba", "bc"). Compact (nulo);
        assertEquals ("esperado: <b [a]> pero fue: <b [c]>", falla);
    }

    public void testEndSame () {
        Error de cadena = nuevo ComparisonCompactor (1, "ab", "cb"). Compact (nulo);
        assertEquals ("esperado: <[a] b> pero fue: <[c] b>", falla);
    }

    public void testSame () {
        Error de cadena = nuevo ComparisonCompactor (1, "ab", "ab"). Compact (nulo);
        assertEquals ("esperado: <ab> pero fue: <ab>", error);
    }

    public void testNoContextStartAndEndSame () {
        Error de cadena = nuevo ComparisonCompactor (0, "abc", "adc"). Compact (nulo);
        assertEquals ("esperado: <... [b] ...> pero fue: <... [d] ...>", falla);
    }
```

1. *Guía de bolsillo de JUnit* , Kent Beck, O'Reilly, 2004, p. 43.

Página 284

Listado 15-1 (continuación)

ComparisonCompactorTest.java

```
    public void testStartAndEndContext () {
        Error de cadena = nuevo ComparisonCompactor (1, "abc", "adc"). Compact (nulo);
        assertEquals ("esperado: <a [b] c> pero fue: <a [d] c>", falla);
    }
```

```
    public void testStartAndEndContextWithEllipses () {
        Fallo de cadena =
            nuevo ComparisonCompactor (1, "abcde", "abfde"). compact (nulo);
        assertEquals ("esperado: <... b [c] d ...> pero fue: <... b [f] d ...>", falla);
    }

    public void testComparisonErrorStartSameComplete () {
        Error de cadena = nuevo ComparisonCompactor (2, "ab", "abc"). Compact (nulo);
        assertEquals ("esperado: <ab []> pero fue: <ab [c]>", falla);
    }

    public void testComparisonErrorEndSameComplete () {
        Error de cadena = nuevo ComparisonCompactor (0, "bc", "abc"). Compact (nulo);
        assertEquals ("esperado: <[] ...> pero fue: <[a] ...>", falla);
    }

    public void testComparisonErrorEndSameCompleteContext () {
        Error de cadena = nuevo ComparisonCompactor (2, "bc", "abc"). Compact (nulo);
        assertEquals ("esperado: <[] bc> pero fue: <[a] bc>", falla);
    }

    public void testComparisonErrorOverlapingMatches () {
        Error de cadena = nuevo ComparisonCompactor (0, "abc", "abbc"). Compact (nulo);
        assertEquals ("esperado: <... [] ...> pero fue: <... [b] ...>", falla);
    }

    public void testComparisonErrorOverlapingMatchesContext () {
        Error de cadena = nuevo ComparisonCompactor (2, "abc", "abbc"). Compact (nulo);
        assertEquals ("esperado: <ab [] c> pero fue: <ab [b] c>", falla);
    }

    public void testComparisonErrorOverlapingMatches2 () {
        Error de cadena = nuevo ComparisonCompactor (0, "abcdde",
"abcde"). compact (nulo);
        assertEquals ("esperado: <... [d] ...> pero fue: <... [] ...>", falla);
    }

    public void testComparisonErrorOverlapingMatches2Context () {
        Fallo de cadena =
            nuevo ComparisonCompactor (2, "abcdde", "abcde"). compact (nulo);
        assertEquals ("esperado: <... cd [d] e> pero fue: <... cd [] e>", falla);
    }

    public void testComparisonErrorWithActualNull () {
        Error de cadena = nuevo ComparisonCompactor (0, "a", nulo) .compact (nulo);
        assertEquals ("esperado: <a> pero fue: <nulo>", error);
    }

    public void testComparisonErrorWithActualNullContext () {
        Error de cadena = nuevo ComparisonCompactor (2, "a", nulo) .compact (nulo);
```


Página 285

Listado 15-1 (continuación)

ComparisonCompactorTest.java

```
        assertEquals ("esperado: <a> pero fue: <nulo>", error);
    }

    public void testComparisonErrorWithExpectedNull () {
        Error de cadena = nuevo ComparisonCompactor (0, nulo, "a"). Compacto (nulo);
        assertEquals ("esperado: <nulo> pero fue: <a>", falla);
    }

    public void testComparisonErrorWithExpectedNullContext () {
        Error de cadena = nuevo ComparisonCompactor (2, nulo, "a"). Compacto (nulo);
        assertEquals ("esperado: <nulo> pero fue: <a>", falla);
    }

    public void testBug609972 () {
        Error de cadena = nuevo ComparisonCompactor (10, "S & P500", "0"). Compact (nulo);
        assertEquals ("esperado: <[S & P50] 0> pero fue: <[] 0>", falla);
    }
}
```

Ejecuté un análisis de cobertura de código en ComparisonCompactor usando estas pruebas. El código

está 100 por ciento cubierto. Cada línea de código, cada instrucción if y bucle for , es ejecutada por los exámenes. Esto me da un alto grado de confianza en que el código funciona y un alto grado de respeto a la artesanía de los autores.

El código de ComparisonCompactor está en el Listado 15-2. Tómate un momento para mirar esto código. Creo que encontrará que está muy bien dividido, es razonablemente expresivo y simple en estructura. Una vez que hayas terminado, recogeremos las liendres juntos.

Listado 15-2

ComparisonCompactor.java (Original)

```
paquete junit.framework;

ComparisonCompactor de clase pública {

    Cadena final estática privada ELLIPSIS = "...";
    Cadena final estática privada DELTA_END = "]";
    Cadena final estática privada DELTA_START = "[";

    private int fContextLength;
    Private String fExpected;
    private String fActual;
    private int fPrefix;
    private int fSuffix;

    ComparisonCompactor público (int contextLength,
                                          Cadena esperada,
                                            String actual) {
        fContextLength = contextLength;
        fEsperado = esperado;
        fActual = actual;
    }
```


Página 286

Listado 15-2 (continuación)

ComparisonCompactor.java (Original)

```
    public String compact (mensaje de cadena) {
        if (fEsperado == nulo || fActual == nulo || sonStringsEqual ())
            return Assert.format (mensaje, fExpected, fActual);

        findCommonPrefix ();
        findCommonSuffix ();
        Cadena esperada = compactString (fExpected);
        Cadena actual = compactString (fActual);
        return Assert.format (mensaje, esperado, actual);
    }

    private String compactString (String source) {
        Resultado de cadena = DELTA_START +
                                    source.substring (fPrefix, source.length () -
                                        fSufijo + 1) + DELTA_END;
        si (fPrefijo> 0)
            resultado = computeCommonPrefix () + resultado;
        si (fSufijo> 0)
            resultado = resultado + computeCommonSuffix ();
        devolver resultado;
    }

    private void findCommonPrefix () {
        fPrefix = 0;
        int end = Math.min (fExpected.length (), fActual.length ());
        para (; fPrefix <end; fPrefix ++) {
            if (fExpected.charAt (fPrefix)! = fActual.charAt (fPrefix))
                rotura;
        }
    }

    private void findCommonSuffix () {
        int Sufijo esperado = f Longitud esperada () - 1;
        int actualSuffix = fActual.length () - 1;
        por (;
                actualSuffix> = fPrefix && esperadoSuffix> = fPrefix;
                  actualSuffix-- ,pectedSuffix--) {
```

```
        if (fExpected.charAt (esperadoSuffix)! = fActual.charAt (actualSuffix))
            rotura;
    }
    fSuffix = fExpected.length () - esperadoSuffix;
}

private String computeCommonPrefix () {
    return (fPrefix> fContextLength? ELLIPSIS: "") +
                fExpected.substring (Math.max (0, fPrefix - fContextLength),
                                                    fPrefix);
}

private String computeCommonSuffix () {
    int end = Math.min (fExpected.length () - fSuffix + 1 + fContextLength,
                                        f Longitud.esperada ());
    return fExpected.substring (fExpected.length () - fSuffix + 1, end) +
            (f Longitud.esperada () - fSufijo + 1 <f Longitud.esperada () -
              fContextLength? ELIPSIS: "");
}
```


Página 287

Listado 15-2 (continuación)

ComparisonCompactor.java (Original)

```
    private boolean areStringsEqual () {
        return fExpected.equals (fActual);
    }
}
```

Es posible que tenga algunas quejas sobre este módulo. Hay algunas expresiones largas y algunos +1 extraños y así sucesivamente. Pero en general, este módulo es bastante bueno. Después de todo, es podría haberse parecido al Listado 15-3.

Listado 15-3

ComparisonCompator.java (predeterminado)

```
paquete junit.framework;

ComparisonCompactor de clase pública {
    int ctxt privado;
    Private String s1;
    Private String s2;
    pfx privado int;
    privado int sfx;

    ComparisonCompactor público (int ctxt, String s1, String s2) {
        this.ctxt = ctxt;
        this.s1 = s1;
        this.s2 = s2;
    }

    public String compact (String msg) {
        si (s1 == nulo || s2 == nulo || s1.equals (s2))
            return Assert.format (msg, s1, s2);

        pfx = 0;
        para (; pfx <Math.min (s1.length (), s2.length ()); pfx ++) {
            si (s1.charAt (pfx)! = s2.charAt (pfx))
                rotura;
        }
        int sfx1 = s1.length () - 1;
        int sfx2 = s2.length () - 1;
        para (; sfx2> = pfx && sfx1> = pfx; sfx2--, sfx1--) {
            si (s1.charAt (sfx1)! = s2.charAt (sfx2))
                rotura;
        }
        sfx = s1.length () - sfx1;
        Cadena cmp1 = compactString (s1);
        Cadena cmp2 = compactString (s2);
        return Assert.format (msg, cmp1, cmp2);
    }

    Private String compactString (String s) {
        Resultado de cadena =
            "[" + subcadena (pfx, s.length () - sfx + 1) + "]";
```

```
si (pfx> 0)
    resultado = (pfx> ctxt? "...": "") +
        s1.substring (Math.max (0, pfx - ctxt), pfx) + resultado;
```


Página 288

Listado 15-3 (continuación)

ComparisonCompator.java (predeterminado)

```
    si (sfx> 0) {
        int end = Math.min (s1.length () - sfx + 1 + ctxt, s1.length ());
        resultado = resultado + (s1.substring (s1.length () - sfx + 1, end) +
            (s1.length () - sfx + 1 <s1.length () - ctxt? "...": ""));
    }
    devolver resultado;
}

}
```

Aunque los autores dejaron este módulo en muy buena forma, la *Regla Boy Scout* [2] dice nosotros debemos dejarlo más limpio de lo que lo encontramos. Entonces, ¿cómo podemos mejorar el original? código en el Listado 15-2?

Lo primero que no me importa es el prefijo `f` para las variables miembro [N6]. De hoy Los entornos hacen que este tipo de codificación de osciloscopio sea redundante. Así que eliminemos todas las `f` .

```
private int contextLength;
Se espera una cadena privada;
Private String actual;
prefijo int privado;
sufijo int privado;
```

A continuación, tenemos un condicional no encapsulado al comienzo de la función `compacta` [G28].

```
public String compact (mensaje de cadena) {
    si (esperado == nulo || actual == nulo || areStringsEqual ())
        return Assert.format (mensaje, esperado, actual);

    findCommonPrefix ();
    findCommonSuffix ();
    Cadena esperada = compactString (this.expected);
    String actual = compactString (this.actual);
    return Assert.format (mensaje, esperado, actual);
}
```

Este condicional debe encapsularse para dejar clara nuestra intención. Así que extraigamos un método Eso lo explica.

```
public String compact (mensaje de cadena) {
    si ( shouldNotCompact () )
        return Assert.format (mensaje, esperado, actual);

    findCommonPrefix ();
    findCommonSuffix ();
    Cadena esperada = compactString (this.expected);
    String actual = compactString (this.actual);
    return Assert.format (mensaje, esperado, actual);
}
```

2. Vea “La regla de los Boy Scouts” en la página 14.

Página 289


```
private boolean shouldNotCompact () {
    retorno esperado == nulo || actual == nulo || areStringsEqual ();
}
```

No me importa mucho la notación this.expected y this.actual en la función compacta ción. Esto sucedió cuando cambiamos el nombre de fExpected a esperado . Por qué hay variables en esta función que tienen los mismos nombres que las variables miembro? Ellos no representar algo más [N4]? Debemos hacer que los nombres sean inequívocos.

```
String compactExpected = compactString ( esperado );
String compactActual = compactString ( actual );
```

Los aspectos negativos son un poco más difíciles de comprender que los positivos [G29]. Así que cambiemos eso si declaración en su cabeza e invertir el sentido del condicional.

```
public String compact (mensaje de cadena) {
    if ( canBeCompacted ()) {
        findCommonPrefix ();
        findCommonSuffix ();
        String compactExpected = compactString (esperado);
        String compactActual = compactString (actual);
        return Assert.format (mensaje, compactExpected, compactActual);
    } demás {
        return Assert.format (mensaje, esperado, actual);
    }
}

private boolean canBeCompacted () {
    retorno esperado ! = nulo && actual ! = nulo &&! areStringsEqual ();
}
```

El nombre de la función es extraño [N7]. Aunque compacta las cuerdas, en realidad, es posible que no compacte las cadenas si canBeCompacted devuelve falso . Así que nombrar esto function compact oculta el efecto secundario de la comprobación de errores. Observe también que la función devuelve un mensaje formateado, no solo las cadenas compactadas. Entonces el nombre de la función debería ser formatCompactedComparison . Eso hace que se lea mucho mejor cuando se toma con el argumento de la función:

```
public String formatCompactedComparison (mensaje de cadena) {
```

El cuerpo de la sentencia if es donde la verdadera compactación de lo esperado y lo real las cadenas están hechas. Deberíamos extraer eso como un método llamado compactExpectedAndActual . Cómo- siempre, queremos que la función formatCompactedComparison haga todo el formateo. La compact ... la función no debería hacer más que compactar [G30]. Así que dividámoslo de la siguiente manera:

```
...
    private String compactExpected;
    private String compactActual;

...

    public String formatCompactedComparison (mensaje de cadena) {
        if (canBeCompacted ()) {
            compactExpectedAndActual ();
            return Assert.format (mensaje, compactExpected, compactActual);
        } demás {
```


Página 290


```
return Assert.format (mensaje, esperado, actual);
```

```
    }
}

private void compactExpectedAndActual () {
    findCommonPrefix ();
    findCommonSuffix ();
    compactExpected = compactString (esperado);
    compactActual = compactString (actual);
}
```

Tenga en cuenta que esto nos obligó a promover compactExpected y compactActual a miembro variables. No me gusta la forma en que las dos últimas líneas de la nueva función devuelven variables, pero los dos primeros no lo hacen. No utilizan convenciones coherentes [G11]. Entonces deberíamos cambie findCommonPrefix y findCommonSuffix para devolver los valores de prefijo y sufijo.

```
private void compactExpectedAndActual () {
    prefixIndex = findCommonPrefix ();
    sufijoIndex = findCommonSuffix ();
    compactExpected = compactString (esperado);
    compactActual = compactString (actual);
}

private int findCommonPrefix () {
    int prefijo Índice = 0;
    int end = Math.min (longitud.esperada (), longitud.real ());
    para (; prefijo Índice <fin; prefijo Índice ++) {
        if (esperaba.charAt ( índice de prefijo )! = actual.charAt ( índice de prefijo ))
            rotura;
    }
    return prefixIndex;
}

private int findCommonSuffix () {
    int Sufijo esperado = longitud esperada () - 1;
    int actualSuffix = actual.length () - 1;
    for (; actualSuffix> = prefix Index && esperadoSuffix> = prefixIndex;
            actualSuffix-- ,pectedSuffix--) {
        if (esperado.charAt (esperadoSufijo)! = actual.charAt (actualSufijo))
            rotura;
    }
    return longitud.esperada () - sufijoesperado;
}
```

También deberíamos cambiar los nombres de las variables miembro para que sean un poco más precisos [N1]; después de todo, ambos son índices.

Una inspección cuidadosa de findCommonSuffix expone un *acoplamiento temporal oculto* [G31]; eso depende del hecho de que prefixIndex es calculado por findCommonPrefix . Si estas dos funciones Si se llamaban fuera de servicio, habría una difícil sesión de depuración por delante. Entonces, para exponer este acoplamiento temporal, hagamos que findCommonSuffix tome el prefixIndex como un argumento.

```
private void compactExpectedAndActual () {
    prefixIndex = findCommonPrefix ();
    sufijoIndex = findCommonSuffix ( prefixIndex );
```


Página 291


```
    compactExpected = compactString (esperado);
    compactActual = compactString (actual);
}

private int findCommonSuffix ( int prefixIndex ) {
    int Sufijo esperado = longitud esperada () - 1;
    int actualSuffix = actual.length () - 1;
    para (; actualSuffix> = prefixIndex && esperadoSuffix> = prefixIndex;
            actualSuffix-- ,pectedSuffix--) {
        if (esperado.charAt (esperadoSufijo)! = actual.charAt (actualSufijo))
            rotura;
    }
    return longitud.esperada () - sufijoesperado;
}
```

No estoy muy contento con esto. El paso del argumento prefixIndex es un poco arbitrario [G32]. Funciona para establecer el orden, pero no hace nada para explicar la necesidad de que

ordenar. Otro programador podría deshacer lo que hemos hecho porque no hay indicios que el parámetro es realmente necesario. Así que tomemos un rumbo diferente.

```
private void compactExpectedAndActual () {
    findCommonPrefixAndSuffix ();
    compactExpected = compactString (esperado);
    compactActual = compactString (actual);
}

private void findCommonPrefixAndSuffix () {
    findCommonPrefix ();
    int Sufijo esperado = longitud esperada () - 1;
    int actualSuffix = actual.length () - 1;
    por (;
            actualSuffix> = prefixIndex && esperadoSuffix> = prefixIndex;
            actualSuffix--, esperadoSuffix--
        ) {
        if (esperado.charAt (esperadoSufijo)! = actual.charAt (actualSufijo))
            rotura;
    }
    sufijoIndex = longitud esperada () - Sufijo esperado;
}

private void findCommonPrefix () {
    prefixIndex = 0;
    int end = Math.min (longitud.esperada (), longitud.real ());
    para (; prefixIndex <end; prefixIndex ++)
        si (esperaba.charAt (prefixIndex)! = actual.charAt (prefixIndex))
            rotura;
}
```

Ponemos findCommonPrefix y findCommonSuffix de nuevo como estaban, cambiando el nombre de findCommonSuffix para findCommon **PrefixAnd** Suffix y hacer que llame a findCommon-Prefijo antes de hacer cualquier otra cosa. Que establece la naturaleza temporal de las dos funciones ciones de una manera mucho más dramática que la solución anterior. También señala lo feo findCommonPrefixAndSuffix es. Vamos a limpiarlo ahora.

```
private void findCommonPrefixAndSuffix () {
    findCommonPrefix ();
    int suffixLength = 1;
```


Página 292


```
    para (;! suffixOverlapsPrefix (suffixLength); suffixLength ++) {
        if (charFromEnd (esperado, suffixLength)! =
                charFromEnd (actual, suffixLength))
            rotura;
    }
    sufijoIndex = sufijoLongitud;
}

char privado charFromEnd (String s, int i) {
    return s.charAt (s.length () - i);}

private boolean suffixOverlapsPrefix (int suffixLength) {
    return actual.length () - suffixLength <prefixLength ||
        longitud.esperada () - suffixLength <prefixLength;
}
```

Esto es mucho mejor. Expone que el suffixIndex es realmente la longitud del sufijo y no está bien nombrado. Lo mismo ocurre con prefixIndex , aunque en ese caso "index" y "Longitud" son sinónimos. Aun así, es más consistente usar "longitud". El problema es que la variable suffixIndex no está basada en cero; se basa en 1 y, por lo tanto, no es una longitud real. Esto es también la razón por la que hay todos esos +1 s en computeCommonSuffix [G33]. Así que arreglemos eso. El resultado está en el Listado 15-4.

Listado 15-4
ComparisonCompactor.java (provisional)

```
ComparisonCompactor de clase pública {
...
    private int sufijoLength ;
...
    private void findCommonPrefixAndSuffix () {
```

```
        findCommonPrefix ();
        sufijoLongitud = 0;
        para (;! suffixOverlapsPrefix (suffixLength); suffixLength ++) {
            if (charFromEnd (esperado, suffixLength)! =
                    charFromEnd (actual, suffixLength))
                rotura;
        }
    }

    char privado charFromEnd (String s, int i) {
        return s.charAt (s.length () - i - 1 );
    }

    private boolean suffixOverlapsPrefix (int suffixLength) {
        return actual.length () - suffixLength <= prefixLength ||
            longitud.esperada () - suffixLength <= prefixLength;
    }

...
    private String compactString (String source) {
        Resultado de cadena =
            DELTA_START +
            source.substring (prefixLength, source.length () - suffixLength ) +
            DELTA_END;
        si (prefixLength> 0)
            resultado = computeCommonPrefix () + resultado;
```


Página 293

Listado 15-4 (continuación)

ComparisonCompactor.java (provisional)

```
        si ( suffixLength > 0)
            resultado = resultado + computeCommonSuffix ();
        devolver resultado;
    }

...
    private String computeCommonSuffix () {
        int end = Math.min (esperado.length () - suffixLength +
            contextLength, esperado.length ()
        );
        regreso
            subcadena esperada ( longitud esperada () - longitud del sufijo , fin) +
            ( longitud.esperada () - suffixLength <
                esperaba.length () - contextLength?
                ELIPSIS: "");
    }
```

Reemplazamos el +1 s en computeCommonSuffix con un -1 en charFromEnd , donde hace sentido perfecto, y dos operadores <= en suffixOverlapsPrefix , donde también hacen perfecto sentido. Esto nos permitió cambiar el nombre de suffixIndex a suffixLength , mejorando enormemente ing la legibilidad del código.

Sin embargo, existe un problema. Mientras eliminaba los +1, noté la siguiente línea en compactString :

```
si (suffixLength> 0)
```

Mírelo en el Listado 15-4. Por derechos, porque suffixLength ahora es uno menos que solía ser, debería cambiar el operador > a un operador > = . Pero eso no tiene sentido. Eso tiene sentido *ahora!* Esto significa que no solía tener sentido y probablemente era un error. Bueno, no es un error. Tras un análisis más detallado, vemos que la declaración if ahora evita una sufijo de longitud cero. Antes de realizar el cambio, la declaración if era no funcional porque suffixIndex nunca podría ser menor que uno.

¡Esto pone en duda *tanto las* declaraciones if en compactString ! Parece como si ellos Ambos podrían ser eliminados. Así que comentémoslos y ejecutemos las pruebas. ¡Ellos pasaron! Entonces vamos a reestructurar compactString para eliminar las extrañas declaraciones if y hacer el función mucho más simple [G9].

```
private String compactString (String source) {
        regreso
            computeCommonPrefix () +
```

```
        DELTA_START +
        source.substring (prefixLength, source.length () - suffixLength) +
        DELTA_END +
        computeCommonSuffix ();
    }
```

¡Esto es mucho mejor! Ahora vemos que la función compactString simplemente compone el fragmentos juntos. Probablemente podamos aclarar esto aún más. De hecho, hay muchos

Página 294

limpiezas que pudimos hacer. Pero en lugar de arrastrarlo a través del resto de los cambios, solo mostrarle el resultado en el Listado 15-5.

Listado 15-5

ComparisonCompactor.java (final)

```
paquete junit.framework;

ComparisonCompactor de clase pública {

    Cadena final estática privada ELLIPSIS = "...";
    Cadena final estática privada DELTA_END = "]";
    Cadena final estática privada DELTA_START = "[";

    private int contextLength;
    Se espera una cadena privada;
    Private String actual;
    private int prefixLength;
    private int sufijoLength;

    ComparisonCompactor público (
        int contextLength, String esperado, String actual
    ) {
        this.contextLength = contextLength;
        this.expected = esperado;
        this.actual = actual;
    }

    public String formatCompactedComparison (mensaje de cadena) {
        String compactExpected = esperado;
        String compactActual = actual;
        if (shouldBeCompacted ()) {
            findCommonPrefixAndSuffix ();
            compactExpected = compacto (esperado);
            compactActual = compact (actual);
        }
        return Assert.format (mensaje, compactExpected, compactActual);
    }

    private boolean shouldBeCompacted () {
        return! shouldNotBeCompacted ();
    }

    private boolean shouldNotBeCompacted () {
        retorno esperado == nulo ||
                    actual == nulo ||
                    esperado.equals (real);
    }

    private void findCommonPrefixAndSuffix () {
        findCommonPrefix ();
        sufijoLongitud = 0;
        para (;! suffixOverlapsPrefix (); suffixLength ++) {
            if (charFromEnd (esperado, suffixLength)! =
                    charFromEnd (actual, suffixLength)
            )
```

Página 295

Listado 15-5 (continuación)

ComparisonCompactor.java (final)

```
            rotura;
    }
}

char privado charFromEnd (String s, int i) {
    return s.charAt (s.length () - i - 1);
}

sufijo booleano privadoOverlapsPrefix () {
    return actual.length () - suffixLength <= prefixLength ||
        longitud.esperada () - suffixLength <= prefixLength;
}

private void findCommonPrefix () {
    prefixLength = 0;
    int end = Math.min (longitud.esperada (), longitud.real ());
    para (; prefixLength <end; prefixLength ++)
        if (esperaba.charAt (prefixLength)! = actual.charAt (prefixLength))
            rotura;
}

Private String compact (String s) {
    devolver nuevo StringBuilder ()
        .append (startEllipsis ())
        .append (startContext ())
        .append (DELTA_START)
        .append (delta (s))
        .append (DELTA_END)
        .append (finalizandoContext ())
        .append (terminandoEllipsis ())
        .Encadenar();
}

Private String startEllipsis () {
    return prefixLength> contextLength? ELIPSIS: "";
}

private String startContext () {
    int contextStart = Math.max (0, prefixLength - contextLength);
    int contextEnd = prefixLength;
    devuelve la subcadena esperada (contextStart, contextEnd);
}

Private String delta (String s) {
    int deltaStart = prefixLength;
    int deltaEnd = s.length () - suffixLength;
    return s.substring (deltaStart, deltaEnd);
}

private StringndingContext () {
    int contextStart = esperaba.length () - suffixLength;
    int contextEnd =
        Math.min (contextStart + contextLength, esperaba.length ());
    devuelve la subcadena esperada (contextStart, contextEnd);
}
```


Página 296

Listado 15-5 (continuación)

ComparisonCompactor.java (final)

```
    private String terminandoEllipsis () {
        return (suffixLength> contextLength? ELLIPSIS: "");
    }
}
```

Esto es bastante bonito. El módulo se divide en un grupo de funciones de análisis.
ciones y otro grupo de funciones de síntesis. Están ordenados topológicamente para que el
La definición de cada función aparece justo después de su uso. Aparecen todas las funciones de análisis
primero, y todas las funciones de síntesis aparecen al final.

Si miras con atención, notarás que revoqué varias de las decisiones que tomé.
anteriormente en este capítulo. Por ejemplo, volví a incluir algunos métodos extraídos en
formatCompactedComparison , y he cambiado el sentido de la shouldNotBeCompacted expre-
sion. Esto es típico. A menudo, una refactorización conduce a otra que conduce a la destrucción de la
primero. La refactorización es un proceso iterativo lleno de prueba y error, que inevitablemente converge en
algo que sentimos que es digno de un profesional.

Conclusión

Y así hemos cumplido la Regla de los Boy Scouts. Hemos dejado este módulo un poco más limpio que
lo encontramos. No es que no estuviera limpio ya. Los autores habían hecho un excelente trabajo con él.
Pero ningún módulo es inmune a la mejora y cada uno de nosotros tiene la responsabilidad de
deja el código un poco mejor de lo que lo encontramos.

Esta página se dejó en blanco intencionalmente

dieciséis

Refactorización de SerialDate

Si va a http://www.jfree.org/jcommon/index.php , encontrará la biblioteca JCommon. En lo profundo de esa biblioteca hay un paquete llamado org.jfree.date . Dentro de ese paquete hay una clase llamada SerialDate . Vamos a explorar esa clase.

El autor de SerialDate es David Gilbert. David es claramente un experimentado y programador petent. Como veremos, muestra un importante grado de profesionalismo y disciplina dentro de su código. A todos los efectos, este es un "buen código". Y yo soy va a romperlo en pedazos.

Página 299

Esta no es una actividad de malicia. Tampoco creo que sea mucho mejor que David que de alguna manera tengo derecho a emitir un juicio sobre su código. De hecho, si encontrara alguna de mi código, estoy seguro de que puede encontrar muchas cosas de las que quejarse.

No, esta no es una actividad de maldad o arrogancia. Lo que estoy a punto de hacer no es nada más y nada menos que una reseña profesional. Es algo que todos deberíamos ser cómodo haciéndolo. Y es algo que deberíamos agradecer cuando se haga por nosotros. Es solo a través de críticas como estas que aprenderemos. Los médicos lo hacen. Los pilotos lo hacen. Los abogados hacen eso. Y los programadores también tenemos que aprender a hacerlo.

Una cosa más sobre David Gilbert: David es más que un buen programador. David tuvo el coraje y la buena voluntad de ofrecer su código a la comunidad en general de forma gratuita. Lo puso al aire libre para que todos lo vieran e invitó al uso público y al escrutinio público. Esto estuvo bien hecho!

SerialDate (Listado B-1, página 349) es una clase que representa una fecha en Java. Porque tener una clase que representa una fecha, cuando Java ya tiene java.util.Date y java.util.Calendar y otros? El autor escribió esta clase en respuesta a un dolor que me a menudo me he sentido. El comentario en su Javadoc de apertura (línea 67) lo explica bien. Nosotros podría objetar su intención, pero ciertamente he tenido que lidiar con este problema, y dé la bienvenida a una clase que trata sobre fechas en lugar de horas.

Primero, haz que funcione

Hay algunas pruebas unitarias en una clase llamada SerialDateTests (Listado B-2, página 366). La todas las pruebas pasan. Desafortunadamente, una inspección rápida de las pruebas muestra que no prueban todas cosa [T1]. Por ejemplo, haciendo una búsqueda de "Buscar usos" en el método MonthCodeToQuarter (línea 334) indica que no se usa [F4]. Por lo tanto, las pruebas unitarias no lo prueban.

Así que encendí Clover para ver qué cubrían las pruebas unitarias y qué no. Trébol informó que las pruebas unitarias ejecutaron solo 91 de las 185 declaraciones ejecutables en SerialDate (~ 50 por ciento) [T2]. El mapa de cobertura parece una colcha de retazos, con grandes cantidades de código ecuadrado esparcido por toda la clase.

Mi objetivo era comprender completamente y también refactorizar esta clase. No pude hacer eso sin una cobertura de prueba mucho mayor. Así que escribí mi propio conjunto de artículos completamente independientes. pruebas unitarias (Listado B-4, página 374).

Al examinar estas pruebas, notará que muchas de ellas están comentadas. Estas pruebas no pasaron. Representan el comportamiento que creo que SerialDate debería tener. Así como Refactorizo SerialDate , trabajaré para que estas pruebas también pasen.

Incluso con algunas de las pruebas comentadas, Clover informa que las nuevas pruebas unitarias están ejecutando 170 (92 por ciento) de las 185 declaraciones ejecutables. Esto es bastante bueno, y yo creo que podremos aumentar este número.

Las primeras pruebas comentadas (líneas 23-63) fueron un poco vanidosas de mi parte. La El programa no fue diseñado para pasar estas pruebas, pero el comportamiento me pareció obvio [G2].

Página 300

No estoy seguro de por qué se escribió el método testWeekdayCodeToString en primer lugar, pero debido a que está allí, parece obvio que no debe distinguir entre mayúsculas y minúsculas. Escribiendo estas pruebas fue trivial [T3]. Hacerlos pasar fue aún más fácil; Acabo de cambiar las líneas 259 y 263 a utilice equalsIgnoreCase .

Dejé las pruebas en la línea 32 y la línea 45 comentó porque no me queda claro que las abreviaturas "tues" y "thurs" deben ser compatibles.

Las pruebas de las líneas 153 y 154 no pasan. Claramente, deberían [G2]. Podemos fácilmente arregle esto, y las pruebas en la línea 163 a la línea 213, haciendo los siguientes cambios en el función stringToMonthCode .

```
457             if ((resultado <1) || (resultado> 12)) {
                    resultado = -1;
458                 for (int i = 0; i <monthNames.length; i ++) {
459                     if (s.equalsIgnoreCase (shortMonthNames [i])) {
460                         resultado = i + 1;
461                         rotura;
462                     }
463                     if (s.equalsIgnoreCase (monthNames [i])) {
464                         resultado = i + 1;
465                         rotura;
466                     }
467                 }
468             }
```

La prueba comentada en la línea 318 expone un error en el método getFollowingDayOfWeek (línea 672). El 25 de Diciembre de 2004 fue Sábado. El sábado siguiente fue el 1 de enero, 2005. Sin embargo, cuando ejecutamos la prueba, vemos que getFollowingDayOfWeek devuelve Decem- 25 de ber como el sábado que sigue al 25 de diciembre. Claramente, esto es incorrecto [G3], [T1]. Nosotros vea el problema en la línea 685. Es un error típico de condición de contorno [T5]. Debería leerse como sigue:

```
685             if (baseDOW> = targetWeekday) {
```

Es interesante notar que esta función fue el objetivo de una reparación anterior. El cambio history (línea 43) muestra que los "errores" fueron corregidos en getPreviousDayOfWeek , getFollowing- DayOfWeek y getNearestDayOfWeek [T6].

La prueba unitaria testGetNearestDayOfWeek (línea 329), que prueba el getNearestDayOfWeek método (línea 705), no comenzó tan largo y exhaustivo como lo es actualmente. Agregué mucho de casos de prueba porque no todos mis casos de prueba iniciales pasaron [T6]. Puedes ver el patrón de falla al observar qué casos de prueba se comentan. Ese patrón es revelador [T7]. Muestra que el algoritmo falla si el día más cercano está en el futuro. Claramente hay algunos tipo de error de condición de contorno [T5].

El patrón de cobertura de prueba informado por Clover también es interesante [T8]. Línea 719 nunca se ejecuta! Esto significa que la declaración if en la línea 718 siempre es falsa. Seguro suficiente, una mirada al código muestra que esto debe ser cierto. La variable de ajuste siempre es negativa. ative y, por lo tanto, no puede ser mayor o igual a 4. Entonces, este algoritmo es simplemente incorrecto.

Página 301

El algoritmo correcto se muestra a continuación:

```
int delta = targetDOW - base.getDayOfWeek ();
int PositiveDelta = delta + 7;
int ajuste = PositiveDelta% 7;
si (ajustar> 3)
    ajustar - = 7;

return SerialDate.addDays (ajustar, base);
```

Finalmente, las pruebas en la línea 417 y la línea 429 se pueden hacer pasar simplemente lanzando un IllegalArgumentException en lugar de devolver una cadena de error de weekInMonthToString y relatedToString .

Con estos cambios pasan todas las pruebas unitarias, y creo que SerialDate ahora funciona. Y ahora es hora de hacerlo "bien".

Entonces hazlo bien

Vamos a caminar de arriba a abajo de SerialDate , mejorándolo a medida que avanzamos. a lo largo de. Aunque no verá esto en la discusión, ejecutaré todos los archivos JCommon pruebas unitarias, incluida mi prueba unitaria mejorada para SerialDate , después de cada cambio que hago. Entonces Tenga la seguridad de que todos los cambios que ve aquí funcionan para todo JCommon .

A partir de la línea 1, vemos una serie de comentarios con información de licencia, derechos de autor, autores e historial de cambios. Reconozco que hay ciertas legalidades que deben ser abordado, por lo que los derechos de autor y las licencias deben permanecer. Por otro lado, el cambio his- La historia es un vestigio de la década de 1960. Tenemos herramientas de control de código fuente que hacen esto por nosotros ahora. Este historial debe eliminarse [C1].

La lista de importación que comienza en la línea 61 podría acortarse usando java.text. * Y java.util. * . [J1]

Me estremezco ante el formato HTML en el Javadoc (línea 67). Tener un archivo fuente con más de un idioma me preocupa. Este comentario tiene *cuatro* idiomas: Java, Inglés, Javadoc y html [G1]. Con tantos idiomas en uso, es difícil mantener las cosas derecho. Por ejemplo, el buen posicionamiento de la línea 71 y la línea 72 se pierde cuando el Javadoc se genera y, sin embargo, ¿quién quiere ver <ul> y <li> en el código fuente? Una mejor estrategia podría ser simplemente rodear todo el comentario con <pre> para que el formato que es aparente en el código fuente se conserva dentro del Javadoc. [1]

La línea 86 es la declaración de clase. ¿Por qué esta clase se llama SerialDate ? ¿Cuál es la sig- significado del mundo "serial"? ¿Es porque la clase se deriva de Serializable ? Que no parece probable.

1. Una solución aún mejor hubiera sido que Javadoc presentara todos los comentarios como formateados previamente, de modo que los comentarios aparezcan como lo mismo tanto en el código como en el documento.

Página 302

No te dejaré adivinando. Sé por qué (o al menos creo que sé por qué) la palabra Se utilizó "serial". La pista está en las constantes SERIAL_LOWER_BOUND y SERIAL_UPPER_BOUND en la línea 98 y la línea 101. Una pista aún mejor está en el comentario que comienza en la línea 830. Esta clase se llama SerialDate porque se implementa mediante un "Número de serie", que resulta ser el número de días desde el 30 de diciembre de 1899.

Tengo dos problemas con esto. Primero, el término "número de serie" no es realmente correcto. Esto puede ser una objeción, pero la representación es más un desplazamiento relativo que un número de serie. ber. El término "número de serie" tiene más que ver con los marcadores de identificación del producto que fechas. Así que no encuentro este nombre particularmente descriptivo [N1]. Un término más descriptivo podría ser "ordinal".

El segundo problema es más significativo. El nombre SerialDate implica una implementación ción. Esta clase es una clase abstracta. No hay necesidad de insinuar nada en absoluto sobre el implementación. De hecho, ¡hay una buena razón para ocultar la implementación! Entonces encuentro esto nombre para estar en el nivel incorrecto de abstracción [N2]. En mi opinión, el nombre de esta clase debería ser simplemente Fecha .

Desafortunadamente, ya hay demasiadas clases en la biblioteca de Java llamadas Date , por lo que probablemente este no sea el mejor nombre para elegir. Porque esta clase se trata de días, en lugar de tiempo, consideré nombrarlo Día , pero este nombre también se usa mucho en otros lugares. En el Al final, elegí DayDate como el mejor compromiso.

De ahora en adelante en esta discusión usaré el término DayDate . Te dejo recordar Tenga en cuenta que los listados que está viendo todavía usan SerialDate .

Entiendo por qué DayDate hereda de Comparable y Serializable . Pero por que lo hace heredar de MonthConstants ? La clase MonthConstants (Listado B-3, página 372) es solo una conjunto de constantes finales estáticas que definen los meses. Heredando de clases con constants es un viejo truco que los programadores de Java usaban para evitar el uso de expresiones siones como MonthConstants.January , pero es una mala idea [J2]. MonthConstants realmente debería ser una enumeración.

```
la clase pública abstracta DayDate implementa Comparable,
                                                    Serializable {
  mes public static enum {
    ENERO 1),
    FEBRERO 2),
    3 DE MARZO),
    4 DE ABRIL),
    5 DE MAYO),
    6 DE JUNIO),
    07 DE JULIO),
    AGOSTO (8),
    (9 de septiembre)
    10 DE OCTUBRE),
    11 DE NOVIEMBRE),
    12 DE DICIEMBRE);

    Mes (índice int) {
      this.index = índice;
    }
```


Página 303


```
mes estático público make (int monthIndex) {
  para (Month m: Month.values ()) {
```

```
            if (m.index == monthIndex)
                return m;
        }
        lanzar nueva IllegalArgumentException ("Índice de mes no válido" + monthIndex);
    }
    índice int final público;
}
```

Cambiar MonthConstants a esta enumeración obliga a realizar bastantes cambios en la clase DayDate y todos sus usuarios. Me tomó una hora hacer todos los cambios. Sin embargo, cualquier función que solía tomar un int durante un mes, ahora toma un enumerador de mes . Esto significa que podemos deshacernos del método isValidMonthCode (línea 326), y toda la verificación de errores del código del mes, como eso en monthCodeToQuarter (línea 356) [G5].

A continuación, tenemos la línea 91, serialVersionUID . Esta variable se utiliza para controlar el serializador. Si lo cambiamos, cualquier DayDate escrito con una versión anterior del software no será legible más y dará como resultado una InvalidClassException . Si no declara el serialVersionUID , luego el compilador genera automáticamente una para usted, y será diferente cada vez que realice un cambio en el módulo. Sé que todos los documentos recomiendan el control manual de esta variable, pero me parece que la con- El control de la serialización es mucho más seguro [G4]. Después de todo, prefiero depurar un InvalidClassException que el comportamiento extraño que se produciría si me olvidara de cambiar el serialVersionUID . Así que voy a eliminar la variable, al menos por el momento. [2]

Encuentro el comentario de la línea 93 redundante. Los comentarios redundantes son solo lugares para compartir leer mentiras y desinformación [C2]. Así que me voy a deshacer de él y de los demás.

Los comentarios en la línea 97 y la línea 100 hablan de números de serie, que discutí antes [C1]. Las variables que describen son las fechas más tempranas y últimas posibles que DayDate puede describir. Esto se puede aclarar un poco [N1].

