Prólogo de James O. Coplien

Robert C. Martin
Coautor del ***Manifiesto Ágil***

Código Limpio

Manual de estilo para el desarrollo ágil de software

Cada año, se invierten innumerables horas y se pierden numerosos recursos debido a código mal escrito, ralentizando el desarrollo, disminuyendo la productividad, generando graves fallos e incluso pudiendo acabar con la organización o empresa.

El reconocido experto de software Robert C. Martin, junto con sus colegas de Object Mentor, nos presentan sus óptimas técnicas y metodologías ágiles para limpiar el código sobre la marcha y crearlo de forma correcta, de este modo mejorará como programador.

Esta obra se divide en tres partes. La primera describe los principios, patrones y prácticas para crear código limpio. La segunda incluye varios casos de estudio cuya complejidad va aumentando. Cada ejemplo es un ejercicio de limpieza y transformación de código con problemas. La tercera parte del libro contiene una lista de heurística y síntomas de código erróneo (smells) confeccionada al crear los casos prácticos. El resultado es una base de conocimientos que describe cómo pensamos cuando creamos, leemos y limpiamos código.

Imprescindible para cualquier desarrollador, ingeniero de software, director de proyectos, jefe de equipo o analista de sistemas interesado en crear código de mejor calidad.

¡El libro que todo programador debe leer!

Robert Cecil Martin

Código limpio

Manual de estilo para el desarrollo ágil de software

ePub r1.1
XcUiDi 21.03.2018

Título original: *Clean code: A handbook of agile software craftsmanship*
Robert Cecil Martin, 2009
Traducción: José Luis Gómez Celador
Ilustraciones: Jeniffer Kohnke & Angela Brooks

Editor digital: XcUiDi

Colaborador: Mario J. C. (PDF-Español)
ePub base r1.2

Para Ann Marie: El verdadero amor de mi vida.

Agradecimientos

Me gustaría dar las gracias a mis dos artistas, Jeniffer Kohnke y Angela Brooks. Jennifer es la encargada de las impresionantes ilustraciones del inicio de cada capítulo y también de los retratos de Kent Beck, Ward Cunningham, Bjarne Stroustrup, Ron Jeffries, Grady Booch, Dave Thomas, Michael Feathers y el mío propio.

Angela se encarga de las ilustraciones internas de los capítulos. Ha realizado muchos dibujos para mí en los últimos años, incluidos muchos de los del libro *Agile Software Development: Principles, Patterns, and Practices*. También es mi primogénita.

Un agradecimiento especial a los revisores Bob Bogetti, George Bullock, Jeffrey Overbey y especialmente Matt Heusser. Han sido increíbles. Han sido inmisericordes. Han sido minuciosos. Me han forzado al máximo para realizar las mejoras necesarias.

Gracias a mi editor, Chris Guzikowski, por su apoyo, aliento y amistad. Gracias a todo el personal de la editorial, incluida Raina Chrobak, que se encargó de que fuera honesto y cumpliera los plazos.

Gracias a Micah Martin y a todos los de 8th Light (www.8thlight.com) por sus críticas y su apoyo.

Gracias a todos los Object Mentor, pasados, presentes y futuros, incluidos Bob Koss, Michael Feathers, Michael Hill, Erik Meade, Jeff Langr, Pascal Roy, David Farber, Brett Schuchert, Dean Wampler, Tim Ottinger, Dave Thomas, James Grenning, Brian Button, Ron Jeffries, Lowell Lindstrom, Angelique Martin, Cindy Sprague, Libby Ottinger, Joleen Craig, Janice Brown, Susan Rosso y el resto.

Gracias Jim Newkirk, mi amigo y socio, que me ha enseñado más de lo que cree. Mi agradecimiento a Kent Beck, Martin Fowler, Ward Cunningham, Bjarne Stroustrup, Grady Booch y todos mis mentores, compatriotas y colegas. Gracias a John Vlissides por estar ahí cuando lo necesitaba. Gracias a todos los de Zebra por permitirme despotricar sobre la extensión que debe tener una función.

Y, por último, darle las gracias por leer estos agradecimientos.

Prólogo

Una de nuestras golosinas preferidas en Dinamarca es *Ga-Jol*, con un fuerte sabor a regaliz, que constituye un complemento perfecto para nuestro húmedo y frío clima. Parte del encanto de *Ga-Jol* para los daneses es la frase que suele incluir en el envoltorio. Esta mañana compré un paquete de dos y me encontré con este antiguo dicho danés:

Ærlighed i små ting er ikke nogen lille ting.

«La honestidad por las cosas pequeñas no es algo menor». Perfecto para que lo que pensaba escribir. Las cosas pequeñas importan. Este libro trata sobre humildes preocupaciones cuyo valor dista mucho de ser menor.

Dios está en los detalles, afirmó el arquitecto Ludwig mies van der Rohe. Esta cita recuerda argumentos contemporáneos sobre el papel de la arquitectura en el desarrollo de *software*, en especial en el universo ágil. Bob y yo hemos tenido esta conversación muchas veces. Y sí, mies van der

Rohe se fijaba en la utilidad y la forma atemporal de la construcción que subyace a las grandes creaciones arquitectónicas. Por otra parte, seleccionaba personalmente los pomos de todas las puertas de todas las casas que diseñaba. ¿Por qué? Porque las cosas pequeñas importan.

En nuestro interminable debate sobre TDD, Bob y yo coincidimos en que la arquitectura del *software* desempeña una importante labor en el desarrollo, aunque tenemos diferentes visiones de lo que esto significa. Estas diferencias carecen de importancia, ya que podemos aceptar que los profesionales responsables dedican parte de su tiempo a planificar un proyecto antes de comenzarlo. Las nociones de diseño controlado únicamente por pruebas y el código, propias de finales de la década de 1990, ya no son válidas. Y la atención al detalle es un pilar fundamental de los profesionales, casi como cualquier visión. Por un lado, la práctica en los detalles otorga dominio a los profesionales, y aumenta su confianza para la práctica a mayor escala. Por otra parte, el más mínimo fallo de construcción, una puerta que no cierre bien o un baldosín mal colocado, acaba con el encanto del todo. De eso se trata el código limpio.

Pero la arquitectura es sólo una metáfora del desarrollo de *software* y en concreto de la parte del *software* que ofrece el producto inicial, de la misma forma que un arquitecto entrega un edificio inmaculado. Hoy en día, el objetivo es comercializar rápidamente los productos. Queremos que las fábricas produzcan *software* a pleno rendimiento. Se trata de fábricas humanas, programadores que piensan, que sienten y que trabajan para crear un producto. La metáfora de la manufacturación es incluso más evidente en este pensamiento. Los aspectos productivos de las fábricas de automóviles japonesas fueron una gran inspiración para Serum.

Pero incluso en la industria automovilística, gran parte del trabajo no radica en la fabricación sino en el mantenimiento, o más bien en cómo evitarlo. En el *software*, el 80 por 100 o más de lo que hacemos se denomina cuantitativamente mantenimiento, el acto de reparar. En lugar de optar por la típica costumbre occidental de crear *software* de calidad, debemos pensar como reparadores o mecánicos. ¿Qué piensan los directores japoneses de todo esto?

En 1951, un enfoque de calidad denominado TPM (*Total Productive Maintenance* o Mantenimiento productivo total) apareció en escena. Se centraba en el mantenimiento y no en la producción. Uno de los pilares de TPM es el conjunto de principios denominados 5S, una serie de disciplinas.

Estos principios 5S son en realidad la base Lean, otro conocido término en la escena occidental, y cada vez más presente en el mundo del *software*. Estos principios no son opcionales. Como indica Uncle Bob, la práctica del *software* correcto requiere disciplina. No siempre se trata de hacer, de producir a la velocidad óptima.

La filosofía 5S incluye estos conceptos:

- ***Seiri* u organización**: Es fundamental saber dónde están las cosas, mediante enfoques como el uso de nombres correctos. ¿Cree que los nombres de los identificadores no son relevantes? Lea los siguientes capítulos.
- ***Seiton* o sistematización**: Existe un antiguo dicho norteamericano: un sitio para cada cosa y cada cosa en su sitio. Un fragmento de código debe estar donde esperamos encontrarlo; en caso contrario, refactorice hasta conseguirlo.
- ***Seiso* o limpieza**: Mantenga limpio el lugar de trabajo. ¿Qué dicen los autores sobre inundar el código de comentarios y líneas que capturan historias o deseos futuros? Elimínelos.
- ***Seiketsu* o estandarización**: El grupo decide cómo mantener limpio el lugar de trabajo. ¿Cree que este libro habla sobre tener un estilo de código coherente y una serie de prácticas dentro del grupo? ¿De dónde provienen esos estándares? Siga leyendo.
- ***Shutsuke* o disciplina**: Significa ser disciplinado en la aplicación de las prácticas y reflejarlas en el trabajo y aceptar los cambios.

Si acepta el reto, ha leído bien, el reto, de leer y llevar a la práctica este libro, podrá comprender y apreciar el último punto. Aquí nos acercamos a la raíz de la profesionalidad responsable de una profesión que debería preocuparse del ciclo vital de un producto. Al igual que mantenemos coches y otras máquinas, el mantenimiento divisible, esperar a que surjan los errores, es la excepción. Por el contrario, ascendemos un nivel: inspeccionamos diariamente las máquinas y arreglamos los componentes gastados antes de que se rompan, o cambiamos el aceite cada varios miles de kilómetros para evitar problemas. En el código, debemos refactorizar sin compasión. Puede ascender otro nivel más, como hizo el movimiento TPM hace 50 años: crear máquinas que se pueden mantener mejor. Crear código

legible es tan importante como crear código ejecutable. La práctica definitiva, que apareció en los círculos TPM en 1960, es la que se centra en introducir nuevas máquinas o sustituir las antiguas. Como Fred Brooks nos advirtió, deberíamos rehacer el *software* cada siete años para eliminar los problemas latentes. Tendríamos que actualizar este plazo por semanas, días e incluso horas en lugar de años. Ahí es donde se encuentra el detalle.

El detalle tiene un gran poder, y es un enfoque vital humilde y profundo, como es de esperar de cualquier enfoque de origen japonés. Pero no es sólo la visión oriental de la vida; también lo encontramos en el pueblo norteamericano. La cita *seiton* anterior proviene de la pluma de un ministro de Ohio que, literalmente, consideraba la limpieza como un remedio para todas las formas del mal. ¿Y *seiso*? *La limpieza es la pureza*. Aunque una casa sea bella, el mobiliario inadecuado acaba con su encanto. ¿Y la opinión de *shutsuke* al respecto? *El que confíe en lo pequeño confiará en lo superior*. ¿Y la predisposición a refactorizar en el momento adecuado, reforzando nuestra posición para las posteriores grandes decisiones, en lugar de dejarlo pasar? *Una puntada a tiempo ahorra ciento. Al que madruga, Dios le ayuda. No dejes para mañana lo que puedas hacer hoy* (éste era el sentido original de la frase «en el momento adecuado» de Lean hasta que cayó en manos de consultores de *software*). ¿Y sobre calibrar la importancia de los pequeños esfuerzos individuales en un todo mayor? *De pequeñas semillas crecen grandes árboles*. ¿Y la integración de sencillas tareas preventivas en la vida diaria? *Más vale prevenir que curar*. El código limpio honra las raíces de la sabiduría popular, de antes o de ahora, y se puede aplicar con atención al detalle.

Incluso en la literatura arquitectónica encontramos ejemplos de estos detalles. Piense en los pomos de mies van der Rohe. Eso es *seiri*. Es la atención a todos los nombres de variables. Debe bautizar a una variable con el mismo cuidado como si fuera su primogénito.

Y como todo sabemos, este cuidado no acaba nunca. El arquitecto Christopher Alexander, padre de patrones y lenguajes de patrones, considera todo acto de diseño como un pequeño acto local de reparación, y considera la maestría de la estructura como competencia única del arquitecto; las formas mayores se ceden a los patrones y su aplicación a los habitantes. El diseño es interminable no sólo al añadir una nueva habitación a una casa, sino al prestar atención a la pintura, a cambiar las alfombras o a instalar un nuevo fregadero en la cocina. Otras artes muestran sentimientos

análogos. En nuestra búsqueda por la importancia de los detalles, nos topamos con el autor francés del siglo XIX Gustav Flaubert. El poeta francés Paul Valery afirma que un poema no se acaba nunca y que tiene que retocarse continuamente, y que dejar de trabajar en el poema es señal de abandono. Tal preocupación por el detalle es común en todas las empresas de excelencia. Puede que esto no sea nada nuevo, pero al leer este libro sentirá la necesidad de adoptar disciplinas rechazadas en su momento por apatía o por un deseo de espontaneidad o una simple respuesta al cambio.

Desafortunadamente, no solemos considerar estas preocupaciones la clave del arte de la programación. Renunciamos pronto a nuestro código, no porque lo hayamos completado, sino porque nuestro sistema de valores se centra en el futuro más que en la sustancia de nuestros productos.

Esto tiene un precio final: *hierba mala nunca muere*. La investigación, ni en el mundo industrial ni en el académico, se reduce a mantener limpio el código. Cuando trabajaba en la organización Bell Labs Software Production Research (sin duda de producción) comprobamos que un estilo de sangrado coherente era uno de los mayores indicadores estadísticamente significativos de una baja densidad de errores. Queremos que una arquitectura, un lenguaje de programación u otra noción superior sea el motivo de la calidad; como seres cuya supuesta profesionalidad se debe al dominio de herramientas y métodos de diseño, nos sentimos insultados por el valor que los programadores añaden con tan sólo aplicar un estilo de sangrado coherente. Para citar mi propio libro de hace 17 años, dicho estilo distingue la excelencia de la simple competencia. La visión japonesa comprende el verdadero valor del trabajador cotidiano y, en especial, de los sistemas de desarrollo que dependen de las sencillas acciones diarias de tales trabajadores. La calidad es el resultado de un millón de acciones cuidadosas, no de un método magnífico caído del cielo. Que dichas acciones sean simples no significa que sean simplistas, y mucho menos que sean sencillas. Son la base de la grandeza y, cada vez más, de cualquier empresa humana. Ignorarlas no es humano en absoluto.

Evidentemente, todavía defiendo el pensamiento global, en especial el valor de los enfoques arquitectónicos cimentados en el conocimiento de los dominios y la capacidad de uso del *software*. Este libro no versa sobre esto, al menos no de forma evidente. Este libro transmite un mensaje más sutil cuya profundidad no debe menospreciarse. Coincide con la visión de gente como Peter Sommerlad, Kevlin Henney y Giovanni Asproni, cuyos mantras

son «El código es el diseño» y «Código simple». Aunque debemos recordar que la interfaz es el programa y que sus estructuras dicen mucho sobre la propia estructura del programa, es fundamental adoptar de forma continuada la humilde posición de que el diseño vive en el código. Y aunque los cambios y la metáfora de la fábrica supongan costes, los cambios de diseño suponen valor. Debemos considerar al código como la articulación de los esfuerzos de diseño, visto como un proceso, no como algo estático. Es en el código donde se desarrollan los conceptos arquitectónicos de conexión y cohesión. Si escucha a Larry Constantine describir la conexión y la cohesión, lo hace en términos del código, no desde conceptos abstractos propios de UML. En su ensayo *Abstraction Descant*, Richard Gabriel afirma que la abstracción es el mal. El código es el remedio al mal y el código limpio puede que sea divino.

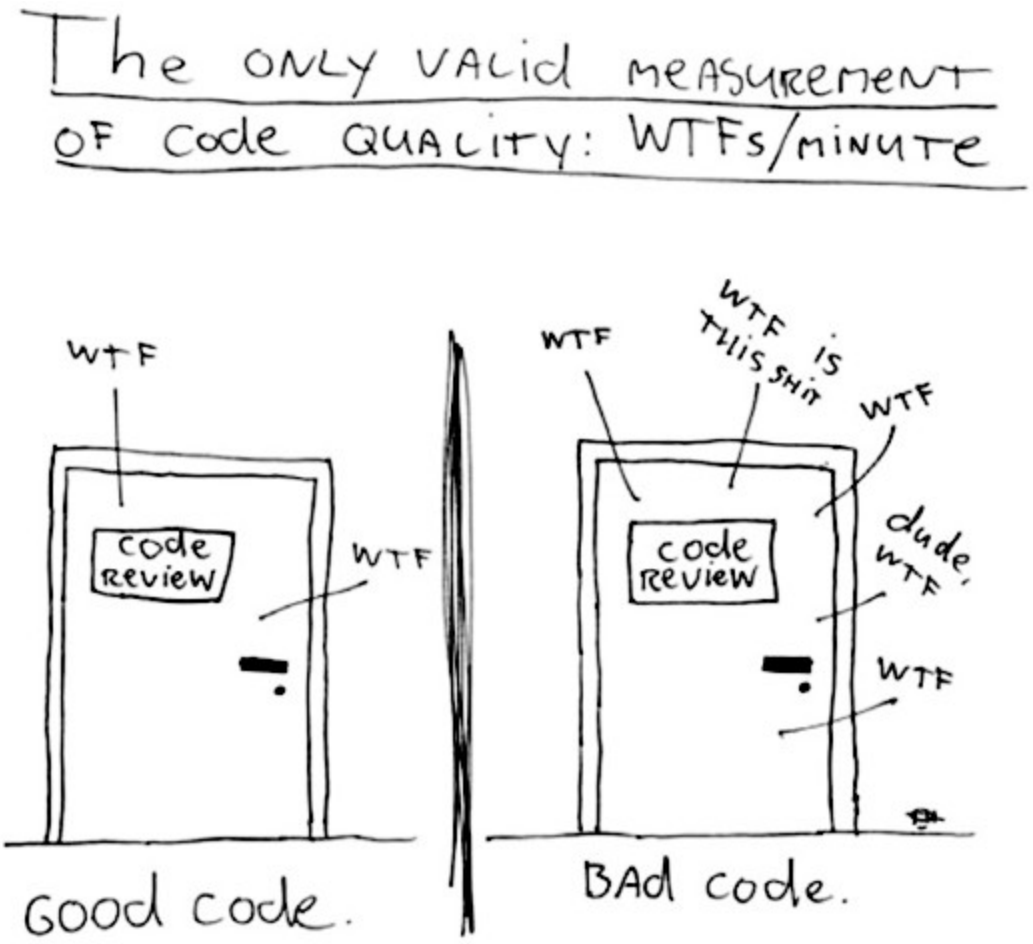
Volviendo a mi caja de *Ga-Jol*, considero importante recordar que la sabiduría danesa nos recomienda no sólo prestar atención a las pequeñas cosas, sino también ser *honestos* con ellas. Esto significa ser honesto con el código, con nuestros colegas sobre el estado del código y, en especial, con nosotros mismos. ¿Hemos hecho todo lo posible para dejar las cosas mejor que como las encontramos? ¿Hemos refactorizado el código antes de terminarlo? No se trata de preocupaciones periféricas, sino que se encuentran en la base misma de los valores Agile. En Serum se recomienda que la refactorización sea parte del concepto de Terminado. Ni la arquitectura ni el código limpio insisten en la perfección, sino en la honestidad y en hacerlo lo mejor posible. *Errar es humano; perdonar es divino*. En Serum, todo lo hacemos de forma visible. Aireamos los trapos sucios. Somos honestos sobre el estado de nuestro código ya que nunca es perfecto. Nos hemos hecho más humanos, más merecedores de lo divino y estamos más próximos a la grandeza de los detalles.

En nuestra profesión, necesitamos desesperadamente toda la ayuda posible. Si un suelo seco reduce el riesgo de resbalones y las herramientas bien organizadas aumentan la productividad, es nuestra meta. Y en cuanto al libro, es la mejor aplicación pragmática de los principios Lean de *software* que he visto nunca en formato impreso. No esperaba menos de este grupo de individuos que durante años se han esforzado no sólo por mejorar sino en ofrecer sus conocimientos a la industria mediante obras como la que ahora tiene entre manos. Hace que el mundo sea un poco mejor que antes de que Uncle Bob me enviara el manuscrito.

Y tras completar este ejercicio, me dispongo a limpiar mi escritorio.

James O. Coplien
Mørdrup, Dinamarca

Introducción



¿Qué puerta representa su código? ¿Qué puerta representa a su equipo o a su empresa? ¿Por qué estamos en esa habitación? ¿Es una sencilla revisión del código o hemos detectado un sinfín de problemas terribles? ¿Depuramos presas del pánico el código que pensábamos que funcionaba? ¿Los clientes huyen despavoridos y los directores nos pisan los talones?

¿Cómo asegurarnos de que abrimos la puerta correcta cuando las cosas se ponen feas? La respuesta: *la maestría*.

La maestría se consigue de dos formas: conocimientos y trabajo. Debe adquirir el conocimiento de los principios, patrones, prácticas y heurística propios de un maestro, y dominar dichos conocimientos a través de la práctica.

Puedo enseñarle la teoría de montar en bicicleta. De hecho, los conceptos matemáticos clásicos son muy sencillos. Gravedad, fricción, velocidad angular, centro de masa, etc., se pueden demostrar en menos de una página repleta de ecuaciones. Con esas fórmulas, puedo demostrar que montar en bicicleta es práctico y proporcionarle los conocimientos necesarios para conseguirlo. Pero la primera vez que se monte en una bici se caerá al suelo.

El diseño de código no es diferente. Podríamos enumerar todos los principios del código limpio y confiar en que se *encargue* del resto (es decir, dejar que se cayera de la bici) pero entonces la pregunta sería qué clase de profesores somos y qué clase de alumno sería.

No. Así no funciona este libro.

Aprender a crear código limpio es *complicado*. Requiere algo más que conocer principios y patrones. Tiene que sudar. Debe practicarlo y fallar. Debe ver cómo otros practican y fallan. Debe observar cómo se caen y recuperan el paso. Debe ver cómo agonizan en cada decisión y el precio que pagan por tomar decisiones equivocadas.

Para leer este libro, prepárese a trabajar duro. No es un libro que se pueda leer en un avión y terminarlo antes de aterrizar. Este libro le hará trabajar, y mucho. ¿Y qué tipo de trabajo? Tendrá que leer código, en abundancia. Y se le pedirá que piense en qué acierta el código y en qué falla. Se le pedirá que siga nuestras descripciones mientras despedazamos módulos y los volvemos a ensamblar. Para ello necesitará tiempo y esfuerzo, pero creemos que merece la pena.

Hemos dividido el libro en tres partes. Los primeros capítulos describen los principios, patrones y prácticas para crear código limpio. Incluyen abundante código y resultan difíciles de leer. Sirven como preparación a la segunda parte. Si abandona tras leer la primera sección, que tenga buena suerte.

La segunda parte del libro es la más difícil. Incluye varios casos de estudio cuya complejidad va aumentando. Cada ejemplo es un ejercicio de

limpieza de código, transformar código con problemas para que tenga menos problemas. El detalle de esta parte es *abundante*. Tendrá que alternar entre el texto y los listados de código. Tendrá que analizar y entender el código, y comprender el razonamiento de cada cambio realizado. Piense en que esta parte *le llevará varios días*.

La tercera parte del libro es un único capítulo que contiene una lista de heurística y síntomas de código erróneo (*smells*) confeccionada al crear los casos prácticos. Al analizar y limpiar el código de los ejemplos, documentamos el motivo de todas nuestras acciones como heurística o síntoma. Intentamos comprender nuestras reacciones al código que leíamos y modificábamos, y nos esforzamos por capturar las sensaciones que tuvimos y las decisiones que adoptamos. El resultado es una base de conocimientos que describe cómo pensamos cuando creamos, leemos y limpiamos código.

Esta base de conocimientos no le servirá de mucho si no lee atentamente los casos de la segunda parte del libro. En esos capítulos hemos anotado con precisión todos los cambios realizados con referencias a la heurística. Estas referencias se muestran entre corchetes, como [H22]. De este modo puede ver el contexto en el que se ha aplicado y creado dicha heurística. No importa tanto el propio valor de las heurísticas sino *la relación entre ellas y las decisiones adoptadas al limpiar el código en los ejemplos*.

Si lee la primera y la tercera parte y se salta los casos prácticos, habrá leído otro libro distinto sobre cómo crear código correcto, pero si dedica tiempo a analizar los casos, sigue todos y cada uno de los pasos, cada una de las decisiones, si se pone en nuestro lugar y se obliga a pensar tal y como lo hicimos nosotros, entonces comprenderá mucho mejor todos los principios, patrones, prácticas y heurística. Ya no será un conocimiento superficial. Se convertirá en algo profundo. Lo integrará de la misma forma que una bicicleta se convierte en una extensión propia una vez dominada la forma de montar.

Sobre la imagen de cubierta

La imagen de la portada es M104: la Galaxia del Sombrero. M104 se encuentra en Virgo, a unos 30 millones de años luz, y su núcleo es un súper agujero negro que pesa aproximadamente mil millones de masas solares.

¿La imagen le recuerda la explosión de la luna Praxis de Klingon? Recuerdo la escena de Star Trek VI en la que se mostraba un anillo de restos flotando tras la explosión. Tras esa escena, el anillo se ha convertido en un elemento habitual de las explosiones de ciencia ficción. Incluso se añadió a la explosión de Alderaan en ediciones posteriores de la primera película de La Guerra de las Galaxias.

¿Qué provocó la formación de este anillo alrededor de M104? ¿Por qué tiene un centro de tales dimensiones y un núcleo tan brillante y diminuto? Parece como si el agujero negro central hubiera provocado un orificio de 30 000 años luz en el centro de la galaxia. La desgracia caería sobre las civilizaciones que se encontraran en el camino de este desastre cósmico.

Los súper agujeros negros desayunan estrellas y convierten parte de su masa en energía. $E=MC^2$ puede bastar, pero cuando M es una masa estelar hay que tener cuidado. ¿Cuántas estrellas habrá engullido este monstruo antes de saciar su apetito? El tamaño del vacío central podría ser una pista.

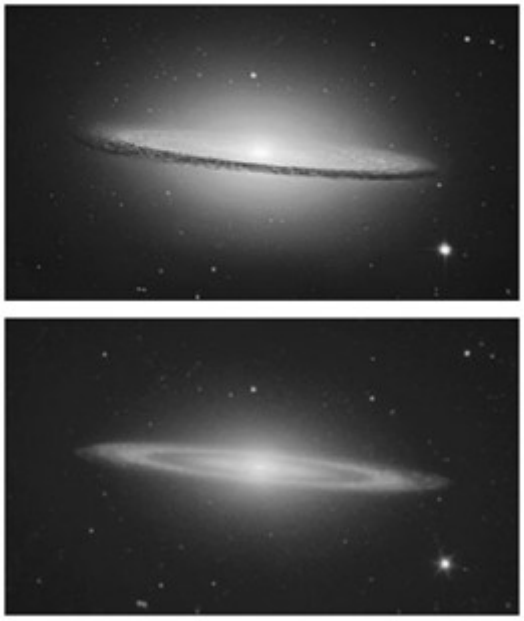


Imagen de portada: © Spitzet Space Telescope.

La imagen de M104 de la portada es una combinación de la famosa fotografía del Hubble (imagen superior) y la reciente imagen de infrarrojos del observatorio orbital Spitzer (inferior).

Esta última muestra claramente la forma de anillo de la galaxia.

A la luz, sólo vemos el borde frontal de la silueta del anillo.

La masa central oculta el resto.

Pero en la imagen de infrarrojos, las partículas calientes del anillo brillan a través de la masa central. Las dos imágenes combinadas nos ofrecen una vista desconocida hasta ahora e implican que hace tiempo era un auténtico infierno activo.

1
Código Limpio

Está leyendo este libro por dos motivos. Por un lado, es programador. Por otro, quiere ser mejor programador. Perfecto. Necesitamos mejores programadores.

Este libro trata sobre programación correcta. Está repleto de código. Lo analizaremos desde todas las direcciones. Desde arriba, desde abajo y desde dentro. Cuando terminemos, sabremos mucho sobre código y, en

especial sabremos distinguir entre código correcto e incorrecto. Sabremos cómo escribir código correcto y cómo transformar código incorrecto en código correcto.

Hágase el código

Se podría afirmar que un libro sobre código es algo obsoleto, que el código ya no es el problema y que deberíamos centrarnos en modelos y requisitos. Hay quienes sugieren que el final del código está próximo. Que los programadores ya no serán necesarios porque los empresarios generarán programas a partir de especificaciones.

No es cierto. El código nunca desaparecerá, ya que representa los detalles de los requisitos. En algún nivel, dichos detalles no se pueden ignorar ni abstraer; deben especificarse, y para especificar requisitos de forma que un equipo pueda ejecutarlos se necesita la programación. Dicha especificación es el código.

Espero que el nivel de abstracción de nuestros lenguajes siga aumentando. También espero que aumente el número de lenguajes específicos de dominios. Será algo positivo, pero no eliminará el código. De hecho, todas las especificaciones creadas en estos lenguajes de nivel superior y específicos de los dominios serán código, y tendrá que ser riguroso, preciso, formal y detallado para que un equipo pueda entenderlo y ejecutarlo.

El que piense que el código va a desaparecer es como el matemático que espera que un día las matemáticas no sean formales. Esperan descubrir una forma de crear máquinas que hagan lo que queramos en lugar de lo que digamos. Esas máquinas tendrían que entendernos de tal forma que puedan traducir necesidades ambiguas en programas perfectamente ejecutados que satisfagan dichas necesidades a la perfección.

Esto nunca sucederá. Ni siquiera los humanos, con toda su intuición y creatividad, han sido capaces de crear sistemas satisfactorios a partir de las sensaciones de sus clientes. En realidad, si la disciplina de la especificación de requisitos nos ha enseñado algo es que los requisitos bien especificados son tan formales como el código y que pueden actuar como pruebas ejecutables de dicho código.

Recuerde que el código es básicamente el lenguaje en el que expresamos los requisitos en última instancia. Podemos crear lenguajes que se asemejen a dichos requisitos. Podemos crear herramientas que nos permitan analizar y combinar dichos requisitos en estructuras formales, pero nunca eliminaremos la precisión necesaria; por ello, siempre habrá código.

Código Incorrecto

Recientemente leí el prólogo del libro *Implementation Pattern*[1] de Kent Beck, donde afirmaba que «…este libro se basa en una frágil premisa: que el código correcto es relevante…». ¿Una *frágil* premisa? En absoluto. Considero que es una de las más robustas, admitidas e importantes de nuestro sector (y creo que Kent lo sabe). Sabemos que el código correcto es relevante porque durante mucho tiempo hemos tenido que sufrir su ausencia.

Sé de una empresa que, a finales de la década de 1980, creó una *magnífica* aplicación, muy popular y que muchos profesionales compraron y utilizaron. Pero los ciclos de publicación empezaron a distanciarse. No se

corrigieron los errores entre una versión y la siguiente. Crecieron los tiempos de carga y aumentaron los fallos. Todavía recuerdo el día en que apagué el producto y nunca más lo volví a usar.

Poco después, la empresa desapareció.

Dos décadas después conocí a uno de los empleados de la empresa y le pregunté sobre lo que había pasado. La respuesta confirmó mis temores. Habían comercializado el producto antes de tiempo con graves fallos en el código. Al añadir nuevas funciones, el código empeoró hasta que ya no pudieron controlarlo. *El código incorrecto fue el motivo del fin de la empresa*.

¿En alguna ocasión el código incorrecto le ha supuesto un obstáculo? Si es programador seguramente sí. De hecho, tenemos una palabra que lo describe: *sortear*. Tenemos que sortear el código incorrecto. Nos arrastramos por una maraña de zarzas y trampas ocultas. Intentamos buscar el camino, una pista de lo que está pasando, pero lo único que vemos es más y más código sin sentido.

Sin duda el código incorrecto le ha supuesto un obstáculo. Entonces, ¿por qué lo escribió?

¿Tenía prisa? ¿Plazos de entrega? Seguramente. Puede que pensara que no tenía tiempo para hacer un buen trabajo; que su jefe se enfadaría si necesitaba tiempo para limpiar su código. O puede que estuviera cansado de trabajar en ese programa y quisiera acabar cuanto antes. O que viera el trabajo pendiente y tuviera que acabar con un módulo para pasar al siguiente. A todos nos ha pasado.

Todos hemos visto el lío en el que estábamos y hemos optado por dejarlo para otro día. Todos hemos sentido el alivio de ver cómo un programa incorrecto funcionaba y hemos decidido que un mal programa que funciona es mejor que nada. Todos hemos dicho que lo solucionaríamos después. Evidentemente, por aquel entonces, no conocíamos la ley de LeBlanc: *Después es igual a nunca*.

El coste total de un desastre

Si es programador desde hace dos o tres años, probablemente haya sufrido los desastres cometidos por otros en el código. Si tiene más experiencia, lo habrá sufrido en mayor medida. El grado de sufrimiento puede ser

significativo. En un periodo de un año o dos, los equipos que avancen rápidamente al inicio de un proyecto pueden acabar a paso de tortuga. Cada cambio en el código afecta a dos o tres partes del mismo. Ningún cambio es trivial. Para ampliar o modificar el sistema es necesario comprender todos los detalles, efectos y consecuencias, para de ese modo poder añadir nuevos detalles, efectos y consecuencias. Con el tiempo, el desastre aumenta de tal modo que no se puede remediar. Es imposible.

Al aumentar este desastre, la productividad del equipo disminuye y acaba por desaparecer. Al reducirse la productividad, el director hace lo único que puede: ampliar la plantilla del proyecto con la esperanza de aumentar la productividad. Pero esa nueva plantilla no conoce el diseño del sistema. No conocen la diferencia entre un cambio adecuado al objetivo de diseño y otro que lo destroce. Por tanto, todos se encuentran sometidos a una gran presión para aumentar la productividad. Por ello, cometen más errores, aumenta el desastre y la productividad se acerca a cero cada vez más (véase la figura 1.1).

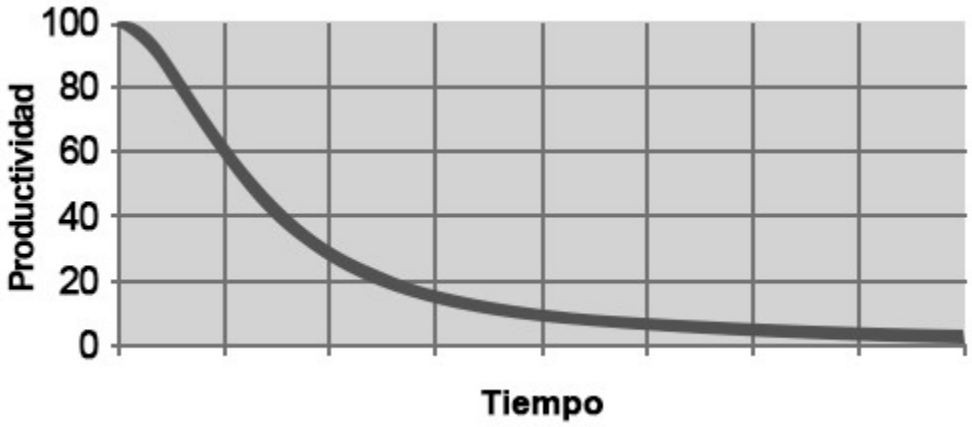


Figura 1.1. Productividad frente a tiempo.

El gran cambio de diseño

En última instancia, el equipo se rebela. Informan al director que no pueden seguir trabajando con ese código. Exigen un cambio de diseño. La dirección no requiere invertir en un cambio de diseño del proyecto, pero no puede ignorar el bajo nivel de productividad.

Acaba por ceder a las exigencias de los programadores y autoriza el gran cambio de diseño. Se selecciona un nuevo equipo. Todos quieren formar parte del nuevo equipo por ser un lienzo en blanco. Pueden empezar de cero y crear algo realmente bello, pero sólo los mejores serán elegidos para el nuevo equipo. Los demás deben continuar con el mantenimiento del sistema actual.

Ahora los dos equipos compiten. El nuevo debe crear un sistema que haga lo que el antiguo no puede. Además, deben asumir los cambios que continuamente se aplican al sistema antiguo. La dirección no sustituirá el sistema antiguo hasta que el nuevo sea capaz de hacer todo lo que hace el antiguo.

Esta competición puede durar mucho tiempo. Conozco casos de casi 10 años. Y cuando acaba, los miembros originales del equipo nuevo han desaparecido y los miembros actuales exigen un cambio de diseño del nuevo sistema porque es un desastre.

Si ha experimentado alguna fase de esta historia, ya sabrá que dedicar tiempo a que el código sea correcto no sólo es rentable, es una cuestión de supervivencia profesional.

Actitud

¿Alguna vez ha tenido que superar un desastre tan grave que ha tardado semanas en lo que normalmente hubiera tardado horas? ¿Ha visto un cambio que debería haberse realizado en una línea, aplicado en cientos de módulos distintos? Son síntomas demasiado habituales.

¿Por qué sucede en el código? ¿Por qué el código de calidad se transforma tan rápidamente en código incorrecto? Hay muchas explicaciones. Nos quejamos de que los requisitos cambian de forma que comprometen el diseño original, de que los plazos de entrega son demasiado exigentes para hacer las cosas bien. Culpamos a directores incompetentes, a usuarios intolerantes y a comerciales sin sentido. Pero la culpa, querido Dilbert, es nuestra. No somos profesionales.

Puede que resulte duro escucharlo. ¿Cómo es posible que *seamos responsables* de tales desastres? ¿Qué pasa con los requisitos? ¿Y los plazos de entrega? ¿Y los directores incompetentes y los comerciales sin sentido? ¿No es también culpa suya?

No. Los directores y los comerciales *nos* exigen la información que necesitan para realizar sus promesas y compromisos, e incluso cuando no recurren a nosotros, no debemos tener miedo a decirles lo que pensamos. Los usuarios acuden a nosotros para validar la forma de encajar los requisitos en el sistema. Los directores de proyectos acuden a nosotros para determinar los objetivos. Somos cómplices en la programación del proyecto y compartimos gran parte de la responsabilidad de los fallos, en especial si tienen que ver con código incorrecto.

Seguramente piense que, si no hace lo que su jefe le dice, le despedirán. Es improbable. Muchos jefes sólo quieren la verdad, aunque lo disimulen. Muchos quieren código correcto, aunque estén obsesionados con los objetivos. Pueden defender apasionadamente los objetivos y los requisitos, pero es su trabajo. El *nuestro* es defender el código con la misma intensidad.

Para resumir, imagine que es médico y un paciente le exige que no se lave las manos antes de una operación porque se pierde demasiado tiempo[2]. En este caso, el paciente es el jefe, pero el médico debe negarse a lo que pide. ¿Por qué? Porque el médico sabe más que el paciente sobre los riesgos de infecciones. No sería profesional (incluso sería ilegal) que el médico cediera a las exigencias del paciente.

Tampoco sería profesional que los programadores cedieran a la voluntad de los jefes que no entienden los riesgos de un posible desastre.

El enigma

Los programadores se enfrentan a un enigma de valores básicos. Los que tienen años de experiencia saben que un desastre ralentiza su trabajo, y aun así todos los programadores sienten la presión de cometer errores para poder cumplir los plazos de entrega. En definitiva, no toman el tiempo necesario para avanzar.

Los verdaderos profesionales saben que la segunda parte del enigma no es cierta. No se cumple un plazo de entrega cometiendo un error. De hecho, el error nos ralentiza de forma inmediata y hace que no lleguemos al plazo de entrega. La *única* forma de cumplirlo, la única forma de avanzar, es intentar que el código siempre sea limpio.

¿El arte del código limpio?

Imagine que cree que el código incorrecto es un obstáculo significativo. Imagine que acepta que la única forma de avanzar es mantener el código limpio. Entonces se preguntará cómo crear código limpio. No tiene sentido intentar crearlo si no sabe lo que es.

La mala noticia es que crear código limpio es como pintar un cuadro. Muchos sabemos si un cuadro se ha pintado bien o no, pero poder reconocer la calidad de una obra no significa que sepamos pintar. Por ello, reconocer código limpio no significa que sepamos cómo crearlo.

Para crearlo se requiere el uso disciplinado de miles de técnicas aplicadas mediante un detallado sentido de la «corrección». Este sentido del código es la clave.

Algunos nacen con este sentido. Otros han de luchar para conseguirlo. No sólo permite distinguir entre código correcto e incorrecto, sino que también muestra la estrategia para aplicar nuestra disciplina y transformar código incorrecto en código correcto.

Un programador sin este sentido puede reconocer el desastre cometido en un módulo, pero no saber cómo solucionarlo. Un programador con este sentido verá las posibles opciones y elegirá la variante óptima para definir una secuencia de cambios.

En definitiva, un programador que cree código limpio es un artista que puede transformar un lienzo en blanco en un sistema de código elegante.

Concepto de código limpio

Existen tantas definiciones como programadores. Por ello, he consultado la opinión de conocidos y experimentados programadores.

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

Me gusta que mi código sea elegante y eficaz. La lógica debe ser directa para evitar errores ocultos, las dependencias deben ser mínimas para facilitar el mantenimiento, el procesamiento de errores completo y sujeto a una estrategia articulada, y el rendimiento debe ser óptimo para que los usuarios no tiendan a estropear el código con optimizaciones sin sentido. El código limpio hace bien una cosa.

Bjarne usa la palabra «elegante». Menuda palabra.

Según el diccionario, «elegante» significa «*dotado de gracia, nobleza y sencillez*». Aparentemente Bjarne piensa que el código limpio es un placer a la hora de leerlo. Su lectura debe hacernos sonreír, como una caja de música o un coche bien diseñado.

Bjarne también menciona la eficacia, en *dos ocasiones*. No debería sorprendemos viniendo del inventor de C++; pero considero que hay algo más que el mero deseo de velocidad. Los ciclos malgastados no son elegantes, no son un placer. Y fíjese en la palabra empleada por Bjarne para describir la consecuencia de esta falta de elegancia. Usa *tiendan*. Una gran verdad. El código incorrecto *tiende* a aumentar el desastre. Cuando otros cambian código incorrecto, tienden a empeorarlo.

Dave Thomas y Andy Hunt lo expresan de forma diferente. Usan la metáfora de las ventanas rotas[3]. Un edificio con ventanas rotas parece abandonado. Y hace que otros lo abandonen. Dejan que se rompan otras

ventanas. E incluso las rompen a propósito. La fachada se ensucia con pintadas y se acumula la basura. Una ventana rota inicia el proceso de la decadencia.

Bjarne también menciona que el procesamiento de errores debe ser completo, lo que se relaciona con la disciplina de prestar atención a los detalles. El procesamiento de errores abreviado es una forma de ignorar los detalles. Otras son las fugas de memoria, las condiciones de carrera o el uso incoherente de los nombres. En definitiva, el código limpio muestra gran atención al detalle.

Bjarne termina afirmando que el *código limpio hace una cosa bien*. No es accidental que existan tantos principios de diseño de *software* que se puedan reducir a esta sencilla máxima. Muchos escritores han tratado de comunicar este pensamiento. El código incorrecto intenta hacer demasiadas cosas y su cometido es ambiguo y enrevesado. El código limpio es *concreto*. Cada función, cada clase y cada módulo muestran una única actitud que se mantiene invariable y no se contamina por los detalles circundantes.

Grady Booch, autor de *Object Oriented Analysis and Design with Applications*

> *El código limpio es simple y directo. El código limpio se lee como un texto bien escrito. El código limpio no oculta la intención del diseñador, sino que muestra nítidas abstracciones y líneas directas de control.*

Grady mantiene las mismas ideas que Bjarne, pero adopta una perspectiva de *legibilidad*. Me gusta especialmente que el código limpio se pueda leer como un texto bien escrito. Piense en un buen libro. Recordará que las palabras desaparecen y se sustituyen por imágenes, como ver una película.

Mejor todavía. Es ver los caracteres, escuchar los sonidos, experimentar las sensaciones.

Leer código limpio nunca será como leer *El Señor de los Anillos*. Pero esta metáfora literaria no es incorrecta. Como una buena novela, el código limpio debe mostrar de forma clara el suspense del problema que hay que resolver. Debe llevar ese suspense hasta un punto álgido para después demostrar al lector que los problemas y el suspense se han solucionado de forma evidente.

La frase «nítida abstracción» de Grady es un oxímoron fascinante. Nítido es casi un sinónimo de concreto, con un potente mensaje. El código debe ser específico y no especulativo. Sólo debe incluir lo necesario. Nuestros lectores deben percibir que hemos tomado decisiones.

«Big» Dave Thomas, fundador de OTI, el padrino de la estrategia Eclipse

> *El código limpio se puede leer y mejorar por parte de un programador que no sea su autor original. Tiene pruebas de unidad y de aceptación. Tiene nombres con sentido. Ofrece una y no varias formas de hacer algo. Sus dependencias son mínimas, se definen de forma explícita y ofrece una API clara y mínima. El código debe ser culto en función del lenguaje, ya que no toda la información necesaria se puede expresar de forma clara en el código.*

Big Dave comparte el deseo de Grady de la legibilidad, pero con una importante variante. Dave afirma que el código limpio facilita las labores de mejora de *otros*. Puede parecer evidente pero no debemos excedernos. Después de todo, existe una diferencia entre el código fácil de leer y el código fácil de cambiar.

Dave vincula la limpieza a las pruebas. Hace 10 años esto hubiera provocado cierta controversia. Pero la disciplina del Desarrollo controlado por pruebas ha tenido un gran impacto en nuestro sector y se ha convertido en uno de sus pilares. Dave tiene razón. El código, sin pruebas, no es limpio. Independientemente de su elegancia, legibilidad y accesibilidad, si no tiene pruebas, no será limpio.

Dave usa dos veces la palabra *mínimo*. Valora el código de tamaño reducido, una opinión habitual en la literatura de *software* desde su concepción. Cuanto más pequeño, mejor.

También afirma que el código debe ser *culto*, una referencia indirecta a la programación de Knuth[4] y que en definitiva indica que el código debe redactarse de una forma legible para los humanos.

Michael Feathers, autor de *Working Effectively with Legacy Code*

Podría enumerar todas las cualidades del código limpio, pero hay una principal que engloba a todas ellas. El código limpio siempre parece que ha sido escrito por alguien a quien le importa. No hay nada evidente que hacer para mejorarlo. El autor del código pensó en todos los aspectos posibles y si intentamos imaginar alguna mejora, volvemos al punto de partida y sólo nos queda disfrutar del código que alguien a quien le importa realmente nos ha proporcionado.

Una palabra; dar importancia. Es el verdadero tema de este libro, que incluso podría usar el subtítulo «*Cómo dar importancia al código*».

Michael ha acertado de pleno. El código limpio es aquél al que se le ha dado importancia. Alguien ha dedicado su tiempo para que sea sencillo y ha prestado atención a los detalles. Se ha preocupado.

Ron Jeffries, autor de *Extreme Programming Installed* y *Extreme Programming Adventures in C#*

Ron comenzó su carrera como programador con Fortran en Strategic Air Command y ha escrito código para la práctica totalidad de lenguajes y equipos. Merece la pena fijarse en sus palabras:

> En los últimos años, comencé y prácticamente terminé con las reglas de código simple de Beck.
> En orden de prioridad, el código simple:
>
> - Ejecuta todas las pruebas.
> - No contiene duplicados.
> - Expresa todos los conceptos de diseño del sistema.
> - Minimiza el número de entidades como clases, métodos, funciones y similares.
>
> De todos ellos, me quedo con la duplicación. Cuando algo se repite una y otra vez, es una señal de que tenemos una idea que no acabamos de representar correctamente en el código. Intento determinar cuál es y, después, expresar esa idea con mayor claridad. Para mí, la expresividad debe incluir nombres con sentido y estoy dispuesto a cambiar los nombres de las cosas varias veces. Con las modernas herramientas de creación de código como Eclipse, el cambio de nombres es muy sencillo, por lo que no me supone problema alguno.

> La expresividad va más allá de los nombres. También me fijo si un objeto o un método hacen más de una cosa. Si se trata de un objeto, probablemente tenga que dividirse en dos o más. Si se trata de un método, siempre recurro a la refactorización de extracción de métodos para generar un método que exprese con mayor claridad su cometido y varios métodos secundarios que expliquen cómo lo hace.
>
> La duplicación y la expresividad son dos factores que permiten mejorar considerablemente código que no sea limpio. Sin embargo, existe otra cosa que también hago conscientemente, aunque sea más difícil de explicar.
>
> Tras años en este trabajo, creo que todos los programas están formados de elementos muy similares. Un ejemplo es la búsqueda de elementos en una colección. Independientemente de que sea una base de datos de registros de empleados o un mapa de claves y valores, o una matriz de elementos, por lo general tenemos que buscar un elemento concreto de esa colección. Cuando esto sucede, suelo incluir esa implementación concreta en un método o una clase más abstractos. De ese modo disfruto de una serie de interesantes ventajas.
>
> Puedo implementar la funcionalidad con algo sencillo, como un mapa hash, por ejemplo, pero como ahora todas las referencias a la búsqueda se ocultan en mi pequeña abstracción, puedo modificar la implementación siempre que desee. Puedo avanzar rápidamente al tiempo que conservo la posibilidad de realizar cambios posteriores.
>
> Además, la abstracción de la colección suele captar mi atención en lo que realmente sucede e impide que implemente comportamientos de colecciones arbitrarias si lo que realmente necesito es una forma sencilla de localizar un elemento.
>
> Reducir los duplicados, maximizar la expresividad y diseñar sencillas abstracciones en las fases iniciales. Para mí, es lo que hace que el código sea limpio.

En estos breves párrafos, Ron resume el contenido de este libro. Nada de duplicados, un objetivo, expresividad y pequeñas abstracciones. Todo está ahí.

Ward Cunningham, inventor de Wiki, Fit, y uno de los inventores de la programación eXtreme. Uno de los impulsores de los patrones de diseño. Una de las mentes tras Smalltalk y la programación orientada a objetos. El padrino de todos a los que les importa el código.

> *Sabemos que estamos trabajando con código limpio cuando cada rutina que leemos resulta ser lo que esperábamos. Se puede denominar código atractivo cuando el código hace que parezca que el lenguaje se ha creado para el problema en cuestión.*

Este tipo de afirmaciones son características de Ward. Las leemos, asentimos y pasamos a la siguiente. Es tan razonable y evidente que apenas parece profundo. Incluso podemos pensar que es lo que esperábamos. Pero preste atención.

«… resulta ser lo que esperábamos». ¿Cuándo fue la última vez que vio un módulo que fuera más o menos lo que esperaba? ¿Lo habitual no es ver módulos complicados y enrevesados? ¿No es esta falta de concreción lo habitual? ¿No está acostumbrado a intentar extraer el razonamiento de un sistema para llegar al módulo que está leyendo? ¿Cuándo fue la última vez que leyó un código y asintió como seguramente haya hecho al leer la afirmación de Ward?

Ward espera que al leer código limpio no le sorprenda. De hecho, ni siquiera tendrá que esforzarse. Lo leerá y será prácticamente lo que

esperaba. Será evidente, sencillo y atractivo. Cada módulo prepara el camino del siguiente. Cada uno indica cómo se escribirá el siguiente. Los programas limpios están tan bien escritos que ni siquiera se dará cuenta. El diseñador consigue simplificarlo todo enormemente, como sucede con todos los diseños excepcionales.

¿Y la noción de atractivo de Ward? Todos hemos criticado que nuestros lenguajes no se hayan diseñado para nuestros problemas. Pero la afirmación de Ward hace que ahora la responsabilidad sea nuestra. Afirma que *el código atractivo hace que el lenguaje parezca creado para el problema*. Por tanto, somos responsables de que el lenguaje parezca sencillo. No es el lenguaje el que hace que los programas parezcan sencillos, sino el programador que consigue que el lenguaje lo parezca.

Escuelas de pensamiento

¿Y yo (Uncle Bob)? ¿Qué es para mí el código limpio? En este libro le contaremos, con todo detalle, lo que yo y mis colegas pensamos del código limpio. Le contaremos lo que pensamos que hace que un nombre de variable, una función o una clase sean limpias.

Presentaremos estas opiniones de forma absoluta, sin disculparnos. En este punto de nuestra carrera, ya son absolutas. Son *nuestra escuela de pensamiento* del código limpio.

Los especialistas de las artes marciales no se ponen de acuerdo sobre cuál es la mejor de todas, ni siquiera sobre cuál es la mejor técnica de un arte marcial. Es habitual que los maestros de las artes marciales creen sus propias escuelas de pensamiento y los alumnos aprendan de ellos. De esta forma nació *Gracie Jiu Jitsu*, creada e impartida por la familia Gracie en Brasil; *Hakkoryu Jiu Jitsu*, fundada e impartida por Okuyama Ryuho en Tokio o *Jeet Kune Do*, fundada e impartida por Bruce Lee en Estados Unidos.

Los alumnos de estas disciplinas se sumergen en las enseñanzas del fundador. Se dedican a aprender lo que su maestro les enseña y suelen excluir las enseñanzas de otros maestros. Después, cuando han mejorado su arte, pueden convertirse en alumnos de otro maestro diferente para ampliar sus conocimientos y su experiencia. Algunos seguirán mejorando sus habilidades, descubriendo nuevas técnicas y fundando sus propias escuelas.

Ninguna de estas escuelas tiene la *razón* absoluta pero dentro de cada una actuamos como si las enseñanzas y las técnicas fueran correctas. Después de todo, existe una forma correcta de practicar Hakkoryu Jiu Jitsu o Jeet Kune Do, pero esta corrección dentro de una escuela determinada no anula las enseñanzas de otra diferente.

Imagine que este libro es una descripción de la *Escuela de mentores del código limpio*. Las técnicas y enseñanzas impartidas son la forma en la que practicamos nuestro arte. Podemos afirmar que, si sigue nuestras enseñanzas, disfrutará de lo que hemos disfrutado nosotros, y aprenderá a crear código limpio y profesional. Pero no cometa el error de pensar que somos los únicos que tenemos razón. Existen otras escuelas y otros maestros tan profesionales como nosotros, y su labor es aprender de ellos también.

De hecho, muchas de las recomendaciones del libro son controvertidas, seguramente no esté de acuerdo con muchas de ellas y puede que rechace algunas de forma definitiva. Es correcto. No somos la autoridad final. Pero, por otra parte, las recomendaciones del libro son algo en lo que hemos pensado mucho. Las hemos aprendido tras décadas de experiencia y ensayo y error. Por lo tanto, esté o no de acuerdo, sería una lástima que no apreciara, y respetara, nuestro punto de vista.

Somos autores

El campo `@author` de un Javadoc indica quiénes somos. Somos autores. Y los autores tienen lectores. De hecho, los autores son *responsables* de comunicarse correctamente con sus lectores. La próxima vez que escriba una línea de código, recuerde que es un autor y que escribe para que sus lectores juzguen su esfuerzo.

Seguramente se pregunte qué cantidad de código se lee realmente y si la mayor parte del esfuerzo no se concentra en crearlo.

¿Alguna vez ha reproducido una sesión de edición? En las décadas de 1980 y 1990 teníamos editores como Emacs que controlaban cada pulsación de tecla. Se podía trabajar durante una hora y después reproducir la sesión de edición completa como una película a alta velocidad. Cuando lo hice, los resultados fueron fascinantes.

La mayor parte de la reproducción eran desplazamientos entre módulos.

> *Bob accede al módulo.*
> *Se desplaza hasta la función que tiene que cambiar.*
> *Se detiene y piensa en las posibles opciones.*
> *Oh, vuelve al inicio del módulo para comprobar la inicialización de una variable.*
> *Ahora vuelve a bajar y comienza a escribir.*
> *Vaya, borra lo que había escrito.*
> *Vuelve a escribirlo.*
> *Lo vuelve a borrar.*
> *Escribe algo diferente pero también lo borra.*
> *Se desplaza a otra función que invoca la función que está modificando para comprobar cómo se invoca.*
> *Vuelve a subir y escribe el mismo código que acaba de borrar.*
> *Se detiene.*
> *Vuelve a borrar el código.*
> *Abre otra ventana y examina las subclases. ¿Se ha reemplazado esa función?*
> …

Se hace una idea. En realidad, la proporción entre tiempo dedicado a leer frente a tiempo dedicado a escribir es de más de 10:1. *Constantemente*

tenemos que leer código antiguo como parte del esfuerzo de crear código nuevo.

Al ser una proporción tan elevada, queremos que la lectura del código sea sencilla, aunque eso complique su creación. Evidentemente, no se puede escribir código sin leerlo, de modo que *si es más fácil de leer será más fácil de escribir*.

Es una lógica sin escapatoria. No se puede escribir código si no se puede leer el código circundante. El código que intente escribir hoy será fácil o difícil de escribir en función de lo fácil o difícil de leer que sea el código circundante. Si quiere avanzar rápidamente, terminar cuanto antes y que su código sea fácil de escribir, haga que sea fácil de leer.

La regla del Boy Scout

No basta con escribir código correctamente. El código debe limpiarse con el tiempo. Todos hemos visto que el código se corrompe con el tiempo, de modo que debemos adoptar un papel activo para evitarlo.

Los Boy Scouts norteamericanos tienen una sencilla regla que podemos aplicar a nuestra profesión:

> *Dejar el campamento más limpio de lo que se ha encontrado*[5].

Si todos entregamos el código más limpio de lo que lo hemos recibido, no se corromperá. No hace falta que la limpieza sea masiva. Cambie el nombre de una variable, divida una función demasiado extensa, elimine elementos duplicados, simplifique una instrucción `if` compuesta.

¿Se imagina trabajar en un proyecto en el que el código *mejorara* con el tiempo? ¿Cree que hay otras opciones que puedan considerarse profesionales? De hecho, ¿la mejora continuada no es una parte intrínseca de la profesionalidad?

Precuela y principios

En muchos aspectos, este libro es una «precuela» de otro que escribí en 2002 titulado *Agile Software Development: Principles, Patterns, and Practices* (PPP). El libro PPP trata sobre los principios del diseño orientado a objetos y muchas de las técnicas empleadas por desarrolladores profesionales. Si no ha leído PPP, comprobará que continúa la historia contada en este libro. Si lo ha leído, encontrará muchas de las sensaciones de ese libro reproducidas en éste a nivel del código.

En este libro encontrará referencias esporádicas a distintos principios de diseño como SRP (*Single Responsibility Principle* o Principio de responsabilidad única), OCP (*Open Closed Principle* o Principio Abierto/Cerrado) y DIP (*Dependency Inversion Principle* o Principio de inversión de dependencias) entre otros. Todos estos principios se describen detalladamente en PPP.

Conclusión

Los libros sobre arte no le prometen que se convertirá en artista. Solamente pueden mostrarle herramientas, técnicas y procesos de pensamiento que otros artistas hayan utilizado. Del mismo modo, este libro no puede prometer que se convierta en un buen programador, que tenga sentido del código. Sólo puede mostrarle los procesos de pensamiento de buenos programadores y los trucos, técnicas y herramientas que emplean.

Al igual que un libro sobre arte, este libro está repleto de detalles. Encontrará mucho código. Verá código correcto y código incorrecto. Verá código incorrecto transformado en código correcto. Verá listas de heurística, disciplinas y técnicas. Verá un ejemplo tras otro. Y después de todo, será responsabilidad suya.

¿Recuerda el chiste sobre el violinista que se pierde camino de un concierto? Se cruza con un anciano y le pregunta cómo llegar al Teatro Real. El anciano mira al violinista y al violín que lleva bajo el brazo y le responde: «Practique joven, practique».

Bibliografía

- **[Beck07]**: *Implementation Patterns*, Kent Beck, Addison-Wesley, 2007.
- **[Knuth92]**: *Literate Programming*, Donald E. Knuth, Center for the Study of Language and Information, Leland Stanford Junior University, 1992.

2
Nombres con sentido

por Tim Ottinger

Introducción

En el *software*, los nombres son omnipresentes. Aparecen en variables, funciones, argumentos, clases y paquetes. Asignamos nombres a archivos y a directorios, a archivos jar, war y ear. Usamos nombres constantemente. Por ello, debemos hacerlo bien. A continuación, veremos algunas reglas básicas para crear nombres correctos.

Usar nombres que revelen las intenciones

Es fácil afirmar que los nombres deben revelar nuestras intenciones. Lo que queremos recalcar es la *importancia* de hacerlo. Elegir nombres correctos lleva tiempo, pero también ahorra trabajo. Por ello, preste atención a los nombres y cámbielos cuando encuentre otros mejores. Todo el que lea su código se lo agradecerá.

El nombre de una variable, función o clase debe responder una serie de cuestiones básicas. Debe indicar por qué existe, qué hace y cómo se usa. Si un nombre requiere un comentario, significa que no revela su cometido.

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

El nombre `d` no revela nada. No evoca una sensación de tiempo transcurrido, ni de días. Debe elegir un nombre que especifique lo que se mide y la unidad de dicha medida:

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

La elección de nombres que revelen intenciones facilita considerablemente la comprensión y la modificación del código. ¿Para qué sirve el siguiente código?

```
public List<int[]> getThem() {
    List<int[]> list1 = new ArrayList<int[]>();
    for (int[] x : theList)
        if (x[0] == 4)
            list1.add(x);
    return list1;
}
```

¿Por qué es complicado saber la función de este código? No hay expresiones complejas. Los espacios y el sangrado son razonables. Sólo hay tres variables y dos constantes. Ni siquiera contiene clases complejas o métodos polimórficos, sólo una lista de matrices (o eso parece).

El problema no es la simplicidad del código sino su carácter *implícito*: el grado en el que el contexto no es explícito en el propio código. Implícitamente, el código requiere que sepamos las respuestas a las siguientes preguntas:

- ¿Qué contiene `theList`?
- ¿Qué significado tiene el subíndice cero de un elemento de `theList`?
- ¿Qué importancia tiene el valor `4`?
- ¿Cómo se usa la lista devuelta?

Las respuestas a estas preguntas no se encuentran en el código, pero se podrían haber incluido. Imagine que trabaja en un juego de buscar minas. El tablero es una lista de celdas llamada `theList`. Cambiemos el nombre por `gameBoard`.

Cada celda del teclado se representa por medio de una matriz. El subíndice cero es la ubicación de un valor de estado que, cuando es `4`, significa que se ha detectado. Al asignar nombres a estos conceptos mejoramos considerablemente el código:

```
public List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<int[]>();
    for (int[] cell : gameBoard)
        if (cell[STATUS_VALUE] == FLAGGED)
            flaggedCells.add(cell);
    return flaggedCells;
}
```

La simplicidad del código no ha cambiado. Sigue teniendo los mismos operadores y constantes y el mismo número de niveles anidados, pero ahora es mucho más explícito. Podemos crear una sencilla clase para celdas en lugar de usar una matriz de elementos `int`. Puede incluir una función que revele el objetivo (con el nombre `isFlagged`) para ocultar los números. El resultado es una nueva versión de la función:

```
public List<Cell> getFlaggedCells() {
    List<Cell> flaggedCells = new ArrayList<Cell>();
    for (Cell cell : gameBoard)
        if (cell.isFlagged())
            flaggedCells.add(cell);
    return flaggedCells;
}
```

Con estos sencillos cambios de nombre, es fácil saber qué sucede. Es la ventaja de seleccionar nombres adecuados.

Evitar la desinformación

Los programadores deben evitar dejar pistas falsas que dificulten el significado del código. Debemos evitar palabras cuyo significado se aleje del que pretendemos. Por ejemplo, `hp`, `aix` y `sco` son nombres de variables pobres ya que son los nombres de plataformas o variantes de Unix. Aunque se trate del código de una hipotenusa y `hp` parezca la abreviatura correcta, puede no serlo.

No haga referencia a un grupo de cuentas como `accountList` a menos que realmente sea una lista (`List`). La palabra lista tiene un significado concreto para los programadores. Si el contenedor de las cuentas no es realmente una lista, puede provocar falsas conclusiones[6]. Por tanto, resulta

más adecuado usar `accountGroup`, `bunchOfAccounts` o simplemente `accounts`.

Evite usar nombres con variaciones mínimas. ¿Cuánto se tarda en apreciar la sutil diferencia entre `XYZControllerForEfficientHandlingOfStrings` y `XYZControllerForEfficientStorageOfStrings` en un módulo? Ambas palabras tienen una forma similar.

La ortografía similar de conceptos parecidos es información; el uso de ortografía incoherente es desinformación. En los entornos modernos de Java, el código se completa de forma automática. Escribimos varios caracteres de un nombre y pulsamos varias teclas para obtener una lista de posibles opciones de un nombre. Es muy útil si los nombres de elementos similares se ordenan alfabéticamente de forma conjunta y si las diferencias son muy evidentes, ya que es probable que el programador elija un objeto por nombre sin fijarse en los comentarios o la lista de métodos proporcionados por una clase.

Un ejemplo de nombre desinformativo sería el uso de la `L` minúscula o la `O` mayúscula como nombres de variables, sobre todo combinados. El problema, evidentemente, es que se parecen a las constantes `1` y `0` respectivamente:

```
int a = l;
if ( O == l )
    a = O1;
else
    l = 01;
```

El lector puede pensar que es una invención, pero hemos visto código con abundancia de estos elementos. En un caso, el autor del código, sugirió usar una fuente distinta para que las diferencias fueran más evidentes, una solución que se hubiera transmitido a todos los futuros programadores como tradición oral o en un documento escrito. El problema se resolvió con carácter definitivo y sin necesidad de crear nuevos productos, con tan sólo cambiar los nombres.

Realizar distinciones con sentido

Los programadores se crean un problema al crear código únicamente dirigido a un compilador o intérprete. Por ejemplo, como se puede usar el mismo nombre para hacer referencia a dos elementos distintos en el mismo ámbito, puede verse tentado a cambiar un nombre de forma arbitraria. En ocasiones se hace escribiéndolo incorrectamente, lo que provoca que los errores ortográficos impidan la compilación[7].

No basta con añadir series de números o palabras adicionales, aunque eso satisfaga al compilador. Si los nombres tienen que ser distintos, también deben tener un significado diferente.

Los nombres de series numéricas (`a1`, `a2`… `aN`) son lo contrario a los nombres intencionados. No desinforman, simplemente no ofrecen información; son una pista sobre la intención del autor. Fíjese en lo siguiente:

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

Esta función se lee mejor cuando se usa `source` y `destination` como nombres de argumentos.

Las palabras adicionales son otra distinción sin sentido. Imagine que tiene la clase Product. Si tiene otra clase con el nombre `ProductInfo` o `ProductData`, habrá creado nombres distintos, pero con el mismo significado. `Info` y `Data` son palabras adicionales, como `a`, `an` y `the`.

No es incorrecto usar prefijos como `a` y `the` mientras la distinción tenga sentido. Imagine que usa `a` para variables locales y `for` para argumentos de funciones[8]. El problema aparece cuando decide invocar la variable `theZork` porque ya tiene otra variable con el nombre `zork`.

Las palabras adicionales son redundantes. La palabra `variable` no debe incluirse nunca en el nombre de una variable. La palabra `table` no debe incluirse nunca en el nombre de una tabla. ¿Es mejor `NameString` que `Name`? ¿Podría ser `Name` un número de coma flotante? En caso afirmativo,

incumple la regla anterior sobre desinformación. Imagine que encuentra una clase con el nombre `Customer` y otra con el nombre `CustomerObject`. ¿Cuál sería la distinción? ¿Cuál representa mejor el historial de pagos de un cliente?

Existe una aplicación que lo ilustra. Hemos cambiado los nombres para proteger al culpable. Veamos el error exacto:

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

¿Cómo saben los programadores de este proyecto qué función deben invocar?

En ausencia de convenciones concretas, la variable `moneyAmount` no se distingue de `money`, `customerInfo` no se distingue de `customer`, `accountData` no se distingue de `account` y `theMessage` no se distingue de `message`. Debe diferenciar los nombres de forma que el lector aprecie las diferencias.

Usar nombres que se puedan pronunciar

A los humanos se nos dan bien las palabras. Gran parte de nuestro cerebro se dedica al concepto de palabras. Y, por definición, las palabras son pronunciables. Sería una pena malgastar esa parte de nuestro cerebro dedicada al lenguaje hablado. Por tanto, cree nombres pronunciables. Si no lo puede pronunciar, no podrá explicarlo sin parecer tonto. Es un factor importante, ya que la programación es una actividad social.

Conozco una empresa que usa `genymdhms` (fecha de generación, año, mes, día, hora, minuto y segundo) y lo pronuncian tal cual. Yo tengo la costumbre de pronunciar todo tal y como lo veo escrito, de forma que muchos analistas y diseñadores acabaron por llamarme algo como «genimedemes». Era un chiste y nos parecía divertido, pero en realidad estábamos tolerando el uso de nombres pobres. Teníamos que explicar las variables a los nuevos programadores y cuando las pronunciaban, usaban palabras inventadas en lugar de nombres correctos. Compare:

```
class DtaRcrd102 {
    private Date genymdhms;
    private Date modymdhms;
    private final String pszqint = “102”;
    /*… */
};
```

con:

```
class Customer {
    private Date generationTimestamp;
    private Date modificationTimestamp;
    private final String recordId = “102”;
    /*… */
};
```

Ahora se puede mantener una conversación inteligente: «Eh, Mikey, fíjate en este registro. La marca de tiempo de generación es para mañana. ¿Cómo es posible?»

Usar nombres que se puedan buscar

Los nombres de una letra y las constantes numéricas tienen un problema: no son fáciles de localizar en el texto. Se puede detectar `MAX_CLASSES_PER_STUDENT`, pero el número 7 resulta más complicado. Las búsquedas pueden devolver el dígito como parte de nombres de archivo, otras definiciones de constantes o expresiones en las que se use con otra intención. Mucho peor si la constante es un número extenso y alguien ha intercambiado los dígitos, lo que genera un error inmediato y no aparece en la búsqueda.

Del mismo modo, el nombre `e` es una opción muy pobre para variables que el programador tenga que buscar. Es la letra más usada en inglés y aparece en la práctica totalidad de los textos de un programa. A este respecto, los nombres extensos superan a los breves y cualquier nombre que se pueda buscar supera a una constante en el código.

Personalmente prefiero nombres de una letra que sólo se puedan usar como variables locales dentro de métodos breves. *La longitud de un nombre debe corresponderse al tamaño de su ámbito* [N5]. Si una variable o constante se usa en varios puntos del código, debe asignarle un nombre que se pueda buscar. Compare:

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

con:

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

En este ejemplo, `sum` no es un nombre especialmente útil, pero al menos se puede buscar. Se usa una función más extensa, pero comprobará

que resulta mucho más fácil buscar `WORK_DAYS_PER_WEEK` que todas las instancias de `5` y filtrar la lista a los casos con el significado adecuado.

Evitar codificaciones

Ya tenemos suficientes codificaciones como para tener que añadir otras nuevas. Al codificar información de tipos o ámbitos en un nombre se dificulta la descodificación. No parece razonable que todos los nuevos empleados tengan que aprender otro lenguaje de codificación además del código con el que van a trabajar. Es una carga mental innecesaria a la hora de intentar solucionar un problema. Los nombres codificados resultan impronunciables y suelen escribirse de forma incorrecta.

Notación húngara

Antiguamente, cuando trabajábamos con lenguajes en los que la longitud de los nombres era un reto, incumplíamos esta regla a nuestro pesar. Fortran forzaba las codificaciones convirtiendo la primera letra de un tipo en código. En sus primeras versiones, BASIC sólo se permitía una letra y un dígito. La notación húngara (HN) lo llevó a un nuevo nivel.

HN se consideraba muy importante en el API C de Windows, donde todo era un control entero, un puntero `long`, un puntero `void` o una de varias implementaciones de `string` (con diferentes usos y atributos). Por aquel entonces, el compilador no comprobaba los tipos, de modo que los programadores tenían que recordarlos.

En los lenguajes modernos disponemos de sistemas de tipos más completos y los compiladores recuerdan los tipos y los aplican. Es más, existe una tendencia a usar clases y funciones más breves para que los usuarios aprecien la declaración de las variables que usan.

Los programadores de Java no tienen que codificar tipos. Los objetos son de tipos fuertes y los entornos de edición han avanzado tanto que detectan un error de tipo antes de ejecutar la compilación. Por ello, en la actualidad HN y otras formas de codificación de tipos no son más que un impedimento. Hacen que sea más complicado cambiar el nombre o el tipo de una variable o clase. Dificultan la legibilidad del código y pueden hacer que el sistema de codificación confunda al lector:

```
PhoneNumber phoneString;
// el nombre no cambia cuando cambia el tipo
```

Prefijos de miembros

Tampoco es necesario añadir `m_` como prefijo a los nombres de variables. Las clases y funciones tienen el tamaño necesario para no tener que hacerlo, y debe usar un entorno de edición que resalte o coloree los miembros para distinguirlos.

```
public class Part {
    private String m_dsc; // La descripción textual
    void setName(String name) {
        m_dsc = name;
    }
}
```

```
public class Part {
    String description;
    void setDescription(String description) {
        this.description = description;
    }

}
```

Además, los usuarios aprenden rápidamente a ignorar el prefijo (o sufijo) y fijarse en la parte con sentido del nombre. Cuanto más código leemos, menos nos fijamos en los prefijos. En última instancia, los prefijos son un indicio de código antiguo.

Interfaces e Implementaciones

Existe un caso especial para usar codificaciones. Imagine por ejemplo que crea una factoría abstracta para crear formas. Esta factoría será una interfaz y se implementará por medio de una clase concreta. ¿Qué nombres debe asignar? ¿`IShapeFactory` y `ShapeFactory`? Prefiero las interfaces sin adornos. La I inicial, tan habitual en los archivos de legado actuales es, en el mejor de los casos, una distracción, y en el peor, un exceso de información. No quiero que mis usuarios sepan que se trata de una interfaz, solamente que se trata de `ShapeFactory`. Si tengo que codificar la interfaz o la implementación, opto por ésta última. Es mejor usar `ShapeFactoryImp` o incluso `CShapeFactory`, que codificar la interfaz.

Evitar asignaciones mentales

Los lectores no tienen que traducir mentalmente sus nombres en otros que ya conocen. Este problema suele aparecer al elegir entre no usar términos de dominio de problemas o de soluciones.

Es un problema de los nombres de variables de una sola letra. Un contador de bucles se podría bautizar como `i`, `j` o `k` (pero nunca `l`) si su ámbito es muy reducido y no hay conflictos con otros nombres, ya que los nombres de una letra son tradicionales en contadores de bucles. Sin embargo, en otros contextos, un nombre de una letra es una opción muy pobre: es como un marcador de posición que el lector debe asignar mentalmente a un concepto real. No hay peor motivo para usar el nombre `c` que `a` y `b` ya estén seleccionados.

Por lo general, los programadores son gente inteligente. A la gente inteligente le gusta presumir de sus habilidades mentales. Si puede recordar que `r` es la versión en minúscula de una URL sin el host y el sistema, debe ser muy listo.

Una diferencia entre un programador inteligente y un programador profesional es que este último sabe que la *claridad es lo que importa*. Los profesionales usan sus poderes para hacer el bien y crean código que otros puedan entender.

Nombres de clases

Las clases y los objetos deben tener nombres o frases de nombre como `Customer`, `WikiPage`, `Account` y `AddressParser`. Evite palabras como `Manager`, `Processor`, `Data`, o `Info` 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 verbo como `postPayment`, `deletePage` o `save`. Los métodos de acceso, de modificación y los predicados deben tener como nombre su valor y usar como prefijo `get`, `set` e `is` de acuerdo al estándar de javabean[9].

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())…
```

Al sobrecargar constructores, use métodos de factoría estáticos con nombres que describan los argumentos. Por ejemplo:

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

es mejor que:

```
Complex fulcrumPoint = new Complex(23.0);
```

Refuerce su uso convirtiendo en privados sus constructores correspondientes.

No se exceda con el atractivo

Si los nombres son demasiado inteligentes, sólo los recordarán los que compartan el sentido del humor de su autor, y sólo mientras se acuerden del chiste. ¿Sabrán qué significa la función `HolyHandGrenade`? Sin duda es atractiva, pero en este caso puede que `DeleteItems` fuera más indicado. Opte por la claridad antes que por el entretenimiento. En el código, el atractivo suele aparecer como formas coloquiales o jergas. Por ejemplo, no use `whack()` en lugar de `kill()`. No recurra a bromas culturales como `eatMyShorts()` si quiere decir `abort()`.

Diga lo que piense. Piense lo que diga.

Una palabra por concepto

Elija una palabra por cada concepto abstracto y manténgala. Por ejemplo, resulta confuso usar `fetch`, `retrieve` y `get` como métodos equivalentes de clases distintas. ¿Cómo va a recordar qué método se corresponde a cada clase? Desafortunadamente, tendrá que recordar qué empresa, grupo o individuo ha creado la biblioteca o clase en cuestión para recordar qué

término se ha empleado. En caso contrario, perderá mucho tiempo buscando en encabezados y fragmentos de código.

Los entornos de edición modernos como Eclipse e IntelliJ ofrecen pistas sensibles al contexto, como la lista de métodos que puede invocar en un determinado objeto. Pero esta lista no suele incluir los comentarios de nombres de funciones y listas de parámetros. Tendrá suerte si muestra los nombres de parámetros de las declaraciones de funciones. Los nombres de funciones deben ser independientes y coherentes para que pueda elegir el método correcto sin necesidad de búsquedas adicionales.

Del mismo modo, resulta confuso tener un controlador, un administrador y un control en la misma base de código. ¿Cuál es la diferencia entre `DeviceManager` y `ProtocolController`? ¿Por qué no son los dos controladores o administradores? ¿Son controladores? El nombre hace que espere que dos objetos tengan un tipo diferente y clases diferentes.

Un léxico coherente es una gran ventaja para los programadores que tengan que usar su código.

No haga juegos de palabras

Evite usar la misma palabra con dos fines distintos. Suele hacerse en juegos de palabras. Si aplica la regla de una palabra por conceptos, acabará con muchas clases que por ejemplo tengan un método `add`. Mientras las listas de parámetros y los valores devueltos de los distintos métodos `add` sean semánticamente equivalentes, no hay problema.

Sin embargo, alguien puede decidir usar la palabra `add` por motivos de coherencia, aunque no sea en el mismo sentido. Imagine que hay varias clases en las que `add` crea un nuevo valor sumando o concatenando dos valores existentes. Imagine ahora que crea una nueva clase con un método que añada su parámetro a una colección. ¿Este método debe tener el método `add`? Parece coherente ya que hay otros muchos métodos `add`, pero en este caso hay una diferencia semántica, de modo que debemos usar un nombre como `insert` o `append`. Llamar `add` al nuevo método sería un juego de palabras.

Nuestro objetivo, como autores, es facilitar la comprensión del código. Queremos que el código sea algo rápido, no un estudio exhaustivo. Queremos usar un modelo en el que el autor sea el responsable de transmitir

el significado, no un modelo académico que exija investigar el significado mostrado.

Usar nombres de dominios de soluciones

Recuerde que los lectores de su código serán programadores. Por ello, use términos informáticos, algoritmos, nombres de patrones, términos matemáticos y demás. No conviene extraer todos los nombres del dominio de problemas ya que no queremos que nuestros colegas tengan que preguntar el significado de cada nombre en especial cuando ya conocen el concepto bajo otro nombre diferente.

El nombre `AccountVisitor` tiene mucho significado para un programador familiarizado con el patrón `VISITOR`. ¿Qué programador no sabe lo que es `JobQueue`? Hay cientos de cosas técnicas que los programadores tienen que hacer y elegir nombres técnicos para dichas cosas suele ser lo más adecuado.

Usar nombres de dominios de problemas

Cuando no exista un término de programación para lo que esté haciendo, use el nombre del dominio de problemas. Al menos el programador que mantenga su código podrá preguntar el significado a un experto en dominios.

Separar los conceptos de dominio de soluciones y de problemas es parte del trabajo de un buen programador y diseñador. El código que tenga más relación con los conceptos del dominio de problemas tendrá nombres extraídos de dicho dominio.

Añadir contexto con sentido

Algunos nombres tienen significado por sí mismos, pero la mayoría no. Por ello, debe incluirlos en un contexto, en clases, funciones y espacios de nombres con nombres adecuados. Cuando todo lo demás falle, pueden usarse prefijos como último recurso.

Imagine que tiene las variables `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state` y `zipcode`. Si las combina, es evidente que forman una dirección. Pero si la variable `state` se usa de forma aislada en un método, ¿sabría que forma parte de una dirección? Puede añadir contexto por medio de prefijos: `addrFirstName`, `addrLastName`, `addrState`, etc. Al menos los lectores comprenderán que estas variables forman parte de una estructura mayor. Evidentemente, es mejor crear la clase `Address`. De ese modo, incluso el compilador sabrá que las variables pertenecen a un concepto más amplio.

Fíjese en el método del Listado 2-1. ¿Las variables necesitan un contexto con más sentido? El nombre de la función sólo ofrece parte del contexto, el resto se obtiene del algoritmo. Tras leer la función, verá que las tres variables `number`, `verb` y `pluralModifier` forman parte del mensaje `guess statistics`. Desafortunadamente, es necesario inferir el contexto. Al leer el método, el significado de las variables no es evidente.

Listado 2-1
Variables en un contexto ambiguo.

```
private void printGuessStatistics(char candidate, int count) {
    String number;
    String verb;
    String pluralModifier;
    if (count == 0) {
        number = “no”;
        verb = “are”;
        pluralModifier = “s”;
    } else if (count == 1) {
        number = “1”;
        verb = “is”;
        pluralModifier = “”;
    } else {
        number = Integer.toString(count);
        verb = “are”;
        pluralModifier = “s”;
    }
    String guessMessage = String.format(
        “There %s %s %s%s”, verb, number, candidate, pluralModifier
    );
    print(guessMessage);
}
```

La función es demasiado extensa y las variables aparecen por todas partes. Para dividir la función en fragmentos más reducidos necesitamos crear una clase `GuessStatisticsMessage` y convertir a las tres variables en campos de la misma. De este modo contamos con un contexto más obvio para las tres variables. Forman parte sin duda de `GuessStatisticsMessage`. La mejora del contexto también permite que el algoritmo sea más limpio y se divida en funciones más reducidas (véase el Listado 2-2).

Listado 2-2

Variables con un contexto.

```
public class GuessStatisticsMessage (
    private String number;
    private String verb;
    private String pluralModifier;


    public String make(char candidate, int count) {
    createPluralDependentMessageParts(count);
    return String.format(
        “There %s %s %s%s,
        verb, number, candidate, pluralModifier);
    }


    private void createPluralDependentMessageParts(int count) {
        if (count == 0) {
            thereAreNoLetters();
        } else if (cout == 1) {
            thereIsOneLetter();
        } else {
            thereAreManyLetters(count);
        }
    }


    private void thereAreManyLetters(int count) {
        number = “1”;
        verb = “is”;
        pluralModifier = “”;
    }


    private void thereIsOneLetter() {
        number = “1”;
        verb = “is”;
        pluralModifier = “”;
    }


    private void thereAreNoLetters() {
        number = “no”;
        verb = “are”;
        pluralModifier = “s”;
    }
}
```

No añadir contextos innecesarios

En la aplicación imaginaria Gas Station Deluxe, no es aconsejable usar el prefijo `GSD` en todas las clases. Es trabajar contra las herramientas proporcionadas. Introduzca `G` y pulse la tecla de finalización para acceder a una lista interminable de todas las clases del sistema. ¿Es lo correcto? ¿Por qué dificultar la ayuda del IDE?

Del mismo modo, imagine que ha creado la clase `MailingAddress` en un módulo de contabilidad de `GSD`, con el nombre `GSDAccountAddress`. Después, necesita una dirección de correo para la aplicación de contacto con el cliente. ¿Usará `GSDAccountAddress`? ¿Le parece el nombre correcto? 10 de los 17 caracteres son redundantes o irrelevantes.

Los nombres breves suelen ser más adecuados que los extensos, siempre que sean claros. No añada más contexto del necesario a un nombre. Los nombres `accountAddress` y `customerAddress` son perfectos para instancias de la clase `Address` pero no sirven como nombres de clase.

`Address` sirve como nombre de clase. Para distinguir entre direcciones MAC, direcciones de puertos y direcciones Web, podría usar `PostalAddress`, `MAC` y `URI`. Los nombres resultantes son más precisos, el objetivo de cualquier nombre.

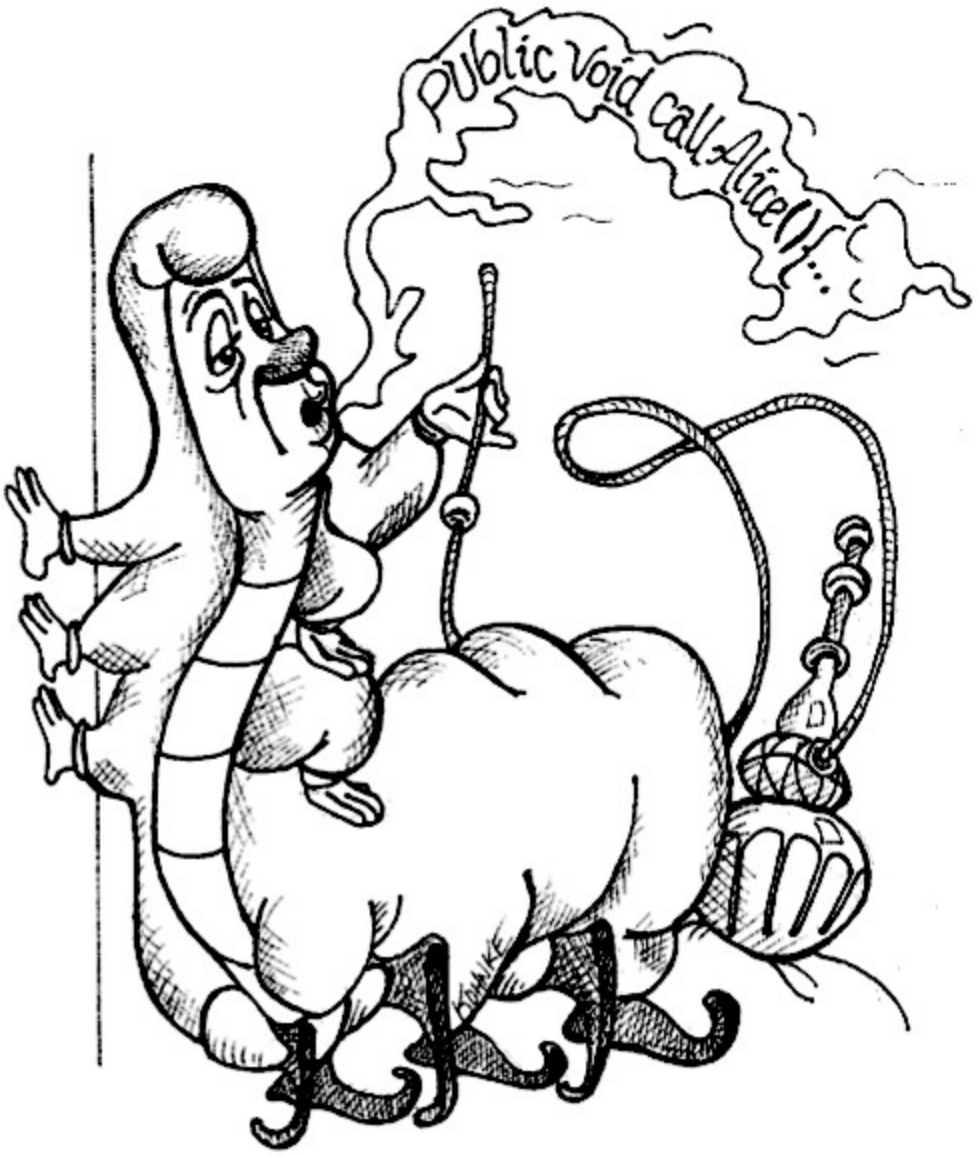
Conclusión

Lo más complicado a la hora de elegir un buen nombre es que requiere habilidad descriptiva y acervo cultural. Es un problema de formación más que técnico, empresarial o administrativo. Como resultado, mucha gente del sector no aprende a hacerlo bien.

La gente teme que al cambiar los nombres otros programadores se quejen. Nosotros no compartimos ese temor y agradecemos los cambios de nombre (siempre que sean a mejor). En muchos casos no memorizamos los nombres de clases y métodos. Usamos herramientas modernas para estos detalles y así poder centrarnos en si el código se lee como frases o párrafos, o al menos como tablas y estructuras de datos (una frase no siempre es la mejor forma de mostrar datos). Seguramente acabará sorprendiendo a alguien cuando cambie los nombres, como puede suceder con cualquier otra mejora del código. No deje que le detenga.

Aplique estas reglas y compruebe si mejora o no la legibilidad de su código. Si es el encargado de mantener código de terceros, use herramientas para solucionar estos problemas. Obtendrá recompensas a corto y largo plazo.

3
Funciones



En los inicios de la programación, creábamos sistemas a partir de rutinas y subrutinas. Después, en la época de Fortran y PL/1, creábamos nuestros sistemas con programas, subprogramas y funciones. En la actualidad, sólo las funciones han sobrevivido. Son la primera línea organizativa en cualquier programa. En este capítulo veremos cómo crearlas.

Fíjese en el código del Listado 3-1. Es complicado encontrar una función extensa en FitNesse[10], pero acabé encontrando ésta. No sólo es extensa, sino que también contiene código duplicado, muchas cadenas y tipos de datos extraños, además de API poco habituales y nada evidentes. Intente comprenderlo en los próximos tres minutos.

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

```
public static String testableHtml {
    PageData pageData,
    boolean includeSuiteSetup
} throws Exception {
    WikiPage wikiPage = pageData.getWikiPage();
    StringBuffer buffer = new StringBuffer();
    if (pageData.hasAttribute(“Test”)) {
        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”);
            }
        }
        WikiPage setup =
            PageCrawlerImpl.getInheritedPage(“SetUp”, wikiPage);
        if (setup != null) {
            WikiPagePath setupPath =
                wikiPage.getPageCrawler().getFullPath(setup);
            String setupPathName = PathParser.render(setupPath);
            buffer.append(“!include -setup .”)
                .append(setupPathName)
                .append(“\n”);
        }
    }
    buffer.append(pageData.getContent());
    if (pageData.hasAttribute(“Test”)) {
        WikiPage teardown =
            PageCrawlerImpl.getInheritedPage(“TearDown”, wikiPage);
        if (teardown != null) {
            WikiPagePath tearDownPath =
                wikiPage.getPageCrawler().getFullPath(teardown);
            String tearDownPathName = PathParser.render(tearDownPath);
            buffer.append(“\n”)
                .append(“!include -teardown .”)
                .append(tearDownPathName)
                .append(“\n”);
        }
        if (includeSuiteSetup) {
            WikiPage suiteTeardown =
                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();
}
```

¿Tras tres minutos entiende la función? Seguramente no. Pasan demasiadas cosas y hay demasiados niveles de abstracción diferentes. Hay cadenas extrañas e invocaciones de funciones mezcladas en instrucciones `if` doblemente anidadas controladas por indicadores. Sin embargo, con sencillas extracciones de código, algún cambio de nombres y cierta reestructuración, pude capturar la intención de la función en las nueve líneas del Listado 3-2. Compruebe si ahora la entiende.

Listado 3-2
HtmlUtil.java (refactorización).