```
public static final int EARLIEST_DATE_ORDINAL = 2; // 1/1/1900
public static final int LATEST_DATE_ORDINAL = 2958465; // 31/12/9999
```

No tengo claro por qué EARLIEST_DATE_ORDINAL es 2 en lugar de 0. Hay una pista en el comentar en la línea 829 que sugiere que esto tiene algo que ver con la forma en que las fechas son representado en Microsoft Excel. Hay una visión mucho más profunda proporcionada en un derivado de DayDate llamado SpreadsheetDate (Listado B-5, página 382). El comentario en la línea 71 describe el problema muy bien.

El problema que tengo con esto es que el problema parece estar relacionado con la implementación- ción de SpreadsheetDate y no tiene nada que ver con DayDate . Concluyo de esto que

2. Varios de los revisores de este texto se han opuesto a esta decisión. Sostienen que en un marco de código abierto es Es mejor afirmar el control manual sobre la ID de serie para que los cambios menores en el software no provoquen que las fechas serializadas antiguas sean inválido. Este es un buen punto. Sin embargo, al menos la falla, por inconveniente que sea, tiene una causa clara. En el otro Por otro lado, si el autor de la clase se olvida de actualizar el ID, entonces el modo de falla no está definido y muy bien podría ser silencioso. I Creo que la verdadera moraleja de esta historia es que no debes esperar deserializar entre versiones.

Página 304

EARLIEST_DATE_ORDINAL y LATEST_DATE_ORDINAL realmente no pertenecen a DayDate y debe moverse a SpreadsheetDate [G6].

De hecho, una búsqueda del código muestra que estas variables se utilizan sólo dentro de SpreadsheetDate . Nada en DayDate , ni en ninguna otra clase en el marco JCommon , utiliza ellos. Por lo tanto, los moveré hacia abajo a SpreadsheetDate .

Las siguientes variables, MINIMUM_YEAR_SUPPORTED y MAXIMUM_YEAR_SUPPORTED (línea 104 y línea 107), proporcionan una especie de dilema. Parece claro que si DayDate es un resumen clase que no proporciona ningún presagio de implementación, entonces no debería informarnos sobre un año mínimo o máximo. Una vez más, estoy tentado a bajar estas variables en SpreadsheetDate [G6]. Sin embargo, una búsqueda rápida de los usuarios de estas variables muestra que otra clase los usa: RelativeDayOfWeekRule (Listado B-6, página 390). Vemos eso uso en la línea 177 y la línea 178 en la función getDate , donde se utilizan para comprobar que el argumento de getDate es un año válido. El dilema es que un usuario de una clase abstracta

necesita información sobre su implementación.

Lo que tenemos que hacer es proporcionar esta información sin contaminar el propio DayDate . Por lo general, obtendríamos información de implementación de una instancia de un derivado. Sin embargo, a la función getDate no se le pasa una instancia de DayDate . Sin embargo, devolver una instancia de este tipo, lo que significa que en algún lugar debe estar creando. Línea 187 a través de la línea 205 proporcione la pista. La instancia de DayDate está siendo creada por uno de los tres funciones, getPreviousDayOfWeek , getNearestDayOfWeek o getFollowingDayOfWeek . Mirando hacia atrás en la lista DayDate , vemos que todas estas funciones (líneas 638–724) devuelven una fecha creada por addDays (línea 571), que llama a createInstance (línea 808), que crea a SpreadsheetDate ! [G7].

Por lo general, es una mala idea que las clases base conozcan sus derivados. Para arreglar esto, nosotros debe utilizar la A ESUMEN F actory [3] patrón y crear un DayDateFactory . Esta fábrica crear las instancias de DayDate que necesitamos y también podemos responder preguntas sobre el implementación, como las fechas máximas y mínimas.

```
clase pública abstracta DayDateFactory {
    fábrica privada estática DayDateFactory = new SpreadsheetDateFactory ();
    public static void setInstance (fábrica DayDateFactory) {
        DayDateFactory.factory = fábrica;
    }

    Resumen protegido DayDate _makeDate (int ordinal);
    resumen protegido DayDate _makeDate (int día, DayDate.Month mes, int año);
    Resumen protegido DayDate _makeDate (int día, int mes, int año);
    resumen protegido DayDate _makeDate (java.util.Date fecha);
    resumen protegido int _getMinimumYear ();
    resumen protegido int _getMaximumYear ();

    public static DayDate makeDate (int ordinal) {
        return factory._makeDate (ordinal);
    }
```

3. [GOF].

Página 305


```
    public static DayDate makeDate (int día, DayDate.Month mes, int año) {
        return factory._makeDate (día, mes, año);
    }

    public static DayDate makeDate (int día, int mes, int año) {
        return factory._makeDate (día, mes, año);
    }

    public static DayDate makeDate (java.util.Date date) {
        return factory._makeDate (fecha);
    }

    public static int getMinimumYear () {
        return factory._getMinimumYear ();
    }

    public static int getMaximumYear () {
        return factory._getMaximumYear ();
    }
}
```

Esta clase de fábrica sustituye a los createInstance métodos con makeDate métodos, que mejora bastante los nombres [N1]. Por defecto es SpreadsheetDateFactory pero puede ser cambiado en cualquier momento para usar una fábrica diferente. Los métodos estáticos que delegan en abstraer métodos utilizan una combinación de la S INGLETON , [4] D ECORATOR , [5] y A ESUMEN F actory patrones que he encontrado útiles.

Los SpreadsheetDateFactory se parece a esto.

```
public class SpreadsheetDateFactory extiende DayDateFactory {
    public DayDate _makeDate (int ordinal) {
        return new SpreadsheetDate (ordinal);
```

```
    }
    public DayDate _makeDate (int día, DayDate.Month mes, int año) {
        return new SpreadsheetDate (día, mes, año);
    }

    public DayDate _makeDate (int día, int mes, int año) {
        return new SpreadsheetDate (día, mes, año);
    }

    public DayDate _makeDate (Fecha fecha) {
        calendario GregorianCalendar final = new GregorianCalendar ();
        calendar.setTime (fecha);
        return new SpreadsheetDate (
            calendar.get (Calendar.DATE),
            DayDate.Month.make (calendar.get (Calendar.MONTH) + 1),
            calendar.get (Calendar.YEAR));
    }
```

4. Ibid.
5. Ibid.

Página 306


```
    protected int _getMinimumYear () {
        return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
    }

    protected int _getMaximumYear () {
        return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
    }
}
```

Como puede ver, ya moví MINIMUM_YEAR_SUPPORTED y MAXIMUM_YEAR_SUPPORTED variables en SpreadsheetDate , donde pertenecen [G6].

El siguiente número en DayDate son las constantes de día que comienzan en la línea 109. Estas deben realmente sea otra enumeración [J3]. Hemos visto este patrón antes, así que no lo repetiré aquí. Usted véalo en los listados finales.

A continuación, vemos una serie de tablas que comienzan con LAST_DAY_OF_MONTH en la línea 140. Mi primera El problema con estas tablas es que los comentarios que las describen son redundantes [C3]. Su los nombres son suficientes. Entonces voy a borrar los comentarios.

No parece haber una buena razón para que esta tabla no sea privada [G8], porque hay un función estática lastDayOfMonth que proporciona los mismos datos.

La siguiente tabla, AGGREGATE_DAYS_TO_END_OF_MONTH , es un poco más misteriosa porque es no se utiliza en ninguna parte del marco JCommon [G9]. Así que lo borré.

Lo mismo ocurre con LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH.

La siguiente tabla, AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH , se usa solo en Spreadsheet Date (línea 434 y línea 473). Esto plantea la pregunta de si debe moverse a SpreadsheetDate . El argumento para no moverlo es que la tabla no es específica de ningún implementación particular [G6]. Por otro lado, ninguna implementación que no sea SpreadsheetDate realmente existe, por lo que la tabla debe moverse cerca de donde está utilizado [G10].

Lo que me resuelve el argumento es que, para ser coherentes [G11], deberíamos hacer tabla privada y exponerla a través de una función como julianDateOfLastDayOfMonth . Nadie parece necesitar una función como esa. Además, la tabla se puede volver a mover fácilmente a DayDate si alguna nueva implementación de DayDate lo necesita. Así que lo moví.

Lo mismo ocurre con la mesa, LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH .

A continuación, vemos tres conjuntos de constantes que se pueden convertir en enumeraciones (líneas 162-205). El primero de los tres selecciona una semana dentro de un mes. Lo cambié a una enumeración llamada WeekInMonth .

```
public enum WeekInMonth {
```

```
    PRIMERO (1), SEGUNDO (2), TERCERO (3), CUARTO (4), ÚLTIMO (0);
    índice int final público;

    WeekInMonth (índice int) {
        this.index = índice;
    }
}
```


Página 307

El segundo conjunto de constantes (líneas 177-187) es un poco más oscuro. El INCLUDE_NONE , Las constantes INCLUDE_FIRST , INCLUDE_SECOND e INCLUDE_BOTH se utilizan para describir si las fechas finales definitorias de un rango deben incluirse en ese rango. Matemáticamente, esto se describe utilizando los términos "intervalo abierto", "intervalo semiabierto" e "intervalo cerrado val. " Creo que es más claro usar la nomenclatura matemática [N3], así que la cambié a una enumeración denominada DateInterval con los enumeradores CLOSED , CLOSED_LEFT , CLOSED_RIGHT y OPEN .

El tercer conjunto de constantes (líneas 18-205) describe si una búsqueda de un determinado día de la semana debe resultar en la última instancia, la siguiente o la más cercana. Decidir que llamar esto es difícil en el mejor de los casos. Al final, me conformé con WeekdayRange con LAST , NEXT y NEAREST enumeradores.

Es posible que no esté de acuerdo con los nombres que he elegido. Tienen sentido para mí, pero puede que no tenga sentido para usted. El punto es que ahora están en una forma que los hace fáciles para cambiar [J3]. Ya no se pasan como números enteros; se pasan como símbolos. puedo utilizar la función "cambiar nombre" de mi IDE para cambiar los nombres, o los tipos, sin preocuparse de que me perdí un -1 o 2 en algún lugar del código o que algún argumento int declaración se deja mal descrita.

El campo de descripción en la línea 208 no parece ser utilizado por nadie. Lo borré a lo largo con su descriptor de acceso y su mutador [G9].

También eliminé el constructor predeterminado degenerado en la línea 213 [G12]. El compilador generelo para nosotros.

Podemos omitir el método isValidWeekdayCode (líneas 216-238) porque eliminamos cuando creamos la enumeración Day .

Esto nos lleva al método stringToWeekdayCode (líneas 242-270). Javadocs que no agregan mucho a la firma del método son simplemente desorden [C3], [G12]. El único valor de este Javadoc agrega es la descripción del valor de retorno -1 . Sin embargo, debido a que cambiamos al Enumeración de días , el comentario es realmente incorrecto [C2]. El método ahora arroja un IllegalArgumentException . Entonces eliminé el Javadoc.

También eliminé todas las palabras clave finales en argumentos y declaraciones de variables. Hasta Me di cuenta de que no agregaron ningún valor real, pero sí agregaron al desorden [G12]. Eliminando final va en contra de la sabiduría convencional. Por ejemplo, Robert Simmons [6] fuertemente nos recomienda “. . . difundir la final por todo el código ". Claramente no estoy de acuerdo. pienso que hay algunos buenos usos para final , como la constante final ocasional , pero por lo demás la palabra clave agrega poco valor y crea mucho desorden. Quizás me siento así porque el Las pruebas unitarias que escribo ya detectan los tipos de errores que puede detectar el final .

No me importaron las declaraciones if duplicadas [G5] dentro del bucle for (línea 259 y línea 263), así que los conecté en una sola declaración if usando el || operador. Yo también usé la enumeración Day para dirigir el bucle for e hizo algunos otros cambios cosméticos.

Se me ocurrió que este método realmente no pertenece a DayDate . Es realmente el función de análisis sintáctico del día . Así que lo moví a la enumeración Día . Sin embargo, eso hizo que el día

6. [Simmons04], pág. 73.

Página 308

enumeración bastante grande. Debido a que el concepto de Día no depende de DayDate , me mudé la enumeración Day fuera de la clase DayDate en su propio archivo fuente [G13].

También moví la siguiente función, weekdayCodeToString (líneas 272–286) al día enumeration y lo llamó toString .

```
public enum Day {
    LUNES (Calendario.LUNES),
    MARTES (Calendario.MARTES),
    MIÉRCOLES (Calendario MIÉRCOLES), s
    JUEVES (Calendario. JUEVES),
    VIERNES (Calendario VIERNES),
    SÁBADO (Calendario.SÁBADO),
    DOMINGO (Calendario DOMINGO);

    índice int final público;
    DateFormatSymbols estático privado dateSymbols = new DateFormatSymbols ();

    Día (int día) {
        índice = día;
    }

    public static Day make (int index) arroja IllegalArgumentException {
        para (Día d: Day.values ())
            si (d.index == índice)
                return d;
        lanzar una nueva IllegalArgumentException (
            String.format ("Índice de días ilegales:% d.", Índice));
    }

    El análisis de día estático público (String s) arroja IllegalArgumentException {
        String [] shortWeekdayNames =
            dateSymbols.getShortWeekdays ();
        String [] weekDayNames =
            dateSymbols.getWeekdays ();

        s = s.trim ();
        para (Day day: Day.values ()) {
            if (s.equalsIgnoreCase (shortWeekdayNames [day.index]) ||
                    s.equalsIgnoreCase (weekDayNames [day.index])) {
                día de regreso;
            }
        }
        lanzar una nueva IllegalArgumentException (
            String.format ("% s no es una cadena de día de la semana válida", s));
    }

    public String toString () {
        return dateSymbols.getWeekdays () [índice];
    }
}
```

Hay dos funciones getMonths (líneas 288-316). El primero llama al segundo. La el segundo nunca es llamado por nadie más que el primero. Por lo tanto, colapsé los dos en uno y los simplificó enormemente [G9], [G12], [F4]. Finalmente, cambié el nombre para ser un poco más descriptivo [N1].

Página 309

```
Cadena estática pública [] getMonthNames () {
    return dateFormatSymbols.getMonths ();
}
```

La función isValidMonthCode (líneas 326–346) se volvió irrelevante por el mes enum, así que lo borré [G9].

La función monthCodeToQuarter (líneas 356–375) huele a F EATURE E NVY [7] [G14] y probablemente pertenece a la enumeración Month como un método llamado trimestre . Así que lo reemplacé.

```
public int quarter () {
    return 1 + (índice-1) / 3;
}
```

Esto hizo que la enumeración Month fuera lo suficientemente grande como para estar en su propia clase. Así que lo saqué de DayDate para que sea coherente con la enumeración Day [G11], [G13].

Los dos métodos siguientes se denominan monthCodeToString (líneas 377–426). De nuevo vemos el patrón de un método que llama a su gemelo con una bandera. Suele ser una mala idea pasar una bandera como un argumento para una función, especialmente cuando esa bandera simplemente selecciona el formato de la salida poner [G15]. Cambié el nombre, simplifiqué y reestructuré estas funciones y las moví a la Enumeración del mes [N1], [N3], [C3], [G14].

```
public String toString () {
    return dateFormatSymbols.getMonths () [índice - 1];
}

public String toShortString () {
    return dateFormatSymbols.getShortMonths () [índice - 1];
}
```

El siguiente método es stringToMonthCode (líneas 428–472). Lo renombré, lo moví a la Mes enum, y lo simplificó [N1], [N3], [C3], [G14], [G12].

```
Análisis de mes estático público (String s) {
    s = s.trim ();
    para (Month m: Month.values ())
        si (m. coincide (s))
            return m;

    intentar {
        return make (Integer.parseInt (s));
    }
    catch (NumberFormatException e) {}
    lanzar una nueva IllegalArgumentException ("Mes no válido" + s);
}
```

7. [Refactorización].

Página 310


```
coincidencias booleanas privadas (String s) {
    devuelve un caso igual a Ignore (toString ()) ||
            s.equalsIgnoreCase (toShortString ());
}
```

El método isLeapYear (líneas 495–517) se puede hacer un poco más expresivo [G16].

```
public static boolean isLeapYear (int year) {
    booleano cuarto = año% 4 == 0;
    centésimo booleano = año% 100 == 0;
```

```
    booleano cuatrocientos = año% 400 == 0;
    devuelve cuarto && (! centésimo || cuatrocientos);
}
```

La siguiente función, leapYearCount (líneas 519–536) no pertenece realmente a DayDate . Nadie lo llama excepto por dos métodos en SpreadsheetDate . Así que lo empujé hacia abajo [G6].

La función lastDayOfMonth (líneas 538–560) hace uso de LAST_DAY_OF_MONTH formación. Esta matriz realmente pertenece a la enumeración Month [G17], así que la moví allí. Yo también simplifico fied la función y la hizo un poco más expresiva [G16].

```
public static int lastDayOfMonth (mes mes, int año) {
    if (month == Month.FEBRUARY && isLeapYear (año))
        return month.lastDay () + 1;
    demás
        return month.lastDay ();
}
```

Ahora las cosas empiezan a ponerse un poco más interesantes. La siguiente función es addDays (líneas 562–576). En primer lugar, debido a que esta función opera en las variables de DayDate , no debería ser estático [G18]. Así que lo cambié a un método de instancia. En segundo lugar, llama a la función toSerial . Esta función debería cambiarse a Ordinal [N1]. Finalmente, el método puede ser simplificado.

```
public DayDate addDays (int días) {
    return DayDateFactory.makeDate (toOrdinal () + días);
}
```

Lo mismo ocurre con addMonths (líneas 578–602). Debe ser un método de instancia [G18]. El algoritmo es un poco complicado, por lo que utiliza E XPLAINING T provisoria V ARIABLES [8] [G19] para hacerlo más transparente. También cambié el nombre del método getYYY a getYear [N1].

```
public DayDate addMonths (int meses) {
    int thisMonthAsOrdinal = 12 * getYear () + getMonth (). index - 1;
    int resultMonthAsOrdinal = thisMonthAsOrdinal + meses;
    int resultYear = resultMonthAsOrdinal / 12;
    Mes resultMonth = Month.make (resultMonthAsOrdinal% 12 + 1);
```

8. [Beck97].

Página 311


```
    int lastDayOfResultMonth = lastDayOfMonth (resultMonth, resultYear);
    int resultDay = Math.min (getDayOfMonth (), lastDayOfResultMonth);
    return DayDateFactory.makeDate (resultDay, resultMonth, resultYear);
}
```

La función addYears (líneas 604–626) no ofrece sorpresas con respecto a las demás.

```
public DayDate plusYears (int años) {
    int resultYear = getYear () + años;
    int lastDayOfMonthInResultYear = lastDayOfMonth (getMonth (), resultYear);
    int resultDay = Math.min (getDayOfMonth (), lastDayOfMonthInResultYear);
    return DayDateFactory.makeDate (resultDay, getMonth (), resultYear);
}
```

Hay una pequeña picazón en el fondo de mi mente que me está molestando por cambiar estos métodos de estático a instancia. ¿La expresión date.addDays (5) la convierte en ¿ Está claro que el objeto de fecha no cambia y que se devuelve una nueva instancia de DayDate ? ¿O implica erróneamente que estamos agregando cinco días al objeto de fecha ? Podrías No creo que sea un gran problema, pero un poco de código que se parece al siguiente puede ser muy engañar [G20].

```
DayDate date = DateFactory.makeDate (5, Month.DECEMBER, 1952);
date.addDays (7); // aumentar la fecha en una semana.
```

Alguien que lea este código probablemente acepte que addDays está cambiando el

objeto de fecha . Entonces necesitamos un nombre que rompa esta ambigüedad [N4]. Así que cambié los nombres a plusDays y plusMonths . Me parece que la intencin del mtodo es capturada muy bien por

```
DayDate date = oldDate.plusDays (5);
```

mientras que lo siguiente no se lee con la suficiente fluidez como para que un lector simplemente acepte que el se cambia el objeto de fecha :

```
date.plusDays (5);
```

Los algoritmos continúan volviéndose más interesantes. getPreviousDayOfWeek (líneas 628–660) funciona pero es un poco complicado. Después de pensar un poco sobre lo que realmente estaba pasando [G21], yo era capaz de simplificar y utilizar E XPLAINING T provisoria V ARIABLES [G19] para hazlo más claro. También lo cambié de un método estático a un método de instancia [G18], y se deshizo del método de instancia duplicada [G5] (líneas 997–1008).

```
public DayDate getPreviousDayOfWeek (Day targetDayOfWeek) {
    int offsetToTarget = targetDayOfWeek.index - getDayOfWeek (). index;
    si (offsetToTarget> = 0)
        offsetToTarget - = 7;
    return plusDays (offsetToTarget);
}
```

Se produjo exactamente el mismo análisis y resultado para getFollowingDayOfWeek (líneas 662–693).

```
public DayDate getFollowingDayOfWeek (Day targetDayOfWeek) {
    int offsetToTarget = targetDayOfWeek.index - getDayOfWeek (). index;
    si (offsetToTarget <= 0)
```


Página 312


```
        offsetToTarget + = 7;
    return plusDays (offsetToTarget);
}
```

La siguiente función es getNearestDayOfWeek (líneas 695–726), que corregimos en la página 270. Pero los cambios que hice en ese entonces no son consistentes con el patrón actual en las dos últimas funciones [G11]. Así que lo hice coherente y utilicé algunos T EMPO DE EXPLICACIÓN Rary V ARIABLES [G19] para aclarar el algoritmo.

```
public DayDate getNearestDayOfWeek (último día targetDay) {
    int offsetToThisWeeksTarget = targetDay.index - getDayOfWeek (). index;
    int offsetToFutureTarget = (offsetToThisWeeksTarget + 7)% 7;
    int offsetToPreviousTarget = offsetToFutureTarget - 7;

    si (offsetToFutureTarget> 3)
        return plusDays (offsetToPreviousTarget);
    demás
        return plusDays (offsetToFutureTarget);
}
```

El método getEndOfCurrentMonth (líneas 728 a 740) es un poco extraño porque es un método de instancia que envidia [G14] a su propia clase tomando un argumento DayDate . Lo hice un verdadero método de instancia y aclaró algunos nombres.

```
public DayDate getEndOfMonth () {
    Mes mes = getMonth ();
    int año = getYear ();
    int lastDay = lastDayOfMonth (mes, año);
    return DayDateFactory.makeDate (lastDay, month, year);
}
```

Refactorizar weekInMonthToString (líneas 742–761) resultó ser muy interesante Por supuesto. Usando las herramientas de refactorización de mi IDE, primero moví el método a WeekInMonth enumeración que creé en la página 275. Luego cambié el nombre del método a toString . Siguiente yo lo cambió de un método estático a un método de instancia. Todas las pruebas aún pasaron. (Puedes adivina a donde voy?)

A continuación, eliminé el método por completo. Cinco afirmaciones fallaron (líneas 411 a 415, Listado B-4,

página 374). Cambié estas líneas para usar los nombres de los enumeradores (PRIMERO, SEGUNDO . . .). Pasaron todas las pruebas. ¿Puedes ver por qué? ¿Puedes ver también por qué cada uno de estos pasos fueron necesarios? La herramienta de refactorización se aseguró de que todos los llamadores anteriores de weekInMonthToString ahora llamado toString en el enumerador weekInMonth porque todos enu-
Los meradores implementan toString para simplemente devolver sus nombres. . . .

Desafortunadamente, fui demasiado inteligente. Tan elegante como esa maravillosa cadena de refactorización. fue, finalmente me di cuenta de que los únicos usuarios de esta función eran las pruebas que acababa de modificar. ified, así que eliminé las pruebas.

Si me engañas una vez, la culpa es tuya. ¡Si me engañas dos veces, la culpa es mía! Entonces, después de determinar eso nadie más que las pruebas llamadas relativasToString (líneas 765–781), simplemente eliminé el función y sus pruebas.

Página 313

Finalmente hemos llegado a los métodos abstractos de esta clase abstracta. Y el primero es tan apropiado como vienen: toSerial (líneas 838–844). De vuelta en la página 279 había cambiado el nombre a toOrdinal . Habiéndolo visto en este contexto, decidí que el nombre debería ser cambiado a getOrdinalDay .

El siguiente método abstracto es toDate (líneas 838–844). Convierte un DayDate en un java.util.Date . ¿Por qué este método es abstracto? Si miramos su implementación en SpreadsheetDate (líneas 198–207, Listado B-5, página 382), vemos que no depende de cualquier cosa en la implementación de esa clase [G6]. Así que lo empujé hacia arriba.

Los métodos getYYYY , getMonth y getDayOfMonth son muy abstractos. sin embargo, el El método getDayOfWeek es otro que debe extraerse de SpreadSheetDate porque no depende de nada que no se pueda encontrar en DayDate [G6]. ¿O lo hace?

Si observa con atención (línea 247, Listado B-5, página 382), verá que el algoritmo implícitamente depende del origen del día ordinal (en otras palabras, el día de la semana de día 0). Entonces, aunque esta función no tiene dependencias físicas que no se puedan mover a DayDate , tiene una dependencia lógica.

Las dependencias lógicas como esta me molestan [G22]. Si algo lógico depende de la implementación, entonces algo físico también debería. Además, me parece que el algoritmo en sí podría ser genérico con una porción mucho más pequeña de él dependiendo de la implementación [G6].

Así que creé un método abstracto en DayDate llamado getDayOfWeekForOrdinalZero y lo implementó en SpreadsheetDate para devolver Day.SATURDAY . Luego moví el getDayOfWeek método hasta DayDate y lo cambió para llamar a getOrdinalDay y getDayOfWeekForOrdinal-Cero .

```
public Day getDayOfWeek () {
        Día de inicio de día = getDayOfWeekForOrdinalZero ();
        int startOffset = startDay.index - Day.SUNDAY.index;
        return Day.make ((getOrdinalDay () + startOffset)% 7 + 1);
    }
```

Como nota al margen, mire detenidamente el comentario de la línea 895 a la línea 899. ¿Fue esto repetición realmente necesaria? Como de costumbre, eliminé este comentario junto con todos los demás.

El siguiente método es comparar (líneas 902–913). Nuevamente, este método es inapropiadamente resumen [G6], así que levanté la implementación en DayDate . Además, el nombre no comunicar lo suficiente [N1]. Este método en realidad devuelve la diferencia en días desde que argumento. Así que cambié el nombre a daysSince . Además, noté que no había pruebas para este método, así que los escribí.

Las siguientes seis funciones (líneas 915-980) son todos métodos abstractos que deben implementarse Mented en DayDate . Así que los saqué todos de SpreadsheetDate .

La última función, isInRange (líneas 982-995) también necesita ser extraída y refaccionado. tored. La declaración de cambio es un poco fea [G23] y se puede reemplazar moviendo las cajas en la enumeración DateInterval .

Página 314


```
public enum DateInterval {
        ABIERTO {
            public boolean isIn (int d, int left, int right) {
                return d> izquierda && d <derecha;
            }
        },
        CLOSED_LEFT {
            public boolean isIn (int d, int left, int right) {
                return d> = izquierda && d <derecha;
            }
        },
        CLOSED_RIGHT {
            public boolean isIn (int d, int left, int right) {
                return d> izquierda && d <= derecha;
            }
        },
        CERRADO {
            public boolean isIn (int d, int left, int right) {
                return d> = izquierda && d <= derecha;
            }
        };

        public abstract boolean isIn (int d, int left, int right);
    }


public boolean isInRange (DayDate d1, DayDate d2, DateInterval interval) {
        int left = Math.min (d1.getOrdinalDay (), d2.getOrdinalDay ());
        int right = Math.max (d1.getOrdinalDay (), d2.getOrdinalDay ());
        return interval.isIn (getOrdinalDay (), izquierda, derecha);
    }
```

Eso nos lleva al final de DayDate . Así que ahora haremos una pasada más sobre el conjunto. clase para ver qué tan bien fluye.

Primero, el comentario de apertura está desactualizado, así que lo acorté y lo mejoré [C2].

A continuación, moví todas las enumeraciones restantes a sus propios archivos [G12].

A continuación, moví la variable estática (dateFormatSymbols) y tres métodos estáticos (getMonthNames , isLeapYear , lastDayOfMonth) en una nueva clase denominada DateUtil [G6].

Moví los métodos abstractos a la parte superior donde pertenecen [G24].

Cambié Month.make a Month.fromInt [N1] e hice lo mismo con todas las demás enumeraciones. También creé un descriptor de acceso toInt () para todas las enumeraciones e hice que el campo de índice fuera privado.

Hubo una duplicación interesante [G5] en plusYears y plusMonths que estaba capaz de eliminar mediante la extracción de un nuevo método llamado correctLastDayOfMonth , haciendo que el los tres métodos mucho más claros.

Me deshice del número mágico 1 [G25], reemplazándolo con Month.JANUARY.toInt () o Day.SUNDAY.toInt () , según corresponda. Pasé un poco de tiempo con SpreadsheetDate , limpiando los algoritmos un poco. El resultado final se encuentra en el Listado B-7, página 394, hasta Listado B-16, página 405.

Página 315

Curiosamente, la cobertura del código en DayDate ha *disminuido* al 84,9 por ciento. Esto no es porque se está probando menos funcionalidad; más bien es porque la clase se ha reducido tanto que las pocas líneas descubiertas tienen un mayor peso. DayDate ahora tiene 45 de 53 ejecu- declaraciones capaces cubiertas por pruebas. Las líneas descubiertas son tan triviales que no valían la pena pruebas.

Conclusión

Una vez más, hemos seguido la regla de los Boy Scouts. Hemos comprobado el código con un poco más de limpieza. que cuando lo comprobamos. Tomó un poco de tiempo, pero valió la pena. La cobertura de la prueba fue aumentado, se corrigieron algunos errores, se aclaró y se redujo el código. La próxima persona en Si mira este código, es de esperar que le resulte más fácil de manejar que a nosotros. Esa persona también probablemente pueda limpiarlo un poco más que nosotros.

Bibliografía

[GOF]: *Patrones de diseño: elementos de software orientado a objetos reutilizables* , Gamma et al., Addison-Wesley, 1996.

[Simmons04]: *Hardcore Java* , Robert Simmons, Jr., O'Reilly, 2004.

[Refactorización]: *Refactorización: Mejora del diseño del código existente* , Martin Fowler et al., Addison-Wesley, 1999.

[Beck97]: *Patrones de mejores prácticas de Smalltalk* , Kent Beck, Prentice Hall, 1997.

17

Olores y heurística

En su maravilloso libro *Refactoring* , [1] Martin Fowler identificó muchos códigos diferentes Huele ". La lista que sigue incluye muchos de los olores de Martin y agrega muchos más de mis propio. También incluye otras perlas y heurísticas que utilizo para practicar mi oficio.

1. [Refactorización].

Página 317

Compilé esta lista recorriendo varios programas diferentes y refactorizando ellos. A medida que hacía cada cambio, me preguntaba *por qué* hice ese cambio y luego escribí el razón aquí abajo. El resultado es una lista bastante larga de cosas que me huelen mal cuando leo código.

Esta lista está pensada para leerse de arriba a abajo y también para usarse como referencia. Hay una referencia cruzada para cada heurística que le muestra dónde se hace referencia en el el resto del texto en el “Apéndice C” en la página 409.

Comentarios

C1: *Información inapropiada*

Es inapropiado que un comentario contenga información mejor mantenida en un tipo diferente de sistema. tem como su sistema de control de código fuente, su sistema de seguimiento de problemas o cualquier otro

sistema de mantenimiento de registros. Cambie los historiales, por ejemplo, simplemente desordene los archivos de origen con volúmenes de texto histórico y poco interesante. En general, metadatos como autores, últimos fecha de modificación, número de SPR, etc., no deberían aparecer en los comentarios. Los comentarios deben reservarse para notas técnicas sobre el código y el diseño.

C2: *Comentario obsoleto*

Un comentario que se ha vuelto antiguo, irrelevante e incorrecto es obsoleto. Los comentarios envejecen rápidamente. Es mejor no escribir un comentario que quede obsoleto. Si encuentra un obsoleto comentario, es mejor actualizarlo o deshacerse de él lo más rápido posible. Comentarios obsoletos tienden a migrar lejos del código que alguna vez describieron. Se convierten en islas flotantes de irrelevancia y desvío en el código.

C3: *Comentario redundante*

Un comentario es redundante si describe algo que se describe a sí mismo adecuadamente. Para ejemplo:

```
i ++; // incrementar i
```

Otro ejemplo es un Javadoc que dice nada más que (o incluso menos) la función firma:

```
/ **
  * @param sellRequest
  * @regreso
  * @throws ManagedComponentException
  * /
public SellResponse beginSellItem (SellRequest sellRequest)
    lanza ManagedComponentException
```

Los comentarios deben decir cosas que el código no puede decir por sí mismo.

Página 318

C4: *Comentario mal escrito*

Vale la pena escribir bien un comentario que valga la pena escribir. Si vas a escribir un comentario, tómese el tiempo para asegurarse de que sea el mejor comentario que pueda escribir. Elige tus palabras con cuidado. Utilice la gramática y la puntuación correctas. No divagues. No diga lo obvio. Ser breve.

C5: *Código comentado*

Me vuelve loco ver fragmentos de código comentados. Quien sabe cuantos años tiene ¿es? ¿Quién sabe si es significativo o no? Sin embargo, nadie lo eliminará porque todos asume que alguien más lo necesita o tiene planes para ello.

Ese código se queda ahí y se pudre, volviéndose cada vez menos relevante con cada día que pasa. Eso llama a funciones que ya no existen. Utiliza variables cuyos nombres han cambiado. Sigue convenciones que han quedado obsoletas durante mucho tiempo. Contamina los módulos que lo contienen y distrae al personas que intentan leerlo. El código comentado es una *abominación* .

Cuando vea el código comentado, *elimínelo.* No se preocupe, el control del código fuente el sistema todavía lo recuerda. Si alguien realmente lo necesita, puede regresar y consultar un versión previa. No sufra el código comentado para sobrevivir.

Ambiente

E1: La *construcción requiere más de un paso*

La construcción de un proyecto debería ser una sola operación trivial. No debería tener que comprobar muchos

pequeñas piezas del control del código fuente. No deberías necesitar una secuencia de combinaciones arcanas. mands o scripts dependientes del contexto para construir los elementos individuales. Debería no tener que buscar de cerca y de lejos todos los pequeños archivos JAR, archivos XML y otros artefactos que requiere el sistema. Usted *debe* ser capaz de revisar el sistema con un simple comando y luego emita otro comando simple para construirlo.

```
svn obtener mySystem
cd mySystem
hormiga todo
```

E2: *Las pruebas requieren más de un paso*

Debería poder ejecutar *todas* las pruebas unitarias con un solo comando. En el mejor de los casos tu puede ejecutar todas las pruebas haciendo clic en un botón en su IDE. En el peor de los casos deberías ser capaz de emitir un solo comando simple en un shell. Poder ejecutar todas las pruebas es tan fundamental y tan importante que debería ser rápido, fácil y obvio de hacer.

Página 319

Funciones

F1: *Demasiados argumentos*

Las funciones deben tener una pequeña cantidad de argumentos. Ningún argumento es mejor, seguido de uno, dos y tres. Más de tres es muy cuestionable y debe evitarse con prejuicios. udice. (Consulte "Argumentos de funciones" en la página 40.)

F2: *Argumentos de salida*

Los argumentos de salida son contradictorios. Los lectores esperan que los argumentos sean entradas, no salidas pone. Si su función debe cambiar el estado de algo, haga que cambie el estado del objeto al que se llama. (Consulte "Argumentos de salida" en la página 45.)

F3: *Argumentos de banderas*

Los argumentos booleanos declaran en voz alta que la función hace más de una cosa. Ellos son confuso y debe eliminarse. (Consulte “Argumentos de banderas” en la página 41.)

F4: *Función muerta*

Los métodos que nunca se llaman deben descartarse. Mantener el código muerto es un desperdicio. No tenga miedo de eliminar la función. Recuerde, su sistema de control de código fuente todavía lo recuerda.

General

G1: *varios idiomas en un archivo de origen*

Los entornos de programación modernos de hoy hacen posible poner muchos lenguajes diferentes en un solo archivo de origen. Por ejemplo, un archivo fuente de Java puede contener fragmentos de XML, HTML, YAML, JavaDoc, inglés, JavaScript, etc. Para otro ejemplo, además de HTML un archivo JSP puede contener Java, una sintaxis de biblioteca de etiquetas, comentarios en inglés, Javadocs, XML, JavaScript, etc. Esto es confuso en el mejor de los casos y descuidado en el peor.

Lo ideal es que un archivo fuente contenga un solo idioma. Siendo realistas, nosotros probablemente tendrá que usar más de uno. Pero debemos esforzarnos por minimizar tanto la número y extensión de idiomas adicionales en nuestros archivos fuente.

G2: El *comportamiento obvio no se ha implementado*

Siguiendo el "Principio de la menor sorpresa", [2] cualquier función o clase debe implementar el comportamientos que otro programador razonablemente podría esperar. Por ejemplo, considere un función que traduce el nombre de un día a una enumeración que representa el día.

2. O "El principio del menor asombro ": http://en.wikipedia.org/wiki/Principio_de_mínimo_asombro

Página 320


```
Día día = DayDate.StringToDay (String dayName);
```

Esperaríamos que la cadena "Monday" se tradujera a Day.MONDAY . También esperaríamos las abreviaturas comunes a traducir, y esperaríamos que la función ignorara caso.

Cuando no se implementa un comportamiento obvio, los lectores y usuarios del código no pueden ya dependen de su intuición sobre los nombres de las funciones. Pierden la confianza en el original autor y debe recurrir a la lectura de los detalles del código.

G3: *Comportamiento incorrecto en los límites*

Parece obvio decir que el código debería comportarse correctamente. El problema es que rara vez darse cuenta de lo complicado que es el comportamiento correcto. Los desarrolladores suelen escribir funciones que creen que funcionará, y luego confían en su intuición en lugar de esforzarse por demostrar que su código funciona en todos los casos de esquina y límite.

No hay sustituto para la debida diligencia. Cada condición de frontera, cada rincón caso, cada peculiaridad y excepción representa algo que puede confundir a un elegante y algoritmo intuitivo. *No confíe en su intuición* . Busque todas las condiciones de contorno y escribe una prueba para ello.

G4: *Seguridad anulada*

Chernobyl se derritió porque el gerente de la planta anuló cada uno de los mecanismos de seguridad. nismos uno por uno. Los dispositivos de seguridad hacían que fuera incómodo realizar un experimento. La El resultado fue que el experimento no se llevó a cabo, y el mundo vio su primer civil importante catástrofe nuclear.

Es arriesgado anular las seguridades. Ejercer control manual sobre serialVersionUID puede ser necesario, pero siempre arriesgado. Desactivar ciertas advertencias del compilador (¡o todas las advertencias!) puede ayudarlo a que la compilación tenga éxito, pero a riesgo de interminables sesiones de depuración. Turno- Salir de las pruebas reprobadas y decirte a ti mismo que las conseguirás aprobar más tarde es tan malo como fingir sus tarjetas de crédito son dinero gratis.

G5: *Duplicación*

Ésta es una de las reglas más importantes de este libro y debe tomarla muy en serio. Prácticamente todos los autores que escriben sobre diseño de software mencionan esta regla. Dave Thomas y Andy Hunt lo llamó el principio DRY [3] (Don't Repeat Yourself). Kent Beck lo hizo uno de los principios básicos de la Programación Extrema y lo llamó: "Una vez, y solo una vez". Ron Jeffries clasifica esta regla en segundo lugar, justo debajo de obtener todas las pruebas para aprobar.

Cada vez que ve una duplicación en el código, representa una oportunidad perdida para abstracción. Esa duplicación probablemente podría convertirse en una subrutina o tal vez en otra clase absoluta. Al doblar la duplicación en tal abstracción, aumenta el vocabulario ulary del lenguaje de su diseño. Otros programadores pueden usar las instalaciones abstractas

3. [PRAG].

Página 321

tu creas. La codificación se vuelve más rápida y menos propensa a errores porque ha aumentado la nivel de abstracción.

La forma más obvia de duplicación es cuando tiene grupos de código idéntico que parece que algunos programadores se volvieron locos con el mouse, pegando el mismo código una y otra vez otra vez. Estos deben reemplazarse con métodos simples.

Una forma más sutil es la cadena switch / case o if / else que aparece una y otra vez en varios módulos, siempre probando el mismo conjunto de condiciones. Estos deben ser reemplazados con polimorfismo.

Aún más sutiles son los módulos que tienen algoritmos similares, pero que no comparten líneas de código similares. Esto sigue siendo una duplicación y debe abordarse utilizando el T EM-M ÉTODO DE PLACA , patrón [4] o S TRATEGIA [5] .

De hecho, la mayora de los patrones de diseo que han aparecido en los ltimos quince aos son Utilice métodos conocidos para eliminar la duplicación. Así también las Formas Normales de Codd son una estrategia egy para eliminar la duplicación en los esquemas de la base de datos. OO en sí mismo es una estrategia para organizar módulos y eliminando la duplicación. No es sorprendente que también lo sea la programación estructurada.

Creo que se ha aclarado el punto. Encuentre y elimine la duplicación siempre que pueda.

G6: *Código en un nivel incorrecto de abstracción*

Es importante crear abstracciones que separen los conceptos generales de nivel superior de los de nivel inferior. conceptos detallados de nivel. A veces hacemos esto creando clases abstractas para contener el conceptos de nivel superior y derivados para mantener los conceptos de nivel inferior. Cuando hacemos esto, debemos asegurarnos de que la separación sea completa. Queremos *todos* los conceptos de nivel inferior estar en las derivadas y *todos* los conceptos de nivel superior estar en la clase base.

Por ejemplo, constantes, variables o funciones de utilidad que pertenecen solo al la implementación no debe estar presente en la clase base. La clase base no debe saber nada hablando de ellos.

Esta regla también se aplica a los archivos, componentes y módulos de origen. Buen software El diseño requiere que separemos conceptos en diferentes niveles y los coloquemos en diferentes contenedores. A veces, estos contenedores son clases base o derivados y, a veces, son archivos, módulos o componentes de origen. Cualquiera que sea el caso, la separación necesita Estar Completo. No queremos que se mezclen conceptos de niveles inferiores y superiores.

Considere el siguiente código:

```
Pila de interfaz pública {
    Object pop () lanza EmptyException;
    void push (Object o) lanza FullException;
    double percentFull ();
```

4. [GOF].
5. [GOF].

Página 322

```
    class EmptyException extiende la excepción {}
    class FullException extiende Exception {}
}
```

La función percentFull está en el nivel de abstracción incorrecto. A pesar de que hay muchas implementaciones de Stack donde el concepto de *plenitud* es razonable, hay otras implementaciones que simplemente *no podían saber* qué tan completas están. Entonces la función sería mejor ubicado en una interfaz derivada como BoundedStack .

Quizás esté pensando que la implementación podría devolver cero si la pila eran ilimitados. El problema con eso es que ninguna pila es realmente ilimitada. No puedes realmente previene una excepción OutOfMemoryException comprobando

```
stack.percentFull () <50.0.
```

Implementar la función para devolver 0 sería mentir.

El punto es que no puedes mentir o fingir para salir de una abstracción fuera de lugar. Yo asi- Hacer abstracciones es una de las cosas más difíciles que hacen los desarrolladores de software, y no hay solución rápida cuando se equivoca.

G7: *Clases base en función de sus derivadas*

La razón más común para dividir conceptos en clases base y derivada es tan que los conceptos de clase base de nivel superior pueden ser independientes de la derivada de nivel inferior conceptos de clase. Por lo tanto, cuando vemos clases base que mencionan los nombres de sus derivadas tivas, sospechamos que hay un problema. En general, las clases base no deben saber nada sobre sus derivados.

Hay excepciones a esta regla, por supuesto. A veces, el número de derivadas es estrictamente fijo, y la clase base tiene un código que selecciona entre las derivadas. Vemos esto un mucho en implementaciones de máquinas de estados finitos. Sin embargo, en ese caso las derivadas y la base class están fuertemente acopladas y siempre se implementan juntas en el mismo archivo jar. En general En caso de que queramos poder implementar derivados y bases en diferentes archivos jar.

Implementar derivados y bases en diferentes archivos jar y asegurarse de que los archivos jar base no saber nada sobre el contenido de los archivos jar derivados nos permite implementar nuestros sistemas en componentes discretos e independientes. Cuando se modifican dichos componentes, pueden volver a implementar sin tener que volver a implementar los componentes base. Esto significa que el El impacto de un cambio se reduce en gran medida y el mantenimiento de los sistemas en el campo se hace mucho más simple.

G8: *demasiada información*

Los módulos bien definidos tienen interfaces muy pequeñas que le permiten hacer mucho con poco. Los módulos mal definidos tienen interfaces amplias y profundas que lo obligan a utilizar muchos gestos para hacer cosas simples. Una interfaz bien definida no ofrece muchas funciones. ciones de las que depender, por lo que el acoplamiento es bajo. Una interfaz mal definida proporciona muchas funciones ciones que debe llamar, por lo que el acoplamiento es alto.

Página 323

Los buenos desarrolladores de software aprenden a limitar lo que exponen en las interfaces de sus clases y módulos. Cuantos menos métodos tenga una clase, mejor. Cuantas menos variables haya una función conocimiento, mejor. Cuantas menos variables de instancia tenga una clase, mejor.

Oculta tus datos. Oculte sus funciones de utilidad. Oculta tus constantes y tus temporales. No cree clases con muchos métodos o muchas variables de instancia. No cree muchos variables y funciones protegidas para sus subclases. Concéntrese en mantener las interfaces muy apretado y muy pequeño. Ayude a mantener el acoplamiento bajo limitando la información.

G9: *Código muerto*

El código muerto es código que no se ejecuta. Lo encuentra en el cuerpo de una declaración if que verifica por una condición que no puede suceder. Lo encuentras en el bloque de captura de un intento que nunca se lanza . Lo encuentra en métodos de pequeña utilidad que nunca se llaman o en condiciones de cambio / caso que nunca ocurrirá.

El problema con el código muerto es que después de un tiempo comienza a oler. Cuanto más viejo es, el más fuerte y agrio se vuelve el olor. Esto se debe a que el código muerto no es completamente actualizado cuando cambian los diseños. Todavía se *compila* , pero no sigue las convenciones más recientes o reglas. Fue escrito en un momento en que el sistema era *diferente* . Cuando encuentre un código muerto, hágalo Lo correcto. Dale un entierro decente. Bórralo del sistema.

G10: *Separación vertical*

Las variables y la función deben definirse cerca de donde se utilizan. Variables locales debe declararse justo por encima de su primer uso y debe tener un alcance vertical pequeño. Nosotros no quiero que las variables locales se declaren a cientos de líneas distantes de sus usos.

Las funciones privadas deben definirse justo debajo de su primer uso. Funciones privadas pertenecen al ámbito de toda la clase, pero aún nos gustaría limitar la distancia vertical entre las invocaciones y las definiciones. Encontrar una función privada debería ser solo una cuestión de escanear hacia abajo desde el primer uso.

G11: *Inconsistencia*

Si hace algo de cierta manera, haga todas las cosas similares de la misma manera. Esto se remonta al principio de la menor sorpresa. Tenga cuidado con las convenciones que elija, y una vez elegido, tenga cuidado de continuar siguiéndolos.

Si dentro de una función en particular usa una variable llamada respuesta para contener una HttpServletResponse , luego use el mismo nombre de variable consistentemente en las otras funciones que utilizan objetos HttpServletResponse . Si nombra un método processVerificationRequest , luego use un nombre similar, como processDeletionRequest , para los métodos que procesan otro tipo de solicitudes.

Una consistencia simple como esta, cuando se aplica de manera confiable, puede hacer que el código sea mucho más fácil de leer y modificar.

Página 324

G12: *Desorden*

¿De qué sirve un constructor predeterminado sin implementación? Todo lo que sirve para hacer es desorden hasta el código con artefactos sin sentido. Variables que no se utilizan, funciones que nunca llamados, comentarios que no agregan información, etc. Todas estas cosas son desorden y debería ser removido. Mantenga sus archivos de origen limpios, bien organizados y libres de desorden.

G13: *Acoplamiento artificial*

Las cosas que no dependen unas de otras no deben acoplarse artificialmente. Por ejemplo, Las enumeraciones generales no deben estar contenidas dentro de clases más específicas porque esto obliga a toda la aplicación para conocer estas clases más específicas. Lo mismo ocurre con el general funciones estáticas de propósito que se declaran en clases específicas.

En general, un acoplamiento artificial es un acoplamiento entre dos módulos que no sirve propósito directo. Es el resultado de poner una variable, constante o función en un ubicación conveniente, aunque inapropiada. Esto es perezoso y descuidado.

Tómese el tiempo para averiguar dónde deberían estar las funciones, constantes y variables declarado. No se limite a tirarlos en el lugar más conveniente a mano y luego dejarlos allí.

G14: *Feature Envy*

Este es uno de los olores de código de Martin Fowler. [6] Los métodos de una clase deberían estar interesados en las variables y funciones de la clase a la que pertenecen, y no las variables y funciones de otras clases. Cuando un método usa descriptores de acceso y mutadores de algún otro objeto para manipular los datos dentro de ese objeto, entonces *envidia* el alcance de la clase de ese otro objeto. Desea estar dentro de esa otra clase para poder tener acceso directo a la variables que está manipulando. Por ejemplo:

```
public class HourlyPayCalculator {
    public Money calculateWeeklyPay (HourlyEmployee e) {
        int TenthRate = e.getTenthRate (). getPennies ();
        int tenthsWorked = e.getTenthsWorked ();
        int tiempo recto = Math.min (400, décimas de trabajo);
        int overTime = Math.max (0, tenthsWorked - straightTime);
        int StraightPay = straightTime * tenthRate;
        int overtimePay = (int) Math.round (overTime * tenthRate * 1.5);
        devolver dinero nuevo (pago directo + pago por tiempo extra);
    }
}
```

El método calculateWeeklyPay llega al objeto HourlyEmployee para obtener los datos que opera. El método calculateWeeklyPay *envidia* el alcance de HourlyEmployee . Eso "Desea" que pueda estar dentro de HourlyEmployee .

6. [Refactorización].

Página 325

En igualdad de condiciones, queremos eliminar Feature Envy porque expone los aspectos internos de una clase a otra. A veces, sin embargo, Feature Envy es un mal necesario. Considera el siguiente:

```
public class HourlyEmployeeReport {
    empleado privado HourlyEmployee;

    Public HourlyEmployeeReport (HourlyEmployee e) {
        this.employee = e;
    }

    String reportHours () {
        return String.format (
            "Nombre:% s \ tHoras:% d.% 1d \ n",
            empleado.getName (),
            employee.getTenthsWorked () / 10,
            employee.getTenthsWorked ()% 10);
    }
}
```

Claramente, el método reportHours envidia a la clase HourlyEmployee . Por otro lado, nosotros no quiero que HourlyEmployee tenga que conocer el formato del informe. Moviendo eso hacia adelante mat en la clase HourlyEmployee violaría varios principios de orientación a objetos diseño. [7] Sería par EmpleadoPorHoras al formato del informe, exponerlo a cambios en ese formato.

G15: *Argumentos del selector*

Difícilmente hay algo más abominable que un falso argumento al final de un Llamada de función. ¿Qué significa? ¿Qué cambiaría si fuera verdad ? No solo es el propósito de un argumento selector difícil de recordar, cada argumento selector combina muchas funciones en una. Los argumentos del selector son solo una forma perezosa de evitar dividir una gran función en varias funciones más pequeñas. Considerar:

```
public int calculateWeeklyPay (horas extra booleanas) {
    int tenthRate = getTenthRate ();
    int tenthsWorked = getTenthsWorked ();
    int tiempo recto = Math.min (400, décimas de trabajo);
    int overTime = Math.max (0, tenthsWorked - straightTime);
    int StraightPay = straightTime * tenthRate;
```

```
    doble tiempo extraRate = tiempo extra? 1.5: 1.0 * décimo;
    int sobretiempoPay = (int) Math.round (sobretiempo * sobretiempoRate);
    volver straightPay + tiempo extraPay;
}
```

Llamas a esta función con un verdadero si las horas extra se pagan como tiempo y medio, y con un falso si las horas extraordinarias se pagan como tiempo normal. Ya es bastante malo que tengas que recordar lo que calcularWeeklyPay (falso) significa cada vez que te encuentras con él. Pero el

7. Específicamente, el Principio de Responsabilidad Única, el Principio de Abierto Cerrado y el Principio de Cierre Común. Ver [PPP].

Página 326

La verdadera vergüenza de una función como esta es que el autor perdió la oportunidad de escribir el siguiente:

```
public int straightPay () {
    return getTenthsWorked () * getTenthRate ();
}

public int overTimePay () {
    int overTimeTenths = Math.max (0, getTenthsWorked () - 400);
    int overTimePay = overTimeBonus (overTimeTenths);
    return straightPay () + overTimePay;
}

private int overTimeBonus (int overTimeTenths) {
    bono doble = 0.5 * getTenthRate () * overTimeTenths;
    return (int) Math.round (bonificación);
}
```

Por supuesto, los selectores no necesitan ser booleanos . Pueden ser enumeraciones, números enteros o cualquier otro tipo de argumento que se utiliza para seleccionar el comportamiento de la función. En general es mejor tienen muchas funciones que pasar un código a una función para seleccionar el comportamiento.

G16: *Intención oculta*

Queremos que el código sea lo más expresivo posible. Expresiones continuas, notación húngara, y los números mágicos oscurecen la intención del autor. Por ejemplo, aquí está el overTimePay funcionar como podría haber aparecido:

```
public int m_otCalc () {
    devolver iThsWkd * iThsRte +
        (int) Math.round (0.5 * iThsRte *
            Math.max (0, iThsWkd - 400)
        );
}
```

Por pequeño y denso que pueda parecer, también es prácticamente impenetrable. Vale la pena tomar tiempo para hacer visible la intención de nuestro código a nuestros lectores.

G17: *Responsabilidad fuera de lugar*

Una de las decisiones más importantes que puede tomar un desarrollador de software es dónde colocar el código. Por ejemplo, ¿adónde debería ir la constante PI ? ¿Debería ser en la clase de matemáticas ? Tal vez sea pertenece a la clase de trigonometría ? ¿O quizás en la clase Circle ?

Aquí entra en juego el principio de la mínima sorpresa. El código debe colocarse donde un el lector, naturalmente, esperaría que lo fuera. La constante PI debe ir donde las funciones trigonométricas se declaran. La constante OVERTIME_RATE debe declararse en HourlyPay-Clase de calculadora .

A veces nos volvemos "inteligentes" acerca de dónde colocar ciertas funciones. Lo pondremos en un función que es conveniente para nosotros, pero no necesariamente intuitiva para el lector. Por ejemplo, quizás necesitemos imprimir un informe con el total de horas que trabajó un empleado. Nosotros

Página 327

podríamos resumir esas horas en el código que imprime el informe, o podríamos intentar mantener una ejecución sin total en el código que acepta tarjetas de tiempo.

Una forma de tomar esta decisión es mirar los nombres de las funciones. Digamos que nuestro módulo de informes tiene una función denominada getTotalHours . Digamos también que el módulo que acepta tarjetas de tiempo tiene una función saveTimeCard . ¿Cuál de estas dos funciones, por su nombre, implica que calcula el total? La respuesta debería ser obvia.

Claramente, a veces hay razones de rendimiento por las que se debe calcular el total. ya que se aceptan tarjetas de tiempo en lugar de cuando se imprime el informe. Eso está bien, pero los nombres de las funciones debe reflejar esto. Por ejemplo, debería haber un computeRunning-Función TotalOfHours en el módulo de tarjeta de tiempo .

G18: *Estática inapropiada*

Math.max (double a, double b) es un buen método estático. No opera en un solo ejemplo; de hecho, sería una tontería tener que decir new Math (). max (a, b) o incluso a.max (b) . Todos los datos que usa max provienen de sus dos argumentos, y no de ninguna "propiedad" objeto. Más concretamente , casi *no hay posibilidad de* que queramos que Math.max sea polimórfico.

A veces, sin embargo, escribimos funciones estáticas que no deberían ser estáticas. Por ejemplo, considerar:

```
HourlyPayCalculator.calculatePay (empleado, sobretiempoRate).
```

Nuevamente, esto parece una función estática razonable . No opera en ningún objeto y obtiene todos sus datos de sus argumentos. Sin embargo, existe una posibilidad razonable de que queremos que esta función sea polimórfica. Es posible que deseemos implementar varios algoritmos para calcular el pago por hora, por ejemplo, OvertimeHourlyPayCalculator y StraightTimeHourlyPayCalculator . Entonces, en este caso, la función no debería ser estática. Eso debe ser una función miembro no estática de Empleado .

En general, debería preferir métodos no estáticos a métodos estáticos. En caso de duda, hacer que la función sea no estática. Si realmente desea que una función sea estática, asegúrese de que haya no hay posibilidad de que quieras que se comporte polimórficamente.

G19: *Utilizar variables explicativas*

Kent Beck escribió sobre esto en su gran libro *Smalltalk Best Practice Patterns* [8] and again más recientemente en su igualmente grandioso libro *Implementation Patterns* . [9] Uno de los más poderosos Una forma completa de hacer que un programa sea legible es dividir los cálculos en valores intermedios. ues que se mantienen en variables con nombres significativos.

8. [Beck97], pág. 108.
9. [Beck07].

Página 328

Considere este ejemplo de FitNesse:

```
Coincidencia de coincidencia = headerPattern.matcher (línea);
si (match.find ())
{
    String key = match.group (1);
    Valor de cadena = match.group (2);
    headers.put (key.toLowerCase (), value);
}
```

El simple uso de variables explicativas deja en claro que el primer grupo emparejado es la *clave* y el segundo grupo coincidente es el *valor* .

Es difícil exagerar con esto. Por lo general, es mejor tener más variables explicativas que menos. Eso Es notable cómo un módulo opaco puede volverse transparente repentinamente simplemente al romperse convertir los cálculos en valores intermedios bien nombrados.

G20: *Los nombres de las funciones deben decir lo que hacen*

Mira este código:

```
Fecha newDate = date.add (5);
```

¿Esperaría que esto agregara cinco días a la fecha? ¿O son semanas u horas? Es la `fecha` instancia cambió o la función simplemente devuelve una nueva `fecha` sin cambiar la anterior? *No se puede saber a partir de la llamada qué hace la función* .

Si la función agrega cinco días a la fecha y cambia la fecha, entonces debería llamarse `addDaysTo` o `increaseByDays` . Si, por otro lado, la función devuelve una nueva fecha que es cinco días después pero no cambia la instancia de fecha, debería llamarse `daysLater` o `díasDesde` .

Si tiene que mirar la implementación (o documentación) de la función para saber qué hace, entonces debería trabajar para encontrar un mejor nombre o reorganizar la funcionalidad para que se puede colocar en funciones con mejores nombres.

G21: *Comprender el algoritmo*

Se escribe mucho código muy divertido porque la gente no se toma el tiempo para entender algoritmo. Consiguen que algo funcione conectando suficientes declaraciones `if` y banderas, sin detenerse realmente a considerar lo que realmente está sucediendo.

La programación es a menudo una exploración. Usted *piensa* que conoce el algoritmo adecuado para algo, pero luego terminas jugando con él, pinchando y pinchando, hasta que lo consigues trabajar." Como sabés que funciona"? Porque pasa los casos de prueba que pueda imaginar.

No hay nada de malo en este enfoque. De hecho, a menudo es la única forma de obtener una función para hacer lo que cree que debería. Sin embargo, no es suficiente dejar la cotización. marcas alrededor de la palabra "trabajo".

Antes de considerar que ha terminado con una función, asegúrese de *comprender* cómo funciona. No es suficiente que pase todas las pruebas. Debes *saber* [10] que el la solución es correcta.

A menudo, la mejor manera de obtener este conocimiento y comprensión es refactorizar la función

en algo que es tan limpio y expresivo que es *obvio* cómo funciona.

G22: Hacer que las dependencias lógicas sean físicas

Si un módulo depende de otro, esa dependencia debe ser física, no solo lógica.
El módulo dependiente no debe hacer suposiciones (en otras palabras, dependencias lógicas
cies) sobre el módulo del que depende. Más bien, debería pedir explícitamente ese módulo para todos
la información de la que depende.

Por ejemplo, imagine que está escribiendo una función que imprime un informe de texto sin formato de horas trabajadas por los empleados. Una clase llamada HourlyReporter reúne todos los datos en un conveniente y luego lo pasa a HourlyReportFormatter para imprimirlo. (Consulte el Listado 17-1).

Listado 17-1

HourlyReporter.java

```
public class HourlyReporter {
    formateador privado HourlyReportFormatter;
    página de lista privada <LineItem>;
    privado final int PAGE_SIZE = 55;

    public HourlyReporter (formateador HourlyReportFormatter) {
        this.formatter = formateador;
        page = new ArrayList <LineItem> ();
    }

    public void generateReport (Lista de empleados <HourlyEmployee>) {
        para (Empleado por hora e: empleados) {
            addLineItemToPage (e);
            if (page.size () == PAGE_SIZE)
                printAndClearItemList ();
        }
        si (page.size ()> 0)
            printAndClearItemList ();
    }

    private void printAndClearItemList () {
        formatter.format (página);
        page.clear ();
    }

    private void addLineItemToPage (HourlyEmployee e) {
        Elemento de elemento de línea = nuevo elemento de línea ();
        item.name = e.getName ();
        item.hours = e.getTenthsWorked () / 10;
```

10. Existe una diferencia entre saber cómo funciona el código y saber si el algoritmo hará el trabajo requerido.
No estar seguro de que un algoritmo sea apropiado es a menudo una realidad. No estar seguro de lo que hace su código es solo pereza.

Página 330

Listado 17-1 (continuación)

HourlyReporter.java

```
        item.tenths = e.getTenthsWorked ()% 10;
        page.add (elemento);
    }

    public class LineItem {
        nombre de cadena pública;
        horas públicas int;
        public int décimos;
    }
}
```

Este código tiene una dependencia lógica que no se ha fisicalizado. ¿Puedes distinguirlo? Eso es la constante PAGE_SIZE . ¿Por qué debería conocer el HourlyReporter el tamaño de la página? Página el tamaño debe ser responsabilidad del HourlyReportFormatter .

El hecho de que PAGE_SIZE se declare en HourlyReporter representa un error responsabilidad [G17] que hace que HourlyReporter asuma que sabe cuál es el tamaño de la página

debería ser. Tal suposición es una dependencia lógica. HourlyReporter depende de la hecho que HourlyReportFormatter puede manejar tamaños de página de 55. Si alguna implementación de HourlyReportFormatter no podría lidiar con tales tamaños, entonces habría un error.

Podemos fisicalizar esta dependencia creando un nuevo método en HourlyReport-Formateador llamado getMaxPageSize () . HourlyReporter llamará a esa función en lugar de utilizando la constante PAGE_SIZE .

G23: *Prefiere polimorfismo a If / Else o Switch / Case*

Esto puede parecer una sugerencia extraña dado el tema del Capítulo 6. Después de todo, en ese capítulo I señalar que las declaraciones de cambio son probablemente apropiadas en las partes del sistema donde es más probable agregar nuevas funciones que agregar nuevos tipos.

Primero, la mayoría de la gente usa declaraciones de cambio porque es la solución obvia de fuerza bruta, no porque sea la solución adecuada para la situación. Así que esta heurística está aquí para recordarnos considere el polimorfismo antes de usar un interruptor.

En segundo lugar, los casos en los que las funciones son más volátiles que los tipos son relativamente raros. Entonces *cada* declaración de cambio debe ser sospechosa.

Utilizo la siguiente regla "O NE S WITCH ": *No puede haber más de un estado de interruptor: ment para un tipo dado de selección. Los casos en esa declaración de cambio deben crear polimor- objetos phic que toman el lugar de otras sentencias de cambio de este tipo en el resto del sistema.*

G24: *Siga las convenciones estándar*

Cada equipo debe seguir un estándar de codificación basado en normas comunes de la industria. Este bacalao ing estándar debe especificar cosas como dónde declarar las variables de instancia; como nombrar clases, métodos y variables; dónde poner los aparatos ortopédicos; y así. El equipo no debería necesitar un documento para describir estas convenciones porque su código proporciona los ejemplos.

Página 331

Todos los miembros del equipo deben seguir estas convenciones. Esto significa que cada equipo El miembro debe ser lo suficientemente maduro para darse cuenta de que no importa un ápice dónde ponga su aparatos ortopédicos siempre que estén de acuerdo en dónde colocarlos.

Si desea saber qué convenciones sigo, las verá en el refactorizado en el Listado B-7 en la página 394, hasta el Listado B-14.

G25: *Reemplazo de números mágicos con constantes nombradas*

Esta es probablemente una de las reglas más antiguas en el desarrollo de software. Recuerdo haberlo leído en el finales de los sesenta en los manuales introductorios de COBOL, FORTRAN y PL / 1. En general es un mal idea de tener números sin procesar en su código. Debería ocultarlos detrás de constantes bien nombradas.

Por ejemplo, el número 86,400 debe ocultarse detrás de la constante SECONDS_PER_DAY . Si está imprimiendo 55 líneas por página, entonces la constante 55 debe estar oculta. den detrás de la constante LINES_PER_PAGE .

Algunas constantes son tan fáciles de reconocer que no siempre necesitan una constante con nombre esconderse detrás siempre que se utilicen junto con un código muy autoexplicativo. Para ejemplo:

```
doble milesWalked = feetWalked / 5280.0;
int dailyPay = hourlyRate * 8;
circunferencia doble = radio * Math.PI * 2;
```

¿Realmente necesitamos las constantes FEET_PER_MILE , WORK_HOURS_PER_DAY y TWO en el ejemplos anteriores? Claramente, el último caso es absurdo. Hay algunas fórmulas en las que los stants simplemente se escriben mejor como números sin procesar. Podrías objetar sobre el Caso WORK_HOURS_PER_DAY porque las leyes o convenciones pueden cambiar. En el otro Por otro lado, esa fórmula se lee tan bien con el 8 en ella que sería reacio a agregar 17 extra personajes a carga de los lectores. Y en el caso FEET_PER_MILE , el número 5280 es tan muy conocida y una constante tan única que los lectores la reconocerían incluso si se mantuviera

solo en una página sin contexto a su alrededor.

Constantes como 3.141592653589793 también son muy conocidas y fácilmente reconocibles. Sin embargo, la posibilidad de error es demasiado grande para dejarlos sin formato. Cada vez que alguien ve 3.1415927535890793, saben que es *p,* por lo que no lo examinan. (Tuviste detectar el error de un solo dígito?) Tampoco queremos que las personas utilicen 3.14, 3.14159, 3.142, etc. adelante. Por lo tanto, es bueno que Math.PI ya se haya definido para nosotros.

El término "Número mágico" no se aplica solo a los números. Se aplica a cualquier token. que tiene un valor que no se describe a sí mismo. Por ejemplo:

```
asertEquals (7777, Employee.find ("John Doe"). employeeNumber ());
```

Hay dos números mágicos en esta afirmación. El primero es obviamente 7777, aunque lo que podría significar no es obvio. El segundo número mágico es "John Doe", y nuevamente el la intención no está clara.

Resulta que "John Doe" es el nombre del empleado n. ° 7777 en una conocida prueba de datos: base creada por nuestro equipo. Todos en el equipo saben que cuando te conectas a este

Página 332

base de datos, tendrá varios empleados ya preparados con valores bien conocidos y atributos. También resulta que "John Doe" representa al único empleado por hora en esa base de datos de prueba. Entonces esta prueba realmente debería leer:

```
asertEquals (
   HOURLY_EMPLOYEE_ID,
   Employee.find (HOURLY_EMPLOYEE_NAME) .employeeNumber ());
```

G26: *Sea preciso*

Esperar que la primera coincidencia sea la *única* coincidencia con una consulta probablemente sea ingenuo. Usando flotante números de puntos para representar la moneda es casi criminal. Evitar bloqueos y / o transacciones administración porque no cree que la actualización simultánea sea, en el mejor de los casos, perezosa. Declarando una variable para ser una ArrayList cuando vence una lista es demasiado restrictiva. Haciendo todas las ables protegidas por defecto no es lo suficientemente restrictivo.

Cuando tome una decisión en su código, asegúrese de hacerlo con *precisión* . Saber porque lo ha logrado y cómo manejará las excepciones. No seas perezoso con el pre decisión de sus decisiones. Si decide llamar a una función que podría devolver un valor nulo , asegúrese de comprueba si es nulo . Si consulta lo que cree que es el único registro en la base de datos, asegúrese de que su código verifique para asegurarse de que no haya otros. Si necesita lidiar con la corriente rencia, use números enteros ni y maneje el redondeo de manera apropiada. Si existe la posibilidad de actualización simultánea, asegúrese de implementar algún tipo de mecanismo de bloqueo.

Las ambigüedades y la imprecisión en el código son el resultado de desacuerdos o de la pereza. En cualquier caso, deberían eliminarse.

G27: *Estructura sobre Convención*

Haga cumplir las decisiones de diseño con estructura sobre convención. Las convenciones de nomenclatura son buenas, pero son inferiores a las estructuras que obligan al cumplimiento. Por ejemplo, cambie / cases con las enumeraciones bien nombradas son inferiores a las clases base con métodos abstractos. Nadie es forzado a implementar la instrucción switch / case de la misma manera cada vez; pero la base las clases hacen cumplir que las clases concretas tienen implementados todos los métodos abstractos.

G28: *Encapsular condicionales*

La lógica booleana es bastante difícil de entender sin tener que verla en el contexto de un if o declaración while . Extrae funciones que expliquen la intención del condicional.

Por ejemplo:

```
if (shouldBeDeleted (timer))
```

es preferible a

```
if (timer.hasExpired () &&! timer.isRecurrent ())
```

11. O mejor aún, una clase Money que usa números enteros.

Página 333

G29: *Evite los condicionales negativos*

Los aspectos negativos son un poco más difíciles de entender que los positivos. Entonces, cuando sea posible, condicione als deben expresarse como positivos. Por ejemplo:

```
si (buffer.shouldCompact ())
```

es preferible a

```
si (! buffer.shouldNotCompact ())
```

G30: Las *funciones deben hacer una cosa*

A menudo es tentador crear funciones que tengan múltiples secciones que realicen una serie de operaciones. Las funciones de este tipo hacen más de *una cosa* y deben convertirse en muchas funciones más pequeñas, cada una de las cuales hace *una cosa* .

Por ejemplo:

```
pago público nulo () {
    para (Empleado e: empleados) {
        if (e.isPayday ()) {
            Pago de dinero = e.calculatePay ();
            e.deliverPay (pagar);
        }
    }
}
```

Este fragmento de código hace tres cosas. Recorre a todos los empleados y comprueba si a cada empleado se le debe pagar, y luego le paga al empleado. Este código sería mejor Escrito como:

```
pago público nulo () {
    para (Empleado e: empleados)
        pagar si es necesario (e);
}

pago nulo privado si es necesario (empleado e) {
    si (e.isPayday ())
        calculateAndDeliverPay (e);
}

private void calculateAndDeliverPay (Empleado e) {
    Pago de dinero = e.calculatePay ();
    e.deliverPay (pagar);
}
```

Cada una de estas funciones hace una cosa. (Consulte "Haga una cosa" en la página 35.)

G31: *Acoplamientos temporales ocultos*

Los acoplamientos temporales suelen ser necesarios, pero no debe ocultarlos. Estructura los argumentos de sus funciones de modo que el orden en el que deben llamarse es obvio ous. Considera lo siguiente:

Página 334


```
public class MoogDiver {
    Gradiente degradado;
    Lista de splines <Spline>;

    inmersión en vacío público (motivo de cadena) {
        saturateGradient ();
        reticulateSplines ();
        diveForMoog (razón);
    }
    ...
}
```

El orden de las tres funciones es importante. Debes saturar el degradado antes que puede reticular las splines, y solo entonces puede bucear para el moog. Desafortunadamente, el código no refuerza este acoplamiento temporal. Otro programador podría llamar reticulate-Estrías antes saturateGradient fue llamado, lo que lleva a una UnsaturatedGradientException . Una mejor solución es:

```
public class MoogDiver {
    Gradiente degradado;
    Lista de splines <Spline>;

    inmersión en vacío público (motivo de cadena) {
        Gradiente gradiente = saturateGradient ();
        List <Spline> splines = reticulateSplines (degradado);
        diveForMoog (splines, razón);
    }
    ...
}
```

Esto expone el acoplamiento temporal mediante la creación de una brigada de cubos. Cada función produce una resultado que necesita la siguiente función, por lo que no hay una forma razonable de llamarlos fuera de servicio.

Puede quejarse de que esto aumenta la complejidad de las funciones, y sería derecho. Pero esa complejidad sintáctica adicional expone la verdadera complejidad temporal de la situación.

Tenga en cuenta que dejé las variables de instancia en su lugar. Supongo que son necesarios por primera vez. métodos alternativos en la clase. Aun así, quiero que los argumentos estén en su lugar para hacer que el temporal acoplamiento explícito.

G32: *No seas arbitrario*

Tenga una razón para la forma en que estructura su código y asegúrese de que la razón sea comunicada cated por la estructura del código. Si una estructura parece arbitraria, otros se sentirán empoderados para cambiarlo. Si una estructura aparece de manera consistente en todo el sistema, otros la usarán. y preservar la convención. Por ejemplo, recientemente fusioné cambios en FitNesse y descubrió que uno de nuestros confirmadores había hecho esto:

```
La clase pública AliasLinkWidget extiende ParentWidget
{
    clase pública estática VariableExpandingWidgetRoot {
        ...

    ...
}
```


Página 335

El problema con esto era que VariableExpandingWidgetRoot no tenía necesidad de ser dentro del alcance de AliasLinkWidget . Además, otras clases no relacionadas hicieron uso de AliasLinkWidget.VariableExpandingWidgetRoot . Estas clases no necesitaban saber sobre AliasLinkWidget .

Quizás el programador había colocado el VariableExpandingWidgetRoot en AliasWidget por conveniencia, o tal vez pensó que realmente necesitaba ser dentro de AliasWidget . Cualquiera sea la razón, el resultado terminó siendo arbitrario. Pub- Las clases lic que no son utilidades de alguna otra clase no deben tener el alcance dentro de otra clase. La convención es hacerlos públicos en el nivel superior de su paquete.

G33: *Condiciones de* contorno *encapsuladas*

Las condiciones de contorno son difíciles de seguir. Ponga el procesamiento para ellos en un solo lugar. No dejes que se filtren por todo el código. No queremos enjambres de +1 sy -1 s dispersos aquí y ahí. Considere este simple ejemplo de FIT:

```
if (nivel + 1 <tags.length)
{
    partes = nuevo Parse (cuerpo, etiquetas, nivel + 1, desplazamiento + endTag);
    cuerpo = nulo;
}
```

Observe que el nivel + 1 aparece dos veces. Esta es una condicin de frontera que debe encapsularse dentro de una variable llamada algo así como nextLevel .

```
int nextLevel = nivel + 1;
if (nextLevel <tags.length)
{
    parts = new Parse (cuerpo, etiquetas, nextLevel, offset + endTag);
    cuerpo = nulo;
}
```

G34: Las *funciones deben descender solo un nivel de abstracción*

Todas las declaraciones dentro de una función deben escribirse en el mismo nivel de abstracción, que debe estar un nivel por debajo de la operación descrita por el nombre de la función. Esto puede ser la más difícil de interpretar y seguir de estas heurísticas. Aunque la idea es simple lo suficiente, los humanos son demasiado buenos para mezclar a la perfección niveles de abstracción. Considerar, por ejemplo, el siguiente código tomado de FitNesse:

```
public String render () arroja una excepción
{
    StringBuffer html = new StringBuffer ("<h");
    si (tamaño> 0)
        html.append ("tamaño = \" "). append (tamaño + 1) .append (" \ "");
    html.append (">");

    return html.toString ();
}
```


Página 336

Un momento de estudio y podrás ver lo que está pasando. Esta función construye el HTML etiqueta que dibuja una regla horizontal en la página. La altura de esa regla se especifica en el variable de tamaño .

Ahora mira de nuevo. Este método mezcla al menos dos niveles de abstracción. El primero es la noción de que una regla horizontal tiene un tamaño. El segundo es la sintaxis de la propia etiqueta HR . Este código viene de la HruleWidget módulo en FitNesse. Este módulo detecta una fila de cuatro o más guiones y lo convierte en la etiqueta HR adecuada. Cuantos más guiones, el mayor el tamaño.

Refactoré este fragmento de código de la siguiente manera. Tenga en cuenta que cambié el nombre del campo de tamaño para reflejar su verdadero propósito. Tenía el número de guiones adicionales.

```
public String render () arroja una excepción
{
    HtmlTag hr = new HtmlTag ("hr");
    if (extraDashes> 0)
        hr.addAttribute ("tamaño", hrSize (extraDashes));
    return hr.html ();
}

Private String hrSize (altura int)
{
    int hrSize = altura + 1;
    return String.format ("% d", hrSize);
}
```

Este cambio separa muy bien los dos niveles de abstracción. La función de renderizado simplemente estructura una etiqueta de recursos humanos, sin tener que saber nada sobre la sintaxis HTML de esa etiqueta. El módulo HtmlTag se encarga de todos los desagradables problemas de sintaxis.

De hecho, al hacer este cambio, detecté un error sutil. El código original no puso la barra de cierre en la etiqueta HR, como lo tendría el estándar XHTML. (En otras palabras, emitido <hr> en lugar de <hr /> .) El módulo HtmlTag se ha cambiado para ajustarse a XHTML hace mucho tiempo.

Separar los niveles de abstracción es una de las funciones más importantes de la refactorización. ing, y es uno de los más difíciles de hacer bien. Como ejemplo, mire el código a continuación. Esto fue mi primer intento de separar los niveles de abstracción en HruleWidget.render método .

```
public String render () arroja una excepción
{
    HtmlTag hr = new HtmlTag ("hr");
    if (tamaño> 0) {
        hr.addAttribute ("tamaño", "" + (tamaño + 1));
    }
    return hr.html ();
}
```

Mi objetivo, en este punto, era crear la separación necesaria y lograr que las pruebas pasaran. Logré ese objetivo fácilmente, pero el resultado fue una función que *todavía* tenía niveles mixtos de abstracción. En este caso los niveles mixtos fueron la construcción de la etiqueta HR y la

Página 337

interpretación y formato de la variable de tamaño . Esto señala que cuando rompes un funcionar a lo largo de líneas de abstracción, a menudo descubres nuevas líneas de abstracción que fueron oscurecido por la estructura anterior.

G35: Mantenga los datos configurables en niveles altos

Si tiene una constante, como un valor predeterminado o de configuración, que se conoce y se espera a un alto nivel de abstracción, no lo entierres en una función de bajo nivel. Exponerlo como un argumento mención a esa función de bajo nivel llamada desde la función de alto nivel. Considera lo siguiente código de FitNesse:

```
public static void main (String [] args) arroja una excepción
{
    Argumentos argumentos = parseCommandLine (args);
    ...
}

Argumentos de clase pública
{
    Cadena final estática pública DEFAULT_PATH = ".";
    Public static final String DEFAULT_ROOT = "FitNesseRoot";
    public static final int DEFAULT_PORT = 80;
    public static final int DEFAULT_VERSION_DAYS = 14;
    ...
}
```

Los argumentos de la línea de comandos se analizan en la primera línea ejecutable de FitNesse. La

los valores predeterminados de esos argumentos se especifican en la parte superior de la clase Argument . Tu no tengo que buscar en niveles bajos del sistema declaraciones como esta:

```
if (argumentos.port == 0) // usa 80 por defecto
```

Las constantes de configuración residen en un nivel muy alto y son fáciles de cambiar. Consiguen transmitido al resto de la aplicación. Los niveles inferiores de la aplicación no son propietarios los valores de estas constantes.

G36: *Evite la navegación transitiva*

En general, no queremos que un solo módulo sepa mucho sobre sus colaboradores. Más espe- En concreto, si A colabora con B y B colabora con C , no queremos módulos que utilicen Una debe saber sobre C . (Por ejemplo, no queremos a.getB (). GetC (). DoSomething ();.)

A esto a veces se le llama la Ley de Demeter. Los programadores pragmáticos lo llaman "Escribiendo código tímido". [12] En cualquier caso, se trata de asegurarse de que los módulos sepan sólo sobre sus colaboradores inmediatos y no conocen el mapa de navegación del conjunto sistema.

Si muchos módulos usaran alguna forma de la declaración a.getB (). GetC () , entonces sería difícil de cambiar el diseño y la arquitectura de interponer un Q entre B y C . Tu habias

12. [PRAG], pág. 138.

Página 338

tienes que encontrar todas las instancias de a.getB (). getC () y convertirlas en a.getB (). getQ (). getC () . Así es como las arquitecturas se vuelven rígidas. Demasiados módulos saben demasiado sobre arquitectura.

Más bien queremos que nuestros colaboradores inmediatos ofrezcan todos los servicios que necesitamos. Nosotros no debería tener que vagar por el gráfico de objetos del sistema, buscando el método que quiero llamar. Más bien, simplemente deberíamos poder decir:

```
myCollaborator.doSomething ().
```

Java

J1: *Evite las listas de importación largas mediante el uso de comodines*

Si usa dos o más clases de un paquete, entonces importe todo el paquete con

```
importar paquete. *;
```

Las largas listas de importaciones son abrumadoras para el lector. No queremos desordenar la parte superior de nuestros modulos con 80 lineas de importacion. Más bien queremos que las importaciones sean una declaración concisa. sobre con qué paquetes colaboramos.

Las importaciones específicas son dependencias estrictas, mientras que las importaciones con comodines no lo son. Si tu espe- importar específicamente una clase, entonces esa clase *debe* existir. Pero si importa un paquete con un comodín tarjeta, no es necesario que existan clases particulares. La declaración de importación simplemente agrega el paquete a la ruta de búsqueda al buscar nombres. Así que no se crea una verdadera dependencia por tales importaciones y, por lo tanto, sirven para mantener nuestros módulos menos acoplados.

Hay ocasiones en las que la larga lista de importaciones específicas puede resultar útil. Por ejemplo, si está tratando con código heredado y desea saber qué clases necesita construir simulacros y resguardos para, puede recorrer la lista de importaciones específicas para averiguar el verdadero nombres calificados de todas esas clases y luego coloque los talones apropiados en su lugar. Sin emabargo, este uso para importaciones específicas es muy raro. Además, la mayoría de los IDE modernos le permitirán para convertir las importaciones con comodines en una lista de importaciones específicas con un solo comando. Entonces incluso en el caso de versiones anteriores, es mejor importar comodines.

Las importaciones de comodines a veces pueden causar ambigüedades y conflictos de nombres. Dos clases con el mismo nombre, pero en paquetes diferentes, deberá ser importado específicamente, o en menos específicamente calificado cuando se utiliza. Esto puede ser una molestia, pero es lo suficientemente raro como para usar

Las importaciones con comodines siguen siendo en general mejores que las importaciones específicas.

J2: No heredes constantes

Lo he visto varias veces y siempre me hace una mueca. Un programador pone algunos constantes en una interfaz y luego obtiene acceso a esas constantes heredando esa interfaz cara. Eche un vistazo al siguiente código:

```
public class HourlyEmployee extiende Employee {
  private int tenthsWorked;
  tarifa por hora doble privada;
```


Página 339


```
  public Money calculatePay () {
    int straightTime = Math.min (décimas trabajadas, DIEZ_PER_SEMANA);
    int overTime = tenthsWorked - straightTime;
    devolver dinero nuevo
      hourlyRate * (décimas de trabajo + OVERTIME_RATE * overTime)
    );
  }
  ...
}
```

¿De dónde las constantes TENTHS_PER_WEEK y OVERTIME_RATE vienen? Ellos pueden tener provienen de la clase Empleado ; así que echemos un vistazo a eso:

```
Empleado de clase abstracta pública implementa PayrollConstants {
  public abstract boolean isPayday ();
  Resumen público Money calculatePay ();
  public abstract void deliverPay (Pago de dinero);
}
```

No, ahí no. ¿Pero entonces dónde? Mire de cerca a la clase Empleado . Implementa PayrollConstants .

```
interfaz pública PayrollConstants {
  public static final int TENTHS_PER_WEEK = 400;
  público estático final doble OVERTIME_RATE = 1.5;
}
```

¡Ésta es una práctica espantosa! Las constantes están ocultas en la parte superior de la jerarquía de herencia. ¡Ick! No use la herencia como una forma de burlar las reglas de alcance del lenguaje. Usa una estática importar en su lugar.