```
public static String renderPageWithSetupsAndTeardowns(
    PageData pageData, boolean isSuite
) throws Exception {
    boolean isTestPage = pageData.hasAttribute("Test");
    if (isTestPage) {
        WikiPage testPage = pageData.getWikiPage();
        StringBuffer newPageContent = new StringBuffer();
        includeSetupPages (testPage, newPageContent, isSuite);
        newPageContent.append(pageData.getContent());
        includeTeardownPages(testPage, newPageContent, isSuite);
        pageData.setContent(newPageContent.toString());
    }

    return pageData.getHtml();
}
```

A menos que sea un alumno de FitNesse, seguramente no entienda los detalles. Entenderá que la función se encarga de añadir páginas de configuración y detalles en una página de prueba, que después muestra en HTML. Si está familiarizado con JUnit[11], verá que esta función pertenece a algún tipo de estructura de pruebas basada en la Web y, evidentemente, es correcto. Resulta sencillo adivinar esta información del Listado 3-2 pero no del Listado 3-1. ¿Qué tiene la función del Listado 3-2 para que resulte sencilla de leer y entender? ¿Qué hay que hacer para que una función transmita su intención? ¿Qué atributos podemos asignar a nuestras funciones para que el lector pueda intuir el tipo de programa al que pertenecen?

Tamaño reducido

La primera regla de las funciones es que deben ser de tamaño reducido. La segunda es que *deben ser todavía más reducidas*. No es una afirmación que pueda justificar. No puedo mostrar referencias a estudios que demuestren

que las funciones muy reducidas sean mejores. Lo que sí puedo afirmar es que durante casi cuatro décadas he creado funciones de diferentes tamaños. He creado monstruos de casi 3000 líneas y otras muchas funciones de entre 100 y 300 líneas. También he creado funciones de 20 a 30 líneas de longitud. Esta experiencia me ha demostrado, mediante ensayo y error, que las funciones deben ser muy reducidas.

En la década de 1980 se decía que una función no debía superar el tamaño de una pantalla. Por aquel entonces, las pantallas VT100 tenían 24 líneas por 80 columnas, y nuestros editores usaban 4 líneas para tareas administrativas. En la actualidad, con una fuente mínima y un monitor de gran tamaño, se pueden encajar 150 caracteres por línea y 100 líneas o más en una pantalla. Las líneas no deben tener 150 caracteres. Las funciones no deben tener 100 líneas de longitud. Las funciones deben tener una longitud aproximada de 20 líneas.

¿Qué tamaño mínimo debe tener una función? En 1999 visité a Kent Beck en su casa de Oregon. Nos sentamos y comenzamos a programar. Me enseñó un atractivo programa de Java/Swing que había llamado *Sparkle*. Generaba un efecto visual en pantalla, similar a la varita mágica del hada de Cenicienta. Al mover el ratón, salían estrellitas del cursor, y descendían a la parte inferior de la pantalla en un campo gravitatorio simulado. Cuando Kent me enseñó el código, me sorprendió la brevedad de las funciones. Estaba acostumbrado a ver programas de Swing con funciones que ocupaban kilómetros de espacio vertical. En este programa, las funciones tenían dos, tres o cuatro líneas de longitud. Todas eran obvias. Todas contaban una historia y cada una llevaba a la siguiente en un orden atractivo. ¡Así de breves deberían ser todas las funciones![12]

¿Qué tamaño mínimo deben tener sus funciones? Deberían ser más breves que las del Listado 3-2. De hecho, el Listado 3-2 debería reducirse como el Listado 3-3.

Listado 3-3
HtmlUtil.java (nueva refactorización).

```
public static String renderPageWithSetupsAndTeardowns(
    PageData pageData, boolean isSuite) throws Exception {
    if (isTestPage(pageData))
        includeSetupAndTeardownPages(pageData, isSuite);
    return pageData.getHtml();
}
```

Bloques y sangrado

Esto implica que los bloques en instrucciones `if`, `else`, `while` y similares deben tener una línea de longitud que, seguramente, sea la invocación de una función. De esta forma, no sólo se reduce el tamaño de la función, sino que también se añade valor documental ya que la función invocada desde el bloque puede tener un nombre descriptivo. También implica que las funciones no deben tener un tamaño excesivo que albergue estructuras anidadas. Por tanto, el nivel de sangrado de una función no debe ser mayor de uno o dos. Evidentemente, de esta forma las funciones son más fáciles de leer y entender.

Hacer una cosa

Es evidente que el Listado 3-1 hace más de una cosa. Crea búferes, obtiene páginas, busca páginas heredadas, añade cadenas antiguas y genera HTML. El Listado 3-1 está muy ocupado realizando varias tareas. Por su parte, el Listado 3-3 sólo hace una cosa: incluye configuraciones y detalles en páginas de prueba.

El siguiente consejo lleva vigente, de una u otra forma, durante más de 30 años:

> ***LAS FUNCIONES SÓLO DEBEN HACER UNA COSA. DEBEN HACERLO BIEN Y DEBE SER LO ÚNICO QUE HAGAN.***

El problema de esta afirmación es saber qué es una cosa. ¿El Listado 3-3 hace una cosa? Se podría pensar que hace tres:

1. Determinar si la página es una página de prueba.
2. En caso afirmativo, incluir configuraciones y detalles.
3. Representar la página en HTML.

¿Cuál será de las tres? ¿La función hace una o tres cosas? Los tres pasos de la función se encuentran un nivel de abstracción por debajo del nombre de la función. Podemos describir la función como un breve párrafo TO (PARA)[13]:

> *Para* `renderPageWithSetupsAndTeardowns`, *comprobamos si la página es de prueba y, en caso afirmativo, incluimos las configuraciones y los detalles. En ambos casos, la representamos en HTML.*

Si una función sólo realiza los pasos situados un nivel por debajo del nombre de la función, entonces hace una cosa. En definitiva, creamos funciones para descomponer conceptos más amplios (es decir, el nombre de la función) en un conjunto de pasos en el siguiente nivel de abstracción. Es evidente que el Listado 3-1 contiene pasos en distintos niveles de abstracción, por lo que es obvio que hace más de una cosa. Incluso el Listado 3-2 tiene tres niveles de abstracción, como ha demostrado la capacidad de reducirlo, pero sería complicado reducir con sentido el Listado 3-3. Podríamos extraer la instrucción `if` en la función `includeSetupsAndTeardownsIfTestPage`, pero sólo reduciríamos el código sin cambiar el nivel de abstracción.

Por ello, otra forma de saber que una función hace más de una cosa es extraer otra función de la misma con un nombre que no sea una reducción de su implementación [G34].

Secciones en funciones

Fíjese en el Listado 4-7. Verá que la función `generatePrimes` se divide en secciones como declaraciones, inicializaciones y filtros. Es un síntoma evidente de que hace más de una cosa. Las funciones que hacen una sola cosa no se pueden dividir en secciones.

Un nivel de abstracción por función

Para que las funciones realicen «una cosa», asegúrese de que las instrucciones de la función se encuentran en el mismo nivel de abstracción. El Listado 3-1 incumple esta regla. Incluye conceptos a un elevado nivel de abstracción, como `getHtml();` otros se encuentran en un nivel intermedio, como `StringpagePathName = PathParser.render(pagePath)` y hay otros en un nivel especialmente bajo, como `.append("\n").`

La mezcla de niveles de abstracción en una función siempre resulta confusa. Los lectores no sabrán si una determinada expresión es un concepto esencial o un detalle. Peor todavía, si se mezclan detalles con conceptos esenciales, aumentarán los detalles dentro de la función.

Leer código de arriba a abajo: la regla descendente

El objetivo es que el código se lea como un texto de arriba a abajo[14]. Queremos que tras todas las funciones aparezcan las del siguiente nivel de abstracción para poder leer el programa, descendiendo un nivel de abstracción por vez mientras leemos la lista de funciones. Es lo que denomino la regla descendente.

Para decirlo de otra forma, queremos leer el programa como si fuera un conjunto de párrafos `TO`, en el que cada uno describe el nivel actual de abstracción y hace referencia a los párrafos `TO` posteriores en el siguiente nivel.

> *Para incluir configuraciones y detalles, incluimos configuraciones, después del contenido de la página de prueba, y por último los detalles.*
> *Para incluir las configuraciones, incluimos la configuración de suite si se trata de una suite, y después la configuración convencional.*

Para incluir la configuración de suite; buscamos la jerarquía principal de la página `SuiteSetUp` y añadimos una instrucción `include` con la ruta de dicha página.
Para buscar la jerarquía principal…

A los programadores les resulta complicado aprender esta regla y crear funciones en un único nivel de abstracción, pero es un truco importante. Es la clave para reducir la longitud de las funciones y garantizar que sólo hagan una cosa. Al conseguir que el código se lea de arriba a abajo, se mantiene la coherencia de los niveles de abstracción.

Fíjese en el Listado 3-7 del final del capítulo. Muestra la función `testableHtml` modificada de acuerdo a estos principios. Cada función presenta a la siguiente y se mantiene en un nivel de abstracción coherente.

Instrucciones Switch

Es complicado usar una instrucción `switch` de tamaño reducido[15]. Aunque sólo tenga dos casos, es mayor de lo que un bloque o función debería ser. También es complicado crear una instrucción `switch` que haga una sola cosa. Por su naturaleza, las instrucciones `switch` siempre hacen N cosas. Desafortunadamente, no siempre podemos evitar las instrucciones `switch` pero podemos asegurarnos de incluirlas en una clase de nivel inferior y de no repetirlas. Para ello, evidentemente, recurrimos al polimorfismo.

Fíjese en el Listado 3-4. Muestra una de las operaciones que pueden depender del tipo de empleado.

Listado 3-4
Payroll.java.

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType (
    switch (e.type) {
        case COMMISSIONED:
            return calculateCommissionedPay(e);
        case HOURLY:
            return calculateHourlyPay(e);
        case SALARIED:
            return calculateSalariedPay(e);
        default:
            throw new InvalidEmployeeType(e.type);
    }
}
```

Esta función tiene varios problemas. Por un lado, es de gran tamaño y cuando se añadan nuevos tipos de empleado, aumentará más. Por otra parte, hace más de una cosa. También incumple el Principio de responsabilidad única (*Single Responsibility Principie* o SRP)[16] ya que hay más de un motivo para cambiarla. Además, incumple el Principio de abierto/cerrado (*Open Closed Principle* u OCP)[17], ya que debe cambiar cuando se añadan nuevos tipos, pero posiblemente el peor de los problemas es que hay un número ilimitado de funciones que tienen la misma estructura.

Por ejemplo, podríamos tener:

```
isPayday(Employee e, Date date),
```

o

```
deliverPay(Employee e, Date date),
```

o muchas otras, todas con la misma estructura.

La solución al problema (véase el Listado 3-5) consiste en ocultar la instrucción `switch` en una factoría abstracta[18] e impedir que nadie la vea. La factoría usa la instrucción `switch` para crear las instancias adecuadas de los derivados de Employee y las distintas funciones, como `calculatePay`, `isPayday` y `deliverPay`, se entregarán de forma polimórfica a través de la interfaz `Employee`.

Listado 3-5
Employee y Factory.

```
public abstract class Employee {
    public abstract boolean isPayday();
    public abstract Money calculatePay();
    public abstract void deliverPay(Money pay);
}
```

```
public interface EmployeeFactory {
    public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;
}
```

```
public class EmployeeFactoryImpl implements EmployeeFactory {
    public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType {
        switch (r.type) {
            case COMMISSIONED:
                return new CommissionedEmployee(r);
            case HOURLY:
                return new HourlyEmployee(r);
            case SALARIED:
                return new SalariedEmployee(r);
            default:
                throw new InvalidEmployeeType(r.type);
        }
    }
}
```

Mi regla general para las instrucciones `switch` es que se pueden tolerar si sólo aparecen una vez, se usan para crear objetos polimórficos y se

ocultan tras una relación de herencia para que el resto del sistema no las pueda ver [G23]. Evidentemente, cada caso es diferente y en ocasiones se puede incumplir una o varias partes de esta regla.

Usar nombres descriptivos

En el Listado 3-7, hemos cambiado el nombre de la función de ejemplo de `testableHtml` a `SetupTeardownIncluder.render`. Es un nombre más apropiado ya que describe mejor el cometido de la función. También hemos asignado a los métodos privados un nombre descriptivo como `isTestable` o `includeSetupAndTeardownPages`. No hay que olvidar el valor de los nombres correctos. Recuerde el principio de Ward: «Sabemos que trabajamos con código limpio cuando cada rutina es más o menos lo que esperábamos». Para alcanzar este principio, gran parte del esfuerzo se basa en seleccionar nombres adecuados para pequeñas funciones que hacen una cosa. Cuanto más reducida y concreta sea una función, más sencillo será elegir un nombre descriptivo. No tema los nombres extensos. Un nombre descriptivo extenso es mucho mejor que uno breve pero enigmático. Use una convención de nombres que permita leer varias palabras en los nombres de las funciones y use esas palabras para asignar a la función un nombre que describa su cometido.

No tema dedicar tiempo a elegir un buen nombre. De hecho, debería probar con varios nombres y leer el código con todos ellos. Los IDE modernos como Eclipse o IntelliJ facilitan el cambio de nombres. Use uno de estos IDE y experimente con diferentes nombres hasta que encuentre uno que sea lo bastante descriptivo.

La elección de nombres descriptivos clarifica el diseño de los módulos y le permite mejorarlos. No es extraño que la búsqueda de nombres adecuados genere una reestructuración favorable del código. Sea coherente con los nombres. Use las mismas frases, sustantivos y verbos en los nombres de función que elija para los módulos. Pruebe, por ejemplo, con `includeSetupAndTeardownPages`, `includeSetupPages`, `includeSuiteSetupPage` e `includeSetupPage`. La estructura similar de estos nombres permite que la secuencia cuente una historia. En realidad, si ve la secuencia anterior, seguramente se pregunte qué ha pasado con

`includeTeardownPages`, `includeSuiteTeardownPage` e `includeTeardownPage`.

Argumentos de funciones

El número ideal de argumentos para una función es cero. Después uno (monádico) y dos (diádico). Siempre que sea posible, evite la presencia de tres argumentos (triádico). Más de tres argumentos (poliádico) requiere una justificación especial y no es muy habitual.

Los argumentos son complejos ya que requieren un gran poder conceptual. Por ello suelo evitarlos en los ejemplos. Fíjese en `StringBuffer`. Podríamos haberlo pasado como argumento en lugar de como variable de instancia, pero los lectores habrían tenido que interpretarlo cada vez que lo vieran. Al leer la historia que cuenta el módulo, `includeSetupPage()` es más sencillo de interpretar que `includeSetupPageInto(newPageContent)`. El argumento se encuentra en un nivel de abstracción diferente que el nombre de la función y nos obliga a conocer un detalle (`StringBuffer`) que no es especialmente importante en ese momento.

Los argumentos son todavía más complicados desde un punto de vista de pruebas. Imagine la dificultad de crear todos los casos de prueba para

garantizar el funcionamiento de las distintas combinaciones de argumentos. Si no hay argumentos, todo es más sencillo. Si hay uno, no es demasiado difícil. Con dos argumentos el problema es más complejo. Con más de dos argumentos, probar cada combinación de valores adecuados es todo un reto. Los argumentos de salida son más difíciles de entender que los de entrada. Al leer una función, estamos acostumbrados al concepto de información añadida a la función a través de argumentos y extraída a través de un valor devuelto. No esperamos que la información se devuelva a través de los argumentos. Por ello, los argumentos de salida suelen obligamos a realizar una comprobación doble.

Un argumento de salida es la mejor opción, después de la ausencia de argumentos. `SetupTeardownIncluder.render(pageData)` se entiende bien. Evidentemente, vamos a representar los datos en el objeto `pageData`.

Formas monádicas habituales

Hay dos motivos principales para pasar un solo argumento a una función. Puede que realice una pregunta sobre el argumento, como en `boolean fileExists("MyFile")`, o que procese el argumento, lo transforme en otra cosa y lo devuelva. Por ejemplo, `InputStream fileOpen("MyFile")` transforma un nombre de archivo `String` en un valor devuelto `InputStream`. Los usuarios esperan estos dos usos cuando ven una función. Debe elegir nombres que realicen la distinción con claridad y usar siempre ambas formas en un contexto coherente (consulte el apartado sobre separación de consultas de comandos).

Una forma menos habitual pero muy útil para un argumento es un evento. En esta forma, hay argumento de entrada pero no de salida. El programa debe interpretar la invocación de la función como evento y usar el argumento para alterar el estado del sistema, por ejemplo, `void passwordAttemptFailedNtimes(int attempts)`. Use esta forma con precaución. Debe ser claro para el lector que se trata de un evento. Elija nombres y contextos con atención. Intente evitar funciones monádicas que no tengan estas formas, por ejemplo, `void includeSetupPageInto(StringBuffer pageText)`. El uso de un argumento de salida en lugar de un valor devuelto para realizar transformaciones resulta confuso. Si una función va a transformar su argumento de entrada, la transformación debe aparecer como valor

devuelto. Sin duda `StringBuffertransform(StringBuffer in)` es mejor que `void transform(StringBuffer out)`, aunque la implementación del primer caso devuelva solamente el argumento de entrada. Al menos se ajusta a la forma de la transformación.

Argumentos de indicador

Los argumentos de indicador son horribles. Pasar un valor Booleano a una función es una práctica totalmente desaconsejable. Complica inmediatamente la firma del método e indica que la función hace más de una cosa. Hace algo si el indicador es `true` y otra cosa diferente si es `false`. En el Listado 3-7 no se puede evitar, porque los invocadores ya pasan el indicador y el objetivo era limitar el ámbito a la función y después, pero la invocación de `render (true)` es confusa para el lector. Si se desplaza el ratón sobre la invocación vemos que `render (boolean isSuite)` puede ayudar, pero no demasiado. Tendremos que dividir la función en dos: `renderForSuite()` y `renderForSingleTest()`.

Funciones diádicas

Una función con dos argumentos es más difícil de entender que una función monádica. Por ejemplo `writeField(name)` es más fácil de entender que `writeField (outputStream, name)`[19]. Aunque en ambos casos el significado es evidente, la primera se capta mejor visualmente. La segunda requiere una breve pausa hasta que ignoramos el segundo parámetro, lo que en última instancia genera problemas ya que no debemos ignorar esa parte del código. Las partes que ignoramos son las que esconden los errores. Pero en ocasiones se necesitan dos argumentos. Por ejemplo. `Point p = new Point(0,0);` es totalmente razonable. Los puntos cartesianos suelen adoptar dos argumentos. De hecho, sería muy sorprendente ver `Point(0)`. Sin embargo, en este caso ambos argumentos son componentes ordenados de un mismo valor, mientras que `outputStream` y `name` carecen de una cohesión o un orden natural.

Incluso funciones diádicas evidentes como `assertEquals(expected, actual)` resultan problemáticas. ¿Cuántas veces ha incluido el valor real en su posición esperada? Los dos argumentos carecen de un orden natural. El

orden real y esperado es una convención que se adquiere gracias a la práctica.

Las combinaciones diádicas no son el mal en persona y tendrá que usarlas. Sin embargo, recuerde que tienen un precio y que debe aprovechar los mecanismos disponibles para convertirlas en unitarias. Por ejemplo, puede hacer que el método `writeField` sea un miembro de `outputStream` para poder usar `outputStream.writeField(name)`, o podría convertir `outputStream` en una variable miembro de la clase actual para no tener que pasarla. Incluso podría extraer una nueva clase como `FieldWriter` que usara `outputStream` en su constructor y tuviera un método `write`.

Triadas

Las funciones que aceptan tres argumentos son sin duda mucho más difíciles de entender que las de dos. Los problemas a la hora de ordenar, ignorar o detenerse en los argumentos se duplican. Piense atentamente antes de crear una triada.

Por ejemplo, fíjese en la sobrecarga de `assertEquals` que acepta tres argumentos: `assertEquals(message, expected, actual)`. ¿Cuántas veces lee el mensaje y piensa que es lo esperado? He visto esta triada en concreto muchas veces. De hecho, siempre que la veo, tengo que repasarla antes de ignorar el mensaje.

Por otra parte, hay otra triada que no es tan negativa: `assertEquals(1.0, amount, .001)`. Aunque también exija doble atención, merece la pena. Conviene recordar siempre que la igualdad de los valores de coma flotante es algo relativo.

Objeto de argumento

Cuando una función parece necesitar dos o más argumentos, es probable que alguno de ellos se incluya en una clase propia. Fíjese en la diferencia entre las dos siguientes declaraciones:

```
Circle makeCircle (double x, double y, double radius);
Circle makeCircle(Point center, double radius);
```

La reducción del número de argumentos mediante la creación de objetos puede parecer una trampa pero no lo es. Cuando se pasan grupos de

variables de forma conjunta, como x e y en el ejemplo anterior, es probable que formen parte de un concepto que se merece un nombre propio.

Listas de argumentos

En ocasiones tendremos que pasar un número variable de argumentos a una función. Fíjese en el método `String.format`:

```
String.format (“%s worked %.2f hours.”, name, hours);
```

Si los argumentos variables se procesan de la misma forma, como en el ejemplo anterior, serán equivalentes a un único argumento de tipo `List`. Por tanto, `String.format` es en realidad diádico. De hecho, la siguiente declaración de `String.format` es claramente diádica.

```
public String format(String format, Object… args)
```

Así pues, se aplican las mismas reglas. Las funciones que aceptan argumentos variables pueden ser monádicas, diádicas o incluso triádicas, pero sería un error asignar más argumentos.

```
void monad(Integer… args);
void dyad(String name, Integer… args);
void triad(String name, int count, Integer… args);
```

Verbos y palabras clave

La selección de nombres correctos para una función mejora la explicación de su cometido, así como el orden y el cometido de los argumentos. En formato monádico, la función y el argumento deben formar un par de verbo y sustantivo. Por ejemplo, `write(name)` resulta muy evocador. Sea lo que sea `name`, sin duda se escribe `(write)`.

Un nombre más acertado podría ser `writeField(name)`, que nos dice que `name` es un campo `(field)`. Éste es un ejemplo de palabra clave como nombre de función. Con este formato codificamos los nombres de los argumentos en el nombre de la función. Por ejemplo, `assertEquals` se podría haber escrito como `assertExpectedEqualsActual(expected, actual)`, lo que mitiga el problema de tener que recordar el orden de los argumentos.

Sin efectos secundarios

Los efectos secundarios son mentiras. Su función promete hacer una cosa, pero también hace otras cosas ocultas. En ocasiones realiza cambios inesperados en las variables de su propia clase. En ocasiones las convierte en las variables pasadas a la función o a elementos globales del sistema. En cualquier caso, se comete un engaño que suele provocar extrañas combinaciones temporales y dependencias de orden.

Fíjese en la función del Listado 3-6, aparentemente inofensiva. Usa un algoritmo estándar para comparar `userName` con `password`. Devuelve `true` si coinciden y `false` si hay algún problema, pero también hay un efecto secundario. ¿Lo detecta?

Listado 3-6
UserValidator.java.

```
public class UserValidator {
    private Cryptographer cryptographer;


    public boolean checkPassword(String userName, String password) {
        User user = UserGateway.findByName(userName);
        if (user != User.NULL) {
            String codedPhrase = user.getPhraseEncodedByPassword();
            String phrase = cryptographer.decrypt(codedPhrase, password);
            if (“Valid Password”.equals(phrase)){
                Session.initialize();
                return true;
            }
        }
        return false;
    }
}
```

El efecto secundario es la invocación de `Session.initialize()`. La función `checkPassword`, por su nombre, afirma comprobar la contraseña. El nombre no implica que inicialice la sesión. Por tanto, un invocador que se crea lo que dice el nombre de la función se arriesga a borrar los datos de sesión actuales cuando decida comprobar la validez del usuario. Este efecto secundario genera una combinación temporal. Es decir, sólo se puede invocar `checkPassword` en determinados momentos (cuando se pueda inicializar la sesión). Si no se invoca en orden, se pueden perder los datos de la sesión. Las combinaciones temporales son confusas, en especial cuando se ocultan como efecto secundario. Si tiene que realizar una combinación temporal, hágalo de forma clara en el nombre de la función. En este caso, podríamos cambiar el nombre de la función por `checkPasswordAndInitializeSession`, pero incumpliría la norma de hacer una sola cosa.

Argumentos de salida

Los argumentos suelen interpretarse como entradas de una función. Si lleva varios años programando, estoy seguro de que habrá visto un argumento que en vez de ser de entrada era de salida. Por ejemplo;

```
appendFooter(s);
```

¿Está función añade `s` al final de algo? ¿O añade el final de algo a `s`? ¿`s` es una entrada o una salida? Lo sabemos al ver la firma de la función:

```
public void appendFooter(StringBuffer report)
```

Esto lo aclara todo, pero para ello hay que comprobar la declaración de la función. Todo lo que le obligue a comprobar la firma de la función es un esfuerzo doble. Es una pausa cognitiva y debe evitarse.

Antes de la programación orientada a objetos, era necesario tener argumentos de salida. Sin embargo, gran parte de su necesidad desaparece en los lenguajes orientados a objetos, pensados para actuar como argumento de salida. Es decir, sería más indicado invocar `appendFooter` como `report.appendFooter();`.

Por lo general, los argumentos de salida deben evitarse. Si su función tiene que cambiar el estado de un elemento, haga que cambie el estado de su objeto contenedor.

Separación de consultas de comando

Las funciones deben hacer algo o responder a algo, pero no ambas cosas. Su función debe cambiar el estado de un objeto o devolver información sobre el mismo, pero ambas operaciones causan confusión. Fíjese en la siguiente función:

```
public boolean set(String attribute, String value);
```

Esta función establece el valor de un atributo y devuelve `true` en caso de éxito o `false` si el atributo no existe. Esto provoca la presencia de una extraña instrucción como la siguiente:

```
if (set(“username”, “unclebob”))…
```

Imagínelo desde el punto de vista del lector. ¿Qué significa? ¿Pregunta si el atributo «`username`» se ha establecido antes en «`unclebob`», o si el atributo «username» se ha establecido correctamente en «`unclebob`»? Es

complicado saberlo por la invocación ya que no es evidente si set es un verbo o un adjetivo.

El autor pretendía que `set` fuera un verbo, pero el contexto de la instrucción `if` parece un adjetivo. La instrucción se lee como «si el atributo `username` se ha establecido previamente en `unclebob`», no como «establecer el atributo `username` en `unclebob` y si funciona, entonces…». Podríamos solucionarlo si cambiamos el nombre de la función `set` por `setAndCheckIfExists`, pero no mejoraría la legibilidad de la instrucción `if`. La verdadera solución es separar el comando de la consulta para evitar la ambigüedad.

```
if (attributeExists(“username”)) {
    setAttribute(“username”, “unclebob”);
    …
}
```

Mejor excepciones que devolver códigos de error

Devolver códigos de error de funciones de comando es un sutil incumplimiento de la separación de comandos de consulta. Hace que los comandos usados asciendan a expresiones en los predicados de las instrucciones `if`.

```
if (deletePage(page) == E_OK)
```

No padece la confusión entre verbo y adjetivo, pero genera estructuras anidadas. Al devolver un código de error se crea un problema: el invocador debe procesar el error de forma inmediata.

```
if (deletePage(page) == E_OK) {
    if (registry.deleteReference(page.name) == E_OK) {
        if (configKeys.deleteKey(page.name.makeKey()) == E_OK) {
            logger.log(“page deleted”);
        } else {
            logger.log(“configKey not deleted”);
        }
    } else {
        logger.log(“deleteReference from registry failed”);
    }
} else {
    logger.log(“delete failed”);
    return E_ERROR;
}
```

Por otra parte, si usa excepciones en lugar de códigos de error, el código de procesamiento del error se puede separar del código de ruta y se puede simplificar:

```
try {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
    logger.log(e.getMessage());
}
```

Extraer bloques Try/Catch

Los bloques `try/catch` no son atractivos por naturaleza. Confunden la estructura del código y mezclan el procesamiento de errores con el normal. Por ello, conviene extraer el cuerpo de los bloques `try` y `catch` en funciones individuales.

```
public void delete(Page page) {
    try {
        deletePageAndAllReferences(page);
    }
    catch (Exception e) {
        logError(e);
    }
}

private void deletePageAndAllReferences(Page page) throws Exception {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}

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

En este caso, la función `delete` es de procesamiento de errores. Es fácil de entender e ignorar. La función `deletePageAndAllReferences` es para los procesos de borrar una página. El procesamiento de errores se puede ignorar. De este modo, la separación facilita la comprensión y la modificación del código.

El procesamiento de errores es una cosa

Las funciones sólo deben hacer una cosa y el procesamiento de errores es un ejemplo. Por tanto, una función que procese errores no debe hacer nada más. Esto implica (como en el ejemplo anterior) que, si una función incluye la palabra clave `try`, debe ser la primera de la función y que no debe haber nada más después de los bloques `catch/finally`.

El imán de dependencias Error.java

La devolución de códigos de error suele implicar que existe una clase o enumeración en la que se definen los códigos de error.

```
public enum Error {
    OK,
    INVALID,
    NO_SUCH,
    LOCKED,
    OUT_OF_RESOURCES,
    WAITING_FOR_EVENT;
}
```

Clases como ésta son un *imán para las dependencias*; otras muchas clases deben importarlas y usarlas. Por ello, cuando cambia la enumeración `Error`, es necesario volver a compilar e implementar dichas clases[20]. Esto añade presión a la clase `Error`. Los programadores no quieren añadir nuevos errores porque tendrán que volver a generar e implementarlo todo. Por ello, reutilizan códigos de error antiguos en lugar de añadir otros nuevos.

Al usar excepciones en lugar de códigos de error, las nuevas excepciones son derivaciones de la clase de la excepción. Se pueden añadir sin necesidad de volver a compilar o implementar[21].

No repetirse[22]

Fíjese de nuevo en el Listado 3-1; verá que hay un algoritmo que se repite cuatro veces, en los casos `SetUp`, `SuiteSetUp`, `TearDown` y `SuiteTearDown`. No es fácil detectar esta repetición ya que las cuatro instancias se mezclan con otro código, pero la duplicación es un problema ya que aumenta el tamaño del código y requerirá una modificación cuádruple si alguna vez cambia el algoritmo.

También se cuadriplica el riesgo de errores.

Esta duplicación se remedia gracias al método `include` del Listado 3-7. Vuelva a leer el código y fíjese en cómo se ha mejorado la legibilidad del código reduciendo la duplicación.

La duplicación puede ser la raíz de todos los problemas del *software*. Existen numerosos principios y prácticas para controlarla o eliminarla. Imagine que todas las formas normales de la base de datos de Codd sirvieran para eliminar la duplicación de datos. Imagine también cómo la

programación orientada a objetos concentra el código en clases base que en otros casos serian redundantes. La programación estructurada, la programación orientada a aspecto y la orientada a componentes son, en parte, estrategias para eliminar duplicados. Parece que, desde la aparición de las subrutinas, las innovaciones en desarrollo de *software* han sido un intento continuado por eliminar la duplicación de nuestro código fuente.

Programación estructurada

Algunos programadores siguen las reglas de programación estructurada de Edsger Dijkstra[23]. Dijkstra afirma que todas las funciones y todos los bloques de una función deben tener una entrada y una salida. Estas reglas implican que sólo debe haber una instrucción `return` en una función, que no debe haber instrucciones `break` o `continue` en un bucle y nunca, bajo ningún concepto, debe haber instrucciones `goto`.

Aunque apreciemos los objetivos y disciplinas de la programación estructurada, no sirven de mucho cuando las funciones son de reducido tamaño. Su verdadero beneficio se aprecia en funciones de gran tamaño.

Por tanto, si sus funciones son de tamaño reducido, una instrucción `return`, `break` o `continue` no hará daño alguno y en ocasiones puede resultar más expresiva que la regla de una entrada y una salida. Por otra parte, `goto` sólo tiene sentido en funciones de gran tamaño y debe evitarse.

Cómo crear este tipo de funciones

La creación de *software* es como cualquier otro proceso creativo. Al escribir un informe o un artículo, primero se estructuran las ideas y después el mensaje hasta que se lea bien. El primer borrador puede estar desorganizado, de modo que lo retoca y mejora hasta que se lea de la forma adecuada.

Cuando creo funciones, suelen ser extensas y complicadas, con abundancia de sangrados y bucles anidados. Con extensas listas de argumentos, nombres arbitrarios y código duplicado, pero también cuento con una serie de pruebas de unidad que abarcan todas y cada una de las líneas de código.

Por tanto, retoco el código, divido las funciones, cambio los nombres y elimino los duplicados. Reduzco los métodos y los reordeno. En ocasiones, elimino clases enteras, mientras mantengo las pruebas.

Al final, consigo funciones que cumplen las reglas detalladas en este capítulo. No las escribo al comenzar y dudo que nadie pueda hacerlo.

Conclusión

Todo sistema se crea a partir de un lenguaje específico del dominio diseñado por los programadores para describir dicho sistema. Las funciones son los verbos del lenguaje y las clases los sustantivos. No es volver a la noción de que los sustantivos y verbos de un documento de requisitos son las clases y funciones de un sistema. Es una verdad mucho más antigua. El arte de la programación es, y ha sido siempre, el arte del diseño del lenguaje.

Los programadores experimentados piensan en los sistemas como en historias que contar, no como en programas que escribir. Recurren a las prestaciones del lenguaje de programación seleccionado para crear un lenguaje expresivo mejor y más completo que poder usar para contar esa historia. Parte de ese lenguaje es la jerarquía de funciones que describen las acciones que se pueden realizar en el sistema. Dichas acciones se crean para usar el lenguaje de dominio concreto que definen para contar su pequeña parte de la historia.

En este capítulo hemos visto la mecánica de la creación de funciones correctas. Si aplica estas reglas, sus funciones serán breves, con nombres correctos, y bien organizadas, pero no olvide que su verdadero objetivo es contar la historia del sistema y que las funciones que escriba deben encajar en un lenguaje claro y preciso que le sirva para contar esa historia.

SetupTeardownIncluder

Listado 3-7
SetupTeardownIncluder.java.

```
package fitnesse.html;
```

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


public class SetupTeardownIncluder {
    private PageData pageData;
    private boolean isSuite;
    private WikiPage testPage;
    private StringBuffer newPageContent;
    private PageCrawler pageCrawler;


    public static String render(PageData pageData) throws Exception {
        return render(pageData, false);
    }


    public static String render(PageData pageData, boolean isSuite)
        throws Exception {
        return new SetupTeardownIncluder(pageData).render(isSuite);
    }


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


    private String render(boolean isSuite) throws Exception {
        this.isSuite = isSuite;
        if (isTestPage())
            includeSetupAndTeardownPages();
        return pageData.getHtml();
    }


    private boolean isTestPage() throws Exception {
        return pageData.hasAttribute(“Test”);
    }


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


    private void includeSetupPages() throws Exception {
        if (isSuite)
            includeSuiteSetupPage();
        includeSetupPage();
    }


    private void includeSuiteSetupPage() throws Exception {
        include(SuiteResponder.SUITE_SETUP_NAME, “-setup”);
    }


    private void includeSetupPage() throws Exception {
        include(“SetUp”, “-setup”);
    }


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


    private void includeTeardownPages() throws Exception {
        includeTeardownPage();
        if (isSuite)
            includeSuiteTeardownPage();
    }


    private void includeTeardownPage() throws Exception {
        include(“TearDown”, “-teardown”);
    }


    private void includeSuiteTeardownPage() throws Exception {
        include(SuiteResponder.SUITE_TEARDOWN_NAME, “-teardown”);
    }


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


    private void include(String pageName, String arg) throws Exception (
        WikiPage inheritedPage = findInheritedPage(pageName);
        if (inheritedPage != null) {
```

```
            String pagePathName = getPathNameForPage(inheritedPage);
            buildIncludeDirective(pagePathName, arg);
        }
    }

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

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

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

Bibliografía

- **[KP78]**: Kernighan and Plaugher, *The Elements of Programming Style*, 2d. ed., McGraw-Hill, 1978.
- **[PPP02]**: Robert C. Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall, 2002.
- **[GOF]**: Design Patterns: Elements of Reusable Object Oriented Software, Gamma et al., Addison Wesley, 1996.
- **[PRAG]**: *The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.
- **[SP72]**: *Structured Programming*, O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Academic Press, London, 1972.

4
Comentarios

«No comente el código incorrecto, reescríbalo».
Brian W. Kernighan y P. J. Plaugher[24]

No hay nada más útil que un comentario bien colocado. No hay nada que colapse más un módulo que comentarios dogmáticos innecesarios. No hay nada más dañino que un comentario antiguo que propague mentiras y desinformación.

Los comentarios no son como la Lista de Schindler. No son pura bondad. De hecho, en el mejor de los casos, son un mal necesario. Si los lenguajes de programación fueran más expresivos o si pudiéramos dominarlos para expresar nuestras intenciones, no necesitaríamos demasiados comentarios, puede que incluso ninguno.

El uso correcto de los comentarios permite compensar nuestra incapacidad para expresarnos en el código. He usado la palabra

incapacidad, a propósito. Los comentarios siempre son fallos. Debemos usarlos porque no siempre sabemos cómo expresarnos sin ellos pero su uso no es motivo de celebración.

Cuando tenga que escribir un comentario, piense si no existe otra forma de expresarse en el código. Siempre que se exprese en el código, debe felicitarse. Siempre que escriba un comentario, debe hacer un gesto de desaprobación y sentir su incapacidad para expresarse.

¿Por qué estoy en contra de los comentarios? Porque mienten. No siempre y no siempre intencionadamente, pero lo hacen. Cuando más antiguo es un comentario y más se aleja del código que describe, mayor es la probabilidad de que sea equivocado. El motivo es sencillo. Los programadores no los pueden mantener.

El código cambia y evoluciona. Los fragmentos cambian de lugar, se bifurcan, se reproducen y se vuelven a combinar para crear quimeras. Desafortunadamente, los comentarios no siempre siguen el ritmo, no siempre pueden hacerlo y suelen separarse del código que describen y se convierten en huérfanos sin precisión alguna. Por ejemplo, fíjese en lo que sucede con este comentario y la línea que pretendía describir:

```
MockRequest request;
private final String HTTP_DATE_REGEXP =
    "[SMTWF][a-z]{2}\\,\\s[0-9]{2}\\s[JFMASOND][a-z]{2}\\s" +
    "[0-9]{4}\\s[0-9]{2}\\:[0-9]{2}\\:[0-9]{2}\\sGMT";
private Response response;
private FitNesseContext context;
private FileResponder responder;
private Locale saveLocale;
// Ejemplo: «Tue, 02 Apr 2003 22:18:49 GMT»
```

Seguramente se añadieron después otras variables de instancia entre la constante `HTTP_DATE_REGEXP` y su comentario explicativo.

Se podría afirmar que los programadores deben ser lo bastante disciplinados como para mantener los comentarios actualizados, relevantes y precisos. De acuerdo, debería, pero esa energía debería invertirse en crear código claro y expresivo que no necesite comentario alguno.

Los comentarios imprecisos son mucho peor que la ausencia de comentarios. Suelen confundir al usuario. Generan expectativas que nunca se cumplen. Definen reglas que no deben seguirse en absoluto.

La verdad sólo se encuentra en un punto: el código. Sólo el código puede contar lo que hace. Es la única fuente de información precisa. Por tanto, aunque los comentarios sean necesarios en ocasiones, dedicaremos nuestra energía a minimizarlos.

Los comentarios no compensan el código incorrecto

Una de las principales motivaciones para crear comentarios es el código incorrecto. Creamos un módulo y sabemos que es confuso y está desorganizado. Sabemos que es un desastre y entonces decidimos comentarlo. Error. Mejor límpielo.

El código claro y expresivo sin apenas comentarios es muy superior al código enrevesado y complejo con multitud de comentarios. En lugar de perder tiempo escribiendo comentarios que expliquen el desastre cometido, dedíquelo a solucionarlo.

Explicarse en el código

En ocasiones, el código es un pobre vehículo de expresión. Desafortunadamente, muchos programadores lo entienden como que el código no es un buen medio de expresión. Esto es falso. ¿Qué prefiere ver? Esto:

```
// Comprobar si el empleado tiene derecho a todos los beneficios
if ((employee.flags & HOURLY_FLAG) &&
    (employee.age > 65))
```

o esto otro:

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

Apenas se tardan unos segundos en explicar nuestras intenciones en el código. En muchos casos, basta con crear una función que diga lo mismo que el comentario que pensaba escribir.

Comentarios de calidad

Algunos comentarios son necesarios o beneficiosos. Veremos algunos de los que considero válidos. No obstante, recuerde que el único comentario realmente bueno es el que no tiene que escribir.

Comentarios legales

En ocasiones, nuestros estándares corporativos de creación de código nos obligan a crear determinados comentarios por motivos legales. Por ejemplo, los comentarios de derechos de autor son necesarios y deben incluirse al inicio de cada archivo.

El siguiente encabezado de comentario se incluye de forma estándar al inicio de todos los archivos fuente de FitNesse. Nuestro IDE evita que este comentario parezca sobrante replegándolo de forma automática.

```
// Copyright (C) 2003,2004,2005 de Object Mentor, Inc. Todos los derechos reservados.
// Publicado bajo las condiciones de la Licencia pública general GNU versión 2 o posterior.
```

Este tipo de comentarios no deben ser contratos ni tomos legales. Siempre que sea posible, haga referencia a una licencia estándar o a otro documento externo en lugar de incluir todos los términos y condiciones en el comentario.

Comentarios informativos

En ocasiones es útil proporcionar información básica con un comentario. Por ejemplo, el siguiente comentario explica el valor devuelto por un método abstracto:

```
// Devuelve una instancia del elemento Responder probado.
protected abstract Responder responderInstance();
```

Estos comentarios pueden ser útiles, pero es mejor usar el nombre de la función para transmitir la información siempre que sea posible. Por ejemplo, en este caso el comentario sería redundante si cambiamos el nombre de la función por responderBeingTested. Veamos un ejemplo mejor:

```
// el formato coincide con kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile(
    "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

En este caso, el comentario nos indica que la expresión regular debe coincidir con una fecha y una hora con el formato aplicado por la función `SimpleDateFormat.format` con la cadena de formato especificada.

Hubiera resultado mejor y más claro si el código se hubiera cambiado a una clase especial que convirtiera los formatos de fechas y horas. De ese modo el comentario habría sido superfluo.

Explicar la intención

En ocasiones, un comentario es algo más que información útil sobre la implementación y proporciona la intención de una decisión. En el siguiente caso, vemos una interesante decisión documentada por un comentario. Al comparar dos objetos, el autor decidió ordenar los objetos de su clase por encima de los objetos de otra.

```
public int compareTo(Object o)
{
    if (o instanceof WikiPagePath)
    {
        WikiPagePath p = (WikiPagePath) o;
        String compressedName = StringUtil.join(names, “”);
        String compressedArgumentName = StringUtil.join(p.names, “”);
        return compressedName.compareTo(compressedArgumentName);
    }
    return 1; // somos mayores porque somos el tipo correcto.
}
```

Veamos otro ejemplo mejor. Puede que no esté de acuerdo con la solución del programador, pero al menos sabe lo que intentaba hacer.

```
public void testConcurrentAddWidgets() throws Exception {
    WidgetBuilder widgetBuilder =
        new WidgetBuilder(new Class[](BoldWidget.class));
    String text = “’’’bold text’’’”;
    ParentWidget parent =
        new BoldWidget(new MockWidgetRoot(), “’’’bold text’’’”);
    AtomicBoolean failFlag = new AtomicBoolean();
    failFlag.set(false);


    //Nuestro mejor intento de obtener una condición de carrera
    //creando un gran número de procesos.
    for (int i = 0; i < 25000; i++) {
        WidgetBuilderThread widgetBuiIderThread =
            new WidgetBuilderThread(WidgetBuilder, text, parent, failFlag);
        Thread thread = new Thread(WidgetBuilderThread);
        thread.start();
    }
    assertEquals(false, failFlag.get());
}
```

Clarificación

En ocasiones, basta con traducir el significado de un argumento o valor devuelto en algo más legible. Por lo general, conviene buscar la forma de que el argumento o el valor devuelto sean claros por sí mismos; pero cuando forma parte de una biblioteca estándar o de código que no se puede alterar, un comentario aclarativo puede ser muy útil.

```
public void testCompareTo() throws Exception
{
    WikiPagePath a = PathParser.parse(“PageA”);
    WikiPagePath ab = PathParser.parse(“PageA.PageB”);
    WikiPagePath b = PathParser.parse(“PageB”);
    WikiPagePath aa = PathParser.parse(“PageA.PageA”);
    WikiPagePath bb = PathParser.parse(“PageB.PageB”);
    WikiPagePath ba = PathParser.parse(“PageB.PageA”);


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

Pero también existe el riesgo de que un comentario aclarativo sea incorrecto. En el ejemplo anterior, compruebe lo difícil que resulta comprobar si los comentarios son correctos. Esto explica por qué la clarificación es necesaria y también arriesgada. Por ello, antes de escribir estos comentarios, asegúrese de que no hay una solución mejor y también de que sean precisos.

Advertir de las consecuencias

En ocasiones es muy útil advertir a otros programadores de determinadas consecuencias. Por ejemplo, el siguiente comentario explica por qué un determinado caso de prueba está desactivado:

```
// No ejecutar a menos
// que le sobre tiempo.
public void _testWithReallyBigFile()
{
    writeLinesToFile(10000000);


    response.setBody(testFile);
    response.readyToSend(this);
    String responseString = output.toString();
    assertSubString("Content-Length: 1000000000", responseString);
    assertTrue(bytesSent > 1000000000);
}
```

En la actualidad, evidentemente, desactivaríamos la prueba por medio del atributo `@Ignore` con la correspondiente cadena explicativa: `@Ignore("Takes too long to run")`, pero antes de la aparición de JUnit 4, era habitual añadir un guion bajo delante del nombre del método. El comentario realizaba su cometido. Veamos otro ejemplo:

```
public static SimpleDateFormat makeStandardHttpDateFormat()
{
    //SimpleDataFormat no es compatible con procesos,
    //por lo que debe crear cada instancia de forma independiente.
    SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss z");
    df.setTimeZone (TimeZone.getTimeZone ("GMT"));
    return df;
}
```

Seguramente conozca soluciones mejores para este problema. Estoy de acuerdo, pero el comentario es perfectamente razonable. Evita que un programador use un inicializador estático por motivos de eficacia.

Comentarios TODO

En ocasiones conviene usar notas con forma de comentarios `//TODO`. En el siguiente caso, el comentario `TODO` explica por qué la función tiene una implementación incorrecta y cuál debe ser su futuro.

```
// TODO-MdM no son necesarios
// Esperamos que desaparezca en el modelo definitivo
protected VersionInfo makeVersion() throws Exception
{
    return null;
}
```

`TODO` son tareas que el programador piensa que debería haber hecho pero que no es así. Pueden ser un recordatorio para eliminar una función obsoleta o una petición para resolver un problema.

Pueden ser una solicitud para buscar un nombre más adecuado o para realizar un cambio que dependa de un evento planeado. Sea lo que sea, no es excusa para mantener código incorrecto en el sistema.

En la actualidad, muchos IDE cuentan con funciones especiales para localizar comentarios `TODO`, por lo que seguramente no se pierda. Sin embargo, no colapse el código con estos comentarios. Examínelos y elimine todos los que pueda.

Amplificación

Se puede usar un comentario para amplificar la importancia de algo que, en caso contrario, parecería irrelevante.

```
String listItemContent = match.group(3).trim();
// el recorte es importante. Elimina los espacios iniciales
// que harían que el elemento se reconociera como
// otra lista.
new ListItemWidget(this, listItemContent, this.level + 1);
return buildList(text.substring(match.end()));
```

Javadoc en API públicas

No hay nada más útil y satisfactorio que una API pública bien descrita. Los javadoc de la biblioteca estándar de Java son un ejemplo. Sería muy complicado crear programas de Java sin ellos.

Si usa una API pública, debe crear javadoc de calidad para la misma, pero recuerde el siguiente consejo a lo largo del capítulo: los javadoc pueden ser tan ambiguos, amplios y descorteses como cualquier otro tipo de documento.

Comentarios incorrectos

Muchos comentarios pertenecen a esta categoría. Suelen ser excusas de código pobre o justificaciones de decisiones insuficientes, algo así como si el programador se hablara a sí mismo.

Balbucear

Añadir un comentario sin razón o porque el proceso lo requiere es un error. Si decide escribir un comentario, tómese el tiempo necesario para asegurarse de que sea el mejor que puede redactar.

El siguiente ejemplo es de FitNesse, donde un comentario sin duda sería de utilidad, pero el autor tenía prisa o no prestó demasiada atención. Su balbuceo generó un enigma:

```
public void loadProperties()
{
    try
    {
        String propertiesPath = propertiesLocation + “/” + PROPERTIES_FILE;
        FileInputStream propertiesStream = new FileInputStream(propertiesPath);
        loadedProperties.load(propertiesStream);
    }
    catch(IOException e)
    {
        // Si no hay archivos de propiedades significan que cargan las predeterminadas
    }
}
```

¿Qué significa el comentario del bloque `catch`? Seguro que algo para el autor, pero el significado no está claro. Aparentemente, si se genera `IOException`, significa que no hay archivo de propiedades y, en ese caso, se cargan los valores predeterminados. ¿Pero quién carga los valores predeterminados? ¿Se cargan antes de la invocación de `loadProperties.load` o `loadProperties.load` captura la excepción, carga los valores predeterminados y después nos pasa la excepción para que la ignoremos? ¿O será que `loadProperties.load` carga todos los valores predeterminados antes de intentar abrir el archivo? ¿Intentaba el autor consolarse por dejar el bloque catch vacío? Ésta es la posibilidad más

temida, ¿se estaba diciendo que volviera más tarde para crear el código para cargar los valores predeterminados?

Nuestro único recurso es examinar el código en otras partes del sistema para determinar qué sucede. Cualquier comentario que le obligue a buscar su significado en otro módulo ha fallado en su intento de comunicación y no merece los bits que consume.

Comentarios redundantes

El Listado 4-1 muestra una sencilla función con un comentario de encabezado totalmente redundante. Seguramente se tarde más en leer que el propio código.

Listado 4-1
waitForClose.

```
// Método de utilidad devuelto cuando this.closed es true. Genera una excepción
// si se alcanza el tiempo de espera.
public synchronized void waitForClose(final long timeoutMillis)
throws Exception
{
    if (!closed)
    {
        wait(timeoutMillis);
        if(!closed)
            throw new Exception (“MockResponseSender could not be closed”);
    }
}
```

¿Para qué sirve este comentario? No es más informativo que el código. No lo justifica ni transmite la intención ni la lógica. No es más fácil de leer que el código. De hecho, es menos preciso y obliga al lector a aceptar la falta de precisión en lugar de a entenderlo. Es como un vendedor de coches de segunda menos que le asegura que no hace falta revisar el motor.

Fíjese ahora en la legión de javadoc inútiles y redundantes del Listado 4-2, obtenido de Tomcat. Estos comentarios únicamente ensucian y oscurecen el código. No tienen ninguna función documental. Para empeorar las cosas, sólo le mostramos algunos. El módulo tiene muchos más.

Listado 4-2
ContainerBase.java (Tomcat).

```
public abstract class ContainerBase
    implements Container, Lifecycle, Pipeline,
    MBeanRegistration, Serializable {


    /**
    * Retardo del procesador para este componente.
    */
```

```
protected int backgroundProcessorDelay = -1;

/**
* Compatibilidad con eventos de ciclo vital de este componente.
*/
protected LifecycleSupport lifecycle =
    new LifecycleSupport(this);

/**
* Escuchadores de eventos de contenedor de este contenedor.
*/
protected ArrayList listeners = new ArrayList();

/**
* Implementación Loader a la que se asocia este contenedor.
*/
protected Loader loader = null;

/**
* Implementación Logger a la que se asocia este contenedor.
*/
protected Log logger = null;

/**
* Nombre de registrador asociado.
*/
protected String logName = null;

/**
* Implementación Manager a la que se asocia este contenedor.
*/
protected Manager manager = null;

/**
* Clúster al que se asocia este contenedor.
*/
protected Cluster cluster = null;

/**
* Nombre legible de este contenedor.
*/
protected String name = null;

/**
* Contenedor principal de este contenedor.
*/
protected Container parent = null;

/**
* Cargador de clase principal que configurar al instalar un elemento
* Loader.
*/
protected ClassLoader parentClassLoader = null;

/**
* Objeto Pipeline al que se asocia este contenedor.
*/
protected Pipeline pipeline = new StandardPipeline(this);

/**
* Objeto Realm al que se asocia este contenedor.
*/
protected Realm realm = null;

/**
* Objeto DirContext de recursos al que se asocia este contenedor.
*/
protected DirContext resources = null;
```

Comentarios confusos

En ocasiones, a pesar de las buenas intenciones, un programador realiza una afirmación en sus comentarios que no es del todo precisa. Fíjese otra vez en el comentario redundante y confuso del Listado 4-1.

¿Sabe por qué es confuso? El método no devuelve nada cuando `this.closed` se convierte en `true`. Devuelve algo si `this.closed` es `true`; en caso contrario, espera y genera una excepción si `this.closed` no es `true`.

Este sutil fragmento, oculto en un comentario más difícil de leer que el cuerpo del código, puede hacer que otro programador invoque la función con la esperanza de que devuelva algo cuando `this.closed` sea `true`. Ese pobre programador se encontrará en una sesión de depuración intentando determinar por qué el código se ejecuta tan lentamente.

Comentarios obligatorios

Es una locura tener una regla que afirme que todas las funciones deben tener un javadoc o que todas las variables deben tener un comentario. Este tipo de comentarios ensucian el código y generan confusión y desorganización. Por ejemplo, los javadoc obligatorios para todas las funciones crean abominaciones como el Listado 4-3. No sirven de nada, complican el código y constituyen posibles engaños y desorientaciones.

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(String title, String author,
                int tracks, int durationInMinutes) {
        CD cd = new CD();
        cd.title = title;
        cd.author = author;
        cd.tracks = tracks;
        cd.duration = duration;
        cdList.add(cd);
    }
```

Comentarios periódicos

En ocasiones, se añade un comentario al inicio de un módulo cada vez que se edita. Estos comentarios acumulan una especie de registro de todos los cambios realizados. He visto módulos con decenas de páginas con estas entradas.

```
* Cambios (11-Oct-2001)
* ----------------------------
* 11-Oct-2001   :   Reorganización de la clase y cambio a un nuevo paquete
```

```
*                  com.jrefinery.date (DG);
* 05-Nov-2001   :   Se añade un método getDescription() y se elimina la clase NotableDate (DG);
* 12-Nov-2001   :   IBD requiere el método setDescription(), una vez eliminada la clase NotableDate
*                  (DG); se cambian getPreviousDayOfWeek(), getFollowingDayOfWeek()
*                  y getNearestDayOfWeek() para corregir errores (DG);
* 05-Dic-2001   :   Error corregido en la clase SpreadsheetDate (DG);
* 29-May-2002   :   Se transfieren todas las constantes de mes a una interfaz
*                  independiente (MonthConstants) (DG);
* 27-Ago-2002   :   Error corregido en el método addMonths(), gracias a Nálevka Petr (DG);
* 03-Oct-2002   :   Errores indicados por Checkstyle (DG) corregidos;
* 13-Mar-2003   :   Implementación de Serializable (DG);
* 29-May-2003   :   Error corregido en el método addMonths (DG);
* 04-Sep-2003   :   Implementación de Comparable. Actualización de los javadoc isInRange (DG);
* 05-Ene-2005   :   Error corregido en el método addYears() (1096202) (DG);
```

Hace tiempo hubo una buena razón para crear y mantener estas entradas de registro al inicio de cada módulo. Carecíamos de sistemas de control de código fuente que se encargaran de ello, pero en la actualidad, estas entradas son elementos sobrantes que complican los módulos. Debe eliminarlas totalmente.

Comentarios sobrantes

En ocasiones vemos comentarios que simplemente sobran. Restan importancia a lo evidente y no ofrecen información nueva.

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

¿En serio? ¿Y este otro?:

```
/** Día del mes. */
private int dayOfMonth;
```

Y aquí el parangón de la redundancia:

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

Estos comentarios son tan inservibles que aprendemos a ignorarlos. Al leer el código, la vista los salta. Con el tiempo, los comentarios empiezan a mentir cuando cambia el código que les rodea.

El primer comentario del Listado 4-4 parece correcto[25]. Explica por qué se ignora el bloque `catch`, pero el segundo comentario sobra. Parece que el programador estaba tan frustrado con crear bloques `try/catch` en la función que necesitaba explotar.

Listado 4-4
startSending.

```
private void startSending()
{
    try
    {
        doSending();
    }
    catch(SocketException e)
    {
        // normal, alguien ha detenido la solicitud.
    }
    catch(Exception e)
    {
        try
        {
            response.add(ErrorResponder.makeExceptionString(e));
            response.closeAll();
        }
        catch(Exception e1)
        {
            //¡Un respiro!
        }
    }
}
```

En lugar de explotar en un comentario sin sentido, el programador debería haber sabido que su frustración se podría aliviar mejorando la estructura del código. Tendría que haber centrado su energía en extraer el último bloque `try/catch` en una función independiente, como muestra el Listado 4-5.

Listado 4-5
startSending (refactorizado).

```
private void startSending()
{
    try
    {
        doSending();
    }
    catch(SocketException e)
    {
        // normal. Alguien ha detenido la solicitud.
    }
    catch(Exception e)
    {
        addExceptionAndCloseResponse(e);
    }
}

private void addExceptionAndCloseResponse(Exception e)
{
    try
    {
        response.add(ErrorResponder.makeExceptionString(e));
        response.closeAll();
    }
    catch(Exception e1)
    {
    }
}
```

Cambie la tentación de crear elementos sobrantes por la determinación de limpiar su código. Mejorará como programador y será más fácil.

Comentarios sobrantes espeluznantes

Los javadoc también pueden ser innecesarios. ¿Para qué sirven los siguientes javadoc (de una conocida biblioteca) de código abierto? La respuesta: para nada. Son comentarios redundantes creados en un intento equivocado de redactar documentación.

```
/** El nombre. */
private String name;

/** La versión. */
private String version;

/** El licenceName. */
private String licenceName;

/** La versión. */
private String info;
```

Vuelva a leer los comentarios. ¿Detecta el error de corta y pega? Si los autores no prestan atención al escribir sus comentarios (o al pegarlos), ¿por qué se espera que sean de utilidad para los lectores?

No usar comentarios si se puede usar una función o una variable

Fíjese en el siguiente código:

```
// ¿el módulo de la lista global <mod> depende del
// subsistema del que formamos parte?
if (smodule.getDependSubsystems().contains(subSysMod.getSubsystem()))
```

Se podría cambiar sin el comentario de esta forma:

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

El autor del código original seguramente escribió primero el comentario (improbable) y después el código para ajustarlo al comentario. Sin embargo, el autor tendría que haber refactorizado el código, como hice yo, para poder eliminar el comentario.

Marcadores de posición

En ocasiones los programadores marcan una determinada posición en un archivo. Por ejemplo, recientemente encontré esto en un programa:

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

Son escasas las ocasiones en las que tiene sentido agrupar funciones bajo esta estructura. Por lo general, debe eliminarse, sobre todo la molesta hilera de barras al final.

Piénselo de esta forma. Estas estructuras son atractivas si no las usa demasiado. Por ello, úselas esporádicamente y sólo cuando el beneficio sea significativo. Si las usa en exceso, acabarán por ser ignoradas.

Comentarios de llave de cierre

En ocasiones, los programadores incluyen comentarios especiales en llaves de cierre, como en el Listado 4-6. Aunque pueda tener sentido en funciones extensas con estructuras anidadas, únicamente estorba a las funciones encapsuladas y de pequeño tamaño que nos gustan. Por ello, si siente el deseo de marcar sus llaves de cierre, pruebe a reducir el tamaño de sus funciones.

Listado 4-6
wc.java.

```
public class wc {
    public static void main(String[] args) {
        BufferedReader in = new BufferedReader(new InputstreamReader(System.in));
        String line;
        int lineCount = 0;
        int charCount = 0;
        int wordCount = 0;
        try {
            while ((line = in.readLine()) != null) {
                lineCount++;
                charCount += line.length();
                String words[] = line.split("\\W");
                wordCount += words.length;
            } //while
            System.out.println("wordCount = " + wordCount);
            System.out.println("lineCount = " + lineCount);
            System.out.println("charCount = " + charCount);
        } // try
        catch (IOException e) {
            System.err.println("Error: " + e.getMessage());
        } //catch
    } //main
}
```

Asignaciones y menciones

```
/* Añadido por Rick */
```

Los sistemas de control de código fuente recuerdan a la perfección quién ha añadido qué y cuándo. No es necesario plagar el código con pequeñas menciones. Puede pensar que estos comentarios son útiles y que ayudan a otros a hablar sobre el código, pero en realidad sobreviven durante años y cada vez son menos precisos y relevantes. El sistema de control de código fuente es el punto idóneo para este tipo de información.

Código comentado

No hay nada más odioso que el código comentado. ¡No lo haga!

```
InputStreamResponse response = new inputStreamResponse();
response.setBody(formatter.getResultStream(), formatter.getByteCount());
// InputStream resultsStream = formatter.getResultStream();
// StreamReader reader = new StreamReader(resultsStream);
// response.setContent(reader.read(formatter.getByteCount()));
```

Los lectores que vean código comentado no tendrán el valor de borrarlo. Pensarán que está ahí por algo y que es demasiado importante para borrarlo. Por ello, el código comentado se acumula como los sedimentos en una botella de vino malo.

Fíjese en este fragmento de apache commons:

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

¿Por qué hay dos líneas comentadas? ¿Son importantes? ¿Se han conservado como recordatorio de un cambio inminente o es algo que alguien comentó hace años y no se ha preocupado de limpiar? Hubo una época, en la década de 1960, en la que el código comentado pudo ser útil, pero hace tiempo que contamos con buenos sistemas de control de código fuente, sistemas que recuerdan el código por nosotros. Ya no tenemos que comentarlo. Elimínelo. No lo perderá. Se lo aseguro.

Comentarios HTML

El HTML en comentarios de código fuente es una aberración, como puede apreciar en el siguiente fragmento. Dificulta la lectura de los comentarios donde debería ser más fácil; el editor o IDE. Si los comentarios se van a extraer con una herramienta (como Javadoc) para mostrarlos en una página Web, debe ser responsabilidad de dicha herramienta y no del programador el adornar los comentarios con el correspondiente HTML.

```
/**
*     Tarea para ejecutar pruebas de aceptación.
*     Esta tarea ejecuta pruebas de aceptación y publica los resultados.
*     <p/>
*     <pre>
*     Uso:
*     &lt;taskdef name=&quot;execute-fitnesse-tests&quot;
*     classname=&quot;fitnesse.ant.ExecuteFitnesseTestsTask&quot;
```

```
*     classpathref=&quot;classpath&quot; /&gt;
*     OR
*     &lt;taskdef classpathref=&quot;classpath&quot;
*     resource=&quot;tasks.properties&quot; /&gt;
*     <p/>
*     &lt;execute-fitnesse-tests
*     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 tiene que escribir un comentario, asegúrese de que describa el código que le rodea. No ofrezca información global del sistema en el contexto de un comentario local. Fíjese en el siguiente comentario javadoc. Aparte de su terrible redundancia, también ofrece información sobre el puerto predeterminado y la función no tiene control alguno sobre el puerto predeterminado. El comentario no describe la función sino otra parte distinta del sistema. Evidentemente, no hay garantías de que el comentario cambie cuando lo haga el código que contiene el valor predeterminado.

```
/**
* Puerto para ejecutar fitnesse. El predeterminado es <b>8082</b>.
*
* @param fitnessePort
*/
public void setFitnessePort(int fitnessePort)
{
    this.fitnessePort = fitnessePort;
}
```

Demasiada información

No incluya en sus comentarios interesantes reflexiones históricas ni irrelevantes descripciones de detalles. El siguiente comentario se ha extraído de un módulo diseñado para probar que una función puede codificar y descodificar base64. Aparte del número RFC, el lector de este código no necesita la información obsoleta que contiene el comentario.

```
/*
    RFC 2045 - Extensiones Multipropósito de correo de Internet (MIME)
    Primera parte: Formato del Cuerpo de los Mensajes de Internet
    sección 6.8. Codificación de transferencia de contenidos Base64
    El proceso de codificación representa grupos de 24 bits de la entrada
    como cadenas de salida de 4 caracteres codificados. Procediendo de
    izquierda a derecha, se forma un grupo de 24 bits de entrada
    concatenando 3 grupos de 8 bits de entrada. Estos 24 bits se tratan
    como 4 grupos concatenados de 6 bits, cada uno de los cuales se
    traduce en un solo dígito del alfabeto base64. Cuando se codifica un
    flujo de bits mediante la codificación base64, el flujo de bits se
    debe considerar ordenado con el bit más significativo primero. Esto
    es, el primer bit del flujo será el bit de orden más alto en el
    primer byte de 8 bits, y el octavo bit será el de orden más bajo en
    el primer byte de 8 bits, y así sucesivamente.
```

```
*/
```

Conexiones no evidentes

La conexión entre un comentario y el código que describe debe ser evidente. Si se ha preocupado de escribir un comentario, lo mínimo es que el lector que lo vea entienda a qué se refiere. Fíjese en este comentario obtenido de apache commons:

```
/*
* comienza con una matriz de tamaño suficiente para albergar todos los pixeles
* (más bytes de filtro), y 200 bytes adicionales para la información de encabezado
*/
this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200];
```

¿Qué es un *byte* de filtro? ¿Está relacionado con `+1`? ¿O con `*3`? ¿Con ambos? ¿Es un pixel un *byte*? ¿Por qué 200? La función de un comentario es explicar código que no se explica por sí mismo. Es una lástima que un comentario requiera su propia explicación.

Encabezados de función

Las funciones breves apenas requieren explicación. Un nombre bien elegido para una función que hace una cosa suele ser mejor que un encabezado de comentario.

Javadocs en código no público

A pesar de la utilidad de los javadoc para las API públicas, no sirven para código no dirigido a consumo público. La generación de páginas javadoc para clases y funciones de un sistema no suele ser útil y la formalidad adicional de los comentarios javadoc no es más que una distracción.

Ejemplo

Escribí el módulo del Listado 4-7 para la primera versión de *XP Immersion*. Debía ser un ejemplo de estilo incorrecto de creación de código y comentarios. Después, Kent Beck refactorizó este código en algo mucho más atractivo delante de varios alumnos. Posteriormente, adapté el ejemplo para mi libro *Agile Software Development, Principles, Patterns, and*

Practices y para el primero de mis artículos *Craftsman* publicados en la revista *Software Development*.

Lo que me fascina de este módulo es que hubo un tiempo en que muchos lo hubiéramos considerado bien documentado. Ahora vemos que es un auténtico desastre. A ver cuántos problemas detecta en los comentarios.

Listado 4-7
GeneratePrimes.java.

```
/**
* Esta clase genera números primos hasta la cantidad máxima especificada por el
* usuario. El algoritmo usado es la Criba de Eratóstenes.
* <p>
* Eratóstenes de Cirene, 276 a. C., Cirene, Libia -
* 194 a. C., Alejandría. El primer hombre en calcular la
* circunferencia de la Tierra. También trabajó con calendarios
* con años bisiestos y fue responsable de la Biblioteca de Alejandría.
* <p>
* El algoritmo es muy simple. Dada una matriz de enteros
* empezando por el 2, se tachan todos los múltiplos de 2. Se busca el siguiente
* entero sin tachar y se tachan todos sus múltiplos.
* Repetir hasta superar la raíz cuadrada del valor
* máximo.
*
* ©author Alphonse
* ©version 13 Feb 2002 atp
*/
import java.util.*;


public class 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
            boolean[] f = new boolean[s];
            int i;


            // inicializar la matriz en true.
            for (i = 0; i < s; i++)
                f[i] = true;


            // eliminar los números no primos conocidos
            f[0] = f[1] = false;


            // cribar
            int j;
            for (i = 2; i < Math.sqrt(s) + 1; i++)
            {
                if (f[i]) // si no está tachado, tachar sus múltiplos.
                {
                for (j = 2 * i; j < s; j += i)
                    f[j] = false; // el múltiplo no es primo
                }
            }


            // ¿cuántos primos hay?
            int count = 0;
            for (i = 0; i < s; i++)
            {
                if (f[i])
                count++; // contador.
            }


            int[] primes = new int[count];


            // enviar primos al resultado
            for (i = 0, j = 0; i < s; i++)
            {
                if (f[i]) // si es primo
                primes[j++] = i;
```

```
            }

            return primes; // devolver los primos
        }
        else // maxValue < 2
            return new int[0]; // devolver matriz null si la entrada no es correcta.
    }
}
```

En el Listado 4-8 puede ver una versión refactorizada del mismo módulo. Se ha limitado considerablemente el uso de comentarios. Hay sólo dos en todo el módulo y ambos claramente descriptivos.

Listado 4-8
PrimeGenerator.java (refactorizado).

```
/**
* Esta clase genera números primos hasta la cantidad máxima especificada por el
* usuario. El algoritmo usado es la Criba de Eratóstenes. Dada una matriz de enteros
* empezando por el 2: buscar el primer entero sin tachar y tachar todos sus
* múltiplos. Repetir hasta que no haya más múltiplos en la matriz.
*/
public class PrimeGenerator
{
    private static boolean[] crossedOut;
    private static int[] result;


    public static int[] generatePrimes(int maxValue)
    {
        if (maxValue < 2)
            return new int[0];
        else
        {
            uncrossIntegersUpTo(maxValue);
            crossOutMultiples();
            putUncrossedIntegersIntoResult();
            return result;
        }
    }


    private static void uncrossIntegersUpTo(int maxValue)
    {
        crossedOut = new boolean[maxValue + 1];
        for (int i = 2; i < crossedOut.length; i++)
            crossedOut[i] = false;
    }


    private static void crossOutMultiples()
    {
        int limit = determineIterationLimit();
        for (int i = 2; i <= limit; i++)
            if (notCrossed(i))
                crossOutMultiplesOf(i);
    }


    private static int determineIterationLimit()
    {
        // Cada múltiplo en la matriz tiene un factor primordial que
        // es menor o igual que la raíz del tamaño de la matriz,
        // entonces no tenemos que tachar múltiplos de números
        // más grande que esa raíz.
        double iterationLimit = Math.sqrt(crossedOut.length);
        return (int) iterationLimit;
    }


    private static void crossOutMultiplesOf (int i)
    {
        for (int multiple = 2 * i;
                multiple < crossedOut.length;
                multiple += i)
            crossedOut[multiple] = true;
    }


    private static boolean notCrossed(int i)
    {
        return crossedOut[i] == false;
    }


    private static void putUncrossedIntegersIntoResult()
```

```
        {
            result = new int[numberOfUncrossedIntegers()];
            for (int j = 0, i = 2; i < crossedOut.length; i++)
                if (notCrossed(i))
                    result[j++] = i;
        }

        private static int numberOfUncrossedIntegers()
        {
            int count = 0;
            for (int i = 2; i < crossedOut.length; i++)
                if (notCrossed(i))
                    count++;

            return count;
        }
    }
```

Se podría decir que el primer comentario es redundante ya que es muy similar a la función `generatePrimes`, pero creo que muestra mejor el algoritmo al lector, motivo por el que lo he mantenido. El segundo argumento es sin duda necesario. Explica la lógica del uso de la raíz cuadrada como límite del bucle. No encontré otro nombre de variable más sencillo ni otra estructura de código que lo aclarara más. Por otra parte, el uso de la raíz cuadrada podría resultar presuntuoso. ¿Realmente se ahorra tanto tiempo limitando la iteración a la raíz cuadrada? ¿El cálculo de la raíz cuadrada llevaría más tiempo del que se ahorra? Conviene analizarlo. El uso de la raíz cuadrada como límite de iteración satisface al viejo *hacker* de C y de lenguajes de ensamblado de mi interior, pero no estoy convencido de que merezca el tiempo y el esfuerzo que los demás puedan dedicar a entenderlo.

Bibliografía

- **[KP78]**: Kernighan and Plaugher, *The Elements of Programming Style*, 2d. ed., McGraw-Hill, 1978.

5
Formato

Cuando los usuarios miran entre bastidores, queremos que queden impresionados por el atractivo, la coherencia y la atención al detalle que perciben. Queremos que el orden les sorprenda, que abran los ojos con asombro cuando se desplacen por los módulos. Queremos que aprecien que se trata de un trabajo de profesionales. Si ven una masa amorfa de código

que parece escrito por un grupo de marineros borrachos, es probable que piensen que sucederá lo mismo en otros aspectos del proyecto.

Debe preocuparse por el formato de su código. Debe elegir una serie de reglas sencillas que controlen el formato del código y después aplicarlas de forma coherente. Si trabaja en equipo, debe acordar una serie de reglas que todos los miembros deben cumplir. También es muy útil usar una herramienta automatizada que se encargue de aplicar las reglas.

La función del formato

En primer lugar, debe ser claro. El formato de código es importante, demasiado importante como para ignorarlo y también demasiado importante como para tratarlo de forma religiosa. El formato del código se basa en la comunicación y la comunicación debe ser el principal pilar de un desarrollador profesional.

Puede que piense que conseguir que algo funcione es la principal preocupación de un programador profesional. Espero que este libro le haga cambiar de idea. La funcionalidad que cree hoy es muy probable que cambie en la siguiente versión, pero la legibilidad de su código afectará profundamente a todos los cambios que realice. El estilo del código y su legibilidad establecen los precedentes que afectan a la capacidad de mantenimiento y ampliación mucho después de que el código cambie. Su estilo y su disciplina sobrevivirán, aunque el código no lo haga.

Veamos qué aspectos del formato nos permiten comunicarnos mejor.

Formato vertical

Comencemos por el tamaño vertical. ¿Qué tamaño debe tener un archivo fuente? En Java, el tamaño de los archivos está relacionado con el tamaño de las clases, como veremos más adelante. Por el momento, nos detendremos en el tamaño de los archivos.

¿Qué tamaño tienen la mayoría de archivos fuente de Java? Existe una amplia gama de tamaños e importantes diferencias de estilo, como se aprecia en la figura 5.1.

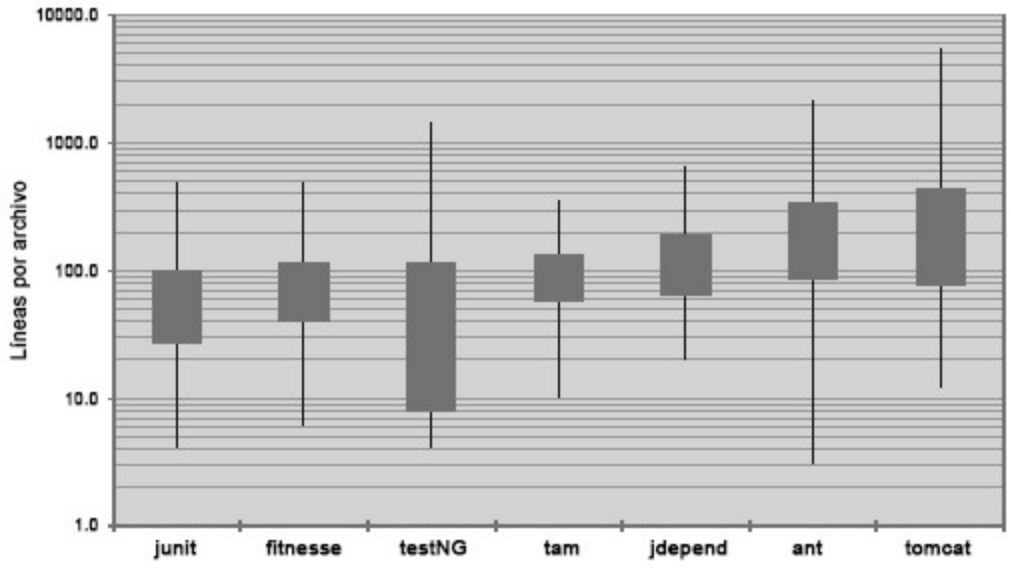


Figura 5.1. Escala LOG de distribuciones de longitud de archivos (altura del cuadro = sigma).

Se describen siete proyectos: Junit, FitNesse, testNG, Time and Money, JDepend, Ant y Tomcat. Las líneas que cruzan los cuadros muestran la longitud máxima y mínima de cada proyecto. El cuadro muestra aproximadamente un tercio (una desviación estándar[26]) de los archivos. La parte central del cuadro es la media. Por tanto, el tamaño de archivo medio del proyecto FitNesse es de 65 líneas y un tercio de los archivos ocupan entre 40 y 100+ líneas.

El mayor archivo de FitNesse tiene unas 400 líneas y el de menor tamaño, 6. Es una escala de registro, de modo que la pequeña diferencia de posición vertical supone una gran diferencia en tamaño absoluto.

Junit, FitNesse y Time and Money tienen archivos relativamente pequeños. Ninguno supera las 500 líneas y la mayoría tienen menos de 200. Tomcat y Ant, por su parte, tienen archivos con varios miles de líneas de longitud y más de la mitad superan las 200.

¿Qué significa todo esto? Aparentemente se pueden crear sistemas (FitNesse se aproxima a las 50 000 líneas) a partir de archivos de unas 200 líneas de longitud, con un límite máximo de 500. Aunque no debería ser una regla, es un intervalo aconsejable. Los archivos de pequeño tamaño se entienden mejor que los grandes.

La metáfora del periódico

Piense en un artículo de periódico bien escrito. En la parte superior espera un titular que indique de qué se trata la historia y le permita determinar si quiere leerlo o no. El primer párrafo ofrece una sinopsis de la historia, oculta los detalles y muestra conceptos generales. Al avanzar la lectura, aumentan los detalles junto con todas las fechas, nombres, citas y otros elementos.

Un archivo de código debe ser como un artículo de periódico. El nombre debe ser sencillo pero claro. Por sí mismo, debe bastar para indicarnos si estamos o no en el módulo correcto. Los elementos superiores del archivo deben proporcionar conceptos y algoritmos de nivel superior. Los detalles deben aumentar según avanzamos, hasta que en la parte final encontremos las funciones de nivel inferior del archivo.

Un periódico se compone de varios artículos, algunos muy reducidos y otros de gran tamaño. No hay muchos que ocupen toda la página con texto, para que el periódico sea manejable. Si el periódico fuera un único y extenso texto con una aglomeración desorganizada de hechos, fechas y nombres, no lo leeríamos.

Apertura vertical entre conceptos

La práctica totalidad del 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. Estos pensamientos deben separarse mediante líneas en blanco.

Fíjese en el Listado 5-1. Hay líneas en blanco que separan la declaración del paquete, las importaciones y las funciones. Es una regla muy sencilla con un profundo efecto en el diseño visual del código. Cada línea en blanco es una pista visual que identifica un nuevo concepto independiente. Al avanzar por el listado, la vista se fija en la primera línea que aparece tras una línea en blanco.

Listado 5-1
BoldWidget.java

```
package fitnesse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
    public static final String REGEXP = “’’’.+?’’’”;
```

```
    private static final Pattern pattern = Pattern.compile (“’’’(.+?)’’’”,
        Pattern.MULTILINE + Pattern.DOTALL
    );

    public BoldWidget(ParentWidget parent, String text) throws Exception {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));
    }

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

Si eliminamos las líneas en blanco, como en el Listado 5-2, se oscurece la legibilidad del código.

Listado 5-2
BoldWidget.java

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
    public static final String REGEXP = “’’’.+?’’’”;
    private static final Pattern pattern = Pattern.compile(“’’’(.+?)’’’”,
        Pattern.MULTILINE + Pattern.DOTALL);
    public BoldWidget(ParentWidget parent, String text) throws Exception {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));}
    public String render() throws Exception {
        StringBuffer html = new StringBuffer(“<b>”);
        html.append(childHtml()).append(“</b>”);
        return html.toString();
    }
}
```

Este efecto aumenta todavía más si no centramos la vista. En el primer ejemplo, los distintos grupos de líneas saltan a la vista, mientras que en el segundo es una mezcla amorfa. La diferencia entre ambos listados es una ligera apertura vertical.

Densidad vertical

Si la apertura separa los conceptos, la densidad vertical implica asociaciones. Por tanto, las líneas de código con una relación directa deben aparecer verticalmente densas. Fíjese en cómo los comentarios sin sentido del Listado 5-3 anulan la asociación entre las dos variables de instancia.

Listado 5-3

```
public class ReporterConfig {
```

```
    /**
    * Nombre de clase del escuchador
    */
    private String m_className;


    /**
    * Propiedades del escuchador
    */
    private List<Property> m_properties = new ArrayList<Property>();


    public void addProperty(Property property) {
        m_properties.add(property);
    }
```

El Listado 5-4 es mucho más fácil de leer. Lo apreciamos a simple vista o al menos yo lo hago. Al mirarlo, veo que es una clase con dos variables y un método, sin tener que mover la cabeza ni la vista. El listado anterior nos obliga a forzar la vista y a mover la cabeza para alcanzar el mismo nivel de comprensión.

Listado 5-4

```
public class ReporterConfig {
    private String m_className;
    private List<Property> m_properties = new ArrayList<Property>();


    public void addProperty(Property property) {
        m_properties.add(property);
    }
```

Distancia vertical

¿Alguna vez ha tenido que recorrer una clase, saltando de una función a otra, desplazándose por el código para intentar adivinar la relación y el funcionamiento de las funciones, y acabar totalmente confundido? ¿Alguna vez ha escudriñado la cadena de herencia buscando la definición de una variable o función? Resulta frustrante porque intenta comprender lo que hace el sistema, pero pierde el tiempo y su energía mental en intentar localizar y recordar sus elementos.

Los conceptos relacionados entre sí deben mantenerse juntos verticalmente [G10]. Esta regla no funciona con conceptos de archivos independientes. Por lo tanto, no debe separar conceptos relacionados en archivos independientes a menos que tenga un motivo de peso. De hecho, es uno de los motivos por los que se debe evitar el uso de variables protegidas.

Para los conceptos relacionados que pertenecen al mismo archivo, su separación vertical debe medir su importancia con respecto a la legibilidad

del otro. Debe evitar que el lector deambule entre archivos y clases.

Declaraciones de variables

Las variables deben declararse de la forma más aproximada a su uso. Como las funciones son muy breves, las variables locales deben aparecer en la parte superior de cada función, como en este ejemplo de Junit4.3.1.

```
private static void readPreferences() {
    InputStream is = null;
    try {
        is = new FileInputStream(getPreferencesFile());
        setPreferences(new Properties(getPreferences()));
        getPreferences().load(is);
    } catch (IOException e) {
        try {
            if (is != null)
                is.close();
        } catch (IOException e1) {
        }
    }
}
```

Las variables de control de bucles deben declararse en la instrucción del bucle, como en esta pequeña función del mismo código fuente:

```
public int countTestCases() {
    int count= 0;
    for (Test each : tests)
        count += each.countTestCases();
    return count;
}
```

En casos excepcionales, una variable puede declararse en la parte superior de un bloque o antes de un bucle en una función extensa. Puede ver este tipo de variable en la siguiente función de TestNG.

```
…
for (XmlTest test: m_suite.getTests()) {
    TestRunner tr = m_runnerFactory.newTestRunner(this, test);
    tr.addListener(m_textReporter);
    m_testRunners.add(tr);


    invoker = tr.getInvoker();


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


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

Variables de instancia

Las variables de instancia, por su parte, deben declararse en la parte superior de la clase. Esto no debe aumentar la distancia vertical de las variables, ya que en una clase bien diseñada se usan en muchos sino en todos sus métodos.

Existen discrepancias sobre la ubicación de las variables de instancia. En C++ suele aplicarse la denominada regla de las tijeras, que sitúa todas las variables de instancia en la parte inferior. En Java, sin embargo, es habitual ubicarlas en la parte superior de la clase. No veo motivos para no hacerlo. Lo importante es declarar las variables de instancia en un punto conocido para que todo el mundo sepa dónde buscarlas.

Fíjese en el extraño caso de la clase `TestSuite` de JUnit 4.3.1. He atenuado considerablemente esta clase para ilustrar este concepto. Si se fija en la mitad del listado, verá dos variables de instancia declaradas. Resultaría complicado ocultarlas en un punto mejor. Cualquiera que lea este código tendría que toparse con las declaraciones por casualidad (como me pasó a mí).

```
public class TestSuite implements Test {
    static public Test createTest(Class<? extends TestCase> theClass,
    String name) {
        …
    }

    public static Constructor<? extends TestCase>
    getTestConstructor(Class<? extends TestCase> theClass)
    throws NoSuchMethodException {
        …
    }

    public static Test warning(final String message) {
        …
    }

    private static String exceptionToString(Throwable t) {
        …
    }

    private String fName;

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

    public TestSuite() {
    }

    public TestSuite(final Class<? extends TestCase> theClass) {
        …
    }

    public TestSuite(Class<? extends TestCase> theClass, String name) {
        …
    }
    … … … …
}
```

Funciones dependientes

Si una función invoca otra, deben estar verticalmente próximas, y la función de invocación debe estar por encima de la invocada siempre que sea posible. De este modo el programa fluye con normalidad. Si la convención se sigue de forma fiable, los lectores sabrán que las definiciones de función

aparecen después de su uso. Fíjese en el fragmento de FitNesse del Listado 5-5.

La función superior invoca las situadas por debajo que, a su vez, invocan a las siguientes. Esto facilita la detección de las funciones invocadas y mejora considerablemente la legibilidad del módulo completo.

Listado 5-5
WikiPageResponder.java.

```
public class WikiPageResponder implements SecureResponder {
    protected WikiPage page;
    protected PageData pageData;
    protected String pageTitle;
    protected Request request;
    protected PageCrawler crawler;


    public Response makeResponse(FitNesseContext context, Request request)
        throws Exception {
        String pageName = getPageNameOrDefault(request, “Frontpage”);
        LoadPage(pageName, context);
        if (page == null)
            return notFoundResponse(context, request);
        else
            return makePageResponse(context);
    }


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


        return pageName;
    }


    protected void loadPage(String resource, FitNesseContext context)
        throws Exception {
        WikiPagePath path = PathParser.parse(resource);
        crawler = context.root.getPageCrawler();
        crawler.setDeadEndStrategy(new VirtualEnabledPageCrawler());
        page = crawler.getPage(context.root, path);
        if (page != null)
            pageData = page.getData();
    }


    private Response notFoundResponse(FitNesseContext context, Request request)
        throws Exception {
        return new NotFoundResponder().makeResponse(context, request);
    }


    private SimpleResponse makePageResponse(FitNesseContext context)
        throws Exception {
        pageTitle = PathParser.render(crawler.getFullPath(page));
        String html = makeHtml(context);


        SimpleResponse response = new SimpleResponse();
        response.setMaxAge(0);
        response.setContent(html);
        return response;
    }
…
```

Además, este fragmento es un buen ejemplo de ubicación de constantes en un nivel correcto [G35]. La constante `FrontPage` se podría haber ocultado en la función `getPageNameOrDefault`, pero eso habría ocultado una constante conocida y esperada en una función de nivel inferior

de forma incorrecta. Es mejor pasar la constante desde un punto en el que tiene sentido a la posición en la que realmente se usa.

Afinidad conceptual

Determinados conceptos de código *deben* estar próximos a otros. Tienen una afinidad conceptual concreta. Cuanto mayor sea esta afinidad, menor distancia vertical debe existir entre ellos.

Como hemos visto, esta afinidad se puede basar en una dependencia directa, como cuando una función invoca a otra, o cuando usa una variable. Pero hay otras causas de afinidad. Puede generarse porque un grupo de funciones realice una operación similar. Fíjese en este fragmento de código de Junit 4.3.1:

```
public class Assert {
    static public void assertTrue(String message, boolean condition) {
        if (!condition)
            fail(message);
    }

    static public void assertTrue(boolean condition) {
        assertTrue (null, condition);
    }

    static public void assertFalse(String message, boolean condition) {
        assertTrue(message, !condition);
    }

    static public void assertFalse(boolean condition) {
        assertFalse(null, condition);
    }
…
```

Estas funciones tienen una elevada afinidad conceptual ya que comparten un sistema de nombres común y realizan variantes de la misma

tarea básica. El hecho de que se invoquen unas a otras es secundario. Aunque no lo hicieran, deberían seguir estando próximas entre ellas.

Orden vertical

Por lo general, las dependencias de invocaciones de funciones deben apuntar hacia abajo. Es decir, la función invocada debe situarse por debajo de la que realice la invocación[27]. Esto genera un agradable flujo en el código fuente, de los niveles superiores a los inferiores.

Como sucede en los artículos del periódico, esperamos que los conceptos más importantes aparezcan antes y que se expresen con la menor cantidad de detalles sobrantes. Esperamos que los detalles de nivel inferior sean los últimos. De este modo, podemos ojear los archivos de código y captar el mensaje en las primeras funciones sin necesidad de sumergirnos en los detalles. El Listado 5-5 se organiza de esta forma. Puede que otros ejemplos mejores sean los listados 15-5 y 3-7.