```
importar PayrollConstants estático. *;

public class HourlyEmployee extiende Employee {
  private int tenthsWorked;
  tarifa por hora doble privada;

  public Money calculatePay () {
    int straightTime = Math.min (décimas trabajadas, DIEZ_PER_SEMANA);
    int overTime = tenthsWorked - straightTime;
    devolver dinero nuevo
      hourlyRate * (décimas de trabajo + OVERTIME_RATE * overTime)
    );
  }
  ...
}
```

J3: Constantes versus enumeraciones

Ahora que se han agregado las enumeraciones al lenguaje (Java 5), ¡úselas! No sigas usando el viejo truco de int finales estáticos públicos . El significado de int s puede perderse. El significado de enum s no puede, porque pertenecen a una enumeración que se nombra.

Además, estudie la sintaxis de enum s detenidamente. Pueden tener métodos y campos. Esto los convierte en herramientas muy poderosas que permiten mucha más expresión y flexibilidad que int s. Considere esta variación en el código de nómina:

Página 340


```
public class HourlyEmployee extiende Employee {
    private int tenthsWorked;
    Grado HourlyPayGrade;

    public Money calculatePay () {
        int straightTime = Math.min (décimas trabajadas, DIEZ_PER_SEMANA);
        int overTime = tenthsWorked - straightTime;
        devolver dinero nuevo
            grade.rate () * (décimas de trabajo + OVERTIME_RATE * overTime)
        );
    }
    ...
}

public enum HourlyPayGrade {
    APRENDIZ {
        tarifa doble pública () {
            return 1.0;
        }
    },
    LEUTENANT_JOURNEYMAN {
        tarifa doble pública () {
            return 1.2;
        }
    },
    JOURNEYMAN {
        tarifa doble pública () {
            return 1.5;
        }
    },
    MAESTRO {
        tarifa doble pública () {
            return 2.0;
        }
    };

    doble tasa pública abstracta ();
}
```

Nombres

N1: *Elija nombres descriptivos*

No se apresure a elegir un nombre. Asegúrese de que el nombre sea descriptivo. Recuérdalo
Los significados tienden a variar a medida que evoluciona el software, por lo que con frecuencia reevaluar la idoneidad
los nombres que elijas.

Esta no es solo una recomendación para "sentirse bien". Los nombres en el software son el 90 por ciento de
qué hace que el software sea legible. Debe tomarse el tiempo para elegirlos sabiamente y mantener
ellos relevantes. Los nombres son demasiado importantes para tratarlos descuidadamente.

Considere el código a continuación. ¿Qué hace? Si te muestro el código con bien elegido
nombres, tendrá perfecto sentido para ti, pero así es solo una mezcolanza de símbolos
y números mágicos.

Página 341

```
public int x () {
        int q = 0;
        int z = 0;
        para (int kk = 0; kk <10; kk ++) {
            si (l [z] == 10)
            {
                q + = 10 + (l [z + 1] + l [z + 2]);
                z + = 1;
            }
            de lo contrario si (l [z] + l [z + 1] == 10)
            {
                q + = 10 + l [z + 2];
                z + = 2;
            } demás {
                q + = l [z] + l [z + 1];
                z + = 2;
            }
        }
        volver q;
    }
```

Aquí está el código de la forma en que debe escribirse. En realidad, este fragmento es menos completo que el de arriba. Sin embargo, puede inferir inmediatamente lo que está tratando de hacer, y podría muy probablemente escriba las funciones que faltan basándose en ese significado inferido. El número mágico Las bers ya no son mágicas, y la estructura del algoritmo es convincentemente descriptiva.

```
public int score () {
    puntuación int = 0;
    int frame = 0;
    for (int frameNumber = 0; frameNumber <10; frameNumber ++) {
        if (isStrike (frame)) {
            puntuación + = 10 + nextTwoBallsForStrike (marco);
            marco + = 1;
        } else if (isSpare (frame)) {
            puntuación + = 10 + nextBallForSpare (marco);
            marco + = 2;
        } demás {
            puntuación + = twoBallsInFrame (marco);
            marco + = 2;
        }
    }
    puntuación de devolución;
}
```

El poder de los nombres cuidadosamente elegidos es que sobrecargan la estructura del código. con descripción. Esa sobrecarga establece las expectativas de los lectores sobre lo que el otro funciones en el módulo hacen. Puede inferir la implementación de isStrike () mirando el código de arriba. Cuando lea el método isStrike , será "más o menos lo que esperado." [13]

```
private boolean isStrike (int frame) {
    rollos de retorno [marco] == 10;
}
```

13. Vea la cita de Ward Cunningham en la página 11.

Página 342

N2: *Elija nombres en el nivel apropiado de abstracción*

No elija nombres que comuniquen la implementación; elegir nombres que reflejen el nivel de abstracción de la clase o función en la que está trabajando. Esto es difícil de hacer. De nuevo la gente son demasiado buenos para mezclar niveles de abstracciones. Cada vez que haces un pase sobre tu código, es probable que encuentre alguna variable cuyo nombre sea demasiado bajo. Deberías llevar

la oportunidad de cambiar esos nombres cuando los encuentre. Hacer que el código sea legible requiere una dedicación a la mejora continua. Considere la interfaz del módem a continuación:

```
Módem de interfaz pública {
  marcación booleana (String phoneNumber);
  desconexión booleana ();
  envío booleano (char c);
  char recv ();
  String getConnectedPhoneNumber ();
}
```

Al principio esto se ve bien. Todas las funciones parecen apropiadas. De hecho, para muchas aplicaciones ellos son. Pero ahora considere una aplicación en la que algunos módems no están conectados por discado. Más bien, están conectados permanentemente mediante cableado fijo (piense en el módems de cable que brindan acceso a Internet a la mayoría de los hogares en la actualidad). Quizás algunos son conectado enviando un número de puerto a un conmutador a través de una conexión USB. Claramente la noción de números de teléfono tiene un nivel de abstracción incorrecto. Una mejor estrategia de nomenclatura para esto El escenario podría ser:

```
Módem de interfaz pública {
  conexión booleana (String connectionLocator);
  desconexión booleana ();
  envío booleano (char c);
  char recv ();
  String getConnectedLocator ();
}
```

Ahora los nombres no se comprometen con los números de teléfono. Todavía se pueden usar para números de teléfono, o podrían usarse para cualquier otro tipo de estrategia de conexión.

N3: *Utilice la nomenclatura estándar siempre que sea posible*

Los nombres son más fáciles de entender si se basan en convenciones o usos existentes. Para examen- Por ejemplo, si está usando el patrón D ECORATOR , debe usar la palabra Decorator en los nombres de las clases de decoración. Por ejemplo, AutoHangupModemDecorator podría ser el nombre de un clase que decora un módem con la capacidad de colgar automáticamente al final de una sesión.

Los patrones son solo un tipo de estándar. En Java, por ejemplo, las funciones que convierten los objetos a las representaciones de cadenas a menudo se denominan toString . Es mejor seguir las convenciones ciones como estas que inventar la tuya propia.

Los equipos suelen inventar su propio sistema estándar de nombres para un proyecto en particular. Eric Evans se refiere a esto como un *lenguaje omnipresente* para el proyecto. [14] Su código debe usar

14. [DDD].

Página 343

los términos de este idioma extensamente. En resumen, cuanto más puedas usar nombres que sean sobrecargado con significados especiales que son relevantes para su proyecto, más fácil será para lectores para saber de qué está hablando su código.

N4: *Nombres inequívocos*

Elija nombres que hagan que el funcionamiento de una función o variable sea inequívoco. Considerar este ejemplo de FitNesse:

```
Private String doRename () arroja una excepción
{
  if (refactorReferences)
    renameReferences ();
  renamePage ();

  pathToRename.removeNameFromEnd ();
  pathToRename.addNameToEnd (nuevoNombre);
  return PathParser.render (pathToRename);
}
```

El nombre de esta función no dice lo que hace la función, excepto en términos amplios y vagos.

condiciones. Esto se enfatiza por el hecho de que hay una función llamada renamePage dentro del función llamada doRename ! ¿Qué le dicen los nombres sobre la diferencia entre dos funciones? Nada.

Un mejor nombre para esa función es renamePageAndOptionallyAllReferences . Esto puede parece largo, y lo es, pero solo se llama desde un lugar en el módulo, por lo que es explicativo el valor supera la longitud.

N5: *Usar nombres largos para alcances largos*

La longitud de un nombre debe estar relacionada con la longitud del alcance. Puedes usar muy corto nombres de variable para ámbitos pequeños, pero para ámbitos grandes debería usar nombres más largos.

Los nombres de variables como i y j están bien si su alcance es de cinco líneas. Considera esto fragmento del antiguo "Juego de bolos" estándar:

```
private void rollMany (int n, int pines)
{
    para (int i = 0; i <n; i ++)
        g. roll (alfileres);
}
```

Esto es perfectamente claro y se ofuscaría si la variable i fuera reemplazada por alguna algo molesto como rollCount . Por otro lado, variables y funciones con nombres cortos pierden su significado en largas distancias. Entonces, cuanto más largo sea el alcance del nombre, más largo y más preciso debería ser el nombre.

N6: *Evite las codificaciones*

Los nombres no deben codificarse con información sobre el tipo o el alcance. Prefijos como m_ o f son inútiles en los entornos actuales. También codificaciones de proyectos y / o subsistemas como

Página 344

vis_ (para el sistema de imágenes visuales) distraen y son redundantes. Una vez más, el entorno actual Los mensajes proporcionan toda esa información sin tener que modificar los nombres. Mantener su nombres libres de contaminación húngara.

N7: Los *nombres deben describir los efectos secundarios*

Los nombres deben describir todo lo que una función, variable o clase es o hace. No te escondas efectos secundarios con un nombre. No use un verbo simple para describir una función que hace más que solo esa simple acción. Por ejemplo, considere este código de TestNG:

```
public ObjectOutputStream getOos () lanza IOException {
    if (m_oos == null) {
        m_oos = new ObjectOutputStream (m_socket.getOutputStream ());
    }
    return m_oos;
}
```

Esta función hace un poco más que obtener un "oos"; crea los "oos" si no se ha creado ya está. Por lo tanto, un mejor nombre podría ser createOrReturnOos .

Pruebas

T1: *Pruebas insuficientes*

¿Cuántas pruebas debe haber en una suite de pruebas? Desafortunadamente, la métrica que usan muchos programadores es "Eso parece suficiente". Un conjunto de pruebas debería probar todo lo que pueda romperse.
Las pruebas son insuficientes siempre que existan condiciones que no hayan sido exploradas por el pruebas o cálculos que no han sido validados.

T2: ¡ *Utilice una herramienta de cobertura!*

Las herramientas de cobertura informan sobre las lagunas en su estrategia de prueba. Facilitan la búsqueda de módulos, clases y funciones que no están suficientemente probadas. La mayoría de los IDE le brindan una indicación visual, marcando las líneas que están cubiertas de verde y las que están descubiertas de rojo. Esto lo hace rápida y fácil de encontrar si es o captura declaraciones cuyos cuerpos no han sido verificados.

T3: *No se salte las pruebas triviales*

Son fáciles de escribir y su valor documental es superior al costo de producción.
ellos.

T4: *Una prueba ignorada es una pregunta sobre una ambigüedad*

A veces no estamos seguros de un detalle de comportamiento porque los requisitos son
poco claro. Podemos expresar nuestra pregunta sobre los requisitos a modo de prueba que se comenta
out, o como una prueba anotada con @Ignore . Lo que elija depende de si el
la ambigüedad se trata de algo que se compilaría o no.

Página 345

T5: *Condiciones límite de prueba*

Tenga especial cuidado al probar las condiciones de contorno. A menudo obtenemos la mitad de un algoritmo correcto pero juzgar mal los límites.

T6: *Prueba exhaustiva de insectos cercanos*

Los insectos tienden a congregarse. Cuando encuentre un error en una función, es aconsejable hacer un exhaustivo prueba de esa función. Probablemente encontrará que el error no estaba solo.

T7: Los *patrones de fracaso son reveladores*

A veces, puede diagnosticar un problema al encontrar patrones en la forma en que fallan los casos de prueba. Este es otro argumento para hacer que los casos de prueba sean lo más completos posible. Prueba completa los casos, ordenados de manera razonable, exponen patrones.

Como ejemplo simple, suponga que notó que todas las pruebas con una entrada mayor de cinco los personajes fallaron? ¿O qué pasa si cualquier prueba que pasa un número negativo en el segundo argumento? fallo de una función? A veces, solo ver el patrón de rojo y verde en la prueba informe es suficiente para provocar el "¡Ajá!" que conduce a la solución. Vuelva a la página 267 para vea un ejemplo interesante de esto en el ejemplo de SerialDate .

T8: Los *patrones de cobertura de prueba pueden ser reveladores*

Observar el código que se ejecuta o no mediante las pruebas pasadas da pistas sobre por qué el las pruebas que fallan fallan.

T9: las *pruebas deben ser rápidas*

Una prueba lenta es una prueba que no se ejecuta. Cuando las cosas se ponen difíciles, son las pruebas lentas las que serán cayó de la suite. Así que *haga lo que sea necesario* para que sus pruebas sean rápidas.

Conclusión

Difícilmente podría decirse que esta lista de heurísticas y olores está completa. De hecho, no estoy seguro esa lista un tipo puede *nunca* ser completa. Pero tal vez la integridad no debería ser el objetivo, porque lo que esta lista *hace* do es implicar un sistema de valores.

De hecho, ese sistema de valores ha sido el objetivo y el tema de este libro. El código limpio es no escrito siguiendo un conjunto de reglas. No te conviertes en un artesano del software aprendiendo haciendo una lista de heurísticas. La profesionalidad y la artesanía provienen de valores que impulsan disciplinas.

Página 346

Bibliografía

[Refactorización]: *Refactorización: Mejora del diseño del código existente* , Martin Fowler et al., Addison-Wesley, 1999.

[PRAG]: *El programador pragmático* , Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

[GOF]: *Patrones de diseño: elementos de software orientado a objetos reutilizables* , Gamma et al., Addison-Wesley, 1996.

[Beck97]: *Patrones de mejores prácticas de Smalltalk* , Kent Beck, Prentice Hall, 1997.

[Beck07]: *Patrones de implementación* , Kent Beck, Addison-Wesley, 2008.

[PPP]: *Desarrollo de software ágil: principios, patrones y prácticas* , Robert C. Martin, Prentice Hall, 2002.

[DDD]: *Diseño basado en dominios* , Eric Evans, Addison-Wesley, 2003.

Página 347

Esta página se dejó en blanco intencionalmente

Página 348

Apéndice A

Simultaneidad II

por Brett L. Schuchert

Este apéndice respalda y amplía el capítulo de *simultaneidad* en la página 177. Está escrito como una serie de temas independientes y, por lo general, puede leerlos en cualquier orden. Hay alguna duplicación entre secciones para permitir dicha lectura.

Ejemplo de cliente / servidor

Imagínese una simple aplicación cliente / servidor. Un servidor se sienta y espera escuchando en un socket un cliente para conectarse. Un cliente se conecta y envía una solicitud.

El servidor

Aquí hay una versión simplificada de una aplicación de servidor. La fuente completa de este ejemplo está disponible. disponible a partir de la página 343, *Cliente / Servidor sin subprocesos* .

```
ServerSocket serverSocket = nuevo ServerSocket (8009);

while (keepProcessing) {
        intentar {
                Socket socket = serverSocket.accept ();
                proceso (socket);
        } captura (Excepción e) {
                manejar (e);
        }
}
```


Página 349

Esta sencilla aplicación espera una conexión, procesa un mensaje entrante y luego espera de nuevo a que llegue la próxima solicitud del cliente. Aquí está el código de cliente que se conecta a este servidor:

```
private void connectSendReceive (int i) {
        intentar {
                Socket socket = new Socket ("localhost", PUERTO);
                MessageUtils.sendMessage (socket, Integer.toString (i));
                MessageUtils.getMessage (socket);
                socket.close ();
        } captura (Excepción e) {
                e.printStackTrace ();

        }

}
```

¿Qué tan bien funciona este par cliente / servidor? ¿Cómo podemos describir formalmente ese desempeño mance? Aquí hay una prueba que afirma que el rendimiento es "aceptable":

```
@Test (tiempo de espera = 10000)
public void shouldRunInUnder10Seconds () lanza Exception {
        Subproceso [] subprocesos = createThreads ();
        startAllThreadsw (hilos);
        waitForAllThreadsToFinish (hilos);
}
```

La configuración se omite para simplificar el ejemplo (consulte “ ClientTest.java ” en la página 344). Esto prueba afirma que debe completarse en 10,000 milisegundos.

Este es un ejemplo clásico de validación del rendimiento de un sistema. Este sistema debe completar una serie de solicitudes de clientes en diez segundos. Siempre que el servidor pueda procesar cada solicitud individual del cliente a tiempo, la prueba pasará.

¿Qué pasa si la prueba falla? Aparte de desarrollar algún tipo de ciclo de sondeo de eventos, no hay mucho que hacer dentro de un solo hilo que haga que este código sea más rápido. Voluntad ¿El uso de varios subprocesos resuelve el problema? Podría, pero necesitamos saber dónde está el tiempo siendo gastado. Hay dos posibilidades:

- E / S: usando un socket, conectándose a una base de datos, esperando el intercambio de memoria virtual, y así.
- Procesador: cálculos numéricos, procesamiento de expresiones regulares, recolección de basura, y así.

Los sistemas suelen tener algunos de cada uno, pero para una operación determinada, uno tiende a dominar. Si el código está vinculado al procesador, más hardware de procesamiento puede mejorar el rendimiento, lo que nuestro pase de prueba. Pero hay una cantidad limitada de ciclos de CPU disponibles, por lo que agregar subprocesos a un El problema relacionado con el procesador no lo hará ir más rápido.

Por otro lado, si el proceso está vinculado a E / S, la concurrencia puede aumentar la eficiencia. eficiencia. Cuando una parte del sistema está esperando E / S, otra parte puede usar ese tiempo de espera para procesar otra cosa, haciendo un uso más eficaz de la CPU disponible.

Página 350

Adición de subprocesos

Suponga por el momento que la prueba de rendimiento falla. ¿Cómo podemos mejorar el poner de modo que pase la prueba de rendimiento? Si el método de proceso del servidor está vinculado a E / S, entonces aquí hay una forma de hacer que el servidor use subprocesos (solo cambie el processMessage):

```
proceso nulo (zócalo de enchufe final) {
        si (enchufe == nulo)
                regreso;

        ClientHandler ejecutable = new Runnable () {
                public void run () {
                        intentar {
                                Mensaje de cadena = MessageUtils.getMessage (socket);
                                MessageUtils.sendMessage (socket, "Procesado:" + mensaje);
                                closeIgnoringException (socket);
                        } captura (Excepción e) {
                                e.printStackTrace ();
                        }
                }
        };

        Thread clientConnection = nuevo Thread (clientHandler);
        clientConnection.start ();
}
```

Suponga que este cambio hace que la prueba pase; ı el código está completo, ¿correcto?

Observaciones del servidor

El servidor actualizado completa la prueba con éxito en poco más de un segundo. Desafortunadamente, esta solución es un poco ingenua e introduce algunos problemas nuevos.

¿Cuántos hilos podría crear nuestro servidor? El código no establece límites, por lo que podríamos alcanzar el límite impuesto por la máquina virtual Java (JVM). Para muchos sistemas simples tems esto puede ser suficiente. Pero, ¿y si el sistema está destinado a admitir a muchos usuarios del público ¿neto? Si demasiados usuarios se conectan al mismo tiempo, es posible que el sistema se detenga.

Pero deje el problema de comportamiento a un lado por el momento. La solución mostrada tiene problemas lemas de limpieza y estructura. ¿Cuántas responsabilidades tiene el código del servidor?

- Gestión de la conexión de socket
- Procesamiento de clientes
- Política de subprocesos
- Política de cierre del servidor

Desafortunadamente, todas estas responsabilidades viven en la función de proceso . además, el el código cruza muchos niveles diferentes de abstracción. Entonces, por pequeña que sea la función del proceso, necesita ser repartido.

1. Puede verificarlo usted mismo probando el código de antes y después. Revise el código sin subprocesos que comienza en la página 343. Revise el código enhebrado a partir de la página 346.

Página 351

El servidor tiene varias razones para cambiar; por tanto viola la Responsabilidad Única Principio. Para mantener limpios los sistemas concurrentes, la administración de subprocesos debe limitarse a unos pocos, lugares bien controlados. Además, cualquier código que gestione subprocesos no debería hacer nada. que no sea la gestión de subprocesos. ¿Por qué? Si no es por otra razón que la de rastrear con Los problemas de divisas son lo suficientemente difíciles sin tener que resolver otros problemas que no sean de divisas en al mismo tiempo.

Si creamos una clase separada para cada una de las responsabilidades enumeradas anteriormente, incluida la responsabilidad de la gestión de subprocesos, luego, cuando cambiamos la estrategia de gestión de subprocesos, el cambio afectará menos al código general y no contaminará las otras responsabilidades. Esto también hace que sea mucho más fácil probar todas las demás responsabilidades sin tener que preocuparse sobre el enhebrado. Aquí hay una versión actualizada que hace precisamente eso:

```
public void run () {
    while (keepProcessing) {
      intentar {
        ClientConnection clientConnection = connectionManager.awaitClient ();
        ClientRequestProcessor requestProcessor
            = nuevo ClientRequestProcessor (clientConnection);
        clientScheduler.schedule (requestProcessor);
        } captura (Excepción e) {
            e.printStackTrace ();
        }

    }
    connectionManager.shutdown ();

}
```

Esto ahora enfoca todas las cosas relacionadas con subprocesos en un solo lugar, clientScheduler . Si hay problemas de concurrencia, solo hay un lugar para buscar:

```
ClientScheduler de interfaz pública {
        programación nula (ClientRequestProcessor requestProcessor);
}
```

La política actual es fácil de implementar:

```
ThreadPerRequestScheduler de clase pública implementa ClientScheduler {
        programación nula pública (final ClientRequestProcessor requestProcessor) {
                Ejecutable ejecutable = new Ejecutable () {
                        public void run () {
                                requestProcessor.process ();
                        }
                };
```

```
            Subproceso subproceso = nuevo subproceso (ejecutable);
            thread.start ();
        }
    }
```

Habiendo aislado toda la gestión de subprocesos en un solo lugar, es mucho más fácil cambiar la forma en que controlamos los hilos. Por ejemplo, pasar al marco Java 5 Executor implica escribir una nueva clase y conectarla (Listado A-1).

Página 352

Listado A-1

ExecutorClientScheduler.java

```
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

ExecutorClientScheduler de clase pública implementa ClientScheduler {
    Albacea albacea;

    public ExecutorClientScheduler (int availableThreads) {
        ejecutor = Ejecutores.newFixedThreadPool (availableThreads);
    }

    programación nula pública (final ClientRequestProcessor requestProcessor) {
        Ejecutable ejecutable = new Ejecutable () {
            public void run () {
                requestProcessor.process ();
            }
        };
        ejecutor.execute (ejecutable);
    }
}
```

Conclusión

La introducción de la simultaneidad en este ejemplo particular demuestra una forma de mejorar la rendimiento de un sistema y una forma de validar ese rendimiento a través de un marco de prueba trabaja. Centrar todo el código de concurrencia en un pequeño número de clases es un ejemplo de aplicando el Principio de Responsabilidad Única. En el caso de la programación concurrente, este se vuelve especialmente importante debido a su complejidad.

Posibles rutas de ejecución

Revise el método `incrementValue` , un método Java de una línea sin bucles ni ramificaciones:

```
IdGenerator de clase pública {
   int lastIdUsed;

   public int incrementValue () {
      return ++ lastIdUsed;
   }
}
```

Ignore el desbordamiento de enteros y asuma que solo un hilo tiene acceso a una sola instancia de `IdGenerator` . En este caso hay una única vía de ejecución y una única garantía resultado:

- El valor devuelto es igual al valor de `lastIdUsed` , ambos son uno mayor que justo antes de llamar al método.

Página 353

¿Qué sucede si usamos dos hilos y dejamos el método sin cambios? Cuales son los posibles resultados si cada hilo llama a incrementValue una vez? Cuantos caminos posibles de ejecución hay? Primero, los resultados (suponga que lastIdUsed comienza con un valor de 93):

- **El** subproceso 1 obtiene el valor de 94, el subproceso 2 obtiene el valor de 95 y lastIdUsed ahora es 95.
- **El** subproceso 1 obtiene el valor de 95, el subproceso 2 obtiene el valor de 94 y lastIdUsed ahora es 95.
- **El** subproceso 1 obtiene el valor de 94, el subproceso 2 obtiene el valor de 94 y lastIdUsed ahora es 94.

El resultado final, aunque sorprendente, es posible. Para ver cómo estos resultados diferentes son posibles ble, necesitamos comprender el número de posibles rutas de ejecución y cómo el Java La máquina virtual los ejecuta.

Numero de caminos

Para calcular el número de posibles rutas de ejecución, comenzaremos con el byte generado: código. La única línea de java (return ++ lastIdUsed;) se convierte en instrucciones de código de ocho bytes. Eso Es posible que los dos hilos intercalen la ejecución de estas ocho instrucciones el forma en que un crupier entrelaza las cartas mientras baraja una baraja. [2] Incluso con solo ocho cartas en en cada mano, hay un número notable de resultados mezclados.

Para este caso simple de *N* instrucciones en una secuencia, sin bucles ni condicionales, y *T* subprocesos, el número total de posibles rutas de ejecución es igual a

$$\frac{()!\,\textit{Nuevo Testamento}}{\overline{T}\;\text{¡}\,N\,!}$$

Calcular los posibles pedidos

Esto proviene de un correo electrónico del tío Bob a Brett:

Con *N* pasos e hilos *T* hay *T* * *N* pasos totales. Antes de cada paso hay un cambio de contexto que elige entre los subprocesos *T.* Cada camino puede por lo tanto, se representará como una cadena de dígitos que denota los cambios de contexto. Dados los pasos A y B y los hilos 1 y 2, los seis caminos posibles son 1122, 1212, 1221, 2112, 2121 y 2211. O, en términos de pasos, es A1B1A2B2, A1A2B1B2, A1A2B2B1, A2A1B1B2, A2A1B2B1 y A2B2A1B1. Para tres hilos la secuencia es 112233, 112323, 113223, 113232, 112233, 121233, 121323, 121332, 123132, 123123,. . . .

Una característica de estas cadenas es que siempre debe haber *N* instancias de cada *T* . Entonces la cadena 111111 no es válida porque tiene seis instancias de 1 y cero instancias de 2 y 3.

2. Esto es un poco simplificado. Sin embargo, para el propósito de esta discusión, podemos usar este modelo simplificador.

Página 354

Cálculo de los posibles ordenamientos (continuación)

Entonces queremos las permutaciones de *N* 1, *N* 2,. . . y *N T* 's. Esto es realmente solo las permutaciones de *N* * *T* cosas tomadas *N* * *T* a la vez, que es (*N* * *T*) !, pero con todos los duplicados eliminados. Entonces el truco es contar el duplica y resta eso de (*N* * *T*) !.

Dados dos pasos y dos hilos, ¿cuántos duplicados hay? Cada La cadena de cuatro dígitos tiene dos unos y dos dos. Cada uno de esos pares podría ser intercambiado sin cambiar el sentido de la cadena. Podrías intercambiar los 1 o los 2 ambos, o ninguno. Entonces hay cuatro isomorfos para cada cadena, que significa que hay tres duplicados. Así que tres de cada cuatro opciones son duplicados; alternativamente, una de las cuatro permutaciones NO son duplicadas cates. 4! * .25 = 6. Entonces, este razonamiento parece funcionar.

¿Cuántos duplicados hay? En el caso donde *N* = 2 y *T* = 2, I podría intercambiar los 1, los 2 o ambos. En el caso donde *N* = 2 y *T* = 3, I podría intercambiar los 1, 2, 3, 1 y 2, 1 y 3, o 2 y 3. Intercambio- ping es sólo las permutaciones de *N* . Digamos que hay *P* permutaciones de *N* . El número de diferentes maneras de organizar las permutaciones son *P* ** *T* .

Por lo que el número de posibles isomorfos es *N* ** *T* . Y entonces el número de caminos es (*T* * *N*)! / (*N* ! ** *T*). Nuevamente, en nuestro caso *T* = 2, *N* = 2 obtenemos 6 (24/4).

Para *N* = 2 y *T* = 3 obtenemos 720/8 = 90.

Para *N* = 3 y *T* = 3 obtenemos 9! / 6 ^ 3 = 1680.

Para nuestro caso simple de una línea de código Java, que equivale a ocho líneas de código de bytes y dos subprocesos, el número total de posibles caminos de ejecución es 12,870. Si el tipo de lastIdUsed es largo , luego cada lectura / escritura se convierte en dos operaciones en lugar de una, y el número de posibles pedidos se convierte en 2.704.156.

¿Qué sucede si realizamos un cambio en este método?

```
incrementValue void sincronizado público () {
      ++ lastIdUsed;
}
```

El número de posibles vías de ejecución se convierte en dos para dos subprocesos y N! en el caso general.

Cavar más profundo

¿Qué pasa con el resultado sorprendente de que dos subprocesos podrían llamar al método una vez (antes agregamos sincronizados) y obtenemos el mismo resultado numérico? ¿Cómo es eso posible? Primero las cosas primero.

¿Qué es una operación atómica? Podemos definir una operación atómica como cualquier operación que es ininterrumpida. Por ejemplo, en el siguiente código, línea 5, donde se asigna 0 a lastid , es atómico porque de acuerdo con el modelo de memoria de Java, la asignación a un 32 bits El valor es ininterrumpido.

Página 355


```
01: Ejemplo de clase pública {
02: int lastId;
03:
04: public void resetId () {
05:                 valor = 0;
06:}
07:
08: public int getNextId () {
09:                 ++ valor;
```

```
10:}
11:}
```

¿Qué sucede si cambiamos el tipo de lastId de int a long ? ¿La línea 5 sigue siendo atómica? No de acuerdo con la especificación JVM. Podría ser atómico en un procesador en particular, pero de acuerdo con la especificación JVM, la asignación a cualquier valor de 64 bits requiere dos Asignaciones de 32 bits. Esto significa que entre la primera asignación de 32 bits y la segunda Asignación de 32 bits, algún otro hilo podría colarse y cambiar uno de los valores.

¿Qué pasa con el operador de preincremento, ++, en la línea 9? El operador de preincremento puede interrumpirse, por lo que no es atómico. Para entenderlo, revisemos el código de bytes de ambos métodos en detalle.

Antes de continuar, aquí hay tres definiciones que serán importantes:

- Marco: cada invocación de método requiere un marco. El marco incluye la devolución dirección, cualquier parámetro pasado al método y las variables locales definidas en el método. Esta es una técnica estándar utilizada para definir una pila de llamadas, que es utilizada por lenguajes modernos para permitir la invocación básica de funciones / métodos y para permitir invocación recursiva.
- Variable local: cualquier variable definida en el alcance del método. Todos los métodos no estáticos ods tienen al menos una variable, **esta** , que representa el objeto actual, el objeto que recibió el mensaje más reciente (en el hilo actual), lo que provocó la invocación del método.
- Pila de operandos: muchas de las instrucciones de la máquina virtual Java toman parámetros ters. La pila de operandos es donde se colocan esos parámetros. La pila es un estándar Estructura de datos de último en entrar, primero en salir (LIFO).

Aquí está el código de bytes generado para resetId () :

Mnemotécnico	Descripción	Operando Apilar después
ALOAD 0	Cargue la variable 0 en la pila de operandos. ¿Cuál es la variable 0? Es **esto** ., La corriente objeto. Cuando se llamó al método, el receptor del mensaje, una instancia de Example , fue empujado a la matriz de variables locales de la marco creado para la invocación del método. Esto es siempre la primera variable puesta en cada instancia método.	esto

Página 356

Mnemotécnico	Descripción	Operando Apilar después
ICONST_0	Ponga el valor constante 0 en la pila de operandos. esto, 0	
PUTFIELD lastId	Almacene el valor superior en la pila (que es 0) en el valor de campo del objeto al que hace referencia el referencia de objeto a una distancia de la parte superior de la apilar, **esto** .	<vacío>

Se garantiza que estas tres instrucciones serán atómicas porque, aunque el hilo ejecutarlos podría ser interrumpido después de cualquiera de ellos, la información para el Instrucción PUTFIELD (el valor constante 0 en la parte superior de la pila y la referencia a este debajo de la parte superior, junto con el valor del campo) no puede ser tocado por otro hilo. Entonces, cuando ocurre la asignación, tenemos la garantía de que el valor 0 se almacenará en el valor de campo. La operación es atómica. Todos los operandos tratan con información local del método, por lo que no hay interferencia entre varios subprocesos.

Entonces, si estas tres instrucciones son ejecutadas por diez subprocesos, hay 4.38679733629e + 24 posibles pedidos. Sin embargo, solo hay un resultado posible, por lo que los diferentes ordenamientos son irrelevantes. Da la casualidad de que el mismo resultado está garantizado durante mucho tiempo en este caso.

también. ¿Por qué? Los diez subprocesos asignan un valor constante. Incluso si se intercalan con entre sí, el resultado final es el mismo.

Con la operación ++ en el método getNextId , habrá problemas.
Suponga que lastId tiene 42 al comienzo de este método. Aquí está el código de bytes para esto Nuevo método:

Mnemotécnico	Descripción	Operando Apilar después
ALOAD 0	Cargue esto en la pila de operandos	esto
DUP	Copie la parte superior de la pila. Ahora tenemos dos copias de esto en la pila de operandos.	esto , esto
GETFIELD lastId	Recupere el valor del campo lastId del objeto apuntado en la parte superior de la pila (esto) y almacenar ese valor en la pila.	esto , 42
ICONST_1	Empuje la constante entera 1 en la pila.	esto , 42, 1
AÑADO	Entero suma los dos valores superiores en el operando apilar y almacenar el resultado de nuevo en el operando apilar.	esto , 43
DUP_X1	Duplica el valor 43 y ponlo antes de este .	43, esto , 43
Valor de PUTFIELD	Almacene el valor superior en la pila de operandos, 43, en el valor de campo del objeto actual, representado por el siguiente valor en la pila de operandos, this .	43
VUELVO	devuelve el valor superior (y único) de la pila.	<vacío>

Página 357

Imagine el caso en el que el primer hilo completa las tres primeras instrucciones, hasta y incluido GETFIELD, y luego se interrumpe. Un segundo hilo toma el control y realiza el método completo, incrementando lastId en uno; obtiene 43 de vuelta. Entonces el primer hilo se levanta donde lo dejó; 42 todavía está en la pila de operandos porque ese era el valor de lastId cuando ejecutado GETFIELD. Agrega uno para obtener 43 nuevamente y almacena el resultado. El valor 43 es volvió al primer hilo también. El resultado es que uno de los incrementos se pierde porque el El primer hilo pisó el segundo hilo después de que el segundo hilo interrumpió el primer hilo.

La sincronización del método getNexId () soluciona este problema.

Conclusión

No es necesario un conocimiento profundo del código de bytes para comprender cómo los subprocesos pueden pisar el uno al otro. Si puede entender este ejemplo, debe demostrar la posi-
posibilidad de que varios subprocesos se pisen entre sí, lo cual es suficiente conocimiento.

Dicho esto, lo que demuestra este ejemplo trivial es la necesidad de comprender la modelo de memoria suficiente para saber qué es y qué no es seguro. Es un error común pensar que el operador ++ (pre o post-incremento) es atómico, y claramente no lo es. Esto significa tu necesito saber:

- Donde hay objetos / valores compartidos
- El código que puede causar problemas de lectura / actualización simultáneos
- Cómo evitar que ocurran estos problemas concurrentes

Conociendo su biblioteca

Marco del ejecutor

Como se demuestra en ExecutorClientScheduler.java en la página 321, el marco Executor-
El trabajo introducido en Java 5 permite una ejecución sofisticada utilizando grupos de subprocesos. Esto es un

class en el paquete java.util.concurrent .

Si está creando subprocesos y no está utilizando un grupo de subprocesos o *está* utilizando un escrito a mano uno, debería considerar usar el Ejecutor . Hará que su código sea más limpio y más fácil de seguir. bajo y más pequeño.

El marco del ejecutor agrupará subprocesos, cambiará el tamaño automáticamente y volverá a crear subprocesos si necesario. También admite *futuros,* una construcción de programación concurrente común. La Executor framework trabaja con clases que implementan Runnable y también trabaja con clases que implementan la interfaz invocable . Un invocable parece un Runnable , pero puede devuelve un resultado, que es una necesidad común en soluciones multiproceso.

Un *futuro* es útil cuando el código necesita ejecutar múltiples operaciones independientes y espera a que ambos terminen:

```
public String processRequest (mensaje de cadena) lanza Exception {
    Invocable <String> makeExternalCall = new Invocable <String> () {
```


Página 358


```
            public String call () lanza Exception {
                Resultado de cadena = "";
                // hacer una solicitud externa
                devolver resultado;
            }
        };

        Future <String> resultado = ejecutorService.submit (makeExternalCall);
        String ParticularResult = doSomeLocalProcessing ();
        return result.get () + parcialResult;
}
```

En este ejemplo, el método comienza a ejecutar el objeto makeExternalCall . El método con- Continúa otros procesos. La línea final llama a result.get () , que bloquea hasta el futuro completa.

Soluciones no bloqueantes

La máquina virtual Java 5 aprovecha el diseño de procesador moderno, que admite actualizaciones sin bloqueo. Considere, por ejemplo, una clase que usa sincronización (y por lo tanto- bloqueo frontal) para proporcionar una actualización segura para subprocesos de un valor:

```
clase pública ObjectWithValue {
    valor int privado;
    public void sincronizado incrementValue () {++ valor; }
    public int getValue () {valor de retorno; }
}
```

Java 5 tiene una serie de nuevas clases para situaciones como esta: AtomicBoolean , AtomicInteger y AtomicReference son tres ejemplos; hay varios más. Podemos reescriba el código anterior para usar un enfoque sin bloqueo de la siguiente manera:

```
clase pública ObjectWithValue {
    valor de AtomicInteger privado = new AtomicInteger (0);

    public void incrementValue () {
        value.incrementAndGet ();
    }
    public int getValue () {
        return value.get ();
    }
}
```

Aunque esto usa un objeto en lugar de una primitiva y envía mensajes como incrementAndGet () en lugar de ++, el rendimiento de esta clase casi siempre superará al versión previa. En algunos casos será solo un poco más rápido, pero los casos en los que será más lentos son prácticamente inexistentes.

¿Cómo es esto posible? Los procesadores modernos tienen una operación que normalmente se llama *Comparar y Swap (CAS)* . Esta operación es análoga al bloqueo optimista en bases de datos, mientras que la versión sincronizada es análoga al bloqueo pesimista.

Página 359

La palabra clave sincronizada siempre adquiere un bloqueo, incluso cuando un segundo hilo no está intentando actualizar el mismo valor. A pesar de que el rendimiento de las cerraduras intrínsecas ha mejorados de una versión a otra, siguen siendo costosos.

La versión sin bloqueo comienza con la suposición de que varios subprocesos generalmente lo hacen No modifique el mismo valor con la frecuencia suficiente para que surja un problema. En cambio, de manera eficiente detecta si se ha producido una situación de este tipo y vuelve a intentarlo hasta que la actualización se realiza correctamente. completamente. Esta detección es casi siempre menos costosa que adquirir un candado, incluso en condiciones moderadas a situaciones de alta contención.

¿Cómo logra esto la máquina virtual? La operación CAS es atómica. Logi- En términos generales, la operación CAS se parece a lo siguiente:

```
int variableBeingSet;

void simulateNonBlockingSet (int newValue) {
        int currentValue;
        hacer {
                currentValue = variableBeingSet
        } while (currentValue! = compareAndSwap (currentValue, newValue));
}

int sincronizado compareAndSwap (int currentValue, int newValue) {
        if (variableBeingSet == currentValue) {
                variableBeingSet = newValue;
                return currentValue;
        }
        return variableBeingSet;
}
```

Cuando un método intenta actualizar una variable compartida, la operación CAS verifica que la variable que se está configurando todavía tiene el último valor conocido. Si es así, se cambia la variable. Si no, entonces la variable no se establece porque otro hilo logró interponerse en el camino. La El método que realiza el intento (utilizando la operación CAS) ve que no se realizó el cambio. y reintentos.

Clases no seguras para subprocesos

Hay algunas clases que intrínsecamente no son seguras para subprocesos. Aquí están algunos ejemplos:

- SimpleDateFormat
- Conexiones de base de datos
- Contenedores en java.util
- Servlets

Tenga en cuenta que algunas clases de colección tienen métodos individuales que son seguros para subprocesos. Sin emabargo, cualquier operación que implique llamar a más de un método no lo es. Por ejemplo, si lo hace no desea reemplazar algo en una HashTable porque ya está allí, puede escribir el siguiente código:

```
if (! hashTable.containsKey (someKey)) {
        hashTable.put (someKey, nuevo SomeValue ());
}
```

Página 360

Cada método individual es seguro para subprocesos. Sin embargo, otro hilo podría agregar un valor en entre las llamadas containsKey y put . Hay varias opciones para solucionar este problema.

- Bloquee HashTable primero y asegúrese de que todos los demás usuarios de HashTable hagan lo mismo: bloqueo basado en el cliente:

```
sincronizado (mapa) {
si (! map.conainsKey (clave))
        map.put (clave, valor);
}
```

- Envuelva la HashTable en su propio objeto y use una API diferente: bloqueo basado en servidor usando un A DAPTER :

```
public class WrappedHashtable <K, V> {
        mapa privado <K, V> mapa = nueva tabla hash <K, V> ();

        putIfAbsent vacío sincronizado público (clave K, valor V) {
                si (map.containsKey (clave))
                        map.put (clave, valor);
        }
}
```

- Utilice las colecciones seguras para subprocesos:

```
ConcurrentHashMap <Integer, String> map = new ConcurrentHashMap <Integer,
Cadena> ();
map.putIfAbsent (clave, valor);
```

Las colecciones en java.util.concurrent tienen operaciones como putIfAbsent () para acomodar fecha tales operaciones.

Las dependencias entre Método s Puede romper el código concurrente

Aquí hay un ejemplo trivial de una forma de introducir dependencias entre métodos:

```
La clase pública IntegerIterator implementa Iterator <Integer>
        entero privado nextValue = 0;

        hasNext () {público sincronizado booleano
                return nextValue <100000;
        }
        Entero público sincronizado next () {
                si (nextValue == 100000)
                        lanzar nueva IteratorPastEndException ();
                return nextValue ++;
        }
        Entero público sincronizado getNextValue () {
                return nextValue;
        }
}
```

Aquí hay un código para usar este IntegerIterator :

```
IntegerIterator iterador = nuevo IntegerIterator ();
while (iterator.hasNext ()) {
```


Página 361


```
        int nextValue = iterator.next ();
        // hacer algo con nextValue
```

```
}
```

Si un hilo ejecuta este código, no habrá ningún problema. Pero que pasa si dos hilos intentar compartir una sola instancia de IngeterIterator con la intención de que cada hilo procesar los valores que obtiene, pero que cada elemento de la lista se procesa solo una vez? La mayoría de el tiempo, no pasa nada malo; los hilos comparten felizmente la lista, procesando los elementos son dados por el iterador y se detienen cuando el iterador está completo. Sin embargo, hay una pequeña posibilidad de que, al final de la iteración, los dos hilos interfieran con cada other y hacer que un subproceso vaya más allá del final del iterador y arroje una excepción.

Aquí está el problema: el hilo 1 hace la pregunta hasNext () , que devuelve verdadero . Hilo 1 se adelanta y luego Thread 2 hace la misma pregunta, que sigue siendo cierta . Hilo 2 luego llama a next () , que devuelve un valor como se esperaba pero tiene el efecto secundario de hacer hasNext () devuelve falso . El hilo 1 se inicia de nuevo, el pensamiento hasNext () sigue siendo cierto , y luego llama a next () . Aunque los métodos individuales están sincronizados, el cliente utiliza ***dos*** métodos.

Este es un problema real y un ejemplo de los tipos de problemas que surgen en código actual. En esta situación particular, este problema es especialmente sutil porque el único el momento en el que esto causa una falla es cuando ocurre durante la iteración final del iterador. Si los hilos se rompen correctamente, entonces uno de los hilos podría ir más allá del final. del iterador. Este es el tipo de error que ocurre mucho después de que un sistema ha estado en proceso. ducción, y es difícil de rastrear.

Tienes tres opciones:

- Tolerar el fracaso.
- Resuelva el problema cambiando el cliente: bloqueo basado en el cliente
- Resuelva el problema cambiando el servidor, que además cambia el cliente: bloqueo basado en servidor

Tolerar el fracaso

A veces, puede configurar las cosas de manera que la falla no cause ningún daño. Por ejemplo, el el cliente anterior podría detectar la excepción y limpiar. Francamente, esto es un poco descuidado. Es bastante como limpiar las pérdidas de memoria reiniciando a la medianoche.

Bloqueo basado en el cliente

Para que IntegerIterator funcione correctamente con varios subprocesos, cambie este cliente (y todos los demás clientes) de la siguiente manera:

```
IntegerIterator iterador = nuevo IntegerIterator ();

    while (verdadero) {
      int nextValue;
```


Página 362


```
      sincronizado (iterador) {
        si (! iterator.hasNext ())
          rotura;
        nextValue = iterator.next ();
      }
      doSometingWith (nextValue);
    }
```

Cada cliente introduce un candado a través de la palabra clave sincronizada . Esta duplicación viola la El principio DRY, pero podría ser necesario si el código utiliza herramientas de terceros no seguras para subprocesos.

Esta estrategia es arriesgada porque todos los programadores que usan el servidor deben recordar bloquéelo antes de usarlo y desbloquéelo cuando termine. Hace muchos (¡muchos!) Años trabajé en un sistema que empleaba el bloqueo basado en el cliente en un recurso compartido. El recurso se utilizó en cientos de lugares diferentes a lo largo del código. Un pobre programador olvidó bloquear el recurso en uno de esos lugares.

El sistema era un sistema de tiempo compartido de múltiples terminales que ejecutaba software de contabilidad para el Local 705 del sindicato de camioneros. La computadora estaba en un piso elevado, ambiente-sala controlada 50 millas al norte de la sede del Local 705. En la sede había docenas de empleados de entrada de datos que escribían las cotizaciones de las cuotas sindicales en las terminales. La terminación Los usuarios estaban conectados a la computadora usando líneas telefónicas dedicadas y semidúplex de 600 bps. módems. (Este fue un muy, *muy* largo tiempo atrás.)

Aproximadamente una vez al día, una de las terminales se "cerraba". No había rima ni razón. hijo a eso. El bloqueo no mostró preferencia por terminales o horarios particulares. Eso era como si alguien tirara los dados eligiendo la hora y la terminal para bloquear. A veces, más de una terminal se bloquea. A veces los días pasaban sin cualquier bloqueo.

Al principio, la única solución fue reiniciar. Pero los reinicios fueron difíciles de coordinar. Tuvimos llamar a la sede y hacer que todos terminen lo que estaban haciendo en todas las terminales nals. Entonces podríamos apagar y reiniciar. Si alguien estuviera haciendo algo importante Eso tomó una hora o dos, la terminal cerrada simplemente tenía que permanecer cerrada.

Después de algunas semanas de depuración, descubrimos que la causa era un contador de búfer de anillo que se había desincronizado con su puntero. Esta salida controlada por búfer al terminal. La El valor del puntero indicó que el búfer estaba vacío, pero el contador dijo que estaba lleno. Porque estaba vacío, no había nada que mostrar; pero como también estaba lleno, nada podía ser agregado al búfer para que se muestre en la pantalla.

Entonces sabíamos por qué los terminales se estaban bloqueando, pero no sabíamos por qué el búfer de anillo se estaba desincronizando. Así que agregamos un truco para solucionar el problema. Era posible lea los interruptores del panel frontal de la computadora. (Este fue un muy, muy, *muy* largo tiempo atrás.) Escribimos una pequeña función de trampa que detectaba cuando se activaba uno de estos interruptores y luego buscó un búfer circular que estuviera vacío y lleno. Si se encuentra uno, se restablece búfer para vaciar. *¡Voila!* Los terminales bloqueados comenzaron a mostrarse nuevamente.

Así que ahora no tuvimos que reiniciar el sistema cuando una terminal se bloqueó. El local simplemente nos llamaría y nos diría que teníamos un bloqueo, y luego simplemente entramos en el complejo sala de ordenadores y pulsó un interruptor.

Página 363

Por supuesto, a veces trabajaban los fines de semana y nosotros no. Entonces agregamos un función al programador que verificó todos los búferes de anillo una vez por minuto y restableció cualquier que estaban vacíos y llenos. Esto provocó que las pantallas se desatascaran antes de que el Local incluso hablar por teléfono.

Fueron varias semanas más de estudiar detenidamente una página tras otra del lenguaje de ensamblaje monolítico. guage code antes de encontrar al culpable. Hicimos los cálculos y calculamos que la frecuencia La frecuencia de los bloqueos fue consistente con un solo uso sin protección del búfer de anillo. Entonces todo lo que teníamos que hacer era encontrar ese uso defectuoso. Desafortunadamente, esto fue hace mucho tiempo que no teníamos herramientas de búsqueda ni referencias cruzadas ni ningún otro tipo de ayuda automatizada. Simplemente teníamos que estudiar detenidamente los listados.

Aprendí una lección importante ese frío invierno de Chicago de 1971. Bloqueo basado en el cliente realmente sopla.

Bloqueo basado en servidor

La duplicación se puede eliminar realizando los siguientes cambios en IntegerIterator :

```
public class IntegerIteratorServerLocked {
        entero privado nextValue = 0;
        Entero público sincronizado getNextOrNull () {
                si (nextValue <100000)
                        return nextValue ++;
                demás
                        devolver nulo;
        }
}
```

Y el código del cliente también cambia:

```
while (verdadero) {
        Integer nextValue = iterator.getNextOrNull ();
        si (siguiente == nulo)
                rotura;
        // hacer algo con nextValue
}
```

En este caso, cambiamos la API de nuestra clase para que sea compatible con múltiples subprocesos. [3] El cliente necesita realizar una verificación nula en lugar de verificar hasNext () .

En general, debería preferir el bloqueo basado en servidor por estas razones:

- Reduce el código repetido: el bloqueo basado en el cliente obliga a cada cliente a bloquear el servidor. adecuadamente. Al poner el código de bloqueo en el servidor, los clientes pueden usar el objeto y no se preocupe por escribir código de bloqueo adicional.

3. De hecho, la interfaz Iterator no es inherentemente segura para subprocesos. Nunca fue diseñado para ser utilizado por varios subprocesos, por lo que este No debería sorprendernos.

Página 364

- Permite un mejor rendimiento: puede cambiar un servidor seguro para subprocesos por uno que no sea subproceso seguro en el caso de implementación de un solo subproceso, evitando así todos gastos generales.
- Reduce la posibilidad de error: todo lo que necesita es que un programador se olvide de bloquear adecuadamente.
- Hace cumplir una sola política: la política está en un lugar, el servidor, en lugar de muchos lugares, cada cliente.
- Reduce el alcance de las variables compartidas: el cliente no las conoce ni sabe cómo están bloqueados. Todo eso está oculto en el servidor. Cuando las cosas se rompen, el número de los lugares para mirar es más pequeño.

¿Qué sucede si no posee el código del servidor?

- Use un A DAPTER para cambiar la API y agregar bloqueo

```
public class ThreadSafeIntegerIterator {
        iterador IntegerIterator privado = new IntegerIterator ();

        Entero público sincronizado getNextOrNull () {
                si (iterador.hasNext ())
                        return iterator.next ();
                devolver nulo;
        }
}
```

- O mejor aún, use las colecciones seguras para subprocesos con interfaces extendidas

Aumento del rendimiento

Supongamos que queremos salir a la red y leer el contenido de un conjunto de páginas de una lista de URL. A medida que se lee cada página, la analizaremos para acumular algunas estadísticas. Una vez Se leen todas las páginas, imprimiremos un informe resumido.

La siguiente clase devuelve el contenido de una página, dada una URL.

```
PageReader de clase pública {
    // ...
    public String getPageFor (String url) {
        Método HttpMethod = new GetMethod (url);

        intentar {
            httpClient.executeMethod (método);
            Respuesta de cadena = method.getResponseBodyAsString ();
```

```
            respuesta de retorno;
        } captura (Excepción e) {
            manejar (e);
        } finalmente {
            method.releaseConnection ();
        }
    }
}
```


Página 365

La siguiente clase es el iterador que proporciona el contenido de las páginas basándose en un iterador de URL:

```
PageIterator de clase pública {
    lector de PageReader privado;
    URL privadas de URLIterator;

    public PageIterator (lector de PageReader, URL de URLIterator) {
        this.urls = urls;
        this.reader = reader;
    }

    público sincronizado String getNextPageOrNull () {
        si (urls.hasNext ())
            getPageFor (urls.next ());
        demás
            devolver nulo;
    }

    public String getPageFor (String url) {
        return reader.getPageFor (url);
    }
}
```

Una instancia de PageIterator se puede compartir entre muchos subprocesos diferentes, cada uno uno que usa su propia instancia del PageReader para leer y analizar las páginas que obtiene del iterador.

Observe que hemos mantenido el bloque sincronizado muy pequeño. Contiene solo lo crítico sección en el interior del PageIterator . Siempre es mejor sincronizar lo menos posible en lugar de sincronizar tanto como sea posible.

Cálculo de rendimiento de un solo hilo

Ahora hagamos algunos cálculos simples. A los efectos de la argumentación, suponga lo siguiente:

- Tiempo de E / S para recuperar una página (promedio): 1 segundo
- Tiempo de procesamiento para analizar la página (promedio): .5 segundos
- **La** E / S requiere el 0 por ciento de la CPU mientras que el procesamiento requiere el 100 por ciento.

Para *N* páginas procesadas por un solo hilo, el tiempo total de ejecución es de 1,5 segundos. onds * *N* . La Figura A-1 muestra una instantánea de 13 páginas o aproximadamente 19,5 segundos.

Figura A-1
Hilo único

Página 366

Cálculo de rendimiento multiproceso

Si es posible recuperar páginas en cualquier orden y procesarlas de forma independiente, entonces
Es posible utilizar varios subprocesos para aumentar el rendimiento. ¿Qué pasa si usamos tres
¿hilos? ¿Cuántas páginas podemos adquirir al mismo tiempo?

Como puede ver en la Figura A-2, la solución multiproceso permite el proceso enlazado
análisis de las páginas para que se superpongan con la lectura de las páginas enlazadas con E / S. En un idealizado
mundo esto significa que el procesador está completamente utilizado. Cada página de un segundo que se lee se termina
bañado con dos parses. Por lo tanto, podemos procesar dos páginas por segundo, que es tres veces
el rendimiento de la solución de un solo subproceso.

Figura A-2
Tres subprocesos concurrentes

Punto muerto

Imagine una aplicación web con dos grupos de recursos compartidos de un tamaño finito:

- Un grupo de conexiones de base de datos para el trabajo local en el almacenamiento de procesos.
- Un grupo de conexiones MQ a un repositorio principal

Suponga que hay dos operaciones en esta aplicación, crear y actualizar:

- Crear: obtenga una conexión al repositorio principal y la base de datos. Habla con el maestro de servicio
  repositorio y luego almacenar el trabajo en el trabajo local en la base de datos del proceso.

Página 367

- Actualizar: Adquiera la conexión a la base de datos y luego al repositorio principal. Leer del trabajo en la base de datos de proceso y luego enviar al repositorio principal

¿Qué sucede cuando hay más usuarios que el tamaño de la piscina? Considere que cada piscina tiene un tamaño de diez.

- Diez usuarios intentan utilizar create, por lo que se adquieren las diez conexiones de base de datos y cada El hilo se interrumpe después de adquirir una conexión a la base de datos pero antes de adquirir una conexión. conexión al repositorio principal.
- Diez usuarios intentan utilizar la actualización, por lo que se adquieren las diez conexiones del repositorio principal. y cada hilo se interrumpe después de adquirir el repositorio maestro pero antes de adquirir una conexión a la base de datos.
- Ahora los diez subprocesos "crear" deben esperar para adquirir una conexión de repositorio principal, pero los diez subprocesos de "actualización" deben esperar para adquirir una conexión a la base de datos.
- Punto muerto. El sistema nunca se recupera.

Esto puede parecer una situación poco probable, pero ¿quién quiere un sistema que se congele? ¿cualquier otra semana? ¿Quién quiere depurar un sistema con síntomas que son tan difíciles de ¿reproducir? Este es el tipo de problema que ocurre en el campo y luego se necesitan semanas para resolverlo.

Una "solución" típica es introducir declaraciones de depuración para averiguar qué está sucediendo. En g. Por supuesto, las declaraciones de depuración cambian el código lo suficiente como para que ocurra el interbloqueo. en una situación diferente y tarda meses en volver a ocurrir. [4]

Para resolver realmente el problema del interbloqueo, debemos comprender qué lo causa. Allí Hay cuatro condiciones necesarias para que se produzca un interbloqueo:

- Exclusión mutua
- Bloquear y esperar
- Sin preferencia
- Espera circular

Exclusión mutua

La exclusión mutua se produce cuando varios subprocesos necesitan utilizar los mismos recursos y esos recursos

- No puede ser utilizado por varios subprocesos al mismo tiempo.
- Son limitados en número.

Un ejemplo común de tal recurso es una conexión de base de datos, un archivo abierto para escritura, un bloqueo de registro, o un semáforo.

4. Por ejemplo, alguien agrega algunos resultados de depuración y el problema "desaparece". El código de depuración "corrige" el problema por lo que permanece en el sistema.

Página 368

Bloquear y esperar

Una vez que un hilo adquiere un recurso, no liberará el recurso hasta que haya adquirido todos de los demás recursos que necesita y ha completado su trabajo.

Sin preferencia

Un subproceso no puede quitar recursos de otro subproceso. Una vez que un hilo tiene un

recurso, la única forma de que otro subproceso lo obtenga es que el subproceso de retención lo libere.

Espera circular

Esto también se conoce como el abrazo mortal. Imagine dos subprocesos, T1 y T2, y dos recursos, R1 y R2. T1 tiene R1, T2 tiene R2. T1 también requiere R2 y T2 también requiere R1. Esto da algo como la Figura A-3:

Figura A-3

Las cuatro de estas condiciones deben cumplirse para que sea posible un punto muerto. Rompe cualquiera de estos condiciones y el interbloqueo no es posible.

Rompiendo la exclusión mutua

Una estrategia para evitar el estancamiento es eludir la condición de exclusión mutua. Tú podría ser capaz de hacer esto por

- Utilizar recursos que permitan el uso simultáneo, por ejemplo, AtomicInteger .
- Aumentar el número de recursos de manera que iguale o supere el número de recursos peting hilos.
- Verificar que todos sus recursos son gratuitos antes de apoderarse de alguno.

Desafortunadamente, la mayoría de los recursos son limitados y no permiten usar. Y no es raro que la identidad del segundo recurso se base en el resultados de operar en el primero. Pero no se desanime; quedan tres condiciones.

Página 369

Romper el bloqueo y esperar

También puede eliminar el punto muerto si se niega a esperar. Verifique cada recurso antes que usted agárrelo, libere todos los recursos y comience de nuevo si se encuentra con uno que está ocupado.

Este enfoque presenta varios problemas potenciales:

- Hambruna: un hilo sigue sin poder adquirir los recursos que necesita (tal vez tiene una combinación única de recursos que rara vez todos están disponibles).
- Livelock: varios subprocesos pueden sincronizarse y todos adquieren un recurso y luego suelte un recurso, una y otra vez. Esto es especialmente probable con simplistas Algoritmos de programación de CPU (piense en dispositivos integrados o escritos a mano simplistas algoritmos de equilibrio de subprocesos).

Ambos pueden causar un rendimiento deficiente. El primero da como resultado una baja utilización de la CPU, mientras que el segundo da como resultado un uso elevado e inútil de la CPU.

Tan ineficaz como suena esta estrategia, es mejor que nada. Tiene el beneficio de que casi siempre se puede implementar si todo lo demás falla.

Rompiendo la preferencia

Otra estrategia para evitar el punto muerto es permitir que los subprocesos quiten recursos de otros hilos. Por lo general, esto se hace a través de un mecanismo de solicitud simple. Cuando un hilo descubre que un recurso está ocupado, le pide al propietario que lo libere. Si el dueño también está esperando para algún otro recurso, los libera a todos y comienza de nuevo.

Esto es similar al enfoque anterior, pero tiene la ventaja de que se permite que un subproceso esperar un recurso. Esto reduce el número de reinicios. Tenga en cuenta, sin embargo, que administrar todas esas solicitudes puede ser complicado.

Rompiendo la espera circular

Este es el enfoque más común para prevenir el estancamiento. Para la mayoría de los sistemas requiere no es más que una simple convención acordada por todas las partes.

En el ejemplo anterior, el hilo 1 desea tanto el recurso 1 como el recurso 2 y El hilo 2 quiere tanto el recurso 2 como el recurso 1, simplemente forzando tanto el hilo 1 como el El hilo 2 para asignar recursos en el mismo orden hace que la espera circular sea imposible.

De manera más general, si todos los hilos pueden ponerse de acuerdo sobre un orden global de recursos y si todos asignan recursos en ese orden, entonces el interbloqueo es imposible. Como todas las otras estrategies, esto puede causar problemas:

- El orden de adquisición puede no corresponder con el orden de uso; por lo tanto un recurso adquirido al principio puede que no se utilice hasta el final. Esto puede hacer que los recursos sean bloqueado más de lo estrictamente necesario.

Página 370

- A veces no se puede imponer una orden a la adquisición de recursos. Si el ID de el segundo recurso proviene de una operación realizada en el primero, luego el pedido es no factible.

Por tanto, hay muchas formas de evitar el estancamiento. Algunos conducen a la inanición, mientras que otros hacer un uso intensivo de la CPU y reducir la capacidad de respuesta. ¡TANSTAAFL! [5]

Aislar la parte relacionada con el hilo de su solución para permitir el ajuste y la experimentación es una forma poderosa de obtener los conocimientos necesarios para determinar las mejores estrategias.

Prueba de código multiproceso

¿Cómo podemos escribir una prueba para demostrar que el siguiente código está roto?

```
01: clase pública ClassWithThreadingProblem {
02: int nextId;
03:
04: public int takeNextId () {
05:                 return nextId ++;
06:}
07:}
```

Aquí hay una descripción de una prueba que demostrará que el código está roto:

- Recuerde el valor actual de `nextId` .
- Cree dos subprocesos, los cuales llaman a `takeNextId ()` una vez.
- Verifique que `nextId` sea dos más de lo que comenzamos.
- Ejecute esto hasta que demostremos que `nextId` solo se incrementó en uno en su lugar de dos.

El Listado A-2 muestra una prueba de este tipo:

Listado A-2
ClassWithThreadingProblemTest.java

```
01: ejemplo de paquete;
02:
03: importar org.junit.Assert.fail estático;
04:
05: importar org.junit.Test;
06:
07: clase pública ClassWithThreadingProblemTest {
08: @Prueba
09: public void twoThreadsShouldFailEventually () lanza Exception {
10:                 final ClassWithThreadingProblem classWithThreadingProblem
                            = nuevo ClassWithThreadingProblem ();
11:
```

5. No existe tal cosa como un almuerzo gratis.

Página 371

Listado A-2 (continuación)

ClassWithThreadingProblemTest.java

```
12:                 Ejecutable ejecutable = new Ejecutable () {
13:                         public void run () {
14:                                 classWithThreadingProblem.takeNextId ();
15:                         }
dieciséis:          };
17:
18:                 para (int i = 0; i <50000; ++ i) {
19:                         int startId = classWithThreadingProblem.lastId;
20:                         int esperadoResultado = 2 + initialId;
21:
22:                         Thread t1 = new Thread (ejecutable);
23:                         Thread t2 = new Thread (ejecutable);
24:                         t1.start ();
25:                         t2.start ();
26:                         t1.join ();
27:                         t2.unir ();
28:
29:                         int endId = classWithThreadingProblem.lastId;
30:
31:                         si (ID de finalización! = Resultado esperado)
32:                                 regreso;
33:                 }
34:
35:                 fail ("Debería haber expuesto un problema de subprocesos, pero no fue así");
36:}
37:}
```

Línea	Descripción
10	Cree una única instancia de ClassWithThreadingProblem . Tenga en cuenta que debemos usar la palabra clave final porque la usamos a continuación en una clase interna anónima.
12-16	Cree una clase interna anónima que use la única instancia de ClassWithThreadingProblem .
18	Ejecute este código "suficientes" veces para demostrar que el código falló, pero no tanto que la prueba "lleva demasiado tiempo". Este es un acto de equilibrio; nosotros no quiere esperar demasiado para demostrar el fracaso. Elegir este número es difícil aunque más adelante veremos que podemos reducir mucho este número.
19	Recuerda el valor inicial. Esta prueba intenta demostrar que el código en ClassWithThreadingProblem está roto. Si esta prueba pasa, demostró que el el código estaba roto. Si esta prueba falla, la prueba no pudo demostrar que el código está roto.
20	Esperamos que el valor final sea dos más que el valor actual.
22-23	Cree dos subprocesos, los cuales utilizan el objeto que creamos en las líneas 12-16. Esto nos da el potencial de dos subprocesos que intentan usar nuestra única instancia de ClassWithThreadingProblem e interfiriendo entre sí.

Página 372

Línea	Descripción
24-25	Haga que nuestros dos subprocesos sean elegibles para ejecutarse.
26-27	Espere a que terminen ambos hilos antes de comprobar los resultados.
29	Registre el valor final real.
31–32	¿Nuestro finalId difiere de lo que esperábamos? Si es así, regrese y termine la prueba. hemos probado que el código está roto. Si no es así, inténtelo de nuevo.
35	Si llegamos hasta aquí, nuestra prueba no pudo probar que el código de producción estaba roto. ken en un período de tiempo "razonable"; nuestro código ha fallado. O el código no está roto o no ejecutamos suficientes iteraciones para obtener la condición de falla que se produzca.

Esta prueba ciertamente establece las condiciones para un problema de actualización simultánea. Sin emabargo, el problema ocurre con tan poca frecuencia que la gran mayoría de las veces esta prueba no lo detecta.

De hecho, para detectar realmente el problema, debemos establecer el número de iteraciones en más de uno. millón. Incluso entonces, en diez ejecuciones con un recuento de bucles de 1.000.000, se produjo el problema sólo una vez. Eso significa que probablemente deberíamos establecer el recuento de iteraciones en más de cien dred millones para obtener fallas confiables. ¿Cuánto tiempo estamos dispuestos a esperar?

Incluso si ajustamos la prueba para obtener fallas confiables en una máquina, probablemente tendremos para reajustar la prueba con diferentes valores para demostrar la falla en otra máquina, sistema operativo o versión de la JVM.

Y este es un problema *simple* . Si no podemos demostrar el código roto fácilmente con esto problema, ¿cómo vamos a detectar problemas realmente complejos?

Entonces, ¿qué enfoques podemos tomar para demostrar este simple fracaso? Y, más importante Ahora bien, ¿cómo podemos escribir pruebas que demuestren fallas en un código más complejo? Cómo ¿Seremos capaces de descubrir si nuestro código tiene fallas cuando no sabemos dónde buscar?

Aqui hay algunas ideas:

- **Pruebas de Monte Carlo.** Flexibilice las pruebas para que se puedan ajustar. Luego ejecuta la prueba y otra vez, digamos en un servidor de prueba, cambiando aleatoriamente los valores de ajuste. Si alguna vez las pruebas falla, el código está roto. Asegúrese de comenzar a redactar esas pruebas temprano para que una el servidor de integración comienza a ejecutarlos pronto. Por cierto, asegúrese de registrar cuidadosamente las condiciones en las que falló la prueba.
- Ejecute la prueba en cada una de las plataformas de implementación de destino. Repetidamente. Continu ously. Cuanto más tiempo se ejecuten las pruebas sin fallas, es más probable que
  - El código de producción es correcto o
  - Las pruebas no son adecuadas para exponer problemas.
- Ejecute las pruebas en una máquina con cargas variables. Si puede simular cargas cercanas a un entorno de producción, hágalo.

Sin embargo, incluso si hace todas estas cosas, todavía no tiene muchas posibilidades de encontrar: problemas de subprocesamiento con su código. Los problemas más insidiosos son los que tienen una sección transversal tan pequeña que solo ocurren una vez en mil millones de oportunidades. Semejante los problemas son el terror de los sistemas complejos.

Soporte de herramientas para probar código basado en subprocesos

IBM ha creado una herramienta llamada ConTest. [6] Instrumenta clases para hacer más probable que el código no seguro para subprocesos falla.

No tenemos ninguna relación directa con IBM o el equipo que desarrolló ConTest. Un colega nuestro nos lo indicó. Notamos una gran mejora en nuestra capacidad para encontrar problemas de subprocesos después de unos minutos de uso.

A continuación, se muestra un resumen de cómo utilizar ConTest:

- Escriba pruebas y código de producción, asegurándose de que haya pruebas diseñadas específicamente para simule múltiples usuarios bajo cargas variables, como se mencionó anteriormente.
- Prueba de instrumentos y código de producción con ConTest.
- Ejecute las pruebas.

Cuando instrumentamos el código con ConTest, nuestra tasa de éxito pasó de aproximadamente un error Ure en diez millones de iteraciones a aproximadamente una falla en *treinta* iteraciones. Aquí están los valores del bucle para varias ejecuciones de la prueba después de la instrumentación: 13, 23, 0, 54, 16, 14, 6, 69, 107, 49, 2. Entonces Claramente, las clases instrumentadas fallaron mucho antes y con mucha mayor confiabilidad.

Conclusión

Este capítulo ha sido una estancia muy breve a través del vasto y traicionero territorio de programación concurrente. Apenas arañamos la superficie. Nuestro énfasis aquí estaba en la disciplinas para ayudar a mantener limpio el código concurrente, pero hay mucho más que debe aprender si vas a escribir sistemas concurrentes. Le recomendamos que comience con Doug Lea's maravilloso libro *Programación concurrente en Java: Principios y patrones de diseño.* [7]

En este capítulo hablamos sobre la actualización concurrente y las disciplinas de sincronización limpia. cronización y bloqueo que pueden prevenirlo. Hablamos sobre cómo los subprocesos pueden mejorar la rendimiento de un sistema vinculado a E / S y mostró las técnicas limpias para lograr tal Mejoras. Hablamos sobre el estancamiento y las disciplinas para prevenirlo de forma limpia.

6. http://www.haifa.ibm.com/projects/verification/contest/index.html
7. Ver [Lea99] p. 191.

camino. Finalmente, hablamos de estrategias para exponer problemas concurrentes instrumentando tu codigo.

Tutorial: ejemplos de código completo

Cliente / servidor no subprocesado

Listado A-3

Server.java

```
paquete com.objectmentor.clientserver.nonthreaded;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

import common.MessageUtils;

El servidor de clase pública implementa Runnable {
    ServerSocket serverSocket;
    volátil booleano keepProcessing = true;

    Public Server (int port, int millisecondsTimeout) lanza IOException {
        serverSocket = nuevo ServerSocket (puerto);
        serverSocket.setSoTimeout (millisecondsTimeout);
    }

    public void run () {
        System.out.printf ("Inicio del servidor \ n");

        while (keepProcessing) {
            intentar {
                System.out.printf ("aceptando cliente \ n");
                Socket socket = serverSocket.accept ();
                System.out.printf ("cliente obtenido \ n");
                proceso (socket);
            } captura (Excepción e) {
                manejar (e);
            }
        }
    }

    identificador de vacío privado (Excepción e) {
        if (! (e instancia de SocketException)) {
            e.printStackTrace ();
        }
    }

    public void stopProcessing () {
        keepProcessing = falso;
        closeIgnoringException (serverSocket);
    }
```


Página 375

Listado A-3 (continuación)

Server.java

```
    proceso de vacío (socket socket) {
        si (enchufe == nulo)
            regreso;

        intentar {
            System.out.printf ("Servidor: recibiendo mensaje \ n");
            Mensaje de cadena = MessageUtils.getMessage (socket);
            System.out.printf ("Servidor: mensaje recibido:% s \ n", mensaje);
            Thread.sleep (1000);
            System.out.printf ("Servidor: enviando respuesta:% s \ n", mensaje);
            MessageUtils.sendMessage (socket, "Procesado:" + mensaje);
            System.out.printf ("Servidor: enviado \ n");
            closeIgnoringException (socket);
        } captura (Excepción e) {
            e.printStackTrace ();
        }
```

```
    }

    private void closeIgnoringException (Socket socket) {
        si (enchufe! = nulo)
            intentar {
                socket.close ();
            } catch (IOException ignore) {
            }
    }

    private void closeIgnoringException (ServerSocket serverSocket) {
        si (serverSocket! = null)
            intentar {
                serverSocket.close ();
            } catch (IOException ignore) {
            }
    }
}
```

Listado A-4

ClientTest.java

```
paquete com.objectmentor.clientserver.nonthreaded;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

import common.MessageUtils;

El servidor de clase pública implementa Runnable {
    ServerSocket serverSocket;
    volátil booleano keepProcessing = true;
```


Página 376

Listado A-4 (continuación)

ClientTest.java

```
    Public Server (int port, int millisecondsTimeout) lanza IOException {
        serverSocket = nuevo ServerSocket (puerto);
        serverSocket.setSoTimeout (millisecondsTimeout);
    }

    public void run () {
        System.out.printf ("Inicio del servidor \ n");

        while (keepProcessing) {
            intentar {
                System.out.printf ("aceptando cliente \ n");
                Socket socket = serverSocket.accept ();
                System.out.printf ("cliente obtenido \ n");
                proceso (socket);
            } captura (Excepción e) {
                manejar (e);
            }
        }
    }

    identificador de vacío privado (Excepción e) {
        if (! (e instancia de SocketException)) {
            e.printStackTrace ();
        }
    }

    public void stopProcessing () {
        keepProcessing = falso;
        closeIgnoringException (serverSocket);
    }

    proceso de vacío (socket socket) {
```

```
        si (enchufe == nulo)
            regreso;

        intentar {
            System.out.printf ("Servidor: recibiendo mensaje \ n");
            Mensaje de cadena = MessageUtils.getMessage (socket);
            System.out.printf ("Servidor: mensaje recibido:% s \ n", mensaje);
            Thread.sleep (1000);
            System.out.printf ("Servidor: enviando respuesta:% s \ n", mensaje);
            MessageUtils.sendMessage (socket, "Procesado:" + mensaje);
            System.out.printf ("Servidor: enviado \ n");
            closeIgnoringException (socket);
        } captura (Excepción e) {
            e.printStackTrace ();
        }

    }

    private void closeIgnoringException (Socket socket) {
        si (enchufe! = nulo)
            intentar {
                socket.close ();
```


Página 377

Listado A-4 (continuación)

ClientTest.java

```
                } catch (IOException ignore) {
                }
    }

    private void closeIgnoringException (ServerSocket serverSocket) {
        si (serverSocket! = null)
            intentar {
                serverSocket.close ();
            } catch (IOException ignore) {
            }
    }
}
```

Listado A-5

MessageUtils.java

```
paquete común;

import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.net.Socket;

MessageUtils de clase pública {
    public static void sendMessage (Socket socket, String mensaje)
            lanza IOException {
        Flujo OutputStream = socket.getOutputStream ();
        ObjectOutputStream oos = nuevo ObjectOutputStream (flujo);
        oos.writeUTF (mensaje);
        oos.flush ();
    }

    public static String getMessage (Socket socket) lanza IOException {
        InputStream stream = socket.getInputStream ();
        ObjectInputStream ois = new ObjectInputStream (flujo);
        return ois.readUTF ();
    }
}
```

Cliente / servidor que usa subprocesos

Cambiar el servidor para usar subprocesos simplemente requiere un cambio en el mensaje de proceso (nuevo las líneas se enfatizan para destacar):

```
proceso nulo (zócalo de enchufe final) {
    si (enchufe == nulo)
        regreso;

    ClientHandler ejecutable = new Runnable () {
        public void run () {
```


Página 378


```
            intentar {
                System.out.printf ("Servidor: recibiendo mensaje \ n");
                Mensaje de cadena = MessageUtils.getMessage (socket);
                System.out.printf ("Servidor: mensaje recibido:% s \ n", mensaje);
                Thread.sleep (1000);
                System.out.printf ("Servidor: enviando respuesta:% s \ n", mensaje);
                MessageUtils.sendMessage (socket, "Procesado:" + mensaje);
                System.out.printf ("Servidor: enviado \ n");
                closeIgnoringException (socket);
            } captura (Excepción e) {
                e.printStackTrace ();
            }
        }
    };

    Thread clientConnection = nuevo Thread (clientHandler);
    clientConnection.start ();
}
```

Página 379

Esta página se dejó en blanco intencionalmente

Página 380

apéndice B

org.jfree.date.SerialDate

Listado B-1

SerialDate.Java

```
1 / * ========== ==========
2 * JCommon: una biblioteca de clases de propósito general gratuita para la plataforma Java (tm)
3 * ========== ==========
4 *
5 * (C) Copyright 2000-2005, por Object Refinery Limited y contribuyentes.
6 *
7 * Información del proyecto: http://www.jfree.org/jcommon/index.html
8 *
9 * Esta biblioteca es software gratuito; puedes redistribuirlo y / o modificarlo
10 * bajo los términos de la Licencia Pública General Reducida GNU publicada por
11 * la Free Software Foundation; ya sea la versión 2.1 de la Licencia, o
12 * (a su elección) cualquier versión posterior.
13 *
14 * Esta biblioteca se distribuye con la esperanza de que sea útil, pero
15 * SIN NINGUNA GARANTÍA; incluso sin la garantía implícita de COMERCIABILIDAD
16 * o APTITUD PARA UN PROPÓSITO DETERMINADO. Ver el público general menor de GNU
17 * Licencia para más detalles.
18 *
19 * Debería haber recibido una copia de GNU Lesser General Public
20 * Licencia junto con esta biblioteca; si no es así, escriba al software libre
21 * Foundation, Inc., 51 Franklin Street, quinto piso, Boston, MA 02110-1301,
22 * Estados Unidos.
23 *
24 * [Java es una marca comercial o una marca comercial registrada de Sun Microsystems, Inc.
25 * en los Estados Unidos y otros países.]
26 *
27 * ----------
28 * SerialDate.java
29 * ----------
30 * (C) Copyright 2001-2005, por Object Refinery Limited.
31 *
32 * Autor original: David Gilbert (para Object Refinery Limited);
33 * Colaborador (es): -;
34 *
35 * $ Id: SerialDate.java, v 1.7 2005/11/03 09:25:17 mungady Exp $
36 *
37 * Cambios (desde el 11 de octubre de 2001)
```


Página 381

Listado B-1 (continuación)

SerialDate.Java

```
38 * ----------
39 * 11-Oct-2001: Reorganizó la clase y la movió a un nuevo paquete.
40 * com.jrefinery.date (DG);
41 * 05-Nov-2001: Se agregó un método getDescription () y se eliminó NotableDate
42 * clase (DG);
43 * 12-Nov-2001: IBD requiere el método setDescription (), ahora que NotableDate
44 * la clase se ha ido (DG); Se cambió getPreviousDayOfWeek (),
45 * getFollowingDayOfWeek () y getNearestDayOfWeek () para corregir
46 * errores (DG);
47 * 05-Dec-2001: Se corrigió un error en la clase SpreadsheetDate (DG);
```

```
48 * 29 de mayo de 2002: se movieron las constantes de mes a una interfaz separada
49 * (MonthConstants) (DG);
50 * 27-Ago-2002: Se corrigió un error en el método addMonths (), gracias a N ??? levka Petr (DG);
51 * 03-Oct-2002: Errores corregidos reportados por Checkstyle (DG);
52 * 13-Mar-2003: Implementado Serializable (DG);
53 * 29 de mayo de 2003: Se corrigió un error en el método addMonths (DG);
54 * 04-Sep-2003: Implementado comparable. Se actualizó el isInRange javadocs (DG);
55 * 05-Jan-2005: Se corrigió un error en el método addYears () (1096282) (DG);
56 *
57 * /
58
59 paquete org.jfree.date;
60
61 import java.io.Serializable;
62 import java.text.DateFormatSymbols;
63 import java.text.SimpleDateFormat;
64 import java.util.Calendar;
65 import java.util.GregorianCalendar;
66
67 / **
68 * Una clase abstracta que define nuestros requisitos para manipular fechas,
69 * sin atar una implementación en particular.
70 * <P>
71 * Requisito 1: igualar al menos lo que hace Excel para las fechas;
72 * Requisito 2: la clase es inmutable;
73 * <P>
74 * ¿Por qué no usar java.util.Date? Lo haremos, cuando tenga sentido. A veces,
75 * java.util.Date puede ser * demasiado * preciso: representa un instante en el tiempo,
76 * con una precisión de 1/1000 de segundo (con la fecha en sí dependiendo de la
77 * zona horaria). A veces solo queremos representar un día en particular (por ejemplo, 21
78 * enero de 2015) sin preocuparnos por la hora del día, o el
79 * zona horaria, o cualquier otra cosa. Para eso hemos definido SerialDate.
80 * <P>
81 * Puede llamar a getInstance () para obtener una subclase concreta de SerialDate,
82 * sin preocuparse por la implementación exacta.
83 *
84 * @autor David Gilbert
85 * /
86 clase pública abstracta SerialDate implementa Comparable,
87                                                 Serializable,
88                                                 MonthConstants {
89
90 / ** Para serialización. * /
91 private static final long serialVersionUID = -2937160404674236 37L;
92
93 / ** Símbolos de formato de fecha. * /
94 público estático final DateFormatSymbols
95                 DATE_FORMAT_SYMBOLS = nuevo SimpleDateFormat (). GetDateFormatSymbols ();
96
97 / ** El número de serie del 1 de enero de 1900. * /
98 public static final int SERIAL_LOWER_BOUND = 2;
99
100 / ** El número de serie del 31 de diciembre de 9999. * /
101 public static final int SERIAL_UPPER_BOUND = 2958465;
102
```


Página 382

Listado B-1 (continuación)

SerialDate.Java

```
103 / ** El valor de año más bajo admitido por este formato de fecha. * /
104 público estático final int MINIMUM_YEAR_SUPPORTED = 1900;
105
106 / ** El valor de año más alto admitido por este formato de fecha. * /
107 público estático final int MAXIMUM_YEAR_SUPPORTED = 9999;
108
109 / ** Constante útil para el lunes. Equivalente a java.util.Calendar.MONDAY. * /
110 public static final int MONDAY = Calendar.MONDAY;
111
112 / **
113 * Constante útil para el martes. Equivalente a java.util.Calendar.TUESDAY.
114 * /
115 public static final int MARTES = Calendario.TUESDAY;
116
117 / **
118 * Constante útil para el miércoles. Equivalente a
119 * java.util.Calendar.WEDNESDAY.
120 * /
121 public static final int WEDNESDAY = Calendar.WEDNESDAY;
122
123 / **
124 * Constante útil para el jueves. Equivalente a java.util.Calendar.THURSDAY.
125 * /
126 public static final int JUEVES = Calendario.JUEVES;
127
128 / ** Constante útil para el viernes. Equivalente a java.util.Calendar.FRIDAY. * /
129 public static final int VIERNES = Calendario.VIERNES;
130
```

```
131 / **
132 * Constante útil para el sábado. Equivalente a java.util.Calendar.SATURDAY.
133 * /
134 public static final int SATURDAY = Calendar.SATURDAY;
135
136 / ** Constante útil para el domingo. Equivalente a java.util.Calendar.SUNDAY. * /
137 public static final int DOMINGO = Calendario.SUNDAY;
138
139 / ** El número de días de cada mes en años no bisiestos. * /
140 estático final int [] LAST_DAY_OF_MONTH =
141 {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
142
143 / ** El número de días en un año (no bisiesto) hasta el final de cada mes. * /
144 estático final int [] AGGREGATE_DAYS_TO_END_OF_MONTH =
145 {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
146
147 / ** El número de días en un año hasta el final del mes anterior. * /
148 static final int [] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
149 {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
150
151 / ** El número de días en un año bisiesto hasta el final de cada mes. * /
152 estático final int [] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH =
153 {0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
154
155 / **
156 * El número de días en un año bisiesto hasta el final del mes anterior.
157 * /
158 estática final int []
159 LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
160 {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
161
162 / ** Una constante útil para referirse a la primera semana de un mes. * /
163 public static final int FIRST_WEEK_IN_MONTH = 1;
164
```


Página 383

Listado B-1 (continuación)

SerialDate.Java