Formato horizontal

¿Qué ancho debe tener una línea? Para responderlo, fíjese en la anchura de las líneas de un programa convencional. De nuevo, examinamos siete proyectos diferentes. En la figura 5.2 puede ver la distribución de longitud de todos ellos. La regularidad es impresionante, en especial en tomo a los 45 caracteres. De hecho, los tamaños entre 20 y 60 representan un uno por cien del número total de líneas. ¡Eso es un 40 por 100! Puede que otro 30 por 100 sea menos de 10 caracteres de ancho. Recuerde que es una escala de registro, de modo que la apariencia lineal es muy significativa. Es evidente que los programadores prefieren líneas menos anchas.

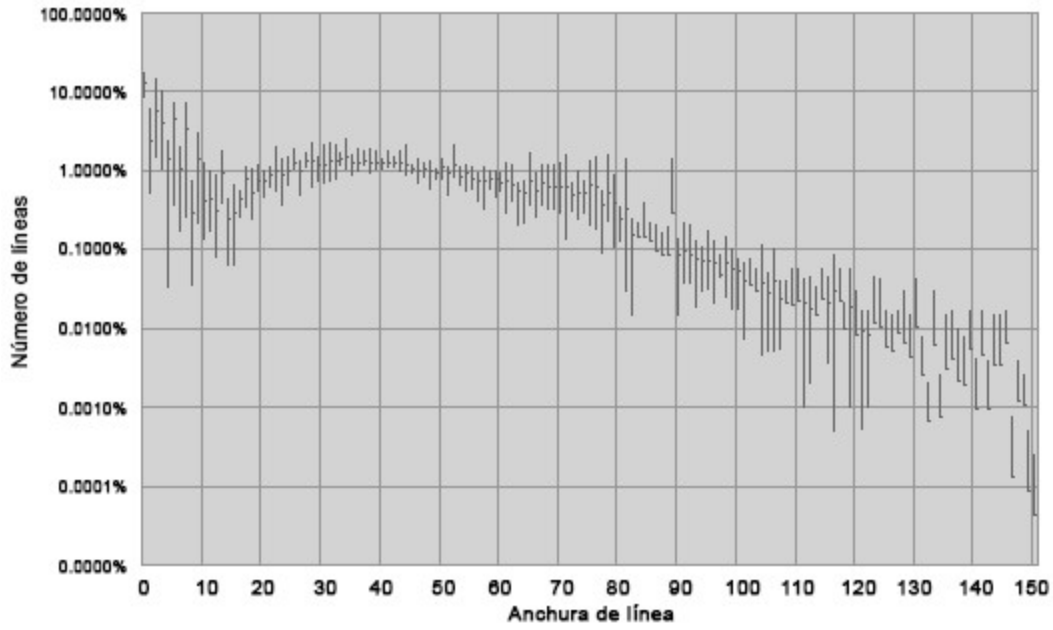


Figura 5.2. Distribución de anchura de líneas en Java.

Esto sugiere que debemos intentar reducir las líneas de código. El antiguo límite Hollerith de 80 es un tanto arbitrario y no me opongo a líneas que tienen 100 o incluso 120, pero no más.

Como norma, no debe tener que desplazarse hacia la derecha. Los monitores modernos son más anchos y los programadores noveles pueden reducir la fuente para encajar hasta 200 caracteres en la pantalla. No lo haga. Mi límite personal es de 120.

Apertura y densidad horizontal

Usamos el espacio en blanco horizontal para asociar elementos directamente relacionados y separar otros con una relación menos estrecha. Fíjese en la siguiente función:

```
private void measureLine(String line) {
    lineCount++;
    int lineSize = line.length();
    totalChars += lineSize;
    lineWidthHistogram.addLine(lineSize, lineCount);
    recordWidestLine(lineSize);
}
```

Hemos rodeado los operadores de asignación con espacios en blanco para destacarlos. Las instrucciones de asignación tienen dos elementos principales: el lado izquierdo y el derecho. Los espacios acentúan esta separación.

Por otra parte, no hemos incluido espacios entre los nombres de las funciones y el paréntesis de apertura, ya que la función y sus argumentos están estrechamente relacionados. Su separación los desconectaría. Separo los argumentos en los paréntesis de invocación de la función para acentuar la coma e indicar que los argumentos son independientes. El espacio en blanco también se usa para acentuar la precedencia de los operadores:

```
public class Quadratic {
    public static double root1(double a, double b, double c) {
        double determinant = determinant(a, b, c);
        return (-b + Math.sqrt(determinant)) / (2*a);
    }

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

    private static double determinant(double a, double b, double c) {
        return b*b - 4*a*c;
    }
}
```

Fíjese en lo bien que se leen las ecuaciones. Los factores carecen de espacios en blanco ya que tienen una mayor precedencia. Los términos se separan mediante espacios en blanco ya que la suma y la resta son de precedencia inferior.

Desafortunadamente, muchas herramientas de formato de código ignoran la precedencia de los operadores e imponen un espaciado uniforme. Por ello, separaciones sutiles como las anteriores suelen perderse tras modificar el formato del código.

Alineación horizontal

Cuando era programador de lenguajes de ensamblado[28], usaba la alineación horizontal para acentuar determinadas estructuras. Cuando comencé a programar en C, C++ y Java, seguía intentando alinear los nombres de variables en un conjunto de declaraciones o todos los valores en un grupo de instrucciones de asignación. El aspecto de mi código era el siguiente:

```
public class FitNesseExpediter implements ResponseSender
{
private     Socket           socket;
private     InputStream      input;
private     OutputStream     output;
private     Request          request;
private     Response         response;
private     FitNesseContext  context;
protected   long             requestParsingTimeLimit;
private     long             requestProgress;
private     long             requestParsingDeadline;
private     boolean          hasError;

public FitNesseExpediter(     Socket s,
```

```
                            FitNesseContext context) throws Exception
{
this.context =              context;
socket =                    s;
input =                     s.getInputStream();
output =                    s.getOutputStream();
requestParsingTimeLimit =   10000;
}
```

Sin embargo, este tipo de alineación no es útil. Parece enfatizar los elementos incorrectos y aleja la vista de la verdadera intención. Por ejemplo, en la lista anterior de declaraciones, nos vemos tentados a leer la lista de nombres de variables sin fijarnos en sus tipos. Del mismo modo, en la lista de instrucciones de asignación, nos fijamos en los valores sin ver el operador. Para empeorarlo todo, las herramientas automáticas de formato suelen eliminar este tipo de alineación. Por tanto, al final, ya no lo uso. Ahora prefiero declaraciones y asignaciones sin alinear, como se muestra a continuación, ya que resaltan una deficiencia importante. Si tengo listas extensas que deben alinearse, el problema es la longitud de las listas, no la falta de alineación. La longitud de la siguiente lista de declaraciones de `FitNesseExpediter` sugiere que esta clase debe dividirse.

```
public class FitNesseExpediter implements ResponseSender
{
    private Socket socket;
    private InputStream input;
    private OutputStream output;
    private Request request;
    private Response response;
    private FitNesseContext context;
    protected long requestParsingTimeLimit;
    private long request Progress;
    private long requestParsingDeadline;
    private boolean hasError;


    public FitNesseExpediter(Socket s, FitNesseContext context) throws Exception
    {
        this.context = context;
        socket = s;
        input = s.getInputStream();
        output = s.getOutputStream();
        requestParsingTimeLimit = 10000;
    }
```

Sangrado

Un archivo de código es una jerarquía más que un contorno. Incluye información que pertenece a la totalidad del archivo, a sus clases individuales, a los métodos de las clases, a los bloques de los métodos y a los bloques de los bloques. Cada nivel de esta jerarquía es un ámbito en el que se pueden declarar nombres y en el que se interpretan declaraciones e instrucciones ejecutables.

Para que esta jerarquía de ámbitos sea visible, sangramos las líneas de código fuente de acuerdo a su posición en la jerarquía. Las instrucciones al

nivel del archivo, como las declaraciones de clases, no se sangran. Los métodos de una clase se sangran un nivel a la derecha de la clase. Las implementaciones de dichos métodos se implementan un nivel a la derecha de la declaración de los métodos. Las implementaciones de bloques se implementan un nivel a la derecha de su bloque contenedor y así sucesivamente.

Los programadores dependen de este sistema de sangrado. Alinean visualmente las líneas a la izquierda para ver el ámbito al que pertenece. De este modo pueden acceder rápidamente a los ámbitos, como por ejemplo a implementaciones de instrucciones `if` o `while`, que no son relevantes para la situación actual. Buscan en la izquierda nuevas declaraciones de métodos, variables e incluso clases. Sin el sangrado, los programas serian prácticamente ilegibles.

Fíjese en los siguientes programas, sintáctica y semánticamente idénticos:

```
public class FitNesseServer implements SocketServer { private FitNesseContext
context; public FitNesseServer(FitNesseContext context) { this.context =
context; } public void serve(Socket s) { serve(s, 10000); } public void
serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new
FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); }
catch(Exception e) { e.printStackTrace(); } } }
```

```
public class FitNesseServer implements SocketServer {
    private FitNesseContext context;

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

    public void serve(Socket s) {
        serve (s, 10000);
    }

    public void serve(Socket s, long requestTimeout) {
        try {
            FitNesseExpediter sender = new FitNesseExpediter(s, context);
            sender.setRequestParsingTimeLimit(requestTimeout);
            sender.start();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

A la vista puede incluso apreciar la estructura del archivo sangrado. Detectamos inmediatamente las variables, constructores y métodos de acceso. En cuestión de segundos vemos que es una especie de interfaz de conexión, con un tiempo de espera. La versión sin sangrar, por su parte, es prácticamente impenetrable.

Romper el sangrado

En ocasiones tenemos la tentación de romper la regla de sangrado con instrucciones `if` breves, bucles `while` breves o funciones breves. Siempre que he sucumbido a esta tentación, he acabado por volver a aplicar el sangrado. Por ello, evito replegar ámbitos a una línea, como en este ejemplo:

```
public class CommentWidget extends TextWidget
{
    public static final String REGEXP = “^#[^\r\n]*(?:(?:\r\n)|\n|\r)?”;


    public CommentWidget(ParentWidget parent, String text){super (parent, text);}
    public String render() throws Exception { return “”; }
}
```

Prefiero desplegar y sangrar los ámbitos:

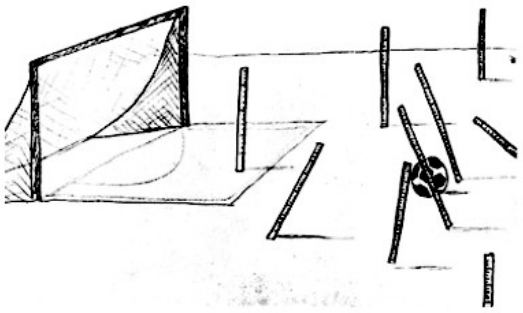
```
public class CommentWidget extends TextWidget {
    public static final String REGEXP = “^#[^\r\n]*(?:(?:\r\n)|\n|\r)?”;


    public CommentWidget(ParentWidget parent, String text) {
        super(parent, text);
    }


    public String render() throws Exception {
        return “”;
    }
}
```

Ámbitos ficticios

En ocasiones, el cuerpo de una instrucción `while` o `for` es ficticio, como se muestra a continuación. No me gustan estas estructuras y prefiero evitarlas. En caso de no poder hacerlo, me aseguro de sangrar el cuerpo ficticio y de incluirlo entre paréntesis. No sabría decir cuántas veces me ha engañado un punto y coma situado al final de un bucle `while` en la misma línea. A menos que lo haga visible y lo sangre en una línea propia, es difícil de ver.

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

Reglas de equipo



Todo programador tiene sus reglas de formato preferidas, pero si forma parte de un equipo, el equipo manda.

Un equipo de programadores debe acordar un único estilo de formato y todos los integrantes del equipo deben aplicarlo. El objetivo es que el *software* tenga un estilo coherente. No queremos que parezca escrito por individuos enfrentados.

Cuando comencé el proyecto FitNesse en 2002, me reuní con el equipo para definir un estilo de código. Tardamos 10 minutos. Decidimos dónde añadir las llaves, qué tamaño de sangrado utilizar, los nombres de clases, variables y métodos, y demás. Tras ello, codificamos las reglas en el IDE y las cumplimos desde entonces. No son las reglas que prefiero, son las que el equipo decidió. Y como miembro de ese equipo, las apliqué cuando creamos el código del proyecto FitNesse.

Recuerde que un buen sistema de *software* se compone de una serie de documentos que se leen fácilmente. Deben tener un estilo coherente y dinámico. El lector debe confiar en que los formatos que ve en nuestro archivo de código significarán lo mismo para otros. Lo último que queremos es aumentar la complejidad del código creando una mezcla de estilos diferentes.

Reglas de formato de Uncle Bob

Las reglas que uso personalmente son sencillas y se ilustran en el código del Listado 5-6. Considérelo un ejemplo de documento estándar de código óptimo.

Listado 5-6
CodeAnalyzer.java.

```
public class CodeAnalyzer implements JavaFileAnalysis {
    private int lineCount;
    private int maxLineWidth;
    private int widestLineNumber;
    private LineWidthHistogram lineWidthHistogram;
    private int totalChars;


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


    public static List<File> findJavaFiles(File parentDirectory) {
        List<File> files = new ArrayList<File>();
        findJavaFiles(parentDirectory, files);
        return files;
    }


    private static void findJavaFiles(File parentDirectory, List<File> files) {
        for (File file : parentDirectory.listFiles()) {
            if (file.getName().endsWith(“.java”))
                files.add(file);
            else if (file.isDirectory())
                findJavaFiles(file, files);
        }
    }


    public void analyzeFile(File javaFile) throws Exception {
        BufferedReader br = new BufferedReader(new FileReader(javaFile));
        String line;
        while ((line = br.readLine()) != null)
            measureLine(line);
    }


    private void measureLine(String line) {
        lineCount++;
        int lineSize = line.length();
        totalChars += lineSize;
        lineWidthHistogram.addLine(lineSize, lineCount);
        recordWidestLine(lineSize);
    }


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


    public int getLineCount() {
        return lineCount;
    }


    public int getMaxLineWidth() {
        return maxLineWidth;
    }


    public int getWidestLineNumber() {
        return widestLineNumber;
    }


    public LineWidthHistogram getLineWidthHistogram() {
        return lineWidthHistogram;
    }


    public double getMeanLineWidth() {
        return (double)totalChars/lineCount;
    }


    public int getMedianLineWidth() {
        Integer[] sortedwidths = getSortedWidths();
        int cumulativeLineCount = 0;
        for (int width : sortedwidths) {
            cumulativeLineCount += lineCountForWidth(width);
            if (cumulativeLineCount > lineCount/2)
                return width;
        }
        throw new Error (“Cannot get here”);
    }


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


    private Integer[] getSortedWidths() {
        Set<Integer> widths = lineWidthHistogram.getWidths();
```

```
        Integer[] sortedwidths = (widths.toArray(new Integer[0]));
        Arrays.sort(sortedwidths);
        return sortedwidths;
    }
}
```

6
Objetos y estructuras de datos

Hay una razón para que las variables sean privadas. No queremos que nadie más dependa de ellas. Queremos poder cambiar su tipo o implementación

cuando deseemos. Entonces, ¿por qué tantos programadores añaden automáticamente métodos de establecimiento y recuperación que muestran sus variables privadas como si fueran públicas?

Abstracción de datos

Fíjese en la diferencia entre los listados 6-1 y 6-2. Ambos representan los datos de un punto cartesiano, pero uno muestra su implementación y otro la oculta totalmente.

Listado 6-1
Punto concreto.

```
public class Point {
    public double x;
    public double y;
}
```

Listado 6-2
Punto abstracto.

```
public interface Point {
    double getX();
    double getY();
    void setCartesian(double x, double y);
    double getR();
    double getTheta();
    void set Polar(double r, double theta);
}
```

Lo mejor del Listado 6-2 es que no hay forma de saber si la implementación está en coordenadas rectangulares o polares. ¡Puede que en ninguna! Y aun así la interfaz representa sin lugar a dudas una estructura de datos.

Pero representa algo más que una estructura de datos. Los métodos refuerzan una política de acceso. Puede leer las coordenadas de forma independiente, pero debe establecerlas de forma conjunta como operación atómica.

El Listado 6-1, por su parte, se implementa claramente en coordenadas rectangulares y nos obliga a manipularlas de forma independiente, lo que muestra la implementación. De hecho, la mostraría igualmente, aunque las variables fueran privadas y usáramos métodos variables de establecimiento y recuperación. Para ocultar la implementación no basta con añadir una capa de funciones entre las variables. Se basa en la abstracción. Una clase

no fuerza sus variables a través de métodos de establecimiento y recuperación. Por el contrario, muestra interfaces abstractas que permiten a sus usuarios manipular la esencia de los datos sin necesidad de conocer su implementación.

Fíjese en los listados 6-3 y 6-4. El primero usa términos concretos para indicar el nivel de combustible de un vehículo mientras que el segundo lo hace con la abstracción del porcentaje. En el caso concreto, podemos estar seguros de que se trata de métodos de acceso de variables. En el caso abstracto, desconocemos la forma de los datos.

Listado 6-3
Vehículo concreto.

```
public interface Vehicle {
    double getFuelTankCapacityInGallons();
    double getGallonsOfGasoline();
}
```

Listado 6-4
Vehículo abstracto.

```
public interface Vehicle {
    double getPercentFuelRemaining();
}
```

En ambos casos, la segunda opción es preferible. No queremos mostrar los detalles de los datos, sino expresarlos en términos abstractos. Esto no se consigue simplemente mediante interfaces o métodos de establecimiento y recuperación. Hay que meditar seriamente la forma óptima de representar los datos que contiene un objeto. La peor opción es añadir métodos de establecimiento y recuperación a ciegas.

Antisimetría de datos y objetos

Estos dos ejemplos ilustran la diferencia entre objetos y estructuras de datos. Los objetos ocultan sus datos tras abstracciones y muestran funciones que operan en dichos datos. La estructura de datos muestra sus datos y carece de funciones con significado. Vuelva a leerlos. Fíjese en la naturaleza complementaria de las dos definiciones. Son virtualmente opuestas. Puede parecer una diferencia menor, pero tiene importantes implicaciones.

Fíjese en el ejemplo del Listado 6-5. La clase `Geometry` opera en las tres clases de formas, que son sencillas estructuras de datos sin comportamiento. Todo el comportamiento se encuentra en la clase Geometry.

Listado 6-5
Forma mediante procedimientos.

```
public class Square {
    public Point topLeft;
    public double side;
}


public class Rectangle {
    public Point topLeft;
    public double height;
    public double width;
}


public class Circle {
    public Point center;
    public double radius;
}


public class Geometry {
    public final double PI = 3.141592653589793;


    public double area(Object shape) throws NoSuchShapeException
    {
        if (shape instanceof Square) {
            Square s = (Square)shape;
            return s.side * s.side;
        }
        else if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle)shape;
            return r.height * r.width;
        }
        else if (shape instanceof Circle) {
            Circle c = (Circle)shape;
            return PI * c.radius * c.radius;
        }
        throw new NoSuchShapeException();
    }
}
```

Los programadores orientados a objetos se quejarán de que es un ejemplo de procedimiento, y tienen razón. Imagine qué pasaría si añadimos la función `perimeter()` a `Geometry`. ¡Las clases de formas no se verían afectadas! ¡Y las demás clases que dependieran de las formas tampoco! Por otra parte, si añado una nueva forma, tendría que cambiar todas las funciones de `Geometry`. Vuélvalo a leer. Comprobará que las dos condiciones son diametralmente opuestas.

Fíjese ahora en la solución orientada a objetos del Listado 6-6. Aquí, el método `area()` es polimórfico. No se necesita una clase `Geometry`. Por tanto, si añado una nueva forma, ninguna de las funciones existentes se ven afectadas, pero si añado otra función, habrá que cambiar todas las formas[29].

Listado 6-6

Formas polimórficas.

```
public class Square implements Shape {
    private Point topLeft;
    private double side;


    public double area() {
    return side*side;
}


public class Rectangle implements Shape {
    private Point topLeft;
    private double height;
    private double width;


    public double area() {
        return height * width;
    }
}


public class Circle implements Shape {
    private Point center;
    private double radius;
    public final double PI = 3.141592653589793;


    public double area() {
        return PI * radius * radius;
    }
}
```

De nuevo, vemos la naturaleza complementaria de estas dos definiciones; totalmente contrarias. Esto ilustra la dicotomía fundamental entre objetos y estructuras de datos:

> *El código por procedimientos (el que usa estructuras de datos) facilita la inclusión de nuevas funciones sin modificar las estructuras de datos existentes. El código orientado a objetos, por su parte, facilita la inclusión de nuevas clases sin cambiar las funciones existentes.*

El complemento también es cierto:

> *El código por procedimientos dificulta la inclusión de nuevas estructuras de datos ya que es necesario cambiar todas las funciones. El código orientado a objetos dificulta la inclusión de nuevas funciones ya que es necesario cambiar todas las clases.*

Por tanto, lo que es difícil para la programación orientada a objetos es fácil para los procedimientos, y viceversa.

En cualquier sistema complejo habrá ocasiones en las que queramos añadir nuevos tipos de datos en lugar de nuevas funciones. En dichos casos,

los objetos y la programación orientada a objetos es lo más adecuado. Por otra parte, en ocasiones tendremos que añadir nuevas funciones en lugar de tipos de datos, para lo que resulta más adecuado usar código por procedimientos y estructuras de datos.

Los programadores experimentados saben que la idea de que todo es un objeto es un mito. En ocasiones solamente *queremos* sencillas estructuras de datos con procedimientos que operen en las mismas.

La ley de Demeter

Existe una conocida heurística denominada *Ley de Demeter*[30] que afirma que un módulo no debe conocer los entresijos de los objetos que manipula. Como vimos en el apartado anterior, los objetos ocultan sus datos y muestran operaciones, lo que significa que un objeto no debe mostrar su estructura interna a través de métodos de acceso ya que, si lo hace, mostraría, no ocultaría, su estructura interna.

En concreto, la ley de Demeter afirma que un método de una clase `C` sólo debe invocar los métodos de:

- `C`.
- Un objeto creado por `f`.
- Un objeto pasado como argumento a `f`.
- Un objeto en una variable de instancia de `C`.

El método *no* debe invocar métodos de objetos devueltos por ninguna de las funciones permitidas. Es decir, no hable con desconocidos, sólo con amigos.

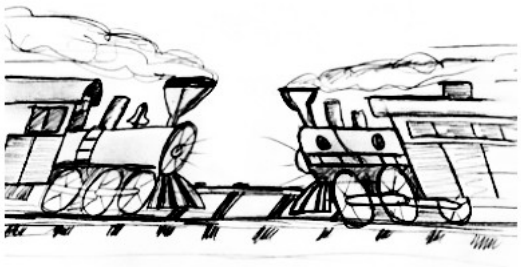
El siguiente código[31] parece incumplir la Ley de Demeter (entre otras cosas) ya que invoca la función `getScratchDir()` en el valor devuelto de `getOptions()` y después invoca `getAbsolutePath()` en el valor devuelto de `getScratchDir()`.

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

Choque de trenes

Ese tipo de código suele denominarse choque de trenes ya que se asemeja a un grupo de vagones de tren. Estas cadenas de invocaciones suelen considerarse un estilo descuidado y deben evitarse [G36]. Conviene dividirlas de esta forma:

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```



¿Incumplen estos dos fragmentos de código la Ley de Demeter? Sin duda el módulo contenedor sabe que el objeto `ctxt` contiene opciones, que contienen un directorio `scratch`, que tiene una ruta absoluta. La función sabe demasiado. La función que realiza la invocación sabe cómo desplazarse por numerosos objetos diferentes.

Si incumple o no la Ley de Demeter depende de si `ctxt`, `Options` y `ScratchDir` son objetos o estructuras de datos. Si son objetos, debería ocultarse su estructura interna, no mostrarse, y conocer sus detalles internos sería un claro incumplimiento de la Ley de Demeter. Por otra parte, si `ctxt`, `Options` y `ScratchDir` son simples estructuras de datos, mostrarán su estructura interna con naturalidad y la Ley de Demeter no se aplica.

El uso de funciones de acceso complica el problema. Si el código se hubiera escrito de esta otra forma, probablemente no nos preocuparíamos de si se incumple la Ley de Demeter o no.

```
final String outputDir = ctxt.options.ScratchDir.absolutePath;
```

El problema sería menos confuso si las estructuras de datos tuvieran variables públicas y no funciones, y los objetos tuvieran variables privadas y funciones públicas. Sin embargo, existen estructuras y estándares (como los *bean*) que exigen que incluso una sencilla estructura de datos tenga elementos de acceso y mutación.

Híbridos

Esta confusión genera ocasionalmente desafortunadas estructuras híbridas mitad objeto y mitad estructura de datos. Tienen funciones que realizan tareas significativas y también variables públicas o método públicos de acceso y mutación que hacen que las variables privadas sean públicas, y tientan a otras funciones externas a usar dichas variables de la misma forma que un programa por procedimientos usaría una estructura de datos[32]. Estos híbridos dificultan la inclusión de nuevas funciones y también de nuevas estructuras de datos. Son lo peor de ambos mundos. Evítelos. Indican un diseño descuidado cuyos autores dudan, o peor todavía, desconocen, si necesitan protegerse de funciones o tipos.

Ocultar la estructura

¿Qué pasaría si `ctxt`, `options` y `scratchDir` fueran objetos con un comportamiento real? Como los objetos deben ocultar su estructura interna, no podríamos desplazarnos por los mismos. Entonces, ¿cómo obtendríamos la ruta absoluta del directorio `scratch`?

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

o

```
ctxt.getScratchDirectoryOption().getAbsolutePath()
```

La primera opción provocaría una explosión de métodos en el objeto `ctxt`. La segunda asume que `getScratchDirectoryOption()` devuelve una estructura de datos, no un objeto. Ninguna de las opciones parece correcta. Si `ctxt` es un objeto, deberíamos indicarle que hiciera algo, no preguntar sobre sus detalles internos. Entonces, ¿para qué queremos la ruta absoluta del directorio `scratch`? ¿Cómo vamos a usarla? Fíjese en este código del mismo módulo (muchas líneas después):

```
String outFile = outputDir + “/” + className.replace(‘.’, ‘/’) + “.class”;
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

La mezcla de distintos niveles de detalle [G34][G6] es preocupante. Puntos, guiones, extensiones de archivo y objetos `File` no deben mezclarse de esta forma, junto al código contenedor. Si lo ignoramos, vemos que la intención de obtener la ruta absoluta del directorio `scratch` es crear un archivo de borrador de un nombre concreto.

¿Y si le dijéramos al objeto `ctxt` que hiciera esto?

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

Parece algo razonable para un objeto. Permite a `ctxt` ocultar sus detalles internos e impide que la función actual incumpla la Ley de Demeter y se desplace por objetos que no debería conocer.

Objetos de transferencia de datos

La quintaesencia de una estructura de datos es una clase con variables públicas y sin funciones. En ocasiones se denomina Objeto de transferencia de datos (*Data Transfer Object* u OTD). Los OTD son estructuras muy útiles, en especial para comunicarse con bases de datos o analizar mensajes de conexiones, etc. Suelen ser los primeros de una serie de fases de traducción que convierten datos sin procesar en objetos en el código de la aplicación. Más común es la forma de bean mostrada en el Listado 6-7. Los bean tienen variables privadas manipuladas por métodos de establecimiento y recuperación. La cuasi-Encapsulación de bean hace que algunos puristas de la programación orientada a objetos se sientan mejor pero no ofrece ningún otro beneficio.

Listado 6-7
address.java

```
public class Address {
    private String street;
    private String streetExtra;
    private String city;
    private String state;
    private String zip;


    public Address(String Street, String streetExtra,
                String city, String state, String zip) {
        this.street = street;
        this.streetExtra = streetExtra;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }


    public String getStreet() {
        return street;
    }


    public String getStreetExtra() {
        return streetExtra;
    }


    public String getCity() {
        return city;
    }


    public String getState() {
        return getState;
    }


    public String getZip() {
        return zip;
    }
```

```
}
```

Registro activo

Los registros activos son una forma especial de OTD. Son estructuras de datos con variables públicas (o de acceso por bean) pero suelen tener métodos de navegación como `save` y `find`. Por lo general, estos registros activos son traducciones directas de tablas de base de datos u otros orígenes de datos. Desafortunadamente, muchos programadores intentan procesar estas estructuras de datos como si fueran objetos y les añaden métodos de reglas empresariales. Es algo extraño ya que crea un híbrido entre una estructura de datos y un objeto.

La solución, evidentemente, consiste en considerar al registro activo una estructura de datos y crear objetos independientes que contengan las reglas empresariales y que oculten sus datos internos (que probablemente sean instancias del propio registro activo).

Conclusión

Los objetos muestran comportamiento y ocultan datos. Esto facilita la inclusión de nuevos tipos de objetos sin necesidad de cambiar los comportamientos existentes. También dificulta la inclusión de nuevos comportamientos en objetos existentes. Las estructuras de datos muestran datos y carecen de comportamiento significativo. Esto facilita la inclusión de nuevos comportamientos en las estructuras de datos existentes, pero dificulta la inclusión de nuevas estructuras de datos en funciones existentes.

En un sistema, en ocasiones necesitaremos la flexibilidad de añadir nuevos tipos de datos, por lo que preferimos objetos para esa parte del sistema. En otros casos, querremos añadir nuevos comportamientos, para lo que preferimos tipos de datos y procedimientos en esa parte del sistema. Los buenos programadores de *software* entienden estos problemas sin prejuicios y eligen el enfoque más adecuado para cada tarea concreta.

Bibliografía

- **[Refactoring]**: *Refactoring: Improving the Design of Existing Code*, Martin Fowler et al., Addison-Wesley, 1999.

7
Procesar errores

por Michael Feathers

Le parecerá extraño encontrar una sección de control de errores en un libro sobre código limpio. El control de errores es algo que todos tenemos que hacer al programar. Las entradas pueden ser incorrectas y los dispositivos pueden fallar, y cuando lo hacen, los programadores somos responsables de comprobar que el código hace lo que debe hacer.

No obstante, la conexión con el código limpio debe ser evidente. Muchas bases de código están totalmente dominadas por el control de errores. Cuando digo que están dominadas, no quiero decir que únicamente realicen control de código, sino que es prácticamente imposible ver lo que el código hace debido a todo ese control de errores. El control de errores es importante, *pero si oscurece la lógica*, *es incorrecto*.

En este capítulo detallaremos diversas técnicas y consideraciones que puede usar para crear código limpio y robusto, código que procese los errores con elegancia y estilo.

Usar excepciones en lugar de códigos devueltos

En el pasado, muchos lenguajes carecían de excepciones. Las técnicas para procesar e informar de errores eran limitadas. Se definía un indicador de error o se devolvía un código de error que el invocador podía comprobar. El código del Listado 7-1 ilustra estos enfoques.

Listado 7-1
DeviceController.java.

```
public class DeviceController {
    …
    public void sendShutDown() {
        DeviceHandle handle = getHandle(DEV1);
        // Comprobar el estado del dispositivo
        if (handle != DeviceHandle.INVALID) {
            // Guardar el estado del dispositivo en el campo de registro
            retrieveDeviceRecord(handle);
            // Si no está suspendido, cerrarlo
            if { record.getStatus() != DEVICE_SUSPENDED) {
                pauseDevice(handle);
                clearDeviceWorkQueue(handle);
                closeDevice(handle);
            } else {
                logger.log(“Device suspended. Unable to shut down”);
            }
        } else {
            logger.log(“Invalid handle for: ” + DEV1.toString());
        }
    }
    …
}
```

El problema de estos enfoques es que confunden al invocador. El invocador debe comprobar inmediatamente los errores después de la invocación. Desafortunadamente, es algo que se suele olvidar. Por ello, es más recomendable generar una excepción al detectar un error. El código de invocación es más limpio. Su lógica no se oscurece por el control de errores.

El Listado 7-2 muestra el código tras generar una excepción en los métodos que pueden detectar errores.

Listado 7-2
DeviceController.java (con excepciones).

```
public class DeviceController {
    …

    public void sendShutDown() {
        try {
            tryToShutDown();
        } catch (DeviceShutDownError e) {
            logger.log(e);
        }
    }

    private void tryToShutDown() throws DeviceShutDownError {
        DeviceHandle handle = getHandle(DEV1);
        DeviceRecord record = retrieveDeviceRecord(handle);
```

```
            pauseDevice(handle);
            clearDeviceWorkQueue(handle);
            closeDevice(handle);
        }

        private DeviceHandle getHandle(DeviceID id) {
            …
            throw new DeviceShutDownError(“Invalid handle for: ” - id.toString());
            …
        }
        …
    }
```

Comprobará que es mucho más limpio. No es cuestión de estética. El código es mejor porque se solventan dos preocupaciones: el algoritmo para apagar el dispositivo y el control de errores ahora se encuentran separados. Puede ver cada uno de ellos y entenderlos de forma independiente.

Crear primero la instrucción try-catch-finally

Uno de los aspectos más interesantes de las excepciones es que *definen un ámbito* en el programa. Al ejecutar código en la parte `try` de una instrucción `try-catch-finally`, indicamos que la ejecución se puede cancelar en cualquier momento y después retomar en `catch`.

Los bloques `try` son como las transacciones, `catch` debe salir del programa en un estado coherente, independientemente de lo que suceda en `try`. Por este motivo, es aconsejable iniciar con una instrucción `try-catch-finally` el código que genere excepciones. De este modo define lo que debe esperar el usuario del código, independientemente de que se produzca un error en el código ejecutado en la cláusula `try`.

Veamos un ejemplo. Imagine que tiene que crear un código que acceda a un archivo y lea objetos serializados.

Comenzamos con una prueba de unidad que muestra que obtendremos una excepción cuando el archivo no exista:

```
@Test(expected = StorageException.class)
public void retrieveSectionShouldThrowOnInvalidFileName() {
    sectionStore.retrieveSection(“invalid - file”);
}
```

La prueba nos lleva a crear lo siguiente:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
    // se devuelve un resultado ficticio hasta tener una implementación real
    return new ArrayList<RecordedGrip>();
}
```

Nuestra prueba falla ya que no genera una excepción. Tras ello, cambiamos la implementación para que intente acceder a un archivo no válido. Esta operación genera una excepción:

```
public List<RecordedGrip> retrieveSection (String sectionName) {
    try {
        FileInputstream stream = new FileInputStream(sectionName)
```

```
        } catch (Exception e) {
            throw new StorageException("retrieval error", e);
        }
        return new ArrayList<RecordedGrip>();
    }
```

Ahora la prueba es correcta ya que capturamos la excepción y ya podemos refactorizar. Podemos reducir el tipo de la excepción capturada para que coincida con el tipo generado desde el constructor `FileInputStream: FileNotFoundException`:

```
    public List<RecordedGrip> retrieveSection(String sectionName) {
        try {
            FileInputStream stream = new FileInputStream(sectionName);
            stream.close();
        } catch (FileNotFoundException e) {
            throw new StorageException("retrieval error", e);
        }
        return new ArrayList<RecordedGrip>();
    }
```

Ahora que hemos definido el ámbito con una estructura `try-catch`, podemos usar TDD para diseñar el resto de la lógica necesaria. Dicha lógica se añade entre la creación de `FileInputStream` y el cierre, y podemos pretender que no pasa nada incorrecto.

Intente crear pruebas que fuercen las excepciones, para después añadir al controlador un comportamiento que satisfaga dichas pruebas. De este modo primero creará el ámbito de transacción del bloque `try` y podrá mantener la naturaleza de transacción del ámbito.

Usar excepciones sin comprobar

El debate ha terminado. Durante años, los programadores de Java han debatido las ventajas y los problemas de las excepciones comprobadas. Cuando aparecieron en la primera versión de Java, parecían una gran idea. La firma de todos los métodos enumeraría todas las excepciones que se podían pasar a su invocador. Es más, estas excepciones formaban parte del tipo del método. El código no se compilaría si la firma no coincidía con lo que el código iba a hacer.

En aquel momento, pensábamos que las excepciones comprobadas eran una gran idea y sí, ofrecían ciertas ventajas. Sin embargo, ahora es evidente que no se necesitan para crear *software* robusto. C# carece de excepciones comprobadas y, a pesar de los intentos, C++ tampoco, como sucede en Python o Ruby. Y en todos estos lenguajes se puede crear *software* robusto. Por ello, debemos decidir si las excepciones comprobadas valen su precio.

¿Qué precio? El precio de las excepciones comprobadas es un incumplimiento del principio abierto/cerrado[33]. Si genera una excepción comprobada desde un método de su código y la cláusula `catch` se encuentra tres niveles por debajo, debe declarar dicha excepción en la firma de todos los métodos comprendidos entre su posición y `catch`. Esto significa que un cambio en un nivel inferior del *software* puede forzar cambios de firma en muchos niveles superiores. Será necesario volver a generar e implementar los módulos cambiados, aunque no cambien los elementos a los que hacen referencia.

Piense en la jerarquía de invocación de un sistema. Las funciones de la parte superior invocan a las funciones situadas debajo, que invocan a otras funciones inferiores y así sucesivamente. Imagine que una de las funciones de nivel inferior se modifica de forma que debe generar una excepción. Si la excepción se comprueba, la firma de la función tendrá que añadir una cláusula `throws`. Pero esto significa que todas las funciones que invoquen nuestra función modificada también tendrán que cambiarse para capturar la nueva excepción o para añadir la correspondiente cláusula `throws` en su firma. Y así indefinidamente. El resultado final es una cascada de cambios que pasan desde los niveles inferiores del *software* hasta los superiores. La encapsulación se rompe ya que todas las funciones en la ruta de `throw` deben conocer detalles de la excepción de nivel inferior. Como el cometido de las excepciones es permitimos procesar errores a distancia, es una lástima que las excepciones comprobadas rompan la encapsulación de esta forma.

Las excepciones comprobadas pueden ser útiles si tiene que crear una biblioteca crítica: tendrá que capturarlas. Pero en el desarrollo de aplicaciones generales, los costes de dependencia superan las ventajas.

Ofrecer contexto junto a las excepciones

Las excepciones que genere deben proporcionar el contexto adecuado para determinar el origen y la ubicación de un error. En Java, puede obtener un rastreo de pila de cualquier excepción; sin embargo, no le indicará el cometido de la función fallida.

Redacte mensajes de error informativos y páselos junto a sus excepciones. Mencione la operación fallida y el tipo de fallo. Si guarda

registros en su aplicación, incluya información suficiente para poder registrar el error en la cláusula `catch`.

Definir clases de excepción de acuerdo a las necesidades del invocador

Existen varias formas de clasificar los errores. Podemos hacerlo por origen (¿provienen de uno u otro componente?) o por tipo (¿son fallos del dispositivo, de la red o errores de programación?). Sin embargo, al definir clases de excepción en una aplicación, debemos preocuparnos principalmente en *cómo se capturan*.

Veamos un pobre ejemplo de clasificación de excepciones. Es una instrucción `try-catch-finally` de la invocación de una biblioteca de terceros. Abarca todas las excepciones que las invocaciones pueden generar:

```
ACMEPort port = new ACMEPort(12);


try {
    port.open();
} catch (DeviceResponseException e) {
    reportPortError(e);
    logger.log{“Device response exception”, e);
} catch (ATM1212UnlockedException e) {
    reportPortError(e);
    logger.log(“Unlock exception”, e);
} catch (GMXError e) {
    reportPortError(e);
    logger.log(“Device response exception”);
} finally {
    …
}
```

Esta instrucción contiene elementos duplicados, algo que no debería sorprendernos. En muchos casos de control de excepciones, el trabajo que realizamos es relativamente estándar independientemente de la causa real. Debemos registrar un error y asegurarnos de poder continuar.

En este caso, como sabemos que el trabajo es el mismo independientemente de la excepción, podemos simplificar el código si incluimos la API invocada y nos aseguramos de que devuelve un tipo de excepción común:

```
LocalPort port = new LocalPort(12);
try {
    port.open();
} catch (PortDeviceFailure e) {
    reportError(e);
    logger.log(e.getMessage(), e);
} finally {
    …
}
```

Nuestra clase `LocalPort` es un simple envoltorio que captura y traduce excepciones generadas por la clase `ACMEPort`:

```
public class LocalPort {
    private ACMEPort innerPort;


    public LocalPort(int portNumber) {
        innerPort = new ACMEPort(portNumber);
    }


    public void open() {
        try {
            innerPort.open();
        } catch (DeviceResponseException e) {
            throw new PortDeviceFailure(e);
        } catch (ATM1212UnlockedException e) {
            throw new PortDeviceFailure(e);
        } catch (GMXError e) {
            throw new PortDeviceFailure(e);
        }
    }
    …
}
```

Los envoltorios como el definido para `ACMEPort` pueden ser muy útiles. De hecho, es recomendable envolver API de terceros. Al hacerlo, se minimizan las dependencias: puede cambiar a otra biblioteca diferente sin apenas problemas y el envoltorio también facilita imitar invocaciones de terceros cuando se prueba el código. Una última ventaja es que no estamos limitados a las decisiones de diseño de API de un determinado fabricante. Puede definir una API que le resulte cómoda. En el ejemplo anterior, definimos un único tipo de excepción para el fallo de puertos y podemos escribir un código mucho más limpio. A menudo, una única clase de excepción es suficiente para una zona concreta del código. La información enviada con la excepción puede distinguir los errores. Use clases diferentes sólo para capturar una excepción y permitir el paso de otra distinta.

Definir el flujo normal

Si sigue los consejos de apartados anteriores, realizará una importante separación entre la lógica empresarial y el control de errores. La mayoría de su código parecerá un algoritmo limpio y sin adornos. Sin embargo, el proceso desplaza la detección de errores hacia los bordes del programa. Debe envolver API externas para poder generar sus propias excepciones y definir un controlador por encima del código para poder procesar cálculos cancelados. En muchos casos es el enfoque más acertado, pero en ocasiones conviene no cancelar.

Veamos un ejemplo, un código extraño que suma gastos en una aplicación de facturación:

```
try {
    MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
    m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
    m_total += getMealPerDiem();
}
```

En esta empresa, si las comidas son gastos, pasan a formar parte del total. Si no lo son, los trabajadores reciben una cantidad diaria para la comida. La excepción entorpece la lógica. Sería más adecuado no tener que procesar el caso especial y el código sería mucho más sencillo:

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
m_total += expenses.getTotal();
```

¿De verdad que el código puede ser tan simple? Pues sí. Podemos cambiar `ExpenseReportDAO` para que siempre devuelva un objeto `MealExpense`. Si no hay gastos de comida, devuelve un objeto `MealExpense` que devuelve la dieta diaria como total:

```
public class PerDiemMealExpenses implements MealExpenses {
    public int getTotal() {
        // devolver la dieta diaria predeterminada
    }
}
```

Es lo que se denomina *Patrón de Caso Especial* [Fowler]. Se crea una clase o se configura un objeto que procese un caso especial. Al hacerlo, el código cliente no tiene que procesar comportamientos excepcionales. Dichos comportamientos se encapsulan en un objeto de caso especial.

No devolver Null

Creo que toda descripción del control de errores debe mencionar los elementos proclives a errores. El primero es devolver null. He perdido la cuenta de la cantidad de aplicaciones en que las que línea sí y línea también se comprueba null:

```
public void registerItem(Item item) {
    if (item != null) {
        ItemRegistry registry = peristentStore.getItemRegistry();
        if (registry != null) {
            Item existing = registry.getItem(item.getID());
            if (existing.getBillingPeriod().hasRetailOwner()) {
                existing.register(item);
            }
        }
    }
}
```

Si trabaja en una base de código como ésta, puede que no le parezca tan mala, pero lo es. Al devolver `null`, básicamente nos creamos trabajo y generamos problemas para los invocadores. Basta con que falte una comprobación de `null` para que la aplicación pierda el control.

¿Se ha fijado en que no hay una comprobación de `null` en la segunda línea de la instrucción `if` anidada? ¿Qué sucedería en tiempo de ejecución si `persistentStore` fuera `null`? Se generaría `NullPointerException` en tiempo de ejecución y se capturaría `NullPointerException` en el nivel superior o no. En ambos casos es incorrecto. ¿Qué debería hace como respuesta a la generación de `NullPointerException` desde el interior de su aplicación? Se puede afirmar que el problema de este código es la ausencia de una comprobación de `null` pero en realidad el problema es su exceso. Si siente la tentación de devolver `null` desde un método, pruebe a generar una excepción o a devolver un objeto de caso especial. Si invoca un método que devuelva `null` desde una API de terceros, envuélvalo en un método que genere una excepción o devuelva un objeto de caso especial. En muchos casos, los objetos de caso especial son un remedio sencillo. Imagine que tiene el siguiente código:

```
List<Employee> employees = getEmployees();
if (employees != null) {
    for(Employee e : employees) {
        totalPay += e.getPay();
    }
}
```

Ahora, `getEmployees` puede devolver `null`, ¿pero es necesario? Si cambiamos `getEmployee` para que devuelva una lista vacía, podremos limpiar el código:

```
List<Employee> employees = getEmployees();
for(Employee e : employees) {
    totalPay += e.getPay();
}
```

Afortunadamente, Java dispone de `Collections.emptyList()` y devuelve una lista inmutable predefinida que podemos usar para este cometido:

```
public List<Employee> getEmployees() {
    if (… there are no employees …)
        return Collections.emptyList();
}
```

Si usa este tipo de código, minimizará la presencia de `NullPointerException` y su código será más limpio.

No pasar Null

Devolver `null` desde métodos es incorrecto, pero es peor pasar `null` a métodos. A menos que trabaje con una API que espere que pase `null`, debe evitarlo siempre que sea posible. Veamos otro ejemplo, un sencillo método que calcula una métrica para dos puntos:

```
public class MetricsCalculator
{
    public double xProjection(Point p1, Point p2) {
        return (p2.x – p1.x) * 1.5;
    }
    …
}
```

¿Qué sucede cuando alguien pasa `null` como argumento?

```
calculator.xProjection(null, new Point (12, 13));
```

Se genera `NullPointerException`, evidentemente.

¿Cómo solucionarlo? Podríamos crear un nuevo tipo de excepción y generarla:

```
public class MetricsCalculator
{
    public double xProjection(Point p1, Point p2) {
        if (p1==null || p2==null) {
            throw InvalidArgumentException(
                “Invalid argument for MetricsCalculator.xProjection”);
        }
        return (p2.x – p1.x) * 1.5; }
    }
}
```

¿Mejor? Puede que sea mejor que una excepción de puntero nulo, pero recuerde que debe definir un controlador para `InvalidArgumentException`. ¿Qué debe hacer el controlador? ¿Hay alguna forma correcta de hacerlo?

Existe otra alternativa, usar un grupo de afirmaciones:

```
public class MetricsCalculator (
    public double xProjection{Point p1, Point p2) {
        assert p1 != null : “p1 should not be null”;
        assert p2 != null : “p2 should not be null”;
        return (p2.x - pl.x) * 1.5;
    }
}
```

Es documentación correcta pero no soluciona el problema. Si alguien pasa `null`, seguirá produciéndose un error de tiempo de ejecución.

En la mayoría de lenguajes de programación no hay una forma correcta de procesar un `null` pasado por accidente. Como éste es el caso, el enfoque racional es impedir que se pase `null` de forma predeterminada. Si

lo hace, puede diseñar código sabiendo que `null` en una lista de argumentos indica un problema y los errores serán menores.

Conclusión

El código limpio es legible pero también debe ser robusto. No son objetivos opuestos. Podemos crear código limpio y robusto si consideramos el control de errores una preocupación diferente, algo que vemos de forma independiente desde nuestra lógica principal. Si somos capaces de lograrlo, razonaremos de forma independiente y podemos aumentar la capacidad de mantenimiento de nuestro código.

Bibliografía

- **[Martin]**: *Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.

8
Límites

por James Grenning

No es habitual que controlemos todo el *software* de nuestros sistemas. En ocasiones, adquirimos paquetes de terceros o usamos código abierto. En otros casos, dependemos de equipos de nuestra propia empresa para producir componentes o subsistemas que utilizamos. De algún modo debemos integrar este código externo con el nuestro. En este capítulo veremos prácticas y técnicas para definir con claridad los límites de nuestro *software*.

Utilizar código de terceros

Existe una tensión natural entre el proveedor de una interfaz y el usuario de la misma. Los proveedores de paquetes y estructuras de terceros abogan por una capacidad de aplicación global para poder trabajar en diversos entornos y atraer a un público más amplio. Los usuarios, por su parte, desean una interfaz centrada en sus necesidades concretas. Esta tensión puede provocar problemas en los límites de nuestros sistemas.

Analicemos `java.util.Map` como ejemplo. Como puede apreciar en la siguiente lista. `Map` tiene una amplia interfaz con numerosas prestaciones. Esta potencia y flexibilidad es muy útil, pero también puede ser un problema. Por ejemplo, nuestra aplicación puede generar un `Map` y compartirlo. Nuestra intención puede que sea que ninguno de los receptores del mapa borre sus elementos. Pero en la parte superior de la lista encontramos el método `clear()`. Cualquier usuario del mapa puede borrarlo. O puede que nuestra convención de diseño determine que sólo se puedan almacenar objetos concretos en el mapa, pero `Map` no limita de forma fiable los tipos de objetos que admite. Cualquier usuario puede añadir elementos de cualquier tipo a cualquier mapa.

- `clear() void - Map`
- `containsKey (Object key) boolean - Map`
- `containsValue (Object value) boolean - Map`
- `entrySet() Set - Map`
- `equals(Object o) boolean - Map`
- `get(Object key) Object - Map`
- `getClass() Class<? extends Object> - Object`
- `hashCode() int - Map`
- `isEmpty() boolean - Map`
- `keySet() Set - Map`
- `notify() void - Object`
- `notifyAll() void - Object`
- `put(Object key, Object value) Object - Map`
- `putAll(Map t) void - Map`
- `remove(Object key) Object - Map`
- `size() int - Map`

- `toString() String - Object`
- `values() Collection - Map`
- `wait() void - Object`
- `wait(long timeout) void - Object`
- `wait(long timeout, int nanos) void - Object`

Figura 8.1. Los métodos de `Map`

Si nuestra aplicación necesita un mapa de `Sensor`, comprobará que los sensores se definen de esta forma:

```
Map sensors = new HashMap();
```

Tras ello, cuando otra parte del código necesite acceder a sensor, vemos este código:

```
Sensor s = (Sensor)sensors.get(sensorId);
```

No lo vemos una sola vez, sino repetidamente a lo largo del código. El cliente de este código es responsable de obtener un objeto de `Map` y convertirlo al tipo correcto. Funciona, pero no es código limpio. Además, este código no cuenta su historia como debería. La legibilidad del código se podría mejorar mediante el uso de genéricos, como se indica a continuación:

```
    Map<Sensor> sensors = new HashMap<Sensor>();
…
    Sensor s = sensors.get(sensorId);
```

Sin embargo, esto no soluciona el problema de que `Map<Sensor>` ofrezca más prestaciones de las que necesitamos o deseamos.

Al pasar una instancia de `Map<Sensor>` en el sistema, significa que habrá muchos puntos que corregir si la interfaz de `Map` cambia. Seguramente piense que son cambios improbables, pero recuerde que se han producido al añadir compatibilidad con genéricos en Java 5. Sin duda hemos visto sistemas que impiden el uso de genéricos debido a la gran cantidad de cambios necesarios para compensar el uso liberal de `Map`.

Una forma más limpia de usar `Map` sería la siguiente. A ningún usuario `Sensor` le importa si se usan genéricos o no. Esa opción se ha convertido (y siempre debería serlo) en un detalle de implementación.

```
public class Sensors {
    private Map sensors = new HashMap();

    public Sensor getById(String id) {
        return (Sensor) sensors.get(id);
    }
    //corte
}
```

La interfaz en el límite (`Map`) está oculta. Ha conseguido evolucionar sin apenas impacto en el resto de la aplicación. El uso de genéricos ya no es un problema ya que la conversión y la administración de tipos se procesa dentro de la clase `Sensors`.

Esta interfaz también se ha ajustado y limitado a las necesidades de la aplicación. Genera código más fácil de entender y con menor probabilidad de errores. La clase `Sensors` puede aplicar las reglas empresariales y de diseño.

No sugerimos que se encapsulen de esta forma todos los usos de Map, sino que no se pase `Map` (ni otras interfaces en el límite) por el sistema. Si usa una interfaz de límite como `Map`, manténgala dentro de la clase o la familia de clases en la que se use. Evite devolverla o aceptarla como argumento de API públicas.

Explorar y aprender límites

El código de terceros nos permite obtener mayor funcionalidad en menos tiempo. ¿Por dónde empezamos cuando queremos utilizar un paquete de terceros? Nuestra labor no es probar el código, pero sí crear pruebas para el código de terceros que utilicemos.

Imagine que no es evidente cómo usar una biblioteca de terceros. Podríamos perder uno o varios días en leer la documentación y decidir cómo usarla. Tras ello, podríamos escribir el código para usar el código de terceros y comprobar si se comporta de la forma esperada. No deberíamos sorprendernos por tener que realizar extensas sesiones de depuración intentando localizar errores en nuestro código o en el suyo.

Aprender el código de terceros es complicado, y también integrarlo. Hacer ambas cosas al mismo tiempo es el doble de complicado. Necesitamos un enfoque diferente. En lugar de experimentar y probar el nuevo material en nuestro código de producción, podríamos crear pruebas que analicen nuestro entendimiento del código de terceros. Jim Newkirk las denomina *pruebas de aprendizaje*[34].

En las pruebas de aprendizaje, invocamos la API de terceros como supuestamente la usaríamos en nuestra aplicación. Básicamente realizamos experimentos controlados para comprobar si la entendemos. Las pruebas se centran en lo que queremos obtener de la API.

Aprender `log4j`

Imagine que desea usar el paquete de Apache `log4j` en lugar de su propio dispositivo de registro personalizado. Lo descarga y abre la página inicial de la documentación. Sin una lectura exhaustiva, crea el primer caso de prueba con la esperanza de que escriba `hello` en la consola.

```
@Test
public void testLogCreate() {
    Logger logger = Logger.getLogger(“MyLogger”);
    logger.info(“hello”);
}
```

Al ejecutarlo, el registrador genera un error que nos indica que necesitamos algo denominado `Appender`. Tras investigar, descubrimos que existe un elemento `ConsoleAppender`. Creamos `ConsoleAppender` y comprobamos si hemos conseguido revelar los secretos del registro en la consola.

```
@Test
public void testLogAddAppender() {
    Logger logger = Logger.getLogger (“MyLogger”);
    ConsoleAppender appender = new ConsoleAppender();
    logger.addAppender(appender);
    logger.info(“hello”);
}
```

En esta ocasión descubrimos que `Appender` carece de flujo de salida, algo extraño, ya que parece lógico que lo tuviera. Tras recurrir a Google, probamos lo siguiente:

```
@Test
public void testLogAddAppender() {
    Logger logger = Logger.getLogger(“MyLogger”);
    logger.removeAllAppenders();
    logger.addAppender(new ConsoleAppender(
        new PatternLayout(“%p %t %m%n”),
        ConsoleAppender.SYSTEM_OUT));
    logger.info(“hello”);
}
```

Funciona; en la consola aparece un mensaje con la palabra `hello`. Resulta extraño tener que indicarle a `ConsoleAppender` que escriba en la consola.

Al eliminar el argumento `ConsoleAppender.SystemOut`, vemos que `hello` sigue impreso. Pero al eliminar `PatternLayout`, de nuevo vemos la queja de la falta de un flujo de salida. Es un comportamiento muy extraño.

Si nos fijamos en la documentación, vemos que el constructor `ConsoleAppender` predeterminado no está configurado, lo que no parece demasiado obvio ni útil. Parece más bien un error o una incoherencia de `log4j`.

Tras nuevas búsquedas en Google, investigaciones y pruebas, conseguimos el Listado 8-1. Hemos descubierto cómo funciona `log4j` y

hemos codificado esos conocimientos en un grupo de sencillas pruebas de unidad.

Listado 8-1
LogTest.java.

```
public class LogTest (
    private Logger logger;

    @Before
    public void initialize() {
        logger = Logger.getLogger(“logger”);
        logger.removeAllAppenders();
        Logger.getRootLogger().removeAllAppenders();
    }
    @Test
    public void basicLogger() {
        BasicConfigurator.configure();
        logger.info(“basicLogger”);
    }

    @Test
    public void addAppenderWithStream() {
        logger.addAppender(new ConsoleAppender(
            new PatternLayout (“%p %t %m%n”),
            ConsoleAppender.SYSTEM_OUT));
        logger.info(“addAppenderWithStream”);
    }

    @Test
    public void addAppenderWithoutStream() {
        logger.addAppender(new ConsoleAppender(
            new PatternLayout(“%p %t %m%n”)));
        logger.info(“addAppenderWithoutStream”);
    }
}
```

Ahora sabemos cómo inicializar un sencillo registrador de consola y encapsular ese conocimiento en nuestra propia clase de registro para que el resto de la aplicación se aísle de la interfaz de límite `log4j`.

Las pruebas de aprendizaje son algo más que gratuitas

Las pruebas de aprendizaje no cuestan nada. De todas formas, hemos tenido que aprender la API y crear las pruebas fue una forma sencilla y aislada de adquirir esos conocimientos. Las pruebas de aprendizaje fueron experimentos precisos que permitieron aumentar nuestros conocimientos.

Las pruebas no sólo son gratuitas, sino también rentables. Cuando aparezcan nuevas versiones del paquete de terceros, ejecutamos las pruebas de aprendizaje para comprobar si hay diferencias de comportamiento.

Las pruebas de aprendizaje demuestran que los paquetes de terceros que usamos funcionan de la forma esperada. Una vez integrados, no hay garantía de que el código de terceros sea compatible con nuestras

necesidades. Los autores originales se verán presionados para cambiar el código y ajustarlo a sus propias necesidades. Corregirán errores y añadirán nuevas funciones. En cada versión surgirán nuevos riesgos. Si el paquete de terceros cambia de una forma incompatible con nuestras pruebas, lo sabremos al instante.

Independientemente de que necesite los conocimientos proporcionados por las pruebas de aprendizaje, un límite claro debe estar respaldado por un conjunto de pruebas que ejerciten la interfaz de la misma forma que hace el código de producción. Sin estas pruebas de límites para facilitar la transición, podríamos conservar la versión antigua más tiempo del necesario.

Usar código que todavía no existe

Existe otro tipo de límite, que separa lo conocido de lo desconocido. En ocasiones, nuestro conocimiento del código parece desvanecerse. Lo que hay al otro lado del límite es desconocido (al menos por el momento). En ocasiones, decidimos no mirar más allá del límite.

Hace años formé parte de un equipo de desarrollo de *software* para un sistema de comunicación por radio. Había un subsistema, el Transmisor, que apenas conocíamos y cuya interfaz todavía no se había diseñado. Como no queríamos quedarnos parados, comenzamos a trabajar alejándonos de la parte desconocida del código.

Sabíamos perfectamente dónde acababa nuestro mundo y comenzaba el nuevo. Mientras avanzábamos, en ocasiones nos topábamos con este límite. Aunque la ignorancia ocultaba nuestra visión más allá del límite, sabíamos cómo queríamos que fuera la interfaz. Queríamos decirle al transmisor algo como lo siguiente:

> *Ajustar el transmisor en la frecuencia proporcionada y emitir una representación analógica de los datos que provienen de este flujo.*

No sabíamos cómo hacerlo ya que todavía no se había diseñado la API. Por ello decidimos determinar después los detalles.

Para no quedarnos bloqueados, definimos nuestra propia interfaz. Le dimos un nombre sencillo, `Transmitter`. Le asignamos el método `transmit` que aceptaba una frecuencia y un flujo de datos. Es la interfaz que deseábamos haber tenido.

Lo mejor de escribir la interfaz que deseábamos haber tenido era que la controlábamos. Esto hace que el código cliente sea más legible y se ciña a los objetivos previstos.

En la figura 8.1 se aprecia que aislamos las clases `CommunicationsController` de la API del transmisor (que no controlábamos y estaba por definir). Al usar nuestra propia interfaz específica de la aplicación, el código de `CommunicationsController` era limpio y expresivo. Una vez definida la API del transmisor, creamos `TransmitterAdapter` para reducir las distancias. El adaptador[35] encapsulaba la interacción con la API y ofrecía un único punto en el que evolucionaba.

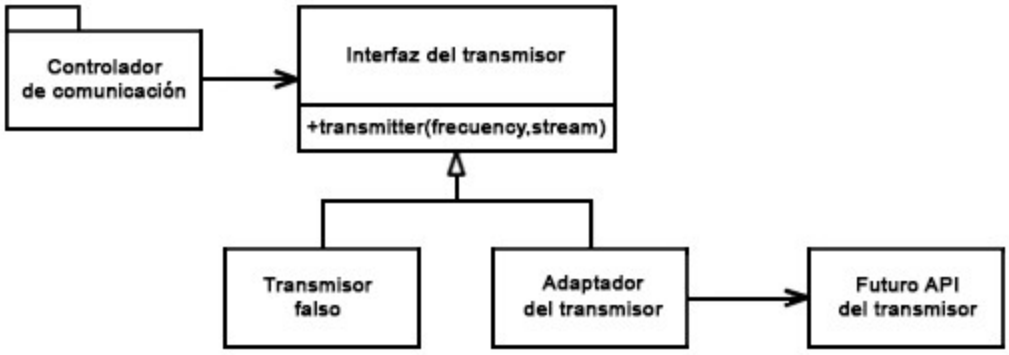


Figura 8.1. Predicción del transmisor

Este diseño también nos ofrece un sello[36] en el código para realizar pruebas. Con un elemento `FakeTransmitter`, podemos probar las clases `CommunicationsController`. También podemos crear pruebas de límite una vez diseñada la API `Transmitter` para asegurarnos de que la utilizamos correctamente.

Límites limpios

En los límites suceden cosas interesantes. Los cambios es una de ellas. Los diseños de código correctos acomodan los cambios sin necesidad de grandes modificaciones. Cuando usamos código que no controlamos, hay

que prestar especial atención a proteger nuestra inversión y asegurarnos de que los cambios futuros no son demasiado costosos. El código en los límites requiere una separación evidente y pruebas que definan expectativas. Debemos evitar que el código conozca los detalles de terceros. Es más aconsejable depender de algo que controlemos que de algo que no controlemos, y menos todavía si nos controla. Los límites de terceros se gestionan gracias a la presencia de puntos mínimos en el código que hagan referencia a los mismos. Podemos envolverlos como hicimos con `Map` o usar un adaptador para convertir nuestra interfaz perfecta en la interfaz proporcionada. En cualquier caso, el código se lee mejor, promueve el uso coherente e interno en el límite y hay menos puntos de mantenimiento cuando cambie el código de terceros.

Bibliografía

- **[BeckTDD]**: *Test Driven Development*, Kent Beck, Addison-Wesley, 2003.
- **[GOF]**: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison Wesley, 19%.
- **[WELC]**: *Working Effectively with Legacy Code*, Addison-Wesley, 2004.

9
Pruebas de unidad

Nuestra profesión ha evolucionado mucho en los últimos 10 años. En 1997 nadie había oído hablar del Desarrollo guiado por pruebas (DGP). Para la mayoría, las pruebas de unidad eran pequeños fragmentos de código desechable que creábamos para asegurarnos de que nuestros programas funcionaban. Escribíamos clases y métodos, y después código *ad hoc* para probarlos, lo que implicaba algún tipo de programa controlador que nos permitiera interactuar manualmente con el programa que habíamos escrito.

Recuerdo crear un programa de C++ para un sistema incrustado de tiempo real a mediados de la década de 1990. El programa era un sencillo temporizador con la siguiente firma:

```
void Timer::ScheduleCommand(Command* theCommand, int milliseconds)
```

La idea era sencilla; el método `execute` de `Command` se ejecutaba en un nuevo subproceso tras el número especificado de milisegundos. El problema era cómo probarlo. Confeccioné un sencillo programa controlador que escuchaba al teclado. Cada vez que se introducía un carácter, se programaba un comando que escribía el mismo carácter cinco segundos después. Introduje una rítmica melodía en el teclado y esperé a que se reprodujera en pantalla cinco segundos después:

> *«I… want-a-girl… just… like-the-girl-who-marr… ied… dear… old… dad.»*

Incluso tarareé la melodía mientras pulsaba la tecla, y la volví a cantar cuando aparecieron los puntos en la pantalla.

Ésa fue mi prueba. Cuando vi que funcionaba y se lo mostré a mis compañeros, me deshice del código de prueba.

Como he afirmado, nuestra profesión ha evolucionado mucho. Ahora crearía una prueba que garantizara el funcionamiento de hasta el más mínimo detalle del código. Aislaría el código del sistema operativo en lugar de invocar las funciones estándar de temporización. Las imitaría para tener control total sobre el tiempo. Programaría comandos que definieran indicadores Booleanos y avanzaría el tiempo, para observar los indicadores y asegurarme de que pasaran de `false` a `true` al cambiar el tiempo al valor correcto. Cuando superara una serie de pruebas, comprobaría que fueran adecuadas para todo el que tuviera que trabajar con el código. Me aseguraría de comprobar las pruebas y el código en el mismo paquete. Sí, hemos avanzado mucho, pero nos queda mucho por avanzar. Los movimientos Agile y TDD han animado a muchos programadores a crear pruebas de unidad automatizadas y cada vez son más. Pero en esta alocada carrera por añadir pruebas a nuestra disciplina, muchos programadores han pasado por alto dos de los aspectos más sutiles e importantes de diseñar pruebas de calidad.

Las tres leyes del DGP

Todos sabemos que el DGP nos pide que primero creemos las pruebas de unidad, antes que el código de producción. Pero esa norma es sólo la punta del iceberg. Tenga en cuenta las tres siguientes leyes[37]:

- **Primera ley**: No debe crear código de producción hasta que haya creado una prueba de unidad que falle.
- **Segunda ley**: No debe crear más de una prueba de unidad que baste como fallida, y no compilar se considera un fallo.
- **Tercera ley**: No debe crear más código de producción que el necesario para superar la prueba de fallo actual.

Estas tres leyes generan un ciclo de unos 30 segundos de duración. Las pruebas y el código de producción se crean de forma conjunta, las pruebas unos segundos antes que el código. Si trabajamos de esta forma, crearemos decenas de pruebas al día, cientos al mes y miles al año. Si trabajamos de esta forma, las pruebas abarcarán todos los aspectos de nuestro código de producción. El tamaño de dichas pruebas, que puede ser similar al del código de producción, puede suponer un problema de administración.

Realizar pruebas limpias

Hace unos años me pidieron que dirigiera un equipo que había decidido explícitamente que su código de prueba no debía mantenerse con los mismos estándares de calidad que su código de producción. Podían incumplir las reglas en sus pruebas de unidad. La premisa era «Rápido y directo». No era necesario que las variables tuvieran nombres adecuados, ni que las funciones de prueba fueran breves y descriptivas. No era necesario que el código de prueba estuviera bien diseñado. Bastaba con que funcionara y abarcara el código de producción.

Puede que algunos lectores empaticen con esta decisión. Puede que en el pasado creara el tipo de pruebas que cree para la clase `Timer`. Supone un gran paso crear ese tipo de pruebas desechables a diseñar una *suite* de

pruebas de unidad automatizadas. Por ello, como el equipo que dirigía, puede decidir que pruebas incorrectas sea mejor que no tener pruebas.

Pero el equipo no se daba cuenta que tener pruebas incorrectas era igual o peor que no tener prueba alguna. El problema es que las pruebas deben cambiar de acuerdo a la evolución del código. Cuanto menos limpias sean, más difícil es cambiarlas. Cuando más enrevesado sea el código de prueba, más probabilidades de que dedique más tiempo a añadir nuevas pruebas a la *suite* que el empleado en crear el nuevo código de producción. Al modificar el código de producción, las pruebas antiguas comienzan a fallar y el desastre impide que las pruebas se superen, por lo que acaban por convertirse en un obstáculo interminable.

Entre versiones, aumentó el coste de mantener la *suite* de pruebas de mi equipo. Acabó por convertirse en la principal queja entre los desarrolladores. Cuando los directores preguntaron sobre este aumento, los desarrolladores culparon a las pruebas. Al final, se vieron obligados a descartar la *suite* de pruebas completa.

Pero sin una *suite* de pruebas perdieron la posibilidad de garantizar el funcionamiento esperado de los cambios en el código. Sin una *suite* de pruebas no podían asegurar que los cambios en una parte del sistema no afectaran a otras diferentes. Los defectos aumentaron, lo que propició que temieran realizar cambios. Dejaron de limpiar su código de producción por miedo a que los cambios fueran dañinos. El código de producción comenzó a corromperse. Al final, se quedaron sin pruebas, con un código de producción enmarañado y defectuoso, clientes frustrados y la sensación de que su esfuerzo les había fallado.

En cierto modo tenían razón. Su esfuerzo les había fallado. Pero fue su decisión de permitir que las pruebas fueran incorrectas lo que provocó el fallo. Si hubieran empleado pruebas limpias, su esfuerzo no habría fallado. Puedo afirmarlo con cierta seguridad porque he participado y dirigido muchos equipos que han tenido éxito gracias a pruebas de unidad limpias.

La moraleja de la historia es sencilla: el código de prueba es tan importante como el de producción. No es un ciudadano de segunda. Requiere concentración, diseño y cuidado. Debe ser tan limpio como el código de producción.

Las pruebas propician posibilidades

Si sus pruebas no son limpias, las perderá. Y sin ellas pierde lo mismo que hace que su código de producción sea flexible. Sí, ha leído bien. Las pruebas de unidad son las que hacen que el código sea flexible y se pueda mantener y reutilizar. La razón es sencilla. Si tiene pruebas, no tendrá miedo a realizar cambios en el código. Sin pruebas, cada cambio es un posible error. Independientemente de la flexibilidad de su arquitectura, de la división del diseño, sin pruebas tendrá miedo a realizar cambios por la posibilidad de añadir errores no detectados.

Pero con las pruebas ese miedo desaparece. Cuanto mayor sea el alcance de sus pruebas, menos miedo tendrá. Podrá modificar el código con total impunidad, aunque su arquitectura no sea la mejor y el diseño sea mediocre. Podrá mejorar la arquitectura y el diseño sin miedo alguno.

Por tanto, disponer de una *suite* automatizada de pruebas de unidad que cubran el código de producción es la clave para mantener limpio el diseño y la arquitectura. Las pruebas proporcionan las posibilidades, ya que permiten el cambio.

Si sus pruebas no son limpias, la capacidad de modificar el código se verá limitada y perderá la posibilidad de mejorar la estructura de dicho código. Cuanto menos limpias sean las pruebas, menos lo será el código. En última instancia perderá las pruebas y el código se corromperá.

Pruebas limpias

¿Qué hace que una prueba sea limpia? Tres elementos: legibilidad, legibilidad y legibilidad. La legibilidad es sin duda más importante en las pruebas de unidad que en el código de producción. ¿Qué hace que una prueba sea legible? Lo mismo que en el código: claridad, simplicidad y densidad de expresión. En una prueba debe decir mucho con el menor número de expresiones posible.

Fíjese en el código de FitNesse del Listado 9-1. Estas tres pruebas son difíciles de entender y sin duda se pueden mejorar. Por un lado, hay mucho código duplicado [G5] en las invocaciones repetidas a `addPage` y `assertSubString`. Sobre todo, este código se carga con detalles que interfieren con la expresividad de la prueba.

Listado 9-1

SerializedPageResponderTest.java.

```
public void testGetPageHieratchyAsXml() throws Exception
{
    crawler.addPage(root, PathParser.parse(“PageOne”));
    crawler.addPage(root, PathParser.parse(“PageOne.ChildOne”));
    crawler.addPage(root, PathParser.parse(“PageTwo”));


    request.setResource(“root”);
    request.addInput(“type”, “pages”);
    Responder responder = new SerializedPageResponder();
    SimpleResponse response =
        (SimpleResponse) responder.makeResponse(
            new FitNesseContext(root), request);
    String xml = response.getContent();


    assertEquals(“text/xml”, response.getContentType());
    assertSubString(“<name>PageOne</name>”, xml);
    assertSubString(“<name>PageTwo</name>”, xml);
    assertSubString(“<name>ChildOne</name>”, xml);
}


public void testGetPageHieratchyAsXmlDoesntContainSymbolicLinks()
throws Exception
{
    WikiPage pageOne = crawler.addPage(root, PathParser.parse(“PageOne”));
    crawler.addPage(root, PathParser.parse(“PageOne.ChildOne”));
    crawler.addPage(root, PathParser.parse(“PageTwo”));


    PageData data = pageOne.getData();
    WikiPageProperties properties = data.getProperties();
    WikiPageProperty symLinks = properties.set(SymbolicPage.PROPERTY_NAME);
    symLinks.set(“SymPage”, “PageTwo”);
    pageOne.commit(data);


    request.setResource(“root”);
    request.addInput(“type”, “pages”);
    Responder responder = new SerializedPageResponder();
    SimpleResponse response =
        (SimpleResponse) responder.makeResponse(
            new FitNesseContext(root), request);
    String xml = response.getContent();


    assertEquals(“text/xml”, response.getContentType());
    assertSubString(“<name>PageOne</name>”, xml);
    assertSubString(“<name>PageTwo</name>”, xml);
    assertSubString(“<name>ChildOne</name>”, xml);
    assertNotSubString(“SymPage”, xml);
}


public void testGetDataAsHtml() throws Exception
{
    crawler.addPage(root, PathParser.parse(“TestPageOne”), “test page”);


    request.setResource(“TestPageOne”);
    request.addInput(“type”, “data”);
    Responder responder = new SerializedPageResponder();
    SimpleResponse response =
        (SimpleResponse) responder.makeResponse(
            new FitNesseContext(root), request);
    String xml = response.getContent();


    assertEquals(“text/xml”, response.getContentType());
    assertSubString(“test page”, xml);
    assertSubString(“<Test”, xml);
}
```

Fíjese en las invocaciones de `PathParser`. Transforman cadenas en instancias de `PagePath` usadas por las arañas. Esta transformación es totalmente irrelevante para la prueba y únicamente complica su cometido. Los detalles circundantes a la creación del respondedor y la obtención y conversión de la respuesta también sobran. También la forma de crear la URL de solicitud a partir de un recurso y un argumento (contribuí a crear este código, por lo que tengo todo el derecho a criticarlo).

Al final, el código no se ha diseñado de forma legible. El lector se ve rodeado de miles de detalles que debe comprender antes de que las pruebas tengan sentido.

Fíjese ahora en las pruebas mejoradas del Listado 9-2. Hacen exactamente lo mismo, pero se han refactorizado de forma más clara y descriptiva.

Listado 9-2
SerializedPageResponderTest.java (refactorizado)

```
public void testGetPageHierarchyAsXml() throws Exception {
    makePages(“PageOne”, “PageOne.ChildOne”, “PageTwo”);

    submitRequest(“root”, “type:pages”);

    assertResponseIsXML();
    assertResponseContains(
        “<name>PageOne</name>”, “<name>PageTwo</name>”, “<name>ChildOne</name>”
    );
}

public void testSymbolicLinksAreNotInXmlPageHierarchy() throws Exception {
    WikiPage page = makePage(“PageOne”);
    makePages(“PageOne.ChildOne”, “PageTwo”);

    addLinkTo(page, “PageTwo”, “SymPage”);

    submitRequest(“root”, “type:pages”);

    assertResponseIsXML();
    assertResponseContains(
        “<name>PageOne</name>”, “<name>PageTwo</name>”, “<name>ChildOne</name>”
    );
    assertResponseDoesNotContain(“SymPage”);
}

public void testGetDataAsXml() throws Exception {
    makePageWithContent(“TestPageOne”, “test page”);

    submitRequest(“TestPageOne”, “type:data”);

    assertResponseIsXML();
    assertResponseContains(“test page”, “<Test”);
}
```

El patrón Generar-Operar-Comprobar[38] es evidente en la estructura de las pruebas. Cada una se divide claramente en tres partes. La primera crea los datos de prueba, la segunda opera en dichos datos y la tercera comprueba que la operación devuelva los resultados esperados.

Comprobará que se ha eliminado gran parte de los detalles molestos. Las pruebas son concisas y sólo usan los tipos de datos y funciones que realmente necesitan. Todo el que lea estas pruebas sabrá rápidamente para qué sirven y no se perderá entre detalles irrelevantes.

Lenguaje de pruebas específico del dominio

Las pruebas del Listado 9-2 ilustran la creación de un lenguaje específico del dominio para sus pruebas. En lugar de usar las API que los programadores emplean para manipular el sistema, creamos una serie de funciones y utilidades que usan dichas API y que facilitan la escritura y la lectura de las pruebas. Estas funciones y utilidades se convierten en una API especializada usada por las pruebas. Son un lenguaje de pruebas que los programadores usan personalmente para crear sus pruebas y para ayudar a los que después las lean.

Esta API de pruebas no se diseña con antelación, sino que evoluciona con la refactorización continuada del código de prueba. Al igual que refactorizamos el Listado 9-1 en el Listado 9-2, los programadores disciplinados refactorizan su código de prueba en versiones más sucintas y expresivas.

Un estándar dual

En un sentido, el equipo que mencionamos antes tenía razón. El código de la API de pruebas tiene un conjunto de estándares de ingeniería diferentes al código de producción. También tiene que ser sencillo, sucinto y expresivo, pero no tan eficaz como el código de producción. Después de todo, se ejecuta en un entorno de prueba, no de producción, y cada entorno tiene sus propias necesidades.

Fíjese en la prueba del Listado 9-3. La creé como parte de un prototipo de sistema de control medioambiental. Sin entrar en detalles, se aprecia que esta prueba comprueba que la alarma de baja temperatura, el calentador y el fuelle estén activados cuando la temperatura sea demasiado fría.

Listado 9-3
EnvironmentControllerTest.java

```
@Test
    public void turnOnLoTempAlarmAtThreashold() throws Exception {
        hw.setTemp(WAY_TOO_COLD);
        controller.tic();
        assertTrue(hw.heaterState());
        assertTrue(hw.blowerState());
        assertFalse(hw.coolerState());
        assertFalse(hw.hiTempAlarm());
        assertTrue(hw.loTempAlarm());
    }
```

Aquí hay muchos detalles. Por ejemplo, ¿para qué sirve la función `tic`? De hecho, la ignoraría mientras leemos esta prueba. Intente centrarse

en saber si está de acuerdo en que el estado final del sistema tiene que ver con que la temperatura sea demasiado baja.

Al leer la prueba, la vista tiene que cambiar entre el nombre del estado comprobado y el sentido del estado comprobado. Vemos `heaterState` y después la vista salta a `assertTrue`. Vemos `coolerState` y nos fijamos en `assertFalse`. Resulta tedioso y dificulta la lectura de la prueba.

He conseguido mejorar la legibilidad de la prueba transformándola en el Listado 9-4.

Listado 9-4
EnvironmentControllerTest.java (refactorizado)

```
@Test
    public void turnOnLoTempAlarmAtThreshold() throws Exception {
        wayTooCold();
        assertEquals("HBchL", hw.getState());
    }
```

Evidentemente, he ocultado el detalle de la función `tic` creando una función `wayTooCold`. Pero lo importante es la extraña cadena de `assertEquals`. Las mayúsculas significan activado y las minúsculas desactivado, y las letras siempre aparece en este orden: `{heater, blower, cooler, hi-temp-alarm, lo-temp-alarm}`.

Aunque prácticamente sea un incumplimiento de las reglas de asignación mental[39], en este caso parece apropiado. Una vez que conocemos el significado, la vista pasa por la cadena y podemos interpretar los resultados. La lectura de la prueba es casi un placer. Fíjese en el Listado 9-5 y compruebe con qué facilidad entiende las pruebas.

Listado 9-5
EnvironmentControllerTest.java (una selección mayor).

```
@Test
    public void turnOnCoolerAndBlowerIfTooHot() throws Exception {
        tooHot();
        assertEquals("hBChl", hw.getState());
    }

    @Test
    public void turnOnHeaterAndBlowerIfTooCold() throws Exception {
        tooCold();
        assertEquals("HBchl", hw.getState());
    }

    @Test
    public void turnOnHiTempAlarmAtThreshold() throws Exception {
        wayTooHot();
        assertEquals("hBCHL", hw.getState());
    }

    @Test
    public void turnOnLoTempAlarmAtThreshold() throws Exception {
        wayTooCold();
```

```
        assertEquals(“HBchL”, hw.getState());
    }
```

La función `getState` se reproduce en el Listado 9-6. No es un código muy eficaz. Para que lo sea, deberíamos haber usado `StringBuffer`.

Listado 9-6
MockControlHardware.java.

```
public String getState() {
    String state = “”;
    state += heater ? “H” : “h”;
    state += blower ? “B” : “b”;
    state += cooler ? “C” : “c”;
    state += hiTempAlarm ? “H” : “h”;
    state += loTempAlarm ? “L” : “l”;
    return state;
}
```

`StringBuffer` es poco atractivo. Incluso en código de producción, intento evitarlo si el coste es mínimo, como podría suceder en el Listado 9-6. Pero esta aplicación es claramente un sistema incrustado en tiempo real y es probable que los recursos del equipo y la memoria estén limitados. Sin embargo, el entorno de pruebas es improbable que lo esté. Es la naturaleza del estándar dual. Hay cosas que nunca haría en un entorno de producción totalmente válidas para un entorno de prueba. Suelen ser problemas de memoria o eficacia de la CPU, pero nunca problemas de limpieza.

Una afirmación por prueba

Existe una escuela de pensamiento[40] que afirma que todas las funciones de prueba de una prueba JUnit sólo deben tener una instrucción de afirmación. Puede parecer una regla draconiana pero la ventaja se aprecia en el Listado 9-5. Las pruebas llegan a una misma conclusión, que se entiende de forma rápida y sencilla.

¿Pero qué sucede con el Listado 9-2? No parece razonable afirmar que el resultado es XML y que contiene determinadas subcadenas. Sin embargo, podemos dividir la prueba en dos, cada una con una afirmación concreta, como se muestra en el Listado 9-7.

Listado 9-7
SerializedPageResponderTest.java (una sola afirmación).

```
public void testGetPageHierarchyAsXml() throws Exception {
    givenPages(“PageOne”, “PageOne.ChildOne”, “PageTwo”);
```

```
    whenRequestIsIssued(“root”, “type:pages”);

    thenResponseShouldBeXML();
}

public void testGetPageHierarchyHasRightTags() throws Exception {
    givenPages(“PageOne”, “PageOne.ChildOne”, “PageTwo”);

    whenRequestIsIssued(“root”, “type:pages”);

    thenResponseShouldContain(
        “<name>PageOne</name>”, “<name>PageTwo</name>”, “<name>ChildOne</name>”
    );
}
```

He cambiado los nombres de las funciones para usar la convención *dado-cuando-entonces*[41]. De este modo las pruebas son más fáciles de leer. Desafortunadamente, al dividir las pruebas se genera código duplicado.

Podemos eliminar los duplicados por medio del patrón Método de plantilla[42] e incluir las partes *dado/cuando* en la clase base, y las partes entonces en derivaciones diferentes. O podríamos crear una clase de prueba independiente e incluir las partes *dado* y *cuando* en la función `@Before` y las partes entonces en cada función `@Test`. Pero parece un mecanismo excesivo para un problema tan menor. Al final, opto por las afirmaciones múltiples del Listado 9-2. Considero que la regla de una sola afirmación es una directriz adecuada[43]. Siempre intento crear un lenguaje de pruebas específico del dominio que la complemente, como en el Listado 9-5, pero no rechazo incluir más de una afirmación en una prueba. Creo que lo mejor que podemos decir es que el número de afirmaciones de una prueba debe ser mínimo.

Un solo concepto por prueba

Puede que una regla más indicada sea probar un único concepto en cada función de prueba. No queremos extensas funciones que prueben una cosa diferente tras otra, como sucede en el Listado 9-8. Esta prueba debería dividirse en tres diferentes que probaran tres cosas distintas. Al combinarlas en la misma función se obliga al lector a determinar por qué cada sección se ubica en ese punto y qué prueba dicha sección.

Listado 9-8

```
/**
* Varias pruebas para el método addMonths().
*/
public void testAddMonths() {
```

```
    SerialDate d1 = SerialDate.createInstance(31, 5, 2004);

    SerialDate d2 = SerialDate.addMonths(1, d1);
    assertEquals(30, d2.getDayOfMonth());
    assertEquals(6, d2.getMonth());
    assertEquals(2004, d2.getYYYY());

    SerialDate d3 = SerialDate.addMonths(2, d1);
    assertEquals(31, d3.getDayOfMonth());
    assertEquals(7, d3.getMonth());
    assertEquals(2004, d3.getYYYY());

    SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
    assertEquals(30, d4.getDayOfMonth());
    assertEquals(7, d4.getMonth());
    assertEquals(2004, d4.getYYYY());
}
```

Las tres funciones deberían ser las siguientes:

- Dado el último día de un mes con 31 días (como mayo):
  1. Cuando se añade un mes, si el último día de ese mes es el 30 (como en junio), entonces la fecha debe ser el día 30 de ese mes, no el 31.
  2. Cuando se añaden dos meses a esa fecha, si el último mes tiene 31 días, entonces la fecha debe ser el día 31.
- Dado el último día de un mes con 30 días (como junio):
  1. Cuando se añade, si el último día de ese mes tiene 31 días, entonces la fecha debe ser el 30, no el 31.

Expresado de esta forma, se aprecia que existe una regla general entre las distintas pruebas. Al incrementar el mes, la fecha no puede ser mayor que su último día. Esto implica que al incrementar el mes en el 28 de febrero debe generarse el 28 de marzo. Falta esa prueba y convendría que la escribiéramos.

Así pues, no son las múltiples afirmaciones del Listado 9-8 las causantes del problema, sino el hecho de que se prueba más de un concepto. Probablemente la regla óptima sea minimizar el número de activos por concepto y probar un solo concepto por función de prueba.

F.I.R.S.T.[44]

Las pruebas limpias siguen otras cinco reglas, cuyas iniciales forman las siglas FIRST en inglés:

Rapidez (*Fast*): Las reglas deben ser rápidas y ejecutarse de forma rápida. Si lo hacen lentamente, no las ejecutará con frecuencia. Al no

hacerlo, no detectará los problemas con la suficiente antelación como para solucionarlos. No se sentirá con libertad para limpiar el código, que acabará corrompiéndose.

Independencia (*Independent*): Las pruebas no deben depender entre ellas. Una prueba no debe establecer condiciones para la siguiente. Debe poder ejecutar cada prueba de forma independiente y en el orden que desee. Si las pruebas dependen unas de otras, la primera que falle provocará una sucesión de fallos, dificultará el diagnóstico y ocultará efectos posteriores.

Repetición (*Repeatable*): Las pruebas deben poder repetirse en cualquier entorno. Debe poder ejecutarlas en el entorno de producción, en el de calidad y en su portátil de camino a casa en un tren sin red. Si no puede repetir las pruebas en cualquier entorno, siempre tendrá una excusa de su fallo. También verá que no puede ejecutar las pruebas si el entorno no está disponible.

Validación automática (*Self-Validating*): Las pruebas deben tener un resultado booleano: o aciertan o fallan. No debe tener que leer un extenso archivo de registro para saber si una prueba ha acertado, ni comparar manualmente dos archivos de texto distintos para ello. Si las pruebas no se validan automáticamente, el fallo puede ser subjetivo y la ejecución de las pruebas puede requerir una extensa evaluación manual.

Puntualidad (*Timely*): Las pruebas deben crearse en el momento preciso: antes del código de producción que hace que acierten. Si crea las pruebas después del código de producción, puede que resulte difícil probarlo. Puede decidir qué parte del código de producción sea demasiado difícil de probar. No diseñe código de producción que no se pueda probar.

Conclusión

Apenas hemos abordado la superficie de este tema. De hecho, se podría crear un libro entero sobre pruebas limpias. Las pruebas son tan importantes para la salud de un proyecto como el código de producción. Puede que incluso más, ya que conservan y mejoran la flexibilidad, capacidad de mantenimiento y reutilización del código de producción. Por ello, intente que sean limpias. Trabaje para que resulten expresivas y concisas. Invente API de prueba que actúen como lenguaje específico del dominio que le ayude a crear las pruebas.

Si deja que las pruebas se corrompan, sucederá lo mismo con el código de producción. Mantenga limpias las pruebas.

Bibliografía

- **[RSpec]**: *RSpec: Behavior Driven Development for Ruby Programmers*, Aslak Hellesay, David Chelimsky, Pragmatic Bookshelf, 2008.
- **[GOF]**: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

10
Clases

con Jeff Langr

Hasta ahora nos hemos centrado en escribir bien líneas y bloques de código. Nos hemos adentrado en la correcta composición de las funciones y en su interrelación. Pero a pesar de la atención dedicada a la expresividad de las instrucciones y las funciones, no tendremos código limpio hasta que nos fijemos en los niveles superiores de su organización. Hablemos sobre clases.

Organización de clases

De acuerdo a la convención estándar de Java, una clase debe comenzar con una lista de variables. Las constantes estáticas públicas, si existen, deben aparecer primero. Tras ello, las variables estáticas privadas y después las variables de instancia privadas. No suele ser necesario usar variables públicas.

Las funciones públicas deben seguir a la lista de variables. Incluimos las utilidades públicas invocadas por una función pública tras la propia función pública. Este sistema cumple la regla descendente y permite que el programa se lea como un artículo de periódico.

Encapsulación

Queremos que nuestras variables y funciones de utilidad sean privadas, pero no es imprescindible. En ocasiones podemos proteger una variable o función de utilidad para que sea accesible para una prueba. Las reglas mandan. Si una regla del mismo paquete tiene que invocar una función o acceder a una variable, hacemos que tenga ámbito `protected` o de paquete. Sin embargo, primero veremos una forma de mantener la privacidad. La relajación de la encapsulación siempre es un último resorte.

Las clases deben ser de tamaño reducido

La primera regla de las clases es que deben ser de tamaño reducido. La segunda regla es que deben ser todavía más reducidas. No, no vamos a repetir el mismo texto en el capítulo sobre las funciones, pero como sucede con las funciones, el tamaño reducido es lo principal a la hora de diseñar una clase. Y la pregunta inmediata es qué nivel de reducción. Con las funciones medimos el tamaño contando líneas físicas. Con las clases usamos otra medida distinta: las responsabilidades[45].