```
165 / ** Una constante útil para referirse a la segunda semana de un mes. * /
166 public static final int SECOND_WEEK_IN_MONTH = 2;
167
168 / ** Una constante útil para referirse a la tercera semana de un mes. * /
169 public static final int THIRD_WEEK_IN_MONTH = 3;
170
171 / ** Una constante útil para referirse a la cuarta semana de un mes. * /
172 público estático final int FOURTH_WEEK_IN_MONTH = 4;
173
174 / ** Una constante útil para referirse a la última semana de un mes. * /
175 público estático final int LAST_WEEK_IN_MONTH = 0;
176
177 / ** Constante de rango útil. * /
178 public static final int INCLUDE_NONE = 0;
179
180 / ** Constante de rango útil. * /
181 public static final int INCLUDE_FIRST = 1;
182
183 / ** Constante de rango útil. * /
184 public static final int INCLUDE_SECOND = 2;
185
186 / ** Constante de rango útil. * /
187 public static final int INCLUDE_BOTH = 3;
188
189 / **
190 * Constante útil para especificar un día de la semana relativo a un fijo
191 * fecha.
192 * /
193 public static final int PRECEDING = -1;
194
195 / **
196 * Constante útil para especificar un día de la semana relativo a un fijo
197 * fecha.
198 * /
199 public static final int NEAREST = 0;
200
201 / **
202 * Constante útil para especificar un día de la semana relativo a un fijo
203 * fecha.
204 * /
205 public static final int FOLLOWING = 1;
206
207 / ** Una descripción de la fecha. * /
208 descripción de cadena privada;
209
210 / **
```

```
211 * Constructor predeterminado.
212 * /
213 protegido SerialDate () {
214}
215
216 / **
217 * Devuelve <code> true </code> si el código entero proporcionado representa un
218 * día de la semana válido y <code> false </code> en caso contrario.
219 *
220 * @param codifica el código que se está verificando para verificar su validez.
221 *
222 * @return <code> true </code> si el código entero proporcionado representa un
223 * día de la semana válido y <code> false </code> de lo contrario.
224 * /
225 public static boolean isValidWeekdayCode (código int final) {
226
```


Página 384

Listado B-1 (continuación)

SerialDate.Java

```
227 cambiar (código) {
228 caso DOMINGO:
229 caso LUNES:
230 caso MARTES:
231 caso MIÉRCOLES:
232 caso JUEVES:
233 caso VIERNES:
234 caso SÁBADO:
235 devuelve verdadero;
236 defecto:
237 falso retorno;
238 }
239
240}
241
242 / **
243 * Convierte la cadena proporcionada en un día de la semana.
244 *
245 * @param es una cadena que representa el día de la semana.
246 *
247 * @return <code> -1 </code> si la cadena no es convertible, el día de
248 * la semana de lo contrario.
249 * /
250 public static int stringToWeekdayCode (String s) {
251
252 Cadena final [] shortWeekdayNames
253 = DATE_FORMAT_SYMBOLS.getShortWeekdays ();
254 Cadena final [] weekDayNames = DATE_FORMAT_SYMBOLS.getWeekdays ();
255
256 int resultado = -1;
257 s = s.trim ();
258 for (int i = 0; i <weekDayNames.length; i ++) {
259 if (es igual a (shortWeekdayNames [i])) {
260 resultado = i;
261 rotura;
262 }
263 if (es igual a (weekDayNames [i])) {
264 resultado = i;
265 rotura;
266 }
267 }
268 devolver resultado;
269
270}
271
272 / **
273 * Devuelve una cadena que representa el día de la semana proporcionado.
274 * <P>
275 * Necesidad de encontrar un enfoque mejor.
276 *
277 * @param weekday el día de la semana.
278 *
279 * @return una cadena que representa el día de la semana proporcionado.
280 * /
281 public static String weekdayCodeToString (final int día de la semana) {
282
283 Cadena final [] días de la semana = DATE_FORMAT_SYMBOLS.getWeekdays ();
284 volver entre semana [día de la semana];
285
286}
287
288 / **
```

Página 385

Listado B-1 (continuación)

SerialDate.Java

```
289 * Devuelve una matriz de nombres de meses.
290 *
291 * @return una matriz de nombres de meses.
292 * /
293 cadena estática pública [] getMonths () {
294
295                 return getMonths (falso);
296
297}
298
299 / **
300 * Devuelve una matriz de nombres de meses.
301 *
302 * @param acortó una bandera que indica que los nombres de meses acortados deben
303 *                                          ser devuelto.
304 *
305 * @return una matriz de nombres de meses.
306 * /
307 public static String [] getMonths (booleano final acortado) {
308
309                 if (abreviado) {
310                         return DATE_FORMAT_SYMBOLS.getShortMonths ();
311                 }
312                 demás {
313                         return DATE_FORMAT_SYMBOLS.getMonths ();
314                 }
315
316}
317
318 / **
319 * Devuelve verdadero si el código entero proporcionado representa un mes válido.
320 *
321 * @param code el código que se está comprobando para verificar su validez.
322 *
323 * @return <code> true </code> si el código entero proporcionado representa un
324 *                          mes válido.
325 * /
326 public static boolean isValidMonthCode (código int final) {
327
328                 cambiar (código) {
329                         caso ENERO:
330                         caso FEBRERO:
331                         caso MARZO:
332                         caso ABRIL:
333                         caso PUEDE:
334                         caso JUNIO:
335                         caso JULIO:
336                         caso AGOSTO:
337                         caso SEPTIEMBRE:
338                         caso OCTUBRE:
339                         caso NOVIEMBRE:
340                         caso DICIEMBRE:
341                                 devuelve verdadero;
342                         defecto:
343                                 falso retorno;
344                 }
345
346}
347
348 / **
349 * Devuelve el trimestre del mes especificado.
350 *
```


Página 386

Listado B-1 (continuación)

SerialDate.Java

```
351 * @param codifica el código del mes (1-12).
352 *
353 * @return el trimestre al que pertenece el mes.
354 * @throws java.lang.IllegalArgumentException
355 * /
356 public static int monthCodeToQuarter (código int final) {
357
358                 cambiar (código) {
359                         caso ENERO:
360                         caso FEBRERO:
361                         caso MARZO: retorno 1;
362                         caso ABRIL:
363                         caso PUEDE:
364                         caso JUNIO: retorno 2;
365                         caso JULIO:
366                         caso AGOSTO:
367                         caso SEPTIEMBRE: retorno 3;
368                         caso OCTUBRE:
369                         caso NOVIEMBRE:
370                         caso DICIEMBRE: retorno 4;
371                         predeterminado: lanzar una nueva IllegalArgumentException (
372                                 "SerialDate.monthCodeToQuarter: código de mes no válido.");
373                 }
374
375}
376
377 / **
378 * Devuelve una cadena que representa el mes proporcionado.
379 * <P>
380 * La cadena devuelta es la forma larga del nombre del mes tomado del
381 * configuración regional predeterminada.
382 *
383 * @param mes el mes.
384 *
385 * @return una cadena que representa el mes proporcionado.
386 * /
387 public static String monthCodeToString (mes int final) {
388
389                 return monthCodeToString (mes, falso);
390
391}
392
393 / **
394 * Devuelve una cadena que representa el mes proporcionado.
395 * <P>
396 * La cadena devuelta es la forma larga o corta del nombre del mes tomado
397 * de la configuración regional predeterminada.
398 *
399 * @param mes el mes.
400 * @param acortado si <code> true </code> devuelve la abreviatura del
401 *                                                 mes.
402 *
403 * @return una cadena que representa el mes proporcionado.
404 * @throws java.lang.IllegalArgumentException
405 * /
406 public static String monthCodeToString (mes int final,
407                                                                                 booleano final acortado) {
408
409                 // comprobar argumentos ...
410                 if (! isValidMonthCode (mes)) {
411                         lanzar una nueva IllegalArgumentException (
412                                 "SerialDate.monthCodeToString: mes fuera del rango válido.");
```


Página 387

Listado B-1 (continuación)

SerialDate.Java

```
413                 }
414
```

```
415             cadena final [] meses;
416
417             if (abreviado) {
418                 meses = DATE_FORMAT_SYMBOLS.getShortMonths ();
419             }
420             demás {
421                 meses = DATE_FORMAT_SYMBOLS.getMonths ();
422             }
423
424             meses de retorno [mes - 1];
425
426}
427
428 / **
429 * Convierte una cadena en un código de mes.
430 * <P>
431 * Este método devolverá una de las constantes ENERO, FEBRERO, ...,
432 * DICIEMBRE que corresponde a la cadena. Si la cuerda no es
433 * reconocido, este método devuelve -1.
434 *
435 * @param es la cadena a analizar.
436 *
437 * @return <code> -1 </code> si la cadena no se puede analizar, el mes del
438 *                   año de lo contrario.
439 * /
440 public static int stringToMonthCode (String s) {
441
442             Cadena final [] shortMonthNames = DATE_FORMAT_SYMBOLS.getShortMonths ();
443             Cadena final [] monthNames = DATE_FORMAT_SYMBOLS.getMonths ();
444
445             int resultado = -1;
446             s = s.trim ();
447
448             // primero intente analizar la cadena como un entero (1-12) ...
449             intentar {
450                 resultado = Integer.parseInt (s);
451             }
452             catch (NumberFormatException e) {
453                 // suprimir
454             }
455
456             // ahora busca los nombres de los meses ...
457             if ((resultado <1) || (resultado> 12)) {
458                 for (int i = 0; i <monthNames.length; i ++) {
459                     if (es igual a (shortMonthNames [i])) {
460                         resultado = i + 1;
461                         rotura;
462                     }
463                     if (es igual a (monthNames [i])) {
464                         resultado = i + 1;
465                         rotura;
466                     }
467                 }
468             }
469
470             devolver resultado;
471
472}
473
474 / **
```


Página 388

Listado B-1 (continuación)

SerialDate.Java

```
475 * Devuelve verdadero si el código entero proporcionado representa un valor válido.
476 * semana-del-mes, y falso en caso contrario.
477 *
478 * @param code el código que se está comprobando para verificar su validez.
479 * @return <code> true </code> si el código entero proporcionado representa un
480 *                   semana válida del mes.
481 * /
482 public static boolean isValidWeekInMonthCode (código int final) {
483
484             cambiar (código) {
485                 caso FIRST_WEEK_IN_MONTH:
486                 caso SECOND_WEEK_IN_MONTH:
487                 caso THIRD_WEEK_IN_MONTH:
488                 caso FOURTH_WEEK_IN_MONTH:
489                 case LAST_WEEK_IN_MONTH: devuelve verdadero;
490                 predeterminado: devuelve falso;
491             }
492
493}
494
```

```
495 / **
496 * Determina si el año especificado es bisiesto o no.
497 *
498 * @param aaaa el año (en el rango de 1900 a 9999).
499 *
500 * @return <code> true </code> si el año especificado es bisiesto.
501 * /
502 public static boolean isLeapYear (final int yyyy) {
503
504 si ((yyyy% 4)! = 0) {
505 falso retorno;
506 }
507 más si ((yyyy% 400) == 0) {
508 devuelve verdadero;
509 }
510 más si ((yyyy% 100) == 0) {
511 falso retorno;
512 }
513 demás {
514 devuelve verdadero;
515 }
516
517}
518
519 / **
520 * Devuelve el número de años bisiestos desde 1900 hasta el año especificado
521 * INCLUIDO.
522 * <P>
523 * Tenga en cuenta que 1900 no es un año bisiesto.
524 *
525 * @param aaaa el año (en el rango de 1900 a 9999).
526 *
527 * @return el número de años bisiestos desde 1900 hasta el año especificado.
528 * /
529 public static int leapYearCount (final int aaaa) {
530
531 salto int final4 = (aaaa - 1896) / 4;
532 final int leap100 = (aaaa - 1800) / 100;
533 final int leap400 = (aaaa - 1600) / 400;
534 return leap4 - leap100 + leap400;
535
536}
```


Página 389

Listado B-1 (continuación)

SerialDate.Java

```
537
538 / **
539 * Devuelve el número del último día del mes, teniendo en cuenta
540 * años bisiestos.
541 *
542 * @param mes el mes.
543 * @param aaaa el año (en el rango de 1900 a 9999).
544 *
545 * @return el número del último día del mes.
546 * /
547 public static int lastDayOfMonth (final int mes, final int aaaa) {
548
549 resultado int final = LAST_DAY_OF_MONTH [mes];
550 if (mes! = FEBRERO) {
551 devolver resultado;
552 }
553 else if (isLeapYear (yyyy)) {
554 devuelve resultado + 1;
555 }
556 demás {
557 devolver resultado;
558 }
559
560}
561
562 / **
563 * Crea una nueva fecha agregando el número especificado de días a la base
564 * fecha.
565 *
566 * @param days la cantidad de días que se agregarán (puede ser negativo).
567 * @param base la fecha base.
568 *
569 * @return una nueva fecha.
570 * /
571 public static SerialDate addDays (final int days, final SerialDate base) {
572
573 final int serialDayNumber = base.toSerial () + días;
574 return SerialDate.createInstance (serialDayNumber);
```

```
575
576}
577
578 / **
579 * Crea una nueva fecha agregando el número especificado de meses a la base
580 * fecha.
581 * <P>
582 * Si la fecha base está cerca del final del mes, el día del resultado
583 * se puede ajustar ligeramente: 31 de mayo + 1 mes = 30 de junio.
584 *
585 * @param meses la cantidad de meses que se deben agregar (puede ser negativo).
586 * @param base la fecha base.
587 *
588 * @return una nueva fecha.
589 * /
590 public static SerialDate addMonths (últimos meses int,
591                                                                 base de SerialDate final) {
592
593                 int final yy = (12 * base.getYYYY () + base.getMonth () + meses - 1)
594                                                 / 12;
595                 final int mm = (12 * base.getYYYY () + base.getMonth () + meses - 1)
596                                                 % 12 + 1;
597                 final int dd = Math.min (
598                         base.getDayOfMonth (), SerialDate.lastDayOfMonth (mm, aa)
```


Página 390

Listado B-1 (continuación)

SerialDate.Java

```
599                 );
600                 return SerialDate.createInstance (dd, mm, aa);
601
602}
603
604 / **
605 * Crea una nueva fecha agregando el número especificado de años a la base
606 * fecha.
607 *
608 * @param años el número de años a agregar (puede ser negativo).
609 * @param base la fecha base.
610 *
611 * @return Una nueva fecha.
612 * /
613 public static SerialDate addYears (años int finales, base final de SerialDate) {
614
615                 final int baseY = base.getYYYY ();
616                 final int baseM = base.getMonth ();
617                 final int baseD = base.getDayOfMonth ();
618
619                 final int targetY = baseY + años;
620                 final int targetD = Math.min (
621                         baseD, SerialDate.lastDayOfMonth (baseM, targetY)
622                 );
623
624                 return SerialDate.createInstance (targetD, baseM, targetY);
625
626}
627
628 / **
629 * Devuelve la última fecha que cae en el día de la semana especificado y
630 * es ANTES de la fecha base.
631 *
632 * @param targetWeekday un código para el día de la semana objetivo.
633 * @param base la fecha base.
634 *
635 * @return la última fecha que cae en el día de la semana especificado y
636 *                                 es ANTES de la fecha base.
637 * /
638 public static SerialDate getPreviousDayOfWeek (final int targetWeekday,
639                                                                                 base de SerialDate final) {
640
641                 // comprobar argumentos ...
642                 if (! SerialDate.isValidWeekdayCode (targetWeekday)) {
643                         lanzar una nueva IllegalArgumentException (
644                                 "Código de día de la semana no válido".
645                         );
646                 }
647
648                 // busca la fecha ...
649                 ajuste int final;
650                 final int baseDOW = base.getDayOfWeek ();
651                 if (baseDOW> targetWeekday) {
652                         ajustar = Math.min (0, targetWeekday - baseDOW);
653                 }
654                 demás {
```

```
655                 ajustar = -7 + Math.max (0, targetWeekday - baseDOW);
656             }
657
658             return SerialDate.addDays (ajustar, base);
659
660}
```


Página 391

Listado B-1 (continuación)

SerialDate.Java

```
661
662 / **
663 * Devuelve la fecha más temprana que cae en el día de la semana especificado
664 * y es DESPUÉS de la fecha base.
665 *
666 * @param targetWeekday un código para el día de la semana objetivo.
667 * @param base la fecha base.
668 *
669 * @return la fecha más temprana que cae en el día de la semana especificado
670 *                      y es DESPUÉS de la fecha base.
671 * /
672 public static SerialDate getFollowingDayOfWeek (final int targetWeekday,
673                                                   base de SerialDate final) {
674
675             // comprobar argumentos ...
676             if (! SerialDate.isValidWeekdayCode (targetWeekday)) {
677                 lanzar una nueva IllegalArgumentException (
678                     "Código de día de la semana no válido".
679                 );
680             }
681
682             // busca la fecha ...
683             ajuste int final;
684             final int baseDOW = base.getDayOfWeek ();
685             if (baseDOW> targetWeekday) {
686                 ajustar = 7 + Math.min (0, targetWeekday - baseDOW);
687             }
688             demás {
689                 ajustar = Math.max (0, targetWeekday - baseDOW);
690             }
691
692             return SerialDate.addDays (ajustar, base);
693}
694
695 / **
696 * Devuelve la fecha que cae en el día de la semana especificado y es
697 * MÁS CERCA de la fecha base.
698 *
699 * @param targetDOW un código para el día de la semana objetivo.
700 * @param base la fecha base.
701 *
702 * @return la fecha que cae en el día de la semana especificado y es
703 *                      MÁS CERCA de la fecha base.
704 * /
705 public static SerialDate getNearestDayOfWeek (final int targetDOW,
706                                                 base de SerialDate final) {
707
708             // comprobar argumentos ...
709             if (! SerialDate.isValidWeekdayCode (targetDOW)) {
710                 lanzar una nueva IllegalArgumentException (
711                     "Código de día de la semana no válido".
712                 );
713             }
714
715             // busca la fecha ...
716             final int baseDOW = base.getDayOfWeek ();
717             int ajustar = -Math.abs (targetDOW - baseDOW);
718             si (ajustar> = 4) {
719                 ajustar = 7 - ajustar;
720             }
721             si (ajustar <= -4) {
722                 ajustar = 7 + ajustar;
```

Página 392

Listado B-1 (continuación)

SerialDate.Java

```
723                 }
724                 return SerialDate.addDays (ajustar, base);
725
726}
727
728 / **
729 * Avanza la fecha hasta el último día del mes.
730 *
731 * @param base la fecha base.
732 *
733 * @return una nueva fecha de serie.
734 * /
735 public SerialDate getEndOfCurrentMonth (base final de SerialDate) {
736                 final int last = SerialDate.lastDayOfMonth (
737                         base.getMonth (), base.getYYYY ()
738                 );
739                 return SerialDate.createInstance (último, base.getMonth (), base.getYYYY ());
740}
741
742 / **
743 * Devuelve una cadena correspondiente al código de la semana del mes.
744 * <P>
745 * Necesidad de encontrar un enfoque mejor.
746 *
747 * @param cuenta un código entero que representa la semana del mes.
748 *
749 * @return una cadena correspondiente al código de la semana del mes.
750 * /
751 cadena estática pública weekInMonthToString (recuento int final) {
752
753                 cambiar (contar) {
754                         case SerialDate.FIRST_WEEK_IN_MONTH: return "Primero";
755                         case SerialDate.SECOND_WEEK_IN_MONTH: return "Segundo";
756                         case SerialDate.THIRD_WEEK_IN_MONTH: return "Tercero";
757                         case SerialDate.FOURTH_WEEK_IN_MONTH: return "Cuarto";
758                         case SerialDate.LAST_WEEK_IN_MONTH: return "Último";
759                         defecto :
760                                 return "SerialDate.weekInMonthToString (): código inválido.";
761                 }
762
763}
764
765 / **
766 * Devuelve una cadena que representa el 'relativo' proporcionado.
767 * <P>
768 * Necesidad de encontrar un enfoque mejor.
769 *
770 * @param relativo una constante que representa el 'relativo'.
771 *
772 * @return una cadena que representa el 'relativo' proporcionado.
773 * /
774 public static String relatedToString (final int relativo) {
775
776                 switch (relativo) {
777                         case SerialDate.PRECEDING: return "Preceding";
778                         case SerialDate.NEAREST: return "Más cercano";
779                         case SerialDate.FOLLOWING: return "Siguiente";
780                         predeterminado: return "ERROR: Relativo a la cadena";
781                 }
782
783}
784
```


Página 393

Listado B-1 (continuación)

SerialDate.Java

```
785 / **
786 * Método de fábrica que devuelve una instancia de alguna subclase concreta de
787 * {@link SerialDate}.
788 *
789 * @param day the day (1-31).
790 * @param mes el mes (1-12).
791 * @param aaaa el año (en el rango de 1900 a 9999).
792 *
793 * @return Una instancia de {@link SerialDate}.
794 * /
795 public static SerialDate createInstance (final int day, final int mes,
796                                                                 int final aaaa) {
797                 return new SpreadsheetDate (día, mes, aaaa);
798}
799
800 / **
801 * Método de fábrica que devuelve una instancia de alguna subclase concreta de
802 * {@link SerialDate}.
803 *
804 * @param serial el número de serie del día (1 de enero de 1900 = 2).
805 *
806 * @return una instancia de SerialDate.
807 * /
808 public static SerialDate createInstance (final int serial) {
809                 return new SpreadsheetDate (serial);
810}
811
812 / **
813 * Método de fábrica que devuelve una instancia de una subclase de SerialDate.
814 *
815 * @param date Un objeto de fecha Java.
816 *
817 * @return una instancia de SerialDate.
818 * /
819 public static SerialDate createInstance (fecha final de java.util.Date) {
820
821                 calendario GregorianCalendar final = new GregorianCalendar ();
822                 calendar.setTime (fecha);
823                 return new SpreadsheetDate (calendar.get (Calendar.DATE),
824                                                                 calendar.get (Calendar.MONTH) + 1,
825                                                                 calendar.get (Calendar.YEAR));
826
827}
828
829 / **
830 * Devuelve el número de serie de la fecha, donde 1 de enero de 1900 = 2 (este
831 * corresponde, casi, al sistema de numeración utilizado en Microsoft Excel para
832 * Windows y Lotus 1-2-3).
833 *
834 * @ devuelve el número de serie de la fecha.
835 * /
836 public abstract int toSerial ();
837
838 / **
839 * Devuelve un java.util.Date. Dado que java.util.Date tiene más precisión que
840 * SerialDate, necesitamos definir una convención para la 'hora del día'.
841 *
842 * @ devuelva esto como <code> java.util.Date </code>.
843 * /
844 resumen público java.util.Date toDate ();
845
846 / **
```


Página 394

Listado B-1 (continuación)

SerialDate.Java

```
847 * Devuelve una descripción de la fecha.
848 *
849 * @return una descripción de la fecha.
850 * /
851 public String getDescription () {
852                 devuelva esta descripción;
853}
854
855 / **
856 * Establece la descripción de la fecha.
857 *
858 * @param description la nueva descripción de la fecha.
```

```
859 * /
860 public void setDescription (descripción final de la cadena) {
861                 this.description = descripción;
862}
863
864 / **
865 * Convierte la fecha en una cadena.
866 *
867 * @return una representación de cadena de la fecha.
868 * /
869 public String toString () {
870                 return getDayOfMonth () + "-" + SerialDate.monthCodeToString (getMonth ())
871                                                 + "-" + getYYYY ();
872}
873
874 / **
875 * Devuelve el año (suponga un rango válido de 1900 a 9999).
876 *
877 * @return el año.
878 * /
879 public abstract int getYYYY ();
880
881 / **
882 * Devuelve el mes (enero = 1, febrero = 2, marzo = 3).
883 *
884 * @return el mes del año.
885 * /
886 public abstract int getMonth ();
887
888 / **
889 * Devuelve el día del mes.
890 *
891 * @return el día del mes.
892 * /
893 public abstract int getDayOfMonth ();
894
895 / **
896 * Devuelve el día de la semana.
897 *
898 * @return el día de la semana.
899 * /
900 public abstract int getDayOfWeek ();
901
902 / **
903 * Devuelve la diferencia (en días) entre esta fecha y la especificada
904 * 'otra' fecha.
905 * <P>
906 * El resultado es positivo si esta fecha es posterior a la fecha 'otra' y
907 * negativo si es anterior a la "otra" fecha.
908 *
```


Página 395

Listado B-1 (continuación)

SerialDate.Java

```
909 * @param other la fecha con la que se compara.
910 *
911 * @return la diferencia entre esta y la otra fecha.
912 * /
913 public abstract int compare (SerialDate otro);
914
915 / **
916 * Devuelve verdadero si este SerialDate representa la misma fecha que el
917 * Fecha de serie especificada.
918 *
919 * @param other la fecha con la que se compara.
920 *
921 * @return <code> true </code> si este SerialDate representa la misma fecha que
922 *                 el SerialDate especificado.
923 * /
924 public abstract boolean isOn (SerialDate otro);
925
926 / **
927 * Devuelve verdadero si este SerialDate representa una fecha anterior en comparación con
928 * el SerialDate especificado.
929 *
930 * @param other La fecha con la que se compara.
931 *
932 * @return <code> true </code> si este SerialDate representa una fecha anterior
933 *                 en comparación con el SerialDate especificado.
934 * /
935 public abstract boolean isBefore (SerialDate otro);
936
937 / **
938 * Devuelve verdadero si este SerialDate representa la misma fecha que el
```

```
939 * Fecha de serie especificada.
940 *
941 * @param other la fecha con la que se compara.
942 *
943 * @return <code> true <code> si este SerialDate representa la misma fecha
944 *                          como el SerialDate especificado.
945 * /
946 public abstract boolean isOnOrBefore (SerialDate otro);
947
948 / **
949 * Devuelve verdadero si este SerialDate representa la misma fecha que el
950 * Fecha de serie especificada.
951 *
952 * @param other la fecha con la que se compara.
953 *
954 * @return <code> true </code> si este SerialDate representa la misma fecha
955 *                          como el SerialDate especificado.
956 * /
957 public abstract boolean isAfter (SerialDate otro);
958
959 / **
960 * Devuelve verdadero si este SerialDate representa la misma fecha que el
961 * Fecha de serie especificada.
962 *
963 * @param other la fecha con la que se compara.
964 *
965 * @return <code> true </code> si este SerialDate representa la misma fecha
966 *                          como el SerialDate especificado.
967 * /
968 public abstract boolean isOnOrAfter (SerialDate otro);
969
970 / **
971 * Devuelve <code> true </code> si este {@link SerialDate} está dentro del
```


Página 396

Listado B-1 (continuación)

SerialDate.Java

```
972 * rango especificado (INCLUIDO). El orden de fecha de d1 y d2 no es
973 * importante.
974 *
975 * @param d1 una fecha límite para el rango.
976 * @param d2 la otra fecha límite para el rango.
977 *
978 * @return A booleano.
979 * /
980 isInRange booleano abstracto público (SerialDate d1, SerialDate d2);
981
982 / **
983 * Devuelve <code> true </code> si este {@link SerialDate} está dentro del
984 * rango especificado (el llamador especifica si los puntos finales son
985 * incluido). El orden de las fechas de d1 y d2 no es importante.
986 *
987 * @param d1 una fecha límite para el rango.
988 * @param d2 la otra fecha límite para el rango.
989 * @param incluye un código que controla si el inicio y el final
990 *                                  las fechas están incluidas en el rango.
991 *
992 * @return A booleano.
993 * /
994 público abstracto booleano isInRange (SerialDate d1, SerialDate d2,
995                                                  int incluir);
996
997 / **
998 * Devuelve la última fecha que cae en el día de la semana especificado y
999 * es ANTES de esta fecha.
1000 *
1001 * @param targetDOW un código para el día de la semana objetivo.
1002 *
1003 * @return la última fecha que cae en el día de la semana especificado y
1004 *                          es ANTES de esta fecha.
1005 * /
1006 public SerialDate getPreviousDayOfWeek (final int targetDOW) {
1007             return getPreviousDayOfWeek (targetDOW, esto);
1008}
1009
1010 / **
1011 * Devuelve la fecha más temprana que cae en el día de la semana especificado
1012 * y es DESPUÉS de esta fecha.
1013 *
1014 * @param targetDOW un código para el día de la semana objetivo.
1015 *
1016 * @return la fecha más temprana que cae en el día de la semana especificado
1017 *                          y es DESPUÉS de esta fecha.
1018 * /
1019 public SerialDate getFollowingDayOfWeek (final int targetDOW) {
```

```
1020                    return getFollowingDayOfWeek (targetDOW, esto);
1021}
1022
1023 / **
1024 * Devuelve la fecha más cercana que cae en el día de la semana especificado.
1025 *
1026 * @param targetDOW un código para el día de la semana objetivo.
1027 *
1028 * @return la fecha más cercana que cae en el día de la semana especificado.
1029 * /
1030 public SerialDate getNearestDayOfWeek (final int targetDOW) {
1031                    return getNearestDayOfWeek (targetDOW, esto);
1032}
1033
1034}
```


Página 397

Listado B-2

SerialDateTest.java

```
1 / * ============================================= ========================
2 * JCommon: una biblioteca de clases de propósito general gratuita para la plataforma Java (tm)
3 * ============================================= ========================
4 *
5 * (C) Copyright 2000-2005, por Object Refinery Limited y contribuyentes.
6 *
7 * Información del proyecto: http://www.jfree.org/jcommon/index.html
8 *
9 * Esta biblioteca es software gratuito; puedes redistribuirlo y / o modificarlo
10 * bajo los términos de la Licencia Pública General Reducida GNU publicada por
11 * la Free Software Foundation; ya sea la versión 2.1 de la Licencia, o
12 * (a su elección) cualquier versión posterior.
13 *
14 * Esta biblioteca se distribuye con la esperanza de que sea útil, pero
15 * SIN NINGUNA GARANTÍA; incluso sin la garantía implícita de COMERCIABILIDAD
16 * o APTITUD PARA UN PROPÓSITO DETERMINADO. Ver el público general menor de GNU
17 * Licencia para más detalles.
18 *
19 * Debería haber recibido una copia de GNU Lesser General Public
20 * Licencia junto con esta biblioteca; si no es así, escriba al software libre
21 * Foundation, Inc., 51 Franklin Street, quinto piso, Boston, MA 02110-1301,
22 * Estados Unidos.
23 *
24 * [Java es una marca comercial o una marca comercial registrada de Sun Microsystems, Inc.
25 * en los Estados Unidos y otros países.]
26 *
27 * ----------
28 * SerialDateTests.java
29 * ----------
30 * (C) Copyright 2001-2005, por Object Refinery Limited.
31 *
32 * Autor original: David Gilbert (para Object Refinery Limited);
33 * Colaborador (es): -;
34 *
35 * $ Id: SerialDateTests.java, v 1.6 2005/11/16 15:58:40 taqua Exp $
36 *
37 * Cambios
38 * -------
39 * 15 de noviembre de 2001: Versión 1 (DG);
40 * 25 de junio de 2002: Se eliminó la importación innecesaria (DG);
41 * 24-Oct-2002: Errores corregidos reportados por Checkstyle (DG);
42 * 13-Mar-2003: Prueba de serialización agregada (DG);
43 * 05-Jan-2005: Prueba agregada para el informe de error 1096282 (DG);
44 *
45 * /
46
47 paquete org.jfree.date.junit;
48
49 import java.io.ByteArrayInputStream;
50 import java.io.ByteArrayOutputStream;
51 import java.io.ObjectInput;
52 import java.io.ObjectInputStream;
53 import java.io.ObjectOutput;
54 import java.io.ObjectOutputStream;
55
56 import junit.framework.Test;
57 import junit.framework.TestCase;
58 import junit.framework.TestSuite;
59
60 import org.jfree.date.MonthConstants;
61 import org.jfree.date.SerialDate;
62
```

Página 398

Listado B-2 (continuación)

SerialDateTest.java

```
63 / **
64 * Algunas pruebas de JUnit para la clase {@link SerialDate}.
sesenta y cinco */
66 SerialDateTests de clase pública extiende TestCase {
67
68 / ** Fecha que representa el 9 de noviembre. * /
69 Private SerialDate nov9Y2001;
70
71 / **
72 * Crea un nuevo caso de prueba.
73 *
74 * @param nombra el nombre.
75 * /
76 Public SerialDateTests (nombre de cadena final) {
77 super (nombre);
78}
79
80 / **
81 * Devuelve un conjunto de pruebas para el ejecutor de pruebas JUnit.
82 *
83 * @return El conjunto de pruebas.
84 * /
85 conjunto de pruebas estáticas públicas () {
86 return new TestSuite (SerialDateTests.class);
87}
88
89 / **
90 * Problema establecido.
91 * /
92 configuración vacía protegida () {
93 this.nov9Y2001 = SerialDate.createInstance (9, MonthConstants.NOVEMBER, 2001);
94}
95
96 / **
97 * 9 de noviembre de 2001 más dos meses debe ser el 9 de enero de 2002.
98 * /
99 public void testAddMonthsTo9Nov2001 () {
100 SerialDate final jan9Y2002 = SerialDate.addMonths (2, this.nov9Y2001);
101 respuesta final de SerialDate = SerialDate.createInstance (9, 1, 2002);
102 asertEquals (respuesta, 9 de enero de 2002);
103}
104
105 / **
106 * Un caso de prueba para un error informado, ahora corregido.
107 * /
108 prueba de vacío públicoAddMonthsTo5Oct2003 () {
109 SerialDate final d1 = SerialDate.createInstance (5, MonthConstants.OCTUBRE, 2003);
110 SerialDate final d2 = SerialDate.addMonths (2, d1);
111 asertEquals (d2, SerialDate.createInstance (5, MonthConstants.DECEMBER, 2003));
112}
113
114 / **
115 * Un caso de prueba para un error informado, ahora corregido.
116 * /
117 public void testAddMonthsTo1Jan2003 () {
118 SerialDate final d1 = SerialDate.createInstance (1, MonthConstants.JANUARY, 2003);
119 SerialDate final d2 = SerialDate.addMonths (0, d1);
120 asertEquals (d2, d1);
121}
122
123 / **
124 * El lunes anterior al viernes 9 de noviembre de 2001 debería ser el 5 de noviembre.
```


Página 399

Listado B-2 (continuación)

SerialDateTest.java

```
125 * /
126 prueba de vacío públicoMondayPrecedingFriday9Nov2001 () {
127                 SerialDate mondayBefore = SerialDate.getPreviousDayOfWeek (
128                         SerialDate.MONDAY, this.nov9Y2001
129                 );
130                 asertEquals (5, mondayBefore.getDayOfMonth ());
131}
132
133 / **
134 * El lunes siguiente al viernes 9 de noviembre de 2001 debería ser el 12 de noviembre.
135 * /
136 prueba de vacío públicoMondayFollowingFriday9Nov2001 () {
137                 SerialDate mondayAfter = SerialDate.getFollowingDayOfWeek (
138                         SerialDate.MONDAY, this.nov9Y2001
139                 );
140                 asertEquals (12, mondayAfter.getDayOfMonth ());
141}
142
143 / **
144 * El lunes más próximo al viernes 9 de noviembre de 2001 debería ser el 12 de noviembre.
145 * /
146 public nulo testMondayNearestFriday9Nov2001 () {
147                 SerialDate mondayNearest = SerialDate.getNearestDayOfWeek (
148                         SerialDate.MONDAY, this.nov9Y2001
149                 );
150                 asertEquals (12, mondayNearest.getDayOfMonth ());
151}
152
153 / **
154 * El lunes más próximo al 22 de enero de 1970 es el 19.
155 * /
156 prueba de vacío públicoMondayNearest22Jan1970 () {
157                 SerialDate jan22Y1970 = SerialDate.createInstance (22, MonthConstants.JANUARY, 1970);
158                 SerialDate mondayNearest = SerialDate.getNearestDayOfWeek (SerialDate.MONDAY, jan22Y1970);
159                 asertEquals (19, mondayNearest.getDayOfMonth ());
160}
161
162 / **
163 * Problema de que la conversión de días a cadenas devuelve el resultado correcto. En realidad, esto
164 * El resultado depende de la configuración regional, por lo que esta prueba debe modificarse.
165 * /
166 public void testWeekdayCodeToString () {
167
168                 prueba de cadena final = SerialDate.weekdayCodeToString (SerialDate.SATURDAY);
169                 asertEquals ("sábado", prueba);
170
171}
172
173 / **
174 * Pruebe la conversión de una cadena en un día de la semana. Tenga en cuenta que esta prueba fallará si el
175 * la configuración regional predeterminada no usa nombres de días de la semana en inglés ... ¡diseñe una prueba mejor!
176 * /
177 public void testStringToWeekday () {
178
179                 int weekday = SerialDate.stringToWeekdayCode ("miércoles");
180                 asertEquals (SerialDate.WEDNESDAY, día de la semana);
181
182                 weekday = SerialDate.stringToWeekdayCode ("miércoles");
183                 asertEquals (SerialDate.WEDNESDAY, día de la semana);
184
```


Página 400

Listado B-2 (continuación)

SerialDateTest.java

```
185                 weekday = SerialDate.stringToWeekdayCode ("Mié");
186                 asertEquals (SerialDate.WEDNESDAY, día de la semana);
187
188}
```

```
189
190 / **
191 * Pruebe la conversión de una cadena en un mes. Tenga en cuenta que esta prueba fallará si el
192 * la configuración regional predeterminada no usa nombres de meses en inglés ... ¡diseñe una prueba mejor!
193 * /
194 public void testStringToMonthCode () {
195
196                 int m = SerialDate.stringToMonthCode ("enero");
197                 asertEquals (MonthConstants.JANUARY, m);
198
199                 m = SerialDate.stringToMonthCode ("enero");
200                 asertEquals (MonthConstants.JANUARY, m);
201
202                 m = SerialDate.stringToMonthCode ("Jan");
203                 asertEquals (MonthConstants.JANUARY, m);
204
205}
206
207 / **
208 * Prueba la conversión de un código de mes a una cadena.
209 * /
210 public void testMonthCodeToStringCode () {
211
212                 prueba de cadena final = SerialDate.monthCodeToString (MonthConstants.DECEMBER);
213                 asertEquals ("diciembre", prueba);
214
215}
216
217 / **
218 * 1900 no es un año bisiesto.
219 * /
220 public void testIsNotLeapYear1900 () {
221                 asertTrue (! SerialDate.isLeapYear (1900));
222}
223
224 / **
225 * 2000 es un año bisiesto.
226 * /
227 public void testIsLeapYear2000 () {
228                 asertTrue (SerialDate.isLeapYear (2000));
229}
230
231 / **
232 * El número de años bisiestos desde 1900 hasta 1899 inclusive es 0.
233 * /
234 public void testLeapYearCount1899 () {
235                 asertEquals (SerialDate.leapYearCount (1899), 0);
236}
237
238 / **
239 * El número de años bisiestos desde 1900 hasta 1903 inclusive es 0.
240 * /
241 public void testLeapYearCount1903 () {
242                 asertEquals (SerialDate.leapYearCount (1903), 0);
243}
244
245 / **
246 * El número de años bisiestos desde 1900 hasta 1904 inclusive es 1.
247 * /
```


Página 401

Listado B-2 (continuación)

SerialDateTest.java

```
248 public void testLeapYearCount1904 () {
249                 asertEquals (SerialDate.leapYearCount (1904), 1);
250}
251
252 / **
253 * El número de años bisiestos desde 1900 hasta 1999 inclusive es 24.
254 * /
255 public void testLeapYearCount1999 () {
256                 asertEquals (SerialDate.leapYearCount (1999), 24);
257}
258
259 / **
260 * El número de años bisiestos desde 1900 hasta 2000 inclusive es 25.
261 * /
262 public void testLeapYearCount2000 () {
263                 asertEquals (SerialDate.leapYearCount (2000), 25);
264}
265
266 / **
267 * Serializar una instancia, restaurarla y verificar la igualdad.
268 * /
```

```
270 public void testSerialization () {
271         SerialDate d1 = SerialDate.createInstance (15, 4, 2000);
272         SerialDate d2 = nulo;
273
274         intentar {
275             ByteArrayOutputStream búfer = nuevo ByteArrayOutputStream ();
276             ObjectOutput out = nuevo ObjectOutputStream (búfer);
277             out.writeObject (d1);
278             out.close ();
279
280             ObjectInput en = nuevo ObjectInputStream (
                                        new ByteArrayInputStream (buffer.toByteArray ()));
281             d2 = (SerialDate) in.readObject ();
282             cercar();
283         }
284         captura (Excepción e) {
285             System.out.println (e.toString ());
286         }
287         asertEquals (d1, d2);
288
289}
290
291 / **
292 * Una prueba para el informe de error 1096282 (ahora corregido).
293 * /
294 prueba de vacío público1096282 () {
295         SerialDate d = SerialDate.createInstance (29, 2, 2004);
296         d = SerialDate.addYears (1, d);
297         SerialDate esperado = SerialDate.createInstance (28, 2, 2005);
298         asertTrue (d.isOn (esperado));
299}
300
301 / **
302 * Pruebas varias para el método addMonths ().
303 * /
304 public void testAddMonths () {
305         SerialDate d1 = SerialDate.createInstance (31, 5, 2004);
306
```


Página 402

Listado B-2 (continuación)

SerialDateTest.java

```
307         SerialDate d2 = SerialDate.addMonths (1, d1);
308         asertEquals (30, d2.getDayOfMonth ());
309         asertEquals (6, d2.getMonth ());
310         asertEquals (2004, d2.getYYYY ());
311
312         SerialDate d3 = SerialDate.addMonths (2, d1);
313         asertEquals (31, d3.getDayOfMonth ());
314         asertEquals (7, d3.getMonth ());
315         asertEquals (2004, d3.getYYYY ());
316
317         SerialDate d4 = SerialDate.addMonths (1, SerialDate.addMonths (1, d1));
318         asertEquals (30, d4.getDayOfMonth ());
319         asertEquals (7, d4.getMonth ());
320         asertEquals (2004, d4.getYYYY ());
321}
322}
```

Página 403

Listado B-3

MonthConstants.java

```
1 / * ================================================ ============================
2 * JCommon: una biblioteca de clases de propósito general gratuita para la plataforma Java (tm)
3 * ================================================ ============================
4 *
5 * (C) Copyright 2000-2005, por Object Refinery Limited y contribuyentes.
6 *
7 * Información del proyecto: http://www.jfree.org/jcommon/index.html
8 *
9 * Esta biblioteca es software gratuito; puedes redistribuirlo y / o modificarlo
10 * bajo los términos de la Licencia Pública General Reducida GNU publicada por
11 * la Free Software Foundation; ya sea la versión 2.1 de la Licencia, o
12 * (a su elección) cualquier versión posterior.
13 *
14 * Esta biblioteca se distribuye con la esperanza de que sea útil, pero
15 * SIN NINGUNA GARANTÍA; incluso sin la garantía implícita de COMERCIABILIDAD
16 * o APTITUD PARA UN PROPÓSITO DETERMINADO. Ver el público general menor de GNU
17 * Licencia para más detalles.
18 *
19 * Debería haber recibido una copia de GNU Lesser General Public
20 * Licencia junto con esta biblioteca; si no es así, escriba al software libre
21 * Foundation, Inc., 51 Franklin Street, quinto piso, Boston, MA 02110-1301,
22 * Estados Unidos.
23 *
24 * [Java es una marca comercial o una marca comercial registrada de Sun Microsystems, Inc.
25 * en los Estados Unidos y otros países.]
26 *
27 * ------------------
28 * MonthConstants.java
29 * ------------------
30 * (C) Copyright 2002, 2003, de Object Refinery Limited.
31 *
32 * Autor original: David Gilbert (para Object Refinery Limited);
33 * Colaborador (es): -;
34 *
35 * $ Id: MonthConstants.java, v 1.4 2005/11/16 15:58:40 taqua Exp $
36 *
37 * Cambios
38 * -------
39 * 29 de mayo de 2002: Versión 1 (código movido de la clase SerialDate) (DG);
40 *
41 * /
42
43 paquete org.jfree.date;
44
45 / **
46 * Constantes útiles durante meses. Tenga en cuenta que estos NO son equivalentes a los
47 * constantes definidas por java.util.Calendar (donde ENERO = 0 y DICIEMBRE = 11).
48 * <P>
49 * Utilizado por las clases SerialDate y RegularTimePeriod.
50 *
51 * @ autor David Gilbert
52 * /
53 MonthConstants de interfaz pública {
54
55 / ** Constante de enero. * /
56 public static final int ENERO = 1;
57
```

```
58 / ** Constante de febrero. * /
59 public static final int FEBRERO = 2;
60
```


Página 404

Listado B-3 (continuación)

MonthConstants.java

```
61 / ** Constante de marzo. * /
62 público estático final int MARZO = 3;
63
64 / ** Constante de abril. * /
65 public static final int APRIL = 4;
66
67 / ** Constante de mayo. * /
68 public static final int MAY = 5;
69
70 / ** Constante de junio. * /
71 public static final int JUNE = 6;
72
73 / ** Constante de julio. * /
74 public static final int JULY = 7;
75
76 / ** Constante de agosto. * /
77 public static final int AGOSTO = 8;
78
79 / ** Constante de septiembre. * /
80 public static final int SEPTIEMBRE = 9;
81
82 / ** Constante de octubre. * /
83 public static final int OCTUBRE = 10;
84
85 / ** Constante de noviembre. * /
86 public static final int NOVIEMBRE = 11;
87
88 / ** Constante de diciembre. * /
89 public static final int DICIEMBRE = 12;
90
91}
```

Página 405

Listado B-4

BobsSerialDateTest.java

```
1 paquete org.jfree.date.junit;
2
3 import junit.framework.TestCase;
4 importar org.jfree.date. *;
5 importar org.jfree.date.SerialDate. * Estático;
6
7 import java.util. *;
8
9 BobsSerialDateTest de clase pública extiende TestCase {
10
11 public void testIsValidWeekdayCode () arroja Exception {
12 para (int día = 1; día <= 7; día ++)
13 asertTrue (isValidWeekdayCode (día));
14 asertFalse (isValidWeekdayCode (0));
15 asertFalse (isValidWeekdayCode (8));
dieciséis }
17
18 public void testStringToWeekdayCode () lanza Exception {
19
20 assertEquals (-1, stringToWeekdayCode ("Hola"));
21 assertEquals (LUNES, stringToWeekdayCode ("Lunes"));
22 assertEquals (LUNES, stringToWeekdayCode ("Mon"));
23 // todo assertEquals (LUNES, stringToWeekdayCode ("lunes"));
24 // assertEquals (LUNES, stringToWeekdayCode ("LUNES"));
25 // assertEquals (LUNES, stringToWeekdayCode ("mon"));
26
27 assertEquals (MARTES, stringToWeekdayCode ("Martes"));
28 assertEquals (MARTES, stringToWeekdayCode ("Tue"));
29 // assertEquals (MARTES, stringToWeekdayCode ("martes"));
30 // assertEquals (MARTES, stringToWeekdayCode ("MARTES"));
31 // assertEquals (MARTES, stringToWeekdayCode ("mar"));
32 // assertEquals (MARTES, stringToWeekdayCode ("tues"));
33
34 assertEquals (MIÉRCOLES, stringToWeekdayCode ("miércoles"));
35 assertEquals (MIÉRCOLES, stringToWeekdayCode ("Mié"));
36 // assertEquals (MIÉRCOLES, stringToWeekdayCode ("miércoles"));
37 // assertEquals (MIÉRCOLES, stringToWeekdayCode ("MIÉRCOLES"));
38 // assertEquals (MIÉRCOLES, stringToWeekdayCode ("mié"));
39
40 assertEquals (JUEVES, stringToWeekdayCode ("Jueves"));
41 assertEquals (JUEVES, stringToWeekdayCode ("Jue"));
42 // assertEquals (JUEVES, stringToWeekdayCode ("jueves"));
43 // assertEquals (JUEVES, stringToWeekdayCode ("JUEVES"));
44 // assertEquals (JUEVES, stringToWeekdayCode ("jue"));
45 // assertEquals (JUEVES, stringToWeekdayCode ("jueves"));
46
47 assertEquals (VIERNES, stringToWeekdayCode ("Viernes"));
48 assertEquals (VIERNES, stringToWeekdayCode ("Vie"));
49 // assertEquals (VIERNES, stringToWeekdayCode ("viernes"));
50 // assertEquals (VIERNES, stringToWeekdayCode ("VIERNES"));
51 // assertEquals (VIERNES, stringToWeekdayCode ("vie"));
52
53 assertEquals (SÁBADO, stringToWeekdayCode ("Sábado"));
54 assertEquals (SÁBADO, stringToWeekdayCode ("Sábado"));
55 // assertEquals (SÁBADO, stringToWeekdayCode ("sábado"));
56 // assertEquals (SÁBADO, stringToWeekdayCode ("SÁBADO"));
57 // assertEquals (SÁBADO, stringToWeekdayCode ("sat"));
58
59 assertEquals (DOMINGO, stringToWeekdayCode ("Domingo"));
60 assertEquals (DOMINGO, stringToWeekdayCode ("Sol"));
61 // assertEquals (DOMINGO, stringToWeekdayCode ("domingo"));
62 // assertEquals (DOMINGO, stringToWeekdayCode ("DOMINGO"));
63 // assertEquals (DOMINGO, stringToWeekdayCode ("sol"));
64}
sesenta y cinco
```


Página 406

Listado B-4 (continuación)

BobsSerialDateTest.java

```
66 public void testWeekdayCodeToString () lanza Exception {
67 assertEquals ("Domingo", weekdayCodeToString (DOMINGO));
68 assertEquals ("Lunes", weekdayCodeToString (MONDAY));
69 assertEquals ("martes", weekdayCodeToString (MARTES));
70 assertEquals ("miércoles", weekdayCodeToString (WEDNESDAY));
71 assertEquals ("jueves", weekdayCodeToString (JUEVES));
72 assertEquals ("Viernes", weekdayCodeToString (VIERNES));
73 assertEquals ("sábado", weekdayCodeToString (SATURDAY));
74}
75
76 public void testIsValidMonthCode () lanza Exception {
77 para (int i = 1; i <= 12; i ++)
78 asertTrue (isValidMonthCode (i));
79 assertFalse (isValidMonthCode (0));
80 asserFalse (isValidMonthCode (13));
81}
82
83 public void testMonthToQuarter () arroja Exception {
84 assertEquals (1, monthCodeToQuarter (ENERO));
85 assertEquals (1, monthCodeToQuarter (FEBRERO));
86 assertEquals (1, monthCodeToQuarter (MARZO));
87 assertEquals (2, monthCodeToQuarter (ABRIL));
88 assertEquals (2, monthCodeToQuarter (MAY));
89 assertEquals (2, monthCodeToQuarter (JUNIO));
90 assertEquals (3, monthCodeToQuarter (JULY));
91 assertEquals (3, monthCodeToQuarter (AGOSTO));
92 assertEquals (3, monthCodeToQuarter (SEPTIEMBRE));
93 assertEquals (4, monthCodeToQuarter (OCTUBRE));
94 assertEquals (4, monthCodeToQuarter (NOVIEMBRE));
95 assertEquals (4, monthCodeToQuarter (DICIEMBRE));
96
97 intento {
98 monthCodeToQuarter (-1);
99 fail ("El código de mes no válido debería generar una excepción");
100} catch (IllegalArgumentException e) {
101}
102}
103
104 public void testMonthCodeToString () lanza Exception {
105 assertEquals ("enero", monthCodeToString (ENERO));
106 assertEquals ("febrero", monthCodeToString (FEBRERO));
107 assertEquals ("marzo", monthCodeToString (MARZO));
108 assertEquals ("abril", monthCodeToString (ABRIL));
109 assertEquals ("mayo", monthCodeToString (MAY));
110 assertEquals ("junio", monthCodeToString (JUNE));
111 assertEquals ("julio", monthCodeToString (JULY));
112 assertEquals ("agosto", monthCodeToString (AGOSTO));
113 assertEquals ("septiembre", monthCodeToString (SEPTIEMBRE));
114 assertEquals ("Octubre", monthCodeToString (OCTUBRE));
115 assertEquals ("noviembre", monthCodeToString (NOVIEMBRE));
116 assertEquals ("diciembre", monthCodeToString (DICIEMBRE));
117
118 assertEquals ("Jan", monthCodeToString (ENERO, verdadero));
119 assertEquals ("Feb", monthCodeToString (FEBRERO, verdadero));
120 assertEquals ("Mar", monthCodeToString (MARZO, verdadero));
121 assertEquals ("Abr", monthCodeToString (ABRIL, verdadero));
122 assertEquals ("mayo", monthCodeToString (MAY, true));
123 assertEquals ("Jun", monthCodeToString (JUNE, verdadero));
124 assertEquals ("julio", monthCodeToString (JULY, true));
125 assertEquals ("Agosto", monthCodeToString (AGOSTO, verdadero));
126 assertEquals ("Sep", monthCodeToString (SEPTIEMBRE, verdadero));
127 assertEquals ("Oct", monthCodeToString (OCTUBRE, verdadero));
```


Página 407

Listado B-4 (continuación)

BobsSerialDateTest.java

```
128 assertEquals ("Nov", monthCodeToString (NOVIEMBRE, verdadero));
129 assertEquals ("diciembre", monthCodeToString (DICIEMBRE, verdadero));
130
131 intento {
132 monthCodeToString (-1);
133 fail ("El código de mes no válido debería generar una excepción");
134} catch (IllegalArgumentException e) {
135}
136
137}
138
139 public void testStringToMonthCode () lanza Exception {
140 assertEquals (ENERO, stringToMonthCode ("1"));
```

```
141 assertEquals (FEBRERO, stringToMonthCode ("2"));
142 assertEquals (MARZO, stringToMonthCode ("3"));
143 assertEquals (ABRIL, stringToMonthCode ("4"));
144 assertEquals (MAY, stringToMonthCode ("5"));
145 assertEquals (JUNIO, stringToMonthCode ("6"));
146 assertEquals (JULIO, stringToMonthCode ("7"));
147 assertEquals (AGOSTO, stringToMonthCode ("8"));
148 assertEquals (SEPTIEMBRE, stringToMonthCode ("9"));
149 assertEquals (OCTUBRE, stringToMonthCode ("10"));
150 assertEquals (NOVIEMBRE, stringToMonthCode ("11"));
151 assertEquals (DICIEMBRE, stringToMonthCode ("12"));
152
153 // todo assertEquals (-1, stringToMonthCode ("0"));
154 // assertEquals (-1, stringToMonthCode ("13"));
155
156 assertEquals (-1, stringToMonthCode ("Hola"));
157
158 para (int m = 1; m <= 12; m ++) {
159 assertEquals (m, stringToMonthCode (monthCodeToString (m, falso)));
160 assertEquals (m, stringToMonthCode (monthCodeToString (m, verdadero)));
161}
162
163 // assertEquals (1, stringToMonthCode ("jan"));
164 // assertEquals (2, stringToMonthCode ("feb"));
165 // assertEquals (3, stringToMonthCode ("mar"));
166 // assertEquals (4, stringToMonthCode ("apr"));
167 // assertEquals (5, stringToMonthCode ("puede"));
168 // assertEquals (6, stringToMonthCode ("jun"));
169 // assertEquals (7, stringToMonthCode ("jul"));
170 // assertEquals (8, stringToMonthCode ("aug"));
171 // assertEquals (9, stringToMonthCode ("sep"));
172 // assertEquals (10, stringToMonthCode ("oct"));
173 // assertEquals (11, stringToMonthCode ("nov"));
174 // assertEquals (12, stringToMonthCode ("dec"));
175
176 // assertEquals (1, stringToMonthCode ("JAN"));
177 // assertEquals (2, stringToMonthCode ("FEB"));
178 // assertEquals (3, stringToMonthCode ("MAR"));
179 // assertEquals (4, stringToMonthCode ("APR"));
180 // assertEquals (5, stringToMonthCode ("MAYO"));
181 // assertEquals (6, stringToMonthCode ("JUN"));
182 // assertEquals (7, stringToMonthCode ("JUL"));
183 // assertEquals (8, stringToMonthCode ("AUG"));
184 // assertEquals (9, stringToMonthCode ("SEP"));
185 // assertEquals (10, stringToMonthCode ("OCT"));
186 // assertEquals (11, stringToMonthCode ("NOV"));
187 // assertEquals (12, stringToMonthCode ("DEC"));
188
189 // assertEquals (1, stringToMonthCode ("enero"));
190 // assertEquals (2, stringToMonthCode ("febrero"));
```


Página 408

Listado B-4 (continuación)

BobsSerialDateTest.java

```
191 // assertEquals (3, stringToMonthCode ("marcha"));
192 // assertEquals (4, stringToMonthCode ("abril"));
193 // assertEquals (5, stringToMonthCode ("puede"));
194 // assertEquals (6, stringToMonthCode ("junio"));
195 // assertEquals (7, stringToMonthCode ("julio"));
196 // assertEquals (8, stringToMonthCode ("agosto"));
197 // assertEquals (9, stringToMonthCode ("septiembre"));
198 // assertEquals (10, stringToMonthCode ("octubre"));
199 // assertEquals (11, stringToMonthCode ("noviembre"));
200 // assertEquals (12, stringToMonthCode ("diciembre"));
201
202 // assertEquals (1, stringToMonthCode ("ENERO"));
203 // assertEquals (2, stringToMonthCode ("FEBRERO"));
204 // assertEquals (3, stringToMonthCode ("MAR"));
205 // assertEquals (4, stringToMonthCode ("ABRIL"));
206 // assertEquals (5, stringToMonthCode ("MAYO"));
207 // assertEquals (6, stringToMonthCode ("JUNIO"));
208 // assertEquals (7, stringToMonthCode ("JULIO"));
209 // assertEquals (8, stringToMonthCode ("AGOSTO"));
210 // assertEquals (9, stringToMonthCode ("SEPTIEMBRE"));
211 // assertEquals (10, stringToMonthCode ("OCTUBRE"));
212 // assertEquals (11, stringToMonthCode ("NOVIEMBRE"));
213 // assertEquals (12, stringToMonthCode ("DICIEMBRE"));
214}
215
216 public void testIsValidWeekInMonthCode () lanza Exception {
217 para (int w = 0; w <= 4; w ++) {
218 asertTrue (isValidWeekInMonthCode (w));
219}
220 assertFalse (isValidWeekInMonthCode (5));
221}
```

```
222
223 public void testIsLeapYear () arroja Exception {
224 assertFalse (isLeapYear (1900));
225 assertFalse (isLeapYear (1901));
226 asertFalse (isLeapYear (1902));
227 assertFalse (isLeapYear (1903));
228 assertTrue (isLeapYear (1904));
229 asertTrue (isLeapYear (1908));
230 aseverarFalso (isLeapYear (1955));
231 asertTrue (isLeapYear (1964));
232 asertTrue (isLeapYear (1980));
233 asertTrue (isLeapYear (2000));
234 assertFalse (isLeapYear (2001));
235 asertFalse (isLeapYear (2100));
236}
237
238 public void testLeapYearCount () arroja Exception {
239 assertEquals (0, leapYearCount (1900));
240 assertEquals (0, leapYearCount (1901));
241 assertEquals (0, leapYearCount (1902));
242 assertEquals (0, leapYearCount (1903));
243 assertEquals (1, leapYearCount (1904));
244 assertEquals (1, leapYearCount (1905));
245 assertEquals (1, leapYearCount (1906));
246 assertEquals (1, leapYearCount (1907));
247 assertEquals (2, leapYearCount (1908));
248 assertEquals (24, leapYearCount (1999));
249 assertEquals (25, leapYearCount (2001));
250 assertEquals (49, leapYearCount (2101));
251 assertEquals (73, leapYearCount (2201));
```


Página 409

Listado B-4 (continuación)

BobsSerialDateTest.java

```
252 assertEquals (97, leapYearCount (2301));
253 assertEquals (122, leapYearCount (2401));
254}
255
256 public void testLastDayOfMonth () arroja Exception {
257 assertEquals (31, lastDayOfMonth (ENERO, 1901));
258 assertEquals (28, lastDayOfMonth (FEBRERO, 1901));
259 assertEquals (31, lastDayOfMonth (MARZO, 1901));
260 assertEquals (30, lastDayOfMonth (ABRIL, 1901));
261 assertEquals (31, lastDayOfMonth (MAYO, 1901));
262 assertEquals (30, lastDayOfMonth (JUNIO, 1901));
263 assertEquals (31, lastDayOfMonth (JULIO, 1901));
264 assertEquals (31, lastDayOfMonth (AGOSTO, 1901));
265 assertEquals (30, lastDayOfMonth (SEPTIEMBRE, 1901));
266 assertEquals (31, lastDayOfMonth (OCTUBRE, 1901));
267 assertEquals (30, lastDayOfMonth (NOVIEMBRE, 1901));
268 assertEquals (31, lastDayOfMonth (DICIEMBRE, 1901));
269 assertEquals (29, lastDayOfMonth (FEBRERO, 1904));
270}
271
272 public void testAddDays () lanza Exception {
273 SerialDate newYears = d (1, ENERO DE 1900);
274 assertEquals (d (2, ENERO, 1900), addDays (1, newYears));
275 assertEquals (d (1, FEBRERO, 1900), addDays (31, newYears));
276 assertEquals (d (1 de enero de 1901), addDays (365, newYears));
277 assertEquals (d (31 de diciembre de 1904), addDays (5 * 365, newYears));
278}
279
280 hoja de cálculo estática privada Fecha d (día int, mes int, año int) {return new
SpreadsheetDate (día, mes, año);}
281
282 public void testAddMonths () arroja Exception {
283 assertEquals (d (1, FEBRERO, 1900), addMonths (1, d (1, ENERO, 1900)));
284 assertEquals (d (28, FEBRERO, 1900), addMonths (1, d (31, ENERO, 1900)));
285 assertEquals (d (28, FEBRERO, 1900), addMonths (1, d (30, ENERO, 1900)));
286 assertEquals (d (28, FEBRERO, 1900), addMonths (1, d (29, ENERO, 1900)));
287 assertEquals (d (28, FEBRERO, 1900), addMonths (1, d (28, ENERO, 1900)));
288 assertEquals (d (27, FEBRERO, 1900), addMonths (1, d (27, ENERO, 1900)));
289
290 assertEquals (d (30, JUNIO, 1900), addMonths (5, d (31, ENERO, 1900)));
291 assertEquals (d (30, JUNIO, 1901), addMonths (17, d (31, ENERO, 1900)));
292
293 assertEquals (d (29, FEBRERO, 1904), addMonths (49, d (31, ENERO, 1900)));
294
295}
296
297 public void testAddYears () arroja Exception {
298 assertEquals (d (1 de enero de 1901), addYears (1, d (1 de enero de 1900)));
299 assertEquals (d (28, FEBRERO, 1905), addYears (1, d (29, FEBRERO, 1904)));
```

```
300 assertEquals (d (28, FEBRERO, 1905), addYears (1, d (28, FEBRERO, 1904)));
301 assertEquals (d (28, FEBRERO, 1904), addYears (1, d (28, FEBRERO, 1903)));
302}
303
304 public void testGetPreviousDayOfWeek () arroja Exception {
305 assertEquals (d (24, FEBRERO, 2006), getPreviousDayOfWeek (VIERNES, d (1, MARZO, 2006)));
306 assertEquals (d (22, FEBRERO, 2006), getPreviousDayOfWeek (WEDNESDAY, d (1, MARZO, 2006)));
307 assertEquals (d (29, FEBRERO, 2004), getPreviousDayOfWeek (SUNDAY, d (3, MARZO, 2004)));
308 assertEquals (d (29, DICIEMBRE, 2004), getPreviousDayOfWeek (WEDNESDAY, d (5, ENERO, 2005)));
309
310 intento {
311 getPreviousDayOfWeek (-1, d (1, ENERO de 2006));
312 fail ("El código de día de la semana no válido debería generar una excepción");
```


Página 410

Listado B-4 (continuación)

BobsSerialDateTest.java

```
313} catch (IllegalArgumentException e) {
314}
315}
316
317 public void testGetFollowingDayOfWeek () arroja Exception {
318 // assertEquals (d (1, ENERO, 2005), getFollowingDayOfWeek (SATURDAY, d (25, DICIEMBRE, 2004)));
319 assertEquals (d (1, ENERO, 2005), getFollowingDayOfWeek (SATURDAY, d (26, DICIEMBRE, 2004)));
320 assertEquals (d (3, MARZO, 2004), getFollowingDayOfWeek (MIÉRCOLES, d (28, FEBRERO, 2004)));
321
322 prueba {
323 getFollowingDayOfWeek (-1, d (1, ENERO de 2006));
324 fail ("El código de día de la semana no válido debería generar una excepción");
325} catch (IllegalArgumentException e) {
326}
327}
328
329 public void testGetNearestDayOfWeek () arroja Exception {
330 assertEquals (d (16, ABRIL, 2006), getNearestDayOfWeek (SUNDAY, d (16, ABRIL, 2006)));
331 assertEquals (d (16, ABRIL, 2006), getNearestDayOfWeek (SUNDAY, d (17, ABRIL, 2006)));
332 assertEquals (d (16, ABRIL, 2006), getNearestDayOfWeek (SUNDAY, d (18, ABRIL, 2006)));
333 assertEquals (d (16, ABRIL, 2006), getNearestDayOfWeek (SUNDAY, d (19, ABRIL, 2006)));
334 assertEquals (d (23, ABRIL, 2006), getNearestDayOfWeek (SUNDAY, d (20, ABRIL, 2006)));
335 assertEquals (d (23, ABRIL, 2006), getNearestDayOfWeek (SUNDAY, d (21, ABRIL, 2006)));
336 assertEquals (d (23, ABRIL, 2006), getNearestDayOfWeek (SUNDAY, d (22, ABRIL, 2006)));
337
338 // todo assertEquals (d (17, ABRIL, 2006), getNearestDayOfWeek (MONDAY, d (16, ABRIL, 2006)));
339 assertEquals (d (17, ABRIL, 2006), getNearestDayOfWeek (MONDAY, d (17, ABRIL, 2006)));
340 assertEquals (d (17, ABRIL, 2006), getNearestDayOfWeek (MONDAY, d (18, ABRIL, 2006)));
341 assertEquals (d (17, ABRIL, 2006), getNearestDayOfWeek (MONDAY, d (19, ABRIL, 2006)));
342 assertEquals (d (17, ABRIL, 2006), getNearestDayOfWeek (MONDAY, d (20, ABRIL, 2006)));
343 assertEquals (d (24, ABRIL, 2006), getNearestDayOfWeek (MONDAY, d (21, ABRIL, 2006)));
344 assertEquals (d (24, ABRIL, 2006), getNearestDayOfWeek (MONDAY, d (22, ABRIL, 2006)));
345
346 // assertEquals (d (18, ABRIL, 2006), getNearestDayOfWeek (MARTES, d (16, ABRIL, 2006)));
347 // assertEquals (d (18, ABRIL, 2006), getNearestDayOfWeek (MARTES, d (17, ABRIL, 2006)));
348 assertEquals (d (18, ABRIL, 2006), getNearestDayOfWeek (MARTES, d (18, ABRIL, 2006)));
349 assertEquals (d (18, ABRIL, 2006), getNearestDayOfWeek (MARTES, d (19, ABRIL, 2006)));
350 assertEquals (d (18, ABRIL, 2006), getNearestDayOfWeek (MARTES, d (20, ABRIL, 2006)));
351 assertEquals (d (18, ABRIL, 2006), getNearestDayOfWeek (MARTES, d (21, ABRIL, 2006)));
352 assertEquals (d (25, ABRIL, 2006), getNearestDayOfWeek (MARTES, d (22, ABRIL, 2006)));
353
354 // assertEquals (d (19, ABRIL, 2006), getNearestDayOfWeek (WEDNESDAY, d (16, ABRIL, 2006)));
355 // assertEquals (d (19, ABRIL, 2006), getNearestDayOfWeek (WEDNESDAY, d (17, ABRIL, 2006)));
356 // assertEquals (d (19, ABRIL, 2006), getNearestDayOfWeek (WEDNESDAY, d (18, ABRIL, 2006)));
357 assertEquals (d (19, ABRIL, 2006), getNearestDayOfWeek (WEDNESDAY, d (19, ABRIL, 2006)));
358 assertEquals (d (19, ABRIL, 2006), getNearestDayOfWeek (WEDNESDAY, d (20, ABRIL, 2006)));
359 assertEquals (d (19, ABRIL, 2006), getNearestDayOfWeek (WEDNESDAY, d (21, ABRIL, 2006)));
360 assertEquals (d (19, ABRIL, 2006), getNearestDayOfWeek (WEDNESDAY, d (22, ABRIL, 2006)));
361
362 // assertEquals (d (13, ABRIL, 2006), getNearestDayOfWeek (JUEVES, d (16, ABRIL, 2006)));
363 // assertEquals (d (20, ABRIL, 2006), getNearestDayOfWeek (JUEVES, d (17, ABRIL, 2006)));
364 // assertEquals (d (20, ABRIL, 2006), getNearestDayOfWeek (JUEVES, d (18, ABRIL, 2006)));
365 // assertEquals (d (20, ABRIL, 2006), getNearestDayOfWeek (JUEVES, d (19, ABRIL, 2006)));
366 assertEquals (d (20, ABRIL, 2006), getNearestDayOfWeek (THURSDAY, d (20, ABRIL, 2006)));
367 assertEquals (d (20, ABRIL, 2006), getNearestDayOfWeek (THURSDAY, d (21, ABRIL, 2006)));
368 assertEquals (d (20, ABRIL, 2006), getNearestDayOfWeek (JUEVES, d (22, ABRIL, 2006)));
369
370 // assertEquals (d (14, ABRIL, 2006), getNearestDayOfWeek (VIERNES, d (16, ABRIL, 2006)));
371 // assertEquals (d (14, ABRIL, 2006), getNearestDayOfWeek (VIERNES, d (17, ABRIL, 2006)));
372 // assertEquals (d (21, ABRIL, 2006), getNearestDayOfWeek (VIERNES, d (18, ABRIL, 2006)));
373 // assertEquals (d (21, ABRIL, 2006), getNearestDayOfWeek (VIERNES, d (19, ABRIL, 2006)));
374 // assertEquals (d (21, ABRIL, 2006), getNearestDayOfWeek (VIERNES, d (20, ABRIL, 2006)));
```

Página 411

Listado B-4 (continuación)

BobsSerialDateTest.java

```
375 assertEquals (d (21, ABRIL, 2006), getNearestDayOfWeek (VIERNES, d (21, ABRIL, 2006)));
376 assertEquals (d (21, ABRIL, 2006), getNearestDayOfWeek (VIERNES, d (22, ABRIL, 2006)));
377
378 // assertEquals (d (15, ABRIL, 2006), getNearestDayOfWeek (SATURDAY, d (16, ABRIL, 2006)));
379 // assertEquals (d (15, ABRIL, 2006), getNearestDayOfWeek (SATURDAY, d (17, ABRIL, 2006)));
380 // assertEquals (d (15, ABRIL, 2006), getNearestDayOfWeek (SATURDAY, d (18, ABRIL, 2006)));
381 // assertEquals (d (22, ABRIL, 2006), getNearestDayOfWeek (SATURDAY, d (19, ABRIL, 2006)));
382 // assertEquals (d (22, ABRIL, 2006), getNearestDayOfWeek (SATURDAY, d (20, ABRIL, 2006)));
383 // assertEquals (d (22, ABRIL, 2006), getNearestDayOfWeek (SATURDAY, d (21, ABRIL, 2006)));
384 assertEquals (d (22, ABRIL, 2006), getNearestDayOfWeek (SATURDAY, d (22, ABRIL, 2006)));
385
386 intento {
387 getNearestDayOfWeek (-1, d (1 de enero de 2006));
388 fail ("El código de día de la semana no válido debería generar una excepción");
389} catch (IllegalArgumentException e) {
390}
391}
392
393 public void testEndOfCurrentMonth () arroja Exception {
394 SerialDate d = SerialDate.createInstance (2);
395 assertEquals (d (31 de enero de 2006), d.getEndOfCurrentMonth (d (1 de enero de 2006)));
396 assertEquals (d (28, FEBRERO, 2006), d.getEndOfCurrentMonth (d (1, FEBRERO, 2006)));
397 assertEquals (d (31, MARZO, 2006), d.getEndOfCurrentMonth (d (1, MARZO, 2006)));
398 assertEquals (d (30, ABRIL, 2006), d.getEndOfCurrentMonth (d (1, ABRIL, 2006)));
399 assertEquals (d (31, MAYO, 2006), d.getEndOfCurrentMonth (d (1, MAY, 2006)));
400 assertEquals (d (30, JUNIO, 2006), d.getEndOfCurrentMonth (d (1, JUNIO, 2006)));
401 assertEquals (d (31, JULIO, 2006), d.getEndOfCurrentMonth (d (1, JULIO, 2006)));
402 assertEquals (d (31, AGOSTO, 2006), d.getEndOfCurrentMonth (d (1, AGOSTO, 2006)));
403 assertEquals (d (30, SEPTIEMBRE, 2006), d.getEndOfCurrentMonth (d (1, SEPTIEMBRE, 2006)));
404 assertEquals (d (31, OCTUBRE, 2006), d.getEndOfCurrentMonth (d (1, OCTUBRE, 2006)));
405 assertEquals (d (30, NOVIEMBRE, 2006), d.getEndOfCurrentMonth (d (1, NOVIEMBRE, 2006)));
406 assertEquals (d (31, DICIEMBRE, 2006), d.getEndOfCurrentMonth (d (1, DICIEMBRE, 2006)));
407 assertEquals (d (29, FEBRERO, 2008), d.getEndOfCurrentMonth (d (1, FEBRERO, 2008)));
408}
409
410 public void testWeekInMonthToString () lanza Exception {
411 assertEquals ("Primero", weekInMonthToString (FIRST_WEEK_IN_MONTH));
412 assertEquals ("Segundo", weekInMonthToString (SECOND_WEEK_IN_MONTH));
413 assertEquals ("Tercero", weekInMonthToString (THIRD_WEEK_IN_MONTH));
414 assertEquals ("Cuarto", weekInMonthToString (CUARTA SEMANA_IN_MONTH));
415 assertEquals ("Último", weekInMonthToString (LAST_WEEK_IN_MONTH));
416
417 // todo intentar {
418 // weekInMonthToString (-1);
419 // fail ("El código de semana no válido debería generar una excepción");
420 //} catch (IllegalArgumentException e) {
421 //}
422}
423
424 public void testRelativeToString () arroja Exception {
425 assertEquals ("Precediendo", relativoToString (PRECEDENTE));
426 assertEquals ("Más cercano", relativoToString (MÁS CERCANO));
427 assertEquals ("Siguiente", relativoToString (SEGUIMIENTO));
428
429 // todo try {
430 // relativoToString (-1000);
431 // fail ("El código relativo no válido debería generar una excepción");
432 //} catch (IllegalArgumentException e) {
433 //}
434}
435
```


Página 412

Listado B-4 (continuación)

BobsSerialDateTest.java

```
436 public void testCreateInstanceFromDDMMYYY () arroja Exception {
437 SerialDate date = createInstance (1, ENERO DE 1900);
438 assertEquals (1, date.getDayOfMonth ());
439 assertEquals (ENERO, date.getMonth ());
440 assertEquals (1900, date.getYYYY ());
441 assertEquals (2, date.toSerial ());
442}
443
444 public void testCreateInstanceFromSerial () lanza Exception {
445 assertEquals (d (1, ENERO DE 1900), createInstance (2));
446 assertEquals (d (1 de enero de 1901), createInstance (367));
447}
448
449 public void testCreateInstanceFromJavaDate () arroja Exception {
450 assertEquals (d (1 de enero de 1900),
                                createInstance (nuevo GregorianCalendar (1900,0,1) .getTime ()));
451 assertEquals (d (1 de enero de 2006),
                                createInstance (nuevo GregorianCalendar (2006,0,1) .getTime ()));
452}
453
454 public static void main (String [] args) {
455 junit.textui.TestRunner.run (BobsSerialDateTest.class);
456}
457}
```


Página 413

Listado B-5

SpreadsheetDate.java

```
1 / * ========================================================== ============================
2 * JCommon: una biblioteca de clases de propósito general gratuita para la plataforma Java (tm)
3 * ========================================================== ============================
4 *
5 * (C) Copyright 2000-2005, por Object Refinery Limited y contribuyentes.
```

```
 6 *
 7 * Información del proyecto: http://www.jfree.org/jcommon/index.html
 8 *
 9 * Esta biblioteca es software gratuito; puedes redistribuirlo y / o modificarlo
10 * bajo los términos de la Licencia Pública General Reducida GNU publicada por
11 * la Free Software Foundation; ya sea la versión 2.1 de la Licencia, o
12 * (a su elección) cualquier versión posterior.
13 *
14 * Esta biblioteca se distribuye con la esperanza de que sea útil, pero
15 * SIN NINGUNA GARANTÍA; incluso sin la garantía implícita de COMERCIABILIDAD
16 * o APTITUD PARA UN PROPÓSITO DETERMINADO. Ver el público general menor de GNU
17 * Licencia para más detalles.
18 *
19 * Debería haber recibido una copia de GNU Lesser General Public
20 * Licencia junto con esta biblioteca; si no es así, escriba al software libre
21 * Foundation, Inc., 51 Franklin Street, quinto piso, Boston, MA 02110-1301,
22 * Estados Unidos.
23 *
24 * [Java es una marca comercial o una marca comercial registrada de Sun Microsystems, Inc.
25 * en los Estados Unidos y otros países.]
26 *
27 * ----------
28 * SpreadsheetDate.java
29 * ----------
30 * (C) Copyright 2000-2005, por Object Refinery Limited y contribuyentes.
31 *
32 * Autor original: David Gilbert (para Object Refinery Limited);
33 * Colaborador (es): -;
34 *
35 * $ Id: SpreadsheetDate.java, v 1.8 2005/11/03 09:25:39 mungady Exp $
36 *
37 * Cambios
38 * -------
39 * 11 de octubre de 2001: Versión 1 (DG);
40 * 05-Nov-2001: Se agregaron los métodos getDescription () y setDescription () (DG);
41 * 12 de noviembre de 2001: nombre cambiado de ExcelDate.java a SpreadsheetDate.java (DG);
42 *                          Se corrigió un error al calcular el día, mes y año a partir de la serie.
43 *                          número (DG);
44 * 24-Ene-2002: Se corrigió un error al calcular el número de serie del día,
45 *                          mes y año. Gracias a Trevor Hills por el informe (DG);
46 * 29 de mayo de 2002: Se agregó el método equals (Object) (SourceForge ID 558850) (DG);
47 * 03-Oct-2002: Errores corregidos reportados por Checkstyle (DG);
48 * 13-Mar-2003: Implementado Serializable (DG);
49 * 04-Sep-2003: Métodos isInRange () completados (DG);
50 * 05-Sep-2003: Implementado Comparable (DG);
51 * 21-Oct-2003: Se agregó el método hashCode () (DG);
52 *
53 * /
54
55 paquete org.jfree.date;
56
57 import java.util.Calendar;
58 import java.util.Date;
59
60 / **
61 * Representa una fecha usando un número entero, de manera similar a la
62 * implementación en Microsoft Excel. El rango de fechas admitido es
```


Listado B-5 (continuación)

SpreadsheetDate.java

```
63 * 1-ene-1900 a 31-dic-9999.
64 * <P>
65 * Tenga en cuenta que hay un error deliberado en Excel que reconoce el año
66 * 1900 como año bisiesto cuando en realidad no es un año bisiesto. Puedes encontrar más
67 * información en el sitio web de Microsoft en el artículo Q181370:
68 * <P>
69 * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
70 * <P>
71 * Excel usa la convención de que 1-Jan-1900 = 1. Esta clase usa el
72 * convención 1-Ene-1900 = 2.
73 * El resultado es que el número de días en esta clase será diferente al
74 * Cifra de Excel para enero y febrero de 1900 ... pero luego Excel agrega un extra
75 * día (29-Feb-1900 que en realidad no existe!) Y desde ese momento en adelante
76 * los números de los días coincidirán.
77 *
78 * @ autor David Gilbert
79 * /
80 public class SpreadsheetDate extiende SerialDate {
81
82 / ** Para serialización. * /
83 private static final long serialVersionUID = -2039586705374454461L;
84
85 / **
```

```
86 * El número de día (1-Ene-1900 = 2, 2-Ene-1900 = 3, ..., 31-Dic-9999 =
87 * 2958465).
88 * /
89 serie int privada;
90
91 / ** El día del mes (1 al 28, 29, 30 o 31 según el mes). * /
92 día int privado;
93
94 / ** El mes del año (1 a 12). * /
95 int mes privado;
96
97 / ** El año (1900 a 9999). * /
98 int año privado;
99
100 / ** Una descripción opcional para la fecha. * /
101 descripción de cadena privada;
102
103 / **
104 * Crea una nueva instancia de fecha.
105 *
106 * @param día el día (en el rango de 1 a 28/29/30/31).
107 * @param mes el mes (en el rango de 1 a 12).
108 * @param año el año (en el rango 1900 a 9999).
109 * /
110 public SpreadsheetDate (día int final, mes int final, año int final) {
111
112 if ((año> = 1900) && (año <= 9999)) {
113 this.year = año;
114 }
115 demás {
116 lanzar una nueva IllegalArgumentException (
117 "El argumento 'año' debe estar en el rango de 1900 a 9999."
118 );
119 }
120
121 if ((mes> = MonthConstants.ENUARY)
122 && (mes <= MonthConstants.DECEMBER)) {
123 this.month = mes;
124 }
```


Página 415

Listado B-5 (continuación)

SpreadsheetDate.java

```
125 demás {
126 lanzar una nueva IllegalArgumentException (
127 "El argumento 'mes' debe estar en el rango de 1 a 12."
128 );
129 }
130
131 if ((día> = 1) && (día <= SerialDate.lastDayOfMonth (mes, año))) {
132 this.day = día;
133 }
134 demás {
135 lanzar una nueva IllegalArgumentException ("Argumento de 'día' no válido.");
136 }
137
138 // el número de serie debe estar sincronizado con el día-mes-año ...
139 this.serial = calcSerial (día, mes, año);
140
141 this.description = null;
142
143}
144
145 / **
146 * Constructor estándar: crea un nuevo objeto de fecha que representa el
147 * número de día especificado (que debe estar en el rango de 2 a 2958465.
148 *
149 * @param serial el número de serie del día (rango: 2 a 2958465).
150 * /
151 public SpreadsheetDate (final int serial) {
152
153 if ((serial> = SERIAL_LOWER_BOUND) && (serial <= SERIAL_UPPER_BOUND)) {
154 this.serial = serial;
155 }
156 demás {
157 lanzar una nueva IllegalArgumentException (
158 "SpreadsheetDate: El número de serie debe estar en el rango de 2 a 2958465.");
159 }
160
161 // el día-mes-año debe estar sincronizado con el número de serie ...
162 calcDayMonthYear ();
163
164}
165
```

```
166 / **
167 * Devuelve la descripción que se adjunta a la fecha. No lo es
168 * requiere que una fecha tenga una descripción, pero para algunas aplicaciones
169 * es útil.
170 *
171 * @return La descripción que se adjunta a la fecha.
172 * /
173 public String getDescription () {
174                 devuelva esta descripción;
175}
176
177 / **
178 * Establece la descripción de la fecha.
179 *
180 * @param description la descripción para esta fecha (<code> null </code>
181 *                                                  permitido).
182 * /
183 public void setDescription (descripción final de la cadena) {
184                 this.description = descripción;
185}
186
```


Página 416

Listado B-5 (continuación)

SpreadsheetDate.java

```
187 / **
188 * Devuelve el número de serie de la fecha, donde 1 de enero de 1900 = 2
189 * (esto corresponde, casi, al sistema de numeración utilizado en Microsoft
190 * Excel para Windows y Lotus 1-2-3).
191 *
192 * @return El número de serie de esta fecha.
193 * /
194 public int toSerial () {
195                 devuelve this.serial;
196}
197
198 / **
199 * Devuelve un <code> java.util.Date </code> equivalente a esta fecha.
200 *
201 * @return La fecha.
202 * /
203 public Date toDate () {
204                 Calendario calendario final = Calendar.getInstance ();
205                 calendar.set (getYYYY (), getMonth () - 1, getDayOfMonth (), 0, 0, 0);
206                 return calendar.getTime ();
207}
208
209 / **
210 * Devuelve el año (suponga un rango válido de 1900 a 9999).
211 *
212 * @return El año.
213 * /
214 public int getYYYY () {
215                 devuelva este año;
216}
217
218 / **
219 * Devuelve el mes (enero = 1, febrero = 2, marzo = 3).
220 *
221 * @return El mes del año.
222 * /
223 public int getMonth () {
224                 devuelva este mes;
225}
226
227 / **
228 * Devuelve el día del mes.
229 *
230 * @return El día del mes.
231 * /
232 public int getDayOfMonth () {
233                 volver este.día;
234}
235
236 / **
237 * Devuelve un código que representa el día de la semana.
238 * <P>
239 * Los códigos se definen en la clase {@link SerialDate} como:
240 * <code> DOMINGO </code>, <code> LUNES </code>, <code> MARTES </code>,
241 * <code> MIÉRCOLES </code>, <code> JUEVES </code>, <code> VIERNES </code> y
242 * <code> SÁBADO </code>.
243 *
244 * @return Un código que representa el día de la semana.
245 * /
```