El Listado 10-1 muestra una clase, `SuperDashboard`, que muestra 70 métodos públicos. Muchos programadores estarán de acuerdo en que es un tamaño excesivo. Algunos denominarían a `SuperDashboard` una clase Dios.

Listado 10-1
Demasiadas responsabilidades.

```
public class SuperDashboard extends JFrame implements 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()
    public void showObject(MetaObject object)
    public void showProgress(String s)
    public boolean isMetadataDirty()
    public void setIsMetadataDirty(boolean isMetadataDirty)
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public void setMouseSelectState(boolean isMouseSelected)
    public boolean isMouseSelected()
    public LanguageManager getLanguageManager()
    public Project getProject()
    public Project getFirstProject()
    public Project getLastProject()
    public String getNewProjectName()
    public void setComponentSizes(Dimension dim)
    public String getCurrentDir()
    public void setCurrentDir(String newDir)
    public void updateStatus(int dotPos, int markPos)
    public Class[] getDataBaseClasses()
    public MetadataFeeder getMetadataFeeder()
    public void addProject(Project project)
    public boolean setCurrentProject(Project project)
    public boolean removeProject(Project project)
    public MetaProjectHeader getProgramMetadata()
    public void resetDashboard()
    public Project loadProject(String fileName, String projectName)
    public void setCanSaveMetadata(boolean canSave)
    public MetaObject getSelectedObject()
    public void deselectObjects()
    public void setProject(Project project)
    public void editorAction(String actionName, ActionEvent event)
    public void setMode(int mode)
    public FileManager getFileManager()
    public void setFileManager(FileManager fileManager)
    public ConfigManager getConfigManager()
    public void setConfigManager(ConfigManager configManager)
    public ClassLoader getClassLoader()
    public void setClassLoader(ClassLoader classLoader)
    public Properties getProps()
    public String getUserHome()
    public String getBaseDir()
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
    public MetaObject pasting(
        MetaObject target, MetaObject pasted, MetaProject project)
    public void processMenuItems(MetaObject metaObject)
    public void processMenuSeparators(MetaObject metaObject)
    public void processTabPages(MetaObject metaObject)
    public void processPlacement(MetaObject object)
    public void processCreateLayout(MetaObject object)
    public void updateDisplayLayer(MetaObject object, int layerIndex)
    public void propertyEditedRepaint(MetaObject object)
    public void processDeleteObject(MetaObject object)
    public boolean getAttachedToDesigner()
    public void processProjectChangedState(boolean hasProjectChanged)
    public void processObjectNameChanged(MetaObject object)
    public void runProject()
    public void setAllowDragging(boolean allowDragging)
    public boolean allowDragging()
    public boolean isCustomizing()
    public void setTitle(String title)
    public IdeMenuBar getIdeMenuBar()
    public void showHelper(MetaObject metaObject, String propertyName)
    //… y otros muchos métodos no públicos…
}
```

¿Y si SuperDashboard sólo incluyera los métodos mostrados en el Listado 10-2?

Listado 10-2
¿Suficientemente reducido?

```
public class SuperDashboard extends JFrame implements MetaDataUser
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
```

```
}
```

Cinco métodos no es demasiado, ¿verdad? En este caso sí ya que a pesar del reducido número de métodos, `SuperDashboard` también tiene demasiadas responsabilidades.

El nombre de una clase debe describir las responsabilidades que desempeña. De hecho, el nombre es la primera forma para determinar el tamaño de una clase. Si no podemos derivar un nombre conciso para una clase, seguramente sea demasiado extenso. Cuanto más ambiguo sea el nombre de la clase, más probabilidades hay de que tenga demasiadas responsabilidades. Por ejemplo, los nombres de clase con palabras como `Processor`, `Manager` o `Super` suelen indicar una desafortunada acumulación de responsabilidades.

También debemos ser capaces de escribir una breve descripción de la clase en unas 25 palabras, sin usar las palabras «`si`», «`o`», «`y`» o «`pero`». ¿Cómo describiríamos `SuperDashboard`?: `SuperDashboard` permite acceder al componente con el enfoque y nos permite controlar los números de versión y producto. El primer `y` indica que `SuperDashboard` tiene demasiadas responsabilidades.

El Principio de responsabilidad única

El Principio de responsabilidad única (*Single Responsibility Principle*, SRP) [46] indica que una clase o módulo debe tener uno y sólo un motivo para cambiar. Este principio nos indica la definición de responsabilidad y una directriz para el tamaño de la clase. Las clases sólo deben tener una responsabilidad, un motivo para cambiar. La clase `SuperDashboard` aparentemente reducida del Listado 10-2 tiene dos motivos para cambiar. Primero, controla información de versión que supuestamente debe actualizarse cada vez que se comercialice el *software*. Por otra parte, gestiona componentes de Java Swing (un derivado de JFrame, la representación Swing de una ventana de IGU de nivel superior). Sin duda, querremos cambiar el número de versión si cambiamos el código Swing, pero lo contrario no es necesario: podríamos cambiar la información de versión en función de los cambios de otro código del sistema. La identificación de responsabilidades (los motivos del cambio) nos permite reconocer y mejorar las abstracciones en nuestro código. Podemos extraer los tres métodos de `SuperDashboard` relacionados con la información de

versiones en una clase independiente como `Version` (véase el Listado 10-3.) La clase `Version` es una construcción que se puede reutilizar en otras aplicaciones.

Listado 10-3
Una clase con una única responsabilidad.

```
public class Version {
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

SRP es uno de los conceptos más importantes del diseño orientado a objetos y también uno de los más sencillos de entender y cumplir, pero también es uno de los que más se abusa al diseñar clases. Habitualmente nos encontramos clases que hacen demasiadas cosas. ¿Por qué?

Crear *software* que funcione y crear *software* limpio son dos actividades diferentes. Muchos tenemos un cerebro limitado, de modo que nos centramos en que el código funcione más que en su organización y limpieza. Es algo totalmente válido. Mantener objetivos separados es tan importante en nuestras actividades de programación como en nuestros programas.

El problema es que muchos creemos que hemos terminado cuando el programa funciona. No cambiamos al *otro* objetivo de organización y limpieza. Pasamos al siguiente problema en lugar de retroceder y dividir las clases en unidades independientes con una única responsabilidad.

Al mismo tiempo, muchos programadores temen que un elevado número de pequeñas clases con un único propósito dificulten la comprensión del conjunto. Les preocupa que tengan que desplazarse entre las clases para determinar cómo funciona un aspecto concreto.

Sin embargo, un sistema con muchas clases reducidas no tiene más elementos móviles que un sistema con algunas clases enormes. En ambos hay que entender lo mismo. La pregunta es si quiere organizar sus herramientas en cajas con muchos pequeños cajones que contengan componentes bien definidos y etiquetados, o usar varios cajones grandes en los que mezcle todo.

Todos los sistemas tienen una gran lógica y complejidad. El objetivo principal para gestionar dicha complejidad es organizarla para que un programador sepa dónde buscar y comprenda la complejidad directamente afectada en cada momento concreto. Por el contrario, un sistema con clases

multipropósito de mayor tamaño nos obliga a buscar entre numerosos elementos que no siempre necesitamos conocer.

Para reformular los puntos anteriores, diremos que los sistemas deben estar formados por muchas claves reducidas, no por algunas de gran tamaño. Cada clase reducida encapsula una única responsabilidad, tiene un solo motivo para cambiar y colabora con algunas otras para obtener los comportamientos deseados del sistema.

Cohesión

Las clases deben tener un número reducido de variables de instancia. Los métodos de una clase deben manipular una o varias de dichas variables. Por lo general, cuantas más variables manipule un método, más cohesión tendrá con su clase. Una clase en la que cada variable se usa en cada método tiene una cohesión máxima.

Por lo general, no es recomendable ni posible crear este tipo de clases pero queremos que la cohesión de nuestras clases sea elevada. Si lo logramos, significa que los métodos y variables de la clase dependen unos de otros y actúan como un todo lógico.

Fíjese en la implementación de `Stack` en el Listado 10-4. Es una clase muy consistente. De los tres métodos, sólo `size()` no usa ambas variables.

Listado 10-4
Stack.java, una clase consistente.

```
public class Stack {
    private int topOfStack = 0;
    List<Integer> elements = new LinkedList<Integer>();


    public int size() {
        return topOfStack;
    }


    public void push(int element) {
        topOfStack++;
        elements.add(element);
    }


    public int pop() throws PoppedWhenEmpty {
        if (topOfStack == 0)
            throw new PoppedWhenEmpty();
        int element = elements.get(--topOfStack);
        elements.remove(topOfStack);
        return element;
    }
}
```

La estrategia de reducir el tamaño de las funciones y de las listas de parámetros suele provocar la proliferación de variables de instancia usadas

por un subconjunto de los métodos. Si esto sucede, siempre existe al menos una clase que intenta huir de la clase de mayor tamaño. Debe intentar separar las variables y métodos en dos o más clases para que las nuevas sean más consistentes.

Mantener resultados consistentes en muchas clases de tamaño reducido

La división de grandes funciones en otras más pequeñas aumenta la proliferación de clases. Imagine una gran función con numerosas variables declaradas. Imagine que desea extraer una pequeña parte de esa función en otra independiente. Sin embargo, el código que extrae usa cuatro de las variables declaradas en la función. ¿Debe pasar las cuatro variables como argumentos a la nueva función?

En absoluto. Si ascendemos estas cuatro variables a variables de instancia de la clase, podremos extraer el código sin pasar las variables. Resultaría más sencillo dividir la función en pequeños fragmentos.

Desafortunadamente, eso significaría que nuestras clases perderían cohesión ya que acumularían más y más variables de instancia que sólo existen para que otras funciones las compartan. Pero un momento. Si apenas existen funciones que compartan determinadas variables, ¿no son entonces una clase con derecho propio? Por supuesto. Cuando las clases pierdan cohesión, divídalas.

Por tanto, dividir una gran función en otras más reducidas también nos permite dividir varias clases más reducidas. De este modo mejora la organización del programa y su estructura resulta más transparente. Como ejemplo, usaremos un ejemplo obtenido del libro de Knuth *Literate Programming*[47]. El Listado 10-5 muestra una traducción a Java del programa `PrintPrimes` de Knuth. Para hacerle justicia, no es el programa que creó sino el resultado generado por su herramienta WEB. Lo usamos aquí por ser un magnífico punto de partida para dividir una función de gran tamaño en varias funciones y clases más reducidas.

Listado 10-5
PrintPrimes.java

```
package literatePrimes;
```

```
public class PrintPrimes {
    public static void main(String[] args) {
        final int M = 1000;
        final int RR = 50;
        final int CC = 4;
        final int WW = 10;
        final int ORDMAX = 30;
        int P[] = new int[M + 1];
        int PAGENUMBER;
        int PAGEOFFSET;
        int ROWOFFSET;
        int C;
        int J;
        int K;
        boolean JPRIME;
        int ORD;
        int SQUARE;
        int N;
        int MULT[] = new int[ORDMAX + 1];


        J = 1;
        K = 1;
        P[1] = 2;
        ORD = 2;
        SQUARE = 9;


        while (K < M) {
            do {
                J = J + 2;
                if (J == SQUARE) {
                ORD = ORD + 1;
                SQUARE = P[ORD] * P[ORD];
                MULT[ORD - 1] = J;
                }
                N = 2;
                JPRIME = true;
                while (N < ORD && JPRIME) {
                while (MULT[N] < J)
                    MULT[N] = MULT[N] + P[N] + P[N];
                if (MULT[N] == J)
                    JPRIME = false;
                N = N + 1;
                }
            } while (!JPRIME);
            K = K + 1;
            P[K] = J;
        }
        {
            PAGENUMBER = 1;
            PAGEOFFSET = 1;
            while (PAGEOFFSET <= M) {
                System.out.println("The First " + M +
                    " Prime Numbers --- Page " + PAGENUMBER);
                System.out.println("");
                for (ROWOFFSET = PAGEOFFSET; ROWOFFSET < PAGEOFFSET + RR; ROWOFFSET++) {
                for (C = 0; C < CC;C++)
                    if (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;
            }
        }
    }
}
```

Este programa, escrito como una sola función, es un desastre. El sangrado de su estructura es excesivo y hay demasiadas variables extrañas. Como mínimo, la función debería dividirse en otras más pequeñas. Los listados del 10-6 al 10-8 muestran la división del código del Listado 10-5 en clases y funciones de menor tamaño, además de los nombres elegidos para dichas clases, funciones y variables.

Listado 10-6
PrimePrinter.java (refactorizado)

```
package literatePrimes;
```

```
public class PrimePrinter (
    public static void main(String[] args) {
        final int NUMBER_OF_PRIME5 = 1000;
        int[] primes = PrimeGenerator.generate(NUMBER_OF_PRIMES);


        final int ROWS_PER_PAGE = 50;
        final int COLUMNS_PER_PAGE = 4;
        RowColumnPagePrinter tablePrinter =
            new RowColumnPagePrinter(ROWS_PER_PAGE,
                        COLUMNS_PER_PAGE,
                        “The First ” + NUMBER_OF_PRIMES +
                        “ Prime Numbers”);
        tablePrinter.print(primes);
    }
}
```

Listado 10-7
RowColumnPagePrinter.java.

```
package literatePrimes;


import java.io.PrintStream;


public class RowColumnPagePrinter {
    private int rowsPerPage;
    private int columnsPerPage;
    private int numbersPerPage;
    private String pageHeader;
    private PrintStream printStream;


    public RowColumnPagePrinter(int rowsPerPage,
                        int columnsPerPage,
                        String pageHeader) {
        this.rowsPerPage = rowsPerPage;
        this.columnsPerPage = columnsPerPage;
        this.pageHeader = pageHeader;
        numbersPerPage = rowsPerPage * columnsPerPage;
        printStream = System.out;
    }


    public void print(int data[]) {
        int pageNumber = 1;
        for (int firstIndexOnPage = 0;
                 firstIndexOnPage < data.length;
                 firstIndexOnPage += numbersPerPage) {
            int lastIndexOnPage =
                Math.min(firstIndexOnPage + numbersPerPage - 1,
                         data.length - 1);
            printPageHeader(pageHeader, pageNumber);
            printPage(firstIndexOnPage, lastIndexOnPage, data);
            printStream.println(“\f”);
            pageNumber++;
        }
    }


    private void printPage (int firstIndexOnPage,
                        int lastIndexOnPage,
                        int[] data) {
        int firstIndexOfLastRowOnPage =
            firstIndexOnPage + rowsPerPage - 1;
        for (int firstIndexInRow = firstIndexOnPage;
                 firstIndexInRow <= firstIndexOfLastRowOnPage;
                 firstIndexInRow++) {
            printRow(firstIndexInRow, lastIndexOnPage, data);
            printStream.println(“”);
        }
    }


    private void printRow(int firstIndexInRow,
                        int lastIndexOnPage,
                        int[] data) {
        for (int column = 0; column < columnsPerPage; column++) {
            int index = firstIndexInRow + column * rowsPerPage;
            if (index <= lastIndexOnPage)
                printStream.format(“%10d”, data[index]);
        }
    }


    private void printPageHeader(String pageHeader,
                        int pageNumber) {
        printStream.println(pageHeader + “ --- Page ” + pageNumber);
        printStream.println(“”);
    }
```

```
    public void setOutput(PrintStream printStream) {
        this.printStream = printStream;
    }
}
```

Listado 10-8
PrimeGenerator.java

```
package literatePrimes;


import java.util.ArrayList;


public class PrimeGenerator {
    private static int[] primes;
    private static ArrayList<Integer> multiplesOfPrimeFactors;


    protected static int[] generate(int n) {
        primes = new int[n];
        multiplesOfPrimeFactors = new ArrayList<Integer>();
        set2AsFirstPrime();
        checkOddNumbersForSubsequentPrimes();
        return primes;
    }


    private static void set2AsFirtsPrime() {
        primes[0] = 2;
        multiplesOfPrimeFactors.add(2);
    }


    private static void checkOddNumbersForSubsequentPrimes() {
        int primeIndex = 1;
        for (int candidate = 3;
                 primeIndex < primes.length;
                 candidate += 2) {
            if (isPrime(candidate))
                 primes[primeIndex++] = candidate;
        }
    }


    private static boolean isPrime(int candidate) {
        if (isLeastRelevantMultipleOfNextLargerPrimeFactor(candidate)) {
            multiplesOfPrimeFactors.add(candidate);
            return false;
        }
        return isNotMultipleOfAnyPreviousPrimeFactor(candidate);
    }


    private static boolean
    isLeastRelevantMultipleOfNextLargerPrimeFactor(int candidate) {
        int nextLargerPrimeFactor = primes[multiplesOfPrimeFactors.size()];
        int leastRelevantMultiple = nextLargerPrimeFactor * nextLargerPrimeFactor;
        return candidate == leastRelevantMultiple;
    }


    private static boolean
    isNotMultipleOfAnyPreviousPrimeFactor(int candidate) {
        for (int n = 1; n < multiplesOfPrimeFactors.size(); n++) {
            if (isMultipleOfNthPrimeFactor(candidate, n))
                return false;
        }
        return true;
    }


    private static boolean
    isMultipleOfNthPrimeFactor(int candidate, int n) {
        return
            candidate == smallestOddNthMultipleNotLessThanCandidate(candidate, n);
    }


    private static int
    smallestOddNthMultipleNotLessThanCandidate(int candidate, int n) {
        int multiple = multiplesOfPrimeFactors.get(n);
        while (multiple < candidate)
            multiple += 2 * primes[n];
        multiplesOfPrimeFactors.set(n, multiple);
        return multiple;
    }
}
```

Lo primero que apreciará es que ha aumentado la longitud del programa, de una a casi tres páginas. Este aumento se debe a varios motivos. En primer lugar, el programa refactorizado usa nombres de variable más extensos y descriptivos. Por otra parte, usa declaraciones de funciones y clases como comentarios del código. Por último, usamos espacios en blanco y técnicas de formato para mantener la legibilidad.

El programa se ha dividido en tres responsabilidades principales. La parte principal se incluye en la clase `PrimePrinter`, responsable de controlar el entorno de ejecución. Cambia si se modifica el método de invocación. Por ejemplo, si este programa se convierte en un servicio SOA, es la clase que se verá afectada.

`RowColumnPagePrinter` sabe cómo aplicar formato a una lista de números con una determinada cantidad de filas y columnas. Si es necesario cambiar el formato del resultado, es la clase que se verá afectada.

La clase `PrimeGenerator` sabe cómo generar una lista de números primos. No se creará una instancia como objeto. La clase es sólo un ámbito útil en el que declarar y ocultar sus variables. Esta clase cambia si se modifica el algoritmo para calcular números primos. No hemos reescrito el programa. No hemos empezado de cero y los hemos vuelto a diseñar. En realidad, si se fija atentamente en los dos programas, verá que usan los mismos algoritmos y mecanismos.

El cambio se ha realizado creando una *suite* de pruebas que verifican el comportamiento preciso del primer programa. Tras ello, se aplican numerosos cambios mínimos, de uno en uno. Tras cada cambio, se ejecuta el programa para garantizar que el comportamiento no varía. Paso a paso, el primer programa se limpia y se transforma en el segundo.

Organizar los cambios

En muchos sistemas, el cambio es continuo. Cada cambio supone un riesgo de que el resto del sistema no funcione de la forma esperada. En un sistema limpio organizamos las clases para reducir los riesgos de los cambios.

La clase `Sql` del Listado 10-9 se usa para generar cadenas SQL de forma correcta con los metadatos adecuados. Es un trabajo continuo y, como tal, no admite funciones SQL como instrucciones `update`. Cuando la clase `Sql` tenga que admitir una instrucción `update`, tendremos que abrirla

para realizar modificaciones. El problema de abrir una clase es el riesgo que conlleva. Cualquier modificación puede afectar a otro código de la clase. Debe probarse concienzudamente.

Listado 10-9
Clase que debemos abrir para realizar cambios.

```
public class Sql {
    public Sql(String table, Column[] columns)
    public String create()
    public String insert(Object[] fields)
    public String selectAll()
    public String findByKey(String keyColumn, String keyValue)
    public String select(Column column, String pattern)
    public String select(Criteria criteria)
    public String preparedInsert()
    private String columnList(Column[] columns)
    private String valuesList(Object[] fields, final Column[] columns)
    private String selectWithCriteria(String criteria)
    private String placeholderList(Column[] columns)
}
```

La clase `Sql` debe cambiar al añadir un nuevo tipo de instrucción. También debe cambiar cuando variemos los detalles de un tipo de instrucción concreto; por ejemplo, si tenemos que modificar la funcionalidad `select` para admitir selecciones secundarias. Estos dos motivos de cambio significan que la clase `Sql` incumple SRP.

Podemos detectar este incumplimiento desde un punto de vista organizativo. El método `outline` de `Sql` muestra que hay métodos privados, como `selectWithCriteria`, que parecen relacionarse únicamente con instrucciones `select`.

El comportamiento de métodos privados aplicados a un pequeño subconjunto de una clase puede ser una heurística útil para detectar zonas que mejorar. Sin embargo, la verdadera razón debe ser el cambio del sistema. Si la clase `Sql` se considera totalmente lógica, no debemos preocuparnos por separar las responsabilidades. Si no necesitamos funcionalidad de actualización en el futuro, podemos olvidarnos de `Sql`. Pero si tenemos que abrir una clase, debemos corregir el diseño.

¿Y si optamos por una solución como la del Listado 10-10? Los métodos públicos de interfaz definidos en `Sql` en el Listado 10-9 se refactorizan en sus propias variantes de la clase `Sql`. Los métodos privados, como `valuesList`, se mueven directamente a las posiciones necesarias. El comportamiento privado se reduce a un par de clases de utilidad: `Where` y `ColumnList`.

Listado 10-10

Un grupo de clases cerradas.

```
Abstract public class Sql {
    public Sql(String table, Column[] columns)
    abstract public String generate();
}


public class CreateSql extends Sql {
    public CreateSql(String table, Column[] columns)
    @Override public String generate()
}


public class SelectSql extends Sql {
    public SelectSql(String table, Column[] columns)
    @Override public String generate()
}


public class InsertSql extends Sql {
    public SelectSql(String table, Column[] columns, Object[] fields)
    @Override public String generate()
    private String valuesList(Object[] fields, final Column[] columns)
}


public class SelectWithCriteriaSql extends Sql {
    public SelectWithCriteriaSql(
        String table, Column[] columns, Criteria criteria)
    @Override public String generate()
}


public class SelectWithMatchSql extends Sql {
    public SelectWithMatchSql(
        String table, Column[] columns, Column column, String pattern)
    @Override public String generate()
}


public class FindByKeySql extends Sql {
    public FindByKeySql(
        String table, Column[] columns, String keyColumn, String keyValue)
    @Override public String generate()
}


public class PreparedInsertSql extends Sql {
    public PreparedInsertSql(String table, Column[] columns)
    @Override public String generate()
    private String placeholderList(Column[] columns)
}


public class Where {
    public Where(String criteria)
    public String generate()
}


public class ColumnList {
    public ColumnList(Column[] columns)
    public String generate()
}
```

El código de cada clase se simplifica enormemente. El tiempo necesario para entender las clases se reduce al mínimo. El riesgo de que una función afecte a otra desaparece casi por completo. Desde el punto de vista de las pruebas, resulta más sencillo probar la lógica de esta solución, ya que las clases se aíslan unas de otras.

Además, cuando llegue el momento de añadir las instrucciones `update`, no cambia ninguna de las clases existentes. Añadimos la lógica para generar instrucciones `update` a una nueva subclase de `Sql`, `UpdateSql`. Este cambio no afecta a otro código del sistema.

Nuestra lógica `Sql` reestructurada representa lo mejor de ambos mundos. Cumple con SRP y también con otro principio clave del diseño de

clases orientadas a objetos, denominado Principio abierto/cerrado[48]: las clases deben abrirse para su ampliación para cerrarse para su modificación. La nueva clase `Sql` se abre a nuevas funcionalidades mediante la creación de subclases pero podemos realizar estos cambios y mantener cerradas las demás clases. Basta con añadir nuestra clase `UpdateSql`.

Debemos estructurar nuestros sistemas para ensuciarlos lo menos posible cuando los actualicemos con nuevas funciones o cambios. En un sistema ideal, incorporamos nuevas funciones ampliándolo, no modificando el código existente.

Aislarnos de los cambios

Las necesidades cambiarán y también lo hará el código. En la programación orientada a objetos aprendemos que hay clases concretas que contienen detalles de implementación (el código) y clases abstractas que sólo representan conceptos. Una clase cliente que dependa de detalles concretos está en peligro si dichos detalles cambian. Podemos recurrir a interfaces y clases abstractas para aislar el impacto de dichos detalles.

Las dependencias de detalles de concretos crean retos para nuestro sistema. Si tenemos que crear la clase `Portfolio` y ésta depende de una API `TokyoStockExchange` externa para obtener su valor, nuestros casos de prueba se verán afectados por la volatilidad de esta búsqueda. Resulta complicado crear una prueba cuando se obtiene una respuesta diferente cada cinco minutos. En lugar de diseñar `Portfolio` para que dependa directamente de `TokyoStockExchange`, creamos una interfaz, `StockExchange`, que declara un único método:

```
public Interface StockExchange {
    Money currentPrice(String symbol);
}
```

Diseñamos `TokyoStockExchange` para implementar esta interfaz. También nos aseguramos de que el constructor de `Portfolio` adopte como argumento una referencia a StockExchange:

```
public Portfolio {
    private StockExchange exchange;
    public Portfolio(StockExchange exchange) {
        this.exchange = exchange;
    }
    //…
}
```

Ahora la prueba puede crear una implementación de la interfaz `StockExchange` que emule `TokyoStockExchange`. Esta implementación de

prueba fijará el valor actual del símbolo que usemos en la prueba.

Si nuestra prueba demuestra la adquisición de cinco acciones de Microsoft para nuestra cartera de valores, diseñe el código de la implementación de prueba para que siempre devuelva 100 dólares por acción de Microsoft. Nuestra implementación de prueba de la interfaz `StockExchange` se reduce a una sencilla búsqueda de tabla. De este modo podemos crear una prueba que espere un valor de cartera total de 500 dólares:

```
public class PortfolioTest {
    private FixedStockExchangeStub exchange;
    private Portfolio portfolio;


    @Before
    protected void setUp() throws Exception {
        exchange = new FixedStockExchangeStub();
        exchange.fix(“MSFT”, 100);
        portfolio = new Portfolio(exchange);
    }


    @Test
    public void GivenFiveMSFTTotalShouldBe500() throws Exception {
        portfolio.add(5, “MSFT”);
        Assert.assertEquals(500, portfolio.value());
    }
}
```

Si diseccionamos un sistema para poder probarlo de esta forma, resultará más flexible y se podrá reutilizar. La ausencia de conexiones significa que los elementos del sistema se aíslan entre ellos y de otros cambios. Este aislamiento hace que comprendamos mejor los elementos del sistema.

Al minimizar las conexiones de esta forma, nuestras clases cumplen otro principio de diseño: *Dependency Inversion Principle* (DIP) o Principio de inversión de dependencias[49]. Básicamente afirma que nuestras clases deben depender de abstracciones, no de detalles concretos.

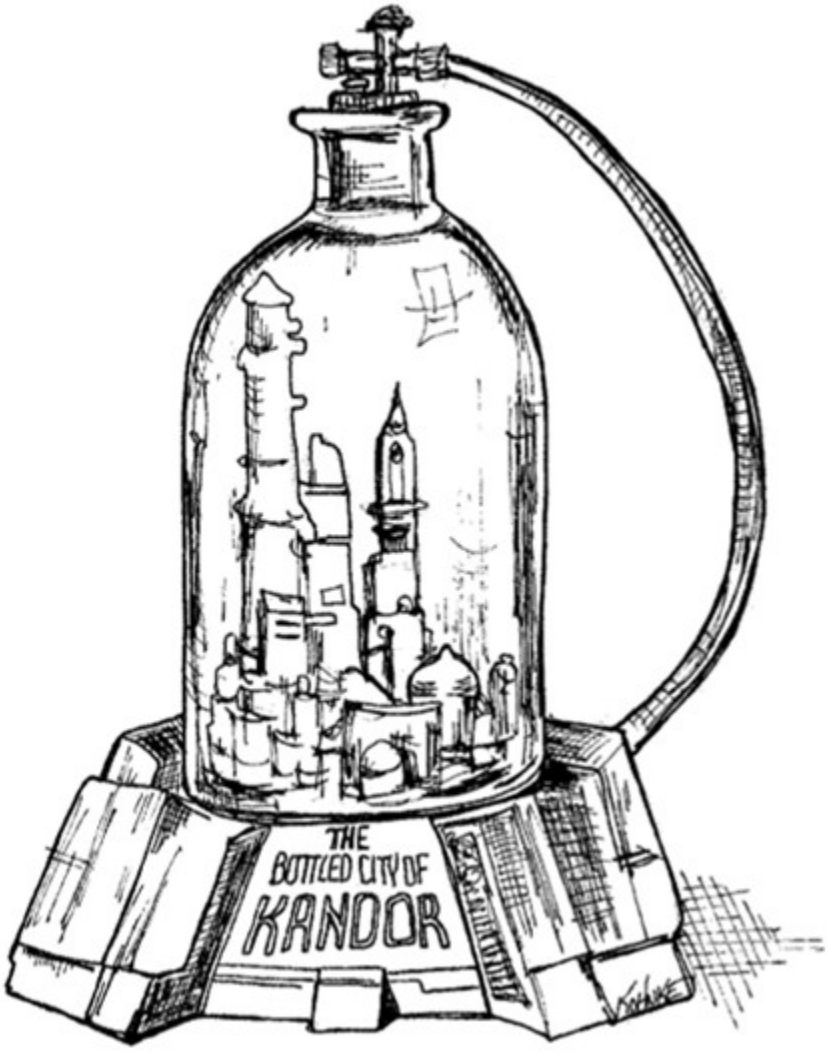
En lugar de depender de los detalles de implementación de la clase `TokyoStockExchange`, nuestra clase Portfolio depende de la interfaz `StockExchange`, que representa el concepto abstracto de solicitar el precio actual de una acción. Esta abstracción aísla todos los datos concretos de la obtención de dicho precio, incluyendo de dónde se obtiene.

Bibliografía

- **[RDD]**: *Object Design: Roles, Responsibilities, and Collaborations*, Rebecca Wirfs-Brock et al., Addison-Wesley, 2002.
- **[PPP]**: *Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.
- **[Knuth92]**: *Literate Programming*, Donald E. Knuth, Center for the Study of language and Information, Leland Stanford Junior University, 1992.

11
Sistemas

por el Dr. Kevin Dean Wampler



«La complejidad es letal. Acaba con los desarrolladores y dificulta la planificación, generación y pruebas de los productos».
—Ray Ozzie, CTO, Microsoft Corporation

Cómo construir una ciudad

¿Podría encargarse de todos los detalles por su cuenta? Seguramente no. Incluso la gestión de una ciudad existente sería demasiado para una sola persona. Y aun así, las ciudades funcionan (en la mayoría de los casos). Funcionan porque tienen equipos que controlan partes concretas de la ciudad, el alcantarillado, la red eléctrica, el tráfico, la seguridad, las normativas urbanísticas, etc. Algunos se encargan de aspectos generales y otros se centran en los detalles.

Las ciudades también funcionan porque disponen de evolucionados niveles de abstracción y modularidad que permiten a individuos y componentes trabajar de forma eficaz, sin necesidad de entender el trasfondo general.

Aunque los equipos de *software* se suelen organizar de esta forma, los sistemas en los que trabajan no suelen contar con la misma separación de aspectos y niveles de abstracción. En este capítulo veremos cómo mantener la limpieza en niveles superiores de abstracción, en el sistema.

Separar la construcción de un sistema de su uso

En primer lugar, recuerde que la construcción es un proceso muy diferente al uso. Mientras escribo estas líneas, a través de la ventana veo un nuevo hotel en construcción en Chicago. Hoy instalarán una gran grúa. Todos los obreros llevan casco. Dentro de un año habrán acabado el hotel. La grúa desaparecerá. El edificio estará terminado, con su reluciente fachada de cristal y su atractiva decoración. La gente que trabajará en él también será diferente.

> *Los sistemas de software deben separar el proceso de inicio, en el que se crean los objetos de la aplicación y se conectan las dependencias, de la lógica de ejecución que toma el testigo tras el inicio.*

El proceso de inicio es un aspecto que toda aplicación debe abordar. Es el primero que veremos en este capítulo. La separación de aspectos es una

de las técnicas de diseño más antiguas e importantes de nuestra profesión.

Desafortunadamente, muchas aplicaciones no lo hacen. El código del proceso de inicio se mezcla con la lógica de tiempo de ejecución. Veamos un ejemplo típico:

```
public Service getService() {
    if (service == null)
        service = new MyServiceImpl (…); //¿Lo bastante predeterminado para la mayoría de los casos?
    return service;
}
```

Es la técnica de *inicialización/evaluación tardía* y tiene sus méritos. No incurrimos en la sobrecarga de la construcción a menos que usemos el objeto realmente, y como resultado el tiempo de inicio se puede acelerar. También evitamos que se devuelva `null`.

Sin embargo, ahora tenemos una dependencia en `MyServiceImpl` y todo lo que su constructor requiere (que he omitido). No podemos compilar sin resolver estas dependencias, aunque nunca usemos un objeto de este tipo en tiempo de ejecución.

Las pruebas también pueden ser un problema. Si `MyServiceImpl` es un objeto pesado, tendremos que asegurarnos de asignar el correspondiente *test double*[50] u objeto simulado al campo de servicio antes de invocar este método en las pruebas de unidad. Como la lógica de la construcción se mezcla con el procesamiento normal de tiempo de ejecución, debemos probar todas las rutas de ejecución (como la prueba `null` y su bloque). Al contar con ambas responsabilidades, el método hace más de una cosa, por lo que se incumple el principio de responsabilidad única.

Lo peor de todo es que no sabemos si `MyServiceImpl` es el objeto correcto en todos los casos. ¿Por qué la clase con este método tiene que conocer el contexto global? ¿Podemos saber realmente cuál es el objeto correcto que usar aquí? ¿Es posible que un mismo tipo sea el correcto para todos los contextos posibles?

Un caso de *inicialización tardía* no es un problema serio. Sin embargo, suele haber muchos casos de este tipo de configuración en las aplicaciones. Por tanto, la estrategia de configuración global (si existe) se disemina por la aplicación, sin apenas modularidad y con una significativa duplicación.

Si somos diligentes sobre el diseño de sistemas robustos y bien formados, no debemos permitir fallos de modularidad. El proceso de inicio de la construcción y conexión de objetos no es una excepción. Debemos modularizar este proceso y asegurarnos de contar con una estrategia global y coherente para resolver las dependencias principales.

Separar Main

Una forma de separar la construcción del uso consiste en trasladar todos los aspectos de la construcción a `main` o a módulos invocados por `main`, y diseñar el resto del sistema suponiendo que todos los objetos se han creado y conectado correctamente (véase la figura 11.1).

El flujo de control es fácil de seguir. La función `main` crea los objetos necesarios para el sistema, los pasa a la aplicación y ésta los utiliza. Verá que las flechas de dependencia atraviesan la barrera entre `main` y la aplicación. Todas van en la misma dirección, alejándose de `main`, lo que significa que la aplicación no tiene conocimiento de `main` ni del proceso de construcción. Simplemente espera que todo se haya construido correctamente.

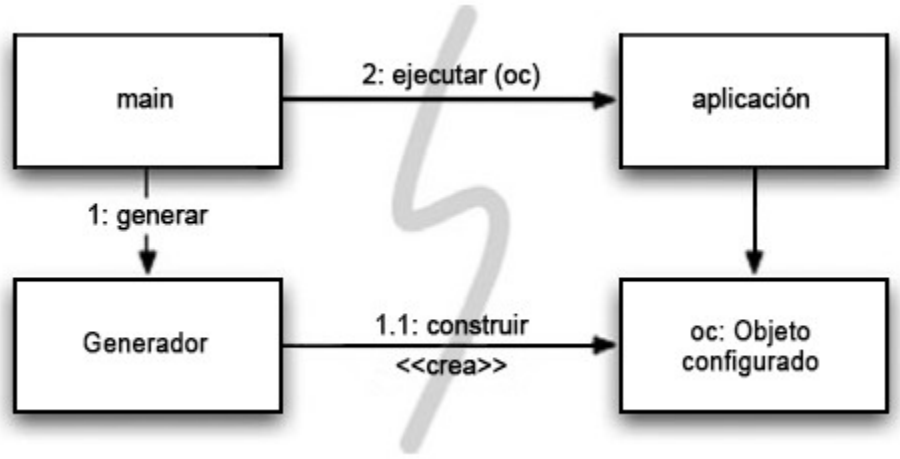


Figura 11.1. Separación de la construcción en `main()`.

Factorías

En ocasiones, la aplicación tendrá que ser responsable de la creación de un objeto. Por ejemplo, en un sistema de procesamiento de pedidos, la aplicación debe crear las instancias `LineItem` que añadir a `Order`. En este caso, podemos usar el patrón de factoría abstracta[51] para que la aplicación controle cuándo crear `LineItem`, pero mantener los detalles de dicha construcción separados del código de la aplicación (véase la figura 11.2).

De nuevo vemos que todas las dependencias se desplazan desde `main` a la aplicación `OrderProcessing`, lo que significa que la aplicación se desconecta de los detalles de creación de `LineItem`. Esta capacidad se

incluye en `LineItemFactoryImplementation`, en el extremo `main` de la línea. Y sin embargo, la aplicación tiene control total sobre cuándo se crean las instancias `LineItem` e incluso puede proporcionar argumentos de constructor específicos de la aplicación.

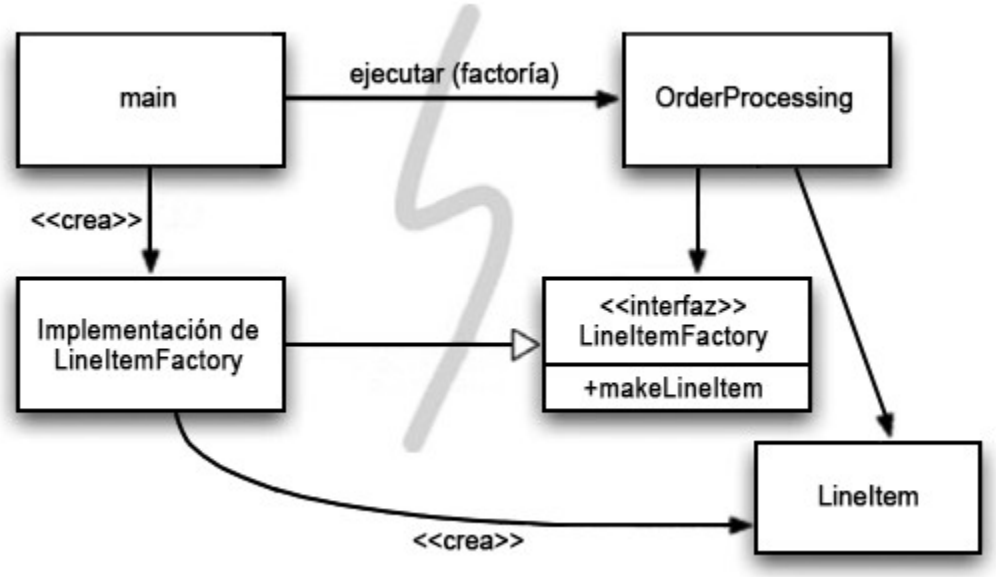


Figura 11.2. Separación de la construcción con una factoría.

Inyectar dependencias

Un potente mecanismo para separar la construcción del uso es la Inyección de dependencias, la aplicación de Inversión de control (*Inversion of Control* o IoC) a la administración de dependencias[52]. La Inversión de control pasa responsabilidades secundarias de un objeto a otros dedicados a ese cometido, por lo que admite el principio de responsabilidad única. En el contexto de la administración de dependencias, un objeto no debe ser responsable de instanciar dependencias, sino que debe delegar esta responsabilidad en otro mecanismo autorizado, de modo que se invierte el control. Como la configuración es un aspecto global, este mecanismo autorizado suele ser la rutina `main` o un contenedor de propósito especial.

Las búsquedas JNDI son una implementación parcial de la inyección de dependencias, en las que un objeto solicita a un servidor de directorios un servicio que coincida con un nombre concreto.

```
MyService myService = (MyService)(jndiContext.lookup("NameOfMyService"));
```

El objeto invocador no controla el tipo de objeto devuelto (siempre que implemente la interfaz correcta, evidentemente), pero es el que resuelve la dependencia de forma activa.

La verdadera inyección de dependencias va un paso más allá. La clase no hace nada directamente para resolver sus dependencias, es totalmente pasiva. Por el contrario, ofrece métodos de establecimiento o argumentos de constructor (o ambos) que se usan para inyectar las dependencias. En el proceso de construcción, el contenedor de inyección de dependencias crea instancias de los objetos necesarios (normalmente bajo demanda) y usa los argumentos de constructor o métodos de establecimiento proporcionados para conectar las dependencias. Los objetos dependientes empleados suelen especificarse a través de un archivo de configuración o mediante programación en un módulo de construcción de propósito especial.

La estructura Spring proporciona el contenedor de inyección de dependencias más conocido para Java[53]. Los objetos que se van a conectar se definen en un archivo de configuración XML y después se solicitan objetos concretos por nombre en código de Java. Veremos un ejemplo en breve.

¿Y qué sucede con las virtudes de la inicialización tardía? En ocasiones es útil con la inyección de dependencias. Por un lado, muchos contenedores de inyección de dependencias no crean un objeto hasta que es necesario. Por otra parte, muchos de estos contenedores cuentan con mecanismos para invocar factorías o crear `proxies` que se pueden usar para evaluación tardía y optimizaciones similares[54].

Evolucionar

Las ciudades nacen de pueblos, que nacen de asentamientos. Inicialmente, los caminos son estrechos y prácticamente inexistentes, después se asfaltan y aumentan de tamaño.

Los pequeños edificios y solares vacíos se llenan de otros mayores que acaban convirtiéndose en rascacielos. Al principio no hay servicios, electricidad, agua, alcantarillado o Internet (¡vaya!). Estos servicios se añaden cuando aumenta la densidad de población.

Este crecimiento no es fácil. Cuántas veces mientras conduce por una carretera llena de baches y ve una señal de obras no se ha preguntado por

qué no la hicieron más ancha desde un principio.

No se podía haber hecho de otra forma. ¿Quién puede justificar el gasto en una autopista de seis carriles que atraviese un pequeño pueblo como anticipación a un supuesto crecimiento? ¿Quién querría una autopista así en su ciudad?

Conseguir sistemas perfectos a la primera es un mito. Por el contrario, debemos implementar hoy, y refactorizar y ampliar mañana. Es la esencia de la agilidad iterativa e incremental. El desarrollo controlado por pruebas, la refactorización y el código limpio que generan hace que funcione a nivel del código.

¿Pero qué sucede en el nivel del sistema? ¿La arquitectura del sistema no requiere una planificación previa? Sin duda no puede aumentar incrementalmente algo sencillo a algo complejo, ¿o sí?

> *Los sistemas de software son únicos si los comparamos con los sistemas físicos. Sus arquitecturas pueden crecer incrementalmente, si mantenemos la correcta separación de los aspectos.*

La naturaleza efímera de los sistemas de *software* hace que sea posible, como veremos. Primero nos centraremos en una arquitectura que no separa correctamente los aspectos. Las arquitecturas EJB1 y EJB2 originales no separaban correctamente los aspectos y por tanto imponían barreras innecesarias al crecimiento orgánico. Imagine un bean de entidad para una clase `Bank` persistente. Un bean de entidad es una representación en memoria de datos relacionales, es decir, una fila de una tabla.

Primero, debe definir una interfaz local (en proceso) o remota (MVJ independiente), que los clientes usen. El Listado 1-1 muestra una posible interfaz local:

Listado 11-1
Una interfaz local EJB2 para el EJB Bank.

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;

public interface BankLocal extends java.ejb.EJBLocalObject {
    String getStreetAddrl{} throws EJBException;
    String getStreetAddr2{} throws EJBException;
    String getCity() throws EJBException;
    String getState() throws EJBException;
```

```
    String getZipCode() throws EJBException;
    void setStreetAddr1(String street1) throws EJBException;
    void setStreetAddr2(String street2) throws EJBException;
    void setCity(String city) throws EJBException;
    void setState(String state) throws EJBException;
    void setZipCode(String zip) throws EJBException;
    Collection getAccounts() throws EJBException;
    void setAccounts(Collection accounts) throws EJBException;
    void addAccount(AccountDTO accountDTO) throws EJBException;
}
```

Mostramos diversos atributos de la dirección de `Bank` y una colección de cuentas del banco, cuyos datos se procesarán por un EJB `Account` diferente. El Listado 11-2 muestra la correspondiente clase de implementación del bean `Bank`.

Listado 11-2
Implementación del bean de entidad EJB2.

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;


public abstract class Bank implements javax.ejb.EntityBean {
    // Lógica empresarial…
    public abstract String getStreerAddr1();
    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(String city);
    public abstract void setState(String state);
    public abstract void setZipCode(String zip);
    public abstract Collection getAccounts();
    public abstract void setAccounts(Collection accounts);
    public void addAccount(AccountPTO accountDTO) {
        InitialContext context = new InitialContext();
        AccountHomeLocal accountHome = context.lookup(“AccountHomeLocal”);
        AccountLocal account = accountHome.create(accountDTO);
        Collection accounts = getAccounts();
        accounts.add(account);
    }
    // Lógica del contenedor EJB
    public abstract void setId(Integer id);
    public abstract Integer getId();
    public Integer ejbCreate(Integer id) {…}
    public void ejbPostCreate(Integer id) {…}
    // El resto tendría que implementarse pero se deja vacío:
    public void setEntityContext(EntityContext ctxt) {}
    public void unsetEntityContext() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void ejbRemove() {}
}
```

No mostramos la correspondiente interfaz `LocalHome`, básicamente una factoría usada para crear objetos, no los métodos de consulta `Bank` que pueda añadir.

Por último, debemos crear uno o varios descriptores de implementación XML que especifiquen los detalles de asignación relacional de objetos en un almacén persistente, el comportamiento deseado de la transacción, limitaciones de seguridad y demás.

La lógica empresarial está directamente conectada al contenedor de la aplicación EJB2. Debe crear subclases de tipos de contenedor y

proporcionar los métodos de ciclo vital necesarios para el contenedor. Debido a esta conexión al contenedor pesado, las pruebas de unidad aisladas son complicadas. Es necesario imitar el contenedor, algo difícil, o perder demasiado tiempo en la implementación de EJB y pruebas en un servidor real. La reutilización fuera de la arquitectura EJB2 es imposible, debido a esta estrecha conexión. Por último, incluso la programación orientada a objetos se ve afectada. Un bean no se puede heredar de otro. Fíjese en la lógica para añadir una nueva cuenta. En bean EJB2 es habitual definir Objetos de transferencia de datos (*Data Transfer Objects* o DTO), estructuras sin comportamiento. Esto suele generar tipos redundantes con los mismos datos y requiere código predefinido para copiar datos entre objetos.

Aspectos transversales

La arquitectura EJB2 se acerca a la verdadera separación de aspectos en determinados aspectos. Por ejemplo, los comportamientos transaccionales, de seguridad y comportamiento deseados se declaran en los descriptores de implementación, independientemente del código fuente. Aspectos como la persistencia suelen cruzar los límites de objeto naturales de un dominio. Por lo general intentará mantener todos sus objetos mediante la misma estrategia, por ejemplo con un determinado DBMS[55] y no archivos planos, usando determinadas convenciones de nomenclatura para tablas y columnas, una semántica transaccional coherente, etc.

En principio, puede razonar su estrategia de persistencia de una forma modular y encapsulada, pero en la práctica tendrá que distribuir el mismo código que implemente la estrategia de persistencia entre varios objetos. Usamos el término transversales para este tipo de aspectos. De nuevo, la estructura de persistencia podría ser modular y la lógica de dominios, aislada, también. El problema es la intersección entre ambos dominios.

De hecho, la forma en que la arquitectura EJB procesa persistencia, seguridad y transacciones es una Programación orientada a aspectos (*Aspect Oriented Programming* o AOP)[56] anticipada, un enfoque de carácter general para restaurar la modularidad en aspectos transversales. En AOP, construcciones modulares denominadas aspectos especifican qué puntos del sistema deben modificar su comportamiento de forma coherente para

admitir un determinado aspecto. Esta especificación se realiza mediante un sucinto mecanismo de declaración o programación.

Si usamos la persistencia como ejemplo, podría declarar qué objetos y atributos (o patrones) deben conservarse y después delegar las tareas de persistencia a su estructura de persistencia. Las modificaciones de comportamiento no son invasivas[57] para el código de destino. Veamos tres aspectos o mecanismos similares en Java.

Proxies de Java

Los `proxies` de Java son útiles en casos sencillos, como envolver invocaciones de métodos en objetos o clases concretas. Sin embargo, los proxies dinámicos proporcionados en el JDK sólo funcionan con interfaces. Para aplicarlos a clases, debe usar una biblioteca de manipulación de código de *bytes*, como CGLIB, ASM o Javassist[58].

El Listado 11-3 muestra la estructura de un proxy JDK para ofrecer asistencia de persistencia a nuestra aplicación Bank; únicamente abarca los métodos para obtener y establecer la lista de cuentas.

Listado 11-3
Ejemplo de proxy del JDK.

```
// Bank.java (eliminando nombres de paquetes…)
import java.utils.*;


// La abstracción de un banco.
public interface Bank {
    Collection<Account> getAccounts();
    void setAccounts(Collection<Accounts> accounts);
}


// BankImpl.java
import java.utils.*;


// “Plain Old Java Object” POJO que implementa la abstracción.
public class BankImpl implements Bank {
    private List<Account> accounts;


    public Collection<Account> getAccounts() {
        return accounts;
    }
    public void setAccounts(Collections<Accounts> accounts) {
        this.accounts = new ArrayList<Accounts>();
        for (Account account: accounts) {
            this.accounts.add(account);
        }
    }
}


// BankProxyHandler.java
import java.lang.reflect.*;
import java.util.*;


// «InvocationHandler» necesario para la API de proxy.
```

```
public class BankProxyHandler implements InvocationHandler {
    private Bank bank;


    public BankHandler (Bank bank) {
        this.bank = bank;
    }


    // Método definido en InvocationHandler
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        String methodName = method.getName();
        if (methodName.equals("getAccounts")) {
            bank.setAccounts(getAccountsFromDatabase());
            return bank.getAccounts();
        } else if (methodName.equals("setAccounts")) {
            bank.setAccounts((Collection<Account>) args[0]);
            setAccountsToDatabase(bank.getAccounts());
            return null;
        } else {
            …
        }
    }


    // Muchos detalles:
    protected Collection<Account> getAccountsFromDatabase() {…}
    protected void setAccountsToDatabase(Collection<Account> accounts) {…}
}

//En otra parte…

Bank bank = (Bank) Proxy.newproxyInstance(
    Bank.class.getClassLoader(),
    new Class[] { Bank.class },
    new BankProxyHandler(new BankImpl()));
```

Definimos la interfaz `Bank`, que envolvemos en el proxy y un POJO (*Plain-Old Object* u Objeto sencillo de Java), `BankImpl`, que implementa la lógica empresarial (encontrará más información sobre POJO en un apartado posterior).

La API Proxy requiere un objeto `InvocationHandler` que invocar para implementar las invocaciones de métodos `Bank` realizadas en el proxy. `BankProxyHandler` usa la API de reflexión de Java para asignar las invocaciones de métodos genéricos a los métodos correspondientes de `BankImpl`, y así sucesivamente.

El código es abundante y complejo, incluso para este sencillo caso[59]. El uso de una de las bibliotecas de manipulación de *bytes* es igualmente complicado. El volumen y la complejidad de este código son dos de los inconvenientes de los `proxies`. Dificultan la creación de código limpio. Además, los proxies no ofrecen un mecanismo para especificar puntos de ejecución globales del sistema, imprescindibles para una verdadera solución AOP[60].

Estructuras AOP Java puras

Afortunadamente, gran parte del código predefinido de proxy se puede procesar de forma automática mediante herramientas. Los proxies se usan

internamente en varias estructuras de Java como Spring AOP y JBoss AOP, para implementar aspectos en Java[61]. En Spring, se crea la lógica empresarial en forma de POJO, específicos de su dominio. No dependen de estructuras empresariales (ni de otros dominios). Por tanto, son conceptualmente más sencillos y más fáciles de probar. Su relativa simplicidad garantiza que se implementen correctamente las correspondientes historias y el mantenimiento y evolución del código en historias futuras.

La infraestructura necesaria de la aplicación, incluidos aspectos transversales como persistencia, transacciones, seguridad, almacenamiento en caché y recuperación ante fallos, se incorpora por medio de archivos de configuración declarativos o API. En muchos casos, se especifican aspectos de bibliotecas Spring o JBoss, en los que la estructura controla el uso de proxies de Java o bibliotecas de código de *bytes* de forma transparente al usuario. Estas declaraciones controlan el contenedor de inyección de dependencias, que crea instancias de los principales objetos y las conecta bajo demanda.

El Listado 11-4 muestra un fragmento tipo de un archivo de configuración de Spring V2.5, app.xml[62].

Listado 11-4
Archivo de configuración de Spring 2.X

```
<beans>
    …
    <bean id=“appDataSource”
    class=“org.apache.commons.dbcp.BasicDataSource”
    destroy-method=“close”
    p:driverClassName=“com.mysql.jdbc.Driver”
    p:url=“jdbc:mysql://localhost:3306/mydb”
    p:username=“me”/>


    <bean id=“bankDataAccessObject”
    class=“com.example.banking.persistence.BankDataAccessObject”
    p:dataSource-ref=“appDataSource”/>


    <bean id=“bank”
    class=“com.example.banking.model.Bank”
    p:dataAccessObject-ref=“bankDataAccessObject”/>
    …
</beans>
```

Cada bean es como una parte de una muñeca rusa anidada, con un objeto de domino de un proxy `Bank` (envuelto) por un Objeto de acceso a datos (*Data Accessor Object*, DAO), que también se procesa a través de un proxy por medio de un origen de datos de controlador JDBC (véase la figura 11.3).

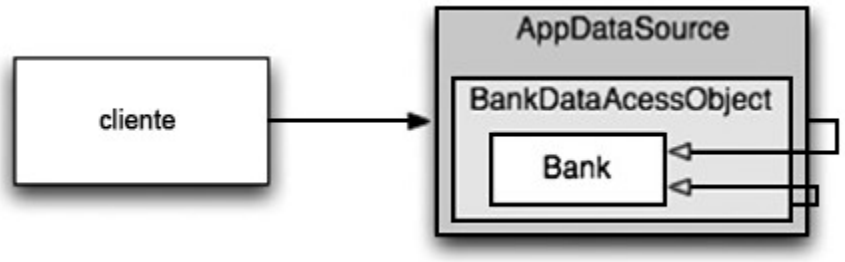


Figura 11.3. La “muñeca rusa” de elementos de decoración.

El cliente cree que invoca `getAccounts()` en un objeto `Bank`, pero en realidad se comunica con el objeto `DECORATOR`[63] más externo de un grupo, un objeto que amplía el comportamiento básico del POJO `Bank`. Podríamos añadir otros objetos de decoración para transacciones, almacenamiento en caché y demás.

En la aplicación, bastan unas líneas para solicitar al contenedor de ID los objetos de nivel superior del sistema, como se especifica en el archivo XML.

```
XmlBeanFactory bf =
    new XmlBeanFactory(new ClassPathResource(“app.xml”, getclass()));
Bank bank = (Bank) bf.getBean(“bank”);
```

Como apenas se necesitan líneas de código Java específico de Spring, la aplicación se desconecta casi por completo de Spring y desaparecen los problemas de conexión de sistemas como EJB2.

Aunque XML puede ser difícil de leer[64], la directiva especificada en estos archivos de configuración es más sencilla que la complicada lógica de proxy y aspectos oculta a la vista y creada de forma automática. Es una arquitectura tan atractiva que sistemas como Spring modificaron totalmente el estándar EJB para la versión 3. EJB3 sigue el modelo de Spring de aspectos transversales admitidos mediante declaraciones con archivos de configuración XML y/o anotaciones de Java 5.

El Listado 11-5 muestra nuestro objeto Bank reescrito en EJB3[65].

Listado 11-5
Un EJB Bank EJB3.

```
package com.example.banking.model;
import javax.persistence;
import java.util.ArrayList;
import java.util.Collection;


@Entity
@Table(name = “BANKS”)
public class Bank implements java.io.Serializable {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
```

```
    @Embeddable // Un objeto en línea en la fila DB de Bank
    public class Address {
        protected String streetAddr1;
        protected String streetAddr2;
        protected String city;
        protected String state;
        protected String zipCode;
    }


    @Embedded
    private Address address;


    @OneToMany (cascade = CascadeType.ALL, fetch = FetchType.EAGER,
                mappedBy="bank")
    private Collection<Account> accounts = new ArrayList<Account>();


    public int getId() {
        return id;
    }


    public void setID(int id) {
        this.id = id;
    }


    public void addAccount(Account account) {
        account.setBank(this);
        accounts.add(account);
    }


    public Collection<Account> getAccounts() {
        return accounts;
    }


    public void setAccounts(Collection<Account> accounts) {
        this.accounts = accounts;
    }
}
```

Este código es mucho más limpio que el código EJB2 original. Se conservan algunos detalles de entidades, en las anotaciones. Sin embargo, como no hay información fuera de las anotaciones, el código es limpio y fácil de probar, mantener y demás.

Parte de la información de persistencia de las anotaciones se puede cambiar a descriptores de implementación XML si es necesario, dejando un POJO puro. Si los detalles de asignación de persistencia no cambian con frecuencia, muchos equipos pueden optar por mantener las anotaciones pero con menos obstáculos que si usaran EJB2.

Aspectos de AspectJ

Por último, la herramienta más completa de separación a través de aspectos es el lenguaje AspectJ[66], una extensión de Java que ofrece compatibilidad de primer nivel para aspectos como construcciones de modularidad. Los enfoques puros de Java proporcionados por Spring AOP y JBoss AOP son suficientes en el 80-90 por 100 de los casos en los que los aspectos son útiles. Sin embargo, AspectJ ofrece un conjunto de herramientas avanzadas y completas para la separación de aspectos. El inconveniente de AspectJ es la necesidad de adoptar nuevas herramientas y aprender nuevas

construcciones del lenguaje. Los problemas de adopción se han mitigado parcialmente gracias a la introducción de un formato de anotación de AspectJ, en el que se usan anotaciones de Java 5 para definir aspectos con código puro de Java. Además, la estructura Spring dispone de funciones que facilitan la incorporación de aspectos basados en anotaciones en un equipo con experiencia limitada con AspectJ.

El análisis completo de AspectJ supera los objetivos de este libro. Si necesita más información al respecto, consulte [AspectJ], [Colyer] y [Spring].

Pruebas de unidad de la arquitectura del sistema

La separación a través de enfoques similares a aspectos no se puede menospreciar. Si puede crear la lógica de dominios de su aplicación mediante POJO, sin conexión con los aspectos arquitectónicos a nivel del código, entonces se podrá probar realmente la arquitectura. Puede evolucionar de simple a sofisticado, de acuerdo a las necesidades, adoptando nuevas tecnologías bajo demanda. No es necesario realizar un *Buen diseño por adelantado* (Big Design Up Front[67], BDUF). De hecho, BDUF puede ser negativo ya que impide la adaptación al cambio, debido a la resistencia fisiológica a descartar esfuerzos previos y a la forma en que las decisiones arquitectónicas influyen en la concepción posterior del diseño.

Los arquitectos deben realizar BDUF ya que no resulta factible aplicar cambios arquitectónicos radicales a una estructura física una vez avanzada la construcción[68]. Aunque el *software* se rige por una física propia[69], es económicamente factible realizar cambios radicales si la estructura del *software* separa sus aspectos de forma eficaz.

Esto significa que podemos iniciar un proyecto de *software* con una arquitectura simple pero bien desconectada, y ofrecer historias funcionales de forma rápida, para después aumentar la infraestructura. Algunos de los principales sitios Web del mundo han alcanzado una gran disponibilidad y rendimiento por medio de sofisticadas técnicas de almacenamiento en caché, seguridad, virtualización y demás, todo ello de forma eficaz y flexible ya que los diseños mínimamente conectados son adecuadamente simples en cada nivel de abstracción y ámbito. Evidentemente, no quiere

decir que acometamos los proyectos sin timón. Debemos tener expectativas del ámbito general, objetivos y un programa, así como la estructura general del sistema resultante. Sin embargo, debemos mantener la capacidad de cambiar de rumbo en respuesta a las circunstancias.

La arquitectura EJB inicial es una de las API conocidas con un exceso de ingeniería y que compromete la separación de aspectos. Incluso las API bien diseñadas pueden ser excesivas cuando no resultan necesarias. Una API correcta debe desaparecer de la vista en la mayoría de los casos, para que el equipo dedique sus esfuerzos creativos a las historias implementadas. En caso contrario, las limitaciones arquitectónicas impedirán la entrega eficaz de un valor óptimo para el cliente. Para recapitular:

> *Una arquitectura de sistema óptima se compone de dominios de aspectos modularizados, cada uno implementado con POJO. Los distintos dominios se integran mediante aspectos o herramientas similares mínimamente invasivas. Al igual que en el código, en esta arquitectura se pueden realizar pruebas.*

Optimizar la toma de decisiones

La modularidad y separación de aspectos permite la descentralización de la administración y la toma de decisiones. En un sistema suficientemente amplio, ya sea una ciudad o un proyecto de *software*, no debe haber una sola persona que adopte todas las decisiones.

Sabemos que conviene delegar las responsabilidades en las personas más cualificadas. Solemos olvidar que también conviene posponer decisiones hasta el último momento. No es falta de responsabilidad; nos permite tomar decisiones con la mejor información posible. Una decisión prematura siempre es subjetiva. Si decidimos demasiado pronto, tendremos menos información del cliente, reflexión mental sobre el proyecto y experiencia con las opciones de implementación.

> *La agilidad que proporciona un sistema POJO con aspectos modularizados nos permite adoptar decisiones óptimas a tiempo,*

basadas en los conocimientos más recientes. Además, se reduce la complejidad de estas decisiones.

Usar estándares cuando añadan un valor demostrable

La construcción de edificios es una maravilla para la vista debido al ritmo empleado (incluso en invierno) y los extraordinarios diseños posibles gracias a la tecnología actual. La construcción es un sector maduro con elementos, métodos y estándares optimizados que han evolucionado bajo presión durante siglos.

Muchos equipos usaron la arquitectura EJB2 por ser un estándar, aunque hubiera bastado con diseños más ligeros y sencillos. He visto equipos obsesionados con estándares de moda y que se olvidaron de implementar el valor para sus clientes.

Los estándares facilitan la reutilización de ideas y componentes, reclutan individuos con experiencia, encapsulan buenas ideas y conectan componentes. Sin embargo, el proceso de creación de estándares puede tardar demasiado para el sector, y algunos pierden el contacto con las verdaderas necesidades de aquello para los que están dirigidos.

Los sistemas necesitan lenguajes específicos del dominio

La construcción de edificios, como muchos dominios, ha desarrollado un rico lenguaje con vocabularios, frases y patrones[70] que comunican información esencial de forma clara y concisa. En el mundo del *software*, ha renacido el interés por crear Lenguajes específicos del dominio (*Domain-Specific Languages* o DSL)[71], pequeños lenguajes independientes de creación de secuencias de comandos o API de lenguajes estándar que permiten crear código que se lea de una forma estructurada, como lo

escribiría un experto del dominio. Un buen DSL minimiza el vacío de comunicación entre un concepto de dominio y el código que lo implementa, al igual que las prácticas ágiles optimizan la comunicación entre un equipo y los accionistas del proyecto. Si tiene que implementar la lógica de dominios en el mismo lenguaje usado por un experto del dominio, hay menos riesgo de traducir incorrectamente el dominio en la implementación.

Los DSL, si se usan de forma eficaz, aumentan el nivel de abstracción por encima del código y los patrones de diseño. Permiten al desarrollador revelar la intención del código en el nivel de abstracción adecuado.

> *Los lenguajes específicos del dominio permiten expresar como POJO todos los niveles de abstracción y todos los dominios de la aplicación, desde directivas de nivel superior a los detalles más mínimos.*

Conclusión

Los sistemas también deben ser limpios. Una arquitectura invasiva afecta a la lógica de dominios y a la agilidad. Si la lógica de dominios se ve afectada, la calidad se resiente, ya que los errores se ocultan y las historias son más difíciles de implementar. Si la agilidad se ve comprometida, la productividad sufre y las ventajas de TDD se pierden.

En todos los niveles de abstracción, los objetivos deben ser claros. Esto sólo sucede si crea POJO y usa mecanismos similares a aspectos para incorporar otros aspectos de implementación de forma no invasiva.

Independientemente de que diseñe sistemas o módulos individuales, no olvide usar los elementos más sencillos que funcionen.

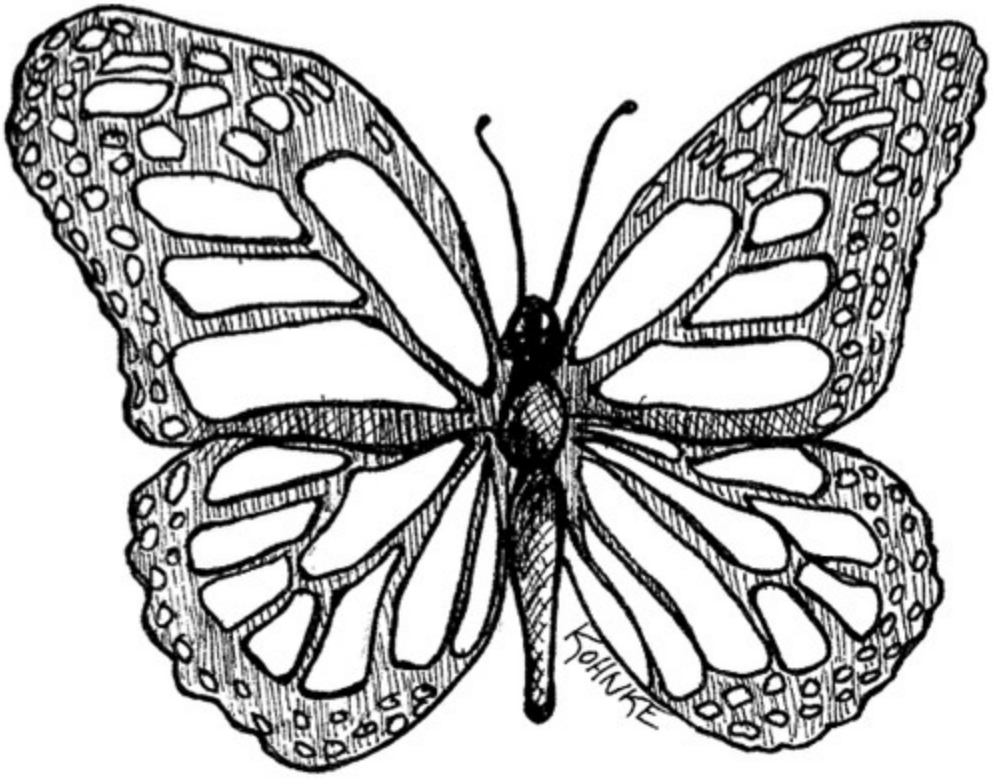
Bibliografía

- **[Alexander]**: Christopher Alexander, *A Timeless Way of Building*, Oxford University Press, New York, 1979.
- **[AOSD]**: Puerto de Desarrollo de *software* orientado a aspectos, http://aosd.net.

- **[ASM]**: Página 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, NJ, 2005.
- **[DSL]**: Lenguaje de programación específico del dominio, http://es.wikipedia.org/wiki/Lenguaje_espec%C3%ADfico_del_dominio.
- **[Fowler]**: Inversión de contenedores de control y el patrón de inyección de dependencias (http://martinfowler.com/articles/injection.html).
- **[Goetz]**: Brian Goetz, *Java Theory and Practice: Decorating with Dynamic Proxies*, http://www.ibm.com/developerworks/java/library/j-jtp08305.html.
- **[Javassist]**: Página de Javassist, http://www.csg.is.titech.ac.jp/chiba/javassist/.
- **[JBoss]**: Página de JBoss, http: //jboss.org.
- **[JMock]**: JMock: Una biblioteca de objetos Mock ligeros para Java, http://jmock.org.
- **[Kolence]**: Kenneth W. Kolence, *Software physics and computer performance measurements*, *Proceedings of the ACM annual conference-Volume* 2, Boston, Massachusetts, pp. 1024-1040,1972.
- **[Spring]**: *The Spring Framework*, http://www.springframework.org.
- **[Mezzaros07]**: *XUnit Patterns*, Gerard Mezzaros, Addison-Wesley, 2007.
- **[GOF]**: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

12
Emergencia

con Jeff Langr



Limpieza a través de diseños emergentes

Imagine que existieran cuatro sencillas reglas para crear diseños de calidad. Imagine que siguiéndolas accediera a la estructura y al diseño de su código y facilitara la aplicación de principios como SRP y DIP. Imagine que estas cuatro reglas facilitaran la emergencia de diseños de calidad.

Muchos consideramos que las cuatro reglas de Kent Beck de diseño sencillo[72] son fundamentales para crear un *software* bien diseñado.

Según Kent, un diseño es sencillo si cumple estas cuatro reglas:

- Ejecuta todas las pruebas.
- No contiene duplicados.
- Expresa la intención del programador.
- Minimiza el número de clases y métodos.

Describiremos estas reglas en orden de importancia.

Primera regla del diseño sencillo: Ejecutar todas las pruebas

En primer lugar, un diseño debe generar un sistema que actúe de la forma prevista. Un sistema puede tener un diseño perfecto sobre el papel pero si no existe una forma sencilla de comprobar que realmente funciona de la forma esperada, el esfuerzo sobre el papel es cuestionable.

Un sistema minuciosamente probado y que supera todas las pruebas en todo momento se denomina sistema testable. Es una afirmación obvia, pero importante. Los sistemas que no se pueden probar no se pueden verificar, y un sistema que no se puede verificar no debe implementarse.

Afortunadamente, crear sistemas testables hace que diseñemos clases de tamaño reducido y un solo cometido. Resulta más sencillo probar clases que cumplen el SRP. Cuantas más pruebas diseñemos, más nos acercaremos a elementos más fáciles de probar. Por lo tanto, hacer que nuestro sistema se pueda probar nos ayuda a crear mejores diseños.

Las conexiones rígidas dificultan la creación de pruebas. Del mismo modo, cuantas más pruebas creemos, más usaremos principios como DIP y herramientas con inyección de dependencias, interfaces y abstracción para minimizar dichas conexiones. Nuestros diseños mejorarán todavía más.

En especial, seguir una sencilla regla que afirme que debemos realizar pruebas y ejecutarlas continuamente afecta el cumplimiento por parte de nuestro sistema de los principales objetivos de la programación orientada a

objetos de baja conexión y elevada cohesión. La creación de pruebas conduce a obtener mejores diseños.

Reglas 2 a 4 del diseño sencillo: Refactorizar

Una vez creadas las pruebas, debemos mantener limpio el código y las clases. Para ello, refactorizamos el código progresivamente. Tras añadir unas líneas, nos detenemos y reflejamos el nuevo diseño. ¿Ha empeorado? En caso afirmativo, lo limpiamos y ejecutamos las pruebas para comprobar que no hay elementos afectados. *La presencia de las pruebas hace que perdamos el miedo a limpiar el código y que resulte dañado*.

En la fase de refactorización, podemos aplicar todos los aspectos del diseño de *software* correcto. Podemos aumentar la cohesión, reducir las conexiones, separar las preocupaciones, modularizar aspectos del sistema, reducir el tamaño de funciones y clases, elegir nombres más adecuados, etc. Aquí también aplicamos las tres últimas reglas del diseño correcto: eliminar duplicados, garantizar la capacidad de expresión y minimizar el número de clases y métodos.

Eliminar duplicados

Los duplicados son los mayores enemigos de un sistema bien diseñado. Suponen un esfuerzo adicional, riesgos añadidos y una complejidad a mayores innecesaria. Los duplicados se manifiestan de diversas formas. Las líneas de código similar pueden modificarse para que parezcan refactorizadas, y hay otras formas de duplicación como la de implementación. Por ejemplo, podríamos tener dos métodos en una clase de colección:

```
int size() {}
boolean isEmpty() {}
```

Podríamos tener implementaciones separadas para cada método. El método `isEmpty` podría controlar un valor booleano y `size` un contador, o podemos eliminar la duplicación y vincular `isEmpty` a la definición de `size`:

```
boolean isEmpty() {
    return 0 == size();
}
```

La creación de un sistema limpio requiere la eliminación de duplicados, aunque sean unas cuantas líneas de código. Fíjese en el siguiente ejemplo:

```
public void scaleToOneDimension {
        float desiredDimension, float imageDimension) {
    if (Math.abs(desiredDimension - imageDimension) < errorThreshold)
        return;
    float scalingFactor = desiredDimension / imageDimension;
    scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);


    RenderedOp newImage = ImageUtilities.getScaledImage(
        image, scalingFactor, scalingFactor);
    image.dispose();
    System.gc();
    image = newImage;
}
public synchronized void rotate(int degrees) {
    RenderedOp newImage = ImageUtilities.getRotatedImage(
        image, degrees);
    image.dispose();
    System.gc();
    image = newImage;
}
```

Para mantener limpio este sistema, debemos eliminar la pequeña cantidad de duplicación entre los métodos `scaleToOneDimension` y `rotate`:

```
public void scaleToOneDimension (
        float desiredDimension, float imageDimension) {
    if (Math.abs(desiredDimension - imageDimension) < errorThreshold)
        return;
    float scalingFactor = desiredDimension / imageDimension;
    scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);
    replaceImage(ImageUtilities.getScaledImage(
        image, scalingFactor, scalingFactor));
}


public synchronized void rotate (int degrees) {
    replaceImage(ImageUtilities.getRotatedImage(image, degrees));
}


private void replaceImage(RenderedOp newImage) (
    image.dispose();
    System.gc();
    image = newImage;
}
```

Al extraer a este reducido nivel, comenzamos a detectar incumplimientos de SRP. Por ello, podríamos cambiar un nuevo método extraído a otra clase. Esto aumenta su visibilidad. Otro miembro del equipo puede ver la necesidad de volver a extraer el nuevo método y usarlo en otro contexto diferente. Esta reutilización mínima puede reducir considerablemente la complejidad del sistema. Saber cómo lograrlo es fundamental para alcanzar la reutilización a gran escala.

El patrón *Método de plantilla*[73] es una técnica muy utilizada para eliminar duplicados de nivel superior. Por ejemplo:

```
public class VacationPolicy {
    public void accrueUSDivisionVacation() {
        // código para calcular las vacaciones en función de las horas trabajadas
        //…
        // código para garantizar que las vacaciones cumplen los mínimos legales
        //…
        // código para aplicar vacation al registro payroll
        //…
    }


    public void accrueEUDivisionVacation() {
        // código para calcular las vacaciones en función de las horas trabajadas
        //…
```

```
        // código para garantizar que las vacaciones cumplen los mínimos legales
        //…
        // código para aplicar vacation al registro payroll
        //…
    }
}
```

El código entre `accrueUSDivisionVacation` y `accrueEuropeanDivisionVacation` es prácticamente idéntico, a excepción del cálculo de mínimos legales. Esa parte del algoritmo cambia en función del tipo de empleado. Podemos eliminar la duplicación evidente si aplicamos el patrón de *Método de plantilla*:

```
abstract public class VacationPolicy {
    public void accrueVacation() {
        calculateBaseVacationHours();
        alterForLegalMinimums();
        applyToPayroll();
    }

    private void calculateBaseVacationHours() { /* … */ };
    abstract protected void alterForLegalMinimums();
    private void applyToPayroll(); { /* … */ };
}

public class USVacationPolicy extends VacationPolicy {
    @Override protected void alterForLegalMinimums() {
        // Lógica específica de EE.UU.
    }
}

public class EUVacationPolicy extends VacationPolicy {
    @Override protected void alterForLegalMinimums() {
        // Lógica específica de la UE.
    }
}
```

Las subclases ocupan el vacío generado en el algoritmo `accrueVacation` y solamente proporcionan los datos que no están duplicados.

Expresividad

Muchos tenemos experiencia con código enrevesado. Muchos lo hemos creado. Es fácil crear código que entendamos, ya que durante su creación nos centramos en comprender el problema que intentamos resolver. Los encargados de mantener el código no lo comprenderán de la misma forma.

El principal coste de un proyecto de *software* es su mantenimiento a largo plazo. Para minimizar los posibles defectos al realizar cambios, es fundamental que comprendamos el funcionamiento del sistema. Al aumentar la complejidad de los sistemas, el programador necesita más tiempo para entenderlo y aumentan las posibilidades de errores. Por tanto, el código debe expresar con claridad la intención de su autor. Cuando más claro sea el código, menos tiempo perderán otros en intentar comprenderlo. Esto reduce los defectos y el coste de mantenimiento.

Puede expresarse si elige nombres adecuados. El objetivo es ver el nombre de una clase y función, y que sus responsabilidades no nos sorprendan.

También puede expresarse si reduce el tamaño de funciones y clases. Al hacerlo, resulta más sencillo asignarles nombres, crearlas y comprenderlas. Otra forma de expresarse es usar una nomenclatura estándar. Los patrones de diseño, por ejemplo, se basan en la comunicación y en la capacidad de expresión. Al usar los nombres de patrones estándar, como `COMMAND` o `VISITOR`, en los nombres de las clases que implementan dichos patrones puede describir sucintamente su diseño a otros programadores.

Las pruebas de unidad bien escritas también son expresivas. Uno de los principales objetivos de una prueba es servir de documentación mediante ejemplos. Los que lean las pruebas deben entender con facilidad para qué sirve una clase.

Pero la forma más importante de ser expresivo es la práctica. A menudo, conseguimos que el código funcione y pasamos al siguiente problema sin detenernos en facilitar la lectura del código para otros. No olvide que seguramente sea el próximo que lea el código.

Por tanto, afronte su creación con orgullo. Dedique tiempo a sus funciones y clases. Seleccione nombres mejores, divida las funciones extensas en otras más reducidas y cuide su obra. El cuidado es un recurso precioso.

Clases y métodos mínimos

Incluso conceptos tan básicos como la eliminación de código duplicado, la expresividad del código y SRP pueden exagerarse. En un esfuerzo por reducir el tamaño de clases y métodos, podemos crear demasiadas clases y métodos reducidos. Esta regla también sugiere minimizar la cantidad de funciones y clases.

Una gran cantidad de clases y métodos suele indicar un dogmatismo sin sentido. Imagine un estándar de código que insista en la creación de una interfaz para todas las clases, o a programadores que insisten en qué campos y comportamientos siempre deben separarse en clases de datos y clases de

comportamiento. Este dogma debe evitarse y cambiarse por un enfoque más pragmático.

Nuestro objetivo es reducir el tamaño general del sistema además del tamaño de clases y funciones, pero recuerde que esta regla es la de menor prioridad de las cuatro. Por ello, aunque sea importante reducir la cantidad de clases y funciones, es más importante contar con pruebas, eliminar duplicados y expresarse correctamente.

Conclusión

¿Existen prácticas sencillas que puedan reemplazar a la experiencia? Por supuesto que no. Sin embargo, las prácticas descritas en este capítulo y en el libro son una forma cristalizada de décadas de experiencia de muchos autores. La práctica del diseño correcto anima y permite a los programadores adoptar principios y patrones que en caso contrario tardarían años en aprender.

Bibliografía

- **[XPE]**: *Extreme Programming Explained: Embrace Change*, Kent Beck, Addison Wesley, 1999.
- **[GOF]**: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

13
Concurrencia

por Brett L. Schuchert

“Los objetos son abstracciones de procesamiento.
Los subprocesos son abstracciones de programaciones”.
—James O. Coplien[74]

La creación de programas concurrentes limpios es complicada, muy complicada. Es mucho más sencillo crear código que se ejecute en un mismo proceso. También es fácil crear código de subprocesamiento múltiple que parezca correcto en la superficie pero que esté dañado a niveles más profundos. Este código funciona correctamente hasta que el sistema se somete a determinadas presiones.

En este capítulo analizaremos la necesidad de la programación concurrente y sus dificultades. Tras ello, presentaremos diversas recomendaciones para superar dichas dificultades y crear código concurrente limpio. Por último, finalizaremos con los problemas relacionados con la prueba de código concurrente.

La concurrencia limpia es un tema complejo, merecedor de un libro propio. Aquí, intentaremos ofrecer una visión general, que después ampliaremos en el apéndice A. Si simplemente tiene curiosidad por el tema, le bastará con este capítulo. Si necesita entender la concurrencia a un nivel más profundo, consulte también el apéndice.

¿Por qué concurrencia?

La concurrencia es una estrategia de desvinculación. Nos permite desvincular lo que se hace de dónde se hace. En aplicación de un solo proceso, el qué y el cuándo están tan firmemente vinculados que el estado de la aplicación se puede determinar analizando la huella de la pila. Un programador que depure este tipo de sistemas puede definir un punto de interrupción (o varios) y saber el estado de la aplicación en función del punto al que se llegue.

La desvinculación del qué del dónde puede mejorar considerablemente el rendimiento y la estructura de una aplicación. Desde un punto de vista estructural, la aplicación parece una serie de equipos colaboradores y no un gran bucle principal. Esto puede hacer que el sistema sea más fácil de comprender y ofrece diversas formas de separar las preocupaciones. Pongamos por caso el modelo Servlet estándar de aplicaciones Web. Estos sistemas se ejecutan bajo un contenedor Web o EJB que gestiona parcialmente la concurrencia. Los servlet se ejecutan de forma asíncrona cuando se reciben solicitudes Web. El programador de los servlet no tiene

que gestionar todas las solicitudes entrantes. En principio, la ejecución de cada servlet vive en un mundo propio y se desvincula del resto.

Evidentemente, si fuera tan sencillo, no necesitaríamos este capítulo. De hecho, la desvinculación proporcionada por los contenedores Web dista mucho de ser perfecta. Los programadores de servlet deben asegurarse de que sus programas sean correctos. No obstante, las ventajas estructurales del modelo de servlet son significativas.

Pero la estructura no es el único motivo para adoptar la concurrencia. Algunos sistemas tienen limitaciones de tiempo de respuesta y producción que requieren soluciones concurrentes manuales. Imagine un dispositivo para añadir información, con un solo proceso, que obtiene datos de distintos sitios Web y los combina en un resumen diario. Al tener un solo proceso, accede por turnos a cada sitio Web y siempre termina uno antes de comenzar el siguiente. Su recorrido diario debe ejecutarse en menos de 24 horas. Sin embargo, al añadir nuevos sitios Web, el tiempo aumenta hasta necesitarse más de 24 horas para recopilar todos los datos. El único proceso implica una prolongada espera para completar la E/S. Podríamos mejorar el rendimiento con ayuda de un algoritmo de subprocesamiento múltiple que visite más de un sitio Web por vez.

Imagine un sistema que procesa un usuario por vez y sólo requiere un segundo por cada uno. Su capacidad de respuesta es válida para un número reducido de usuarios pero si aumenta, también lo hace el tiempo de respuesta del sistema. Ningún usuario querrá esperar a otros 150. Podríamos mejorar el tiempo de respuesta de este sistema procesando varios usuarios a la vez. Imagine un sistema que interprete grandes conjuntos de datos pero que sólo ofrezca una solución completa tras procesarlos todos. Se podría procesar cada conjunto de datos en un equipo distinto, para poder procesarlos todos en paralelo.

Mitos e imprecisiones

También existen motivos evidentes para adoptar la concurrencia aunque, como indicamos antes, sea complicada. Si no presta la suficiente atención, pueden darse casos desagradables. Veamos los mitos e imprecisiones más habituales:

- *La concurrencia siempre mejora el rendimiento*: En ocasiones lo hace pero sólo cuando se puede compartir tiempo entre varios procesos o procesadores. Ninguna situación es trivial.
- *El diseño no cambia al crear programas concurrentes*: De hecho, el diseño de un algoritmo concurrente puede ser muy distinto al de un sistema de un solo proceso. La desvinculación entre el qué y el cuándo suele tener un efecto importante en la estructura del sistema.
- *No es importante entender los problemas de concurrencia al trabajar con un contenedor Web o EJB*: En realidad, debe saber lo que hace su contenedor y protegerlo de problemas de actualizaciones concurrentes y bloqueo, como veremos después.

Veamos otros aspectos relacionados con la creación de *software* concurrente:

- *La concurrencia genera cierta sobrecarga*, tanto en rendimiento como en la creación de código adicional.
- *La concurrencia correcta es compleja*, incluso para problemas sencillos.
- *Los errores de concurrencia no se suelen repetir*, de modo que se ignoran[75] en lugar de considerarse verdaderos problemas.
- *La concurrencia suele acarrear un cambio fundamental de la estrategia de diseño*.

Desafíos

¿Qué hace que la programación concurrente sea tan complicada? Fíjese en la siguiente clase:

```
public class X {
    private int lastIdUsed;

    public int getNextId() {
        return ++lastIdUsed;
    }
}
```

Imagine que creamos una instancia `X`, establecemos el campo `lastIdUsed` en 42 y después compartimos la instancia entre dos procesos. Imagine ahora que esos dos procesos invocan el método `getNextId()`; hay tres resultados posibles:

- El primer proceso obtiene el valor 43, el segundo el valor 44 y `lastIdUsed` es 44.
- El primer proceso obtiene el valor 44, el segundo el valor 43 y `lastIdUsed` es 44.
- El primer proceso obtiene el valor 43, el segundo el valor 43 y `lastIdUsed` es 43.

El sorprendente tercer resultado[76] se produce cuando los dos procesos coinciden. Se debe a que pueden adoptar varias rutas posibles en una línea de código de Java y algunas generan resultados incorrectos. ¿Cuántas rutas distintas existen? Para responder, debemos entender lo que hace el compilador justo a tiempo con el código de *bytes* generado, y lo que el modelo de memoria de Java considera atómico.

Una rápida respuesta, con el código de *bytes* generado, es que existen 12 870 rutas de ejecución diferentes[77] para los dos procesos ejecutados en el método `getNextId`. Si el tipo de `lastIdUsed` cambia de `int` a `long`, el número de rutas asciende a 2 704 156. Evidentemente, muchas generan resultados válidos. El problema es que *algunas no lo hacen*.

Principios de defensa de la concurrencia

A continuación le mostramos una serie de principios y técnicas para proteger a sus sistemas de los problemas del código concurrente.

Principio de responsabilidad única (SRP)

SRP[78] establece que un método, clase o componente sólo debe tener un motivo para cambiar. El diseño de concurrencia es lo bastante complejo como para ser un motivo de cambio con derecho propio y, por tanto, debe separarse del resto del código. Desafortunadamente, es habitual incrustar los detalles de la implementación de concurrencia directamente en otro código de producción.

Tenga en cuenta los siguientes aspectos:

- *El código relacionado con la concurrencia tiene su propio ciclo de desarrollo*, cambios y ajustes.
- *El código relacionado con la concurrencia tiene sus propios desafíos*, diferentes y más complicados, que los del código no relacionado con la concurrencia.
- El número de formas en las que el código incorrecto basado en la concurrencia puede fallar lo complica ya de por sí, sin la carga añadida del código de aplicación circundante.

Recomendación: *Separe el código de concurrencia del resto del código*[79].

Corolario: Limitar el ámbito de los datos

Como hemos visto, dos procesos que modifican el mismo campo u objeto compartido pueden interferir entre ellos y provocar un comportamiento inesperado. Una solución consiste en usar la palabra clave `synchronized` para proteger una sección importante del código que use el objeto compartido, aunque conviene limitar la cantidad de estas secciones. Cuantos más puntos actualicen datos compartidos, es más probable que:

- Se olvide de proteger uno o varios de esos puntos, y se dañe el código que modifica los datos compartidos.
- Se duplique el esfuerzo necesario para garantizar la protección de todos los elementos (incumplimiento de DRY[80]).
- Resulta complicado determinar el origen de los fallos, que por naturaleza son difíciles de detectar.

Recomendación: *Encapsule los datos y limite el acceso a los datos compartidos*.

Corolario: Usar copias de datos

Una forma de evitar datos compartidos es no compartirlos. En algunos casos se pueden copiar objetos y procesarlos como si fueran de sólo lectura. En otros, se pueden copiar objetos, recopilar los resultados de varios

procesos en las copias y después combinar los resultados en un mismo proceso. Si existe una forma sencilla de evitar los objetos compartidos, el código resultante tendrá menos problemas. Puede que le preocupe el coste de la creación de objetos adicionales. Merece la pena experimentar y comprobar si es un problema real. No obstante, si el uso de copias de objetos permite al código evitar la sincronización, las ventajas de evitar el bloque compensan la creación adicional y la sobrecarga de la recolección de elementos sin usar.

Corolario: Los procesos deben ser independientes

Pruebe a crear el código de sus procesos de forma que cada uno sea independiente y no comparta datos con otros. Cada uno procesa una solicitud cliente y todos los datos necesarios provienen de un origen sin compartir y se almacenan como variables locales. De este modo, los procesos se comportan como si fueran los únicos del mundo y no existieran requisitos de sincronización. Por ejemplo, las subclases de `HttpServlet` reciben toda su información como parámetros pasados en los métodos `doGet` y `doPost`. Esto hace que cada servlet actúe como si dispusiera de su propio equipo. Mientras el código del servlet sólo use variables locales, es imposible que cause problemas de sincronización. Evidentemente, muchas aplicaciones que usan `servlet` se topan con recursos compartidos como conexiones de base de datos.

Recomendación: *Intente dividir los datos en subconjuntos independientes que se puedan procesar en procesos independientes, posiblemente en distintos procesadores.*

Conocer las bibliotecas

Java 5 ofrece muchas mejoras para el desarrollo concurrente con respecto a versiones anteriores. Existen diversos aspectos que tener en cuenta a la hora de crear código de procesos en Java 5:

- Usar las colecciones compatibles con procesos proporcionadas.
- Usar la estructura de ejecución de tareas no relacionadas.

- Usar soluciones antibloqueo siempre que sea posible.
- Varias clases de bibliotecas no son compatibles con procesos.

Colecciones compatibles con procesos

En los albores de Java, Doug Lea escribió el conocido libro[81] *Concurrent Programming in Java*. Al mismo tiempo, desarrolló varias colecciones compatibles con procesos, que posteriormente pasaron a formar parte del JDK en el paquete `java.util.concurrent`. Las colecciones de dicho paquete son compatibles con casos de procesos múltiples y tienen un rendimiento adecuado. De hecho, la implementación `ConcurrentHashMap` tiene mejor rendimiento que `HashMap` en la mayoría de los casos. También permite lecturas y escrituras simultáneas, y dispone de métodos que admiten operaciones de composición habituales que en caso contrario serian incompatibles con subprocesos. Si Java 5 es su entorno de desarrollo, comience con `ConcurrentHashMap`.

Existen otras clases añadidas para admitir diseño avanzado de concurrencia. Veamos algunos ejemplos:

`ReentrantLock`	Bloqueo que se puede adquirir en un método y liberar en otro.
`semaphore`	Una implementación del clásico semáforo, un bloqueo con un contador.
`CountDownLatch`	Bloqueo que espera un número de eventos antes de liberar todos los subprocesos retenidos. De este modo todos tienen la misma oportunidad de iniciarse al mismo tiempo.

Recomendación: *Revise las clases de las que disponga. En el caso de Java, debe familiarizarse con java.util.concurrent, java.util.concurrent.atomic y java.util.concurrent.locks*.

Conocer los modelos de ejecución

Existen diversas formas de dividir el comportamiento de una aplicación concurrente. Para describirlos debe conocer ciertas definiciones básicas.

`Recursos vinculados`	Recursos de tamaño o número fijo usados en un entorno concurrente, como por ejemplo conexiones de base de datos y búfer de lectura/escritura de tamaño fijo.
`Exclusión mutua`	Sólo un proceso puede acceder a datos o a un recurso compartido por vez.
`Inanición`	Se impide que un proceso o grupo de procesos continúen demasiado tiempo o indefinidamente. Por ejemplo, si permite primero la ejecución de los procesos más rápidos, los que se ejecutan durante más tiempo pueden perecer de inanición si los primeros no terminan nunca.
`Bloqueo`	Dos o más procesos esperan a que ambos terminen. Cada proceso tiene un recurso y ninguno puede terminar hasta que obtenga el otro recurso.
`Bloqueo activo`	Procesos bloqueados, intentando realizar su labor pero estorbándose unos a otros. Por motivos de resonancia, los procesos siguen intentando avanzar pero no pueden durante demasiado tiempo, o de forma indefinida.

Tras mostrar estas definiciones, ya podemos describir los distintos modelos de ejecución empleados en la programación concurrente.

Productor-Consumidor[82]

Uno o varios procesos productores crean trabajo y lo añaden a un búfer o a una cola. Uno o varios procesos consumidores adquieren dicho trabajo de la cola y lo completan. La cola entre productores y consumidores es un recurso vinculado, lo que significa que los productores deben esperar a que se libere espacio en la cola antes de escribir y los consumidores deben esperar hasta que haya algo que consumir en la cola. La coordinación entre productores y consumidores a través de la cola hace que unos emitan señales a otros. Los productores escriben en la cola e indican que ya no está vacía. Los consumidores leen de la cola e indican que ya no está llena. Ambos esperan la notificación para poder continuar.

Lectores-Escritores[83]

Cuando un recurso compartido actúa básicamente como fuente de información para lectores pero ocasionalmente se actualiza por parte de escritores, la producción es un problema. El énfasis de la producción puede provocar la inanición y la acumulación de información caducada. Las actualizaciones pueden afectar a la producción. La coordinación de lectores para que no lean algo que un escritor está actualizando y viceversa es complicada. Los escritores tienden a bloquear a los lectores durante periodos prolongados, lo que genera problemas de producción.

El desafío consiste en equilibrar las necesidades de ambos para satisfacer un funcionamiento correcto, proporcionar una producción razonable y evitar la inanición. Una sencilla estrategia hace que los escritores esperen hasta que deje de haber lectores antes de realizar una actualización. Si hay lectores continuos, los escritores perecen de inanición.

Por otra parte, si hay escritores frecuentes y se les asigna prioridad, la producción se ve afectada. Determinar el equilibrio y evitar problemas de actualización concurrente es el objetivo de este modelo.

La cena de los filósofos[84]

Imagine varios filósofos sentados alrededor de una mesa redonda. A la izquierda de cada uno hay un tenedor. En el centro de la mesa, una gran fuente de espaguetis. Los filósofos pasan el tiempo pensando a menos que tengan hambre. Cuando tienen hambre, utilizan los tenedores situados a ambos lados para comer. No pueden comer a menos que tengan dos tenedores. Si el filósofo situado a la derecha o izquierda de otros ya tiene uno de los tenedores que necesita, tendrá que esperar a que termine de comer y deje los tenedores. Cuando un filósofo termina de comer, vuelve a colocar los tenedores en la mesa hasta que vuelve a tener hambre. Cambie los filósofos por procesos y los tenedores por recursos y tendrá un problema habitual en muchas aplicaciones en las que los procesos compiten por recursos. A menos que se diseñen correctamente, los sistemas que compiten de esta forma experimentan problemas de bloqueo, bloqueo mutuo, producción y degradación de la eficacia. La mayoría de problemas de concurrencia que encontrará serán alguna variante de éstos. Analice los

algoritmos y cree soluciones propias para estar preparado cuando surjan problemas de concurrencia.

Recomendación: *Aprenda estos algoritmos básicos y comprenda sus soluciones*.

Dependencias entre métodos sincronizados

Las dependencias entre métodos sincronizados generan sutiles errores en el código concurrente. Java cuenta con `synchronized`, que protege métodos individuales. No obstante, si hay más de un método sincronizado en la misma clase compartida, puede que su sistema sea incorrecto[85].

Recomendación: *Evite usar más de un método en un objeto compartido*.

En ocasiones tendrá que usar más de un método en un objeto compartido. En ese caso, hay tres formas de crear código correcto:

- **Bloqueo basado en clientes**: El cliente debe bloquear al servidor antes de invocar el primer método y asegurarse de que el alcance del bloque incluye el código que invoque el último método.
- **Bloqueo basado en servidores**: Debe crear un método en el servidor que bloquee el servidor, invoque todos los métodos y después anule el bloqueo. El cliente debe invocar el nuevo método.
- **Servidor adaptado**: Cree un intermediario que realice el bloque. Es un ejemplo de bloqueo basado en servidores en el que el servidor original no se puede modificar.

Reducir el tamaño de las secciones sincronizadas

La palabra clave `synchronized` presenta un bloqueo. Todas las secciones de código protegidas por el mismo bloque sólo tendrán un proceso que las ejecute en un momento dado. Los bloqueos son costosos ya que generan retrasos y añaden sobrecarga. Por ello, no conviene colapsar el código con instrucciones `synchronized`. Por otra parte, las secciones críticas[86] deben

protegerse, de modo que debemos diseñar nuestro código con el menor número posible de secciones críticas.

Algunos programadores intentan lograrlo ampliando el tamaño de sus secciones críticas. Sin embargo, al ampliar la sincronización más allá de la sección crítica mínima aumentan los problemas y afecta negativamente al rendimiento[87].

Recomendación: *Reduzca al máximo el tamaño de las secciones* `synchronized`.

Crear código de cierre correcto es complicado

Crear un sistema activo y que se ejecute indefinidamente es distinto a crear algo que funcione de forma temporal y después se cierre correctamente. Entre los problemas más habituales destacan los bloqueos[88], con procesos que esperan una señal para continuar que nunca se produce.

Imagine, por ejemplo, un sistema con un proceso principal que genera varios procesos secundarios y que espera a que todos terminen antes de liberar sus recursos y cerrarse. ¿Qué sucede si uno de los procesos secundarios está bloqueado? El principal esperará indefinidamente y el sistema nunca se cerrará.

Imagine ahora un sistema similar al que se le indica que se cierre. El proceso principal indica a todos los secundarios que abandonen sus tareas y terminen. Pero imagine que dos procesos secundarios funcionan como par productor/consumidor y que el productor recibe una señal del principal y se cierra rápidamente. El consumidor espera un mensaje del productor y puede quedar bloqueado en un estado en el que no recibe la señal del principal, lo que también impide que éste finalice.

Son situaciones habituales. Por tanto, si tiene que crear código concurrente con cierres correctos, tendrá que dedicar tiempo a que el cierre se produzca de forma correcta.

Recomendación: *Planifique con antelación el proceso de cierre y pruébelo hasta que funcione. Le llevará más tiempo del que espera. Repase los algoritmos existentes porque será complicado*.

Probar código con procesos

Demostrar que el código es correcto no resulta práctico. Las pruebas no garantizan su corrección. Sin embargo, las pruebas adecuadas pueden minimizar los riesgos, en especial en aplicaciones de un solo proceso. Cuando hay dos o más procesos que usan el mismo código y trabajan con datos compartidos, la situación se vuelve más compleja.

Recomendación: *Cree pruebas que puedan detectar problemas y ejecútelas periódicamente, con distintas configuraciones de programación y del sistema, y cargas. Si las pruebas fallan, identifique el fallo. No lo ignore porque las pruebas superen una ejecución posterior*.

Hay muchos factores que tener en cuenta. Veamos algunas recomendaciones concretas:

- Considere los fallos como posibles problemas de los procesos.
- Consiga que primero funcione el código sin procesos.
- El código con procesos se debe poder conectar a otros elementos.
- El código con procesos debe ser modificable.
- Ejecute con más procesos que procesadores.
- Ejecute en diferentes plataformas.
- Diseñe el código para probar y forzar fallos.

Considerar los fallos como posibles problemas de los procesos

El código con procesos hace que fallen elementos que no deberían fallar. Muchos desarrolladores desconocen cómo interactúan los procesos con otro tipo de código. Los problemas del código con procesos pueden mostrar sus síntomas una vez cada mil o un millón de ejecuciones.

Los intentos por repetir los sistemas pueden resultar frustrantes, lo que suele provocar que los programadores consideren el fallo como algo aislado. Es recomendable asumir que los fallos aislados no existen. Cuanto más los ignore, mayor será la cantidad de código que se acumule sobre un enfoque defectuoso.

Recomendación: *No ignore los fallos del sistema como algo aislado*.

Conseguir que primero funcione el código sin procesos

Puede parecer evidente pero no está de más recordarlo. Asegúrese de que el código funciona fuera de sus procesos. Por lo general, esto significa crear algunos POJO que los procesos deban invocar. Los POJO no son compatibles con los procesos y por tanto se pueden probar fuera de su entorno. Conviene incluir en los POJO la mayor cantidad posible del sistema.

Recomendación: *No intente identificar fallos de procesos y que no sean de procesos al mismo tiempo. Asegúrese de que su código funciona fuera de los procesos*.

El código con procesos se debe poder conectar a otros elementos

Cree el código compatible con la concurrencia de forma que se pueda ejecutar en distintas configuraciones:

- Un proceso, varios procesos y variarlo durante la ejecución.
- El código con procesos interactúa con algo que puede ser real o probado.
- Ejecutar con pruebas dobles ejecutadas de forma rápida, lenta y variable.
- Configurar pruebas que ejecutar en diferentes iteraciones.

Recomendación: *El código con procesos debe poder conectar a otros elementos y ejecutar en distintas configuraciones*.

El código con procesos debe ser modificable

La obtención del equilibrio adecuado de procesos suele requerir operaciones de ensayo y error. En las fases iniciales, compruebe el rendimiento del sistema bajo diferentes configuraciones. Permita que se puedan modificar los distintos procesos y también durante la ejecución del sistema. También puede permitir la modificación automática en función de la producción y la utilización del sistema.

Ejecutar con más procesos que procesadores

Cuando el sistema cambia de tarea, se producen reacciones. Para promover el intercambio de tareas, realice la ejecución con más procesos que procesadores o núcleos. Cuanto mayor sea la frecuencia de intercambio de las tareas, más probabilidades existen de que el código carezca de una sección crítica o se produzcan bloqueos.

Ejecutar en diferentes plataformas

En 2007 diseñamos un curso sobre programación concurrente, principalmente en OS X. La clase se presentó con Windows XP ejecutado en una MV. Se crearon pruebas para ilustrar condiciones de fallo que fallaban con más frecuencia en OS X que en XP.

En todos los casos, el código probado era incorrecto. Esto refuerza el hecho de que cada sistema operativo tiene una política de procesos diferente que afecta a la ejecución del código. El código con procesos múltiples se comporta de forma distinta en cada entorno[89]. Debe ejecutar sus pruebas en todos los entornos de implementación posibles.

Recomendación: *Ejecute el código con procesos en todas las plataformas de destino con frecuencia y en las fases iniciales*.

Diseñar el código para probar y forzar fallos

Es habitual que los fallos del código concurrente se oculten. Las pruebas sencillas no suelen mostrarlos. En realidad, suelen ocultarse durante el procesamiento normal. Pueden aparecer horas, días o semanas después.

La razón de que los problemas de procesos sean infrecuentes, esporádicos y apenas se repitan es que sólo fallan algunas de las miles de rutas posibles que recorren una sección vulnerable. Por tanto, la probabilidad de adoptar una ruta fallida es realmente baja, lo que dificulta la detección y la depuración.

Se preguntará cómo aumentar las posibilidades de capturar estos casos. Puede diseñar el código y forzarle a que se ejecute en diferentes órdenes añadiendo métodos como `Object.wait()`, `Object.sleep()`, `Object.yield()` y `Object.priority()`.

Estos métodos afectan al orden de ejecución y, por tanto, aumentan las posibilidades de detectar un error. Resulta más adecuado que el código incorrecto falle lo antes posible y con frecuencia. Hay dos opciones de instrumentación de código:

- Manual.
- Automática.

Manual

Puede añadir invocaciones de `wait()`, `sleep()`, `yield()` y `priority()` manualmente a su código, en especial si tiene que probar un fragmento especialmente escabroso. Veamos un ejemplo:

```
public synchronized String nextUrlOrNull() {
    if (hasNext()) {
        String url = urlGenerator.next();
        Thread.yield(); // se añade para pruebas.
        updateHasNext();
        return url;
    }
    return null;
}
```

La invocación de `yield()` cambia la ruta de ejecución adoptada por el código y posiblemente hace que el código falla donde no lo hacía antes. Si el código falla, no se debe a la invocación de yield() añadida[90]. Se debe a que el código es incorrecto y hemos hecho que el fallo sea más evidente. Este enfoque presenta varios problemas:

- Tendrá que buscar manualmente los puntos adecuados donde hacerlo.
- ¿Cómo sabe dónde incluir la invocación y qué tipo de invocación usar?
- La presencia de este código en un entorno de producción ralentiza innecesariamente el código.
- Es un enfoque que puede o no detectar los fallos; de hecho, no las tiene todas consigo.

Lo que necesitamos es una forma de hacerlo durante la fase de pruebas, no de producción. También debemos poder mezclar configuraciones entre ejecuciones, lo que aumenta las probabilidades de detectar los errores.

Evidentemente, si dividimos el sistema POJO que no sepa nada los procesos en clases que controlen los procesos, resultará más sencillo ubicar los puntos en los que instrumentar el código. Es más, podríamos crear diferentes pruebas que invoquen los POJO bajo distintos regímenes de invocaciones a `sleep`, `yield` y demás.

Automática

Puede usar herramientas como la estructura orientada a aspectos, CGLIB o ASM para instrumentar su código mediante programación. Por ejemplo, podría usar una clase con un único método:

```
public class ThreadJigglePoint {
    public static void jiggle() {
    }
}
```

Puede añadir invocaciones en distintos puntos del código:

```
public synchronized String nextUrlOrNull() {
    if(hasNext()) {
        ThreadJigglePoint.jiggle();
        String url = urlGenerator.next();
        ThreadJigglePoint.jiggle();
        updateHasNext();
        ThreadJigglePoint.jiggle();
        return url;
    }
    return null;
}
```

Tras ello, use un sencillo aspecto que seleccione aleatoriamente entre no hacer nada, pausar o generar un resultado.

Imagine que la clase `ThreadJigglePoint` tiene dos implementaciones. La primera implementa `jiggle` para no hacer nada y se usa en producción. La segunda genera un número aleatorio para elegir entre `sleep`, `yield` o nada. Si ejecuta sus pruebas mil veces con `jiggle` de forma aleatoria, puede descubrir algunos fallos. Si la prueba es satisfactoria, al menos puede felicitarse por haber actuado correctamente. Aunque sea un tanto simple, puede resultar una opción razonable en lugar de recurrir a una herramienta más sofisticada. La herramienta ConTest[91], desarrollada por IBM, tiene un funcionamiento similar pero es más sofisticada.

El objetivo es que los procesos del código se ejecuten en distinto orden en momentos diferentes. La combinación de pruebas bien escritas y ejecuciones aleatorias puede aumentar considerablemente la capacidad de detectar errores.

Recomendación: *Use estas estrategias para detectar errores.*

Conclusión

Es complicado conseguir código concurrente correcto. El código sencillo se puede complicar al añadir varios procesos y datos compartidos. Si tiene que crear código concurrente, tendrá que hacerlo con rigor o se enfrentará a sutiles y esporádicos fallos.

En primer lugar, siga el principio de responsabilidad única. Divida su sistema en varios POJO que separen el código compatible con procesos del resto. Asegúrese de probar únicamente el código compatible con procesos y nada más, por lo que este código debe ser de tamaño reducido y específico.

Conozca los orígenes de los problemas de concurrencia: varios procesos que operen en datos compartidos o usen una agrupación de recursos común. Los casos de límites, como el cierre correcto o la conclusión de la iteración de un bucle, pueden ser especialmente espinosos.

Conozca su biblioteca y los algoritmos fundamentales. Debe comprender cómo las funciones de la biblioteca permiten resolver problemas similares a los de los algoritmos fundamentales.

Aprenda a localizar regiones del código que se puedan bloquear y bloquéelas. No bloquee otras regiones que no lo necesiten. Evite invocar una sección bloqueada desde otra. Para ello debe saber si un elemento está compartido o no. Reduzca la cantidad de objetos compartidos y su ámbito. Cambie los diseños de los objetos con datos compartidos para acomodar clientes en lugar de obligar a los clientes a gestionar el estado compartido.

Los problemas se acumularán. Los que no aparezcan inicialmente suelen considerarse esporádicos y suelen producirse en la fase de carga o de modo aparentemente aleatorio. Por tanto, debe poder ejecutar su código con procesos en diferentes configuraciones y plataformas de forma repetida y continua. La capacidad de prueba, algo natural si aplica las tres leyes de TDD, implica cierto nivel de conectividad, lo que ofrece la compatibilidad necesaria para ejecutar código en distintas configuraciones.

La probabilidad de detectar errores mejora si se toma el tiempo necesario para instrumentar su código. Puede hacerlo manualmente o mediante tecnologías automatizadas. Hágalo en las fases iniciales. Es aconsejable ejecutar el código basado en procesos durante el mayor tiempo posible antes de pasarlo a producción.

Si adopta un enfoque limpio, aumentarán las probabilidades de hacerlo de forma correcta.

Bibliografía

- **[Lea99]**: *Concurrent Programming in Java: Design Principles and Patterns*, 2d. ed., Doug Lea, Prentice Hall, 1999.
- **[PPP]**: *Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.
- **[PRAG]**: *The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

14
Refinamiento sucesivo

Caso práctico de un analizador de argumentos de línea de comandos

Este capítulo es un caso práctico de refinamiento sucesivo. Veremos un módulo que comienza correctamente pero no mantiene dicha corrección. Tras ello, veremos cómo se refactoriza y se limpia. Muchos hemos tenido que analizar argumentos de línea de comando. Si no disponemos de una utilidad para ello, recorremos la matriz de cadenas pasadas a la función

principal. Puede encontrar utilidades de calidad pero ninguna hace exactamente lo que necesitamos. Por ello, decidí crear una propia, a la que he denominado `Args`. `Args` es muy fácil de usar. Basta crearla con los argumentos de entrada y una cadena de formato, y después consultar a la instancia de `Args` los valores de los argumentos. Fíjese en el siguiente ejemplo:

Listado 14-1
Uso de Args

```
public static void main(String[] args) {
    try {
        Args arg = new Args("l,p#,d*", args);
        boolean logging = arg.getBoolean('l');
        int port = arg.getInt('p');
        String directory = arg.getString('d');
        executeApplication(logging, port, directory);
    } catch (ArgsException e) {
        System.out.printf("Argument error: %s\n", e.errorMessage());
    }
}
```

Comprobará lo sencillo que es. Creamos una instancia de la clase `Args` con dos parámetros. El primero es la cadena de formato o esquema: `"l,p#,d*"`. Define tres argumentos de línea de comandos. El primero, `-l`, es un argumento booleano. El segundo, `-p`, es un argumento entero. El tercero, `-d`, es un argumento de cadena. El segundo parámetro del constructor `Args` es la matriz de argumentos de línea de comandos pasada a `main`. Si el constructor no genera `ArgsException`, la línea de comandos entrante se ha analizado y se puede consultar la instancia `Args`. Se usan métodos como `getBoolean`, `getInteger` y `getString` para acceder a los valores de los argumentos por sus nombres.

Si hay un problema, ya sea en la cadena de formato o en los argumentos de línea de comandos, se genera `ArgsException`. La descripción del error se puede recuperar del método `errorMessage` de la excepción.

Implementación de Args

El Listado 14-2 es la implementación de la clase `Args`. Examínela con atención. El estilo y la estructura se han trabajado concienzudamente y espero que los imite.

Listado 14-2
Args.java

```
package com.objectmentor.utilities.args;

import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;
import java.util.*;

public class Args {
    private Map<Character, ArgumentMarshaler> marshalers;
    private Set<Character> argsFound;
    private ListIterator<String> currentArgument;

    public Args(String schema, String[] args) throws ArgsException {
        marshalers = new HashMap<Character, ArgumentMarshaler>();
        argsFound = new HashSet<Character>();

        parseSchema(schema);
        parseArgumentStrings(Arrays.asList(args));
    }

    private void parseSchema(String schema) throws ArgsException {
        for (String element : schema.split(“,”))
            if (element.length() > 0)
                parseSchemaElement(element.trim());
    }

    private void parseSchemaElement(String element) throws ArgsException {
        char elementId = element.charAt(0);
        String elementTail = element.substring(1);
        validateSchemaElementId(element Id);
        if (elementTail.length() == 0)
            marshalers.put(elementId, new BooleanArgumentMarshaler());
        else if (elementTail.equals(“*”))
            marshalers.put(elementId, new StringArgumentMarshaler());
        else if (elementTail.equals(“#”))
            marshalers.put(elementId, new IntegerArgumentMarshaler());
        else if (elementTail.equals(“##”))
            marshalers.put(elementId, new DoubleArgumentMarshaler());
        else if (elementTail.equals(“[*]”))
            marshalers.put(elementId, new StringArrayArgumentMarshaler());
        else
            throw new ArgsException(INVALID_ARGUMENT_FORMAT, elementId, elementTail);
    }

    private void validateSchemaElementId(char elementId) throws ArgsException {
        if {!Character.isLetter(elementId))
            throw new ArgsException(INVALID_ARGUMENT_NAME, elementId, null);
    }

    private void parseArgumentStrings(List<String> argsList) throws ArgsException
    {
        for (currentArgument = argsList.listIterator(); currentArgument.hasNext();)
        {
            String argString = currentArgument.next();
            if (argString.startsWith(“-”)) {
                parseArgumentCharacters(argString.substring(1));
            } else {
                currentArgument.previous();
                break;
            }
        }
    }

    private void parseArgumentCharacters(String argChars) throws ArgsException {
        for (int i = 0; i < argChars.length(); i++)
            parseArgumentCharacter(argChars.charAt(i));
    }

    private void parseArgumentCharacter(char argChar) throws ArgsException {
        ArgumentMarshaler m = marshalers.get(argChar);
        if (m == null) {
            throw new ArgsException (UNEXPECTED_ARGUMENT, argChar, null);
        } else {
            argsFound.add(argChar);
            try {
                m.set(currentArgument);
            } catch (ArgsException e) {
                e.setErrorArgumentId(argChar);
                throw e;
            }
        }
    }
```

```
    public boolean has(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));
    }
}
```

Puede leer el código de arriba a abajo sin necesidad de saltar de un punto a otro ni buscar hacia adelante. Lo que seguramente busque es la definición de `ArgumentMarshaler`, que hemos omitido intencionadamente. Tras leer el código, comprenderá la interfaz `ArgumentMarshaler` y la función de sus variantes. Veamos algunas de ellas (entre los listados 14-3 y 14-6).

Listado 14-3
ArgumentMarshaler.java

```
public interface ArgumentMarshaler {
    void set(Iterator<String> currentArgument) throws ArgsException;
}
```

Listado 14-4
BooleanArgumentMarshaler.java

```
public class BooleanArgumentMarshaler implements ArgumentMarshaler {
    private boolean booleanValue = false;

    public void set (Iterator<String> currentArgument) throws ArgsException {
        booleanValue = true;
    }

    public static boolean getValue(ArgumentMarshaler am) {
    if (am != null && am instanceof BooleanArgumentMarshaler)
        return ((BooleanArgumentMarshaler) am).booleanValue;
    else
        return false;
    }
}
```

Listado 14-5
StringArgumentMarshaler.java

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class StringArgumentMarshaler implements ArgumentMarshaler {
    private String stringValue = "";

    public void set(Iterator<String> currentArgument) throws ArgsException {
        try {
            stringValue = currentArgument.next();
        } catch (NoSuchElementException e) {
            throw new ArgsException(MISSING_STRING);
        }
    }

    public static String getValue(ArgumentMarshaler am) {
        if (am != null && am instanceof StringArgumentMarshaler)
            return ((StringArgumentMarshaler) am).stringValue;
        else
            return "";
    }
}
```

Listado 14-6
IntegerArgumentMarshaler.java

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class IntegerArgumentMarshaler implements ArgumentMarshaler {
    private int intValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            IntValue = Integer.parseInt(parameter);
        } catch (NoSuchElementException e) {
            throw new ArgsException(MISSING_INTEGER);
        } catch (NumberFormatException e) {
            throw new ArgsException(INVALID_INTEGER, parameter);
        }
    }

    public static int getValue (ArgumentMarshaler am) {
        if (am != null && am instanceof IntegerArgumentMarshaler)
            return ((IntegerArgumentMarshaler) am).intValue;
        else
            return 0;
    }
}
```

Las otras variantes de `ArgumentMarshaler` simplemente repiten este patrón en matrices `double` y `String` y sólo complicarían el capítulo. Puede consultarlas como ejercicio. Otro fragmento que puede resultar complicado es la definición de las constantes de código de error, incluidas en la clase `ArgsException` (véase el Listado 14-7).

Listado 14-7
ArgsException.java

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class ArgsException extends Exception {
    private char errorArgumentId = '\0';
    private String errorParameter = null;
    private ErrorCode errorCode = OK;

    public ArgsException() {}

    public ArgsException(String message) { super(message); }
```

```
    public ArgsException(ErrorCode errorCode) {
        this.errorCode = errorCode;
    }


    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() {
        switch (errorCode) {
            case OK:
                return “TILT: Should not get here.”;
            case UNEXPECTED_ARGUMENT:
                return String.format(“Argument -%c unexpected.”, errorArgumentId);
            case MISSING_STRING:
                return String.format(“Could not find string parameter for -%c.”,
                        errorArgumentId);
            case INVALID INTEGER:
                return String.format(“Argument -%c expects an integer but was ‘%s’.”,
                        errorArgumentId, errorParameter);
            case MISSING_INTEGER:
                return String.format(“Could not find integer parameter for -%c.”,
                        errorArgumentId);
            case INVALID_DOUBLE:
                return String.format(“Argument -%c expects a double but was ‘%s’.”,
                        errorArgumentId, errorParameter);
            case MISSING_DOUBLE:
                return String.format(“Could not find double parameter for -%c.”,
                        errorArgumentId);
            case INVALID_ARGUMENT_NAME:
                return String.format(“‘%c’ is not a valid argument name.”,
                        errorArgumentId);
            case INVALID_ARGUMENT_FORMAT:
                return String.format(“‘%s’ is not a valid argument format.”,
                        errorParameter);
        }
        return “”;
    }


    public enum ErrorCode {
        OK, INVALID_ARGUMENT_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
        MISSING_STRING,
        MISSING_INTEGER, INVALID_INTEGER,
        MISSING_DOUBLE, INVALID_DOUBLE }
}
```

Es sorprendente la cantidad de código necesario para detallar este sencillo concepto. Uno de los motivos es el uso de un lenguaje especialmente profuso. Java, al ser un lenguaje de tipos estáticos, requiere

muchas palabras para satisfacer el sistema de tipos. En lenguajes como Ruby, Python o Smalltalk, este programa es mucho más reducido[92].

Vuelva a leer el código. Fíjese especialmente en los nombres de los elementos, el tamaño de las funciones y el formato. Si tiene experiencia como programador, partes del estilo o la estructura no le convencerán, pero espero que, desde un punto de vista global, considere que el programa está bien escrito y tiene una estructura limpia.

Por ejemplo, debería ser evidente cómo añadir un nuevo tipo de argumento, como una fecha o un número complejo, y que dicha inclusión apenas requeriría código. En definitiva, bastaría con una nueva variante de `ArgumentMarshaler`, una nueva función `getXXX` y una nueva instrucción `case` en la función `parseSchemaElement`. También habría un nuevo código `ArgsException.ErrorCode` y un nuevo mensaje de error.

Cómo se ha realizado

No diseñé este programa de principio a fin en su forma actual y, sobre todo, no espero que pueda crear programas limpios y elegantes a la primera. Si algo hemos aprendido en las dos últimas décadas es que la programación es un arte más que una ciencia. Para escribir código limpio, primero debe crear código imperfecto y después limpiarlo. No debería sorprenderle. Ya lo aprendimos en el colegio cuando los profesores (normalmente en vano) nos obligaban a crear borradores de nuestras redacciones. El proceso, nos decían, era escribir un primer borrador, después otro, y después otros muchos hasta lograr una versión definitiva. Para escribir redacciones limpias, el refinamiento debía ser continuado.

Muchos programadores noveles (como sucede con los alumnos) no siguen este consejo. Creen que el objetivo principal es que el programa funcione. Una vez que lo consiguen, pasan a la siguiente tarea, y conservan el estado funcional del programa, sea cual sea. Los programadores experimentados saben que esto es un suicidio profesional.

Args: El primer borrador

El Listado 14-8 muestra una versión inicial de la clase `Args`. Funciona, pero es un desastre.

Listado 14-8
Args.java (primer borrador)

```
import java.text.ParseException;
import java.util.*;


public class Args {
    private String schema;
    private String[] args;
    private boolean valid = true;
    private Set<Character> unexpectedArguments = new TreeSet<Character>();
    private Map<Character, Boolean> booleanArgs =
        new HashMap<Character, Boolean>();
    private Map<Character, String> stringArgs =
        new HashMap<Character, String>();
    private Map<Character, Integer> intArgs =
        new HashMap<Character, Integer>();
    private Set<Character> argsFound = new HashSet<Character>();
    private int currentArgument;
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ErrorCode errorCode = ErrorCode.OK;


    private enum ErrorCode {
        OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}


    public Args(String schema. String[] args) throws ParseException {
        this.schema = schema;
        this.args = args;
        valid = parse();
    }


    private boolean parse() throws ParseException {
        if (schema.length() == 0 && args.length == 0)
            return true;
        parseSchema();
        try {
            parseArguments();
        } catch (ArgsException e) {
        }
        return valid;
    }


    private boolean parseSchema() throws ParseException {
        for (String element : schema.split(",")) {
            if (element.length() > 0) {
                String trimmedElement = element.trim();
                parseSchemaElement(trimmedElement);
            }
        }
        return true;
    }


    private void parseSchemaElement(String element) throws ParseException {
        char elementId = element.charAt(0);
        String elemenTail = element.substring(1);
        validateSchemaElementId(elementId);
        if (isBooleanSchemaElement(elementTail));
            parseBooleanSchemaElement(elementId);
        else if (isStringSchemaElement(elementTail))
            parseStringSchemaElement(elementId);
        else if (isIntegerSchemaElement(elementTail)) {
            parseIntegerSchemaElement(elementId);
        } else {
            throw new ParseException(
                String.format("Argument: %c has invalid format: %s.",
                        elementId, elementTail), 0);
        }
    }


    private void validateSchemaElementId(char elementId) throws ParseException {
        if (!Character.isLetter(elementId)) {
            throw new ParseException(
                "Bad character:" + elementId + "in Args format: " + schema, 0);
        }
    }


    private void parseBooleanSchemaElement(char elementId) {
        booleanArgs.put(elementId, false);
    }


    private void parseIntegerSchemaElement(char elementId) {
        intArgs.put(elementId, 0);
    }
```

```
private void parseStringSchemaElement(char elementId) {
    stringArgs.put(elementId, “”);
}


private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals(“*”);
}


private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
}


private boolean isIntegerSchemaElement(String elementTail) {
    return elementTail.equals(“#”);
}


private boolean parseArguments() throws ArgsException {
    for (currentArgument = 0; currentArgument < args.length; currentArgument++)
    {
        String arg = args[currentArgument];
        parseArgument(arg);
    }
    return true;
}


private void parseArgument(String arg) throws ArgsException {
    if (arg.startsWith(“-”))
        parseElements(arg);
}


private void parseElements(String arg) throws ArgsException {
    for (int i = 1; i < arg.length(); i++)
        parseElement(arg.charAt(i));
}


private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
        argsFound.add(argChar);
    else {
        unexpectedArguments.add(argChar);
        errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
        valid = false;
    }
}


private boolean setArgument(char argChar) throws ArgsException {
    if (isBooleanArg(argChar))
        setBooleanArg(argChar, true);
    else if (isStringArg(argChar))
        setStringArg(argChar);
    else if (isIntArg(argChar))
        setIntArg(argChar);
    else
        return false;


    return true;
}


private boolean isIntArg(char argChar) {
    return intArgs.containsKey(argChar);
}


private void setIntArg(char argChar) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
        parameter = args[currentArgument];
        intArgs.put(argChar, new Integer(parameter));
    } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
    } catch (NumberFormatException e) {
        valid = false;
        errorArgumentId = argChar;
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw new ArgsException();
    }
}


private void setStringArg(char argChar) throws ArgsException {
    currentArgument++;
    try {
        stringArgs.put(argChar, args[currentArgument]);
```

```
    } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
    }
}

private boolean isStringArg(char argChar) {
    return stringArgs.containsKey(argChar);
}

private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.put(argChar, value);
}

private boolean isBooleanArg(char argChar) {
    return booleanArgs.containsKey(argChar);
}

public int cardinality() {
    return argsFound.size();
}

public String usage() {
    if (schema.length() > 0)
    return "-[" + schema + “]”;
    else
    return “”;
}

public String errorMessage() throws Exception {
    switch (errorCode) {
        case OK:
            throw new Exception(“TILT: Should not get here.”);
        case UNEXPECTED_ARGUMENT:
            return unexpectedArgumentMessage();
        case MISSING_STRING:
            return String.format(“Could not find string parameter for -%c.”,
                    errorArgumentId);
        case INVALID_INTEGER:
            return String.format(“Argument - %c expects an integer but was ‘%s’.”,
                    errorArgumentId, errorParameter);
        case MISSING_INTEGER:
            return String.format(“Could not find integer parameter for -%c.”,
                    errorArgumentId);
    }
    return “”;
}

private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer(“Arguments(s) -”);
    for (char c : unexpectedArguments) {
        message.append(c);
    }
    message.append(“ unexpected.”);

    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 has(char arg) (
        return argsFound.contains(arg);
    }

    public boolean isValid() {
        return valid;
    }

    private class ArgsException extends Exception {
    }
}
```

Espero que su reacción inicial ante tal cantidad de código es alegrarse por no haberlo conservado tal cual. Si ha sido su reacción, recuerde que será la que tengan otros que lean un borrador de su código.

En realidad, primer borrador es lo mejor que se puede decir sobre este código. Evidentemente es un trabajo en progreso. La cantidad de variables de instancia es apabullante. Cadenas extrañas como «`TILT`», `HashSet` y `TreeSet`, y los bloques `try-catch-catch` aumentan el desastre.

No era mi intención crear este desastre. En realidad, intentaba mantener cierta organización, como demuestra la elección de nombres de funciones y variables, y la estructura del programa. Pero es evidente que el problema se me fue de las manos.

El desastre aumentó gradualmente. Las versiones anteriores no fueron tan malas. Por ejemplo, el Listado 14-9 muestra una versión inicial en la que sólo funcionaban los argumentos booleanos.

Listado 14-9
Args.java (sólo argumentos booleanos)

```
package com.objectmentor.utilities.getopts;

import java.util.*;

public class Args {
    private String schema;
    private String[] args;
    private boolean valid;
    private Set<Character> unexpectedArguments = new TreeSet<Character>();
    private Map<Character, Boolean> booleanArgs =
        new HashMap<Character, Boolean>();
    private int numberOfArguments = 0;

    public Args(String schema, String[] args) {
        this.schema = schema;
        this.args = args;
        valid = parse();
    }

    public boolean isValid() {
        return valid;
    }

    private boolean parse() {
        if (schema.length() == 0 && args.length == 0)
            return true;
        parseSchema();
        parseArguments();
        return unexpectedArguments.size() == 0;
    }
```

```
    private boolean parseSchema() {
        for (String element : schema.split(“,”)) {
            parseSchemaElement(element);
        }
        return true;
    }


    private void parseSchemaElement(String element) {
        if (element.length() == 1) {
            parseBooleanSchemaElement(element);
        }
    }


    private void parseBooleanSchemaElement(String element) {
        char c = element.charAt(0);
        if (Character.isLetter(c)) {
            booleanArgs.put(c, false);
        }
    }


    private boolean parseArguments() {
        for (String arg : args)
            parseArgument(arg);
        return true;
    }


    private void parseArgument(String arg) {
        if (arg.startsWith(“-”))
            parseElement(arg);
    }


    private void parseElements(String arg) {
        for (int i = 1; i < arg.length(); i++)
            parseElement(arg.charAt(i));
    }


    private void parseElement(char argChar) {
        if (isBoolean(argChar)) {
            numberOfArguments++;
            setBooleanArg(argChar, true);
        } else
            unexpectedArguments.add(argChar);
    }


    private void setBooleanArg(char argChar, boolean value) {
        booleanArgs.put(argChar, value);
    }


    private boolean isBoolean(char argChar) {
        return booleanArgs.containsKey(argChar);
    }


    public int cardinality() {
        return numberOfArguments;
    }


    public String usage() {
        if (schema.length() > 0)
            return “-[”+schema+“]”;
        else
            return “”;
    }


    public String errorMessage() {
        if (unexpectedArguments.size() > 0) {
            return unexpectedArgumentMessage();
        } else
            return “”;
    }


    private String unexpectedArgumentMessage() {
        StringBuffer message = new StringBuffer(“Argument(s) -”);
        for (char c : unexpectedArguments) {
            message.append(c);
        }
        message.append(“ unexpected.”);


        return message.toString();
    }


    public boolean getBoolean(char arg) {
        return booleanArgs.get(arg);
    }
}
```

Aunque hay motivos para quejarse del código, no es tan malo. Es compacto y sencillo, y fácil de entender. Sin embargo, en este código se aprecia la semilla del desastre posterior y resulta evidente porqué.

La versión posterior sólo tiene dos tipos de argumentos más que ésta: `String` e `integer`. La inclusión de sólo dos tipos más tiene un tremendo impacto negativo en el código. Lo convierte de algo que sería razonablemente mantenible en algo que seguramente esté plagado de errores.

Añadí los dos tipos de argumento de forma incremental. Primero, el argumento `String`, que genera lo siguiente:

Listado 14-10
Args.java (booleano y String)

```
package com.objectmentor.utilities.getopts;


import java.text.ParseException;
import java.util.*;


public class Args {
    private String schema;
    private String[] args;
    private boolean valid = true;
    private Set<Character> unexpectedArguments = new TreeSet<Character>();
    private Map<Character, Boolean> booleanArgs =
        new HashMap<Character, Boolean>();
    private Map<Character, String> stringArgs =
        new HashMap<Character, String>();
    private Set<Character> argsFound = new HashSet<Character>();
    private int currentArgument;
    private char errorArgument = '\0';


    enum ErrorCode { OK, MISSING_STRING }


    private ErrorCode errorCode = ErrorCode.OK;


    public Args(String schema, String[] args) throws ParseException {
        this.schema = schema;
        this.args = args;
        valid = parse();
    }


    private boolean parse() throws ParseException {
        if (schema.length() == 0 && args.length == 0)
            return true;
        parseSchema();
        parseArguments();
        return valid;
    }


    private boolean parseSchema() throws ParseException {
        for (String element : schema.split(",")) {
            if (element.length() > 0) {
                String trimmedElement = element.trim();
                parseSchemaElement(trimmedElement);
            }
        }
        return true;
    }


    private void parseSchemaElement(String element) throws ParseException {
        char elementId = element.charAt(0);
        String elementTail = element.substring(1);
        validateSchemaElementId(elementId);
        if (isBooleanSchemaElement(elementTail))
            parseBooleanSchemaElement(elementId);
        else if (isStringSchemaElement(elementTail))
            parseStringSchemaElement(elementId);
```

```
}


private void validateSchemaElementId(char elementId) throws ParseException {
    if (!Character.isLetter(elementId)) {
        throw new ParseException(
            “Bad character:” + elementId + “in Args format: ” + schema, 0);
    }
}


private void parseStringSchemaElement(char elementId) {
    stringArgs.put(elementId, “”);
}


private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals(“*”);
}


private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
}


private void parseBooleanSchemaElement(char elementId) {
    booleanArgs.put(elementId, false);
}


private boolean parseArguments() {
    for (currentArgument = 0; currentArgument < args.length; currentArgument++)
    {
        String arg = args[currentArgument];
        parseArgument(arg);
    }
    return true;
}


private void parseArgument(String arg) {
    if (arg.startsWith(“-”))
        parseElements(arg);
}


private void parseElements(String arg) {
    for (int i = 1; i < arg.length(); i++)
        parseElement(arg.charAt(i));
}


private void parseElement(char argChar) {
    if (setArgument(argChar))
        argsFound.add(argChar);
    else {
        unexpectedArguments.add(argChar);
        valid = false;
    }
}


private boolean setArgument(char argChar) {
    boolean set = true;
    if (isBoolean(argChar))
        setBooleanArg(argChar, true);
    else if (isString(argChar))
        setStringArg (argChar, “”);
    else
        set = false;


    return set;
}


private void setStringArg(char argChar, String s) {
    currentArgument++;
    try {
        stringArgs.put(argChar, args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgument = argChar;
        errorCode = ErrorCode.MISSING_STRING;
    }
}


private boolean isString(char argChar) {
    return stringArgs.containsKey(argChar);
}


private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.put(argChar, value);
}
```

```
    private boolean isBoolean(char argChar) {
        return booleanArgs.containsKey(argChar);
    }


    public int cardinality() {
        return argsFound.size();
    }


    public String usage() {
        if (schema.length() > 0)
            return "-[" + schema + "]";
        else
            return "";
    }


    public String errorMessage() throws Exception {
        if (unexpectedArguments.size() > 0) {
            return unexpectedArgumentMessage();
        } else
            switch (errorCode) {
                case MISSING_STRING:
                return String.format ("Could not find string parameter for -%c.",
                        errorArgument);
                case OK:
                throw new Exception("TILT: Should not get here.");
            }
        return "";
    }


    private String unexpectedArgumentMessage() {
        StringBuffer message = new StringBuffer("Argument(s) -");
        for (char c : unexpectedArguments) {
            message.append(c);
        }
        message.append(" unexpected.");


        return message.toString();
    }


    public boolean getBoolean(char arg) {
        return falseIfNull(booleanArgs.get(arg));
    }


    private boolean falseIfNull(Boolean b) {
        return b == null ? false : b;
    }


    public String getString(char arg) {
        return blankIfNull(stringArgs.get(arg));
    }


    private String blankIfNull(String s) {
        return s == null ? "" : s;
    }


    public boolean has(char arg) {
        return argsFound.contains(arg);
    }


    public boolean isValid() {
        return valid;
    }
}
```

Comprobará que empieza a desbocarse. No es terrible pero el desastre se está gestando. Basta con incluir el tipo de argumento `integer` para que resulte fatídico.

Entonces me detuve

Todavía tenía que añadir otros dos tipos de argumentos y sabía que empeorarían las cosas. Si los forzaba, seguramente funcionarían pero

provocaría un desastre demasiado complicado de arreglar. Si la estructura del código tenía que poder mantenerse, era el momento de corregirla.

Por ello dejé de añadir elementos y comencé la refactorización. Tras añadir los argumentos `String` e `integer`, sabía que cada uno necesitaría nuevo código en tres puntos principales. En primer lugar, cada tipo de argumento necesita una forma de analizar su elemento de esquema para poder seleccionar el `HashMap` de ese tipo. Tras ello, sería necesario analizar cada tipo de argumento en las cadenas de línea de comandos y convertirlos en su tipo correcto. Por último, cada tipo de argumento necesitaría un método `getXXX` para poder devolverlo al invocador como su tipo correcto.

Muchos tipos diferentes y todos con métodos similares, lo que en realidad era una clase. Y de este modo nació el concepto de `ArgumentMarshaler`.

Sobre el incrementalismo

Una de las mejores formas de acabar con un programa es realizar cambios masivos con la intención de mejorarlo. Algunos programas nunca se recuperan de estas mejoras. El problema es lo complicado que resulta conseguir que el programa funcione de la misma forma que antes de la mejora.

Para evitarlo, recurro a la disciplina TDD (*Test-Driven Development* o Desarrollo guiado por pruebas). Una de las doctrinas centrales de este enfoque es mantener la ejecución del sistema en todo momento. Es decir, con TDD no puedo realizar cambios que afecten al funcionamiento del sistema. Todos los cambios deben mantenerlo como antes de los cambios. Para lograrlo, necesito una serie de pruebas automatizadas que ejecutar rápidamente y que verifiquen que el comportamiento del sistema no ha variado. Para la clase `Args`, creé una serie de pruebas de unidad y aceptación. Las pruebas de unidad se crearon en Java y se administraron con JUnit. Las pruebas de aceptación se crearon como páginas wiki en FitNesse. Podría haber ejecutado estas pruebas en cualquier momento y, si eran satisfactorias, sabría que el sistema funcionaba de la forma especificada.

Así pues, comencé a realizar pequeños cambios. Cada uno desplazaba la estructura del sistema hacia el concepto `ArgumentMarshaler`, y cada cambio mantenía el funcionamiento del sistema. El primer cambio realizado

fue añadir el esqueleto de `ArgumentMarshaller` al final del desastre anterior (véase el Listado 14-11).

Listado 14-11
ArgumentMarshaller añadido a Args.java

```
private class ArgumentMarshaler {
    private boolean booleanValue = false;

    public void setBoolean(boolean value) {
        booleanValue = value;
    }

    public boolean getBoolean() { return booleanValue; }

    private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    }

    private class StringArgumentMarshaler extends ArgumentMarshaler {
    }

    private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    }
}
```

Evidentemente, esto no afectaría a nada, por lo que realicé la modificación más sencilla posible que afectara a la mínima cantidad de código. Cambié `HashMap` para que los argumentos Boolean aceptaran `ArgumentMarshaler`.

```
private Map<Character, ArgumentMarshaler> booleanArgs =
    new HashMap<Character, ArgumentMarshaler>();
```

Esto afectaba a varias instrucciones que corregí rápidamente.

```
…
    private void parseBooleanSchemaElement(char elementId) {
        booleanArgs.put(elementId, new BooleanArgumentMarshaler());
    }
…
    private void setBooleanArg(char argChar, boolean value) {
        booleanArgs.get(argChar).setBoolean(value);
    }
…
    public boolean getBoolean(char arg) {
        return falseIfNull (booleanArgs.get(arg).getBoolean());
    }
```

Estos cambios se aplican a las zonas que mencionamos antes: `parse`, `set` y `get` para el tipo de argumento. Desafortunadamente, aunque sean cambios menores, algunas de las pruebas comenzaron a fallar. Si se fija atentamente en `getBoolean`, comprobará que se puede invocar con `y` pero no existe un argumento `y`, por lo que `booleanArgs.get('y')` devolverá `null` y la función generará `NullPointerException`. La función `falseIfNull` se usa como protección ante este hecho pero el cambio aplicado hace que la función sea irrelevante.

El incrementalismo exigía que esto funcionara antes de realizar otros cambios. La solución no era demasiado complicada; bastaba con cambiar la

comprobación de `null`. Ya no era necesario comprobar `null` en boolean, sino en `ArgumentMarshaller`.

Primero, eliminé la invocación de `falseIfNull` en la función `getBoolean`. Ya no servía de nada, por lo que eliminé directamente la función. Las pruebas seguían fallando igual, lo que suponía que no había nuevos errores.

```
public boolean getBoolean(char arg) {
    return booleanArgs.get(arg).getBoolean();
}
```

Tras ello, dividí la función en dos líneas y añadí `ArgumentMarshaller` a una variable propia: `argumentMarshaller`. No me preocupaba el extenso nombre de la variable; era redundante y estorbaba a la función, por lo que lo reduje a `am` [N5].

```
public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am.getBoolean();
}
```

Y tras ello añadí la lógica de detección de `null`.

```
public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am != null && am.getBoolean();
}
```

Argumentos de cadena

La inclusión de los argumentos `String` fue similar a la de los argumentos `boolean`. Tuve que cambiar `HashMap` y conseguir que funcionaran `parse`, `set` y `get`. No deberían producirse sorpresas posteriores a excepción de que la implementación completa se incluía en la clase `ArgumentMarshaller` en lugar de distribuirla en variantes.

```
    private Map<Character, ArgumentMarshaler> stringArgs =
        new HashMap<Character, ArgumentMarshaler>();
…
    private void parseStringSchemaElement(char elementId) {
        stringArgs.put(elementId, new StringArgumentMarshaler());
    }
…
    private void setStringArg(char argChar) throws ArgsException {
        currentArgument++;
        try {
            stringArgs.get(argChar).setString(args[currentArgument]);
        } catch (ArrayIndexOutOfBoundsException e) {
            valid = false;
            errorArgumentId = argChar;
            errorCode = ErrorCode.MISSING_STRING;
            throw new ArgsException();
        }
    }
…
    public String getString (char arg) {
        Args.ArgumentMarshaler am = stringArgs.get(arg);
        return am == null ? "" : am.getString();
    }
…
    private class ArgumentMarshaler {
        private boolean booleanValue = false;
        private String stringValue;
```

```
        public void setBoolean(boolean value) {
            booleanValue = value;
        }

        public boolean getBoolean() {
            return booleanValue;
        }

        public void setString(String s) {
            stringValue = s;
        }

        public String getString() {
            return stringValue == null ? "" : stringValue;
        }
    }
```

De nuevo, estos cambios se realizaron individualmente para conservar las pruebas, aunque fallaran. Si una prueba fallaba, me aseguraba de que fuera correcta antes de continuar con el siguiente cambio.

Ya debería reconocer mi intención. Tras incluir el comportamiento de señalización en la clase base `ArgumentMarshaler`, comencé a transferirlo a las variantes, para de esta forma mantener el funcionamiento mientras cambiaba gradualmente la forma del programa.

El siguiente paso consistía en transferir la funcionalidad del argumento `int` a `ArgumentMarshaler`. De nuevo, no hubo sorpresas.

```
private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character, ArgumentMarshaler>();
…
    private void parseIntegerSchemaElement(char elementId) {
        intArgs.put(elementId, new IntegerArgumentMarshaler());
    }
…
    private void setIntArg(char argChar) throws ArgsException {
        currentArgument++;
        String parameter = null;
        try {
            parameter = args[currentArgument];
            intArgs.get(argChar).setInteger(Integer.parseInt(parameter));
        } catch (ArrayIndexOutOfBoundsException e) {
            valid = false;
            errorArgumentId = argChar;
            errorCode = ErrorCode.MISSING_INTEGER;
            throw new ArgsException();
        } catch (NumberFormatException e) {
            valid = false;
            errorArgumentId = argChar;
            errorParameter = parameter;
            errorCode = ErrorCode.INVALID_INTEGER;
            throw new ArgsException();
        }
    }
…
    public int getInt(char arg) {
        Args.ArgumentMarshaler am = intArgs.get(arg);
        return am == null ? 0 : am.getInteger();
    }
…
    private class ArgumentMarshaler {
        private boolean booleanValue = false;
        private String stringValue;
        private int integerValue;

        public void setBoolean(boolean value) {
            booleanValue = value;
        }

        public boolean getBoolean() {
            return booleanValue;
        }

        public void setString(String s) {
            stringValue = s;
        }
```

```
        public String getString() {
            return stringValue == null ? "" : stringValue;
        }

        public void setInteger(int i) {
            integerValue = i;
        }

        public int getInteger() {
            return integerValue;
        }
    }
```

Tras transferir la señalización a `ArgumentMarshaler`, comencé a transferir la funcionalidad a las variantes. El primer paso fue pasar la función `setBoolean` a `BooleanArgumentMarshaller` y garantizar su correcta invocación. Para ello creé un método `set` abstracto.

```
private abstract class ArgumentMarshaler {
    protected boolean booleanValue = false;
    private String stringValue;
    private int integerValue;

    public void setBoolean(boolean value) {
        booleanValue = value;
    }

    public boolean getBoolean() {
        return booleanValue;
    }

    public void setString(String s) {
        stringValue = s;
    }

    public String getString() {
        return stringValue == null ? "" : stringValue;
    }

    public void set Integer(int i) {
        integerValue = i;
    }

    public int getInteger() {
        return integerValue;
    }

    public abstract void set(String s);
}
```

Tras ello, implementé el método `set` en `BooleanArgumentMarshaller`.

```
private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    public void set(String s) {
        booleanValue = true;
    }
}
```

Y por último cambié la invocación de `setBoolean` por la de `set`.

```
private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.get(argChar).set("true");
}
```

Las pruebas seguían siendo satisfactorias. Como este cambio hacía que `set` se implementara en `BooleanArgumentMarshaler`, eliminé el método `setBoolean` de la clase base `ArgumentMarshaler`.

La función abstracta `set` acepta un argumento `String` pero la implementación de `BooleanArgumentMarshaler` no lo usa. He incluido el

argumento porque sabía que `StringArgumentMarshaler` e `IntegerArgumentMarshaler` lo utilizarían.

Tras ello, el objetivo era implementar el método `get` en `BooleanArgumentMarshaler`. La implementación de funciones get siempre es escabrosa ya que el tipo devuelto tiene que ser `Object` y en este caso debe convertirse a `Boolean`.

```
public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am != null && (Boolean)am.get();
}
```

Para compilarlo, añadí la función `get` a `ArgumentMarshaler`.

```
private abstract class ArgumentMarshaler {
    …

    public Object get() {
        return null;
    }
}
```

Se compila y las pruebas fallan. Para que vuelvan a funcionar, basta con convertir `get` en abstracto e implementarlo en `BooleanArgumentMarshaler`.

```
private abstract class ArgumentMarshaler {
    protected boolean booleanValue = false;
    …

    public abstract Object get();
}

private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    public void set (String s) {
        booleanValue = true;
    }

    public Object get() {
        return booleanValue;
    }
}
```

De nuevo, las pruebas son satisfactorias. Ahora tanto `get` como `set` se implementan en `BooleanArgumentMarshaler`. Esto me permite eliminar la antigua función `getBoolean` de `ArgumentMarshaler`, cambiar la variable protegida `booleanValue` a `BooleanArgumentMarshaler` y convertirla en privada.

Repetí el mismo patrón de cambios con las cadenas. Implementé `set` y `get`, eliminé las funciones sin usar y desplacé las variables.

```
    private void setStringArg(char argChar) throws ArgsException {
        currentArgument++;
        try {
            stringArgs.get(argChar).set(args[currentArgument]);
        } catch (ArrayIndexOutOfBoundsException e) {
            valid = false;
            errorArgumentId = argChar;
            errorCode = ErrorCode.MISSING_STRING;
            throw new ArgsException();
        }
    }


…
    public String getString(char arg) {
        Args.ArgumentMarshaler am = stringArgs.get(arg);
```

```
        return am == null ? "" : (String) am.get();
    }
…

    private abstract class ArgumentMarshaler {
        private int integerValue;

        public void setInteger(int i) {
            integerValue = i;
        }

        public int getInteger() {
            return integerValue;
        }

        public abstract void set(String s);

        public abstract Object get();
    }

    private class BooleanArgumentMarshaler extends ArgumentMarshaler {
        private boolean booleanValue = false;

        public void set(String s) {
            booleanValue = true;
        }

        public Object get() {
            return booleanValue;
        }
    }

    private class StringArgumentMarshaler extends ArgumentMarshaler {
        private String stringValue = "";

        public void set(String s) {
            stringValue = s;
        }

        public Object get() {
            return stringValue;
        }
    }

    private class IntegerArgumentMarshaler extends ArgumentMarshaler {

        public void set(String s){

        }

        public Object get() {
            return null;
        }
    }
}
```

Por último, repetí el proceso con los enteros. Resulta más complicado ya que los enteros deben analizarse y la operación de análisis puede generar una excepción, pero el resultado es más indicado ya que el concepto de `NumberFormatException` se oculta totalmente en `IntegerArgumentMarshaler`.

```
private boolean isIntArg(char argChar) { return intArgs.containsKey(argChar); }

    private void setIntArg(char argChar) throws ArgsException {
        currentArgument++;
        String parameter = null;
        try {
            parameter = args[currentArgument];
            intArgs.get(argChar).set(parameter);
        } catch (ArrayIndexOutOfBoundsException e) {
            valid = false;
            errorArgumentId = argChar;
            errorCode = ErrorCode.MISSING_INTEGER;
            throw new ArgsException();
        } catch (ArgsException e) {
```

```
            valid = false;
            errorArgumentId = argChar;
            errorParameter = parameter;
            errorCode = ErrorCode.INVALID_INTEGER;
            throw e;
        }
    }
…
    private void setBooleanArg(char argChar) {
        try {
            booleanArgs.get(argChar).set("true");
        } catch (ArgsException e) {
        }
    }
…
    public int getInt(char arg) {
        Args.ArgumentMarshaler am = intArgs.get(arg);
        return am == null ? 0 : (Integer) am.get();
    }
…
    private abstract class ArgumentMarshaler {
        public abstract void set(String s) throws ArgsException;
        public abstract Object get();
    }
…
    private class IntegerArgumentMarshaler extends ArgumentMarshaler {
        private int intValue = 0;

        public void set(String s) throws ArgsException {
            try {
                intValue = Integer.parseInt(s);
            } catch (NumberFormatException s) {
                throw new ArgsException();
            }
        }

        public Object get() {
            return intValue;
        }
    }
```

Evidentemente, las pruebas seguían funcionando. Tras ello, me deshice de las distintas asignaciones de la parte superior del algoritmo, lo que hace que el sistema sea mucho más genérico. Sin embargo, no las puede eliminar ya que afectaría a la integridad del sistema. En su lugar, añadí un nuevo `Map` para `ArgumentMarshaler` y, tras ello, cambié uno a uno los métodos para que usaran la nueva asignación en lugar de las originales.

```
public class Args {
…
    private Map<Character, ArgumentMarshaler> booleanArgs =
        new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> stringArgs =
        new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> intArgs =
        new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> marshalers =
        new HashMap<Character, 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 = new StringArgumentMarshaler();
        stringArgs.put(elementId, m);
        marshalers.put(elementId, m);
    }
```

Las pruebas seguían funcionando. Tras ello, cambié `isBooleanArg` de esto:

```
private boolean isBooleanArg(char argChar) {
    return booleanArgs.containsKey(argChar);

}
```

a este otro:

```
private boolean isBooleanArg(char argChar) {
    ArgumentMarshaler m = marshalers.get(argChar);
    return m instanceof BooleanArgumentMarshaler;
}
```

Las pruebas funcionaban, por lo que apliqué el mismo cambio en `isIntArg` e `isStringArg`.

```
private boolean isIntArg(char argChar) {
    ArgumentMarshaler m = marshalers.get(argChar);
    return m instanceof IntegerArgumentMarshaler;
}


private boolean isStringArg(char argChar) {
    ArgumentMarshaler m = marshalers.get(argChar);
    return m instanceof StringArgumentMarshaler;
}
```

Las pruebas eran correctas, por lo que eliminé las invocaciones duplicadas de `marshalers.get`:

```
private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (isBooleanArg(m))
        setBooleanArg(argChar);
    else if (isStringArg(m))
        setStringArg(argChar);
    else if (isIntArg(m))
        setIntArg(argChar);
    else
        return false;


    return true;
}


private boolean isIntArg (ArgumentMarshaler m) {
    return m instanceof IntegerArgumentMarshaler;
}


private boolean isStringArg (ArgumentMarshaler m) {
    return m instanceof StringArgumentMarshaler;
}


private boolean isBooleanArg (ArgumentMarshaler m) {
    return m instanceof BooleanArgumentMarshaler;
}
```

Los tres argumentos `isxxxArg` ya no tenían sentido, de modo que los reubiqué:

```
private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m instanceof BooleanArgumentMarshaler)
        setBooleanArg(argChar);
    else if (m instanceof StringArgumentMarshaler)
        setStringArg(argChar);
    else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(argChar);
    else
        return false;


    return true;
}
```

Tras ello, empecé a usar la asignación `marshalers` en las funciones `set`, dividiendo el uso de las otras tres asignaciones. Comencé por los elementos `boolean`.

```
    private boolean setArgument(char argChar) throws ArgsException {
        ArgumentMarshaler m = marshalers.get(argChar);
```

```
        if (m instanceof BooleanArgumentMarshaler)
            setBooleanArg(m);
        else if (m instanceof StringArgumentMarshaler)
            setStringArg(argChar);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(argChar);
        else
            return false;
        return true;
    }

…

    private void setBooleanArg(ArgumentMarshaler m) {
        try {
            m.set("true"); // era: booleanArgs.get(argChar).set("true");
        } catch (ArgsException e) {
        }
    }
```

Las pruebas seguían siendo correctas de modo que repetí la operación con las cadenas y los enteros. De esta manera se puede integrar parte del desagradable código de gestión de excepciones en la función `setArgument`.

```
private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    try {
        if (m instanceof BooleanArgumentMarshaler)
            setBooleanArg(m);
        else if (m instanceof StringArgumentMarshaler)
            setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(m);
        else
            return false;
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}

private void setIntArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
        parameter = args[currentArgument];
        m.set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
    } catch (ArgsException e) {
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw e;
    }
}

private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    try {
        m.set(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
    }
}
```

Ya podía eliminar las tres asignaciones antiguas. Primero, debía cambiar la función `getBoolean` de:

```
public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am != null && (Boolean) am.get();
}
```

a:

```
public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    boolean b = false;
    try {
        b = am != null && (Boolean) am.get();
    } catch (ClassCastException e) {
        b = false;
```

```
        }
        return b;
    }
```

Este último cambio puede parecer sorprendente. ¿Por qué de repente decidí enfrentarme a `ClassCastException`? Por tener una serie de pruebas de unidad y otra serie independiente de pruebas de aceptación creadas en FitNesse. Las pruebas de FitNesse garantizan que si se invoca `getBoolean` en un argumento no Booleano, se obtiene `false`. No sucede lo mismo con las pruebas de unidad. Hasta el momento, sólo había ejecutado las pruebas de unidad[93].

Este último cambio me permitió extraer otro uso de la asignación boolean:

```
private void parseBooleanSchemaElement(char elementId) {
    ArgumentMarshaler m = new BooleanArgumentMarshaler();
    booleanArgs.put(elementId, m);
    marshalers.put(elementId, m);
}
```

Y ahora ya podemos eliminar la asignación boolean.

```
public class Args {
…
    private Map<Character, ArgumentMarshaler> booleanArgs =
    new HashMap<Character, ArgmentMarshaler>();
    private Map<Character, ArgumentMarshaler> stringArgs =
    new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
…
```

Tras ello, cambié los argumentos `String` e `Integer` de la misma forma y limpié los valores `boolean`.

```
private void parseBooleanSchemaElement(char elementId) {
    marshalers.put(elementId, new BooleanArgumentMarshaler());
}


private void parseIntegerSchemaElement(char elementId) {
    marshalers.put(elementId, new IntegerArgumentMarshaler());
}


private void parseStringSchemaElement(char elementId) {
        marshalers.put(elementId, new StringArgumentMarshaler());
}
…
public String getString(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
        return am null ? “” : (String) am.get();
    } catch (ClassCastException e) {
        return “”;
    }
}


public int getInt(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
        return am == null ? 0 : (Integer) am.get();
    } catch (Exception e) {
        return 0;
    }
}
…
public class Args {
…
    private Map<Character, ArgumentMarshaler> stringArgs =
    new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
```

…

Seguidamente, dispuse en línea los tres métodos `parse` ya que no servían para mucho:

```
private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
        marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (isStringSchemaElement(elementTail))
        marshalers.put(elementId, new StringArgumentMarshaler());
    else if (isIntegerSchemaElement(elementTail)) {
        marshalers.put(elementId, new IntegerArgumentMarshaler());
    } else {
        throw new ParseException(String.format(
        "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
    }
}
```

Es el momento de ver la estructura completa. El Listado 14-12 muestra la clase `Args` actual.

Listado 14-12
Args.java (tras la primera refactorización)

```
package com.objectmentor.utilities.getopts;

import java.text.ParseException;
import java.util.*;

public class Args {
    private String schema;
    private String[] args;
    private boolean valid = true;
    private Set<Character> unexpectedArguments = new TreeSet<Character>();
    private Map<Character, ArgumentMarshaler> marshalers =
        new HashMap<Character, ArgumentMarshaler>();
    private Set<Character> argsFound = new HashSet<Character>();
    private int currentArgument;
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ErrorCode errorCode = ErrorCode.OK;

    private enum ErrorCode {
        OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}

    public Args(String schema, String[] args) throws ParseException {
        this.schema = schema;
        this.args = args;
        valid = parse();
    }

    private boolean parse() throws ParseException {
        if (schema.length() == 0 && args.length == 0)
            return true;
        parseSchema();
        try {
            parseArguments();
        } catch (ArgsException e) {
        }
        return valid;
    }

    private boolean parseSchema() throws ParseException {
        for (String element : schema.split(",")) {
            if (element.length() > 0) {
                String trimmedElement = element.trim();
                parseSchemaElement(trimmedElement);
            }
        }
        return true;
    }

    private void parseSchemaElement(String element) throws ParseException {
        char elementId = element.charAt(0);
        String elementTail = element.substring(1);
```

```
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
        marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (isStringSchemaElement(elementTail))
        marshalers.put(elementId, new StringArgumentWarshaler());
    else if (isIntegerSchemaElement(elementTail)) {
        marshalers.put(elementId, new IntegerArgumentMarshaler());
    } else {
        throw new ParseException(String.format(
        "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
    }
}


private void validateSchemaElementId(char elementId) throws ParseException {
    if (!Character.isLetter(elementId)) {
        throw new ParseException(
        "Bad character:" + elementId + "in Args format: " + schema, 0);
    }
}


private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals("*");
}


private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
}


private boolean isIntegerSchemaElement(String elementTail) {
    return elementTail.equals("#");
}


private boolean parseArguments() throws ArgsException {
    for (currentArgument=0; currentArgument<args.length; currentArgument++) {
        String arg = args[currentArgument];
        parseArgument(arg);
    }
    return true;
}


private void parseArgument(String arg) throws ArgsException {
    if (arg.startsWith("-"))
        parseElements(arg);
}


private void parseElements(String arg) throws ArgsException {
    for (int i = 1; i < arg.length(); i++)
        parseElement(arg.charAt(i));
}


private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
        argsFound.add(argChar);
    else {
        unexpectedArguments.add(argChar);
        errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
        valid = false;
    }
}


private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    try {
        if (m instanceof BooleanArgumentMarshaler)
            setBooleanArg(m);
        else if (m instanceof StringArgumentMarshaler)
            setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(m);
        else
            return false;
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}


private void setIntArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
        parameter = args[currentArgument];
        m.set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
```

```
    } catch (ArgsException e) {
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw e;
    }
}

private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    try {
        m.set (args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
    }
}

private void setBooleanArg(ArgumentMarshaler m) {
    try {
        m.set("true");
    } catch (ArgsException e) {
    }
}

public int cardinality() {
    return argsFound.size();
}

public String usage() {
    if (schema.length() > 0)
        return "-[" + schema + "]";
    else
        return "";
}

public String errorMessage() throws Exception {
    switch (errorCode) {
        case OK:
            throw new Exception("TILT: Should not get here.");
        case UNEXPECTED_ARGUMENT:
            return unexpectedArgumentMessage();
        case MISSING_STRING:
            return String.format("Could not find string parameter for -%c.",
                    errorArgumentId);
        case INVALID_INTEGER:
            return String.format("Argument -%c expects an integer but was '%s'.",
                    errorArgumentId, errorParameter);
        case MISSING_INTEGER:
            return String.format("Could not find integer parameter for -%c.",
                    errorArgumentId);
    }
    return "";
}

private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for {char c : unexpectedArguments) {
        message.append(c);
    }
    message.append(" unexpected.");


    return message.toString();
}

public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    boolean b = false;
    try {
        b = am != null && (Boolean) am.get();
    } catch (ClassCastException e) {
        b = false;
    }
    return b;
}

public String getString(char arg) {
    Args.ArgumentMarshaler am = marshalers.get (arg);
    try {
        return am == null ? "" : (String) am.get();
    } catch (ClassCastException e) {
        return "";
    }
}

public int getInt(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
        return am == null ? 0 : (Integer) am.get();
```

```
            } catch (Exception e) {
                return 0;
            }
        }

        public boolean has(char arg) {
            return argsFound.contains(arg);
        }

        public boolean isValid() {
            return valid;
        }

        private class ArgsException extends Exception {
        }

        private abstract class ArgumentMarshaler {
            public abstract void set(String s) throws ArgsException;
            public abstract Object get();
        }

        private class BooleanArgumentMarshaler extends ArgumentMarshaler {
            private boolean booleanValue = false;

            public void set(String s) {
                booleanValue = true;
            }

            public Object get() {
                return booleanValue;
            }
        }

        private class StringArgumentMarshaler extends ArgumentMarshaler {
            private String stringValue = "";

            public void set(String s) {
                stringValue = s;
            }

            public Object get() {
                return stringValue;
            }
        }

        private class IntegerArgumentMarshaler extends ArgumentMarshaler {
            private int intValue = 0;

            public void set(String s) throws ArgsException {
                try {
                    intValue = Integer.parseInt(s);
                } catch (NumberFormatException e) {
                    throw new ArgsException();
                }
            }

            public Object get() {
                return intValue;
            }
        }
    }
```

Tras todo este esfuerzo, es un tanto decepcionante. La estructura ha mejorado pero todavía hay demasiadas variables en la parte superior; se mantiene un terrible caso de tipos en `setArgument`; y todas las funciones `set`. Sin mencionar el procesamiento de errores. Todavía nos queda mucho trabajo por hacer.

Mi intención es eliminar el caso de tipos de `setArgument` [G23] y que sólo incluya una invocación a `ArgumentMarshaler.set`. Para ello, debo desplazar `setIntArg`, `setStringArg` y `setBooleanArg` a las correspondientes variantes de `ArgumentMarshaler`. Pero hay un problema.

Si se fija atentamente en `setIntArg`, comprobará que usa dos variables de instancia: `args` y `currentArg`. Para desplazar `setIntArg` hasta `BooleanArgumentMarshaler`, tengo que pasar `args` y `currentArgs` como argumentos de función. Muy desagradable [F1]. Resultaría más indicado pasar un argumento y no dos. Afortunadamente, la solución es sencilla. Podemos convertir la matriz `args` en `list` y pasar `Iterator` a las funciones `set`. Para el siguiente cambio necesité diez pasos, y superar todas las pruebas tras cada uno. Pero sólo mostraremos el resultado. Debería determinar la mayoría de estos pequeños pasos.

```
public class Args {
    private String schema;
    private String[] args;
    private boolean valid = true;
    private Set<Character> unexpectedArguments = new TreeSet<Character>();
    private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
    private Set<Character> argsFound = new HashSet<Character>();
    private Iterator<String> currentArgument;
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ErrorCode errorCode = ErrorCode.OK;
    private List<String> argsList;


    private enum ErrorCode {
        OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT }


    public Args(String schema. String[] args) throws ParseException {
        this.schema = schema;
        argsList = Arrays.asList(args);
        valid = parse();
    }


    private boolean parse() throws ParseException {
        if (schema.length() == 0 && argsList.size() == 0)
            return true;
        parseSchema();
        try {
            parseArguments();
        } catch (ArgsException e) {
        }
        return valid;
    }
…
    private boolean parseArguments() throws ArgsException {
        for (currentArgument = argsList.iterator(); currentArgument.hasNext();) {
            String arg = currentArgument.next();
            parseArgument(arg);
        }


        return true;
    }
…
    private void setIntArg(ArgumentMarshaler m) throws ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            m.set(parameter);
        } catch (NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_INTEGER;
            throw new ArgsException();
        } catch (ArgsException e) {
            errorParameter = parameter;
            errorCode = ErrorCode.INVALID_INTEGER;
            throw e;
        }
    }


    private void setStringArg(ArgumentMarshaler m) throws ArgsException {
        try {
            m.set (currentArgument.next());
        } catch (NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_STRING;
            throw new ArgsException();
        }
    }
```

Son pequeños cambios que conservan el funcionamiento de las pruebas. Ahora podemos empezar a desplazar las funciones `set` a las correspondientes variantes. Primero, debemos realizar el siguiente cambio en `setArgument`:

```
private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;
    try {
        if (m instanceof BooleanArgumentMarshaler)
            setBooleanArg(m);
        else if (m instanceof StringArgumentMarshaler)
            setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(m);
        else
            return false;
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}
```

Es un cambio importante ya que queremos eliminar totalmente la cadena `if-else`. Por tanto, debemos excluir la condición de error.

Ya podemos empezar a desplazar las funciones `set`. La función `setBooleanArg` es trivial, de modo que la prepararemos en primer lugar. El objetivo es cambiar la función `setBooleanArg` para redirigirla a `BooleanArgumentMarshaler`.

```
private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;
    try {
        if (m instanceof BooleanArgumentMarshaler)
            setBooleanArg(m, currentArgument);
        else if (m instanceof StringArgumentMarshaler)
            setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(m);


    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}
---
private void setBooleanArg (ArgumentMarshaler m,
                            Iterator<String> currentArgument)
                            throws ArgsException {
    try {
        m.set("true");
    catch (ArgsException e) {
    }
}
```

¿No acabamos de incluir el procesamiento de excepciones? Añadir elementos para después excluirlos es habitual en los procesos de refactorización. Los pasos reducidos y la necesidad de que las pruebas sigan siendo correctas implican que los elementos cambien de posición. La refactorización es como resolver el cubo de Rubik. Se necesitan muchos pasos pequeños para lograr un objetivo mayor. Cada paso habilita el siguiente.

Se preguntará por qué pasamos `iterator` si `setBooleanArg` no lo necesita. Pues porque `setIntArg` y `setStringArg` sí. Y como el objetivo es implementar las tres funciones a través de un método abstracto en `ArgumentMarshaller`, es necesario pasarlo a `setBooleanArg`.

Ahora `setBooleanArg` no sirve de nada. Si hubiera una función `set` en `ArgumentMarshaler`, podríamos invocarla directamente. Es el momento de crear dicha función. El primer paso consiste en añadir el nuevo método abstracto a `ArgumentMarshaler`.

```
private abstract class ArgumentMarshaler {
    public abstract void set(Iterator<String> currentArgument)
                                throws ArgsException;
    public abstract void set (String s) throws ArgsException;
    public abstract Object get();
}
```

Evidentemente, esto afecta a todas las variantes, de modo que implementamos el nuevo método en cada una.

```
private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    private boolean booleanValue = false;

    public void set(Iterator<String> currentArgument) throws ArgsException {
        booleanValue = true;
    }

    public void set(String s) {
        booleanValue = true;
    }

    public Object get() {
        return booleanValue;
    }
}

private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = "";

    public void set(Iterator<String> currentArgument) throws ArgsException {
    }

    public void set(String s) {
        stringValue = s;
    }

    public Object get() {
        return stringValue;
    }
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
    }

    public void set(String s) throws ArgsException {
        try {
            intValue = Integer.parseInt(s);
        } catch (NumberFormatException e) {
            throw new ArgsException();
        }
    }

    public Object get() {
        return intValue;
    }
}
```

Y ahora ya podemos eliminar `setBooleanArg`:

```
private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;
    try {
        if (m instanceof BooleanArgumentMarshaler)
            m.set(currentArgument);
        else if (m instanceof StringArgumentMarshaler)
            setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(m);


    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}
```

Las pruebas siguen siendo satisfactorias y la función `set` se implementa en Boolean `ArgumentMarshaler`. Podemos repetir la operación con las cadenas y los enteros.

```
private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;
    try {
        if (m instanceof BooleanArgumentMarshaler)
        m.set(currentArgument);
        else if (m instanceof StringArgumentMarshaler)
        m.set(currentArgument);
        else if (m instanceof IntegerArgumentMarshaler)
        m.set(currentArgument);


    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}
---
private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = "";


    public void set(Iterator<String> currentArgument) throws ArgsException {
        try {
            stringValue = currentArgument.next();
        } catch (NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_STRING;
            throw new ArgsException();
        }
    }


    public void set(String s){
    }


    public Object get() {
        return stringValue;
    }
}


private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;


    public void set(Iterator<String> currentArgument) throws ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            set(parameter);
        } catch (NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_INTEGER;
            throw new ArgsException();
        } catch (ArgsException e) {
            errorParameter = parameter;
            errorCode = ErrorCode.INVALID_INTEGER;
            throw e;
        }
    }


    public void set(String s) throws ArgsException {
        try {
            intValue = Integer.parseInt(s);
```

```
        } catch (NumberFormatException e) {
            throw new ArgsException();
        }
    }

    public Object get() {
        return intValue;
    }
}
```

Y el golpe de gracia: se elimina el caso de tipos.

```
private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;

    try {
        m.set(currentArgument);
        return true;
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
}
```

Ya podemos deshacernos de las funciones de `IntegerArgumentMarshaler` y limpiar el resto.

```
private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0

    public void set (Iterator<String> currentArgument) throws ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            intValue = Integer.parseInt(parameter);
        } catch (NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_INTEGER;
            throw new ArgsException();
        } catch (NumberFormatException e) {
            errorParameter = parameter;
            errorCode = ErrorCode.INVALID_INTEGER;
            throw new ArgsException();
        }
    }

    public Object get() {
        return intValue;
    }
}
```

También podemos convertir `ArgumentMarshaler` en una interfaz.

```
private interface ArgumentMarshaler {
    void set (Iterator<String> currentArgument) throws ArgsException;
    Object get();
}
```

Veamos ahora lo sencillo que resulta añadir un nuevo tipo de argumento a la estructura. Apenas necesitaremos cambios y los que apliquemos tendrán que ser aislados. En primer lugar, añadimos un nuevo caso de prueba para comprobar que el argumento `double` funciona correctamente:

```
public void testSimpleDoublePresent() throws Exception {
    Args args = new Args("x##", new String[] {"-x","42.3"});
    assertTrue(args.isValid());
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals(42.3, args.getDouble('x'), .001);
}
```

Limpiamos el código de análisis de esquemas y añadimos la detección `##` para el tipo de argumento `double`.

```
private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (elementTail.length() == 0)
        marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals("*"))
        marshalers.put(elementId, new StringArgumentMarshaler());
    else if (elementTail.equals("#"))
        marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals("##"))
        marshalers.put(elementId, new DoubleArgumentMarshaler());
    else
        throw new ParseException(String.format(
            "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
}
```

Seguidamente, creamos la clase `DoubleArgumentMarshaler`.

```
private class DoubleArgumentMarshaler implements ArgumentMarshaler {
    private double doubleValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            doubleValue = Double.parseDouble(parameter);
        } catch (NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_DOUBLE;
            throw new ArgsException();
        } catch (NumberFormatException e) {
            errorParameter = parameter;
            errorCode = ErrorCode.INVALID_DOUBLE;
            throw new ArgsException();
        }
    }

    public Object get() {
        return doubleValue;
    }
}
```

Esto nos obliga a añadir un nuevo código de error (`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);
    try {
        return am = null ? 0 : (Double) am.get();
    } catch (Exception e) {
        return 0.0;
    }
}
```

Y todas las pruebas son correctas. Ha sido sencillo. A continuación comprobamos que el procesamiento de errores funciona correctamente. El siguiente caso de prueba comprueba que se declare un error si se proporciona una cadena que no se puede analizar a un argumento `##`.

```
public void testInvalidDouble() throws Exception {
    Args args = new Args("x##", new String[] {"-x", "Forty two"});
    assertFalse(args.isValid());
    assertEquals(0, args.cardinality());
    assertFalse(args.has('x'));
    assertEquals(0, args.getInt('x'));
    assertEquals("Argument -x expects a double but was 'Forty two'.",
                 args.errorMessage());
}
---
public String errorMessage() throws Exception {
    switch (errorCode) {
        case OK:
            throw new Exception("TILT: Should not get here.");
        case UNEXPECTED_ARGUMENT:
            return unexpectedArgumentMessage();
        case MISSING_STRING:
            return String.format("Could not find string parameter for -%c.",
                        errorArgumentId);
        case INVALID_INTEGER:
```

```
            return String.format("Argument -%c expects an integer but was '%s'.",
                    errorArgumentId, errorParameter);
        case MISSING_INTEGER:
            return String.format("Could not find integer parameter for -%c.",
                    errorArgumentId);
        case INVALID_DOUBLE:
            return String.format("Argument -%c expects a double but was '%s'.",
                    errorArgumentId, errorParameter);
        case MISSING_DOUBLE:
            return String.format("Could not find double parameter for -%c",
                    errorArgumentId);
    }
    return "";
}
```

Y las pruebas son satisfactorias. La siguiente prueba garantiza que se detecte correctamente la ausencia de un argumento `double`.

```
public void testMissingDouble() throws Exception {
    Args args = new Args("x##", new String[]{"-x"});
    assertFalse (args.isValid());
    assertEquals(0, args.cardinality());
    assertFalse(args.has('x'));
    assertEquals(0.0, args.getDouble('x'), 0.01);
    assertEquals("Could not find double parameter for -x.",
                 args.errorMessage());
}
```

Es correcto. La incluimos para que el ejemplo resulte más completo.

El código de excepciones no es atractivo y no pertenece realmente a la clase `Args`. También generamos `ParseException`, que no nos pertenece. Por ello, combinamos todas las excepciones en una única clase `ArgsException` y la incluimos en su propio módulo.

```
public class ArgsException extends Exception {
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ErrorCode errorCode = ErrorCode.OK;


    public ArgsException() {}


    public ArgsException(String message) { super(message); }


    public enum ErrorCode {
        OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,
        MISSING_DOUBLE, INVALID_DOUBLE }
}
…
public class Args {
    …
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ArgsException.ErrorCode errorCode = ArgsException.ErrorCode.OK;
    private List<String> argsList;


    public Args(String schema, String[] args) throws ArgsException {
        this.schema = schema;
        argsList = Arrays.asList(args);
        valid = parse();
    }


    private boolean parse() throws ArgsException {
        if (schema.length() == 0 && argsList.size() == 0)
            return true;
        parseSchema();
        try {
            parseArguments();
        } catch (ArgsException e) {
        }
        return valid;
    }


    private boolean parseSchema() throws ArgsException {
        …
    }


    private void parseSchemaElement(String element) throws ArgsException {
        …
        else
            throw new ArgsException(
```

```
            String.format("Argument: %c has invalid format: %s.",
                    elementId, elementTail));
    }

    private void validateSchemaElementId(char elementId) throws ArgsException {
        if (!Character.isLetter(elementId)) {
            throw new ArgsException(
                "Bad character:" + elementId + "in Args format: " + schema);
        }
    }

    …

    private void parseElement(char argChar) throws ArgsException {
        if (setArgument(argChar))
        argsFound.add(argChar);
        else {
            unexpectedArguments.add(argChar);
            errorCode = ArgsException.ErrorCode.UNEXPECTED_ARGUMENT;
            valid = false;
        }
    }

…

    private class StringArgumentMarshaler implements ArgumentMarshaler {
        private String stringValue = "";

        public void set(Iterator<String> currentArgument) throws ArgsException {
            try {
                stringValue = currentArgument.next();
            } catch (NoSuchElementException e) {
                errorCode = ArgsException.ErrorCode.MISSING_STRING;
                throw new ArgsException();
            }
        }

        public Object get() {
            return stringValue;
        }
    }

    private class IntegerArgumentMarshaler implements ArgumentMarshaler {
        private int intValue = 0;

        public void set (Iterator<String> currentArgument) throws ArgsException {
            String parameter = null;
            try {
                parameter = currentArgument.next();
                intValue = Integer.parseInt(parameter);
            } catch (NoSuchElementException e) {
                errorCode = ArgsException.ErrorCode.MISSING_INTEGER;
                throw new ArgsException();
            } catch (NumberFormatException e) {
                errorParameter = parameter;
                errorCode = ArgsException.ErrorCode.INVALID_INTEGER;
                throw new ArgsException();
            }
        }

        public Object get() {
            return intValue;
        }
    }

    private class DoubleArgumentMarshaler implements ArgumentMarshaler {
        private double doubleValue = 0;

        public void set(Iterator<String> currentArgument) throws ArgsException {
            String parameter = null;
            try {
                parameter = currentArgument.next();
                doubleValue = Double.parseDouble(parameter);
            } catch (NoSuchElementException e) {
                errorCode = ArgsException.ErrorCode.MISSING_DOUBLE;
                throw new ArgsException();
            } catch (NumberFormatException e) {
                errorParameter = parameter;
                errorCode = ArgsException.ErrorCode.INVALID_DOUBLE;
                throw new ArgsException);
            }
        }

        public Object get() {
            return doubleValue;
```

```
        }
    }
}
```

Muy bien. Ahora, `Args` solamente genera `ArgsException`. Al desplazar `ArgsException` a un módulo propio, podemos añadir a dicho módulo gran parte del código de error y extraerlo del módulo `Args`. Es una posición natural y evidente para incluir todo el código y nos permitirá limpiar posteriormente el módulo `Args`.

Ya hemos separado el código de excepciones y de error del módulo `Args` (véanse los listados del 14-13 al 14-16). Para ello realizamos una serie de 30 pasos mínimos y las pruebas fueron satisfactorias entre todos ellos.

Listado 14-13
ArgsTest.java.

```
package com.objectmentor.utilities.args;

import junit.framework.TestCase;

public class ArgsTest extends TestCase {
    public void testCreateWithNoSchemaOrArguments() throws Exception {
        Args args = new Args("", new String[0]);
        assertEquals(0, args.cardinality());
    }

    public void testWithNoSchemaButWithOneArgument() throws Exception {
        try {
            new Args("", new String[]{"-x"});
            fail();
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
            e.getErrorCode());
            assertEquals('x', e.getErrorArgumentId());
        }
    }

    public void testWithNoSchemaButWithMultipleArguments() throws Exception {
        try {
            new Args("", new String[]{"-x", "-y"});
            fail();
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
            e.getErrorCode());
            assertEquals('x', e.getErrorArgumentId());
        }

    }

    public void testNonLetterSchema() throws Exception {
        try {
            new Args("*", new String[]{});
            fail("Args constructor should have thrown exception");
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
                         e.getErrorCode());
            assertEquals('*', e.getErrorArgumentId());
        }
    }

    public void testInvalidArgumentFormat() throws Exception {
        try {
            new Args("f~", new String[]{});
            fail("Args constructor should have throws exception");
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.INVALID_FORMAT, e.getErrorCode());
            assertEquals('f', e.getErrorArgumentId());
        }
    }

    public void testSimpleBooleanPresent() throws Exception {
```

```
        Args args = new Args(“x”, new String []{“-x”});
        assertEquals(1, args.cardinality());
        assertEquals(true, args.getBoolean(‘x’));
    }

    public void testSimpleStringPresent() throws Exception {
        Args args = new Args(“x*”, new String[]{“-x”, “param”});
        assertEquals(1, args.cardinality());
        assertTrue(args.has(‘x’));
        assertEquals(“param”, args.getString(‘x’));
    }

    public void testMissingStringArgument() throws Exception {
        try {
            new Args(“x*”, new String[]{"-x"});
            fail();
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.MISSING_STRING, e.getErrorCode());
            assertEquals(‘x’, e.getErrorArgumentId());
        }
    }

    public void testSpacesInFormat() throws Exception {
        Args args = new Args(“x, y”, new String[]{“-xy”});
        assertEquals(2, args.cardinality());
        assertTrue(args.has(‘x’));
        assertTrue(args.has(‘y’));
    }

    public void testSimpleIntPresent() throws Exception {
        Args args = new Args(“x#”, new String[]{“-x”, “42”});
        assertEquals(1, args.cardinality());
        assertTrue(args.has(‘x’));
        assertEquals(42, args.getInt(‘x’));
    }

    public void testInvalidInteger() throws Exception {
        try {
            new Args(“x#”, new String[] {“-x”, “Forty two”});
            fail();
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.INVALID_INTEGER, e.getErrorCode());
            assertEquals(‘x’, e.getErrorArgumentId());
            assertEquals(“Forty two”, e.getErrorParameter());
        }
    }

    public void testMissingInteger() throws Exception {
        try {
            new Args(“x#”, new String[]{“-x”});
            fail();
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.MISSING_INTEGER, e.getErrorCode());
            assertEquals(‘x’, e.getErrorArgumentId());
        }
    }

    public void testSimpleDoublePresent() throws Exception {
        Args args = new Args(“x##”, new String[](“-x”, “42.3”});
        assertEquals(1, args.cardinality());
        assertTrue(args.has(‘x’));
        assertEquals(42.3, args.getDouble(‘x’), .001);
    }

    public void testInvalidDouble() throws Exception {
        try {
            new Args(“x##”, new String []{“-x”, “Forty two”});
            fail();
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.INVALID_DOUBLE, e.getErrorCode());
            assertEquals(‘x’, e.getErrorArgumentId());
            assertEquals(“Forty two”, e.getErrorParameter());
        }
    }

    public void testMissingDouble() throws Exception {
        try {
            new Args(“x##”, new String[]{“-x”});
            fail();
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.MISSING_DOUBLE, e.getErrorCode());
            assertEquals(‘x’, e.getErrorArgumentId());
        }
    }
}
```

Listado 14-14
ArgsExceptionTest.java.