```
246 public int getDayOfWeek () {
247                 return (esta.serie + 6)% 7 + 1;
248}
```


Página 417

Listado B-5 (continuación)

SpreadsheetDate.java

```
249
250 / **
251 * Prueba la igualdad de esta fecha con un objeto arbitrario.
252 * <P>
253 * Este método devolverá verdadero SÓLO si el objeto es una instancia del
254 * {@link SerialDate} clase base, y representa el mismo día que este
255 * {@link SpreadsheetDate}.
256 *
257 * @param object el objeto a comparar (<code> null </code> permitido).
258 *
259 * @return A booleano.
260 * /
261 public boolean equals (objeto de objeto final) {
262
263                 if (instancia de objeto de SerialDate) {
264                         final SerialDate s = (SerialDate) objeto;
265                         return (s.toSerial () == this.toSerial ());
266                 }
267                 demás {
268                         falso retorno;
269                 }
270
271}
272
273 / **
274 * Devuelve un código hash para esta instancia de objeto.
275 *
276 * @return Un código hash.
277 * /
278 public int hashCode () {
279                 volver aSerial ();
280}
281
282 / **
283 * Devuelve la diferencia (en días) entre esta fecha y la especificada
284 * 'otra' fecha.
285 *
286 * @param other la fecha con la que se compara.
287 *
288 * @return La diferencia (en días) entre esta fecha y la especificada
289 *                         'otra' fecha.
290 * /
291 public int compare (final SerialDate otro) {
292                 devuelve this.serial - other.toSerial ();
293}
294
295 / **
296 * Implementa el método requerido por la interfaz Comparable.
297 *
298 * @param otro el otro objeto (normalmente otro SerialDate).
299 *
300 * @return Un entero negativo, cero o un entero positivo como este objeto
301 *                         es menor, igual o mayor que el objeto especificado.
302 * /
303 public int compareTo (objeto final otro) {
304                 return compare ((SerialDate) otro);
305}
306
307 / **
308 * Devuelve verdadero si este SerialDate representa la misma fecha que el
309 * Fecha de serie especificada.
310 *
```

Página 418

Listado B-5 (continuación)

SpreadsheetDate.java

```
311 * @param other la fecha con la que se compara.
312 *
313 * @return <code> true </code> si este SerialDate representa la misma fecha que
314 *                             el SerialDate especificado.
315 * /
316 public boolean isOn (final SerialDate otro) {
317                 return (this.serial == other.toSerial ());
318}
319
320 / **
321 * Devuelve verdadero si este SerialDate representa una fecha anterior en comparación con
322 * el SerialDate especificado.
323 *
324 * @param other la fecha con la que se compara.
325 *
326 * @return <code> true </code> si este SerialDate representa una fecha anterior
327 *                             en comparación con el SerialDate especificado.
328 * /
329 public boolean isBefore (final SerialDate other) {
330                 return (this.serial <other.toSerial ());
331}
332
333 / **
334 * Devuelve verdadero si este SerialDate representa la misma fecha que el
335 * Fecha de serie especificada.
336 *
337 * @param other la fecha con la que se compara.
338 *
339 * @return <code> true </code> si este SerialDate representa la misma fecha
340 *                             como el SerialDate especificado.
341 * /
342 public boolean isOnOrBefore (final SerialDate otro) {
343                 return (this.serial <= other.toSerial ());
344}
345
346 / **
347 * Devuelve verdadero si este SerialDate representa la misma fecha que el
348 * Fecha de serie especificada.
349 *
350 * @param other la fecha con la que se compara.
351 *
352 * @return <code> true </code> si este SerialDate representa la misma fecha
353 *                             como el SerialDate especificado.
354 * /
355 public boolean isAfter (final SerialDate otro) {
356                 return (this.serial> other.toSerial ());
357}
358
359 / **
360 * Devuelve verdadero si este SerialDate representa la misma fecha que el
361 * Fecha de serie especificada.
362 *
363 * @param other la fecha con la que se compara.
364 *
365 * @return <code> true </code> si este SerialDate representa la misma fecha que
366 *                             el SerialDate especificado.
367 * /
368 public boolean isOnOrAfter (final SerialDate otro) {
369                 return (this.serial> = other.toSerial ());
370}
371
372 / **
373 * Devuelve <code> true </code> si este {@link SerialDate} está dentro del
```


Página 419

Listado B-5 (continuación)
SpreadsheetDate.java

```
374 * rango especificado (INCLUIDO). El orden de fecha de d1 y d2 no es
375 * importante.
376 *
377 * @param d1 una fecha límite para el rango.
378 * @param d2 la otra fecha límite para el rango.
379 *
380 * @return A booleano.
381 * /
382 public boolean isInRange (final SerialDate d1, final SerialDate d2) {
383                 return isInRange (d1, d2, SerialDate.INCLUDE_BOTH);
384}
385
386 / **
387 * Devuelve verdadero si este SerialDate está dentro del rango especificado (llamador
388 * especifica si se incluyen o no los puntos finales). El orden de d1
389 * y d2 no es importante.
390 *
391 * @param d1 una fecha límite para el rango.
392 * @param d2 una segunda fecha límite para el rango.
393 * @param incluye un código que controla si el inicio y el final
394 *                                      las fechas están incluidas en el rango.
395 *
396 * @return <code> true </code> si este SerialDate está dentro del especificado
397 *                          distancia.
398 * /
399 public boolean isInRange (final SerialDate d1, final SerialDate d2,
400                                                         final int include) {
401                 final int s1 = d1.toSerial ();
402                 final int s2 = d2.toSerial ();
403                 final int inicio = Math.min (s1, s2);
404                 final int end = Math.max (s1, s2);
405
406                 final int s = toSerial ();
407                 if (include == SerialDate.INCLUDE_BOTH) {
408                         return (s> = inicio && s <= fin);
409                 }
410                 else if (include == SerialDate.INCLUDE_FIRST) {
411                         return (s> = inicio && s <fin);
412                 }
413                 else if (include == SerialDate.INCLUDE_SECOND) {
414                         return (s> inicio && s <= fin);
415                 }
416                 demás {
417                         return (s> inicio && s <fin);
418                 }
419}
420
421 / **
422 * Calcule el número de serie a partir del día, mes y año.
423 * <P>
424 * 1-enero-1900 = 2.
425 *
426 * @param d el día.
427 * @param m el mes.
428 * @param y el año.
429 *
430 * @ Devuelve el número de serie del día, mes y año.
431 * /
432 private int calcSerial (final int d, final int m, final int y) {
433                 int final yy = ((y - 1900) * 365) + SerialDate.leapYearCount (y - 1);
434                 int mm = SerialDate.AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH [m];
435                 if (m> MonthConstants.FEBRUARY) {
```


Página 420

Listado B-5 (continuación)
SpreadsheetDate.java

```
436                         if (SerialDate.isLeapYear (y)) {
437                                 mm = mm + 1;
438                         }
439                 }
440                 final int dd = d;
441                 devuelve aa + mm + dd + 1;
442}
443
444 / **
445 * Calcule el día, mes y año a partir del número de serie.
446 * /
447 private void calcDayMonthYear () {
448
449                 // obtener el año de la fecha de serie
450                 días int finales = this.serial - SERIAL_LOWER_BOUND;
```

```
451             // sobreestimado porque ignoramos los días bisiestos
452             int final sobreestimadoYYYY = 1900 + (días / 365);
453             saltos int finales = SerialDate.leapYearCount (sobreestimadoYYYY);
454             final int nonleapdays = días - saltos;
455             // subestimado porque sobreestimamos años
456             int subestimadoYYYY = 1900 + (días no bisiestos / 365);
457
458             if (subestimadoYYYY == sobreestimadoYYYY) {
459                 this.year = subestimadoYYYY;
460             }
461             demás {
462                 int ss1 = calcSerial (1, 1, subestimadoYYYY);
463                 while (ss1 <= this.serial) {
464                     subestimadoYYYY = subestimadoYYYY + 1;
465                     ss1 = calcSerial (1, 1, subestimadoYYYY);
466                 }
467                 this.year = subestimadoYYYY - 1;
468             }
469
470             final int ss2 = calcSerial (1, 1, this.year);
471
472             int [] daysToEndOfPrecedingMonth
473                     = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
474
475             if (isLeapYear (this.year)) {
476                 daysToEndOfPrecedingMonth
477                         = LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
478             }
479
480             // obtener el mes a partir de la fecha de serie
481             int mm = 1;
482             int sss = ss2 + daysToEndOfPrecedingMonth [mm] - 1;
483             while (sss <this.serial) {
484                 mm = mm + 1;
485                 sss = ss2 + daysToEndOfPrecedingMonth [mm] - 1;
486             }
487             this.month = mm - 1;
488
489             // lo que queda es d (+1);
490             this.day = this.serial - ss2
491                             - daysToEndOfPrecedingMonth [this.month] + 1;
492
493}
494
495}
```


Página 421

Listado B-6

RelativeDayOfWeekRule.java

```
1 / * ========================================== ========================
2 * JCommon: una biblioteca de clases de propósito general gratuita para la plataforma Java (tm)
3 * ========================================== ========================
4 *
5 * (C) Copyright 2000-2005, por Object Refinery Limited y contribuyentes.
6 *
7 * Información del proyecto: http://www.jfree.org/jcommon/index.html
8 *
9 * Esta biblioteca es software gratuito; puedes redistribuirlo y / o modificarlo
10 * bajo los términos de la Licencia Pública General Reducida GNU publicada por
11 * la Free Software Foundation; ya sea la versión 2.1 de la Licencia, o
12 * (a su elección) cualquier versión posterior.
13 *
14 * Esta biblioteca se distribuye con la esperanza de que sea útil, pero
15 * SIN NINGUNA GARANTÍA; incluso sin la garantía implícita de COMERCIABILIDAD
16 * o APTITUD PARA UN PROPÓSITO DETERMINADO. Ver el público general menor de GNU
17 * Licencia para más detalles.
18 *
19 * Debería haber recibido una copia de GNU Lesser General Public
20 * Licencia junto con esta biblioteca; si no es así, escriba al software libre
21 * Foundation, Inc., 51 Franklin Street, quinto piso, Boston, MA 02110-1301,
22 * Estados Unidos.
23 *
24 * [Java es una marca comercial o una marca comercial registrada de Sun Microsystems, Inc.
25 * en los Estados Unidos y otros países.]
26 *
27 * ----------------------------
28 * RelativeDayOfWeekRule.java
29 * ----------------------------
30 * (C) Copyright 2000-2003, por Object Refinery Limited y contribuyentes.
31 *
32 * Autor original: David Gilbert (para Object Refinery Limited);
33 * Colaborador (es): -;
```

```
34 *
35 * $ Id: RelativeDayOfWeekRule.java, v 1.6 2005/11/16 15:58:40 taqua Exp $
36 *
37 * Cambios (desde el 26 de octubre de 2001)
38 * -----------------------
39 * 26 de octubre de 2001: paquete modificado a com.jrefinery.date. *;
40 * 03-Oct-2002: Errores corregidos reportados por Checkstyle (DG);
41 *
42 * /
43
44 paquete org.jfree.date;
45
46 / **
47 * Una regla de fecha anual que devuelve una fecha para cada año basada en (a) una
48 * regla de referencia; (b) un día de la semana; y (c) un parámetro de selección
49 * (SerialDate.PRECEDING, SerialDate. NEAREST, SerialDate.FOLLOWING).
50 * <P>
51 * Por ejemplo, el Viernes Santo se puede especificar como 'el Viernes PRECEDENTE de Pascua
52 * Domingo '.
53 *
54 * @ autor David Gilbert
55 * /
56 clase pública RelativeDayOfWeekRule extiende AnnualDateRule {
57
58 / ** Una referencia a la regla de fecha anual en la que se basa esta regla. * /
59 subregla privada AnnualDateRule;
60
61 / **
62 * El día de la semana (SerialDate.MONDAY, SerialDate.TUESDAY, etc.).
```


Página 422

Listado B-6 (continuación)

RelativeDayOfWeekRule.java

```
63 * /
64 privado int dayOfWeek;
sesenta y cinco
66 / ** Especifica qué día de la semana (PRECEDENTE, MÁS CERCANO o SIGUIENTE). * /
67 familiar íntimo privado;
68
69 / **
70 * Constructor predeterminado: crea una regla para el lunes siguiente al 1 de enero.
71 * /
72 public RelativeDayOfWeekRule () {
73                 this (new DayAndMonthRule (), SerialDate.MONDAY, SerialDate.FOLLOWING);
74}
75
76 / **
77 * Constructor estándar: crea una regla basada en la subregla proporcionada.
78 *
79 * @param subregla la regla que determina la fecha de referencia.
80 * @param dayOfWeek el día de la semana relativo a la fecha de referencia.
81 * @param relativo indica * qué * día de la semana (anterior, más cercano
82 *                                                 o siguiendo).
83 * /
84 public RelativeDayOfWeekRule (subregla final de AnnualDateRule,
85                                final int dayOfWeek, final int relativo) {
86                 this.subrule = subregla;
87                 this.dayOfWeek = dayOfWeek;
88                 this.relative = relativo;
89}
90
91 / **
92 * Devuelve la subregla (también llamada regla de referencia).
93 *
94 * @return La regla de fecha anual que determina la fecha de referencia para este
95 *                                 regla.
96 * /
97 public AnnualDateRule getSubrule () {
98                 devuelve esta subrregla;
99}
100
101 / **
102 * Establece la subregla.
103 *
104 * @param subregla la regla de fecha anual que determina la fecha de referencia
105 *                                                 para esta regla.
106 * /
107 public void setSubrule (subregla final de AnnualDateRule) {
108                 this.subrule = subregla;
109}
110
111 / **
112 * Devuelve el día de la semana para esta regla.
113 *
```

```
114 * @return el día de la semana para esta regla.
115 * /
116 public int getDayOfWeek () {
117                 return this.dayOfWeek;
118}
119
120 / **
121 * Establece el día de la semana para esta regla.
122 *
123 * @param dayOfWeek el día de la semana (SerialDate.MONDAY,
124 *                                       SerialDate.TUESDAY, etc.).
```


Página 423

Listado B-6 (continuación)

RelativeDayOfWeekRule.java

```
125 * /
126 public void setDayOfWeek (final int dayOfWeek) {
127                 this.dayOfWeek = dayOfWeek;
128}
129
130 / **
131 * Devuelve el atributo 'relativo', que determina * cuál *
132 * día de la semana en el que estamos interesados (SerialDate.PRECEDING,
133 * SerialDate. CERCA o SerialDate.FOLLOWING).
134 *
135 * @return El atributo 'relativo'.
136 * /
137 public int getRelative () {
138                 devuelve este pariente;
139}
140
141 / **
142 * Establece el atributo 'relativo' (SerialDate.PRECEDING, SerialDate.NEAREST,
143 * SerialDate. SIGUIENTE).
144 *
145 * @param relativo determina * qué * día de la semana es seleccionado por este
146 *                                       regla.
147 * /
148 public void setRelative (final int relativo) {
149                 this.relative = relativo;
150}
151
152 / **
153 * Crea un clon de esta regla.
154 *
155 * @return un clon de esta regla.
156 *
157 * @throws CloneNotSupportedException, esto nunca debería suceder.
158 * /
159 public Object clone () lanza CloneNotSupportedException {
160                 final RelativeDayOfWeekRule duplicado
161                         = (RelativeDayOfWeekRule) super.clone ();
162                 duplicate.subrule = (AnnualDateRule) duplicate.getSubrule (). clone ();
163                 devolver duplicado;
164}
165
166 / **
167 * Devuelve la fecha generada por esta regla, para el año especificado.
168 *
169 * @param año el año (1900 & lt; = año & lt; = 9999).
170 *
171 * @return La fecha generada por la regla para el año dado (posiblemente
172 *                         <code> nulo </code>).
173 * /
174 public SerialDate getDate (final int year) {
175
176                 // comprobar argumento ...
177                 if ((año <SerialDate.MINIMUM_YEAR_SUPPORTED)
178                         || (año> SerialDate.MAXIMUM_YEAR_SUPPORTED)) {
179                         lanzar una nueva IllegalArgumentException (
180                                 "RelativeDayOfWeekRule.getDate (): año fuera del rango válido.");
181                 }
182
183                 // calcula la fecha ...
184                 SerialDate result = null;
185                 final SerialDate base = this.subrule.getDate (año);
186
```

Página 424

Listado B-6 (continuación)

RelativeDayOfWeekRule.java

```
187             if (base! = null) {
188                     cambiar (esto.relativo) {
189                             caso (SerialDate.PRECEDING):
190                                     resultado = SerialDate.getPreviousDayOfWeek (this.dayOfWeek,
191                                             base);
192                                     rotura;
193                             case (SerialDate. NEAREST):
194                                     resultado = SerialDate.getNearestDayOfWeek (this.dayOfWeek,
195                                             base);
196                                     rotura;
197                             case (SerialDate.FOLLOWING):
198                                     resultado = SerialDate.getFollowingDayOfWeek (this.dayOfWeek,
199                                             base);
200                                     rotura;
201                             defecto:
202                                     rotura;
203                     }
204             }
205             devolver resultado;
206
207}
208
209}
```


Página 425

Listado B-7

DayDate.java (final)

```
1 / * ============================================= ========================
2 * JCommon: una biblioteca de clases de propósito general gratuita para la plataforma Java (tm)
3 * ============================================= ========================
4 *
5 * (C) Copyright 2000-2005, por Object Refinery Limited y contribuyentes.
...
36 * /
37 paquete org.jfree.date;
38
39 import java.io.Serializable;
40 import java.util. *;
41
42 / **
43 * Una clase abstracta que representa fechas inmutables con una precisión de
44 * un día. La implementación mapeará cada fecha a un número entero que
45 * representa un número ordinal de días desde algún origen fijo.
46 *
47 * ¿Por qué no usar java.util.Date? Lo haremos, cuando tenga sentido. A veces,
48 * java.util.Date puede ser * demasiado * preciso: representa un instante en el tiempo,
49 * con una precisión de 1/1000 de segundo (con la fecha en sí dependiendo de la
50 * zona horaria). A veces solo queremos representar un día en particular (por ejemplo, 21
51 * enero de 2015) sin preocuparnos por la hora del día, o el
52 * zona horaria, o cualquier otra cosa. Para eso hemos definido DayDate.
53 *
54 * Utilice DayDateFactory.makeDate para crear una instancia.
55 *
56 * @autor David Gilbert
57 * @autor Robert C. Martin hizo muchas refactorizaciones.
58 * /
59
60 clases públicas abstractas DayDate implementa Comparable, Serializable {
61 public abstract int getOrdinalDay ();
62 public abstract int getYear ();
63 mes público abstracto getMonth ();
64 public abstract int getDayOfMonth ();
sesenta y cinco
66 día abstracto protegido getDayOfWeekForOrdinalZero ();
67
68 PublicDate plusDays (int días) {
69 return DayDateFactory.makeDate (getOrdinalDay () + días);
70}
71
72 público DayDate plusMonths (int meses) {
73 int thisMonthAsOrdinal = getMonth (). ToInt () - Month.ENUARY.toInt ();
74 int thisMonthAndYearAsOrdinal = 12 * getYear () + thisMonthAsOrdinal;
75 int resultMonthAndYearAsOrdinal = thisMonthAndYearAsOrdinal + meses;
76 int resultYear = resultMonthAndYearAsOrdinal / 12;
77 int resultMonthAsOrdinal = resultMonthAndYearAsOrdinal% 12 + Month.ENUARY.toInt ();
78 meses resultMonth = Month.fromInt (resultMonthAsOrdinal);
79 int resultDay = correctLastDayOfMonth (getDayOfMonth (), resultMonth, resultYear);
80 return DayDateFactory.makeDate (resultDay, resultMonth, resultYear);
81}
82
83 público DayDate plusYears (int años) {
84 int resultYear = getYear () + años;
85 int resultDay = correctLastDayOfMonth (getDayOfMonth (), getMonth (), resultYear);
86 return DayDateFactory.makeDate (resultDay, getMonth (), resultYear);
87}
88
89 private int correctLastDayOfMonth (int día, mes mes, int año) {
90 int lastDayOfMonth = DateUtil.lastDayOfMonth (mes, año);
91 if (día> lastDayOfMonth)
```


Página 426

Listado B-7 (continuación)

DayDate.java (final)

```
92 day = lastDayOfMonth;
93 día de regreso;
94}
95
96 público DayDate getPreviousDayOfWeek (Day targetDayOfWeek) {
97 int offsetToTarget = targetDayOfWeek.toInt () - getDayOfWeek (). ToInt ();
98 si (offsetToTarget> = 0)
```

```
 99 offsetToTarget - = 7;
100 return plusDays (offsetToTarget);
101}
102
103 public DayDate getFollowingDayOfWeek (Day targetDayOfWeek) {
104 int offsetToTarget = targetDayOfWeek.toInt () - getDayOfWeek (). ToInt ();
105 si (offsetToTarget <= 0)
106 offsetToTarget + = 7;
107 return plusDays (offsetToTarget);
108}
109
110 public DayDate getNearestDayOfWeek (Day targetDayOfWeek) {
111 int offsetToThisWeeksTarget = targetDayOfWeek.toInt () - getDayOfWeek (). ToInt ();
112 int offsetToFutureTarget = (offsetToThisWeeksTarget + 7)% 7;
113 int offsetToPreviousTarget = offsetToFutureTarget - 7;
114
115 si (offsetToFutureTarget> 3)
116 return plusDays (offsetToPreviousTarget);
117 más
118 return plusDays (offsetToFutureTarget);
119}
120
121 public DayDate getEndOfMonth () {
122 Mes mes = getMonth ();
123 int año = getYear ();
124 int lastDay = DateUtil.lastDayOfMonth (mes, año);
125 return DayDateFactory.makeDate (lastDay, month, year);
126}
127
128 public Date toDate () {
129 Calendario calendario final = Calendar.getInstance ();
130 int ordinalMonth = getMonth (). ToInt () - Month.ENUARY.toInt ();
131 calendar.set (getYear (), ordinalMonth, getDayOfMonth (), 0, 0, 0);
132 return calendar.getTime ();
133}
134
135 public String toString () {
136 return String.format ("% 02d-% s-% d", getDayOfMonth (), getMonth (), getYear ());
137}
138
139 public Day getDayOfWeek () {
Día de inicio de 140 días = getDayOfWeekForOrdinalZero ();
141 int startOffset = startDay.toInt () - Day.SUNDAY.toInt ();
142 int ordinalOfDayOfWeek = (getOrdinalDay () + startOffset)% 7;
143 return Day.fromInt (ordinalOfDayOfWeek + Day.SUNDAY.toInt ());
144}
145
146 public int daysSince (DayDate date) {
147 return getOrdinalDay () - date.getOrdinalDay ();
148}
149
150 public boolean isOn (DayDate otro) {
151 return getOrdinalDay () == other.getOrdinalDay ();
152}
153
```


Página 427

Listado B-7 (continuación)

DayDate.java (final)

```
154 public boolean isBefore (DayDate otro) {
155 return getOrdinalDay () <otro.getOrdinalDay ();
156}
157
158 public boolean isOnOrBefore (DayDate otro) {
159 return getOrdinalDay () <= otro.getOrdinalDay ();
160}
161
162 public boolean isAfter (DayDate otro) {
163 return getOrdinalDay ()> otro.getOrdinalDay ();
164}
165
166 public boolean isOnOrAfter (DayDate otro) {
167 return getOrdinalDay ()> = otro.getOrdinalDay ();
168}
169
170 public boolean isInRange (DayDate d1, DayDate d2) {
171 return isInRange (d1, d2, DateInterval.CLOSED);
172}
173
174 public boolean isInRange (DayDate d1, DayDate d2, DateInterval interval) {
175 int izquierda = Math.min (d1.getOrdinalDay (), d2.getOrdinalDay ());
176 int right = Math.max (d1.getOrdinalDay (), d2.getOrdinalDay ());
177 return interval.isIn (getOrdinalDay (), izquierda, derecha);
178}
```

```
179}
```


Página 428

Listado B-8

Month.java (final)

```
1 paquete org.jfree.date;
2
3 import java.text.DateFormatSymbols;
4
5 mes public enum {
6 DE ENERO (1), FEBRERO (2), MARZO (3),
7 DE ABRIL (4), MAYO (5), JUNIO (6),
8 DE JULIO (7), AGOSTO (8), SEPTIEMBRE (9),
9 DE OCTUBRE (10), NOVIEMBRE (11), DICIEMBRE (12);
10 DateFormatSymbols estáticos privados dateFormatSymbols = new DateFormatSymbols ();
11 int final estático privado [] LAST_DAY_OF_MONTH =
12 {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
13
14 índice int privado;
15
16 meses (índice int) {
17 this.index = índice;
18}
19
20 mes público estático fromInt (int monthIndex) {
21 para (Month m: Month.values ()) {
22 si (m.index == monthIndex)
23 return m;
24}
25 lanzar una nueva IllegalArgumentException ("Índice de mes no válido" + monthIndex);
26}
27
28 public int lastDay () {
29 return LAST_DAY_OF_MONTH [índice];
30}
31
32 public int quarter () {
33 return 1 + (índice - 1) / 3;
34}
35
36 public String toString () {
37 return dateFormatSymbols.getMonths () [índice - 1];
38}
39
40 cadena pública toShortString () {
41 return dateFormatSymbols.getShortMonths () [índice - 1];
42}
43
```

```
44 análisis de mes estático público (String s) {
45 s = s.trim ();
46 para (Month m: Month.values ())
47 si (m. Partidos)
48 return m;
49
50 intento {
51 return fromInt (Integer.parseInt (s));
52}
53 captura (NumberFormatException e) {}
54 lanzar una nueva IllegalArgumentException ("Mes no válido" + s);
55}
56
57 coincidencias booleanas privadas (String s) {
58 devuelve un caso igual a Ignore (toString ()) ||
59 s.equalsIgnoreCase (toShortString ());
60}
61
62 public int toInt () {
63 índice de retorno;
64}
sesenta y cinco }
```


Página 429

Listado B-9

Day.java (final)

```
1 paquete org.jfree.date;
2
3 importar java.util.Calendar;
4 import java.text.DateFormatSymbols;
5
6 día de enumeración pública {
7 LUNES (Calendario.LUNES),
8 MARTES (Calendario. MARTES),
MIERCOLES 9 (Calendario MIERCOLES),
10 JUEVES (Calendario. JUEVES),
11 VIERNES (Calendario VIERNES),
12 SÁBADO (Calendario. SÁBADO),
13 DOMINGO (Calendario DOMINGO);
14
15 índice int final privado;
16 DateFormatSymbols estáticos privados dateSymbols = new DateFormatSymbols ();
17
18 días (int día) {
19 índice = día;
20}
21
22 public static Day fromInt (int index) arroja IllegalArgumentException {
23 para (Día d: Día.valores ())
24 si (d.index == índice)
25 return d;
26 lanzar una nueva IllegalArgumentException (
27 String.format ("Índice de días ilegales:% d.", Índice));
28}
29
El análisis de 30 días estáticos públicos (String s) arroja IllegalArgumentException {
31 Cadena [] shortWeekdayNames =
32 dateSymbols.getShortWeekdays ();
33 String [] weekDayNames =
34 dateSymbols.getWeekdays ();
35
36 s = s.trim ();
37 para (Day day: Day.values ()) {
38 if (s.equalsIgnoreCase (shortWeekdayNames [day.index]) ||
39 s.equalsIgnoreCase (weekDayNames [day.index])) {
40 día de regreso;
41}
42}
43 lanzar una nueva IllegalArgumentException (
44 String.format ("% s no es una cadena de día de la semana válida", s));
45}
46
47 public String toString () {
48 return dateSymbols.getWeekdays () [índice];
49}
50
51 public int toInt () {
52 índice de retorno;
53}
54}
```

Página 430

Listado B-10

DateInterval.java (final)

```
1 paquete org.jfree.date;
2
3 public enum DateInterval {
4 ABIERTO {
5 public boolean isIn (int d, int left, int right) {
6 volver d> izquierda && d <derecha;
7}
8},
9 CLOSED_LEFT {
10 public boolean isIn (int d, int left, int right) {
11 return d> = izquierda && d <derecha;
12}
13},
14 CLOSED_RIGHT {
15 public boolean isIn (int d, int left, int right) {
16 return d> izquierda && d <= derecha;
17}
18},
19 CERRADO {
20 public boolean isIn (int d, int left, int right) {
21 return d> = izquierda && d <= derecha;
22}
23};
24
25 public abstract boolean isIn (int d, int left, int right);
26}
```

Página 431

Listado B-11

WeekInMonth.java (final)

```
1 paquete org.jfree.date;
2
3 public enum WeekInMonth {
4 PRIMERO (1), SEGUNDO (2), TERCERO (3), CUARTO (4), ÚLTIMO (0);
5 índice int final privado;
6
7 WeekInMonth (indice int) {
8 this.index = índice;
9}
10
11 public int toInt () {
12 índice de retorno;
13}
14}
```


Página 432

Listado B-12

WeekdayRange.java (final)

```
1 paquete org.jfree.date;
2
3 public enum WeekdayRange {
4 ÚLTIMO, MÁS CERCANO, SIGUIENTE
5}
```


Página 433

Listado B-13

DateUtil.java (final)

```
 1 paquete org.jfree.date;
 2
 3 import java.text.DateFormatSymbols;
 4
 5 clase pública DateUtil {
 6 DateFormatSymbols estáticos privados dateFormatSymbols = new DateFormatSymbols ();
 7
 8 Cadena estática pública [] getMonthNames () {
 9 return dateFormatSymbols.getMonths ();
10}
11
12 public static boolean isLeapYear (int year) {
13 booleano cuarto = año% 4 == 0;
14 centésimo booleano = año% 100 == 0;
15 booleano cuatrocientos = año% 400 == 0;
16 devuelve cuarto && (! Centésimo || cuatrocientos);
17}
```

```
18
19 public static int lastDayOfMonth (mes mes, int año) {
20 if (month == Month.FEBRUARY && isLeapYear (año))
21 mes de regreso.lastDay () + 1;
22 más
23 regreso mes.último día ();
24}
25
26 public static int leapYearCount (int año) {
27 int leap4 = (año - 1896) / 4;
28 int leap100 = (año - 1800) / 100;
29 int leap400 = (año - 1600) / 400;
30 retorno leap4 - leap100 + leap400;
31}
32}
```


Página 434

Listado B-14

DayDateFactory.java (final)

```
1 paquete org.jfree.date;
2
3 clase pública abstracta DayDateFactory {
4 fábrica privada estática DayDateFactory = new SpreadsheetDateFactory ();
5 setInstance vacío estático público (fábrica DayDateFactory) {
6 DayDateFactory.factory = fábrica;
7}
8
9 Resumen protegido DayDate _makeDate (int ordinal);
10 abstracto protegido DayDate _makeDate (int día, mes mes, int año);
11 Resumen protegido DayDate _makeDate (int día, int mes, int año);
12 resumen protegido DayDate _makeDate (java.util.Date fecha);
13 resumen protegido int _getMinimumYear ();
14 resumen protegido int _getMaximumYear ();
15
16 public static DayDate makeDate (int ordinal) {
17 return factory._makeDate (ordinal);
18}
19
20 público estático DayDate makeDate (int día, mes mes, int año) {
21 return factory._makeDate (día, mes, año);
22}
23
24 public static DayDate makeDate (int día, int mes, int año) {
25 return factory._makeDate (día, mes, año);
26}
27
28 public static DayDate makeDate (java.util.Date date) {
29 return factory._makeDate (fecha);
30}
31
32 public static int getMinimumYear () {
33 return factory._getMinimumYear ();
34}
35
```

```
36 public static int getMaximumYear () {
37 return factory._getMaximumYear ();
38}
39}
```


Página 435

Listado B-15

SpreadsheetDateFactory.java (final)

```
 1 paquete org.jfree.date;
 2
 3 importar java.util. *;
 4
 5 clase pública SpreadsheetDateFactory extiende DayDateFactory {
 6 public DayDate _makeDate (int ordinal) {
 7 return new SpreadsheetDate (ordinal);
 8}
 9
10 public DayDate _makeDate (int día, mes mes, int año) {
11 return new SpreadsheetDate (día, mes, año);
12}
13
14 public DayDate _makeDate (int día, int mes, int año) {
15 return new SpreadsheetDate (día, mes, año);
dieciséis }
17
18 public DayDate _makeDate (Date date) {
19 calendario final GregorianCalendar = new GregorianCalendar ();
20 calendar.setTime (fecha);
21 return new SpreadsheetDate (
22 calendar.get (Calendar.DATE),
23 Month.fromInt (calendar.get (Calendar.MONTH) + 1),
24 calendar.get (Calendar.YEAR));
25}
26
27 protegido int _getMinimumYear () {
28 return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
29}
30
31 protegido int _getMaximumYear () {
32 return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
33}
34}
```

Página 436

Listado B-16

SpreadsheetDate.java (final)

```
1 / * ============================================ ========================
2 * JCommon: una biblioteca de clases de propósito general gratuita para la plataforma Java (tm)
3 * ============================================ ========================
4 *
5 * (C) Copyright 2000-2005, por Object Refinery Limited y contribuyentes.
6 *
...
52 *
53 * /
54
55 paquete org.jfree.date;
56
57 importar org.jfree.date.Month.FEBRUARY estático;
58
59 import java.util. *;
60
61 / **
62 * Representa una fecha usando un número entero, de manera similar a la
63 * implementación en Microsoft Excel. El rango de fechas admitido es
64 * 1-ene-1900 a 31-dic-9999.
65 * <p />
66 * Tenga en cuenta que hay un error deliberado en Excel que reconoce el año
67 * 1900 como año bisiesto cuando en realidad no es un año bisiesto. Puedes encontrar más
68 * información en el sitio web de Microsoft en el artículo Q181370:
69 * <p />
70 * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
71 * <p />
72 * Excel usa la convención de que 1-Jan-1900 = 1. Esta clase usa el
73 * convención 1-Ene-1900 = 2.
74 * El resultado es que el número de días en esta clase será diferente al
75 * Cifra de Excel para enero y febrero de 1900 ... pero luego Excel agrega un extra
76 * día (29-Feb-1900 que en realidad no existe!) Y desde ese momento en adelante
77 * los números de los días coincidirán.
78 *
79 * @autor David Gilbert
80 * /
81 clase pública SpreadsheetDate extiende DayDate {
82 público estático final int EARLIEST_DATE_ORDINAL = 2; // 1/1/1900
83 public static final int LATEST_DATE_ORDINAL = 2958465; // 31/12/9999
84 público estático final int MINIMUM_YEAR_SUPPORTED = 1900;
85 público estático final int MAXIMUM_YEAR_SUPPORTED = 9999;
86 estático final int [] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
87 {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
88 estático final int [] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
89 {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
90
91 private int ordinalDay;
92 día int privado;
93 mes mes privado;
94 año int privado;
95
96 public SpreadsheetDate (int día, mes mes, int año) {
97 si (año <MINIMUM_YEAR_SUPPORTED || año> MAXIMUM_YEAR_SUPPORTED)
98 lanzar una nueva IllegalArgumentException (
99 "El argumento 'año' debe estar dentro del rango" +
100 MINIMUM_YEAR_SUPPORTED + "a" + MAXIMUM_YEAR_SUPPORTED + ".");
101 if (día <1 || día> DateUtil.lastDayOfMonth (mes, año))
102 lanzar una nueva IllegalArgumentException ("Argumento de 'día' no válido.");
103
104 this.year = año;
105 this.month = mes;
```

Página 437

Listado B-16 (continuación)

SpreadsheetDate.java (final)

```
106 este.día = día;
107 ordinalDay = calcOrdinal (día, mes, año);
108}
109
110 public SpreadsheetDate (int día, int mes, int año) {
111 este (día, Month.fromInt (mes), año);
112}
113
114 public SpreadsheetDate (int serial) {
115 si (serial <EARLIEST_DATE_ORDINAL || serial> LATEST_DATE_ORDINAL)
116 lanzar una nueva IllegalArgumentException (
117                 "SpreadsheetDate: El número de serie debe estar en el rango de 2 a 2958465.");
118
119 ordinalDay = serial;
120 calcDayMonthYear ();
121}
122
123 public int getOrdinalDay () {
124 return ordinalDay;
125}
126
127 public int getYear () {
128 año de retorno;
129}
130
131 public Month getMonth () {
132 mes de vuelta;
133}
134
135 public int getDayOfMonth () {
136 día de regreso;
137}
138
139 día protegido getDayOfWeekForOrdinalZero () {return Day.SATURDAY;}
140
141 público booleano es igual a (objeto objeto) {
142 if (! (Instancia de objeto de DayDate))
143 devuelve falso;
144
145 DayDate date = (DayDate) objeto;
146 return date.getOrdinalDay () == getOrdinalDay ();
147}
148
149 public int hashCode () {
150 return getOrdinalDay ();
151}
152
153 public int compareTo (Objeto otro) {
154 días de devolución desde ((DayDate) otro);
155}
156
157 private int calcOrdinal (int día, mes mes, int año) {
158 int leapDaysForYear = DateUtil.leapYearCount (año - 1);
159 int daysUpToYear = (año - MINIMUM_YEAR_SUPPORTED) * 365 + leapDaysForYear;
160 int daysUpToMonth = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH [month.toInt ()];
161 if (DateUtil.isLeapYear (año) && month.toInt ()> FEBRUARY.toInt ())
162 daysUpToMonth ++;
163 int daysInMonth = día - 1;
164 días de devoluciónUpToYear + daysUpToMonth + daysInMonth + EARLIEST_DATE_ORDINAL;
165}
166
```


Página 438

Listado B-16 (continuación)

SpreadsheetDate.java (final)

```
167 private void calcDayMonthYear () {
168 int días = ordinalDay - EARLIEST_DATE_ORDINAL;
169 int sobreestimadoYear = MINIMUM_YEAR_SUPPORTED + días / 365;
170 int nonleapdays = days - DateUtil.leapYearCount (sobreestimadoYear);
171 int año subestimado = MINIMUM_YEAR_SUPPORTED + no días festivos / 365;
172
173 año = huntForYearContaining (ordinalDay, subestimatedYear);
174 int firstOrdinalOfYear = firstOrdinalOfYear (año);
175 mes = huntForMonthContaining (ordinalDay, firstOrdinalOfYear);
176 día = ordinalDay - firstOrdinalOfYear - daysBeforeThisMonth (month.toInt ());
177}
178
179 mes privado huntForMonthContaining (int anOrdinal, int firstOrdinalOfYear) {
180 int daysIntoThisYear = anOrdinal - firstOrdinalOfYear;
181 int aMonth = 1;
182 while (daysBeforeThisMonth (aMonth) <daysIntoThisYear)
183 aMonth ++;
184
185 return Month.fromInt (aMonth - 1);
186}
187
188 privados int daysBeforeThisMonth (int aMonth) {
189 si (DateUtil.isLeapYear (año))
190 return LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH [aMonth] - 1;
191 más
192 return AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH [aMonth] - 1;
193}
194
195 private int huntForYearContaining (int anOrdinalDay, int startYear) {
196 int aYear = startYear;
197 while (firstOrdinalOfYear (aYear) <= anOrdinalDay)
198 aYear ++;
199
200 return aYear - 1;
201}
202
203 private int firstOrdinalOfYear (int año) {
204 return calcOrdinal (1, Month.JANUARY, año);
205}
206
207 public static DayDate createInstance (Fecha fecha) {
208 Calendario Gregoriano = nuevo Calendario Gregoriano ();
209 calendar.setTime (fecha);
210 return new SpreadsheetDate (calendar.get (Calendar.DATE),
211                             Month.fromInt (calendar.get (Calendar.MONTH) + 1),
212                             calendar.get (Calendar.YEAR));
213
214}
215}
```


Página 439

Esta página se dejó en blanco intencionalmente

Apéndice C

Referencias cruzadas de heurísticas

Referencias cruzadas de olores y heurística. Todas las demás referencias cruzadas se pueden eliminar.

C1 .... 16-276, 16-279, 17-292
C2 .... 16-279, 16-285, 16-295, 17-292

Página 441

Epílogo

En 2005, mientras asistía a la conferencia Agile en Denver, Elisabeth Hedrickson [1] me entregó una muñequera verde similar a la que Lance Armstrong hizo tan popular. Éste dijo "Test Obsessed" en él. Con mucho gusto me lo puse y lo usé con orgullo. Desde que aprendí TDD de Kent Beck en 1999, de hecho me obsesioné con el desarrollo basado en pruebas.

Pero entonces sucedió algo extraño. Descubrí que no podía quitarme la banda. No porque estaba atascado físicamente, pero porque estaba atrapado *moralmente* . La banda hizo un manifiesto declaración sobre mi ética profesional. Fue una indicación visible de mi compromiso con escribiendo el mejor código que pude escribir. Quitarlo parecía una traición a esa ética y de ese compromiso.

Así que todavía está en mi muñeca. Cuando escribo código, lo veo en mi visión periférica. Es un recordatorio constante de la promesa que me hice a mí mismo de escribir código limpio.

1. http://www.qualitytree.com/

Página 443

Esta página se dejó en blanco intencionalmente

Índice

Página 446

C

Página 447

Página 448

D

Página 449

Página 450

mi

F

Página 451

GRAMO

Página 452

H

Página 453

J

K

L

Página 454

Página 455

norte

O

Página 456

PAG

Página 457

Q

R

Página 458

Página 459

Página 460

T

Página 461

U

V

Página 462

W

X