```
public class ArgsExceptionTest extends TestCase {
    public void testUnexpectedMessage() throws Exception {
        ArgsException e =
            new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                              'x', null);
        assertEquals("Argument -x unexpected.", e.errorMessage());
    }


    public void testMissingStringMessage() throws Exception {
        ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_STRING,
                          'x', null);
        assertEquals("Could not find string parameter for –x.", e.errorMessage());
    }


    public void testInvalidIntegerMessage() throws Exception {
        ArgsException e =
            new ArgsException(ArgsException.ErrorCode.INVALID_INTEGER,
                          'x', "Forty two");
        assertEquals("Argument –x expects an integer but was 'Forty two'.",
                          e.errorMessage());
    }


    public void testMissingIntegerMessage() throws Exception {
        ArgsException e =
            new ArgsException(ArgsException.ErrorCode.MISSING_INTEGER, 'x', null);
        assertEquals("Could not find integer parameter for -x.", e.errorMessage());
    }


    public void testInvalidDoubleMessage() throws Exception {
        ArgsException e = new ArgsException(ArgsException.ErrorCode.INVALID_DOUBLE,
                          'x', "Forty two");
        assertEquals("Argument -x expects a double but was 'Forty two'.",
                          e.errorMessage());
    }


    public void testMissingDoubleMessage() throws Exception {
        ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_DOUBLE,
                          'x', null);
        assertEquals("Could not find double parameter for -x.", e.errorMessage());
    }
}
```

Listado 14-15
ArgsException.java.

```
public class ArgsException extends Exception {
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ErrorCode errorCode = ErrorCode.OK;


    public ArgsException() {}


    public ArgsException(String message) {super(message);}


    public ArgsException(ErrorCode errorCode) {
        this.errorCode = errorCode;
    }


    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() throws Exception {
        switch (errorCode) {
            case OK:
                throw new Exception("TILT: Should not get here.");
            case UNEXPECTED_ARGUMENT:
                return String.format("Argument -%c unexpected.", errorArgumentId);
            case MISSING_STRING:
                return String.format("Could not find string parameter for -%c.",
                        errorArgumentId);
            case INVALID_INTEGER:
                return String.format("Argument -%c expects an integer but was '%s'.",
                        errorArgumentId, errorParameter);
            case MISSING_INTEGER:
                return String.format("Could not find integer parameter for -%c.",
                        errorArgumentId);
            case INVALID_DOUBLE:
                return String.format("Argument -%c expects a double but was '%s'.",
                        errorArgumentId, errorParameter);
            case MISSING_DOUBLE:
                return String.format("Could not find double parameter for -%c.",
                        errorArgumentId);
        }
        return "";
    }

    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 {
    private String schema;
    private Map<Character, ArgumentMarshaler> marshalers =
        new HashMap<Character, ArgumentMarshaler>();
    private Set<Character> argsFound = new HashSet<Character>();
    private Iterator<String> currentArgument;
    private List<String> argsList;

    public Args(String schema, String[] args) throws ArgsException {
        this.schema = schema;
        argsList = Arrays.asList(args);
        parse();
    }

    private void parse() throws ArgsException {
        parseSchema();
        parseArguments();
    }

    private boolean parseSchema() throws ArgsException {
        for (String element : schema.split(",")) {
            if (element.length() > 0) {
                parseSchemaElement(element.trim());
            }
        }
        return true;
    }
```

```
private void parseSchemaElement(String element) throws ArgsException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (elementTail.length() == 0)
        marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals(“*”))
        marshalers.put(elementId, new StringArgumentMarshaler());
    else if (elementTail.equals(“#”))
        marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals(“##”))
        marshalers.put(elementId, new DoubleArgumentMarshaler());
    else
        throw new ArgsException(ArgsException.ErrorCode.INVALID_FORMAT,
                    elementId, elementTail);
}


private void validateSchemaElementId(char elementId) throws ArgsException {
    if (!Character.isLetter(elementId)) {
        throw new ArgsException(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
                    elementId, null);
    }
}


private void parseArguments() throws ArgsException {
    for (currentArgument = argsList.iterator(); currentArgument.hasNext();;) {
        String arg = currentArgument.next();
        parseArgument(arg);
    }
}


private void parseArgument(String arg) throws ArgsException {
    if (arg.startsWith(“-”))
        parseElements(arg);
}


private void parseElements(String arg) throws ArgsException {
    for (int i = 1; i < arg.length(); i++)
        parseElement(arg.charAt(i));
}


private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
        argsFound.add(argChar);
    else {
        throw new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                    argChar, null);
    }
}


private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;
    try {
        m.set(currentArgument);
        return true;
    } catch (ArgsException e) {
        e.setErrorArgumentId(argChar);
        throw e;
    }
}


public int cardinality() {
    return argsFound.size();
}


public String usage() {
    if (schema.length() > 0)
        return "-[" + schema + “]”;
    else
        return ““;
}


public boolean getBoolean(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    boolean b = false;
    try {
        b = am != null && (Boolean) am.get();
    } catch (ClassCastException e) {
        b = false;
    }
    return b;
}


public String getString(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    try {
```

```
            return am == null ? "" : (String) am.get();
        } catch (ClassCastException e) {
            return "";
        }
    }

    public int getInt(char arg) {
        ArgumentMarshaler am = marshalers.get(arg);
        try {
            return am == null ? 0 : (Integer) am.get();
        } catch (Exception e) {
            return 0;
        }
    }

    public double getDouble(char arg) {
        ArgumentMarshaler am = marshalers.get(arg);
        try {
            return am == null ? 0 : (Double) am.get();
        } catch (Exception e) {
            return 0.0;
        }
    }

    public boolean has(char arg) {
        return argsFound.contains(arg);
    }

}
```

La mayoría de los cambios realizados en la clase `Args` han sido eliminaciones. Gran parte del código se extrajo de `Args` y se añadió a `ArgsException`. Perfecto. También cambiamos todos los elementos `ArgumentMarshaller` a sus propios archivos. Mejor todavía.

El diseño de *software* correcto se basa gran parte en las particiones, en crear zonas adecuadas para incluir distintos tipos de código. Esta separación hace que el código sea más fácil de entender y mantener.

Especialmente interesante es el método `errorMessage` de `ArgsException`. Incumple claramente el SRP al incluir el formato de mensajes de error en `Args`. `Args` debe centrarse en el procesamiento de argumentos, no en el formato de los mensajes de error. Sin embargo, ¿realmente tiene sentido incluir el código de formato de mensajes de error en `ArgsException`?

Francamente es un compromiso. Los usuarios que no deseen los mensajes de error proporcionados por `ArgsException` tendrán que crear los suyos propios, pero la utilidad de mensajes de error ya preparados es evidente.

Ya debería haberse dado cuenta de la distancia recorrida con respecto a la solución mostrada al inicio del capítulo. Las transformaciones finales puede examinarlas por su cuenta.

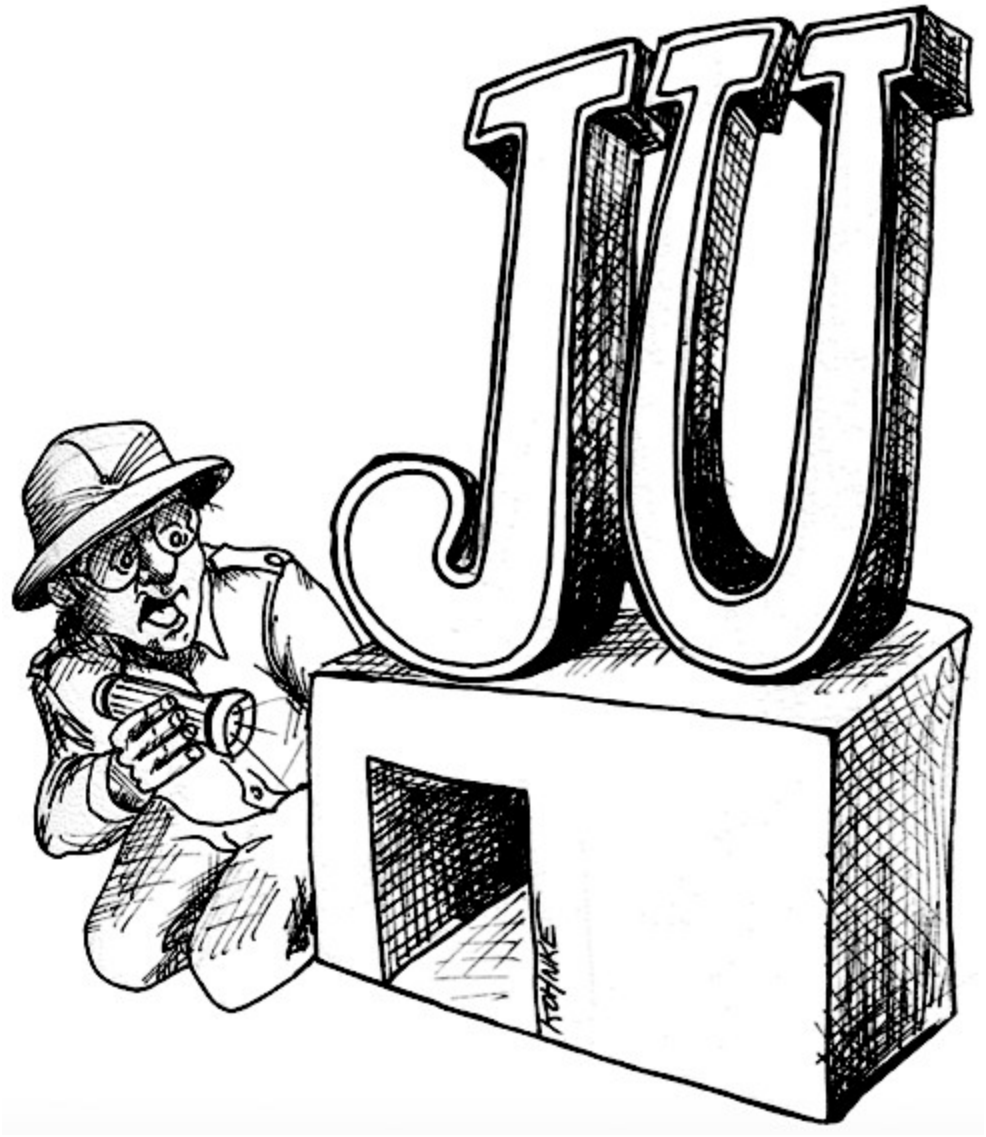
Conclusión

No basta con que el código funcione. El código que funciona suele ser incorrecto. Los programadores que se conforman con código funcional no se comportan de forma profesional. Puede que teman que no tienen tiempo para mejorar la estructura y el diseño del código, pero discrepo. No hay nada que afecte más negativamente a un proyecto de desarrollo que el código incorrecto. Los plazos incorrectos se pueden rehacer y los requisitos equivocados se pueden volver a definir. La dinámica incorrecta de un equipo se puede reparar pero el código incorrecto se corrompe y se convierte en una carga que arrastra al equipo completo. He visto equipos dominados por el desastre que han generado y que han dominado su destino.

Evidentemente, el código incorrecto se puede limpiar pero resulta muy costoso. Cuando el código se corrompe los módulos se insinúan unos a otros y generan multitud de dependencias ocultas y entrelazadas. La localización y división de dependencias antiguas es una tarea larga y complicada. Por otra parte, resulta relativamente sencillo mantener código limpio. Si comete un error en un módulo, es más fácil limpiarlo directamente. Mejor todavía, si cometió un error hace cinco minutos, es muy fácil limpiarlo ahora.

Por tanto, la solución consiste en mantener el código limpio y sencillo siempre que se pueda y no dejar que llegue a corromperse.

15
Aspectos internos de JUnit



JUnit es una de las estructuras de Java más conocidas. De concepción sencilla, definición precisa y documentación elegante. ¿Y su código? En este capítulo analizaremos un ejemplo extraído de la estructura JUnit.

La estructura JUnit

JUnit ha tenido muchos autores, comenzando por Kent Beck y Eric Gamma en un vuelo a Atlanta. Kent quería aprender Java y Eric quería saber más sobre la estructura de pruebas Smalltalk de Kent. "¿Hay algo más natural que dos fanáticos enciendan sus portátiles y empiecen a escribir código?"[94] Tras tres horas de trabajo de altura, habían creado los fundamentos de JUnit.

El módulo que analizaremos es un inteligente fragmento de código que permite identificar errores de comparación de cadenas. El nombre del módulo es `ComparisonCompactor`. Dadas dos cadenas diferentes, como `ABCDE` y `ABXDE`, muestra la diferencia entre ambas generando una cadena como `<…B[X]D…>`.

Podríamos explicarlo más, pero los casos de prueba son mejores. Fíjese en el Listado 15-1 para comprender los requisitos de este módulo. Analice la estructura de las pruebas. ¿Podrían ser más simples o más evidentes?

Listado 15-1
ComparisonCompactorTest.java.

```
package junit.tests.framework;

import junit.framework.ComparisonCompactor;
import junit.framework.TestCase;

public class ComparisonCompactorTest extends TestCase {

    public void testMessage() {
        String failure= new ComparisonCompactor(0, "b", "c").compact("a");
        assertTrue("a expected:<[b]> but was:<[c]>".equals(failure));
    }

    public void testStartSame() {
        String failure= new ComparisonCompactor(1, "ba", "bc").compact(null);
        assertEquals("expected:<b[a]> but was:<b[c]>", failure);
    }

    public void testEndSame() {
        String failure= new ComparisonCompactor(1, "ab", "cb").compact(null);
        assertEquals("expected:<[a]b> but was:<[c]b>", failure);
    }

    public void testSame() {
        String failure= new ComparisonCompactor(1, "ab", "ab").compact(null);
```

```
        assertEquals("expected:<ab> but was:<ab>", failure);
    }


    public void testNoContextStartAndEndSame() {
        String failure= new ComparisonCompactor(0, "abc", "adc").compact(null);
        assertEquals("expected:<…[b]…> but was:<…[d]…>", failure);
    }


    public void testStartAndEndContext() {
        String failure= new ComparisonCompactor(1, "abc", "adc").compact(null);
        assertEquals("expected:<a[b]c> but was:<a[d]c>", failure);
    }


    public void testStartAndEndContextWithEllipses() {
        String failure=
            new ComparisonCompactor(1, "abcde", "abfde").compact(null);
        assertEquals("expected:<…b[c]d…> but was:<…b[f]d…>", failure);
    }


    public void testComparisonErrorStartSameComplete() {
        String failure= new ComparisonCompactor(2, "ab", "abc").compact(null);
        assertEquals("expected:<ab[]> but was:<ab[c]>", failure);
    }


    public void testComparisonErrorEndSameComplete() {
        String failure= new ComparisonCompactor(0, "bc", "abc").compact(null);
        assertEquals("expected:<[]…> but was:<[a]…>", failure);
    }


    public void testComparisonErrorEndSameCompleteContext() {
        String failure= new ComparisonCompactor(2, "bc", "abc").compact(null);
        assertEquals("expected:<[]bc> but was:<[a]bc>", failure);
    }


    public void testComparisonErrorOverlapingMatches() {
        String failure= new ComparisonCompactor(0, "abc", "abbc").compact(null);
        assertEquals("expected:<…[]…> but was:<…[b]…>", failure);
    }


    public void testComparisonErrorOverlapingMatchesContext() {
        String failure= new ComparisonCompactor(2, "abc", "abbc").compact(null);
        assertEquals("expected:<ab[]c> but was:<ab[b]c>", failure);
    }
    public void testComparisonErrorOverlapingMatches2() {
        String failure= new ComparisonCompactor(0, "abcdde",
            "abcde").compact(null);
        assertEquals("expected:<…[d]…> but was:<…[]…>", failure);
    }


    public void testComparisonErrorOverlapingMatches2Context() {
        String failure=
            new ComparisonCompactor(2, "abcdde", "abcde").compact(null);
        assertEquals("expected:<…cd[d]e> but was:<…cd[]e>", failure);
    }


    public void testComparisonErrorWithActualNull() {
        String failure= new ComparisonCompactor(0, "a", null).compact(null);
        assertEquals("expected:<a> but was:<null>", failure);
    }


    public void testComparisonErrorWithActualNullContext() {
        String failure= new ComparisonCompactor(2, "a", null).compact(null);
        assertEquals("expected:<a> but was:<null>", failure);
    }


    public void testComparisonErrorWithExpectedNull() {
        String failure= new ComparisonCompactor(0, null, "a").compact(null);
        assertEquals("expected:<null> but was:<a>", failure);
    }


    public void testComparisonErrorWithExpectedNullContext() {
        String failure= new ComparisonCompactor(2, null, "a").compact(null);
        assertEquals("expected:<null> but was:<a>", failure);
    }


    public void testBug609972() {
        String failure= new ComparisonCompactor(10, "S&P500", "0").compact(null);
        assertEquals("expected:<[S&P50]0> but was:<[]0>", failure);
    }
}
```

Realicé un análisis de alcance de código en `ComparisonCompactor` con estas pruebas. El código se cubre en un 100 por 100. Cada línea, cada instrucción `if` y cada bucle `for` se ejecuta con las pruebas. De este modo sé que el código funciona y sus autores me merecen el mayor de los respetos.

El código `ComparisonCompactor` se reproduce en el Listado 15-2. Examínelo. Creo que lo encontrará bien distribuido, razonablemente expresivo y estructuralmente sencillo. Cuando termine, lo diseccionaremos.

Listado 15-2
ComparisonCompactor.java (Original).

```
package junit.framework;

public class ComparisonCompactor {

    private static final String ELLIPSIS = "…";
    private static final String DELTA_END = "]";
    private static final String DELTA_START = "[";

    private int fContextLength;
    private String fExpected;
    private String fActual;
    private int fPrefix;
    private int fSuffix;

    public ComparisonCompactor(int contextLength,
                        String expected,
                        String actual) {
        fContextLength = contextLength;
        fExpected = expected;
        fActual = actual;
    }

    public String compact(String message) {
        if (fExpected == null || fActual == null || areStringsEqual())
            return Assert.format(message, fExpected, fActual);

        findCommonPrefix();
        findCommonSuffix();
        String expected = compactString(fExpected);
        String actual = compactString(fActual);
        return Assert.format(message, expected, actual);
    }

    private String compactString(String source) {
        String result = DELTA_START +
                        source.substring(fPrefix, source.length() -
                        fSuffix + 1) + DELTA_END;
        if (fPrefix > 0)
            result = computeCommonPrefix() + result;
        if (fSuffix > 0)
            result = result + computeCommonSuffix();
        return result;
    }

    private void findCommonPrefix() {
        fPrefix = 0;
        int end = Math.min(fExpected.length(), fActual.length());
        for (; fPrefix < end; fPrefix++) {
            if (fExpected.charAt(fPrefix) != fActual.charAt(fPrefix))
                break;
        }
    }

    private void findCommonSuffix() {
        int expectedSuffix = fExpected.length() - 1;
        int actualSuffix = fActual.length() - 1;
        for (;
                actualSuffix >= fPrefix && expectedSuffix >= fPrefix;
                actualSuffix--, expectedSuffix--) {
            if (fExpected.charAt(expectedSuffix) != fActual.charAt(actualSuffix))
```

```
                break;
            }
            fSuffix = fExpected.length() - expectedSuffix;
        }

        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,
                            fExpected.length());
            return fExpected.substring(fExpected.length() - fSuffix + 1, end) +
                            (fExpected.length() - fSuffix + 1 < fExpected.length() -
                            fContextLength ? ELLIPSIS : "");
        }

        private boolean areStringsEqual() {
            return fExpected.equals(fActual);
        }
    }
```

Puede que tenga varias quejas sobre el módulo. Incluye expresiones extensas y extraños elementos `+1`. Pero en general, está bastante bien. Después de todo, podría haber sido como el Listado 15-3.

Listado 15-3
ComparisonCompator.java (defactorizado)

```
package junit.framework;

public class ComparisonCompactor {
    private int ctxt;
    private String s1;
    private String s2;
    private int pfx;
    private int sfx;

    public ComparisonCompactor(int ctxt, String s1, String s2) {
        this.ctxt = ctxt;
        this.s1 = s1;
        this.s2 = s2;
    }

    public String compact(String msg) {
        if (s1 == null || s2 == null || s1.equals(s2))
            return Assert.format(msg, s1, s2);

        pfx = 0;
        for (; pfx < Math.min(s1.length(), s2.length()); pfx++) {
            if (s1.charAt(pfx) != s2.charAt(pfx))
                break;
        }
        int sfx1 = s1.length() - 1;
        int sfx2 = s2.length() - 1;
        for (; sfx2 >= pfx && sfx1 >= pfx; sfx2--, sfx1--) {
            if (s1.charAt(sfx1) != s2.charAt(sfx2))
                break;
        }
        sfx = s1.length() - sfx1;
        String cmp1 = compactString(s1);
        String cmp2 = compactString(s2);
        return Assert.format(msg, cmp1, cmp2);
    }

    private String compactString(String s) {
        String result =
            "[" + s.substring(pfx, s.length() - sfx + 1) + "]";
        if (pfx > 0)
            result = (pfx > ctxt ? "..." : "") +
                s1.substring(Math.max(0, pfx - ctxt), pfx) + result;
        if (sfx > 0) {
            int end = Math.min(s1.length() - sfx + 1 + ctxt, s1.length());
            result = result + (s1.substring(s1.length() - sfx + 1, end) +
                (s1.length() - sfx + 1 < s1.length() - ctxt ? "..." : ""));
        }
        return result;
```

```
    }

}
```

Aunque los autores hicieron un buen trabajo con este módulo, la *Regla del Boy Scout*[95] muestra que podrían haberlo dejado más limpio de lo que se encontró. ¿Cómo podemos mejorar el código original del Listado 15-2? Lo primero que no necesitamos es el prefijo `f` de las variables miembro [N6]. Los entornos actuales hacen que este tipo de código de ámbito sea redundante, por lo que eliminaremos todas las `f`.

```
private int contextLength;
private String expected;
private String actual;
private int prefix;
private int suffix;
```

Tras ello, tenemos una condicional sin encapsular al inicio de la función compact [G28].

```
public String compact(String message) {
    if (expected == null || actual == null || areStringsEqual())
        return Assert.format(message, expected, actual);

    findCommonPrefix();
    findCommonSuffix();
    String expected = compactString(this.expected);
    String actual = compactString(this.actual);
    return Assert.format(message, expected, actual);
}
```

Es necesario encapsular esta condicional para que nuestra intención sea más clara. Por tanto, extraemos un método que la explique.

```
public String compact(String message) {
    if (shouldNotCompact())
        return Assert.format(message, expected, actual);

    findCommonPrefix();
    findCommonSuffix();
    String expected = compactString(this.expected);
    String actual = compactString(this.actual);
    return Assert.format(message, expected, actual);
}

private boolean shouldNotCompact() {
    return expected == null || actual == null || areStringsEqual();
}
```

En la función `compact`, `this.expected` y `this.actual` no son demasiado relevantes. Sucede al cambiar el nombre de `fExpected` por `expected`. ¿Por qué esta función tiene variables con los mismos nombres que las variables miembro? ¿No representan cosas diferentes?[N4]. Los nombres deben ser exclusivos.

```
String compactExpected = compactString(expected);
String compactActual = compactString(actual);
```

Los negativos son más difíciles de entender que los positivos [G29]. Por ello, invertimos esa instrucción `if` para cambiar el sentido de la condicional.

```
public String compact(String message) {
    if (canBeCompacted()) {
        findCommonPrefix();
        findCommonSuffix();
```

```
            String compactExpected = compactString(expected);
            String compactActual = compactString(actual);
            return Assert.format(message, compactExpected, compactActual);
        } else {
            return Assert.format(message, expected, actual);
        }
    }

    private boolean canBeCompacted() {
        return expected != null && actual != null && !areStringsEqual();
    }
```

El nombre de la función es extraño [N7]. Aunque compacta las cadenas, puede que lo haga si `canBeCompacted` devuelve `false`. Al asignar el nombre `compact` a esta función se oculta el efecto secundario de la comprobación de errores. Además, la función devuelve un mensaje con formato, no sólo las cadenas compactadas. Por tanto, el nombre de la función debería ser `formatCompactedComparison`. De esta forma, se lee mejor junto al argumento de la función:

```
public String formatCompactedComparison(String message) {
```

El cuerpo de la instrucción `if` es donde se realiza la verdadera compactación de las cadenas. Debemos extraerlo como método con el nombre `compactExpectedAndActual`. Sin embargo, queremos que la función `formatCompactedComparison` realice todo el formato. La función `compact`… sólo debe realizar la compactación [G30], de modo que la dividimos de esta forma:

```
…
    private String compactExpected;
    private String compactActual;
…


    public String formatCompactedComparison(String message) {
        if (canBeCompacted()) {
            compactExpectedAndActual();
            return Assert.format(message, compactExpected, compactActual);
        } else {
            return Assert.format(message, expected, actual);
        }
    }

    private void compactExpectedAndActual() {
        findCommonPrefix();
        findCommonSuffix();
        compactExpected = compactString(expected);
        compactActual = compactString(actual);
    }
```

Para ello, hemos tenido que ascender `compactExpected` y `compactActual` a variables miembro. No me gusta la forma en que las dos últimas líneas de la nueva función devuelven variables pero las dos primeras no lo hacen. No utilizan convenciones coherentes [G11]. Debemos cambiar `findCommonPrefix` y `findCommonSuffix` para que devuelvan los valores de prefijo y sufijo.

```
private void compactExpectedAndActual() {
    prefixIndex = findCommonPrefix();
    suffixIndex = findCommonSuffix();
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
}
```

```
private int findCommonPrefix() {
    int prefixIndex = 0;
    int end = Math.min(expected.length(), actual.length());
    for (; prefixIndex < end; prefixIndex++) {
        if (expected.charAt(prefixIndex) != actual.charAt(prefixIndex))
            break;
    }
    return prefixIndex;
}


private int findCommonSuffix() {
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
            actualSuffix--, expectedSuffix--) {
        if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
        break;
    }
    return expected.length() - expectedSuffix;
}
```

También debemos cambiar los nombres de las variables miembro para que sean más preciosas [N1], ya que en el fondo son índices.

Al examinar `findCommonSuffix` vemos una conexión temporal oculta [G31]; depende de que `prefixIndex` se calcule por `findCommonPrefix`. Si estas dos funcione se invocan de forma desordenada, la sesión de depuración posterior sería complicada. Por ello, para mostrar esta combinación temporal, haremos que `findCommonSuffix` acepte `prefixIndex` como argumento.

```
private void compactExpectedAndActual() {
    prefixIndex = findCommonPrefix();
    suffixIndex = findCommonSuffix(prefixIndex);
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
}


private int findCommonSuffix(int prefixIndex) {
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
            actualSuffix--, expectedSuffix--) {
        if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
        break;
    }
    return expected.length() - expectedSuffix;
}
```

No me convence del todo. El hecho de pasar `prefixIndex` como argumento es un tanto arbitrario [G32]. Permite establecer el orden pero no explica la necesidad del mismo. Otro programador podría deshacer esta operación ya que no se indica en ningún momento para qué sirve el parámetro. Adoptemos un enfoque diferente.

```
private void compactExpectedAndActual() {
    findCommonPrefixAndSuffix();
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
}


private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (;
            actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
            actualSuffix--, expectedSuffix--
            ) {
        if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
            break;
    }
    suffixIndex = expected.length() - expectedSuffix;
}
```

```
private void findCommonPrefix() {
    prefixIndex = 0;
    int end = Math.min(expected.length(), actual.length());
    for (; prefixIndex < end; prefixIndex++)
        if (expected.charAt(prefixIndex) != actual.charAt(prefixIndex))
            break;
}
```

Devolvemos `findCommonPrefix` y `findCommonSuffix` a su posición original, cambiamos el nombre de `findCommonSuffix` por `findCommonPrefixAndSuffix` y hacemos que invoque `findCommonPrefix` antes de hacer nada más. De ese modo se establece la naturaleza temporal de ambas funciones de forma más evidente que antes. Además, se muestra el mínimo atractivo de `findCommonPrefixAndSuffix`, que limpiaremos a continuación:

```
private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    int suffixLength = 1;
    for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {
        if (charFromEnd(expected, suffixLength) !=
                charFromEnd(actual, suffixLength))
            break;
    }
    suffixIndex = suffixLength;
}

private char charFromEnd(String s, int i) {
    return s.charAt(s.length()-i);}

private boolean suffixOverlapsPrefix(int suffixLength) {
    return actual.length() - suffixLength < prefixLength ||
    expected.length() - suffixLength < prefixLength;
}
```

Mucho mejor. Muestra que `suffixIndex` es en realidad la longitud del sufijo y que su nombre no es correcto. Lo mismo sucede con `prefixIndex`, aunque en ese caso índice y longitud son sinónimos. Incluso así, es más coherente usar `length`. El problema es que la variable `suffixIndex` no es de base cero, sino de base `1` y no es una verdadera longitud. Éste es el motivo de la abundancia de `+1` en `computeCommonSuffix` [G33]. Lo corregimos. En el Listado 15-4 puede ver el resultado.

Listado 15-4
ComparisonCompactor.java (versión intermedia).

```
public class ComparisonCompactor {
…
    private int suffixLength;
…
    private void findCommonPrefixAndSuffix() {
        findCommonPrefix();
        suffixLength = 0;
        for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {
            if (charFromEnd(expected, suffixLength) !=
                charFromEnd(actual, suffixLength))
                break;
        }
    }

    private char charFromEnd(String s, int i) {
        return s.charAt(s.length() - i - 1);
    }

    private boolean suffixOverlapsPrefix(int suffixLength) {
```

```
        return actual.length() - suffixLength <= prefixLength ||
            expected.length() - suffixLength <= prefixLength;
    }

…
    private String compactString(String source) {
        String result =
            DELTA_START +
            source.substring(prefixLength, source.length() - suffixLength) +
            DELTA_END;
        if (prefixLength > 0)
            result = computeCommonPrefix() + result;
        if (suffixLength > 0)
            result = result + computeCommonSuffix();
        return result;
    }

…
    private String computeCommonSuffix() {
        int end = Math.min(expected.length() - suffixLength +
            contextLength, expected.length()
        );
        return
            expected.substring(expected.length() - suffixLength, end) +
            (expected.length() - suffixLength <
                expected.length() - contextLength ?
                ELLIPSIS : "");
    }
```

Cambiamos `+1` en computeCommonSuffix por un `-1` en charFromEnd, donde tiene sentido, y dos operadores `<=` en `suffixOverlapsPrefix`, totalmente correctos. De este modo podemos cambiar el nombre de `suffixIndex` por `suffixLength`, lo que mejora considerablemente la legibilidad del código.

Pero hay un problema. Al comenzar a eliminar los `+1`, me fijé en la siguiente línea de `compactstring`:

```
if (suffixLength > 0)
```

Búsquela en el Listado 15-4. Como ahora `suffixLength` es una unidad menos que antes, debemos cambiar el operador `>` por `>=`. Pero eso no tiene sentido. Ahora sí. Significa que no tenía sentido antes y que seguramente fuera un error. Bueno, no del todo. Tras un análisis detallado, vemos que ahora la instrucción `if` impide que se añada un sufijo de longitud cero. Antes de realizar el cambio, la instrucción `if` no funcionaba ya que `suffixIndex` nunca podía ser menos de uno.

Esto cuestiona ambas instrucciones `if` en `compactString`. Parece como si se pudieran eliminar. Por ello, las comentamos y ejecutamos las pruebas. Satisfactorias. Reestructuremos `compactString` para eliminar las instrucciones `if` sobrantes y simplificar la función [G9].

```
private String compactString(String source) {
    return
        computeCommonPrefix() +
        DELTA_START +
        source.substring(prefixLength, source.length() - suffixLength) +
        DELTA_END +
        computeCommonSuffix();
}
```

Mucho mejor. Ahora vemos que la función `compactString` simplemente combina los fragmentos. Probablemente lo podríamos limpiar

más, en pequeñas operaciones, pero en lugar de desarrollar el resto de los cambios, mostraremos el resultado final en el Listado 15-5.

Listado 15-5
ComparisonCompactor.java (versión definitiva).

```
package junit.framework;

public class ComparisonCompactor {

    private static final String ELLIPSIS = "…";
    private static final String DELTA_END = "]";
    private static final String DELTA_START = "[";

    private int contextLength;
    private String expected;
    private String actual;
    private int prefixLength;
    private int suffixLength;

    public ComparisonCompactor(
        int contextLength, String expected, String actual
    ) {
        this.contextLength = contextLength;
        this.expected = expected;
        this.actual = actual;
    }

    public String formatCompactedComparison(String message) {
        String compactExpected = expected;
        String compactActual = actual;
        if (shouldBeCompacted()) {
            findCommonPrefixAndSuffix();
            compactExpected = compact(expected);
            compactActual = compact(actual);
        }
        return Assert.format(message, compactExpected, compactActual);
    }

    private boolean shouldBeCompacted() {
        return !shouldNotBeCompacted();
    }

    private boolean shouldNotBeCompacted() {
        return expected == null ||
            actual == null ||
            expected.equals(actual);
    }

    private void findCommonPrefixAndSuffix() {
        findCommonPrefix();
        suffixLength = 0;
        for (; !suffixOverlapsPrefix(); suffixLength++) {
            if (charFromEnd(expected, suffixLength) !=
                charFromEnd(actual, suffixLength)
            )
                break;
        }
    }

    private char charFromEnd(String s, int i) {
        return s.charAt(s.length() - i - 1);
    }

    private boolean suffixOverlapsPrefix() {
        return actual.length() - suffixLength <= prefixLength ||
            expected.length() - suffixLength <= prefixLength;
    }

    private void findCommonPrefix() {
        prefixLength = 0;
        int end = Math.min(expected.length(), actual.length());
        for (; prefixLength < end; prefixLength++)
            if (expected.charAt(prefixLength) != actual.charAt(prefixLength))
                break;
    }
```

```
    private String compact(String s) {
        return new StringBuilder()
            .append(startingEllipsis())
            .append(startingContext())
            .append(DELTA_START)
            .append(delta(s))
            .append(DELTA_END)
            .append(endingContext())
            .append(endingEllipsis())
            .toString();
    }

    private String startingEllipsis() {
        return prefixLength > contextLength ? ELLIPSIS : "";
    }

    private String startingContext() {
        int contextStart = Math.max(0, prefixLength - contextLength);
        int contextEnd = prefixLength;
        return expected.substring(contextStart, contextEnd);
    }

    private String delta(String s) {
        int deltaStart = prefixLength;
        int deltaEnd = s.length() - suffixLength;
        return s.substring(deltaStart, deltaEnd);
    }

    private String endingContext() {
        int contextStart = expected.length() - suffixLength;
        int contextEnd =
            Math.min(contextStart + contextLength, expected.length());
        return expected.substring(contextStart, contextEnd);
    }

    private String endingEllipsis() {
        return (suffixLength > contextLength ? ELLIPSIS : "");
    }
}
```

Bastante atractivo. El módulo se separa en un grupo de funciones de análisis y otro grupo de funciones de síntesis. Se ordenan topológicamente para que la definición de cada función aparezca donde realmente se usa. Primero se muestran las funciones de análisis y después las de síntesis. Si se fija atentamente, verá que he invertido algunas de las decisiones adoptadas inicialmente. Por ejemplo, he añadido algunos métodos extraídos a `formatCompactedComparison` y he modificado el sentido de la expresión `shouldNotBeCompacted`. Es algo habitual. A menudo, un cambio de refactorización lleva a otro que a su vez lleva a deshacer el primero. La refactorización es un proceso iterativo de ensayo y error, e inevitablemente converge en algo que consideramos digno de un profesional.

Conclusión

Hemos cumplido la Regla del Boy Scout. Hemos dejado este módulo más limpio de como lo encontramos. No es que no estuviera limpio originalmente, ya que el trabajo de sus autores es excelente, pero cualquier módulo se puede mejorar y es nuestra responsabilidad dejar el código más limpio de lo que lo encontramos.

16
Refactorización de SerialDate

Si visita `http://www.jfree.org/jcommon/index.php`, encontrará la biblioteca JCommon. En su interior incluye el paquete `org.jfree.date` y, dentro de éste, la clase `SerialDate`. Vamos a analizar esta clase.

El autor de `SerialDate` es David Gilbert. David es un programador experimentado y competente. Como veremos, muestra un elevado grado de profesionalidad y disciplina en su código. En lo que a éste respecta, se puede considerar de calidad. Y voy a despedazarlo.

No es un acto de malicia, ni tampoco me creo mejor que David y con el derecho de juzgar su código. De hecho, si leyera algún código que he creado, seguramente tendría que objetar muchos aspectos del mismo. No es un acto de arrogancia. Lo que voy a hacer no es más que una revisión

profesional, algo con lo que todos deberíamos sentirnos cómodos y algo que deberíamos agradecer si alguien lo hace. A través de las críticas es como podemos aprender, como hacen médicos, pilotos o abogados. Y nosotros, como programadores, también tenemos que aprender a hacerlo.

Otra cosa más sobre David Gilbert: es más que un buen programador. David ha tenido el valor y la buena voluntad de ofrecer este código al público gratuitamente, para que cualquiera pueda usarlo y examinarlo. ¡Bien hecho!

`SerialDate` (véase el Listado B-1) es una clase que representa una fecha en Java. ¿Para qué se necesita una clase que represente una fecha si Java ya cuenta con `java.util.Date` y `java.util.Calendar`, entre otras? El autor creó esta clase como respuesta a un problema que yo también he padecido. El comentario de su Javadoc inicial (línea 67) lo explica. Podríamos cuestionar su intención, pero yo también he sufrido este problema y se agradece una clase sobre fechas en lugar de horas.

Primero, conseguir que funcione

Hay varias pruebas de unidad en la clase `SerialDateTests` (véase el Listado B-2). Todas son satisfactorias. Desafortunadamente, un rápido examen demuestra que no comprueban todos los aspectos [T1]. Por ejemplo, al realizar una búsqueda de usos en el método `MonthCodeToQuarter` (línea 334) se indica que no se usa [F4]. Por lo tanto, las pruebas de unidad no lo comprueban. Por ello, recurrí a Clover para ver el alcance de las pruebas de unidad. Clover indicó que las pruebas sólo ejecutan 91 de las 185 instrucciones ejecutables de `SerialDate` (aproximadamente el 50 por 100) [T2]. El mapa de alcance muestra grandes fragmentos de código sin ejecutar desperdigados por la clase.

Mi objetivo era comprender la clase y refactorizarla, algo que no podía lograr sin una cobertura mayor de las pruebas. Por ello diseñé mi propia *suite* de pruebas de unidad independientes (véase el Listado B-4).

Si se fija en las pruebas, comprobará que muchas están comentadas, ya que no se superaron. Representan un comportamiento que considero debería incluirse en `SerialDate`. Por tanto, al refactorizar `SerialDate`, intentaré que estas pruebas funcionen.

Incluso con algunas de las pruebas comentadas, el informe de Clover indica que ahora ejecutan 170 (el 92 por ciento) de las 185 instrucciones ejecutables. Un gran resultado que creo que puedo mejorar.

Las primeras pruebas comentadas (líneas 23-63) son un tanto pretenciosas. El programa no fue diseñado para superar estas pruebas, pero el comportamiento me parecía evidente [G2]. Desconozco por qué se ha creado el método `testWeekdayCodeToString` pero ya que está ahí, parece obvio que no debe distinguir entre mayúsculas y minúsculas. El diseño de las pruebas fue sencillo [T3] y más todavía que fueran satisfactorias; simplemente cambié las líneas 259 y 263 para usar `equalsIgnoreCase`.

Comenté las pruebas de las líneas 32 y 45 ya que no estaba seguro de si las abreviaturas `tues` y `thurs` se admitían o no. Las pruebas de las líneas 153 y 154 no se superaron, aunque deberían haberlo hecho [G2]. Podemos corregirlas, junto a las pruebas de las líneas 163 a la 213, si realizamos los siguientes cambios en la función `stringToMonthCode`.

```
457  if ((result < 1) || (result > 12)) { result = -1;
458  for (int i = 0; i < monthNames.length; i++) {
459  if (s.equalsIgnoreCase(shortMonthNames[i])) {
460  result = i + 1;
461  break;
462  }
463  if (s.equalsIgnoreCase(monthNames[i])) {
464  result = i + 1;
465  break;
466  }
467  }
468  }
```

La prueba comentada de la línea 318 descubre un error en el método `getFollowingDayOfWeek` (línea 672). El 25 de diciembre de 2004 fue sábado y el siguiente sábado fue el 1 de enero de 2005. Sin embargo, al ejecutar la prueba, vemos que `getFollowingDayOfWeek` devuelve el 25 de diciembre como siguiente sábado después del 25 de diciembre, un error evidente [G3], [T1]. Vemos el problema en la línea 685. Es un error de condición de límite típico [T5]. Debería ser lo siguiente:

```
685 if (baseDOW >= targetWeekday) {
```

Conviene destacar que esta función sufrió una reparación anterior. El historial de cambios (línea 43) muestra que se corrigieron los errores en `getPreviousDayOfWeek`, `getFollowingDayOfWeek` y `getNearestDayOfWeek` [T6].

La prueba de unidad `testGetNearestDayOfWeek` (línea 329), que prueba el método `getNearestDayOfWeek` (línea 705), inicialmente no era tan extensa y completa. Añadí multitud de casos de prueba ya que los

iniciales no se superaban [T6]. Puede ver el patrón de fallos si se fija en los casos de prueba comentados. El patrón es revelador [T7]. Muestra que el algoritmo falla si el día más próximo es de una fecha futura. Evidentemente se trata de algún tipo de error de condición de límite [T5].

El patrón de alcance de las pruebas generado por Clover también es interesante [T8]. La línea 719 nunca se ejecuta, lo que significa que la instrucción `if` de la línea 718 siempre es `false`, pero si nos fijamos en el código, indica que debe ser `true`. La variable `adjust` siempre es negativa y no puede ser mayor o igual a 4, por lo que el algoritmo es incorrecto.

A continuación se muestra el algoritmo correcto:

```
int delta = targetDOW - base.getDayOfWeek();
int positiveDelta = delta + 7;
int adjust = positiveDelta % 7;
if (adjust > 3)
    adjust -= 7;


return SerialDate.addDays (adjust, base);
```

Por último, las pruebas de la líneas 417 y 429 se pueden superar si se genera `IllegalArgumentException` en lugar de devolver una cadena de error desde `weekInMonthToString` y `relativeToString`. Con estos cambios, todas las pruebas de unidad se superan y creo que ahora `SerialDate` funciona. Llega el momento de hacer que sea correcta.

Hacer que sea correcta

Describiremos `SerialDate` de arriba a abajo para mejorarla en nuestro recorrido. Aunque no lo veamos en este análisis, ejecutaré todas las pruebas de unidad de `JCommon`, incluida mi prueba de unidad mejorada para `SerialDate`, con todos los cambios efectuados. Por ello, tenga la seguridad de que todos los cambios que vea funcionan para `JCommon`.

En la línea 1 vemos abundantes comentarios sobre información de licencia, derechos de autor, autores e historial de cambios. Asumo que hay ciertos aspectos legales que mostrar, por lo que los derechos de autor y las licencias deben conservarse. Por otra parte, el historial de cambios es una rémora de la década de 1960. Ahora tenemos herramientas de control de código fuente que se encargan de ello. Hay que eliminar este historial [C1].

La lista de importación que comienza en la línea 61 se puede reducir por medio de `java.text.*` y `java.util.*`. [J1]

No me convence el formato HTML del Javadoc (línea 67). Un archivo fuente con más de un lenguaje me parece un problema. Este comentario tiene *cuatro* lenguajes: Java, español, Javadoc y html [G1]. Con tantos lenguajes se hace difícil mantener la coherencia. Por ejemplo, la ubicación de las líneas 71 y 72 se pierde al generar el Javadoc y además, ¿quién quiere ver `<ul>` y `<li>` en el código fuente? Una estrategia más acertada consiste en rodear el comentario con `<pre>` para que el formato del código fuente se conserve en el Javadoc[96].

La línea 86 es la declaración de la clase. ¿Por qué se le asigna el nombre `SerialDate`? ¿Qué sentido tiene la palabra `serial`? ¿Es porque la clase se deriva de `Serializable`? Parece improbable.

Basta de adivinanzas. Sé por qué (o al menos eso creo) se usa la palabra `serial`. La clave se encuentra en las constantes `SERIAL_LOWER_BOUND` y `SERIAL_UPPER_BOUND` de las líneas 98 y 101. Y una clave todavía mejor es el comentario de la línea 830. El nombre de la clase es `SerialDate` ya que se implementa con un número de serie, que parece ser el número de días desde el 30 de diciembre de 1899.

Pero esto supone un problema. Por un lado, el término «número de serie» no es realmente correcto. Puede ser un detalle menor pero la representación es más un desplazamiento relativo que un número de serie. El término «número de serie» tiene que ver más con marcadores de identificación de productos que con fechas. Por ello, no lo considero especialmente descriptivo [N1]. Un término más descriptivo sería «ordinal».

El segundo problema es más significativo. El nombre `SerialDate` implica una implementación. Esta clase es abstracta. No es necesario que implique nada sobre la implementación; de hecho, es aconsejable ocultarla. Por ello, creo que el nombre se encuentra en un nivel de abstracción incorrecto [N2]. En mi opinión, el nombre de esta clase debería ser simplemente `Date`.

Desafortunadamente, hay demasiadas clases con el nombre `Date` en la biblioteca de Java, de modo que no es el más adecuado. Como esta clase trabaja con días y no horas, podríamos usar `Day`, pero ya se usa en otros muchos puntos. Al final, opté por `DayDate` como mejor opción.

A partir de ahora, usaremos `DayDate`. Recuerde que los listados que va a leer siguen usando `SerialDate`.

Entiendo porque `DayDate` se hereda de `Comparable` y `Serializable`. ¿Pero de `MonthConstants`? La clase `MonthConstants` (véase el Listado B-3) es una serie de constantes finales estáticas que definen los meses. Heredar de clases con constantes es un viejo truco que los programadores de Java usan para evitar expresiones como `MonthConstants.January`, pero es una mala idea [J2]. `MonthConstants` debería ser una enumeración.

```
public abstract class DayDate implements Comparable,
                        Serializable {
    public static enum Month {
        JANUARY(1),
        FEBRUARY(2),
        MARCH(3),
        APRIL(4),
        MAY(5),
        JUNE(6),
        JULY(7),
        AUGUST(8),
        SEPTEMBER(9),
        OCTOBER(10),
        NOVEMBER(11),
        DECEMBER(12);


        Month(int index) {
            this.index = index;
        }


        public static Month make(int monthIndex) {
            for (Month m : Month.values()) {
                if (m.index == monthIndex)
                    return m;
            }
            throw new IllegalArgumentException(“Invalid month index ” + monthIndex);
        }
        public final int index;
    }
```

Al cambiar `MonthConstants` por esta enumeración se modifica la clase `DayDate` y todos sus usuarios. Tardé una hora en realizar todos los cambios. Sin embargo, las funciones que antes aceptaban un valor `int` para el mes, ahora aceptan un enumerador `Month`. Esto significa que podemos deshacernos del método `isValidMonthCode` (línea 326) y de la comprobación de errores del código de los meses como en `monthCodeToQuarter` (línea 356) [G5]. Tras ello, en la línea 91, tenemos `serialVersionUID`. Esta variable se usa para controlar el señalizador. Si la cambiamos, con lo que todos los elementos `DayDate` escritos con una versión antigua del *software* serán ilegibles y se generará `InvalidClassException`. Si no declara la variable `serialVersionUID`, el compilador genera una automáticamente y será diferente cada vez que modifique el módulo. Ya sé que todos los documentos recomiendan el control manual de esta variable, pero creo que el control automático de la señalización es más seguro [G4]. Después de todo, prefiero depurar una `InvalidClassException` que el extraño comportamiento que se produciría si me olvido de cambiar `serialVersionUID`. Por ello, eliminaré la variable, al menos por ahora[97].

Creo que el comentario de la línea 93 es redundante. Los comentarios redundantes sólo sirven para acumular mentiras y desinformación [C2]. Por ello los eliminaré.

Los comentarios de las líneas 97 y 100 hablan sobre números de serie, que ya hemos mencionado antes [C1]. Las variables que describen son la primera y última fecha posible que DayDate puede describir. Podríamos hacer que fuera más claro [N1].

```
public static final int EARLIEST_DATE_ORDINAL = 2; // 1/1/1900
public static final int LATEST_DATE_ORDINAL = 2958465; // 12/31/9999
```

Desconozco por qué `EARLIEST_DATE_ORDINAL` es 2 en lugar de 0. El comentario de la línea 829 sugiere que tiene que ver con la forma de representar fechas en Microsoft Excel. Hay información mucho más completa en una variante de `DayDate`: `SpreadsheetDate` (véase el Listado B-5). El comentario de la línea 71 describe este problema.

El problema parece relacionado con la implementación de `SpreadsheetDate` y no con `DayDate`. Mi conclusión es que `EARLIEST_DATE_ORDINAL` y `LATEST_DATE_ORDINAL` no pertenecen a `DayDate` y deberían cambiarse a `SpreadsheetDate` [G6].

De hecho, una búsqueda en el código demuestra que estas variables sólo se usan en `SpreadsheetDate`. Ni en `DayDate`, ni en otras clases de la estructura JCommon. Por lo tanto, las cambio por `SpreadsheetDate`.

Las siguientes variables, `MINIMUM_YEAR_SUPPORTED` y `MAXIMUM_YEAR_SUPPORTED` (líneas 104 y 107), constituyen un dilema. Parece evidente que si `DayDate` es una clase abstracta que no dice nada sobre implementación, no debería informarnos de un año mínimo o máximo. De nuevo, siento la necesidad de cambiar las variables a `SpreadsheetDate` [G6]. Pero una búsqueda rápida de los usuarios de estas variables muestra que otra clase las utiliza: `RelativeDayOfWeekRule` (véase el Listado B-6), Vemos dicho uso en las líneas 177 y 178, en la función `getDate`, donde se usan para comprobar que el argumento de `getDate` sea un año válido. El dilema es que un usuario de una clase abstracta necesita información sobre su implementación.

Tendremos que proporcionar esta información sin contaminar `DayDate`. Por lo general, obtendríamos la información de implementación de una instancia de una variante. Sin embargo, la función `getDate` no recibe una instancia de `DayDate`, aunque sí la devuelve, lo que significa que debe crearla en alguna parte. La solución está en las líneas 187-205. La instancia `DayDate` se crea por medio de una de estas tres funciones:

`getPreviousDayOfWeek`, `getNearestDayOfWeek` o `getFollowingDayOfWeek`. Si nos fijamos en el listado `DayDate`, vemos que estas funciones (líneas 638-724) devuelven una fecha creada por `addDays` (línea 571), que invoca `createInstance` (línea 808), que crea `SpreadsheetDate` [G7].

No es recomendable que las clases base conozcan sus variantes. Para corregirlo, debemos usar el patrón de *factoría abstracta*[98] y crear `DayDateFactory`. Esta factoría creará las instancias de `DayDate` que necesitamos y también responderá a preguntas sobre la implementación, como las fechas máxima y mínima.

```
public abstract class DayDateFactory {
    private static DayDateFactory factory = new SpreadsheetDateFactory();
    public static void set Instance(DayDateFactory factory) {
        DayDateFactory.factory = factory;
    }


    protected abstract DayDate _makeDate(int ordinal);
    protected abstract DayDate _makeDate(int day, DayDate.Month month, int year);
    protected abstract DayDate _makeDate(int day, int month, int year);
    protected abstract DayDate _makeDate(java.util.Date date);
    protected abstract int _getMinimumYear();
    protected abstract int _getMaximumYear();


    public static DayDate makeDate(int ordinal) {
        return factory._makeDate(ordinal);
    }


    public static DayDate makeDate(int day, DayDate.Month month, int year) {
        return factory._makeDate(day, month, year);
    }


    public static DayDate makeDate(int day, int month, int year) {
        return factory._makeDate(day, month, year);
    }


    public static DayDate makeDate(java.util.Date date) {
        return factory._makeDate(date);
    }


    public static int getMinimumYear() {
        return factory._getMinimumYear();
    }


    public static int getMaximumYear() {
        return factory._getMaximumYear();
    }
}
```

Esta clase de factoría sustituye los métodos `createInstance` por métodos `makeDate`, lo que mejora ligeramente los nombres [N1]. De forma predeterminada es `SpreadsheetDateFactory` pero se puede cambiar por otra factoría. Los métodos estáticos delegados en métodos abstractos usan una combinación de los *patrones de instancia única*[99], *decorador*[100] y *factoría abstracta* que considero muy útil. `SpreadsheetDateFactory` tiene este aspecto:

```
public class SpreadsheetDateFactory extends DayDateFactory {
    public DayDate _makeDate(int ordinal) {
        return new SpreadsheetDate(ordinal);
    }


    public DayDate _makeDate(int day, DayDate.Month month, int year) {
```

```
            return new SpreadsheetDate(day, month, year);
        }

        public DayDate _makeDate(int day, int month, int year) {
            return new SpreadsheetDate(day, month, year);
        }

        public DayDate _makeDate(Date date) {
            final GregorianCalendar calendar = new GregorianCalendar();
            calendar.setTime(date);
            return new SpreadsheetDate(
                calendar.get(Calendar.DATE),
                DayDate.Month.make(calendar.get(Calendar.MONTH) + 1),
                calendar.get(Calendar.YEAR));
        }

        protected int _getMinimumYear() {
            return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
        }

        protected int _getMaximumYear() {
            return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
        }
    }
```

Como puede apreciar, hemos enviado las variables `MINIMUM_YEAR_SUPPORTED` y `MAXIMUM_YEAR_SUPPORTED` a `SpreadsheetDate`, donde pertenecen [G6].

El siguiente problema de `DayDate` son las constantes de días, comenzando en la línea 109. Deberían ser otra enumeración [J3]. Ya hemos visto este patrón, de modo que no lo repetiremos. Se incluye en los listados definitivos.

Seguidamente, vemos una serie de tablas que comienzan en `LAST_DAY_OF_MONTH` (línea 140). El primer problema con estas tablas es que los comentarios que las describen son redundantes [C3]. Basta con sus nombres, de modo que eliminamos los comentarios.

No hay motivos para que la tabla no sea privada [G8], ya que existe una función estática `lastDayOfMonth` que proporciona los mismos datos.

La siguiente tabla, `AGGREGATE_DAYS_TO_END_OF_MONTH`, es más misteriosa, ya que no se usa en ninguna parte de la estructura JCommon [G9], de modo que la elimino.

Lo mismo sucede con `LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH`.

La siguiente tabla, `AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH`, sólo se usa en `SpreadsheetDate` (líneas 434 y 473), lo que me hace dudar si transferirla a `SpreadsheetDate`. La razón de no cambiarla es que la tabla no es específica de ninguna implementación concreta [G6]. Por otra parte, sólo existe la implementación `SpreadsheetDate`, de modo que la tabla debe acercarse a donde se vaya a usar [G10],

Para zanjar la duda y ser coherentes [G11], deberíamos privatizar la tabla y mostrarla a través de una función como `julianDateOfLastDayOfMonth`. Pero nadie parece que la necesita. Es más,

la tabla se puede cambiar a `DayDate` si una nueva implementación de `DayDate` la necesita. Así que la cambiamos.

Lo mismo sucede con `LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH`.

Tras ello, vemos tres grupos de constantes que se pueden convertir en enumeraciones (líneas 162-205). La primera selecciona una semana de un mes. La transformo en la enumeración `WeekInMonth`.

```
public enum WeekInMonth {
    FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);
    public final int index;

    WeekInMonth(int index) {
        this.index = index;
    }
}
```

El segundo grupo de constantes (líneas 177-187) es más complicado. Las constantes `INCLUDE_NONE`, `INCLUDE_FIRST`, `INCLUDE_SECOND` e `INCLUDE_BOTH` se usan para describir si las fechas finales de un intervalo deben incluirse en el mismo. Matemáticamente, se describe como intervalo abierto, intervalo a medio abrir e intervalo cerrado. Creo que resulta más claro con la nomenclatura matemática [N3], de modo que lo cambio por la enumeración `DateInterval` con los enumeradores `CLOSED`, `CLOSED_LEFT`, `CLOSED_RIGHT` y `OPEN`.

El tercer grupo de constantes (líneas 18-205) describen si la búsqueda de un día concreto de la semana devuelve la última instancia, la siguiente o la más próxima. Decidir un nombre adecuado es complicado. Al final, opté por `WeekdayRange` con los enumeradores `LAST`, `NEXT` y `NEAREST`.

Puede que no esté de acuerdo con los nombres elegidos. Para mí tienen sentido. Lo importante es que ahora son más fáciles de cambiar [J3]. Ya no se pasan como enteros, sino como símbolos. Puedo usar la función de cambio de nombre de mi IDE para cambiar los nombres o los tipos sin preocuparme de haberme olvidado de un `-1` o un `2` en alguna parte del código o de que la declaración de un argumento `int` no estén bien descrita.

El campo de descripción de la línea 208 no parece que se use en ninguna parte. Lo elimino junto a sus elementos de acceso y mutación [G9]. También elimino el constructor predeterminado de la línea 213 [G12]. El compilador se encargará de generarlo.

Podemos ignorar el método `isValidWeekdayCode` (líneas 216-238) ya que lo eliminamos al crear la enumeración `Day`.

Llegamos al método `stringToWeekdayCode` (líneas 242-270). Los Javadoc que no suponen demasiado para la firma del método sobran [C3], [G12]. El único valor de este Javadoc es la descripción del valor devuelto

`-1`. Sin embargo, como cambiamos a la enumeración `Day`, el comentario es en realidad incorrecto [C2]. Ahora el método genera `IllegalArgumentException`. Por ello, eliminamos el Javadoc.

También elimino las palabras clave `final` de argumentos y declaraciones de variables, ya que no parecen servir de mucho [G12]. La eliminación de `final` no goza de gran aceptación. Por ejemplo, Robert Simmons[101] recomienda «… diseminar `final` por la totalidad del código». No estoy de acuerdo. Creo que existen casos para usar `final`, por ejemplo como constante ocasional, pero en general, esta palabra clave apenas añade valor y suele ser un estorbo. Puede que lo piense porque el tipo de errores que puede capturar `final` ya se capturan en las pruebas de unidad que he creado.

Las instrucciones `if` duplicadas [G5] del bucle `for` (líneas 259 y 263) son irrelevantes, de modo que las conecté en una única instrucción `if` con el operador `||`. También usé la enumeración `Day` para dirigir el bucle `for` y realicé otros cambios estéticos.

Este método no pertenece realmente a `DayDate`. En realidad es la función de análisis de `Day`. Por lo tanto, lo cambié a la enumeración `Day`, lo que hizo que aumentara considerablemente de tamaño. Como el concepto de `Day` no depende de `DayDate`, extraje la enumeración `Day` de la clase `DayDate` a un archivo propio [G13].

También cambié la siguiente función, `weekdayCodeToString` (líneas 272-286) a la enumeración `Day` y le asigné el nombre `toString`.

```
public enum Day {
    MONDAY(Calendar.MONDAY),
    TUESDAY(Calendar.TUESDAY),
    WEDNESDAY(Calendar.WEDNESDAY),
    THURSDAY(Calendar.THURSDAY),
    FRIDAY(Calendar.FRIDAY),
    SATURDAY(Calendar.SATURDAY),
    SUNDAY(Calendar.SUNDAY);


    public final int index;
    private static DateFormatSymbols dateSymbols = new DateFormatSymbols();


    Day(int day) {
        index = day;
    }


    public static Day make(int index) throws IllegalArgumentException {
        for (Day d : Day.values())
            if (d.index == index)
                return d;
        throw new IllegalArgumentException(
            String.format("Illegal day index: %d.", index));
    }


    public static Day parse(String s) throws IllegalArgumentException {
        String[] shortWeekdayNames =
            dateSymbols.getShortWeekdays();
        String[] weekDayNames =
            dateSymbols.getWeekdays();


        s = s.trim();
        for (Day day : Day.values()) {
```

```
            if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
                s.equalsIgnoreCase(weekDayNames[day.index])) {
                return day;
            }
        }
        throw new IllegalArgumentException(
        String.format("%s is not a valid weekday string", s));
    }

    public String toString() {
        return dateSymbols.getWeekdays()[index];
    }
}
```

Hay dos funciones `getMonths` (líneas 288-316). La primera invoca la segunda. La segunda solamente se invoca desde la primera. Por ello, las he combinado en una y las he simplificado considerablemente [G9], [G12], [F4]. Por último, he cambiado el nombre por otro más descriptivo [N1].

```
public static String[] getMonthNames() {
    return dateFormatSymbols.getMonths();
}
```

La función `isValidMonthCode` (líneas 326-346) es ahora irrelevante gracias a la enumeración `Month`, de modo que la elimino [G9].

La función `monthCodeToQuarter` (líneas 356-375) parece sufrir *envidia de las características*[102] [G14] y seguramente pertenezca a la enumeración `Month` como método `quarter`, motivo por el que la sustituyo.

```
public int quarter() {
    return 1 + (index-1)/3;
}
```

De este modo, la enumeración `Month` tiene tamaño suficiente como para estar en una clase propia. La extraigo de DayDate para mantener la coherencia con la enumeración `Day` [G11], [G13].

Los dos siguientes métodos tienen el nombre `monthCodeToString` (líneas 377-426). Vemos de nuevo que uno invoca al otro con un indicador. No es recomendable pasar un indicador como argumento de una función, en especial si dicho indicador sólo selecciona el formato del resultado [G15]. Por tanto, cambio de nombre, simplifico y reestructuro estas funciones y las incluyo en la enumeración `Month` [N1], [N3], [C3], [G14].

```
public String toString() {
    return dateFormatSymbols.getMonths()[index - 1];
}

public String toShortString() {
    return dateFormatSymbols.getShortMonths()[index – 1];
}
```

El siguiente método es `stringToMonthCode` (líneas 428-472). Lo cambio de nombre, lo paso a la enumeración `Month` y lo simplifico [N1], [N3], [C3], [G14], [G12].

```
public static Month parse(String s) {
    s = s.trim();
    for (Month m : Month.values())
        if (m.matches(s))
            return m;
    try {
        return make(Integer.parseInt(s));
```

```
        }
        catch (NumberFormatException e) {}
            throw new IllegalArgumentException("Invalid month " + s);
    }

    private boolean matches(String a) {
        return s.equalsIgnoreCase(toString()) ||
            s.equalsIgnoreCase(toShortString());
    }
```

El método `isLeapYear` (líneas 495-517) se puede modificar para que sea más expresivo [G16].

```
    public static boolean isLeapYear(int year) {
        boolean fourth = year % 4 == 0;
        boolean hundredth = year % 100 == 0;
        boolean fourHundredth = year % 400 == 0;
        return fourth && (!hundredth || fourHundredth);
    }
```

La siguiente función, `leapYearCount` (líneas 519-536) no pertenece realmente a `DayDate`. Nadie la invoca, excepto los dos métodos de `SpreadsheetDate`, de modo que la desplazo hacia abajo [G6].

La función `lastDayOfMonth` (líneas 538-560) usa la matriz `LAST_DAY_OF_MONTH`, que en realidad pertenece a la enumeración `Month` [G17], por lo que la cambio de ubicación. También simplifico la función y aumento su expresividad [G16].

```
    public static int lastDayOfMonth(Month month, int year) {
        if (month == Month.FEBRUARY && isLeapYear(year))
            return month.lastDay() + 1;
        else
            return month.lastDay();
    }
```

Ahora empieza a ponerse interesante. La siguiente función es `addDays` (líneas 562-576). En primer lugar, como opera en las variables de `DayDate`, no debería ser estática [G18]. La cambio por un método de instancia. Por otra parte, invoca la función `toSerial`, cuyo nombre deberíamos cambiar por `toOrdinal` [N1]. Por último, el método se puede simplificar.

```
    public DayDate addDays(int days) {
        return DayDateFactory.makeDate(toOrdinal() + days);
    }
```

Lo mismo sucede con `addMonths` (líneas 578-602). Debería ser un método de instancia [G18]. El algoritmo es un tanto complicado, de modo que recurro a la explicación de variables temporales[103] [G19] para que sea más transparente. También cambio el nombre del método `getYYY` por `getYear` [N1].

```
    public DayDate addMonths(int months) {
        int thisMonthAsOrdinal = 12 * getYear() + getMonth().index - 1;
        int resultMonthAsOrdinal = thisMonthAsOrdinal + months;
        int resultYear = resultMonthAsOrdinal / 12;
        Month resultMonth = Month.make(resultMonthAsOrdinal % 12 + 1);
        int lastDayOfResultMonth = lastDayOfMonth(resultMonth, resultYear);
        int resultDay = Math.min(getDayOfMonth(), lastDayOfResultMonth);
        return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);
    }
```

La función addYears (líneas 604-626) es similar al resto.

```
    public DayDate plusYears(int years) {
        int resultYear = getYear() + years;
```

```
    int lastDayOfMonthInResultYear = lastDayOfMonth(getMonth(), resultYear);
    int resultDay = Math.min(getDayOfMonth(), lastDayOfMonthInResultYear);
    return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);
}
```

Hay algo que me preocupa sobre el cambio de estos métodos de estáticos a métodos de instancia. ¿La expresión `date.addDays(5)` aclara que el objeto `date` no cambia y que se devuelve una nueva instancia de `DayDate` o se supone, equivocadamente, que se añaden cinco días al objeto `date`? Pensará que no es un gran problema, pero un fragmento de código como el siguiente puede ser muy engañoso [G20].

```
DayDate date = DateFactory.makeDate(5, Month.DECEMBER, 1952);
date.addDays(7); // desplazar la fecha una semana
```

Un lector de este código podría aceptar que `addDays` cambia el objeto `date`, de modo que necesitamos un nombre que acabe con la ambigüedad [N4]: `plusDays` y `plusMonths`. Creo que la intención del método se captura correctamente por medio de

```
DayDate date = oldDate.plusDays(5);
```

mientras que el siguiente no transmite con fluidez al lector que el objeto `date` ha cambiado:

```
date.plusDays(5);
```

Los algoritmos son cada vez más interesantes. `getPreviousDayOfWeek` (líneas 628-660) funciona pero es complicado. Tras meditar en lo que sucedía [G21], pude simplificarlo y aplicar la explicación de variables temporales [G19] para aclarar su significado. También lo cambié de método estático a método de instancia [G18] y me deshice del método de instancia duplicado [G5] (líneas 997-1008).

```
public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {
    int offsetToTarget = targetDayOfWeek.index = getDayOfWeek().index;
    if (offsetToTarget >= 0)
        offsetToTarget - 7;
    return plusDays(offsetToTarget);
}
```

Sucede exactamente lo mismo con `getFollowingDayOfWeek` (líneas 662-693).

```
public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {
    int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index;
    if (offsetToTarget <= 0)
        offsetToTarget += 7;
    return plusDays(offsetToTarget);
}
```

La siguiente función es `getNearestDayOfWeek` (líneas 695-726), que corregimos en un apartado anterior. Pero esos cambios no son coherentes con el patrón actual de las dos últimas funciones [G11]. Por ello, recurro a la explicación de *variables temporales* [G19] para aclarar el algoritmo.

```
public DayDate getNearestDayOfWeek(final Day targetDay) {
    int offsetToThisWeeksTarget = targetDay.index - getDayOfWeek().index;
    int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
    int offsetToPreviousTarget = offsetToFutureTarget - 7;
```

```
        if (offsetToFutureTarget > 3)
            return plusDays(offsetToPreviousTarget);
        else
            return plusDays(offsetToFutureTarget);
    }
```

El método `getEndOfCurrentMonth` (líneas 728-740) es un tanto extraño ya que es un método de instancia que envidia [G14] a su propia clase aceptado un argumento `DayDate`. Lo convierto en un verdadero método de instancia y clarifico algunos de los nombres.

```
    public DayDate getEndOfMonth() {
        Month month = getMonth();
        int year = getYear();
        int lastDay = lastDayOfMonth(month, year);
        return DayDateFactory.makeDate(lastDay, month, year);
    }
```

La refactorización de `weekInMonthToString` (líneas 742-761) resultó ser muy interesante. Mediante las herramientas de refactorización de mi IDE, primero cambié el método a la enumeración `WeekInMonth` creada antes y después cambié el nombre por `toString`. Tras ello, lo convertí en método de instancia. Todas las pruebas fueron correctas (¿adivina hacia dónde nos dirigimos?).

Seguidamente, eliminé el método. Fallaron cinco afirmaciones (líneas 411-415 del Listado B-4). Cambié estas líneas para usar los nombres de los enumeradores (`FIRST`, `SECOND`, etc.). Las pruebas fueron correctas. ¿Ve por qué? ¿Puede ver también por qué son necesarios estos pasos? La herramienta de refactorización se encargó de que los invocadores anteriores de `weekInMonthToString` invocaran ahora `toString` en el enumerador `weekInMonth` ya que todos los enumeradores implementan `toString` para devolver sus nombres…

Desafortunadamente, me pasé de listo. A pesar de la elegancia de la cadena de refactorización, comprobé que los únicos usuarios de esta función eran las pruebas que acababa de modificar, de modo que las eliminé. Así pues, tras determinar que sólo las pruebas invocaban `relativeToString` (líneas 765-781), eliminé directamente la función y sus pruebas.

Hemos llegado a los métodos abstractos de esta clase abstracta. Y el primero es `toSerial` (líneas 838-844). En un apartado anterior cambié el nombre por `toOrdinal`. Al verlo en este contexto, decidí que el cambio de nombre debería ser por `getOrdinalDay`. El siguiente método abstracto es `toDate` (líneas 838-844). Convierte `DayDate` en `java.util.Date`. ¿Por qué es abstracto? Si analizamos su implementación en `SpreadsheetDate` (líneas 198-207 del Listado B-5), vemos que no depende de la implementación de esa clase [G6]. Por tanto, lo desplazo hacia arriba.

Los métodos `getYYYY`, `getMonth` y `getDayOfMonth` son evidentemente abstractos. Sin embargo, `getDayOfWeek` debería ascender desde `SpreadSheetDate` ya que no depende de nada de lo que encontremos en `DayDate` [G6]. ¿O sí?

Si se fija atentamente (línea 247 del Listado B-5), verá que el algoritmo depende implícitamente del origen del día ordinal (es decir, el día de la semana del día 0). Por ello, aunque esta función carezca de dependencias físicas que no se puedan cambiar a `DayDate`, cuenta con una dependencia lógica. Este tipo de dependencias lógicas me molestan [G22]. Si algo lógico depende de la implementación, también debería haber algo físico. Además, me parece que el propio algoritmo podría ser genérico y que debería depender en menor medida de la implementación [G6]. Por tanto, creé un método abstracto en `DayDate` con el nombre `getDayOfWeekForOrdinalZero` y lo implementé en `SpreadsheetDate` para devolver `Day.SATURDAY`. Tras ello, envié el método `getDayOfWeek` a `DayDate` y lo cambié para que invocara `getOrdinalDay` y `getDayOfWeekForOrdinalZero`.

```
public Day getDayOfWeek() {
    Day startingDay = getDayOfWeekForOrdinalZero();
    int startingOffset = startingDay.index - Day.SUNDAY.index;
    return Day.make((getOrdinalDay() + startingOffset) % 7 + 1);
}
```

Fíjese en el comentario de las líneas 895-899. ¿Necesitamos realmente esta repetición? Como de costumbre, eliminé este comentario junto a los demás.

El siguiente método es `compare` (líneas 902-913). De nuevo, es incorrectamente abstracto [G6], por lo que cambio la implementación a `DayDate`. Además, el nombre no es descriptivo [N1]. En realidad, este método devuelve la diferencia en días desde el argumento, por lo que cambié el nombre por `daysSince`. Tampoco existían pruebas para este método, de modo que las creé.

Las seis siguientes funciones (líneas 915-980) son métodos abstractos que deben implementarse en `DayDate`, por lo que las extraje de `SpreadsheetDate`.

La última función, `isInRange` (líneas 982-995), también debe extraerse y refactorizarse. La instrucción `switch` no es agradable [G23] y se puede modificar si enviamos los casos a la enumeración `DateInterval`.

```
public enum DateInterval {
    OPEN {
        public boolean isIn(int d, int left, int right) {
            return d > left && d < right;
        }
    },
```

```
        CLOSED_LEFT {
            public boolean isIn(int d, int left, int right) {
                return d >= left && d < right;
            }
        },
        CLOSED_RIGHT {
            public boolean isIn(int d, int left, int right) {
                return d > left && d <= right;
            }
        },
        CLOSED {
            public boolean isIn(int d, int left, int right) {
                return d >= left && d <= right;
            }
        };


        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(), left, right);
    }
```

Con esto llegamos al final de `DayDate`. Realizaremos una nueva pasada por la clase completa para comprobar cómo fluye. Primero, el comentario inicial está desfasado, de modo que lo reduzco y lo mejoro [C2].

Tras ello, desplazo las enumeraciones restantes a sus propios archivos [G12].

Seguidamente, desplazo la variable estática (`dateFormatSymbols`) y tres métodos estáticos (`getMonthNames`, `isLeapYear`, `lastDayOfMonth`) a una nueva clase con el nombre `DateUtil` [G6].

Cambio los métodos abstractos a una posición superior, donde pertenecen [G24].

Cambio `Month.make` por `Month.fromInt` [N1] y repito la operación con las demás enumeraciones. También creo un método de acceso `toInt()` para todas las enumeraciones y convierto en privado el campo `index`.

Se produce una interesante duplicación [G5] en `plusYears` y `plusMonths` que conseguí eliminar extrayendo un nuevo método con el nombre `correctLastDayOfMonth`, lo que aclaraba el significado de los tres métodos.

Me deshice del número mágico 1 [G25] y lo sustituí por `Month.JANUARY.toInt()` o `Day.SUNDAY.toInt()`, según el caso. Me detuve en limpiar los algoritmos de `SpreadsheetDate`. El resultado final se puede comprobar en los listados B.7 a B.16.

El alcance del código en `DayDate` se *ha reducido* al 84.9 por 100, no porque se pruebe una cantidad menor de funcionalidad, sino porque la clase se ha reducido tanto que las líneas sin alcance tienen un peso mayor. Ahora, en `DayDate` las pruebas se aplican a 45 de las 53 instrucciones ejecutables. Las líneas sin alcance son tan triviales que no merece la pena probarlas.

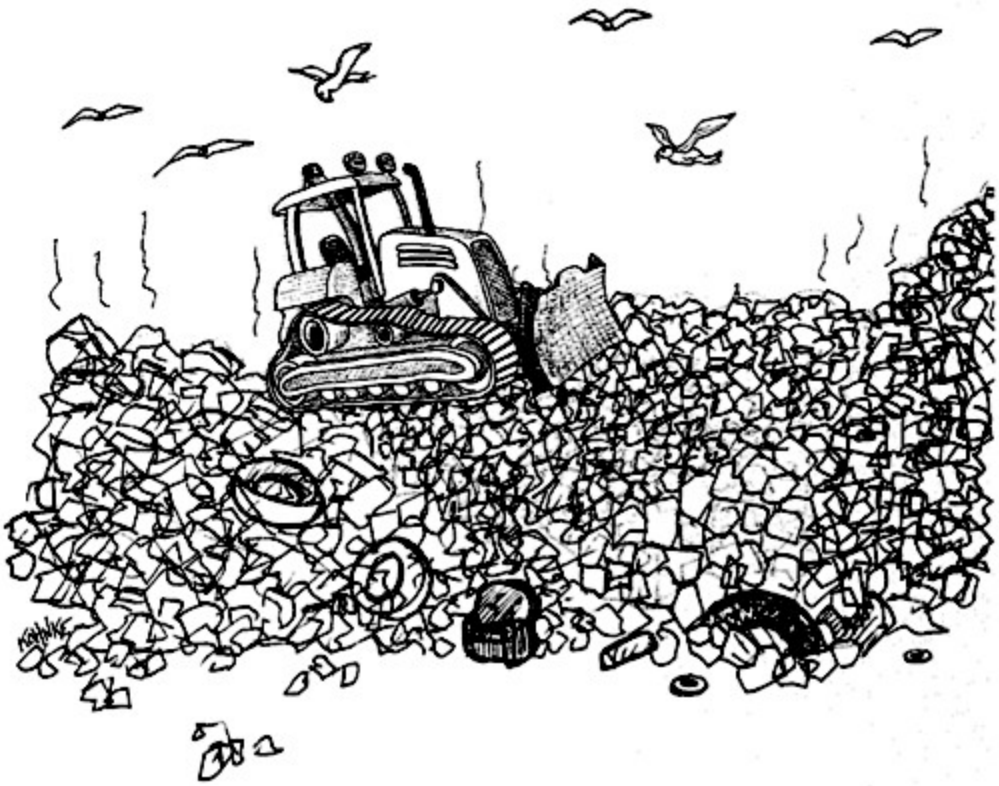
Conclusión

Otra vez hemos aplicado la Regla del Boy Scout. Hemos entregado el código más limpio de lo que lo recibimos. Nos ha llevado tiempo, pero ha merecido la pena. El alcance de las pruebas ha aumentado, hemos corregido algunos errores y hemos aclarado y reducido el tamaño del código. La próxima persona que lo lea seguramente lo encontrará más fácil de leer. Y probablemente esa persona sea capaz de limpiarlo algo más de lo que hemos hecho nosotros.

Bibliografía

- **[GOF]**: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.
- **[Simmons04]**: *Hardcore Java*, Robert Simmons, Jr., O'Reilly, 2004.
- **[Refactoring]**: *Refactoring: Improving the Design of Existing Code*, Martin Fowler et al., Addison-Wesley, 1999.
- **[Beck97]**: *Smalltalk Best Practice Patterns*, Kent Beck, Prentice Hall, 1997.

17
Síntomas y heurística



En su magnífico libro *Refactoring*[104], Martin Fowler identifica diversos síntomas de código (*Smells*). La lista que mostramos a continuación incluye muchos de los síntomas de Martin y otros propios. También contiene otras perlas y heurística que suelo emplear en mi trabajo.

Para compilar esta lista he examinado diversos programas y los he refactorizado. Al aplicar un cambio, me preguntaba el por qué y anotaba el motivo. El resultado es una extensa lista de aspectos que no me «huelen» bien cuando leo código.

La lista se debe leer de arriba a abajo, y también se puede usar como referencia.

Comentarios

C1: Información inapropiada

No es apropiado que un comentario contenga información que se pueda almacenar en otro tipo de sistema como un sistema de control de código fuente, de seguimiento de problemas o de mantenimiento de registros. Los historiales de cambios, por ejemplo, abarrotan los archivos de código con abundante texto sin interés alguno. Por lo general, metadatos como autores, fechas de modificación, números SPR y similares no deben aparecer en los comentarios. Los comentarios deben reservarse para notas técnicas sobre el código y el diseño.

C2: Comentario obsoleto

Un comentario anticuado, irrelevante e incorrecto es obsoleto. Los comentarios envejecen rápidamente. Es recomendable no escribir un comentario que vaya a quedar obsoleto. Si detecta un comentario obsoleto, conviene actualizarlo o eliminarlo lo antes posible. Los comentarios obsoletos tienden a alejarse del código que describían. Se convierten en islas de irrelevancia y desorientación en el código.

C3: Comentario redundante

Un comentario es redundante si describe algo que ya se define correctamente por sí mismo. Por ejemplo:

```
i++; // incrementar i
```

Otro ejemplo es un Javadoc que no dice más (o incluso menos) que la firma de una función:

```
/**
* @param sellRequest
* @return
* @throws ManagedComponentException
```

```
*/
public SellResponse beginSellItem(SellRequest sellRequest)
    throws ManagedComponentException
```

Los comentarios deben comunicar lo que el código no pueda expresar por sí mismo.

C4: Comentario mal escrito

Un comentario que merezca la pena escribir merece la pena ser leído. Si piensa escribir un comentario, asegúrese de que es el mejor que puede crear. Elija las palabras con atención. Use gramática y puntuación correctas. No divague. No afirme lo evidente. Sea breve.

C5: Código comentado

Me molesta ver grandes fragmentos de código comentado. ¿Quién sabe qué antigüedad tienen? ¿Quién sabe si tiene sentido o no? Pero nadie lo borra porque piensa que alguien más lo necesita.

Ese código se estanca y se corrompe, y cada día que pasa es menos relevante. Invoca funciones que ya no existen. Usa variables cuyos nombres han cambiado. Se rige por convenciones obsoletas. Contamina los módulos en los que aparece y distrae a los usuarios que lo leen. El código comentado es una aberración.

Cuando vea código comentado, elimínelo. No se preocupe, el sistema de control de código fuente lo recordará. Si alguien lo necesita, puede consultar una versión anterior. No sufra el código comentado para sobrevivir.

Entorno

E1: La generación requiere más de un paso

La generación de un proyecto debería ser una operación sencilla. No debería tener que comprobar demasiados elementos del control de código fuente. No debería necesitar una secuencia de antiguos comandos ni

secuencias de comandos dependientes del contexto para generar cada elemento. No debería tener que buscar los distintos archivos JAR, XML y similares necesarios para el sistema. Debería finalizar el sistema con un sencillo comando y después ejecutar otro igual de sencillo para generarlo.

```
svn get mySystem
cd mySystem
ant all
```

E2: Las pruebas requieren más de un paso

Debería poder ejecutar todas las pruebas de unidad con un solo comando. En el mejor de los casos, debería poder ejecutarlas pulsando un botón de su IDE. En el peor, debería poder ejecutar un único comando en una línea de comandos. La capacidad de ejecutar todas las pruebas es tan importante que debe ser algo rápido, sencillo y obvio.

Funciones

F1: Demasiados argumentos

Las funciones deben tener un número reducido de argumentos. Lo mejor es que no tengan, seguido de uno, dos y tres argumentos. Más de tres ya es cuestionable y debería evitarse (véase el capítulo 3).

F2: Argumentos de salida

Los argumentos de salida son ilógicos. El lector espera que los argumentos sean entradas, no salidas. Si su función tiene que cambiar el estado de algo, haga que cambie el estado del objeto en el que se invoca (véase el capítulo 3).

F3: Argumentos de indicador

Los argumentos booleanos declaran abiertamente que la función hace más de una cosa. Resultan confusos y deben eliminarse (véase el capítulo 3).

F4: Función muerta

Los métodos que nunca se invocan deben descartarse. La presencia de código muerto es innecesaria. No tema eliminar la función. Su sistema de control de código fuente la recordará.

General

G1: Varios lenguajes en un archivo de código

Los modernos entornos de programación actuales permiten incluir varios lenguajes diferentes en el mismo archivo de código. Por ejemplo, un archivo de Java puede contener fragmentos de XML, HTML, YAML, JavaDoc, JavaScript, y similares. Además de HTML, un archivo JSP podría incluir Java, sintaxis de biblioteca de etiquetas, comentarios en español, Javadoc, XML, JavaScript, etc. Resulta confuso en el mejor de los casos y un desastre en el peor.

Lo ideal sería que el archivo de código incluyera un solo lenguaje pero, en realidad, seguramente tendremos que usar más de uno. Debemos intentar minimizar la cantidad y el alcance de los lenguajes adicionales en nuestros archivos de código.

G2: Comportamiento evidente no implementado

De acuerdo al *Principio de la Mínima Sorpresa*[105], una función o clase debe implementar los comportamientos que otro programador esperaría. Por ejemplo, imagine una función que traduce el nombre de un día en una *enumeración* que represente dicho día.

```
Day day = DayDate.StringToDay(String dayName);
```

Esperaríamos que la cadena «`Monday`» se tradujera en Day.MONDAY. También esperaríamos la traducción de las abreviaturas habituales y que la función ignorara mayúsculas y minúsculas.

Cuando un comportamiento obvio no se implementa, los lectores y usuarios del código ya no dependen de su intuición sobre los nombres de las

funciones. Pierden su confianza en el autor original y se ven obligados a leer los detalles del código.

G3: Comportamiento incorrecto en los límites

Parece evidente afirmar que el código debe comportarse de forma correcta. El problema es que no nos damos cuenta de lo complicado que es dicho comportamiento correcto. Los programadores suelen crear funciones que esperan que funcionen y confían en su intuición más que en comprobar que el código funciona en todos los casos de límites.

No existe sustituto para la meticulosidad. Las condiciones de límite, los casos extremos, las excepciones, representan algo que puede confundir a un algoritmo elegante e intuitivo. *No dependa de su intuición*. Busque todas las condiciones de límite y cree pruebas para cada una.

G4: Medidas de seguridad canceladas

Chernobyl se derritió porque el director de la central ignoró todos y cada uno de los mecanismos de seguridad. Impedían que se realizara un experimento. El resultado fue que el experimento no salió bien y el mundo fue testigo de la primera gran catástrofe nuclear para la población.

Anular las medidas de seguridad es un riesgo. Puede que sea necesario ejercer el control manual sobre `serialVersionUID` pero siempre es arriesgado. La desactivación de determinadas advertencias del compilador (o de todas) puede ayudarle a conseguir la generación, pero corre el riesgo de sufrir interminables sesiones de depuración. Desactivar las pruebas que fallan y convencerse de que conseguirá que después sean satisfactorias es tan erróneo como pensar que sus tarjetas de crédito son dinero gratuito.

G5: Duplicación

Una de las reglas más importantes del libro y que debe tomarse muy en serio. La práctica totalidad de los autores que escriben sobre diseño de *software* mencionan esta regla. Dave Thomas y Andy Hunt la denominaron principio DRY[106] (*Don't Repeat Yourself*, No repetirse). Kent Beck la convirtió en uno de los principios fundamentales de la programación

Extreme y la denominó «Una sola vez». Ron Jeffries sitúa esta regla en segunda posición, por debajo de la consecución satisfactoria de todas las pruebas.

Siempre que vea duplicados en el código, indican una oportunidad de abstracción fallida. La duplicación podría convertirse en una subrutina o en otra clase. Al incluir la duplicación en una abstracción, aumenta el vocabulario del lenguaje del diseño. Otros programadores pueden usar sus creaciones abstractas. El código se vuelve más rápido y menos proclive a errores ya que ha aumentado el nivel de abstracción.

El caso más evidente de duplicación es la presencia de fragmentos de código idéntico que parecen pegados repetidamente por el programador, sin sentido. Conviene reemplazarlos por métodos simples.

Una forma más sutil es la cadena `switch/case` o `if/else` que aparece repetidamente en diversos módulos y que siempre prueba las mismas condiciones. Conviene reemplazar estas cadenas por polimorfismo.

Y más sutiles todavía son los módulos con algoritmos similares pero que no comparten las mismas líneas de código. Sigue siendo duplicación y debe corregirse por medio del patrón de *método de plantilla*[107] o *estrategia*[108].

En realidad, la mayoría de patrones de diseño aparecidos en los últimos 15 años son formas de eliminar la duplicación. Las Formas normales de Codd también son una estrategia para eliminar la duplicación en esquemas de base de datos. Incluso la programación orientada a objetos es una estrategia para organizar módulos y eliminar la duplicación. No debería sorprenderle, ya que se trata de programación estructurada. Creo que el objetivo es evidente: localice los elementos duplicados y elimínelos siempre que pueda.

G6: Código en un nivel de abstracción incorrecto

Es importante crear abstracciones que separen conceptos generales de nivel superior de conceptos detallados de nivel inferior. Para ello, en ocasiones creamos clases abstractas que contengan los conceptos de nivel superior y variantes los de nivel inferior. Si lo hacemos, debemos asegurarnos de que la separación sea completa. Todos los conceptos de nivel inferior deben estar en las variantes y los de nivel superior en la clase base.

Por ejemplo, constantes, variables o funciones de utilidad que solamente pertenezcan a la implementación detallada no deben aparecer en la clase base. La clase base no debe saber nada al respecto de estos elementos.

Esta regla también se aplica a archivos fuente, componentes y módulos. El diseño correcto de *software* requiere la separación de conceptos en distintos niveles y su inclusión en contenedores diferentes. En ocasiones, dichos contenedores son clases base o variantes, y en otros casos son archivos fuente, módulos o componentes. Independientemente del caso, la separación debe ser completa. No queremos que conceptos de nivel inferior y superior se mezclen.

Fíjese en este código:

```
public interface Stack {
    Object pop() throws EmptyException;
    void push(Object o) throws FullException;
    double percentFull();
    class EmptyException extends Exception {}
    class FullException extends Exception {}
}
```

La función `percentFull` se encuentra en el nivel de abstracción equivocado. Aunque hay implementaciones de `Stack` en las que el concepto de amplitud es razonable, otras no pueden conocer su nivel de amplitud. Por tanto, la función debería incluirse en una interfaz derivada como `BoundedStack`.

Pensará que la implementación podría devolver cero si la pila no tuviera límites. El problema es que no existen pilas totalmente sin límites. No se puede evitar `OutOfMemoryException` mediante la comprobación de

```
stack.percentFull() < 50.0.
```

La implementación de esta función para que devuelva 0 sería una mentira.

La moraleja es que no puede mentir o escapar de una abstracción mal ubicada. El aislamiento de abstracciones es una de las operaciones más complicadas para los desarrolladores de *software* y no se puede corregir cuando se realiza de forma incorrecta.

G7: Clases base que dependen de sus variantes

El motivo más habitual para dividir conceptos en clases base y derivadas es para que los conceptos de nivel superior de la clase base sean independientes de los de nivel inferior de las derivadas. Por ello, cuando

vemos clases base que mencionan los nombres de sus variantes, se intuye un problema. Por lo general, las clases base no deben saber nada sobre su derivadas.

Evidentemente, hay excepciones. En ocasiones, el número de variantes es fijo y la clase base tiene código que elegir entre las variantes. Es muy habitual en implementaciones de equipos con estado finito. Sin embargo, en ese caso las variantes y la clase base están íntimamente unidas y siempre se implementan en el mismo archivo jar. En el caso general, deben implementarse en archivos independientes.

Al implementar variantes y clases base en archivos diferentes y garantizar que los archivos de la clase base desconocen el contenido de los archivos de las variantes podemos implementar nuestros sistemas en componentes discretos e independientes. Al modificar dichos componentes, se pueden volver a implementar sin necesidad de implementar de nuevo los componentes base. De este modo se reduce significativamente el impacto del cambio y se facilita el mantenimiento de los sistemas.

G8: Exceso de información

Los módulos bien definidos tienen interfaces reducidas que nos permiten hacer mucho con poco. Los módulos definidos de forma incorrecta tienen interfaces más amplias que nos obligan a usar distintos gestos para realizar operaciones sencillas. Una interfaz bien definida no ofrece demasiadas funciones y las conexiones son reducidas. Una interfaz definida de forma incorrecta ofrece multitud de funciones que invocar y, por tanto, las conexiones son elevadas. Los buenos programadores de *software* aprenden a limitar la parte de sus clases y módulos que muestran en sus interfaces. Cuantos menos métodos tenga una clase, mejor. Cuantas menos variables conozca una función, mejor. Cuantas menos variables de instancia tenga una clase, mejor.

Oculte sus datos. Oculte sus funciones de utilidad. Oculte sus constantes y elementos temporales. No cree clases con multitud de métodos y variables de instancia. No cree multitud de variables y funciones protegidas para sus subclases. Concéntrese en crear interfaces concisas y de tamaño reducido. Limite la información para reducir las conexiones.

G9: Código muerto

El código muerto es el que no se ejecuta. Se encuentra en el cuerpo de una instrucción `if` que comprueba una condición que no sucede. Se encuentra en el bloque `catch` de una instrucción `try` que carece de `throws`. Se encuentra en pequeños métodos de utilidad que nunca se invocan o en condiciones `switch/case` inexistentes.

El problema del código muerto es que con el tiempo empieza a oler. Cuanto más antiguo es, más profundo el hedor que despide. Se debe a que el código muerto no se actualiza al cambiar los diseños. Sigue *compilándose* pero no se rige por nuevas convenciones o reglas. Se creó en un momento en el que el sistema era *diferente*. Debe tener un entierro digno. Bórrelo del sistema.

G10: Separación vertical

Variables y funciones deben definirse cerca de donde se utilicen. Las variables locales deben declararse por encima de su primer uso y deben tener un reducido ámbito vertical. No deben declararse a cientos de líneas de distancia de su uso.

Las funciones privadas deben definirse justo debajo de su primer uso. Pertenecen al ámbito de la clase completa pero conviene limitar la distancia vertical entre las invocaciones y las definiciones. Para localizar una función privada debe bastar con buscar debajo de su primer uso.

G11: Incoherencia

Si hace algo de una forma concreta, aplique la misma técnica a operaciones similares. Esto entronca con el principio de mínima sorpresa. Preste atención a las convenciones que elija y, una vez elegidas, asegúrese de mantenerlas. Si en una función concreta usa la variable `response` para almacenar `HttpServletResponse`, use el mismo nombre de variable en las demás funciones que usen objetos `HttpServletResponse`. Si asigna el nombre `processVerificationRequest` a un método, use un nombre similar, como `processDeletionRequest`, para los métodos que procesen otros tipos de solicitudes.

Este tipo de coherencia, si se aplica repetidamente, facilita la lectura y modificación del código.

G12: Desorden

¿Para qué sirve un constructor predeterminado sin implementación? Únicamente desordena el código y lo inunda de elementos sin sentido. Variables sin usar, funciones que nunca se invocan, comentarios que no añaden información, etc. Todos estos elementos sobran y deben eliminarse. Mantenga limpios sus archivos, bien organizados y sin elementos sobrantes.

G13: Conexiones artificiales

Los elementos que no dependen unos de otros no deben conectarse de forma artificial. Por ejemplo, las *enumeraciones* generales no deben incluirse en clases más específicas ya que esto obliga a la aplicación a saber más sobre dichas clases. Lo mismo sucede con funciones `static` de propósito general declaradas en clases específicas.

Por lo general, una conexión artificial es la que se establece entre dos módulos sin un propósito directo. Es el resultado de incluir una variable, constante o función en una ubicación temporalmente útil pero inadecuada. Es un síntoma de falta de atención.

Piense en dónde debe declarar sus funciones, constantes y variables. No las deje en el punto más cómodo.

G14: Envidia de las características

Uno de los síntomas de Martin Fowler[109]. Los métodos de una clase deben interesarse por las variables y funciones de la clase a la que pertenecen, no por las variables y funciones de otras clases. Cuando un método usa elementos de acceso y mutación de otro objeto para manipular los datos de éste, envidia el ámbito de la clase de dicho objeto. Desea formar parte de la otra clase para tener acceso directo a las variables que manipula.

Por ejemplo:

```
public class HourlyPayCalculator {
    public Money calculateWeeklyPay(HourlyEmployee e) {
        int tenthRate = e.getTenthRate().getPennies();
        int tenthsWorked = e.getTenthsWorked();
```

```
        int straightTime = Math.min(400, tenthsWorked);
        int overTime = Math.max(0, tenthsWorked - straightTime);
        int straightPay = straightTime * tenthRate;
        int overtimePay = (int)Math.round(overTime*tenthRate*1.5);
        return new Money(straightPay + overtimePay);
    }
}
```

El método `calculateWeeklyPay` se acerca al objeto `HourlyEmployee` para obtener los datos sobre los que opera. El método `calculateWeeklyPay` envidia el ámbito de `HourlyEmployee`. Su deseo es formar parte de `HourlyEmployee`.

Es recomendable suprimir la envidia de características ya que muestra los detalles internos de una clase a otra. Sin embargo, en ocasiones es un mal necesario. Fíjese en lo siguiente:

```
public class HourlyEmployeeReport {
    private HourlyEmployee employee;

    public HourlyEmployeeReport(HourlyEmployee e) {
        this.employee = e;
    }

    String reportHours() {
        return String.format(
            “Name: %s\tHours:%d.%1d\n”,
            employee.getName(),
            employee.getTenthsWorked()/10,
            employee.getTenthsWorked()%10);
    }
}
```

Evidentemente, el método `reportHours` envidia la clase `HourlyEmployee`. Por otra parte, no queremos que `HourlyEmployee` tenga que conocer el formato del informe. Al incluir la cadena de formato en la clase `HourlyEmployee` incumpliríamos varios de los principios del diseño orientado a objetos[110]. Conectaría `HourlyEmployee` al formato del informe y lo mostraría en los cambios de dicho formato.

G15: Argumentos de selector

No hay nada más abominable que un argumento `false` aislado al final de la invocación de una función. ¿Qué significa? ¿Qué cambiaría si fuera `true`? No sólo el propósito de un argumento de selector es difícil de recordar, sino que cada argumento de selector combina varias funciones en una. Los argumentos de selector son una forma indolente de evitar dividir una función de gran tamaño en otras menores. Fíjese en lo siguiente:

```
public int calculateWeeklyPay(boolean overtime) {
    int tenthRate = getTenthRate();
    int tenthsWorked = getTenthsWorked();
    int straightTime = Math.min(400, tenthsWorked);
    int overTime = Math.max(0, tenthsWorked - straightTime);
    int straightPay = straightTime * tenthRate;
    double overtimeRate = overtime ? 1.5 : 1.0 * tenthRate;
    int overtimePay = (int)Math.round(overTime*overtimeRate);
    return straightPay + overtimePay;
}
```

Esta función se invoca con `true` si las horas extras se pagan como hora y media, y con `false` si se pagan como una hora normal. Ya es bastante malo tener que recordar lo que significa `calculateWeeklyPay(false)` cada vez que aparezca. Pero lo peor de esta función es que el autor ha perdido la oportunidad de crear lo 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) {
    double bonus = 0.5 * getTenthRate() * overTimeTenths;
    return (int) Math.round(bonus);
}
```

Evidentemente, los selectores no deben ser `boolean`. Pueden ser enumeraciones, enteros u otro tipo de argumento que se use para seleccionar el comportamiento de la función. Es más recomendable tener varias funciones que pasar código a una función para seleccionar el comportamiento.

G16: Intención desconocida

Queremos que el código sea lo más expresivo posible. Expresiones extensas, notación Húngara y números mágicos distorsionan la intención del autor. Por ejemplo, veamos la función `overTimePay` cómo podría haber aparecido:

```
public int m_otCalc() {
    return iThsWkd * iThsRte +
        (int) Math.round(0.5 * iThsRte *
            Math.max(0/ iThsWkd - 400)
        );
}
```

Aunque parezca reducida y densa, también es prácticamente impenetrable. Es recomendable dedicar tiempo a lograr que la intención de nuestro código sea aparente para nuestros lectores.

G17: Responsabilidad desubicada

Una de las principales decisiones de un programador de *software* es dónde ubicar el código. Por ejemplo, dónde incluir la constante `PI`. ¿En la clase `Math`? ¿Pertenece a la clase `Trigonometry`? ¿O a la clase `Circle`?

El principio de mínima sorpresa vuelve a aparecer. El código debe ubicarse donde el lector espera encontrarlo. La constante `PI` debe incluirse

junto a la declaración de las funciones trigonométricas. La constante `OVERTIME_RATE` debe declararse en la clase `HourlyPayCalculator`.

En ocasiones presumimos de dónde añadimos una determinada funcionalidad. Incluimos una función porque nos resulta cómodo pero no porque sea intuitivo para el lector. Por ejemplo, puede que tengamos que imprimir un informe con el total de horas que ha trabajado un empleado. Podríamos sumar las horas en el código que imprime el informe o intentar mantener un total en el código que acepte horarios de trabajo.

Una forma de tomar esta decisión consiste en analizar el nombre de las funciones. Imagine que el módulo del informe tiene la función `getTotalHours`. Imagine también que el módulo que acepta horarios de trabajo tiene la función `saveTimeCard`. ¿Cuál de las dos, por nombre, implica que calcula el total? La respuesta es evidente.

Existen motivos de rendimiento para calcular el total como horarios de trabajo y no como informe impreso. Es correcto, pero el nombre de las funciones debería reflejarlo. Por ejemplo, debería haber una función `computeRunningTotalOfHours` en el módulo de horarios.

G18: Elementos estáticos incorrectos

`Math.max (double a, double b)` es un método estático correcto. No opera en una única instancia; de hecho, sería un error tener que usar `new Math().max(a,b)` o incluso `a.max(b)`. Todos los datos que usa `max` provienen de sus dos argumentos, no de un objeto. Además, es prácticamente imposible que queramos que `Math.max` sea polimórfico. Sin embargo, en ocasiones creamos funciones estáticas que no deben serlo. Fíjese en este ejemplo:

```
HourlyPayCalculator.calculatePay(employee, overtimeRate).
```

De nuevo, parece una función estática razonable. No opera en un objeto concreto y recibe todos los datos de sus argumentos. Sin embargo, existe la posibilidad de que queramos que sea polimórfica. Puede que queramos implementar distintos algoritmos para calcular el precio de la hora, como por ejemplo. `OvertimeHourlyPayCalculator` y `StraightTimeHourlyPayCalculator`. En este caso, la función no debe ser estática. Debería ser una función miembro no estática de `Employee`.

Por lo general, debe decantarse por métodos no estáticos. En caso de duda, convierta la función en no estática. Si realmente quiere que una

función sea estática, asegúrese de que nunca querrá que sea polimórfica.

G19: Usar variables explicativas

Kent Beck escribió sobre este tema en su magnífico libro *Smalltalk Best Practice Patterns*[111] y, más recientemente en *Implementation Patterns*[112]. Una de las técnicas más completas para que un programa sea legible consiste en dividir los cálculos en valores intermedios almacenados en variables con nombres descriptivos. Fíjese en este ejemplo de FitNesse:

```
Matcher match = headerPattern.matcher(line);
if(match.find())
{
    String key = match.group(1);
    String value = match.group(2);
    headers.put(key.toLowercase(), value);
}
```

El simple uso de variables explicativas ilustra con claridad que el primer grupo comparado es la *clave* y el segundo es el *valor*.

Es complicado excederse en esta técnica. Por lo general, es mejor tener más variables explicativas que menos. Es sorprendente que un módulo opaco se vuelva más transparente con tan sólo dividir los cálculos en valores intermedios con los nombres adecuados.

G20: Los nombres de función deben indicar lo que hacen

Fíjese en este código:

```
Date newDate = date.add(5);
```

¿Intuye que se añaden cinco días a la fecha o son semanas u horas? ¿La instancia `date` cambia y la función simplemente devuelve un nuevo objeto `Date` sin cambiar el antiguo? *Por la invocación no podemos saber qué hace la función.*

Si la función añade cinco días a la fecha y después la cambia, el nombre debería ser `addDaysTo` o `increaseByDays`. Si, por otra parte, la función devuelve una nueva fecha con cinco días más pero no cambia la instancia `date`, el nombre debería ser `daysLater` o `daysSince`.

Si tiene que fijarse en la implementación (o documentación) de la función para saber qué hace, tendrá que elegir un nombre más apropiado o modificar la funcionalidad para que se pueda incluir en funciones con nombres más acertados.

G21: Comprender el algoritmo

Se crea gran cantidad de código extraño porque los autores no se esfuerzan en comprender el algoritmo. Consiguen que algo funcione combinando instrucciones `if` e indicadores sin pararse a pensar en qué sucede realmente.

La programación es una tarea de exploración. *Creemos* que conocemos el algoritmo adecuado para algo pero después lo modificamos y variamos hasta conseguir que *funcione*. ¿Cómo sabemos que *funciona*? Porque supera los casos de prueba que pensamos.

No es un enfoque equivocado. De hecho, suele ser la única forma de conseguir que una función haga lo que pensamos que debe hacer. Sin embargo, no basta con conseguir que *funcione*.

Antes de creer que hemos terminado con una función, asegúrese de entender su funcionamiento. No basta con que supere todas las pruebas. Tiene que *estar seguro*[113] de que la solución es la correcta.

Por lo general, la forma óptima de saberlo consiste en refactorizar la función en algo tan limpio y expresivo que su funcionamiento sea *evidente*.

G22: Convertir dependencias lógicas en físicas

Si un módulo depende de otro, dicha dependencia debe ser física, no sólo lógica. El módulo dependiente no debe asumir aspectos (es decir, dependencias lógicas) sobre el módulo del que depende. Por el contrario, debe solicitar de forma explícita al módulo toda la información de la que depende.

Por ejemplo, imagine que tiene que crear una función que imprima un informe de las horas trabajadas por cada empleado. La clase `HourlyReporter` recopila los datos y los pasa a `HourlyReportFormatter` para imprimirlos (véase el Listado 17-1).

Listado 17-1
HourlyReporter.java.

```
public class HourlyReporter {
    private HourlyReportFormatter formatter;
    private List<LineItem> page;
    private final int PAGE_SIZE = 55;

    public HourlyReporter(HourlyReportFormatter formatter) {
        this.formatter = formatter;
        page = new ArrayList<LineItem>();
```

```
    }

    public void generateReport(List<HourlyEmployee> employees) {
        for (HourlyEmployee e : employees) {
            addLineItemToPage(e);
            if (page.size() == PAGE_SIZE)
                printAndClearItemList();
        }
        if (page.size() > 0)
            printAndClearItemList();
    }

    private void printAndClearItemList() {
        formatter.format(page);
        page.clear();
    }

    private void addLineItemToPage(HourlyEmployee e) {
        LineItem item = new LineItem();
        item.name = e.getName();
        item.hours = e.getTenthsWorked() / 10;
        item.tenths = e.getTenthsWorked() % 10;
        page.add(item);
    }

    public class LineItem {
        public String name;
        public int hours;
        public int tenths;
    }
}
```

Este código tiene una dependencia lógica que no se ha convertido en física. ¿La detecta? Es la constante `PAGE_SIZE`. ¿Para qué necesita `HourlyReporter` saber el tamaño de la página? El tamaño de la página debe ser responsabilidad de `HourlyReportFormatter`. La declaración de `PAGE_SIZE` en `HourlyReporter` representa una responsabilidad desubicada [G17] que hace que `HourlyReporter` asuma que conoce el tamaño que debe tener la página. Esta presunción es una dependencia lógica. `HourlyReporter` depende de que `HourlyReportFormatter` pueda procesar tamaños de página de hasta 55. Si alguna implementación de `HourlyReportFormatter` no puede asumir esos tamaños, se producirá un error. Podemos convertir en física esta dependencia si creamos un nuevo método en `HourlyReportFormatter` con el nombre `getMaxPageSize()`. Tras ello, `HourlyReporter` invoca esta función en lugar de usar la constante `PAGE_SIZE`.

G23: Polimorfismo antes que If/Else o Switch/Case

Puede parecer una sugerencia extraña dado el tema descrito en el capítulo 6. En este capítulo, afirmo que las instrucciones `switch` son adecuadas en partes del sistema en las que se añadan más funciones nuevas que tipos nuevos.

Por un lado, la mayoría usamos instrucciones `switch` por ser una solución de fuerza bruta evidente, no por ser la solución perfecta. Por tanto,

esta heurística nos recuerda que debemos considerar el uso de polimorfismo antes de usar `switch`.

Por otra parte, los casos en que las funciones son más volátiles que los tipos son escasos. Por tanto, toda instrucción `switch` es sospechosa.

Suelo aplicar la siguiente regla de una instrucción `switch`: *No puede haber más de una instrucción `switch` por cada tipo de selección. Los casos de esa instrucción `switch` deben crear objetos polimórficos que ocupen el lugar de otras instrucciones `switch` similares en el resto del sistema*.

G24: Seguir las convenciones estándar

Todos los equipos deben seguir un estándar de diseño de código basado en normas comunes de la industria. Este estándar debe especificar aspectos como dónde declarar variables de instancia, cómo asignar nombres a clases, métodos y variables, dónde añadir llaves, etc. El equipo no debe necesitar un documento que describa estas convenciones ya que su código proporciona los ejemplos.

Todos los miembros del equipo deben seguir estas convenciones, lo que significa que no importa dónde añada cada uno las llaves mientras todos estén de acuerdo en dónde añadirlas.

Si desea saber qué convenciones aplico, puede verlas en el código refactorizado de los listados B.7 a B.14 del apéndice B.

G25: Sustituir números mágicos por constantes con nombre

Es probablemente una de las reglas más antiguas del desarrollo de *software*. Recuerdo haberla leído a finales de la década de 1960 en manuales de COBOL, FORTRAN y PL/1. Por lo general, no es recomendable incluir números sin procesar en el código; debe ocultarlos tras constantes con nombres correctos. Por ejemplo, el número 86 400 debe ocultarse tras la constante `SECONDS_PER_DAY`. Si va a imprimir 55 líneas por página, la constante 55 debe ocultarse tras la constante `LINES_PER_PAGE`.

Algunas constantes son tan fáciles de reconocer que no siempre necesitan una constante con nombre tras la que ocultarse mientras se usen junto a código explicativo. Por ejemplo:

```
double milesWalked = feetWalked/5280.0;
int dailyPay = hourlyRate * 8;
double circumference = radius * Math.PI * 2;
```

¿Necesitamos realmente las constantes `FEET_PER_MILE`, `WORK_HOURS_PER_DAY` y `TWO` en los ejemplos anteriores? El último caso es absurdo. Existen ciertas fórmulas en las que las constantes se escriben mejor como números sin procesar. Puede cuestionar el caso de `WORK_HOURS_PER_DAY` ya que las leyes o las convenciones pueden cambiar. Por otra parte, esa fórmula se lee perfectamente si se incluye el 8 por lo que no es necesario añadir 17 más. En el caso de `FEET_PER_MILE`, el número 5280 es una constante tan conocida y exclusiva que los lectores la reconocerán aunque se muestre de forma independiente en una página sin contexto alguno.

Constantes como 3.141592653589793 también son conocidas y reconocibles. Sin embargo, la probabilidad de errores es alta y no conviene mostrarlas tal cual. Siempre que alguien ve 3.1415927535890793, sabe que es π, y no se molestan en examinarlo (¿ha visto el error de un dígito?). Tampoco queremos que la gente use 3.14, 3.14159, 3.142, y similares. Por lo tanto, es una suerte contar con `Math.PI`.

El término número mágico no sólo se aplica a números, sino a todo símbolo que tenga un valor que no sea descriptivo por sí mismo. Por ejemplo:

```
assertEquals(7777, Employee.find(“John Doe”).employeeNumber());
```

En esta afirmación hay dos números mágicos. El primero es obviamente 7777, aunque no significa que no sea obvio. El segundo es «`John Doe`» y su cometido tampoco está claro.

“`John Doe`” es el nombre del empleado #7777 en una conocida base de datos de pruebas creada por nuestro equipo. Todo el mundo sabe que al conectarse a la base de datos, ya cuenta con varios empleados con sus valores y atributos. Además, «`John Doe`» representa el único empleado por horas de la base de datos. Por tanto, la prueba debería ser la siguiente:

```
assertEquals(
    HOURLY_EMPLOYEE_ID,
    Employee.find(HOURLY_EMPLOYEE_NAME).employeeNumber());
```

G26: Precisión

Esperar que la primera coincidencia de una consulta sea la única es una ingenuidad. El uso de números de coma flotante para representar divisas es casi un delito. Evitar bloqueos y/o la administración de transacciones por creer que las actualizaciones concurrentes no son posibles es pura

indolencia. Declarar una variable como `ArrayList` cuando se necesita `List` es un exceso de restricciones. Crear todas las variables como `protected` de forma predeterminada es falta de restricciones.

Al adoptar una decisión en el código, debe hacerlo de forma precisa. Debe saber por qué la adopta y cómo afrontará las excepciones. No sea indolente sobre la precisión de sus decisiones. Si decide invocar una función que pueda devolver `null`, asegúrese de comprobar `null`. Si consulta el que considera el único registro de una base de datos, asegúrese de que el código comprueba que no haya otros. Si tiene que trabajar con divisas, use enteros[114] y aplique el redondeo correcto. Si existe la posibilidad de una actualización concurrente, asegúrese de implementar algún tipo de mecanismo de bloqueo.

En el código, la ambigüedad y las imprecisiones son el resultado de desacuerdos o de indolencia. En cualquier caso, elimínelas.

G27: Estructura sobre convención

Aplique las decisiones de diseño con estructura y no convenciones. Las convenciones de nomenclatura son correctas pero resultan inferiores a estructuras que refuerzan la compatibilidad. Por ejemplo, los casos `switch` con enumeraciones de nombres correctos son inferiores a clases base con métodos abstractos. No estamos obligados a implementar siempre la instrucción `switch/case` de la misma forma, pero las clases base hacen que las clases concretas implementen métodos abstractos.

G28: Encapsular condicionales

La lógica booleana es difícil de entender sin necesidad de verla en el contexto de una instrucción `if` o `while`. Extraiga funciones que expliquen el cometido de la condicional. Por ejemplo:

```
if (shouldBeDeleted(timer))
```

es preferible a

```
if (timer.hasExpired() && !timer.isRecurrent())
```

G29: Evitar condicionales negativas

Las condicionales negativas son más difíciles de entender que las positivas. Por ello, siempre que sea posible, debe expresar las condiciones como positivas. Por ejemplo:

```
if (buffer.shouldCompact())
```

es preferible a

```
if (!buffer.shouldNotCompact())
```

G30: Las funciones sólo deben hacer una cosa

Es tentador crear funciones con varias secciones que realicen una serie de operaciones. Este tipo de funciones hacen más de *una cosa* y deben convertirse en funciones de menor tamaño, cada una para *una cosa*. Por ejemplo:

```
public void pay() {
    for (Employee e : employees) {
        if (e.isPayday()) {
            Money pay = e .calculatePay();
            e.deliverPay(pay);
        }
    }
}
```

Este fragmento de código realiza tres operaciones. Itera por todos los empleados, comprueba si cada uno debe recibir su paga y después paga al empleado. Se podría reescribir de esta forma:

```
public void pay() {
    for (Employee e : employees)
        payifNecessary(e);
}


private void payifNecessary(Employee e) {
    if (e.isPayday())
        calculateAndDeliverPay(e);
}


private void calculateAndDeliverPay(Employee e) {
    Money pay = e.calculatePay();
    e.deliverPay(pay);
}
```

Cada una de estas funciones hace una sola cosa (véase el capítulo 3).

G31: Conexiones temporales ocultas

Las conexiones temporales suelen ser necesarias pero no debe ocultar la conexión. Estructure los argumentos de sus funciones de modo que el orden de invocación sea evidente. Fíjese en lo siguiente:

```
public class MoogDiver {
    Gradient gradient;
    List<Spline> splines;


    public void dive(String reason) {
        saturateGradient();
```

```
            reticulateSplines();
            diveForMoog(reason);
        }
        …
    }
```

El orden de las tres funciones es importante. Debe saturar el degradado antes de poder entrelazar las tiras, para después continuar. Desafortunadamente, el código no aplica esta conexión temporal. Otro programador podría invocar `reticulateSplines` antes de `saturateGradient`, lo que generaría `UnsaturatedGradientException`. Una solución más acertada sería:

```
public class MoogDiver {
    Gradient gradient;
    List<Spline> splines;


    public void dive(String reason) {
        Gradient gradient = saturateGradient();
        List<Spline> splines = reticulateSplines(gradient);
        diveForMoog(splines, reason);
    }
    …
}
```

De este modo se muestra la conexión temporal generando una especie de embudo. Cada función genera un resultado que la siguiente necesita de modo que no se pueden invocar en otro orden.

Puede argumentar que esto aumenta la complejidad de las funciones y tiene razón, pero ese incremento de complejidad sintáctica muestra la verdadera complejidad temporal de la situación.

Observará que he mantenido las variables de instancia. Imagino que son necesarias para los métodos privados de las clases. Incluso así, conservo los argumentos para que la conexión temporal sea explícita.

G32: Evitar la arbitrariedad

Argumente la estructura de su código y asegúrese de que la estructura del código comunica dicho argumento. Si la estructura parece arbitraria, otros se verán con derecho a modificarla.

Si la estructura parece coherente en la totalidad del sistema, otros la usarán y conservarán la convención. Por ejemplo, recientemente repasaba cambios realizados en FitNesse y descubrí lo siguiente:

```
public class AliasLinkWidget extends ParentWidget
{
    public static class VariableExpandingWidgetRoot {
        …


    …
}
```

El problema es que `VariableExpandingWidgetRoot` no debía estar en el ámbito de `AliasLinkWidget`. Es más, otras clases sin relación usaban `AliasLinkWidget.VariableExpandingWidgetRoot` y no tenían por qué saber nada de `AliasLinkWidget`. Puede que el programador añadiera `VariableExpandingWidgetRoot` a `AliasWidget` por comodidad o que realmente pensara que debía formar parte del ámbito de `AliasWidget`. Independientemente del motivo, el resultado será arbitrario. Las clases públicas que no son utilidades de otra clase no deben incluirse en el ámbito de otra clase. La convención es convertirlas en públicas en el nivel superior de su paquete.

G33: Encapsular condiciones de límite

Las condiciones de límite son difíciles de controlar. Aísle su procesamiento y no permita que se transfieran al resto del código. No necesitamos legiones de `+1` y `-1` por todas partes. Fíjese en este ejemplo de FIT:

```
if (level + 1 < tags.length)
{
    parts = new Parse(body, tags, level + 1, offset + endTag);
    body = null;
}
```

`level+1` aparece dos veces. Es una condición de límite que debe encapsularse en una variable con un nombre como `nextLevel`.

```
int nextLevel = level + 1;
if(nextLevel < tags.length) {
    parts = new Parse(body, tags, nextLevel, offset + endTag);
    body = null;
}
```

G34: Las funciones sólo deben descender un nivel de abstracción

Las instrucciones de una función deben crearse en el mismo nivel de abstracción, un nivel por debajo de la operación descrita por el nombre de la función. Puede que sea la heurística más difícil de interpretar y aplicar. Aunque la idea es simple, como humanos nos cuesta mezclar niveles de abstracción. Fíjese en el siguiente código de FitNesse:

```
public String render() throws Exception
{
    StringBuffer html = new StringBuffer("<hr");
    if(size > 0)
        html.append(" size=\"").append(size + 1).append("\"");
    html.append(">");


    return html.toString();
}
```

Si lo analiza, verá lo que sucede. Esta función crea la etiqueta HTML que traza una regla horizontal por la página. La altura de la regla se especifica en la variable `size`.

Fíjese otra vez en el código. 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`. El código proviene del módulo `HruleWidget` de FitNesse. Este módulo detecta una fila de cuatro o más guiones y la convierte en la correspondiente etiqueta `HR`. Cuantos más guiones haya, mayor será el tamaño.

A continuación le muestro la refactorización del código. He cambiado el nombre del campo `size` para reflejar su verdadero cometido. Contenía el número de guiones adicionales.

```
public String render() throws Exception
{
    HtmlTag hr = new HtmlTag(“hr”);
    if (extraDashes > 0)
        hr.addAttribute(“size”, hrSize(extraDashes));
    return hr.html();
}

private String hrSize(int height)
{
    int hrSize = height + 1;
    return String.format(“%d”, hrSize);
}
```

Este cambio separa correctamente los dos niveles de abstracción. La función `render` simplemente crea una etiqueta `HR` sin tener que saber nada sobre su sintaxis HTML. El módulo `HtmlTag` se encarga de los problemas sintácticos.

De hecho, al realizar este cambio detecté un sutil error. El código original no incluía la barra final en la etiqueta `HR`, como haría el estándar XHTML (es decir, generaba `<hr>` en lugar de `<hr/>`). El módulo `HtmlTag` se había modificado hace tiempo para ajustarlo a XHTML.

La separación de niveles de abstracción es una de las tareas más importantes de la refactorización, y también una de las más complejas. Por ejemplo, fíjese en el siguiente código. Fue mi primer intento de separar los niveles de abstracción del método `HruleWidget.render`.

```
public String render() throws Exception
{
    HtmlTag hr = new HtmlTag(“hr”);
    if (size > 0) {
        hr.addAttribute (“size”, “”+(size+1));
    }
    return hr.html();
}
```

Mi objetivo, en esta fase, es crear la separación necesaria y conseguir superar las pruebas. El objetivo lo alcancé fácilmente pero el resultado fue una función con niveles de abstracción mezclados. En este caso, fueron

obra de la etiqueta HR y de la interpretación y el formato de la variable size. Esto indica que al dividir una función en líneas de abstracción, suelen aparecer nuevas líneas de abstracción ocultas por la estructura anterior.

G35: Mantener los datos configurables en los niveles superiores

Si tiene una constante como un valor predeterminado o de configuración que se conoce y se espera en un nivel superior de abstracción, no debe sepultarla en una función de nivel inferior. Muéstrela como argumento para esa función de nivel inferior invocado desde la función de nivel superior. Fíjese en este ejemplo de FitNesse:

```
        public static void main(String[] args) throws Exception
        {
            Arguments arguments = parseCommandLine(args);
            …
        }


    public class Arguments
    {
        public static final String 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 línea de comandos se analizan en la primera línea ejecutable de FitNesse. Los valores predeterminados de dichos argumentos se especifican al inicio de la clase `Argument`. No tiene que buscar instrucciones como la siguiente en los niveles inferiores del sistema:

```
if (arguments.port == 0) // usar 80 de forma predeterminada
```

Las constantes de configuración se encuentran en un nivel superior y son fáciles de cambiar. Se pasan al resto de la aplicación. Los niveles inferiores de la aplicación no poseen los valores de estas constantes.

G36: Evitar desplazamientos transitivos

Por lo general, no es recomendable que un módulo sepa demasiado sobre sus colaboradores. En concreto, si A colabora con B y B con C, no queremos que los módulos que usan A sepan nada sobre C (por ejemplo, o queremos `a.getB().getC().doSomething();`).

Es lo que en ocasiones se denomina Ley de Demeter. Los programadores pragmáticos lo denominan *Crear código silencioso*[115]. En cualquier caso, se trata de garantizar que los módulos sólo tienen conocimiento de sus colaboradores inmediatos y no del mapa de

navegación completo del sistema. Si varios módulos usan alguna variante de la instrucción `a.getB().getC()`, sería complicado cambiar el diseño y la arquitectura para intercalar Q entre B y C. Tendría que localizar todas las instancias de `a.getB().getC()` y convertirlas a `a.getB().getQ().getC()`. Es la forma en que las arquitecturas se vuelven rígidas. Demasiados módulos saben demasiado sobre la arquitectura. Por el contrario, queremos que nuestros colaboradores intermedios ofrezcan todos los servicios que necesitamos. No debemos deambular por el gráfico de objetos del sistema en busca del método que necesitamos invocar. Bastaría con poder usar:

```
myCollaborator.doSomething().
```

Java

J1: Evitar extensas listas de importación mediante el uso de comodines

Si usa dos o más clases de un paquete, importe el paquete completo con

```
import package.*;
```

Las listas extensas de importaciones intimidan al lector. No queremos colapsar la parte superior de los módulos con 80 líneas de importaciones, sino que sean una instrucción concisa de los paquetes con los que colaboramos.

Las importaciones específicas son dependencias rígidas, mientras que las importaciones de comodín no. Si importa una clase concreta, esa clase *debe* existir, pero si importa un paquete con un comodín, no es necesario que existan clases concretas. La instrucción de importación simplemente añade el paquete a la ruta de búsqueda al localizar los nombres. Por tanto, no se genera una verdadera dependencia en estas importaciones y permiten aligerar las conexiones de nuestros módulos. En ocasiones, la lista extensa de importaciones puede resultar útil. Por ejemplo, si tiene que trabajar con código de legado y desea saber para qué clases crear elementos ficticios, puede examinar la lista de importaciones concretas para determinar los verdaderos nombres cualificados de todas esas clases y después añadirlos. No obstante, este uso de las importaciones concretas no es habitual. Es más, muchos IDE modernos le permiten convertir las importaciones con

comodines en una lista de importaciones concretas con un solo comando. Por tanto, incluso en el caso anterior, es recomendable usar comodines. Las importaciones de comodín pueden probar conflictos de nombres y ambigüedades. Dos clases con el mismo nombre pero en paquetes diferentes tienen que importarse de forma concreta o al menos cualificarse de forma específica cuando se usen. Puede resultar molesto pero no es habitual que el uso de importaciones de comodín sea más indicado que el de importaciones concretas.

J2: No heredar constantes

Lo he visto muchas veces y siempre me molesta. Un programador añade constantes a una interfaz y después accede a las mismas heredando dicha interfaz. Fíjese en el siguiente código:

```
public class HourlyEmployee extends Employee {
    private int tenthsWorked;
    private double hourlyRate;
    public Money calculatePay() {
        int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
        int overTime = tenthsWorked - straightTime;
        return new Money(
            hourlyRate * (tenthsWorked + OVERTIME_RATE * overTime)
        );
    }
    …
}
```

¿De dónde salen las constantes `TENTHS_PER_WEEK` y `OVERTIME_RATE`? Puede que provengan de la clase `Employee`; comprobémoslo:

```
public abstract class Employee implements PayrollConstants {
    public abstract boolean isPayday();
    public abstract Money calculatePay();
    public abstract void deliverPay(Money pay);
}
```

No, de ahí no. ¿Entonces de dónde? Fíjese atentamente en la clase `Employee`. Implementa `PayrollConstants`.

```
public interface PayrollConstants {
    public static final int TENTHS_PER_WEEK = 400;
    public static final double OVERTIME_RATE = 1.5;
}
```

Es horrible. Las constantes se ocultan en la parte superior de la jerarquía de herencia. No use la herencia para burlar las reglas de ámbito del lenguaje. Use una importación estática:

```
import static PayrollConstants.*;

public class HourlyEmployee extends Employee {
    private int tenthsWorked;
    private double hourlyRate;

    public Money calculatePay() {
        int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
        int overTime = tenthsWorked - straightTime;
        return new Money(
            hourlyRate * (tenthsWorked + OVERTIME_RATE * overTime)
        );
```

```
    }
    …
}
```

J3: Constantes frente a enumeraciones

Ahora que se han añadido enumeraciones al lenguaje (Java 5), ¡úselas! No recurra al viejo truco de `public static final int`. El significado de `int` se puede perder. El de `enum` no, ya que pertenece a una enumeración con nombre.

Es más, analice atentamente la sintaxis de las *enumeraciones*. Pueden tener métodos y campos, lo que las convierte en potentes herramientas que ofrecen mayor expresividad y flexibilidad que los `int`. Fíjese en esta variante del código:

```
public class HourlyEmployee extends Employee {
    private int tenthsWorked;
    HourlyPayGrade grade;


    public Money calculatePay() {
        int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
        int overTime = tenthsWorked - straightTime;
        return new Money(
            grade.rate() * (tenthsWorked + OVERTIME_RATE * overTime)
        );
    }
    …
}

public enum HourlyPayGrade {
    APPRENTICE {
        public double rate() {
            return 1.0;
        }
    },
    LEUTENANT_JOURNEYMAN (
        public double rate() {
            return 1.2;
        }
    },
    JOURNEYMAN {
        public double rate() {
            return 1.5;
        }
    },
    MASTER {
        public double rate() {
            return 2.0;
        }
    };


    public abstract double rate();
}
```

Nombres

N1: Elegir nombres descriptivos

No sea demasiado rápido a la hora de elegir un nombre. Asegúrese de que sea descriptivo. Recuerde que los significados suelen variar cuando el código evoluciona, de modo que debe revisar frecuentemente la corrección de los nombres elegidos.

No es una recomendación de sensaciones. En *software*, los nombres constituyen el 90 por 100 de su legibilidad. Dedique tiempo a seleccionarlos con atención y mantenga su relevancia. Los nombres son demasiado importantes como para tratarlos mal.

Fíjese en el siguiente código. ¿Para qué sirve? Si le muestro el mismo código con nombres bien elegidos, tendrá sentido, pero con este formato no es más que una masa de símbolos y números mágicos.

```
public int x() {
    int q = 0;
    int z = 0;
    for (int kk = 0; kk < 10; kk++) {
        if (l[z] == 10)
        {
            q += 10 + (l[z + 1] + l[z + 2]);
            z += 1;
        }
        else if (l[z] + l[z + 1] == 10)
        {
            q += 10 + l[z + 2];
            z += 2;
        } else {
            q += l[z] + l[z + 1];
            z += 2;
        }
    }
    return q;
}
```

A continuación, el código como debería haberse escrito. Este fragmento es en realidad menos completo que el anterior, pero detectará inmediatamente lo que intenta hacer y es probable que pudiera crear las funciones que faltan en función de ese significado que intuye. Los números mágicos ya no lo son y la estructura del algoritmo es descriptiva y atractiva:

```
public int score() {
    int score = 0;
    int frame = 0;
    for (int frameNumber = 0; frameNumber < 10; frameNumber++) {
        if (isStrike(frame)) {
            score += 10 + nextTwoBallsForStrike(frame);
            frame += 1;
        } else if (isSpare(frame)) {
            score += 10 + nextBallForSpare(frame);
            frame += 2;
        } else {
            score += twoBallsInFrame(frame);
            frame += 2;
        }
    }
    return score;
}
```

Los nombres bien elegidos inundan la estructura del código con descripciones. Dicha inundación define las expectativas del lector sobre el cometido de otras funciones del módulo.

Puede inferir la implementación de `isStrike()` si se fija en el código anterior. Cuando lea el método `isStrike`, será prácticamente lo que

esperaba[116].

```
private boolean isStrike(int frame) {
    return rolls[frame] = 10;
}
```

N2: Elegir nombres en el nivel correcto de abstracción

No elija nombres que comuniquen implementación; seleccione nombres que reflejen el nivel de abstracción de la clase o la función con la que trabaje. Es complicado. De nuevo, nos cuesta mezclar niveles de abstracción. Siempre que realice una pasada por su código, es probable que encuentre una variable con nombre en un nivel demasiado bajo. Cambie esos nombres cuando los vea. Para que el código sea legible se necesita una mejora continua. Fíjese en la siguiente interfaz `Modem`:

```
public interface Modem {
    boolean dial(String phoneNumber);
    boolean disconnect();
    boolean send(char c);
    char recv();
    String getConnectedPhoneNumber();
}
```

Inicialmente parece correcta. Las funciones parecen las adecuadas. De hecho lo son para muchas aplicaciones, pero piense ahora en una aplicación en la que algunos módems no se conecten mediante marcación telefónica, sino mediante cables (como los usados para conexiones domésticas a Internet). Puede que algunos se conecten enviando un número de puerto a un concentrador a través de una conexión USB. Es evidente que la noción de números de teléfono se encuentra en un nivel de abstracción equivocado. Una estrategia de nomenclatura más adecuada para este caso sería la siguiente:

```
public interface Modem {
    boolean connect(String connectionLocator);
    boolean disconnect();
    boolean send(char c);
    char recv();
    String getConnectedLocator();
}
```

Ahora los nombres no se limitan a números de teléfono. Se pueden usar para números de teléfono o para otros tipos de estrategia de conexión.

N3: Usar nomenclatura estándar siempre que sea posible

Los nombres son más fáciles de entender si se basan en una convención o un uso existente. Por ejemplo, si emplea el patrón `DECORATOR`, debería usar la palabra `Decorator` en los nombres de las clases. Por ejemplo,

`AutoHangupModemDecorator` podría ser el nombre de una clase que permite a un módem colgar automáticamente al final de una sesión. Los patrones son un tipo de estándar. En Java, por ejemplo, las funciones que convierten objetos en representaciones de cadena suelen tener el nombre `toString`. Es mejor seguir estas convenciones que inventar otras propias.

Los equipos suelen inventar su propio sistema estándar de nombres para un proyecto concreto. Eric Evans lo denomina *lenguaje omnipresente* del proyecto[117]. Su código debe usar los términos de este lenguaje. En definitiva, cuantos más nombres con significado especial y relevante para su proyecto utilice, más fácil será para los lectores saber de qué trata el código.

N4: Nombres inequívocos

Seleccione nombres que ilustren de forma inequívoca el funcionamiento de funciones y variables. Fíjese en este ejemplo de FitNesse:

```
private String doRename() throws Exception
{
    if (refactorReferences)
        renameReferences();
    renamePage();


    pathToRename.removeNameFromEnd();
    pathToRename.addNameToEnd(newName);
    return PathParser.render(pathToRename);
}
```

El nombre de esta función no indica qué hace, al menos en términos amplios y sin concretar. Además, se refuerza por la presencia de la función `renamePage` dentro de la función `doRename`. ¿Qué indican los nombres sobre la diferencia entre ambas funciones? Nada. Un nombre más acertado para la función sería `renamePageAndOptionallyAllReferences`. Puede parecerle extenso, y lo es, pero sólo se invoca desde un punto del módulo, de modo que su valor descriptivo supera su longitud.

N5: Usar nombres extensos para ámbitos extensos

La longitud de un nombre debe estar relacionada con la de su ámbito. Puede usar nombres de variables breves para ámbitos diminutos pero en ámbitos mayores debe emplear nombres extensos.

Los nombres de variables como `i` y `j` son correctos si su ámbito tiene cinco líneas de longitud. Fíjese en el siguiente fragmento del conocido

juego de los bolos:

```
private void rollMany(int n, int pins)
{
    for (int i=0; i<n; i++)
        g.roll(pins);
}
```

Es totalmente claro y se complicaría si la variable `i` se cambiara por algo como `rollCount`. Por otra parte, las variables y funciones con nombres breves pierden su significado en las grandes distancias. Por tanto, cuanto mayor sea el ámbito del nombre, más extenso y preciso tendrá que ser el nombre.

N6: Evitar codificaciones

Los nombres no deben codificarse con información de tipos o ámbitos. Prefijos como `m_` o `f` no sirven de nada en los entornos actuales. Además, codificaciones de proyecto y/o subsistema como `vis_` (para un sistema de imágenes visuales) distraen la atención y son redundantes. Los entornos actuales proporcionan toda esa información sin tener que modificar los nombres. Aleje sus nombres de la contaminación húngara.

N7: Los nombres deben describir efectos secundarios

Los nombres deben describir todo lo que haga una función, variable o clase. No oculte efectos secundarios con un nombre. No utilice un simple verbo para describir una función que realiza algo más que una simple acción. Fíjese en este código de TestNG:

```
public ObjectOutputStream getOos() throws IOException {
    if (m_oos == null) {
        m_oos = new ObjectOutputStream(m_socket.getOutputStream());
    }
    return m_oos;
}
```

Esta función hace algo más que obtener `oos`; lo crea si todavía no se ha creado. Por lo tanto, un nombre más acertado sería `createOrReturnOos`.

Pruebas (Test)

T1: Pruebas insuficientes

¿Cuántas pruebas debe incluir una *suite* de pruebas? Desafortunadamente, muchos programadores dirían que las que parezcan suficientes. Una *suite* de pruebas debe probar todo lo que pueda fallar. Las pruebas son insuficientes mientras haya condiciones que no se hayan examinado o cálculos que no se hayan validado.

T2: Usar una herramienta de cobertura

Las herramientas de cobertura indican vacíos en su estrategia de pruebas. Facilitan la detección de módulos, clases y funciones insuficientemente probadas. Muchos IDE le ofrecen un indicador visual y marcan en verde las líneas cubiertas y en rojo las no cubiertas. De este modo es más rápido y sencillo detectar instrucciones `if` o `catch` cuyos cuerpos no se han comprobado.

T3: No ignorar pruebas triviales

Son fáciles de redactar y su valor documental es mayor que el coste de crearlas.

T4: Una prueba ignorada es una pregunta sobre una ambigüedad

En ocasiones dudamos de un detalle de comportamiento porque los requisitos no son claros. Podemos expresar nuestra duda sobre los requisitos en forma de prueba comentada o como prueba anotada con `@Ignore`. La decisión depende de si la ambigüedad es sobre algo que se compila o no.

T5: Probar condiciones de límite

Preste especial atención a las pruebas de condiciones de límite. Solemos acertar con la parte central de un algoritmo pero malinterpretar los límites.

T6: Probar de forma exhaustiva junto a los errores

Los errores suelen congregarse. Si detecta un error en una función, es recomendable probarla de forma exhaustiva. Seguramente no sea el único error.

T7: Los patrones de fallo son reveladores

En ocasiones diagnosticamos un problema detectando patrones de fallo en los casos de prueba. Es otro argumento para crear casos de prueba lo más completos posibles. Los casos de prueba completos, si se ordenan de forma razonable, revelan patrones.

Como ejemplo, imagine que ha detectado que todas las pruebas con un entero mayor de cinco caracteres fallan. O que fallan todas las pruebas que pasan un número negativo al segundo argumento de una función. En ocasiones, con ver el patrón de rojos y verdes de un informe de pruebas basta para hacer saltar la chispa y llegar a una solución. En el capítulo 16 encontrará un interesante ejemplo en el caso de `SerialDate`.

T8: Los patrones de cobertura de pruebas pueden ser reveladores

El análisis del código que se ejecuta o no en las pruebas superadas suele indicar porqué fallan las pruebas no superadas.

T9: Las pruebas deben ser rápidas

Una prueba lenta no se ejecuta. Cuando las cosas se ponen feas, las pruebas lentas se eliminan de la *suite*. Por lo tanto, intente que sus pruebas sean rápidas.

Conclusión

Esta lista de heurística y síntomas no se podría considerar completa. De hecho, dudo de que alguna vez exista alguna. Pero puede que ese no sea el objetivo, ya que lo que implica esta lista es un sistema de valores.

El sistema de valores ha sido el objetivo y la base de este libro. El código limpio no se crea siguiendo una serie de reglas. No se convertirá en un maestro del *software* aprendiendo una lista de heurísticas. La profesionalidad y la maestría provienen de los valores que impulsan las disciplinas.

Bibliografía

- **[Refactoring]**: *Refactoring: Improving the Design of Existing Code*, Martin Fowler et al., Addison-Wesley, 1999.
- **[PRAG]**: *The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.
- **[GOF]**: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.
- **[Beck97]**: *Smalltalk Best Practice Patterns*, Kent Beck, Prentice Hall, 1997.
- **[Beck07]**: *Implementation Patterns*, Kent Beck, Addison-Wesley, 2008.
- **[PPP]**: *Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.
- **[DDD]**: *Domain Driven Design*, Eric Evans, Addison-Wesley, 2003.

Apéndice A
Concurrencia II

por Brett L. Schuchert

Este apéndice complementa y amplía el capítulo 13 sobre concurrencia. Se ha escrito como una serie de temas independientes que puede leer en el orden que desee. Algunas secciones están duplicadas para facilitar dicha lectura.

Ejemplo cliente/servidor

Imagine una sencilla aplicación cliente/servidor. Un servidor espera a que un cliente se conecte. Un cliente se conecta y envía una solicitud.

El servidor

A continuación le mostramos una versión simplificada de una aplicación de servidor. El código completo de este ejemplo se recoge en el Listado A-3.

```
ServerSocket serverSocket = new ServerSocket(8009);

while (keepProcessing) {
    try {
        Socket socket = serverSocket.accept();
        process(socket);
    } catch (Exception e) {
        handle(e);
    }
}
```

Esta sencilla aplicación espera una conexión, procesa un mensaje entrante y vuelve a esperar a la siguiente solicitud cliente. El código cliente para conectarse al servidor es el siguiente:

```
private void connectSendReceive(int i) {
    try {
        Socket socket = new Socket (“localhost”, PORT);
        MessageUtils.sendMessage(socket, Integer.toString(i));
        MessageUtils.getMessage(socket);
        socket.close();
    } catch (Exception e) {
        e.printStackTrace();
```

```
    }
}
```

¿Cómo se comporta esta combinación de cliente y servidor? ¿Cómo podemos describir formalmente ese rendimiento? La siguiente prueba afirma que el rendimiento es aceptable:

```
@Test(timeout = 10000)
public void shouldRunInUnder10Seconds() throws Exception {
    Thread[] threads = createThreads();
    startAllThreadsw(threads);
    waitForAllThreadsToFinish(threads);
}
```

Se omite la configuración para que el ejemplo sea sencillo (véase "`ClientTest.java`" más adelante). Esta prueba afirma que debe completarse en 10 000 milisegundos.

Es un ejemplo clásico de validación del rendimiento de un sistema. Este sistema debe completar una serie de solicitudes cliente en 10 segundos. Mientras el servidor pueda procesar cada solicitud cliente a tiempo, la prueba será satisfactoria.

¿Qué sucede si la prueba falla? Aparte de desarrollar algún tipo de bucle de consulta de eventos, no hay mucho que hacer en un único proceso para aumentar la velocidad de este código. ¿Se solucionaría el problema con varios procesos? Puede, pero necesitamos saber cómo se consume el tiempo. Hay dos posibilidades:

- **E/S:** Con un socket, conectándose a la base de datos, esperando al intercambio de memoria virtual, etc.
- **Procesador:** Cálculos numéricos, procesamiento de expresiones regulares, recolección de elementos sin usar, etc.

Los sistemas suelen tener uno de cada, pero para una operación concreta suele haber uno dominante. Si el código se vincula al procesador, mayor cantidad de *hardware* de procesamiento puede mejorar el rendimiento y hacer que se supere la prueba, pero no hay tantos ciclos de CPU disponibles, de modo que añadir procesos a un problema vinculado al procesador no hará que aumente la velocidad.

Por otra parte, si el proceso está vinculado a E/S, la concurrencia puede aumentar la eficacia. Cuando una parte del sistema espera a E/S, otra puede usar ese tiempo de espera para procesar algo distinto, maximizando el uso eficaz de la CPU disponible.

Añadir subprocesos

Imagine que la prueba de rendimiento falla. ¿Cómo podemos mejorar la producción para que la prueba de rendimiento sea satisfactoria? Si el método `process` del servidor está vinculado a la E/S, existe una forma de conseguir que el servidor use subprocesos (basta con cambiar processMessage):

```
void process (final Socket socket) {
    if (socket == null)
        return;

    Runnable clientHandler = new Runnable() {
        public void run() {
            try {
                String message = MessageUtils.getMessage(socket);
                MessageUtils.sendMessage(socket, “Processed: ” + message);
                closeIgnoringException(socket);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    };

    Thread clientConnection = new Thread(clientHandler);
    clientConnection.start();
}
```

Asuma que este cambio hace que la prueba se supere[118]; el código es completo, ¿correcto?

Observaciones del servidor

El servidor actualizado completa satisfactoriamente la prueba en algo más de un segundo. Desafortunadamente, la solución genera ciertos problemas.

¿Cuántos subprocesos podría crear nuestro servidor? El código no define límites de modo que podríamos alcanzar el impuesto por la Máquina virtual de Java (MVJ), suficiente en muchos sistemas sencillos. ¿Pero y si el sistema tiene que asumir multitud de usuarios de una red pública? Si se conectan demasiados usuarios al mismo tiempo, el sistema podría colapsarse.

Pero dejemos temporalmente este problema de comportamiento. La solución mostrada tiene problemas de limpieza y estructura. ¿Cuántas responsabilidades tiene el código del servidor?

- Administración de conexiones.
- Procesamiento de clientes.
- Política de subprocesos.

- Política de cierre del servidor.

Desafortunadamente, todas estas responsabilidades se encuentran en la función `process`. Además, el código cruza varios niveles diferentes de abstracción. Por tanto, a pesar de la reducida función `process`, es necesario dividirlo.

Existen varios motivos para cambiar el servidor; por tanto, incumple el principio de responsabilidad única. Para mantener la limpieza de un sistema concurrente, la administración de subprocesos debe limitarse a una serie de puntos controlados. Es más, el código que gestione los subprocesos únicamente debe encargarse de la gestión de subprocesos. ¿Por qué? Si no existe otro motivo, el control de problemas de concurrencia ya es lo suficientemente complicado como para generar simultáneamente otros problemas no relacionados con la concurrencia.

Si creamos una lista independiente para cada una de las responsabilidades anteriores, incluyendo la administración de subprocesos, al cambiar la estrategia de administración de subprocesos, el cambio tiene un menor impacto sobre el código y no contamina a otras responsabilidades. De este modo también es más sencillo probar las demás responsabilidades sin necesidad de preocuparse de los subprocesos. Veamos la versión actualizada que se encarga de ello:

```
public void run() {
    while (keepProcessing) {
        try {
            ClientConnection clientConnection = connectionManager.awaitClient();
            ClientRequestProcessor requestProcessor
                = new ClientRequestProcessor(clientConnection);
            clientScheduler.schedule(requestProcessor);
        } catch (Exception e) {
            e.printStackTrace();
        }


    }
    connectionManager.shutdown();


}
```

Ahora centra en el mismo punto todos los aspectos relacionados con los subprocesos: `clientScheduler`. Si hay problemas de concurrencia, bastará con examinar un punto concreto:

```
public interface ClientScheduler {
    void schedule(ClientRequestProcessor requestProcessor);
}
```

La política actual es fácil de implementar:

```
public class ThreadPerRequestScheduler implements ClientScheduler {
    public void schedule(final ClientRequestProcessor requestProcessor) {
        Runnable runnable = new Runnable() {
            public void run() {
                requestProcessor.process();
            }
        };
```

```
        Thread thread = new Thread(runnable);
        thread.start();
    }
}
```

Tras aislar la administración de subprocesos, resulta más sencillo cambiar el control de los mismos. Por ejemplo, para cambiar a la estructura `Executor` de Java 5 es necesario crear una nueva clase y conectarla (véase el Listado A-1).

Listado A-1
ExecutorClientScheduler.java.

```
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

public class ExecutorClientScheduler implements ClientScheduler {
    Executor executor;

    public ExecutorClientScheduler(int availableThreads) {
        executor = Executors.newFixedThreadPool(availableThreads);
    }

    public void schedule(final ClientRequestProcessor requestProcessor) {
        Runnable runnable = new Runnable() {
            public void run() {
                requestProcessor.process();
            }
        };
        executor.execute(runnable);
    }
}
```

Conclusión

En este ejemplo concreto, la presencia de la concurrencia ilustra una forma de mejorar la producción de un sistema y otra de validar dicha producción a través de una estructura de pruebas. Al centrar el código de concurrencia en un número reducido de clases, aplicamos el Principio de responsabilidad única. En el caso de la programación concurrente, resulta especialmente importante debido a su complejidad.

Posibles rutas de ejecución

Repase el método `incrementValue`, un método de Java de una línea sin bucles ni ramificaciones:

```
public class IdGenerator {
    int lastIdUsed;

    public int incrementValue() {
        return ++lastIdUsed;
    }
}
```

Ignore el desbordamiento de enteros e imagine que solamente un subproceso accede a una instancia de `IdGenerator`. En este caso existe una sola ruta de ejecución y un único resultado garantizado:

- El valor devuelto es igual al valor de `lastIdUsed`, y ambos son una unidad mayores que antes de invocar el método.

¿Qué sucede si usamos dos subprocesos y no cambiamos el método? ¿Cuáles son los posibles resultados si cada subproceso invoca `incrementValue` una vez? ¿Cuántas rutas de ejecución posibles hay? Primero, los resultados (imagine que el valor inicial de `lastIdUsed` es 93):

- El primer subproceso obtiene el valor 94, el segundo el valor 95 y `lastIdUsed` es 95.
- El primer subproceso obtiene el valor 95, el segundo el valor 94 y `lastIdUsed` es 95.
- El primer subproceso obtiene el valor 94, el segundo el valor 94 y `lastIdUsed` es 94.

El resultado final, aunque sorprendente, es posible. Para ver los distintos resultados, debemos comprender las diferentes rutas de ejecución posibles y cómo las ejecuta la MVJ.

Número de rutas

Para calcular el número de rutas de ejecución posibles, comenzaremos con el código de *bytes* generado. La única línea de Java (`return ++lastIdUsed;`) se convierte en ocho instrucciones de código de *bytes*. Los dos subprocesos pueden intercambiar la ejecución de estas ocho instrucciones del mismo modo que mezclamos las cartas de una baraja[119]. Incluso con sólo ocho cartas en cada mano, el número de posibles resultados es sorprendente.

Para este sencillo caso de N instrucciones en una secuencia, sin bucles ni condicionales y T subprocesos, el número total de posibles rutas de ejecución es igual a:

$$\frac{(NT)!}{N!^T}$$

Calcular las órdenes posibles

Extraído de un correo electrónico de Uncle Bob a Brett:

Con *N* pasos y *T* subprocesos hay *T * N* pasos totales. Antes de cada paso hay un conmutador de contexto que elige entre los subprocesos. Por tanto, cada ruta se representa como una cadena de dígitos que denota los cambios de contexto. Dados los pasos A y B y los subprocesos 1 y 2, las seis rutas posibles son 1122, 1212, 1221, 2112, 2121 y 2211. O, en términos de pasos, A1B1A2B2, A1A2B1B2, A1A2B2B1, A2A1B1B2, A2A1B2B1 y A2B2A1B1. Para tres subprocesos, la secuencia sería 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*. Por tanto, la cadena 111111 no es válida ya que tiene seis instancias de 1 y ninguna de 2 y 3.
Por tanto, necesitamos las permutaciones de *N* 1, *N* 2… y *N T*. En realidad son las permutaciones de *N * T* tomando cada vez *N * T*, que es *(N * T)!*, pero sin los duplicados. Por tanto, el truco consiste en contar los duplicados y restarlos de *(N * T)!*.
Dados dos pasos y dos subprocesos, ¿cuántos duplicados hay? Cada cadena de cuatro dígitos tiene dos 1 y dos 2. Estos pares se pueden intercambiar sin modificar el sentido de la cadena. Podríamos intercambiar los 1 o los 2, o ninguno. Por tanto hay cuatro isomorfas por cada cadena, lo que significa que hay tres duplicados, de modo que tres de las cuatro opciones son duplicados; por otra parte, una de las cuatro permutaciones no son duplicados. 4! * .25 = 6. Este razonamiento parece funcionar.
¿Cuántos duplicados hay? Si *N* = 2 y *T* = 2, podría intercambiar los 1, los 2, o ambos. En el caso de *N* = 2 y *T* = 3, podría intercambiar los 1, los 2, los 3, 1 y 2, 1 y 3, o 2 y 3. El intercambio son las permutaciones de *N*. Imagine que hay *P* permutaciones de *N*. El número de formas diferentes de organizar dichas permutaciones es *P**T*.
Por tanto el número de isomorfas posibles es *N!**T*. Y el número de rutas es (*T*N*)!/(*N!**T*). De nuevo, 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.

En nuestro sencillo caso de una sola línea de código Java, que equivale a ocho líneas de código de *bytes* y a dos subprocesos, el número total de posibles rutas de ejecución es 12 870. Si el tipo de `lastIdUsed` es `long`, cada lectura y escritura se convierte en dos operaciones y no una, y el número de posibilidades asciende a 2 704 156.

¿Qué sucede si realizamos un cambio en este método?

```
public synchronized void incrementValue() {
    ++lastIdUsed;
}
```

El número de posibles rutas de ejecución es dos para dos subprocesos y N! para el caso general.

Un examen más profundo

¿Qué piensa del sorprendente resultado de dos subprocesos que invoquen el método una vez (antes de añadir `synchronized`) y obtengan el mismo resultado numérico? ¿Cómo es posible? Vayamos por partes.

¿Qué es una operación atómica? Podemos definir una operación atómica como toda operación ininterrumpible. Por ejemplo, en el siguiente código, la línea 5, donde se asigna `0` a `lastId`, es atómica ya que de acuerdo

al modelo de memoria de Java, la asignación a un valor de 32 bits es ininterrumpible.

```
01:  public class Example {
02:  int lastId;
03:
04:  public void resetId() {
05:  value = 0;
06:  }
07:
08:  public int getNextId() {
09:  ++value;
10:  }
11:  }
```

¿Qué sucede si cambiamos el tipo de `lastId` de `int` a `long`? ¿Sigue siendo atómica la línea 5? No de acuerdo a la especificación de la MVJ. Podría ser atómica en un procesador concreto, pero según la especificación de la MVJ, la asignación a un valor de 64 bits requiere dos asignaciones de 32 bits. Esto significa que entre la primera y la segunda podría irrumpir otro subproceso y cambiar uno de los valores.

¿Y qué sucede con el operador de preincremento, `++`, de la línea 9? Este operador se puede interrumpir, de modo que no es atómico. Para entenderlo, repasemos el código de *bytes* de ambos métodos.

Antes de continuar, hay tres definiciones importantes:

- **Marco:** La invocación de un método requiere un marco, el cual incluye la dirección de devolución, los parámetros pasados al método y las variables locales definidas en el mismo. Es una técnica estándar empleada para definir una pila de invocaciones, que se usa en muchos lenguajes modernos para permitir la invocación de funciones y métodos básicos, además de invocaciones recursivas.
- **Variable local:** Las variables definidas en el ámbito del método. Todos los métodos no estáticos tienen al menos una variable, `this`, que representa el objeto actual, el objeto que ha recibido el último mensaje (en el subproceso actual) que ha propiciado la invocación del método.
- **Pila de operandos:** Muchas instrucciones de la MVJ aceptan parámetros. La pila de operandos es donde se incluyen dichos parámetros. La pila es una estructura de datos LIFO (*Last-In, First-Out* o Último en entrar, primero en salir) estándar.

Veamos el código de *bytes* generado para `resetId()`.

Nemónico	Descripción	Pila de operandos posterior
`ALOAD 0`	Cargar la variable 0ª en la pila de operandos. ¿Qué es la variable 0ª? Es `this.`, el objeto actual. Al invocar el método, el receptor del mensaje, una instancia de `Example` , se envía a la matriz de variables locales del marco creado para la invocación de métodos. Siempre es la primera variable que se añade a todos los métodos de instancia.	`this`
`ICONST_0`	Incluir el valor constante 0 en la pila de operandos.	`this, 0`
`PUTFIELD lastId`	Almacenar el valor superior de la pila (0) en el valor de campo del objeto denominado por la referencia de objeto una posición alejada de la parte superior de la pila, `this` .	`<Vacío>`

Estas tres instrucciones son atómicas ya que a pesar de que el subproceso que las ejecuta podría verse interrumpido por cualquiera de ellas, la información para la instrucción `PUTFIELD` (el valor constante `0` de la parte superior de la pila y la referencia a éste una posición inferior, junto con el valor del campo) no se ve alterada por ningún otro subproceso. Por tanto, al producirse la asignación, sabemos que el valor 0 se almacena en el valor del campo. La operación es atómica. Todos los operandos procesan información local del método, de modo que no hay interferencias entre subprocesos.

Si estas instrucciones se ejecutan en diez subprocesos, hay 4.38679733629e+24 ordenaciones posibles. Sin embargo, sólo hay un resultado posible, de modo que las distintas ordenaciones son irrelevantes. Y además, se garantiza el mismo resultado para valores `long` en este caso. ¿Por qué? Los diez subprocesos asignan un valor constante. Aunque se entremezclen, el resultado final será el mismo. Habrá problemas con la operación `++` en el método `getNextId`. Imagine que `lastId` contiene `42` al inicio de este método. Veamos el código de *bytes* de este nuevo método:

Nemónico	Descripción	Pila de

		operandos posterior
`ALOAD 0`	Cargar `this` en la pila de operandos.	`this`
`DUP`	Copiar la parte superior de la pila. Ahora tenemos dos copias de `this` en la pila de operandos.	`this, this`
`GETFIELD lastId`	Recuperar el valor del campo `lastId` del objeto al que se apunta en la parte superior de la pila (`this`) y volver a almacenar el valor en la pila.	`this, 42`
`ICONST_1`	Desplazar la constante entera 1 en la pila.	`this, 42, 1`
`IADD`	Suma entera de los dos valores superiores de la pila de operandos y volver a almacenar el resultado en la pila.	`this, 43`
`DUP_X1`	Duplicar el valor 43 y añadirlo delante de `this`.	`43, this, 43`
`PUTFIELD value`	Almacenar el valor superior de la pila de operandos, 43, en el valor de campo del objeto actual, representado por el siguiente valor superior de la pila de operandos, `this`.	`43`
`IRETURN`	Devolver el valor superior (y único) de la pila de operandos.	`<Vacío>`

Imagine que el primer subproceso completa las tres primeras instrucciones, hasta `GETFIELD` incluida y después se interrumpe. Aparece un segundo subproceso y ejecuta el método completo, incrementando `lastId` en uno; devuelve 43. Tras ello, el primer subproceso retoma desde donde se detuvo; 42 sigue en la pila de operandos por ser el valor de `lastId` cuando ejecutó `GETFIELD`. Suma uno para obtener 43 y almacena el resultado.

El valor 43 también se devuelve al primer subproceso. Como resultado, uno de los incrementos se pierde ya que el primer subproceso interfiere con el segundo después de que éste haya interrumpido al primero.

Al convertir el método `getNextId()` en `synchronized` se corrige este problema.

Conclusión

No se necesita un conocimiento extenso del código de *bytes* para entender cómo unos subprocesos interrumpen a otros. Si consigue entender este ejemplo, demostrará la posibilidad de varios subprocesos entrelazados, un conocimiento suficiente.

Dicho esto, lo que este sencillo ejemplo revela es la necesidad de entender el modelo de memoria para saber qué se permite y qué no. Equivocadamente se piensa que el operador ++ (pre o postincremento) es atómico, y evidentemente no lo es. Esto significa que tiene que saber:

- Dónde están los objetos y valores compartidos.
- El código que provoca problemas de lectura/actualización concurrente.
- Cómo evitar que se produzcan dichos problemas.

Conocer su biblioteca

La estructura Executor

Como mostramos en `ExecutorClientScheduler.java`, la estructura `Executor` de Java 5 permite la ejecución sofisticada por medio de agrupaciones de subprocesos. Es una clase del paquete `java.util.concurrent`. Si va a crear subprocesos y no usa una agrupación de subprocesos o utiliza una creada a mano, considere el uso de Executor. Hace que el código sea más limpio, más fácil de entender y de menor tamaño.

La estructura `Executor` agrupa subprocesos, los cambia automáticamente de tamaño y los vuelve a crear si es necesario. También admite *futuros*, una construcción de programación concurrente habitual. La estructura `Executor` funciona con clases que implementan `Runnable` y también con clases que implementan la interfaz `Callable`. `Callable` se parece a `Runnable`, pero puede devolver un resultado, una necesidad habitual en soluciones de múltiples subprocesos.

Un *futuro* resulta muy útil cuando el código tiene que ejecutar varias operaciones independientes y esperar a que terminen:

```
public String processRequest(String message) throws Exception {
    Callable<String> makeExternalCall = new Callable<String>() {
```

```
            public String call() throws Exception {
                String result = “”;
                // realizar solicitud externa
                return result;
            }
        };

        Future<String> result = executorService.submit(makeExternalCall);
        String partialResult = doSomeLocalProcessing();
        return result.get() + partialResult;
    }
```

En este ejemplo, el método comienza a ejecutar el objeto `makeExternalCall`, prosigue con otro procesamiento y la última línea invoca `result.get(),` que se bloquea hasta que el futuro termina.

Soluciones no bloqueantes

La MV Java 5 aprovecha el diseño de los procesadores modernos que admiten actualizaciones fiables y no bloqueantes. Imagine una clase que usa sincronización (y por tanto bloqueo) para proporcionar la actualización compatible con subprocesos de un valor:

```
public class ObjectWithValue {
    private int value;
    public void synchronized incrementValue() { ++value; }
    public int getValue() { return value; }
}
```

Java 5 dispone de varias clases nuevas para este tipo de situaciones, como por ejemplo `AtomicBoolean`, `AtomicInteger` y `AtomicReference`. Podemos modificar el código anterior para usar un enfoque no bloqueante:

```
public class ObjectWithValue {
    private AtomicInteger value = new AtomicInteger(0);

    public void incrementValue() {
        value.incrementAndGet();
    }
    public int getValue() {
        return value.get();
    }
}
```

Aunque use un objeto en lugar de una primitiva y envíe mensajes como `incrementAndGet()` en lugar de `++`, el rendimiento de esta clase supera en la mayoría de los casos al de la versión anterior. En algunos casos será ligeramente más rápido pero los casos en los que es más lento son prácticamente inexistentes.

¿Cómo es posible? Los procesadores modernos disponen de una operación denominada CAS (Compare and Swap, Comparar e intercambiar). Es una operación similar al bloqueo optimista de una base de datos, mientras que la versión sincronizada es similar al bloqueo pesimista.

La palabra clave `synchronized` siempre adquiere un bloqueo, incluso cuando un segundo subproceso no intenta actualizar el mismo valor.

Aunque el rendimiento de los bloqueos intrínsecos ha mejorado con respecto a versiones anteriores, sigue siendo muy costoso.

La versión no bloqueante asume inicialmente que varios subprocesos no modifican el mismo valor con la suficiente periodicidad como para generar un problema. Por el contrario, detecta de forma eficaz si se produce dicha situación y la reintenta hasta que la actualización es satisfactoria. Esta detección suele ser menos costosa que la adquisición de un bloqueo, incluso en situaciones de contención moderada o alta.

¿Cómo lo hace la MV? La operación CAS es atómica. Por tanto, la operación CAS tiene este aspecto:

```
int variableBeingSet;


void simulateNonBlockingSet (int newValue) {
    int currentValue;
    do {
        currentValue = variableBeingSet
    } while(currentValue != compareAndSwap(currentValue, newValue));
}


int synchronized 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 comprueba que la variable establecida sigue teniendo el último valor conocido. En caso afirmativo, se cambia la variable. En caso contrario, la variable no se establece ya que otro subproceso ha conseguido acceder.

El método que realiza el intento (mediante la operación CAS) ve que el cambio no se ha realizado y lo intenta de nuevo.

Clases incompatibles con subprocesos

Existen clases que no son compatibles con subprocesos, como las siguientes:

- `SimpleDateFormat`.
- Conexiones de base de datos.
- Contenedores de `java.util`.
- Servlet.

Algunas clases de colección tienen métodos concretos compatibles con subprocesos. Sin embargo, cualquier operación que invoque más de un método no lo es. Por ejemplo, si no quiere reemplazar algo en `HashTable` porque ya existe, podría crear el siguiente código:

```
if(!hashTable.containsKey(someKey)) {
    hashTable.put(someKey, new SomeValue());
}
```

Cada uno de los métodos es compatible con subprocesos. Sin embargo, otro subproceso podría añadir un valor entre las invocaciones de `containsKey` y `put`. Existen varias formas de solucionar este problema:

- Bloquear primero `HashTable` y comprobar que los demás usuarios de `HashTable` hagan lo mismo; bloqueo basado en clientes:

```
synchronized(map) {
    if(!map.containsKey(key))
        map.put(key,value);
}
```

- Envolver `HashTable` en su propio objeto y usar dos API distintas; bloqueo basado en servidores con un adaptador:

```
public class WrappedHashtable<K, V> {
    private Map<K, V> map = new Hashtable<K, V>();

    public synchronized void putIfAbsent(K key, V value) {
        if (map.containsKey(key))
            map.put(key, value);
    }
}
```

- Usar colecciones compatibles con subprocesos:

```
ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<Integer, String>();
map.putIfAbsent(key, value);
```

Las colecciones de `java.util.concurrent` incluyen operaciones como `putIfAbsent()` para acomodar este tipo de operaciones.

Las dependencias entre métodos pueden afectar al código concurrente

El siguiente ejemplo es una forma de añadir dependencias entre métodos:

```
public class IntegerIterator implements Iterator<Integer>
    private Integer nextValue = 0;
```

```
    public synchronized boolean hasNext() {
        return nextValue < 100000;
    }
    public synchronized Integer next() {
        if (nextValue == 100000)
            throw new IteratorPastEndException();
        return nextValue++;
    }
    public synchronized Integer getNextValue() {
        return nextValue;
    }
}
```

Veamos otro código que usa `IntegerIterator`:

```
IntegerIterator iterator = new IntegerIterator();
while(iterator.hasNext()) {
    int nextValue = iterator.next();
    // hacer algo con nextValue
}
```

Si un subproceso ejecuta este código no habrá problemas. ¿Qué sucede si dos subprocesos intentan compartir una misma instancia de `IntegerIterator` para procesar el valor que reciba cada uno pero cada elemento de la lista sólo se procesa una vez? En la mayoría de los casos, no hay consecuencias negativas; los subprocesos comparten la lista, procesan los elementos devueltos por el iterador y se detienen cuando éste termina. Sin embargo, existe la posibilidad de que al final de la iteración los dos subprocesos interfieran entre ellos y provoquen que uno supere el final del iterador y se genere una excepción.

El problema es el siguiente: El subproceso 1 pregunta `hasNext()`, que devuelve `true`. El subproceso 1 se evita y el subproceso 2 realiza la misma pregunta, que sigue siendo `true`. Tras ello, el subproceso 2 invoca `next()`, que devuelve un valor, como era de esperar, pero como efecto secundario hace que `hasNext()` devuelva `false`.

Se vuelve a iniciar el subproceso 1, pensando que `hasNext()` sigue siendo `true`, y después invoca `next()`. Aunque los métodos concretos están sincronizados, el cliente usa dos métodos.

Es un problema real y un ejemplo que puede surgir en código concurrente. En este caso concreto, el problema es especialmente sutil ya que la única ocasión en la que produce un fallo es durante la iteración final del iterador. Si los subprocesos se dividen de forma correcta, puede que uno supere el final del iterador. Es el tipo de error que surge en un sistema que lleva tiempo en producción, y es difícil de detectar. Tiene tres opciones:

- Tolerar el fallo.
- Solucionar el problema cambiando el cliente: bloqueo basado en el cliente.

- Solucionar el problema cambiando el servidor, lo que también provoca que cambie el cliente: bloqueo basado en el servidor.

Tolerar el fallo

En ocasiones, los sistemas se configuran para que un fallo no produzca daños. Por ejemplo, el cliente anterior podía capturar la excepción y limpiarla, aunque sería un tanto torpe. Es como limpiar fugas de memoria reiniciando a medianoche.

Bloqueo basado en el cliente

Para que `IntegerIterator` funcione correctamente con varios subprocesos, cambie el cliente (y los demás) como se indica a continuación:

```
IntegerIterator iterator = new IntegerIterator();

while (true) {
    int nextValue;
    synchronized (iterator) {
        if (!iterator.hasNext())
            break;
        nextValue = iterator.next();
    }
    doSometingWith(nextValue);
}
```

Cada cliente añade un bloqueo a través de la palabra clave `synchronized`. Esta duplicación incumple el principio DRY, pero puede ser necesaria si el código usa agrupaciones de terceros no compatibles con subprocesos.

La estrategia es arriesgada ya que todos los programadores que usen el servidor deben acordarse de bloquearlo antes de usarlo y de desbloquearlo cuando terminen. Hace muchos años, trabajé en un sistema que usaba el bloqueo basado en el cliente en un recurso compartido. El recurso se usaba en cientos de puntos distintos del código. Un pobre programador se olvidó de bloquear el recurso en uno de esos puntos.

Era un sistema de varios terminales con *software* de contabilidad para el sindicato de transportistas. Local 705. El ordenador se encontraba en una sala de temperatura controlada de un piso elevado, a unas 50 millas al norte de la sede de Local 705. En la sede, decenas de trabajadores introducían datos en las terminales, conectadas al ordenador mediante líneas telefónicas dedicadas y módem semidúplex de 600bps (esto fue hace *mucho*, mucho tiempo).

Una vez al día, una de las terminales se bloqueaba, sin razón aparente. El bloqueo no tenía preferencia alguna por una terminal o una hora concreta. Es como si alguien echara a suertes la terminal que bloquear y la hora del bloqueo. En ocasiones, se bloqueaba más de una terminal. En ocasiones, podían pasar varios días sin bloqueos.

Inicialmente, se optó por reiniciar como solución, pero era complicado coordinar los reinicios. Tenemos que avisar a la sede y esperar a que todos terminaran lo que estuvieran haciendo en todas las terminales. Tras ello, se apagaba el sistema y se reiniciaba. Si alguien estaba haciendo algo importante para lo que necesitaba una o dos horas, la terminal bloqueada tenía que seguir bloqueada.

Tras varias semanas de depuración, descubrimos que la causa era un contador de búfer circular desincronizado con su puntero. Este búfer controlaba la salida a la terminal. El valor del puntero indicaba que el búfer estaba vacío pero el contador mostraba que estaba lleno. Como estaba vacío, no había nada que mostrar; pero como también estaba lleno, no se podía añadir nada al búfer que mostrar en la pantalla.

Sabíamos qué era lo que bloqueaba las terminales pero no qué provocaba la desincronización del búfer circular, por lo que añadimos un truco para resolver el problema. Se podían leer los conmutadores del panel frontal en el ordenador (esto fue hace mucho, mucho, mucho tiempo). Diseñamos una función de trampa que detectaba si uno de los conmutadores se había generado y después buscábamos un búfer circular que estuviera tanto lleno como vacío. Si lo encontrábamos, lo variábamos. ¡*Voilá*! La terminal bloqueada volvía a funcionar. De este modo no era necesario reiniciar el sistema si una terminal se bloqueaba. La sede nos llamaba y nos decía que había un bloqueo, nos acercábamos hasta la sala de ordenadores y pulsábamos un conmutador.

En ocasiones ellos trabajan los fines de semana pero nosotros no. Por ello, añadimos una función al programador que comprobaba los búfer circulares una vez por minuto y restablecía los que estuvieran tanto llenos como vacíos. De este modo se descongestionaban las pantallas antes de que la dirección llegara al teléfono.

Necesitamos varias semanas de análisis de código de lenguaje de ensamblado antes de localizar al culpable. Habíamos calculado que la frecuencia de los bloqueos se debía a un uso desprotegido del búfer circular, así que sólo *era* necesario determinar el uso fallido. Desafortunadamente,

esto fue hace mucho tiempo y no disponíamos de herramientas de búsqueda, referencias cruzadas ni de otras técnicas automáticas de ayuda. Teníamos que escudriñar los listados. En aquel frío invierno de 1971 en Chicago aprendí que los bloqueos basados en el cliente son verdaderamente terribles.

Bloqueo basado en el servidor

La duplicación se puede eliminar si modificamos `IntegerIterator` de esta forma:

```
public class IntegerIteratorServerLocked {
    private Integer nextValue = 0;
    public synchronized Integer getNextOrNull() {
        if (nextValue < 100000)
            return nextValue++;
        else
            return null;
    }
}
```

Y también cambia el código cliente:

```
while (true) {
    Integer nextValue = iterator.getNextOrNull();
    if (next == null)
        break;
    // hacer algo con nextValue
}
```

En este caso, en realidad cambiamos la API de la clase para que sea compatible con el subproceso[120]. El cliente debe realizar una comprobación de `null` en lugar de comprobar `hasNext()`.

Por lo general, el bloqueo basado en el servidor es preferible por estos motivos:

- Reduce el código repetido: El bloqueo basado en el servidor hace que el cliente bloquee correctamente el servidor. Al incluir el código de bloqueo en el servidor, se libera a los clientes para usar el objeto y no tener que preocuparse de crear código de bloqueo adicional.
- Permite un mejor rendimiento: Puede intercambiar un servidor compatible con subprocesos por otro incompatible en caso de desarrollo de un solo subproceso, lo que evita la sobrecarga.
- Reduce las posibilidades de error: Sólo se necesita un programador que se olvide del bloqueo.
- Aplica una única política: La política se aplica solamente al servidor, no a todos los clientes.

- Reduce el ámbito de las variables compartidas: El cliente las desconoce y tampoco sabe cómo se bloquean. Todo se oculta en el servidor. Cuando se produce un fallo, su origen se busca en menos puntos.

  ¿Y si no es el propietario del código de servidor?

- Usar un adaptador para cambiar la API y añadir bloqueo

```
public class ThreadSafeIntegerIterator {
    private IntegerIterator iterator = new IntegerIterator();

    public synchronized Integer getNextOrNull() {
        if(iterator.hasNext())
            return iterator.next();
        return null;
    }
}
```

- Mejor todavía, usar colecciones compatibles con subprocesos con interfaces ampliadas.

Aumentar la producción

Imagine que desea leer el contenido de una serie de páginas de una lista de URL en la red. Al leer cada página, la analizamos para acumular estadísticas. Después de leer todas, imprimimos un informe de resumen.

La siguiente clase devuelve el contenido de una página, dada una URL.

```
public class PageReader {
    //…
    public String getPageFor(String url) {
        HttpMethod method = new GetMethod(url);

        try {
            httpClient.executeMethod(method);
            String response = method.getResponseBodyAsString();
            return response;
        } catch (Exception e) {
            handle(e);
        } finally {
            method.releaseConnection();
        }
    }
}
```

La siguiente clase es el iterador que proporciona el contenido de las páginas en función de un iterador de URL:

```
public class PageIterator {
    private PageReader reader;
    private URLIterator urls;
```

```
    public PageIterator(PageReader reader, URLIterator urls) {
        this.urls = urls;
        this.reader = reader;
    }

    public synchronized String getNextPageOrNull() {
        if (urls.hasNext())
            getPageFor(urls.next());
        else
            return null;
    }

    public String getPageFor(String url) {
        return reader.getPageFor(ur1);
    }
}
```

Se puede compartir una instancia de `PageIterator` entre varios subprocesos distintos, cada uno con su propia instancia de `PageReader` para leer las páginas que obtiene del iterador.

Hemos reducido el tamaño del bloque `synchronized`. Simplemente contiene la sección crítica de `PageIterator`. Siempre conviene sincronizar lo menos posible.

Cálculo de producción de un solo subproceso

Vayamos con los cálculos. Imagine lo siguiente, de acuerdo al argumento anterior:

- Tiempo de E/S para recuperar una página (de media): 1 segundo.
- Tiempo de procesamiento para analizar la página (de media): .5 segundos.
- E/S requiere 0 por 100 de la CPU mientras que el procesamiento requiere 100 por 100.

Si se procesan *N* páginas en un mismo subproceso, el tiempo de ejecución total es de 1.5 segundos * *N*. En la figura A.1 puede ver una instantánea de 13 páginas, aproximadamente 19.5 segundos.

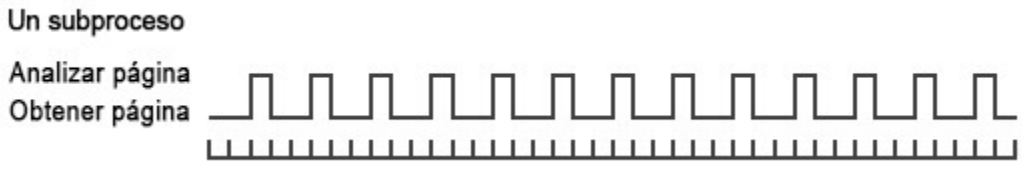


Figura A.1. Un único subproceso

Cálculo de producción con varios subprocesos

Si se pueden recuperar páginas en cualquier orden y procesarlas de forma independiente, entonces es posible usar varios subprocesos para aumentar la producción. ¿Qué sucede si usamos tres subprocesos? ¿Cuántas páginas podemos obtener en el mismo tiempo?

Como se aprecia en la figura A.2, la solución con varios procesos permite que el análisis de las páginas vinculado al proceso se solape con la lectura de las mismas, vinculada a E/S. En un mundo ideal, significaría que el procesador se utiliza totalmente. Cada lectura de página por segundo se solapa con dos análisis. Por tanto, podemos procesar dos páginas por segundo, lo que triplica la producción de la solución con un solo proceso.

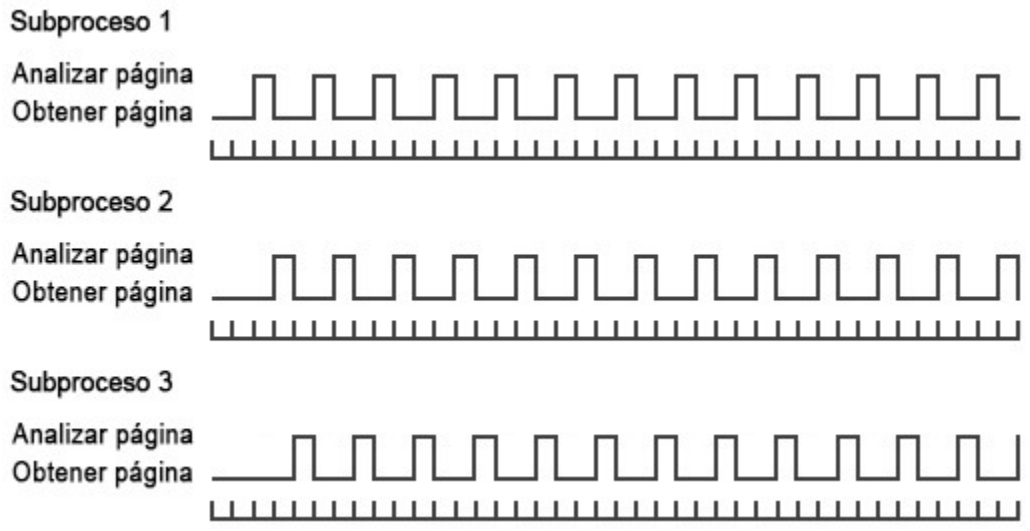


Figura A.2. Tres subprocesos concurrentes.

Bloqueo mutuo

Imagine una aplicación Web con dos agrupaciones de recursos compartidos de tamaño finito:

- Una agrupación de conexiones de base de datos para tareas locales de almacenamiento de procesos.
- Una agrupación de conexiones MQ a un repositorio principal.

Imagine que hay dos operaciones en la aplicación: crear y actualizar:

- Crear: Adquirir una conexión al repositorio principal y la base de datos. Comunicarse con el repositorio principal y después almacenar el trabajo local en la base de datos de procesos.
- Actualizar: Adquirir una conexión a la base de datos y después al repositorio principal. Leer el trabajo de la base de datos y enviarlo al repositorio principal.

¿Qué sucede con los usuarios que superan el tamaño de la agrupación? Imagine que el tamaño de cada agrupación es **10**.

- 10 usuarios intentan usar crear, de modo que se adquieren diez conexiones de base de datos y cada subproceso se interrumpe después de esta adquisición pero antes de adquirir una conexión al repositorio principal.
- 10 usuarios intentan usar actualizar, de modo que se adquieren las diez conexiones al repositorio principal y cada subproceso se interrumpe después de adquirir el repositorio principal pero antes de adquirir una conexión a la base de datos.
- Ahora los 10 subprocesos crear deben esperar a adquirir una conexión al repositorio principal pero los 10 subprocesos actualizar deben esperar a adquirir una conexión a la base de datos.
- Bloqueo mutuo. El sistema no se recupera nunca.

Puede parecerle una situación improbable pero ¿quién desea un sistema que se colapsa cada semana? ¿Quién quiere depurar un sistema con síntomas tan difíciles de reproducir? Es el tipo de problema que tarda semanas en resolverse.

Una solución habitual consiste en añadir instrucciones de depuración para determinar qué sucede. Evidentemente, estas instrucciones cambian tanto el código que el bloqueo mutuo se genera en otras situaciones y tarda meses en volver a producirse[121].

Para solucionar realmente el problema del bloqueo absoluto, debemos entender sus causas. Para que se produzca, deben darse cuatro condiciones:

- Exclusión mutua.
- Bloqueo y espera.
- No expropiación.
- Espera circular.

Exclusión mutua

La exclusión mutua se produce cuando varios subprocesos deben usar los mismos recursos y dichos recursos

- No se pueden usar en varios subprocesos al mismo tiempo.
- Son de número limitado.

Un ejemplo típico de este tipo de recurso es una conexión de base de datos, un archivo abierto para escritura, un bloqueo de registro o un semáforo.

Bloqueo y espera

Cuando un subproceso adquiere un recurso, no lo libera hasta adquirir los demás recursos que necesita y terminar su trabajo.

No expropiación

Un subproceso no puede adueñarse de los recursos de otro. Cuando un subproceso obtiene un recurso, la única forma de que otro lo consiga es que el primero lo libere.

Espera circular

También se denomina abrazo mortal. Imagine dos subprocesos, T1 y T2, y dos recursos, R1 y R2. T1 tiene R1, T2 tiene R2. T1 también necesita R2 y T2 también necesita R1. Es similar al diagrama de la figura A.3:

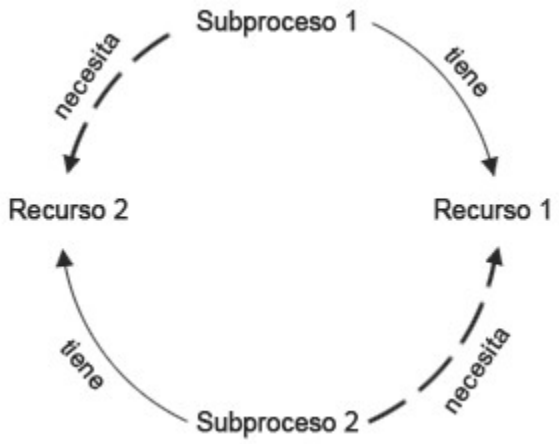


Figura A.3.

Estas cuatro condiciones deben cumplirse para que se produzca un bloqueo mutuo. Si se incumple alguna de ellas, no se producirá.

Evitar la exclusión mutua

Una estrategia para evitar el bloqueo mutuo es impedir la condición de exclusión mutua, por medio de lo siguiente:

- Usar recursos que permitan un uso simultáneo, como por ejemplo,

  `AtomicInteger`.

- Incrementar el número de recursos para que sea igual o mayor que el número de subprocesos implicados.
- Comprobar que todos los recursos están libres antes de adquirir ninguno.

Desafortunadamente, la mayoría de recursos son limitados y no permiten un uso simultáneo, y es habitual que la identidad del segundo recurso se base en los resultados de operar sobre el primero, pero no se desanime, todavía quedan tres condiciones.

Evitar bloqueo y espera

También puede eliminar el bloqueo mutuo si rechaza la espera. Compruebe cada uno de los recursos antes de obtenerlos y libere todos los recursos y comience de nuevo si detecta uno que esté ocupado. Este enfoque genera algunos problemas:

- Inanición: Un subproceso no consigue adquirir los recursos que necesita (puede que tenga una combinación exclusiva de recursos que casi nunca esté disponible).
- Bloqueo activo: Varios subprocesos pueden actuar al unísono, adquirir un recurso y liberarlo, de forma repetida. Es especialmente probable en algoritmos de programación de CPU simples (como dispositivos incrustados o algoritmos de equilibrio de subprocesos escritos a mano).

En ambos casos, se puede reducir la producción. El primero reduce la utilización de la CPU, mientras que el segundo genera una elevada utilización de la CPU sin sentido.

Aunque esta estrategia parezca ineficaz, es mejor que nada. Como ventaja, siempre se puede implementar si todo lo demás falla.

Evitar la expropiación

Otra estrategia para evitar el bloqueo mutuo consiste en permitir que todos los subprocesos se apropien de los recursos de otros. Suele realizarse a través de un sencillo mecanismo de solicitudes. Cuando un subproceso descubre que hay un recurso ocupado, le solicita al propietario que lo libere. Si el propietario también espera a otro recurso, lo libera y comienza de nuevo.

Es similar al enfoque anterior, pero, como ventaja, un subproceso puede esperar a un recurso, lo que reduce el número de reinicios. Sin embargo, la gestión de todas estas solicitudes puede resultar complicada.

Evitar la espera circular

Es el enfoque más habitual para impedir el bloqueo mutuo. En la mayoría de sistemas, basta con una sencilla convención acordada entre ambas partes.

En el ejemplo anterior del subproceso 1 que quiere tanto el recurso 1 como el 2, y el subproceso 2 que desea tanto el recurso 2 como el 1, al forzar a ambos subprocesos a que asignen los recursos en el mismo orden se imposibilita la espera circular.

En general, si todos los subprocesos pueden acordar un orden global de los recursos y si todos asignan los recursos en ese orden, el bloqueo mutuo es imposible. Pero como todas las estrategias, también se pueden producir problemas:

- El orden de adquisición puede no corresponderse al orden de uso; por tanto, un recurso adquirido al inicio puede que no se use hasta el final. Esto puede bloquear recursos más tiempo de lo estrictamente necesario.
- En ocasiones no se puede imponer un orden de adquisición de recursos. Si el ID del segundo recurso proviene de una operación realizada en el primero, ese orden no es factible.

Por tanto, existen varias formas de evitar el bloqueo mutuo. Algunas provocan inanición, mientras que otras usan la CPU en exceso y reducen la capacidad de respuesta. ¡*TANSTAAFL*![122]

El aislamiento de la parte relacionada con subprocesos de su solución para permitir ajustes y experimentación es una forma de aprender a determinar las estrategias óptimas.

Probar código con múltiples subprocesos

¿Cómo se puede crear una prueba que demuestre que el siguiente código no es correcto?

```
01:   public class ClassWithThreadingProblem {
02:   int nextId;
03:
04:   public int takeNextId() {
05:   return nextId++;
06:   }
07:   }
```

Veamos la descripción de una prueba que lo demuestre:

- Recordar el valor actual de `nextId`.
- Crear dos subprocesos y que cada uno invoque `takeNextId()` una vez.
- Comprobar que el valor de `nextId` es dos más que el inicial.
- Ejecutar hasta demostrar que `nextId` sólo se ha incrementado en uno y no en dos.

En el Listado A-2 se reproduce la prueba:

Listado A-2
ClassWithThreadingProblemTest.java.

```
01:  package example;
02:
03:  import static org.junit.Assert.fail;
04:
05:  import org.junit.Test;
06:
07:  public class ClassWithThreadingProblemTest {
08:  @Test
09:  public void twoThreadsShouldFailEventually() throws Exception {
10:  final ClassWithThreadingProblem classWithThreadingProblem
     = new ClassWithThreadingProblem();
11:
12:  Runnable runnable = new Runnable() {
13:  public void run() {
14:  ClassWithThreadingProblem.takeNextId();
15:  }
16:  };
17:
18:  for (int i = 0; i < 50000; ++i) {
19:  int startingId = classWithThreadingProblem.lastId;
20:  int expectedResult = 2 + startingId;
21:
22:  Thread t1 = new Thread(runnable);
23:  Thread t2 = new Thread(runnable);
24:  t1.start();
25:  t2.start();
26:  t1.join();
27:  t2.join();
28:
29:  int endingId = classWithThreadingProblem.lastId;
30:
31:  if (endingId != expectedResult)
32:  return;
33:  }
34:
35:  fail(“Should have exposed a threading issue but it did not.”);
36:  }
37:  }
```

Línea Descripción

10 Crear una sola instancia de `ClassWithThreadingProblem` . Debemos usar la palabra clave final ya que se usa después en una clase interna anónima.

12-16 Crear una clase interna anónima que use la instancia de `ClassWithThreadingProblem` .

18 Ejecutar este código hasta demostrar que falla, pero no tanto como para que la prueba tarde demasiado. Es un acto de equilibrio; no queremos esperar demasiado para demostrar el fallo. Elegir la cantidad de ejecuciones es complicado, aunque como veremos después, esta cifra se puede reducir considerablemente.

19 Recordar el valor inicial, la prueba intenta demostrar que el código de `ClassWithThreadingProblem` es incorrecto. Si se supera la prueba, lo habrá demostrado. Si la prueba falla, habrá sido incapaz de demostrarlo.

20 Esperamos que el valor final sea dos más que el actual.

22-23 Crear dos subprocesos que usen el objeto creado en las líneas 12-16. De este modo contamos con dos posibles subprocesos que intentan usar nuestra instancia de `ClassWithThreadingProblem` y ambos interfieren entre sí.

24-25 Hacer que los dos subprocesos se puedan ejecutar.

26-27 Esperar a que terminen los dos subprocesos antes de comprobar los resultados.

29 Registrar el valor final.

31-32 ¿Es diferente `endingId` a lo que esperábamos? En caso afirmativo, se finaliza la prueba; hemos demostrado que el código es incorrecto. En caso negativo, volver a intentarlo.

35 Si hemos llegado hasta aquí, la prueba no ha podido demostrar que el código de producción era incorrecto en una cantidad de tiempo razonable; el código ha fallado. O no es incorrecto o no hemos realizado suficientes iteraciones para que se produzca la condición de fallo.

Esta prueba establece las condiciones de un problema de actualización concurrente. Sin embargo, el problema es tan infrecuente que la mayoría de las veces la prueba no lo detecta.

En realidad, para detectar el problema debemos establecer el número de iteraciones en más de un millón. Incluso con esa cantidad, en diez ejecuciones de un bucle de 1 000 000, el problema sólo apareció una vez, lo que significa que debemos aumentar las iteraciones para obtener fallos fiables. ¿Cuánto estamos dispuestos a esperar?

Aunque ajustáramos la prueba para obtener fallos fiables en un equipo, seguramente tendríamos que ajustarla con otros valores para ilustrar el fallo en otro equipo, sistema operativo o versión de la MVJ.

Y es un problema *sencillo*. Si no podemos demostrarlo, ¿qué pasará cuando detectemos problemas realmente complejos?

¿Qué enfoques debemos adoptar para demostrar este sencillo fallo? Y, sobre todo, ¿cómo podemos crear pruebas que demuestren fallos en un código más complejo? ¿Cómo podremos saber si el código tiene fallos cuando ni siquiera sabemos dónde buscar?

Veamos algunas sugerencias:

- Pruebas Monte Carlo: Crear pruebas flexibles que se puedan ajustar. Después, ejecutarlas repetidamente, por ejemplo, en un servidor de prueba, y cambiar los valores de ajuste aleatoriamente. Si las pruebas fallan, el código es incorrecto. Diseñe las pruebas en las fases iniciales para que un servidor de integración continua las ejecute lo antes posible. Registre las condiciones de fallo de las pruebas.
- Ejecutar la prueba en todas las plataformas de desarrollo: De forma repetida y continuada. Cuanto más tiempo se ejecuten las pruebas sin fallos, más probable es que
    - El código de producción sea correcto o
    - Las pruebas no sean adecuadas para revelar los problemas.
- Ejecutar las pruebas en un equipo con distintas cargas: Si puede simular cargas similares a las del entorno de producción, hágalo.

Sin embargo, aunque realice todos estos pasos, no es probable que detecte problemas de subprocesos en el código. Los problemas más complicados son los que sólo se producen una vez cada mil millones de oportunidades. Son el azote de los sistemas complejos.

Herramientas para probar código basado en subprocesos

IBM ha creado la herramienta ConTest[123]. Lo que hace es instrumentar las clases para aumentar las probabilidades de que falle el código sin subprocesos.

No tenemos relación directa con IBM ni con el equipo que ha desarrollado ConTest. Un colega nos la descubrió. Tras varios minutos de usarla, notamos una gran mejoría en la detección de errores.

A continuación, le indicamos cómo usar ConTest:

- Crear pruebas y código de producción, asegurándonos que haya pruebas diseñadas específicamente para simular varios usuarios con diferentes cargas, como mencionamos antes.
- Instrumentar el código de pruebas y producción con ConTest.
- Ejecutar las pruebas.

Al instrumentar el código con ConTest, la tasa de éxito pasó de un fallo por cada millón de iteraciones a un fallo en 30 iteraciones. Los valores de bucle de las distintas ejecuciones de la prueba tras la instrumentación son los siguientes: 13, 23, 0, 54, 16, 14, 6, 69, 107, 49, 2. Evidentemente, las clases instrumentadas fallaban antes y con mayor fiabilidad.

Conclusión

En este capítulo hemos realizado un breve recorrido por el vasto y complejo territorio de la programación concurrente. Apenas hemos mostrado la superficie. Nos hemos centrado en disciplinas para mantener la limpieza del código concurrente, pero hay mucho más que aprender si tiene pensado diseñar sistemas concurrentes. Le recomendamos que empiece por el libro de Doug Lea *Concurrent Programming in Java: Design Principles and Patterns*[124].

En este capítulo hemos presentado la actualización concurrente y las disciplinas de sincronización y bloqueo para evitarla. Hemos visto cómo los subprocesos pueden mejorar la producción de un sistema vinculado a E/S y

las técnicas limpias para lograr dichas mejoras. Hemos descrito el bloqueo mutuo y las disciplinas para evitarlo de forma limpia.

Por último, hemos analizado estrategias para mostrar problemas de concurrencia mediante la instrumentación del código.

Ejemplos de código completos

Cliente/Servidor sin subprocesos

Listado A-3
Server.java

```
package com.objectmentor.clientserver.nonthreaded;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

import common.MessageUtils;

public class Server implements Runnable {
    ServerSocket serverSocket;
    volatile boolean keepProcessing = true;

    public Server(int port, int millisecondsTimeout) throws IOException {
        serverSocket = new ServerSocket(port);
        serverSocket.setSoTimeout(millisecondsTimeout);
    }

    public void run() {
        System.out.printf("Server Starting\n");

        while (keepProcessing) {
            try {
                System.out.printf("accepting client\n");
                Socket socket = serverSocket.accept();
                System.out.printf("got client\n");
                process(socket);
            } catch (Exception e) {
                handle(e);
            }
        }
    }

    private void handle(Exception e) {
        if (!(e instanceof SocketException)) {
            e.printStackTrace();
        }
    }

    public void stopProcessing() {
        keepProcessing = false;
        closeIgnoringException(serverSocket);
    }

    void process(Socket socket) {
        if (socket == null)
            return;

        try {
            System.out.printf("Server: getting message\n");
```

```
            String message = MessageUtils.getMessage(socket);
            System.out.printf("Server: got message: %s\n", message);
            Thread.sleep(1000);
            System.out.printf("Server: sending reply: %s\n", message);
            MessageUtils.sendMessage(socket, "Processed: " + message);
            System.out.printf("Server: sent\n");
            closeIgnoringException(socket);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void closeIgnoringException(Socket socket) {
        if (socket != null)
            try {
                socket.close();
            } catch (IOException ignore) {
        }
    }

    private void closeIgnoringException(ServerSocket serverSocket) {
        if (serverSocket != null)
            try {
                serverSocket.close();
            } catch (IOException ignore) {
            }
    }
}
```

Listado A-4
ClientTest.java.

```
package com.objectmentor.clientserver.nonthreaded;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;


import common.MessageUtils;


public class Server implements Runnable {
    ServerSocket serverSocket;
    volatile boolean keepProcessing = true;
    public Server(int port, int millisecondsTimeout) throws IOException {
        serverSocket = new ServerSocket(port);
        serverSocket.setSoTimeout(millisecondsTimeout);
    }


    public void run() {
        System.out.printf("Server Starting\n");


        while (keepProcessing) {
            try {
                System.out.printf("accepting client\n");
                Socket socket = serverSocket.accept();
                System.out.printf("got client\n");
                process(socket);
            } catch (Exception e) {
                handle(e);
            }
        }
    }


    private void handle(Exception e) {
        if (!(e instanceof SocketException)) {
            e.printStackTrace();
        }
    }


    public void stopProcessing() {
        keepProcessing = false;
        closeIgnoringException(serverSocket);
    }


    void process(Socket socket) {
        if (socket == null)
            return;


        try {
            System.out.printf("Server: getting message\n");
            String message = MessageUtils.getMessage(socket);
            System.out.printf("Server: got message: %s\n", message);
```

```
            Thread.sleep(1000);
            System.out.printf("Server: sending reply: %s\n", message);
            MessageUtils.sendMessage(socket, "Processed: " + message);
            System.out.printf("Server: sent\n");
            closeIgnoringException(socket);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void closeIgnoringException(Socket socket) {
        if (socket != null)
            try {
                socket.close();
            } catch (IOException ignore) {
            }
    }

    private void closeIgnoringException(ServerSocket serverSocket) {
        if (serverSocket != null)
            try {
                serverSocket.close();
            } catch (IOException ignore) {
            }
    }
}
```

Listado A-5
MessageUtils.java.

```
package common;

import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.net.Socket;

public class MessageUtils {
    public static void sendMessage(Socket socket, String message)
            throws IOException {
        OutputStream stream = socket.getOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(stream);
        oos.writeUTF(message);
        oos.flush();
    }

    public static String getMessage(Socket socket) throws IOException {
        InputStream stream = socket.getInputStream();
        ObjectInputStream ois = new ObjectInputStream(stream);
        return ois.readUTF();
    }
}
```

Cliente/Servidor con subprocesos

Para cambiar el servidor para que use subprocesos basta con cambiar el mensaje `process` (las nuevas líneas se muestran en negrita para destacarlas):

```
void process(final Socket socket) {
    if (socket == null)
        return;

    Runnable clientHandler = new Runnable() {
        public void run() {
            try {
                System.out.printf("Server: getting message\n");
                String message = MessageUtils.getMessage(socket);
                System.out.printf("Server: got message: %s\n", message);
                Thread.sleep(1000);
                System.out.printf("Server: sending reply: %s\n", message);
                MessageUtils.sendMessage(socket, "Processed: " + message);
```

```
                System.out.printf("Server: sent\n");
                closeIgnoringException(socket);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    };

    Thread clientConnection = new Thread(clientHandler);
    clientConnection.start();
}
```

Apéndice B
org.jfree.date.SerialDate

Listado B-1
SerialDate.Java

```
/*===========================================================
* JCommon: biblioteca gratuita de clases de propósito general para Java(tm)
*===========================================================
*
* (C) Copyright 2000-2005, de Object Refinery Limited y colaboradores.
*
* Información del proyecto: http://www.jfree.org/jcommon/index.html
*
* Esta biblioteca es software gratuito; puede distribuirla y/o modificarla
* bajo las condiciones de la Licencia pública general GNU publicada por
* la Free Software Foundation; ya sea la versión 2.1 de la licencia, u
* otra versión posterior (de su elección).
*
* Esta biblioteca se distribuye con la intención de que sea útil, pero
* SIN GARANTÍA ALGUNA, incluida la garantía implícita de COMERCIABILIDAD
* e IDONEIDAD PARA UN DETERMINADO FIN. Consulte la Licencia
* pública general GNU si necesita más información al respecto.
*
* Debería haber recibido una copia de la Licencia pública general GNU
* junto a esta biblioteca; en caso contrario, contacte con la Free Software
* Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
* EE.UU.
*
* [Java es una marca comercial o marca comercial registrada de Sun
* Microsystems, Inc. en Estados Unidos y otros países.]
*
* ------------------
* SerialDate.java
* ------------------
* (C) Copyright 2001-2005, de Object Refinery Limited.
*
* Autor original: David Gilbert (para Object Refinery Limited);
* Colaborador(es): -;
*
* $Id: SerialDate.java,v 1.7 2005/11/03 09:25:17 mungady Exp $
*
* Cambios (11-Oct-2001)
* -------------------------------------
* 11-Oct-2001: Reorganización de la clase y cambio a un nuevo paquete
* com.jrefinery.date (DG);
* 05-Nov-2001: Se añade un método getDescription() y se elimina la clase
* NotableDate (DG);
* 12-Nov-2001: IBD requiere el método setDescription(), una vez eliminada la clase
```

```
 * NotableDate (DG); Se cambian getPreviousDayOfWeek(),
 * getFollowingDayOfWeek() y getNearestDayOfWeek() para corregir
 * errores (DG);
 * 05-Dic-2001: Error corregido en la clase SpreadsheetDate (DG);
 * 29-May-2002: Se transfieren las constantes de mes a una interfaz independiente
 * (MonthConstants) (DG);
 * 27-Ago-2002: Error corregido en el método addMonths(), gracias a Nálevka Petr (DG);
 * 03-Oct-2002: Errores indicados por Checkstyle (DG) corregidos;
 * 13-Mar-2003: Implementación de Serializable (DG);
 * 29-May-2003: Error corregido en el método addMonths (DG);
 * 04-Sep-2003: Implementación de Comparable. Actualización de los javadoc isInRange (DG);
 * 05-Ene-2005: Error corregido en el método addYears() (1096282) (DG);
 *
 */

package org.jfree.date;

import java.io.Serializable;
import java.text.DateFormatSymbols;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.GregorianCalendar;

/**
* Clase abstracta que define nuestros requisitos para manipular fechas,
* sin limitación a una determinada implementación.
* <P>
* Requisito 1: coincidir al menos con el procesamiento de fechas en Excel;
* Requisito 2: la clase es inmutable;
* <P>
* ¿Por qué no usar java.util.Date? Lo haremos, cuando tenga sentido. En ocasiones,
* java.util.Date puede ser demasiado precisa; representa un instante en el tiempo,
* con una precisión de 1/1000 de segundo (y la fecha depende de la
* zona horaria). En ocasiones sólo querremos representar un día concreto (como el 21
* de enero de 2015) sin preocuparnos de la hora del día, la
* zona horaria u otros aspectos. Para eso hemos definido DayDate.
* <P>
* Puede invocar getInstance() para obtener una subclase concreta de SerialDate,
* sin preocuparse de su implementación exacta
*
* @author David Gilbert
*/
public abstract class SerialDate implements Comparable,
Serializable,
MonthConstants {

/** Para serialización. */
private static final long serialVersionUID = -293716040467423637L;

/** Símbolos de formato de fecha. */
public static final DateFormatSymbols
DATE_FORMAT_SYMBOLS = new SimpleDateFormat().getDateFormatSymbols();

/** Número de serie para el 1 de enero de 1900. */
public static final int SERIAL_LOWER_BOUND = 2;

/** Número de serie para el 31 de diciembre de 9999. */
public static final int SERIAL_UPPER_BOUND = 2958465;

/** Valor de año más bajo admitido por este formato de fecha. */
public static final int MINIMUM_YEAR_SUPPORTED = 1900;

/** Valor de año más alto admitido por este formato de fecha. */
public static final int MAXIMUM_YEAR_SUPPORTED = 9999;

/** Constante útil para lunes; equivale a java.util.Calendar.MONDAY. */
```

```
110   public static final int MONDAY = Calendar.MONDAY;
111
112   /**
113   * Constante útil para martes; equivale a java.util.Calendar.TUESDAY.
114   * /
115   public static final int TUESDAY = Calendar.TUESDAY;
116
117   /**
118   * Constante útil para miércoles; equivale a
119   * java.util.Calendar.WEDNESDAY.
120   */
121   public static final int WEDNESDAY = Calendar.WEDNESDAY;
122
123   /**
124   * Constante útil para jueves; equivale a java.util.Calendar.THURSDAY.
125   */
126   public static final int THURSDAY = Calendar.THURSDAY;
127
128   /** Constante útil para viernes; equivale a java.util.Calendar.FRIDAY. */
129   public static final int FRIDAY = Calendar.FRIDAY;
130
131   /**
132   * Constante útil para sábado; equivale a java.util.Calendar.SATURDAY.
133   */
134   public static final int SATURDAY = Calendar.SATURDAY;
135
136   /** Constante útil para domingo; equivale a java.util.Calendar.SUNDAY. */
137   public static final int SUNDAY = Calendar.SUNDAY;
138
139   /** Número de días de cada mes en años no bisiestos. */
140   static final int[] LAST_DAY_OF_MONTH =
141   {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
142
143   /** Número de días en un año (no bisiesto) hasta el final de cada mes. */
144   static final int[] AGGREGATE_DAYS_TO_END_OF_MONTH =
145   {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
146
147   /** 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   /** Número de días en un año bisiesto hasta el final de cada mes. */
152   static 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   * Número de días en un año bisiesto hasta el final del mes anterior.
157   */
158   static 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 hacer referencia a la primera semana del mes. */
163   public static final int FIRST_WEEK_IN_MONTH = 1;
164
165   /** Una constante útil para hacer referencia a la segunda semana del mes. */
166   public static final int SECOND_WEEK_IN_MONTH = 2;
167
168   /** Una constante útil para hacer referencia a la tercera semana del mes. */
169   public static final int THIRD_WEEK_IN_MONTH = 3;
170
171   /** Una constante útil para hacer referencia a la cuarta semana del mes. */
172   public static final int FOURTH_WEEK_IN_MONTH = 4;
173
174   /** Una constante útil para hacer referencia a la última semana del mes. */
175   public static final int LAST_WEEK_IN_MONTH = 0;
```

```
176
177   /** Constante de intervalo. */
178   public static final int INCLUDE_NONE = 0;
179
180   /** Constante de intervalo. */
181   public static final int INCLUDE_FIRST = 1;
182
183   /** Constante de intervalo. */
184   public static final int INCLUDE_SECOND = 2;
185
186   /** Constante de intervalo. */
187   public static final int INCLUDE_BOTH = 3;
188
189   /**
190   * Constante útil para especificar un día de la semana con respecto a una fecha
191   * fija.
192   */
193   public static final int PRECEDING = -1;
194
195   /**
196   * Constante útil para especificar un día de la semana con respecto a una fecha
197   * fija.
198   */
199   public static final int NEAREST = 0;
200
201   /**
202   * Constante útil para especificar un día de la semana con respecto a una fecha
203   * fija.
204   */
205   public static final int FOLLOWING = 1;
206
207   /** Una descripción para la fecha. */
208   private String description;
209
210   /**
211   * Constructor predeterminado.
212   */
213   protected 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 code el código del que se comprueba la validez.
221   *
222   * @return <code>true</code> si el código entero proporcionado representa un
223   * día de la semana válido y <code>false</code> en caso contrario.
224   */
225   public static boolean isValidWeekdayCode(final int code) {
227   switch(code) {
228   case SUNDAY:
229   case MONDAY:
230   case TUESDAY:
231   case WEDNESDAY:
232   case THURSDAY:
233   case FRIDAY:
234   case SATURDAY:
235   return true;
236   default:
237   return false;
238   }
239
240   }
241
242   /**
```

```
243   * Convierte la cadena proporcionada en un día de la semana.
244   *
245   * @param s una cadena que representa el día de la semana.
246   *
247   * @return <code>-1</code> si la cadena no se puede convertir o el día de
248   * la semana en caso contrario.
249   */
250   public static int stringToWeekdayCode(String s) {
251
252   final String[] shortWeekdayNames
253   = DATE_FORMAT_SYMBOLS.getShortWeekdays();
254   final String[] weekDayNames = DATE_FORMAT_SYMBOLS.getWeekdays();
255
256   int result = -1;
257   s = s.trim();
258   for (int i = 0; i < weekDayNames.length; i++) {
259   if (s.equals(shortWeekdayNames[i])) {
260   result = i;
261   break;
262   }
263   if (s.equals(weekDayNames[i])) {
264   result = i;
265   break;
266   }
267   }
268   return result;
269
270   }
271
272   /**
273   * Devuelve una representación en cadena del día de la semana proporcionado.
274   * <P>
275   * Necesitamos 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 weekday) {
282
283   final String[] weekdays = DATE_FORMAT_SYMBOLS.getWeekdays();
284   return weekdays[weekday];
285
286   }
287
288   /**
289   * Devuelve una matriz de nombres de mes.
290   *
291   * @return una matriz de nombres de mes.
292   */
293   public static String[] getMonths() {
294
295   return getMonths(false);
296
297   }
298
299   /**
300   * Devuelve una matriz de nombres de mes.
301   *
302   * @param shortened un indicador para indicar que deben devolverse los nombres
303   * de mes en formato reducido.
304   *
305   * @return una matriz de nombres de mes.
306   */
307   public static String[] getMonths(final boolean shortened) {
308
```

```
309   if (shortened) {
310   return DATE_FORMAT_SYMBOLS.getShortMonths();
311   }
312   else {
313   return DATE_FORMAT_SYMBOLS.getMonths();
314   }
315
316   }
317
318   /**
319   * Devuelve true si el código entero proporcionado representa un mes válido.
320   *
321   * @param code el código del que se comprueba la validez.
322   *
323   * return <code>true</code> si el código entero proporcionado representa un
324   * mes válido.
325   */
326   public static boolean isValidMonthCode(final int code) {
327
328   switch(code) {
329   case JANUARY:
330   case FEBRUARY:
331   case MARCH:
332   case APRIL:
333   case MAY:
334   case JUNE:
335   case JULY:
336   case AUGUST:
337   case SEPTEMBER:
338   case OCTOBER:
339   case NOVEMBER:
340   case DECEMBER:
341   return true;
342   default:
343   return false;
344   }
345
346   }
347
348   /**
349   * Devuelve el trimestre del mes especificado.
350   *
351   * @param code 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(final int code) {
357
358   switch(code) {
359   case JANUARY:
360   case FEBRUARY:
361   case MARCH: return 1;
362   case APRIL:
363   case MAY:
364   case JUNE: return 2;
365   case JULY:
366   case AUGUST:
367   case SEPTEMBER: return 3;
368   case OCTOBER:
369   case NOVEMBER:
370   case DECEMBER: return 4;
371   default: throw new IllegalArgumentException(
372   “SerialDate.monthCodeToQuarter: invalid month code.”);
373   }
374
```

```
375   }
376
377   /**
378   * Devuelve una cadena que representa el mes proporcionado.
379   * <P>
380   * La cadena devuelta es la forma extensa del nombre del mes obtenido de la
381   * configuración regional.
382   *
383   * @param month el mes.
384   *
385   * @return una cadena que representa el mes proporcionado
386   */
387   public static String monthCodeToString(final int month) {
388
389   return monthCodeToString(month, false);
390
391   }
392
393   /**
394   * Devuelve una cadena que representa el mes proporcionado.
395   * <P>
396   * La cadena devuelta es la forma extensa o reducida del nombre del mes
397   * obtenido de la configuración regional.
398   *
399   * @param month el mes.
400   * @param shortened 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(final int month,
407   final boolean shortened) {
408
409   // comprobar argumentos…
410   if (!isValidMonthCode(month)) {
411   throw new IllegalArgumentException(
412   “SerialDate.monthCodeToString: month outside valid range.”);
413   }
414
415   final String[] months;
416
417   if (shortened) {
418   months = DATE_FORMAT_SYMBOLS.getShortMonths();
419   }
420   else {
421   months = DATE_FORMAT_SYMBOLS.getMonths();
422   }
423
424   return months[month - 1];
425
426   }
427
428   /**
429   * Convierte una cadena en el código del mes.
430   * <P>
431   * Este método devuelve una de las constantes JANUARY, FEBRUARY, …,
432   * DECEMBER correspondientes a la cadena. Si la cadena no se
433   * reconoce, este método devuelve -1.
434   *
435   * @param s la cadena que analizar.
436   *
437   * @return <code>-1</code> si la cadena no se puede analizar, o el mes del
438   * año en caso contrario.
439   */
440   public static int stringToMonthCode(String s) {
```

```
441
442   final String[] shortMonthNames = DATE_FORMAT_SYMBOLS.getShortMonths();
443   final String[] monthNames = DATE_FORMAT_SYMBOLS.getMonths();
444
445   int result = -1;
446   s = s.trim();
447
448   // primero intentar analizar la cadena como entero (1-12)…
449   try {
450   result = Integer.parseInt(s);
451   }
452   catch (NumberFormatException e) {
453   // suprimir
454   }
455
456   // buscar por los nombres de los meses…
457   if ((result < 1) || (result > 12)) {
458   for (int i = 0; i < monthNames.length; i++) {
459   if (s.equals(shortMonthNames[i])) {
460   result = i + 1;
461   break;
462   }
463   if (s.equals(monthNames[i])) {
464   result = i + 1;
465   break;
466   }
467   }
468   }
469
470   return result;
471
472   }
473
474   /**
475   * Devuelve true si el código entero proporcionado representa una semana
476   * del mes válida y false en caso contrario
477   *
478   * @param code el código del que se comprueba la validez.
479   * @return <code>true</code> si el código entero proporcionado representa una
480   * semana del mes válida.
481   */
482   public static boolean isValidWeekInMonthCode(final int code) {
483
484   switch(code) {
485   case FIRST_WEEK_IN_MONTH:
486   case SECOND_WEEK_IN_MONTH:
487   case THIRD_WEEK_IN_MONTH:
488   case FOURTH_WEEK_IN_MONTH:
489   case LAST_WEEK_IN_MONTH: return true;
490   default: return false;
491   }
492
493   }
494
495   /**
496   * Determina si el año especificado es bisiesto o no.
497   *
498   * @param yyyy el año (entre 1900 y 9999).
499   *
500   * @return <code>true</code> si el año especificado es bisiesto.
501   */
502   public static boolean isLeapYear(final int yyyy) {
503
504   if ((yyyy % 4) != 0) {
505   return false;
506   }
```

```
507   else if ((yyyy % 400) == 0) {
508   return true;
509   }
510   else if ((yyyy % 100) == 0) {
511   return false;
512   }
513   else {
514   return true;
515   }
516
517   }
518
519   /**
520   * Devuelve el número de años bisiestos desde 1900 hasta el año especificado
521   * INCLUSIVE.
522   * <P>
523   * 1900 no es un año bisiesto.
524   *
525   * @param yyyy el año (entre 1900 y 9999).
526   *
527   * @return el número de años bisiestos desde 1900 hasta el año especificado.
528   */
529   public static int leapYearCount(final int yyyy) {
530
531   final int leap4 = (yyyy - 1896) / 4;
532   final int leap100 = (yyyy - 1800) / 100;
533   final int leap400 = (yyyy - 1600) / 400;
534   return leap4 - leap100 + leap400;
535
536   }
537
538   /**
539   * Devuelve el número del último día del mes, teniendo en cuenta los
540   * años bisiestos.
541   *
542   * @param month el mes.
543   * @param yyyy el año (entre 1900 y 9999).
544   *
545   * @return el número del último día del mes.
546   */
547   public static int lastDayOfMonth(final int month, final int yyyy) {
548
549   final int result = LAST_DAY_OF_MONTH[month];
550   if (month != FEBRUARY) {
551   return result;
552   }
553   else if (isLeapYear(yyyy)) {
554   return result + 1;
555   }
556   else {
557   return result;
558   }
559
560   }
561
562   /**
563   * Crea una nueva fecha añadiendo el número especificado de días a la fecha
564   * base.
565   *
566   * @param days el número de días que añadir (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() + days;
574   return SerialDate.createInstance(serialDayNumber);
575
576   }
577
578   /**
579   * Crea una nueva fecha añadiendo el número especificado de meses a la fecha
580   * base.
581   * <P>
582   * Si la fecha base es próxima al final del mes, el día del resultado
583   * se puede ajustar ligeramente: 31 Mayo + 1 mes = 30 Junio.
584   *
585   * @param months el número de meses que añadir (puede ser negativo).
586   * @param base la fecha base.
587   *
588   * @return una nueva fecha.
589   */
590   public static SerialDate addMonths(final int months,
591   final SerialDate base) {
592
593   final int yy = (12 * base.getYYYY() + base.getMonth() + months - 1)
594   / 12;
595   final int mm = (12 * base.getYYYY() + base.getMonth() + months – 1)
596   % 12 + 1;
597   final int dd = Math.min(
598   base.getDayOfMonth(), SerialDate.lastDayOfMonth(mm, yy)
599   );
600   return SerialDate.createInstance(dd, mm, yy);
601
602   }
603
604   /**
605   * Crea una nueva fecha añadiendo el número especificado de años a la fecha
606   * base.
607   *
608   * @param years el número de años que añadir (puede ser negativo).
609   * @param base la fecha base.
610   *
611   * @return Una nueva fecha.
612   */
613   public static SerialDate addYears(final int years, final SerialDate base) {
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 + years;
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 correspondiente al día de la semana especificado y
630   * ANTERIOR a la fecha base.
631   *
632   * @param targetWeekday un código para el día de la semana de destino.
633   * @param base la fecha base.
634   *
635   * @return la última fecha correspondiente al día de la semana especificado y
636   * ANTERIOR a la fecha base.
637   */
638   public static SerialDate getPreviousDayOfWeek(final int targetWeekday,
```

```
639   final SerialDate base) {
640
641   // comprobar argumentos…
642   if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
643   throw new IllegalArgumentException(
644   “Invalid day-of-the-week code.”
645   );
646   }
647
648   // buscar la fecha…
649   final int adjust;
650   final int baseDOW = base.getDayOfWeek();
651   if (baseDOW > targetWeekday) {
652   adjust = Math.min(0, targetWeekday - baseDOW);
653   }
654   else {
655   adjust = -7 + Math.max(0, targetWeekday - baseDOW);
656   }
657
658   return SerialDate.addDays(adjust, base);
659
660   }
661
662   /**
663   * Devuelve la primera fecha que coincide con el día de la semana especificado
664   * y POSTERIOR a la fecha base.
665   *
666   * @param targetWeekday un código para el día de la semana de destino.
667   * @param base la fecha base.
668   *
669   * @return la primera fecha que coincide con el día de la semana especificado
670   * y POSTERIOR a la fecha base.
671   */
672   public static SerialDate getFollowingDayOfWeek(final int targetWeekday,
673   final SerialDate base) {
674
675   // comprobar argumentos…
676   if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
677   throw new IllegalArgumentException(
678   “Invalid day-of-the-week code.”
679   );
680   }
681
682   // buscar la fecha…
683   final int adjust;
684   final int baseDOW = base.getDayOfWeek();
685   if (baseDOW > targetWeekday) {
686   adjust = 7 + Math.min(0, targetWeekday - baseDOW);
687   }
688   else {
689   adjust = Math.max(0, targetWeekday - baseDOW);
690   }
691
692   return SerialDate.addDays(adjust, base);
693   }
694
695   /**
696   * Devuelve la fecha que coincide con el día de la semana especificado y más
697   * PRÓXIMA a la fecha base.
698   *
699   * @param targetDOW un código para el día de la semana de destino.
700   * @param base la fecha base.
701   *
702   * @return la fecha que coincide con el día de la semana especificado y más
703   * PRÓXIMA a la fecha base.
704   */
```

```
705   public static SerialDate getNearestDayOfWeek(final int targetDOW,
706   final SerialDate base) {
707
708   // comprobar argumentos…
709   if (!SerialDate.isValidWeekdayCode(targetDOW)) {
710   throw new IllegalArgumentException(
711   “Invalid day-of-the-week code.”
712   );
713   }
714
715   // buscar la fecha…
716   final int baseDOW = base.getDayOfWeek();
717   int adjust = -Math.abs(targetDOW - baseDOW);
718   if (adjust >= 4) {
719   adjust = 7 - adjust;
720   }
721   if (adjust <= -4) {
722   adjust = 7 + adjust;
723   }
724   return SerialDate.addDays(adjust, base);
725
726   }
727
728   /**
729   * Avanzar 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(final SerialDate base) {
736   final int last = SerialDate.lastDayOfMonth(
737   base.getMonth(), base.getYYYY()
738   );
739   return SerialDate.createInstance(last, base.getMonth(), base.getYYYY());
740   }
741
742   /**
743   * Devuelve una cadena correspondiente al código de la semana del mes.
744   * <P>
745   * Necesitamos un enfoque mejor.
746   *
747   * @param count un código entero que representa la semana del mes.
748   *
749   * @return una cadena correspondiente al código de la semana del mes.
750   */
751   public static String weekInMonthToString(final int count) {
752
753   switch (count) {
754   case SerialDate.FIRST_WEEK_IN_MONTH : return “First”;
755   case SerialDate.SECOND_WEEK_IN_MONTH : return “Second”;
756   case SerialDate.THIRD_WEEK_IN_MONTH : return “Third”;
757   case SerialDate.FOURTH_WEEK_IN_MONTH : return “Fourth”;
758   case SerialDate.LAST_WEEK_IN_MONTH : return “Last”;
759   default :
760   return “SerialDate.weekInMonthToString(): invalid code.”;
761   }
762
763   }
764
765   /**
766   * Devuelve una cadena que representa el valor ‘relativo’ proporcionado.
767   * <P>
768   * Necesitamos un enfoque mejor.
769   *
770   * @param relative una constante que representa el valor ‘relativo’.
```

```
771    *
772    * @return una cadena que representa el valor ‘relativo’ proporcionado.
773    */
774    public static String relativeToString(final int relative) {
775
776    switch (relative) {
777    case SerialDate.PRECEDING : return “Preceding”;
778    case SerialDate.NEAREST : return “Nearest”;
779    case SerialDate.FOLLOWING : return “Following”;
780    default : return “ERROR : Relative To String”;
781    }
782
783    }
784
785    /**
786    * Método de factoría que devuelve una instancia de una subclase concreta de
787    * {@link SerialDate}.
788    *
789    * @param day el día (1-31).
790    * @param month el mes (1-12).
791    * @param yyyy el año (entre 1900 y 9999).
792    *
793    * @return Una instancia de {@link SerialDate}
794    */
795    public static SerialDate createInstance(final int day, final int month,
796    final int yyyy) {
797    return new SpreadsheetDate(day, month, yyyy);
798    }
799
800    /**
801    * Método de factoría que devuelve una instancia de una subclase concreta de
802    * {@link SerialDate}.
803    *
804    * @param serial numero 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 factoría que devuelve una instancia de una subclase de SerialDate.
814    *
815    * @param date Un objeto de fecha de Java.
816    *
817    * @return una instancia de SerialDate.
818    */
818    public static SerialDate createInstance(final java.util.Date date) {
820
821    final GregorianCalendar calendar = new GregorianCalendar();
822    calendar.setTime(date);
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, siendo el 1 de enero de 1900 = 2 (se
831    * corresponde, casi totalmente, al sistema de numeración empleado en Microsoft
832    * Excel para Windows y Lotus 1-2-3).
833    *
834    * @return el número de serie de la fecha.
835    */
836    public abstract int toSerial();
```

```
837
838    /**
839    * Devuelve java.util.Date. Como java.util.Date tiene mayor precisión que
840    * SerialDate, debemos definir una convención para “la hora del día”.
841    *
842    * @return this como <code>java.util.Date</code>.
843    */
844    public abstract java.util.Date toDate();
845
846    /**
847    * Devuelve una descripción de la fecha.
848    *
849    * @return una descripción de la fecha.
850    */
851    public String getDescription() {
852    return this.description;
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(final String description) {
861    this.description = description;
862    }
863
864    /**
865    * Convierte la fecha en una cadena.
866    *
867    * @return una representación en cadena de la fecha.
868    */
869    public String toString() {
870    return getDayOfMonth() + “-” + SerialDate.monthCodeToString(getMonth())
871    + “-” + getYYYY();
872    }
873
874    /**
875    * Devuelve el año (con un intervalo 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
904     * 'otra' fecha especificada.
905     * <P>
906     * El resultado es positivo si esta fecha es posterior a la 'otra' y
907     * negativo si es anterior.
908     *
909     * @param other la fecha con la que se compara.
910     *
911     * @return la diferencia entre esta fecha y la otra.
912     */
913    public abstract int compare(SerialDate other);
914
915    /**
916     * Devuelve true si esta SerialDate representa la misma fecha que la
917     * SerialDate especificada.
918     *
919     * @param other la fecha con la que se compara.
920     *
921     * @return <code>true</code> si esta SerialDate representa la misma fecha que
922     * la SerialDate especificada.
923     */
924    public abstract boolean isOn(SerialDate other);
925
926    /**
927     * Devuelve true si esta SerialDate representa una fecha anterior en
928     * comparación a la SerialDate especificada.
929     *
930     * @param other la fecha con la que se compara.
931     *
932     * @return <code>true</code> si esta SerialDate representa una fecha anterior
933     * en comparación a la SerialDate especificada.
934     */
935    public abstract boolean isBefore(SerialDate other);
936
937    /**
938     * Devuelve true si esta SerialDate representa la misma fecha que la
939     * SerialDate especificada.
940     *
941     * @param other la fecha con la que se compara.
942     *
943     * @return <code>true</code> si esta SerialDate representa la misma fecha
944     * que la SerialDate especificada.
945     */
946    public abstract boolean isOnOrBefore(SerialDate other);
947
948    /**
949     * Devuelve true si esta SerialDate representa la misma fecha que la
950     * SerialDate especificada.
951     *
952     * @param other la fecha con la que se compara.
953     *
954     * @return <code>true</code> si esta SerialDate representa la misma fecha
955     * que la SerialDate especificada.
956     */
957    public abstract boolean isAfter(SerialDate other);
958
959    /**
960     * Devuelve true si esta SerialDate representa la misma fecha que la
961     * SerialDate especificada.
962     *
963     * @param other la fecha con la que se compara.
964     *
965     * @return <code>true</code> si esta SerialDate representa la misma fecha
966     * que la SerialDate especificada.
967     */
968    public abstract boolean isOnOrAfter(SerialDate other);
```

```
969
970    /**
971    * Devuelve <code>true</code> si {@link SerialDate} se encuentra en el
972    * rango especificado (INCLUSIVE). El orden de fecha de d1 y d2 no es
973    * importante.
974    *
975    * @param d1 fecha límite del rango.
976    * @param d2 la otra fecha límite del rango.
977    *
978    * @return Un valor booleano.
979    */
980    public abstract boolean isInRange(SerialDate d1, SerialDate d2);
981
982    /**
983    * Devuelve <code>true</code> si {@link SerialDate} se encuentra en el
984    * rango especificado (el invocador especifica si los puntos finales se
985    * incluyen o no). El orden de fecha de d1 y d2 no es importante.
986    *
987    * @param d1 fecha límite del rango.
988    * @param d2 la otra fecha límite del rango.
989    * @param include un código que controla si las fechas inicial y final
990    * se incluyen o no en el rango.
991    *
992    * @return Un valor booleano.
993    */
994    public abstract boolean isInRange(SerialDate d1, SerialDate d2,
995    int include);
996
997    /**
998    * Devuelve la última fecha que coincide con el día de la semana especificado y
999    * que es ANTERIOR a esta fecha.
1000   *
1001   * @param targetDOW un código para el día de la semana de destino.
1002   *
1003   * @return la última fecha que coincide con el día de la semana especificado y
1004   * que es ANTERIOR a esta fecha.
1005   */
1006   public SerialDate getPreviousDayOfWeek(final int targetDOW) {
1007   return getPreviousDayOfWeek(targetDOW, this);
1008   }
1009
1010   /**
1011   * Devuelve la primera fecha que coincide con el día de la semana especificado
1012   * y que es POSTERIOR a esta fecha.
1013   *
1014   * @param targetDOW un código para el día de la semana de destino.
1015   *
1016   * @return la primera fecha que coincide con el día de la semana especificado
1017   * que es POSTERIOR a esta fecha.
1018   */
1019   public SerialDate getFollowingDayOfWeek(final int targetDOW) {
1020   return getFollowingDayOfWeek(targetDOW, this);
1021   }
1022
1023   /**
1024   * Devuelve la fecha más próxima que coincide con el día de la semana especificado.
1025   *
1026   * @param targetDOW un código para el día de la semana de destino.
1027   *
1028   * @return la fecha más próxima que coincide con el día de la semana especificado.
1029   */
1030   public SerialDate getNearestDayOfWeek(final int targetDOW) {
1031   return getNearestDayOfWeek(targetDOW, this);
1032   }
1033
1034   }
```

Listado B-2
SerialDateTest.java

```
/* ===========================================================
* JCommon : biblioteca gratuita de clases de propósito general para Java(tm)
* ===========================================================
*
* (C) Copyright 2000-2005, de Object Refinery Limited y colaboradores.
*
* Información del projecto: http://www.jfree.org/jcommon/index.html
*
* Esta biblioteca es software gratuito; puede distribuirla y/o modificarla
* bajo las condiciones de la Licencia pública general GNU publicada por
* la Free Software Foundation; ya sea la versión 2.1 de la licencia, u
* otra versión posterior (de su elección).
*
* Esta biblioteca se distribuye con la intención de que sea útil, pero
* SIN GARANTÍA ALGUNA, incluida la garantía implícita de COMERCIABILIDAD
* e IDONEIDAD PARA UN DETERMINADO FIN. Consulte la Licencia pública general GNU
* si necesita más información al respecto.
*
* Debería haber recibido una copia de la Licencia pública general GNU
* junto a esta biblioteca; en caso contrario, contacte con la Free Software
* Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
* EE.UU.
*
* [Java es una marca comercial o marca comercial registrada de Sun
* Microsystems, Inc. en Estados Unidos y otros países.]
*
* -------------------------
* SerialDateTests.java
* -------------------------
* (C) Copyright 2001-2005, por Object Refinery Limited.

* Autor original: David Gilbert (por Object Refinery Limited);
* Colaborador(es): -;

* $Id: SerialDateTests.java,v 1.6 2005/11/16 15:58:40 taqua Exp $

* Cambios
* -----------
* 15-Nov-2001: Version 1 (DG);
* 25-Jun-2002: Se elimina la importación innecesaria (DG);
* 24-Oct-2002: Errores indicados Checkstyle corregidos (DG);
* 13-Mar-2003: Se añade prueba de serialización (DG);
* 05-Jan-2005: Se añade prueba para el informe de errores 1096282 (DG);
*
*/

package org.jfree.date.junit;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

import org.jfree.date.MonthConstants;
import org.jfree.date.SerialDate;

```

```
63  /**
64  * Pruebas JUnit para la clase {@link SerialDate}.
65  */
66  public class SerialDateTests extends TestCase {
67
68  /** Fecha que representa 9 de noviembre.
69  private SerialDate nov9Y2001;
70
71  /**
72  * Crea un nuevo caso de prueba.
73  *
74  * @param name el nombre.
75  */
76  public SerialDateTests(final String name) {
77  super(name);
78  }
79
80  /**
81  * Devuelve una suite de pruebas para el ejecutor de pruebas JUnit.
82  *
83  * @return La suite de pruebas.
84  */
85  public static Test suite() {
86  return new TestSuite(SerialDateTests.class);
87  }
88
89  /**
90  * Problema.
91  */
92  protected void setUp() {
93  this.nov9Y2001 = SerialDate.createInstance(9, MonthConstants.NOVEMBER, 2001);
94  }
95
96  /**
97  * 9 Nov 2001 más dos meses debe ser 9 Ene 2002.
98  */
99  public void testAddMonthsTo9Nov2001() {
100 final SerialDate jan9Y2002 = SerialDate.addMonths(2, this.nov9Y2001);
101 final SerialDate answer = SerialDate.createInstance(9, 1, 2002);
102 assertEquals(answer, jan9Y2002);
103 }
104
105 /**
106 * Caso de prueba de un error, ya corregido.
107 */
108 public void testAddMonthsTo5Oct2003() {
109 final SerialDate d1 = SerialDate.createInstance(5, MonthConstants.OCTOBER, 2003);
110 final SerialDate d2 = SerialDate.addMonths(2, d1);
111 assertEquals(d2, SerialDate.createInstance(5, MonthConstants.DECEMBER, 2003));
112 }
113
114 /**
115 * Caso de prueba de un error, ya corregido.
116 */
117 public void testAddMonthsTo1Jan2003() {
118 final SerialDate d1 = SerialDate.createInstance(1, MonthConstants.JANUARY, 2003);
119 final SerialDate d2 = SerialDate.addMonths(0, d1);
120 assertEquals(d2, d1);
121 }
122
123 /**
124 * El lunes anterior al viernes 9 de noviembre de 2001 debe ser el 5 de noviembre.
125 */
126 public void testMondayPrecedingFriday9Nov2001() {
127 SerialDate mondayBefore = SerialDate.getPreviousDayOfWeek(
128 SerialDate.MONDAY, this.nov9Y2001
```

```
);
assertEquals(5, mondayBefore.getDayOfMonth());
}

/**
* El lunes posterior al viernes 9 de noviembre de 2001 debe ser el 12 de noviembre.
*/
public void testMondayFollowingFriday9Nov2001() {
SerialDate mondayAfter = SerialDate.getFollowingDayOfWeek(
SerialDate.MONDAY, this.nov9Y2001
);
assertEquals(12, mondayAfter.getDayOfMonth());
}

/**
* El lunes más próximo al viernes 9 de noviembre de 2001 debe ser el 12 de noviembre.
*/
public void testMondayNearestFriday9Nov2001() {
SerialDate mondayNearest = SerialDate.getNearestDayOfWeek(
SerialDate.MONDAY, this.nov9Y2001
);
assertEquals(12, mondayNearest.getDayOfMonth());
}

/**
* El lunes más próximo al 22 de enero de 1970 cae en el 19.
*/
public void testMondayNearest22Jan1970() {
SerialDate jan22Y1970 = SerialDate.createInstance(22, MonthConstants.JANUARY, 1970);
SerialDate mondayNearest = SerialDate.getNearestDayOfWeek(SerialDate.MONDAY, jan22Y1970);
assertEquals(19, mondayNearest.getDayOfMonth());
}

/**
* El problema es que la conversión de días en cadenas devuelva el resultado
* correcto. En realidad este resultado depende de la configuración regional.
*/
public void testWeekdayCodeToString() {

final String test = SerialDate.weekdayCodeToString(SerialDate.SATURDAY);
assertEquals(“Saturday”, test);

}

/**
* Probar la conversión de una cadena en día de la semana. Esta prueba falla si
* la configuración regional predeterminada no usa nombres de días en inglés
*/
public void testStringToWeekday() {

int weekday = SerialDate.stringToWeekdayCode(“Wednesday”);
assertEquals(SerialDate.WEDNESDAY, weekday);

weekday = SerialDate.stringToWeekdayCode(“ Wednesday ”);
assertEquals(SerialDate.WEDNESDAY, weekday);

weekday = SerialDate.stringToWeekdayCode(“Wed”);
assertEquals(SerialDate.WEDNESDAY, weekday);

}

/**
* Probar la conversión de una cadena en mes. Esta prueba falla si la
* configuración regional predeterminada no usa nombres de días en inglés
*/
public void testStringToMonthCode() {
```

```
195
196  int m = SerialDate.stringToMonthCode(“January”);
197  assertEquals(MonthConstants.JANUARY, m);
198
199  m = SerialDate.stringToMonthCode(“ January ”);
200  assertEquals(MonthConstants.JANUARY, m);
201
202  m = SerialDate.stringToMonthCode(“Jan”);
203  assertEquals(MonthConstants.JANUARY, m);
204
205  }
206
207  /**
208  * Probar la conversión de un código de mes en cadena.
209  */
210  public void testMonthCodeToStringCode() {
211
212  final String test = SerialDate.monthCodeToString(MonthConstants.DECEMBER);
213  assertEquals(“December”, test);
214
215  }
216
217  /**
218  * 1900 no es un año bisiesto.
219  */
220  public void testIsNotLeapYear1900() {
221  assertTrue(!SerialDate.isLeapYear(1900));
222  }
223
224  /**
225  * 2000 es un año bisiesto.
226  */
227  public void testIsLeapYear2000() {
228  assertTrue(SerialDate.isLeapYear(2000));
229  }
230
231  /**
232  * El número de años bisiestos desde 1900 y hasta 1899 incluido es 0.
233  */
234  public void testLeapYearCount1899() {
235  assertEquals(SerialDate.leapYearCount(1899), 0);
236  }
237
238  /**
239  * El número de años bisiestos desde 1900 y hasta 1903 incluido es 0.
240  */
241  public void testLeapYearCount1903() {
242  assertEquals(SerialDate.leapYearCount(1903), 0);
243  }
244
245  /**
246  * El número de años bisiestos desde 1900 y hasta 1904 incluido es 1.
247  */
248  public void testLeapYearCount1904() {
249  assertEquals(SerialDate.leapYearCount(1904), 1);
250  }
251
252  /**
253  * El número de años bisiestos desde 1900 y hasta 1999 incluido es 24.
254  */
255  public void testLeapYearCount1999() {
256  assertEquals(SerialDate.leapYearCount(1999), 24);
257  }
258
259  /**
260  * El número de años bisiestos desde 1900 y hasta 2000 incluido es 25.
```

```
261   */
262   public void testLeapYearCount2000() {
263   assertEquals(SerialDate.leapYearCount(2000), 25);
264   }
265
266   /**
267   * Serializar una instancia, restaurarla y comprobar la igualdad.
268   */
269   public void testSerialization() {
270
271   SerialDate d1 = SerialDate.createInstance(15, 4, 2000);
272   SerialDate d2 = null;
273
274   try {
275   ByteArrayOutputStream buffer = new ByteArrayOutputStream();
276   ObjectOutput out = new ObjectOutputStream(buffer);
277   out.writeObject(d1);
278   out.close();
279
280   ObjectInput in = new ObjectInputStream(
new   ByteArrayInputStream(buffer.toByteArray()));
281   d2 = (SerialDate) in.readObject();
282   in.close();
283   }
284   catch (Exception e) {
285   System.out.println(e.toString());
286   }
287   assertEquals(d1, d2);
288
289   }
290
291   /**
292   * Prueba para el informe de error 1096282 (ya corregido).
293   */
294   public void test1096282() {
295   SerialDate d = SerialDate.createInstance(29, 2, 2004);
296   d = SerialDate.addYears(1, d);
297   SerialDate expected = SerialDate.createInstance(28, 2, 2005);
298   assertTrue(d.isOn(expected));
299   }
300
301   /**
302   * Diversas pruebas para el método addMonths().
303   */
304   public void testAddMonths() {
305   SerialDate d1 = SerialDate.createInstance(31, 5, 2004);
307   SerialDate d2 = SerialDate.addMonths(1, d1);
308   assertEquals(30, d2.getDayOfMonth());
309   assertEquals(6, d2.getMonth());
310   assertEquals(2004, d2.getYYYY());
311
312   SerialDate d3 = SerialDate.addMonths(2, d1);
313   assertEquals(31, d3.getDayOfMonth());
314   assertEquals(7, d3.getMonth());
315   assertEquals(2004, d3.getYYYY());
316
317   SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
318   assertEquals(30, d4.getDayOfMonth());
319   assertEquals(7, d4.getMonth());
320   assertEquals(2004, d4.getYYYY());
321   }
322   }
```

Listado B-3

MonthConstants.java

```
/* ===========================================================
* JCommon : biblioteca gratuita de clases de propósito general para Java(tm)
* ===========================================================
*
* (C) Copyright 2000-2005, de Object Refinery Limited y colaboradores.
*
* Información del proyecto: http://www.jfree.org/jcommon/index.html
*
* Esta biblioteca es software gratuito; puede distribuirla y/o modificarla
* bajo las condiciones de la Licencia pública general GNU publicada por
* la Free Software Foundation; ya sea la versión 2.1 de la licencia, u
* otra versión posterior (de su elección).
*
* Esta biblioteca se distribuye con la intención de que sea útil, pero
* SIN GARANTÍA ALGUNA, incluida la garantía implícita de COMERCIABILIDAD
* e IDONEIDAD PARA UN DETERMINADO FIN. Consulte la Licencia pública general GNU
* si necesita más información al respecto.
*
* Debería haber recibido una copia de la Licencia pública general GNU
* junto a esta biblioteca; en caso contrario, contacte con la Free Software
* Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
* EE.UU.
*
* [Java es una marca comercial o marca comercial registrada de Sun
* Microsystems, Inc. en Estados Unidos y otros países.]
*
* ----------------------
* MonthConstants.java
* ----------------------
* (C) Copyright 2002, 2003, de Object Refinery Limited.
*
* Autor original: David Gilbert (para Object Refinery Limited);
* Colaborador(es): -;
*
* $Id: MonthConstants.java,v 1.4 2005/11/16 15:58:40 taqua Exp $
*
* Cambios
* ----------
* 29-May-2002 : Version 1 (code moved from SerialDate class) (DG);
*
*/

package org.jfree.date;

/**
* Constantes útiles para los meses. NO son equivalentes a las
* constantes definidas por java.util.Calendar (donde JANUARY=0 y DECEMBER=11).
* <P>
* Se usa en las clases SerialDate y RegularTimePeriod.
*
* @author David Gilbert
*/
public interface MonthConstants {

/** Constante para Enero. */
public static final int JANUARY = 1;

/** Constante para Febrero. */
public static final int FEBRUARY = 2;

/** Constante para Marzo. */
public static final int MARCH = 3;

```

```
64  /** Constante para Abril. */
65  public static final int APRIL = 4;
66
67  /** Constante para Mayo. */
68  public static final int MAY = 5;
69
70  /** Constante para Junio. */
71  public static final int JUNE = 6;
72
73  /** Constante para Julio. */
74  public static final int JULY = 7;
75
76  /** Constante para Agosto. */
77  public static final int AUGUST = 8;
78
79  /** Constante para Septiembre. */
80  public static final int SEPTEMBER = 9;
81
82  /** Constante para Octubre. */
83  public static final int OCTOBER = 10;
84
85  /** Constante para Noviembre. */
86  public static final int NOVEMBER = 11;
87
88  /** Constante para Diciembre. */
89  public static final int DECEMBER = 12;
90
91  }
```

Listado B-4
BobsSerialDateTest.java

```
1    package org.jfree.date.junit;
2
3    import junit.framework.TestCase;
4    import org.jfree.date.*;
5    import static org.jfree.date.SerialDate.*;
6
7    import java.util.*;
8
9    public class BobsSerialDateTest extends TestCase {
10
11   public void testIsValidWeekdayCode() throws Exception {
12   for (int day = 1; day <= 7; day++)
13   assertTrue(isValidWeekdayCode(day));
14   assertFalse(isValidWeekdayCode(0));
15   assertFalse(isValidWeekdayCode(8));
16   }
17
18   public void testStringToWeekdayCode() throws Exception {
19
20   assertEquals(-1, stringToWeekdayCode(“Hello”));
21   assertEquals(MONDAY, stringToWeekdayCode(“Monday”));
22   assertEquals(MONDAY, stringToWeekdayCode(“Mon”));
23   //todo assertEquals(MONDAY,stringToWeekdayCode(“monday”));
24   // assertEquals(MONDAY,stringToWeekdayCode(“MONDAY”));
25   // assertEquals(MONDAY, stringToWeekdayCode(“mon”));
26
27   assertEquals(TUESDAY, stringToWeekdayCode(“Tuesday”));
28   assertEquals(TUESDAY, stringToWeekdayCode(“Tue”));
29   // assertEquals(TUESDAY,stringToWeekdayCode(“tuesday”));
30   // assertEquals(TUESDAY,StringToWeekdayCode(“TUESDAY”));
31   // assertEquals(TUESDAY, stringToWeekdayCode(“tue”));
```

```
// assertEquals(TUESDAY, stringToWeekdayCode(“tues”));

assertEquals(WEDNESDAY, stringToWeekdayCode (“Wednesday”));
assertEquals(WEDNESDAY, stringToWeekdayCode(“Wed”));
// assertEquals(WEDNESDAY,stringToWeekdayCode(“wednesday”));
// assertEquals(WEDNESDAY,stringToWeekdayCode(“WEDNESDAY”));
// assertEquals(WEDNESDAY, StringToWeekdayCode(“wed”));

assertEquals(THURSDAY, stringToWeekdayCode(“Thursday”));
assertEquals(THURSDAY, stringToWeekdayCode(“Thu”));
// assertEquals(THURSDAY,stringToWeekdayCode(“thursday”));
// assertEquals(THURSDAY,stringToWeekdayCode(“THURSDAY”));
// assertEquals(THURSDAY, stringToWeekdayCode(“thu”));
// assertEquals(THURSDAY, stringToWeekdayCode(“thurs”));

assertEquals(FRIDAY, stringToWeekdayCode(“Friday”));
assertEquals(FRIDAY, stringToWeekdayCode(“Fri”));
// assertEquals(FRIDAY,stringToWeekdayCode(“friday”));
// assertEquals(FRIDAY,stringToWeekdayCode(“FRIDAY”));
// assertEquals(FRIDAY, StringToWeekdayCode(“fri”));

assertEquals(SATURDAY, stringToWeekdayCode(“Saturday”));
assertEquals(SATURDAY, stringToWeekdayCode(“Sat”));
// assertEquals(SATURDAY,stringToWeekdayCode(“saturday”));
// assertEquals(SATURDAY,stringToWeekdayCode(“SATURDAY”));
// assertEquals(SATURDAY, stringToWeekdayCode(“sat”));

assertEquals(SUNDAY, stringToWeekdayCode(“Sunday”));
assertEquals(SUNDAY, stringToWeekdayCode(“Sun”));
// assertEquals(SUNDAY,stringToWeekdayCode(“sunday”));
// assertEquals(SUNDAY,stringToWeekdayCode(“SUNDAY”));
// assertEquals(SUNDAY, stringToWeekdayCode(“sun”));
}

public void testWeekdayCodeToString() throws Exception {
assertEquals(“Sunday”, weekdayCodeToString(SUNDAY));
assertEquals(“Monday”, weekdayCodeToString(MONDAY));
assertEquals(“Tuesday”, weekdayCodeToString(TUESDAY));
assertEquals(“Wednesday”, weekdayCodeToString(WEDNESDAY));
assertEquals(“Thursday”, weekdayCodeToString(THURSDAY));
assertEquals(“Friday”, weekdayCodeToString(FRIDAY));
assertEquals(“Saturday”, weekdayCodeToString(SATURDAY));
}

public void testIsValidMonthCode() throws Exception {
for (int i = 1; i <= 12; i++)
assertTrue(isValidMonthCode(i));
assertFalse(isValidMonthCode(0));
assertFalse(isValidMonthCode(13));
}

public void testMonthToQuarter() throws Exception {
assertEquals(1, monthCodeToQuarter(JANUARY));
assertEquals(1, monthCodeToQuarter(FEBRUARY));
assertEquals(1, monthCodeToQuarter(MARCH));
assertEquals(2, monthCodeToQuarter(APRIL));
assertEquals(2, monthCodeToQuarter(MAY));
assertEquals(2, monthCodeToQuarter(JUNE));
assertEquals(3, monthCodeToQuarter(JULY));
assertEquals(3, monthCodeToQuarter(AUGUST));
assertEquals(3, monthCodeToQuarter(SEPTEMBER));
assertEquals(4, monthCodeToQuarter(OCTOBER));
assertEquals(4, monthCodeToQuarter(NOVEMBER));
assertEquals(4, monthCodeToQuarter(DECEMBER));

try {
```

```
monthCodeToQuarter(-1);
fail(“Invalid Month Code should throw exception”);
} catch (IllegalArgumentException e) {
}
}

public void testMonthCodeToString() throws Exception {
assertEquals(“January”, monthCodeToString(JANUARY));
assertEquals(“February”, monthCodeToString(FEBRUARY));
assertEquals(“March”, monthCodeToString(MARCH));
assertEquals(“April”, monthCodeToString(APRIL));
assertEquals(“May”, monthCodeToString(MAY));
assertEquals(“June”, monthCodeToString(JUNE));
assertEquals(“July”, monthCodeToString(JULY));
assertEquals(“August”, monthCodeToString(AUGUST));
assertEquals(“September”, monthCodeToString(SEPTEMBER));
assertEquals(“October”, monthCodeToString(OCTOBER));
assertEquals(“November”, monthCodeToString(NOVEMBER));
assertEquals(“December”, monthCodeToString(DECEMBER));

assertEquals(“Jan”, monthCodeToString(JANUARY, true));
assertEquals(“Feb”, monthCodeToString(FEBRUARY, true));
assertEquals(“Mar”, monthCodeToString(MARCH, true));
assertEquals(“Apr”, monthCodeToString(APRIL, true));
assertEquals(“May”, monthCodeToString(MAY, true));
assertEquals(“Jun”, monthCodeToString(JUNE, true));
assertEquals(“Jul”, monthCodeToString(JULY, true));
assertEquals(“Aug”, monthCodeToString(AUGUST, true));
assertEquals(“Sep”, monthCodeToString(SEPTEMBER, true));
assertEquals(“Oct”, monthCodeToString(OCTOBER, true));
assertEquals(“Nov”, monthCodeToString(NOVEMBER, true));
assertEquals(“Dec”, monthCodeToString(DECEMBER, true));

try {
monthCodeToString(-1);
fail(“Invalid month code should throw exception”);
} catch (IllegalArgumentException e) {
}

}

public void testStringToMonthCode() throws Exception {
assertEquals(JANUARY,stringToMonthCode(“1”));
assertEquals(FEBRUARY,stringToMonthCode(“2”));
assertEquals(MARCH,stringToMonthCode(“3”));
assertEquals(APRIL,stringToMonthCode(“4”));
assertEquals(MAY,stringToMonthCode(“5”));
assertEquals(JUNE,stringToMonthCode(“6”));
assertEquals(JULY,stringToMonthCode(“7”));
assertEquals(AUGUST,stringToMonthCode(“8”));
assertEquals(SEPTEMBER,stringToMonthCode(“9”));
assertEquals(OCTOBER,stringToMonthCode(“10”));
assertEquals(NOVEMBER, stringToMonthCode(“11”));
assertEquals(DECEMBER,stringToMonthCode(“12”));

//todo assertEquals(-1, stringToMonthCode(“0”));
// assertEquals(-1, stringToMonthCode(“13”));

assertEquals(-1,stringToMonthCode(“Hello”));

for (int m = 1; m <= 12; m++) {
assertEquals(m, stringToMonthCode(monthCodeToString(m, false)));
assertEquals(m, stringToMonthCode(monthCodeToString(m, true)));
}

// assertEquals(1,stringToMonthCode(“jan”));
```

```
// assertEquals(2,stringToMonthCode(“feb”));
// assertEquals(3,stringToMonthCode(“mar”));
// assertEquals(4,stringToMonthCode(“apr”));
// assertEquals(5,stringToMonthCode(“may”));
// assertEquals(6,stringToMonthCode(“jun”));
// assertEquals(7,stringToMonthCode(“jul”));
// assertEquals(8,stringToMonthCode(“aug”));
// assertEquals(9,stringToMonthCode(“sep”));
// assertEquals(10,stringToMonthCode(“oct”));
// assertEquals(11,stringToMonthCode(“nov”));
// assertEquals(12,stringToMonthCode(“dec”));

// assertEquals(1,stringToMonthCode(“JAN”));
// assertEquals(2,stringToMonthCode(“FEB”));
// assertEquals(3,stringToMonthCode(“MAR”));
// assertEquals(4,stringToMonthCode(“APR”));
// assertEquals(5,stringToMonthCode(“MAY”));
// assertEquals(6,stringToMonthCode(“JUN”));
// assertEquals(7,stringToMonthCode(“JUL”));
// assertEquals(8,stringToMonthCode(“AUG”));
// assertEquals(9,stringToMonthCode(“SEP”));
// assertEquals(10,stringToMonthCode(“OCT”));
// assertEquals(11,stringToMonthCode(“NOV”));
// assertEquals(12,stringToMonthCode(“DEC”));

// assertEquals(1,stringToMonthCode(“january”));
// assertEquals(2,stringToMonthCode(“february”));
// assertEquals(3,stringToMonthCode(“march”));
// assertEquals(4,stringToMonthCode(“april”));
// assertEquals(5,stringToMonthCode(“may”));
// assertEquals(6,stringToMonthCode(“june”));
// assertEquals(7,stringToMonthCode(“july”));
// assertEquals(8,stringToMonthCode(“august”));
// assertEquals(9,stringToMonthCode(“september”));
// assertEquals(10,stringToMonthCode(“october”));
// assertEquals(11,stringToMonthCode(“november”));
// assertEquals(12,stringToMonthCode(“december”));

// assertEquals(1,stringToMonthCode(“JANUARY”));
// assertEquals(2,stringToMonthCode(“FEBRUARY”));
// assertEquals(3,stringToMonthCode(“MAR”));
// assertEquals(4,stringToMonthCode(“APRIL”));
// assertEquals(5,stringToMonthCode(“MAY”));
// assertEquals(6,stringToMonthCode(“JUNE”));
// assertEquals(7,stringToMonthCode(“JULY”));
// assertEquals(8,stringToMonthCode(“AUGUST”));
// assertEquals(9,stringToMonthCode(“SEPTEMBER”));
// assertEquals(10,stringToMonthCode(“OCTOBER”));
// assertEquals(11,stringToMonthCode(“NOVEMBER”));
// assertEquals(12,stringToMonthCode(“DECEMBER”));
}

public void testIsValidWeekInMonthCode() throws Exception {
for (int w = 0; w <= 4; w++) {
assertTrue(isValidWeekInMonthCode(w));
}
assertFalse(isValidWeekInMonthCode(5));
}

public void testIsLeapYear() throws Exception {
assertFalse(isLeapYear(1900));
assertFalse(isLeapYear(1901));
assertFalse(isLeapYear(1902));
assertFalse(isLeapYear(1903));
assertTrue(isLeapYear(1904));
assertTrue(isLeapYear(1908));
```

```
230   assertFalse(isLeapYear(1955));
231   assertTrue(isLeapYear(1964));
232   assertTrue(isLeapYear(1980));
233   assertTrue(isLeapYear(2000));
234   assertFalse(isLeapYear(2001));
235   assertFalse(isLeapYear(2100));
236   }
237
238   public void testLeapYearCount() throws 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));
252   assertEquals(97, leapYearCount(2301));
253   assertEquals(122, leapYearCount(2401));
254   }
255
256   public void testLastDayOfMonth() throws Exception {
257   assertEquals(31, lastDayOfMonth(JANUARY, 1901));
258   assertEquals(28, lastDayOfMonth(FEBRUARY, 1901));
259   assertEquals(31, lastDayOfMonth(MARCH, 1901));
260   assertEquals(30, lastDayOfMonth(APRIL, 1901));
261   assertEquals(31, lastDayOfMonth(MAY, 1901));
262   assertEquals(30, lastDayOfMonth(JUNE, 1901));
263   assertEquals(31, lastDayOfMonth(JULY, 1901));
264   assertEquals(31, lastDayOfMonth(AUGUST, 1901));
265   assertEquals(30, lastDayOfMonth(SEPTEMBER, 1901));
266   assertEquals(31, lastDayOfMonth(OCTOBER, 1901));
267   assertEquals(30, lastDayOfMonth(NOVEMBER, 1901));
268   assertEquals(31, lastDayOfMonth(DECEMBER, 1901));
269   assertEquals(29, lastDayOfMonth(FEBRUARY, 1904));
270   }
271
272   public void testAddDays() throws Exception {
273   SerialDate newYears = d(1, JANUARY, 1900);
274   assertEquals(d(2, JANUARY, 1900), addDays(1, newYears));
275   assertEquals(d(1, FEBRUARY, 1900), addDays(31, newYears));
276   assertEquals(d(1, JANUARY, 1901), addDays(365, newYears));
277   assertEquals(d(31, DECEMBER, 1904), addDays(5 * 365, newYears));
278   }
279
280   private static SpreadsheetDate d(int day, int month, int year)
      { return new SpreadsheetDate(day, month, year); }
281
282   public void testAddMonths() throws Exception {
283   assertEquals(d(1, FEBRUARY, 1900), addMonths(1, d(1, JANUARY, 1900)));
284   assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(31, JANUARY, 1900)));
285   assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(30, JANUARY, 1900)));
286   assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(29, JANUARY, 1900)));
287   assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(28, JANUARY, 1900)));
288   assertEquals(d(27, FEBRUARY, 1900), addMonths(1, d(27, JANUARY, 1900)));
289
290   assertEquals(d(30, JUNE, 1900), addMonths(5, d(31, JANUARY, 1900)));
291   assertEquals(d(30, JUNE, 1901), addMonths(17, d(31, JANUARY, 1900)));
292
293   assertEquals(d(29, FEBRUARY, 1904), addMonths(49, d(31, JANUARY, 1900)));
294
```

```
}

public void testAddYears() throws Exception {
assertEquals(d(1, JANUARY, 1901), addYears(1, d(1, JANUARY, 1900)));
assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(29, FEBRUARY, 1904)));
assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(28, FEBRUARY, 1904)));
assertEquals(d(28, FEBRUARY, 1904), addYears(1, d(28, FEBRUARY, 1903)));
}

public void testGetPreviousDayOfWeek() throws Exception {
assertEquals(d(24, FEBRUARY, 2006), getPreviousDayOfWeek(FRIDAY, d(1, MARCH, 2006)));
assertEquals(d(22, FEBRUARY, 2006), getPreviousDayOfWeek(WEDNESDAY, d(1, MARCH, 2006)));
assertEquals(d(29, FEBRUARY, 2004), getPreviousDayOfWeek(SUNDAY, d(3, MARCH, 2004)));
assertEquals(d(29, DECEMBER, 2004), getPreviousDayOfWeek(WEDNESDAY, d(5, JANUARY, 2005)));

try {
getPreviousDayOfWeek(-1, d(1, JANUARY, 2006));
fail(“Invalid day of week code should throw exception”);
} catch (IllegalArgumentException e) {
}
}

public void testGetFollowingDayOfWeek() throws Exception {
// assertEquals(d(1, JANUARY, 2005),getFollowingDayOfWeek(SATURDAY, d(25, DECEMBER, 2004)));
assertEquals(d(1, JANUARY, 2005), getFollowingDayOfWeek(SATURDAY, d(26, DECEMBER, 2004)));
assertEquals(d(3, MARCH, 2004), getFollowingDayOfWeek(WEDNESDAY, d(28, FEBRUARY, 2004)));

try {
getFollowingDayOfWeek(-1, d(1, JANUARY, 2006));
fail(“Invalid day of week code should throw exception”);
} catch (IllegalArgumentException e) {
}
}

public void testGetNearestDayOfWeek() throws Exception {
assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(16, APRIL, 2006)));
assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(17, APRIL, 2006)));
assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(18, APRIL, 2006)));
assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(19, APRIL, 2006)));
assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(20, APRIL, 2006)));
assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(21, APRIL, 2006)));
assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(22, APRIL, 2006)));

//todo assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(16, APRIL, 2006)));
assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(17, APRIL, 2006)));
assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(18, APRIL, 2006)));
assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(19, APRIL, 2006)));
assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(20, APRIL, 2006)));
assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(21, APRIL, 2006)));
assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(22, APRIL, 2006)));

// assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(16, APRIL, 2006)));
// assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(17, APRIL, 2006)));
assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(18, APRIL, 2006)));
assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(19, APRIL, 2006)));
assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(20, APRIL, 2006)));
assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(21, APRIL, 2006)));
assertEquals(d(25, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(22, APRIL, 2006)));

// assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(16, APRIL, 2006)));
// assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(17, APRIL, 2006)));
// assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(18, APRIL, 2006)));
assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(19, APRIL, 2006)));
assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(20, APRIL, 2006)));
assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(21, APRIL, 2006)));
assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(22, APRIL, 2006)));
```

```
361
362  // assertEquals(d(13, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(16, APRIL, 2006)));
363  // assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(17, APRIL, 2006)));
364  // assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(18, APRIL, 2006)));
365  // assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(19, APRIL, 2006)));
366  assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(20, APRIL, 2006)));
367  assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(21, APRIL, 2006)));
368  assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(22, APRIL, 2006)));
369
370  // assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(16, APRIL, 2006)));
371  // assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(17, APRIL, 2006)));
372  // assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(18, APRIL, 2006)));
373  // assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(19, APRIL, 2006)));
374  // assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(20, APRIL, 2006)));
375  assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(21, APRIL, 2006)));
376  assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(22, APRIL, 2006)));
377
378  // assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(16, APRIL, 2006)));
379  // assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(17, APRIL, 2006)));
380  // assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(18, APRIL, 2006)));
381  // assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(19, APRIL, 2006)));
382  // assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(20, APRIL, 2006)));
383  // assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(21, APRIL, 2006)));
384  assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(22, APRIL, 2006)));
385
386  try {
387  getNearestDayOfWeek(-1, d(1, JANUARY, 2006));
388  fail(“Invalid day of week code should throw exception”);
389  } catch (IllegalArgumentException e) {
390  }
391  }
392
393  public void testEndOfCurrentMonth() throws Exception {
394  SerialDate d = SerialDate.createInstance(2);
395  assertEquals(d(31, JANUARY, 2006), d.getEndOfCurrentMonth(d(1, JANUARY, 2006)));
396  assertEquals(d(28, FEBRUARY, 2006), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2006)));
397  assertEquals(d(31, MARCH, 2006), d.getEndOfCurrentMonth(d(1, MARCH, 2006)));
398  assertEquals(d(30, APRIL, 2006), d.getEndOfCurrentMonth(d(1, APRIL, 2006)));
399  assertEquals(d(31, MAY, 2006), d.getEndOfCurrentMonth(d(1, MAY, 2006)));
400  assertEquals(d(30, JUNE, 2006), d.getEndOfCurrentMonth(d(1, JUNE, 2006)));
401  assertEquals(d(31, JULY, 2006), d.getEndOfCurrentMonth(d(1, JULY, 2006)));
402  assertEquals(d(31, AUGUST, 2006), d.getEndOfCurrentMonth(d(1, AUGUST, 2006)));
403  assertEquals(d(30, SEPTEMBER, 2006), d.getEndOfCurrentMonth(d(1, SEPTEMBER, 2006)));
404  assertEquals(d(31, OCTOBER, 2006), d.getEndOfCurrentMonth(d(1, OCTOBER, 2006)));
405  assertEquals(d(30, NOVEMBER, 2006), d.getEndOfCurrentMonth(d(1, NOVEMBER, 2006)));
406  assertEquals(d(31, DECEMBER, 2006), d.getEndOfCurrentMonth(d(1, DECEMBER, 2006)));
407  assertEquals(d(29, FEBRUARY, 2008), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2008)));
408  }
409
410  public void testWeekInMonthToString() throws Exception {
411  assertEquals(“First”,weekInMonthToString(FIRST_WEEK_IN_MONTH));
412  assertEquals(“Second”,weekInMonthToString(SECOND_WEEK_IN_MONTH));
413  assertEquals(“Third”,weekInMonthToString(THIRD_WEEK_IN_MONTH));
414  assertEquals(“Fourth”,weekInMonthToString(FOURTH_WEEK_IN_MONTH));
415  assertEquals(“Last”,weekInMonthToString(LAST_WEEK_IN_MONTH));
416
417  //todo try {
418  // weekInMonthToString(-1);
419  // fail(“Invalid week code should throw exception”);
420  // } catch (IllegalArgumentException e) {
421  // }
422  }
423
424  public void testRelativeToString() throws Exception {
425  assertEquals(“Preceding”,relativeToString(PRECEDING));
426  assertEquals(“Nearest”,relativeToString(NEAREST));
```

```
427  assertEquals(“Following”,relativeToString(FOLLOWING));
428
429  //todo try {
430  // relativeToString(-1000);
431  // fail(“Invalid relative code should throw exception”);
432  // } catch (IllegalArgumentException e) {
433  // }
434  }
435
436  public void testCreateInstanceFromDDMMYYYY() throws Exception {
437  SerialDate date = createInstance(1, JANUARY, 1900);
438  assertEquals(1,date.getDayOfMonth());
439  assertEquals(JANUARY,date.getMonth());
440  assertEquals(1900,date.getYYYY());
441  assertEquals(2,date.toSerial());
442  }
443
444  public void testCreateInstanceFromSerial() throws Exception {
445  assertEquals(d(1, JANUARY, 1900),createInstance(2));
446  assertEquals(d(1, JANUARY, 1901), createInstance(367));
447  }
448
449  public void testCreateInstanceFromJavaDate() throws Exception {
450  assertEquals(d(1, JANUARY, 1900),
     createInstance(new GregorianCalendar(1900,0,1).getTime()));
451  assertEquals(d(1, JANUARY, 2006),
     createInstance(new GregorianCalendar(2006,0,1).getTime()));
452  }
453
454  public static void main(String[] args) {
455  junit.textui.TestRunner.run(BobsSerialDateTest.class);
456  }
457  }
```

Listado B-5
SpreadsheetDate.java.

```
/* ===========================================================
* JCommon: biblioteca gratuita de clases de propósito general para Java(tm)
* ===========================================================
*
* (C) Copyright 2000-2005, de Object Refinery Limited y colaboradores.
*
* Información del proyecto: http://www.jfree.org/jcommon/index.html
*
* Esta biblioteca es software gratuito; puede distribuirla y/o modificarla
* bajo las condiciones de la Licencia pública general GNU publicada por
* la Free Software Foundation; ya sea la versión 2.1 de la licencia, u
* otra versión posterior (de su elección).
*
* Esta biblioteca se distribuye con la intención de que sea útil, pero
* SIN GARANTÍA ALGUNA, incluida la garantía implícita de COMERCIABILIDAD
* e IDONEIDAD PARA UN DETERMINADO FIN. Consulte la Licencia pública general GNU
* si necesita más información al respecto.
*
* Debería haber recibido una copia de la Licencia pública general GNU
* junto a esta biblioteca; en caso contrario, contacte con la Free Software
* Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
* EE.UU.
*
* [Java es una marca comercial o marca comercial registrada de Sun
* Microsystems, Inc. en Estados Unidos y otros países.]
*
* --------------------------
* SpreadsheetDate.java
* --------------------------
* (C) Copyright 2000-2005, de Object Refinery Limited y colaboradores.
*
* Autor original: David Gilbert (por Object Refinery Limited);
* Colaboradores(s): -;
*
* $Id: SpeadsheetDate.java,v 1.8 2005/11/03 09:25:39 mungady Exp $
*
* Cambios
* ----------
* 11-Oct-2001 : Version 1 (DG);
* 05-Nov-2001 : Se añaden los métodos getDescription() y setDescription() (DG);
* 12-Nov-2001 : Se cambia el nombre ExcelDate.java por SpreadsheetDate.java (DG);
* Se corrige un error a la hora de calcular el día, mes y año a
* partir del número de serie (DG);
* 24-Jan-2002 : Se corrige un error a la hora de calcular el número de serie a
* partir del día, mes y año. Gracias a Trevor Kills por el informe (DG);
* 29-May-2002 : Se añade el método equals(Object) (SourceForge ID 558850) (DG);
* 03-Oct-2002 : Se corrigen los errores detectados por Checkstyle (DG);
* 13-Mar-2003 : Implementación de Serializable (DG);
* 04-Sep-2003 : Métodos isInRange() completados (DG);
* 05-Sep-2003 : Implementación de Comparable (DG);
* 21-Oct-2003 : Se añade el método hashCode() (DG);
*
/*

package org.jfree.date;

import java.util.Calendar;
import java.util.Date;
/**
```

```
60
61    * Representa una con un entero, de forma similar a la
62    * implementación en Microsoft Excel. El intervalo de fechas admitido es
63    * 1-Ene-1900 a 31-Dic-9999.
64    * <P>
65    * Recuerde que Excel tiene un error que reconoce el año
66    * 1900 como bisiesto cuando en realidad no lo es. Encontrará 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 como convención que el 1-Ene-1900 = 1. Esta clase usa la
72    * convención de que 1-Ene-1900 = 2.
73    * Como resultado, el número de día de esta clase será diferente al de
74    * Excel para enero y febrero de 1900… pero Excel añade un día
75    * más (29-Feb-1900 que en realidad no existe) y a partir de ahí
76    * los números de los días coinciden.
77    *
78    * @author David Gilbert
79    */
80    public class SpreadsheetDate extends 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    private int serial;
90
91    /** El día del mes (de 1 a 28, 29, 30 o 31 en función del mes). */
92    private int day;
93
94    /** El mes del año (de 1 a 12). */
95    private int month;
96
97    /** El año (de 1900 a 9999). */
98    private int year;
99
100   /** Una descripción opcional para la fecha. */
101   private String description;
102
103   /**
104   * Crear una nueva instancia de la fecha.
105   *
106   * @param day el día (entre 1 y 28/29/30/31).
107   * @param month el mes (entre 1 y 12).
108   * @param year el año (entre 1900 y 9999).
109   */
110   public SpreadsheetDate(final int day, final int month, final int year) {
111
112   if ((year >= 1900) && (year <= 9999)) {
113   this.year = year;
114   }
115   else {
116   throw new IllegalArgumentException(
117   “The ‘year’ argument must be in range 1900 to 9999.”
118   );
119   }
120
121   if ((month >= MonthConstants.JANUARY)
122   && (month <= MonthConstants.DECEMBER)) {
123   this.month = month;
124   }
125   else {
```

```
126  throw new IllegalArgumentException(
127  “The ‘month’ argument must be in the range 1 to 12.”
128  );
129  }
130
131  if ((day >= 1) && (day <= SerialDate.lastDayOfMonth(month, year))) {
132  this.day = day;
133  }
134  else {
135  throw new IllegalArgumentException(“Invalid ‘day’ argument.”);
136  }
137
138  // es necesario sincronizar el número de serie con el día-mes-año…
139  this.serial = calcSerial(day, month, year);
140
141  this.description = null;
142
143  }
144
145  /**
146  * Constructor estándar: crear un nuevo objeto de fecha que representa el
147  * número de día especificado (que debe estar comprendido entre 2 y 2958465).
148  *
149  * @param serial número de serie para el día (entre 2 y 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  else {
157  throw new IllegalArgumentException(
158  “SpreadsheetDate: Serial must be in range 2 to 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 adjuntada a la fecha. No es
168  * obligatorio que la fecha tenga una descripción, pero resulta útil
169  * en algunas aplicaciones.
170  *
171  * @return La descripción adjuntada a la fecha.
172  */
173  public String getDescription() {
174  return this.description;
175  }
176
177  /**
178  * Establece la descripción de la fecha.
179  *
180  * @param description la descripción de esta fecha (<code>null</code>
181  * se permite)
182  */
183  public void setDescription(final String description) {
184  this.description = description;
185  }
186
187  /**
188  * Devuelve el número de serie de la fecha, siendo el 1 de enero 1900 = 2
189  * (se corresponde, casi totalmente, al sistema de numeración empleado en
190  * Microsoft Excel para Windows y Lotus 1-2-3).
191  *
```

```
* @return El número de serie de la fecha.
*/
public int toSerial() {
return this.serial;
}

/**
* Devuelve una <code>java.util.Date</code> equivalente a esta fecha.
*
* @return La fecha.
*/
public Date toDate() {
final Calendar calendar = Calendar.getInstance();
calendar.set(getYYYY(), getMonth() - 1, getDayOfMonth(), 0, 0, 0);
return calendar.getTime();
}

/**
* Devuelve el año (con un intervalo válido de 1900 a 9999).
*
* @return El año.
*/
public int getYYYY() {
return this.year;
}

/**
* Devuelve el mes (Enero = 1, Febrero = 2, Marzo = 3).
*
* @return El mes del año.
*/
public int getMonth() {
return this.month;
}

/**
* Devuelve el día del mes.
*
* @return El día del mes.
*/
public int getDayOfMonth() {
return this.day;
}

/**
* Devuelve un código que representa el día de la semana.
* <P>
* Los códigos se definen en la clase {@link SerialDate} como:
* <code>SUNDAY</code>, <code>MONDAY</code>, <code>TUESDAY</code>,
* <code>WEDNESDAY</code>, <code>THURSDAY</code>, <code>FRIDAY</code>, y
* <code>SATURDAY</code>.
*
* @return Un código que representa el día de la semana.
*/
public int getDayOfWeek() {
return (this.serial + 6) % 7 + 1;
}

/**
* Prueba la igualdad de esta fecha con un objeto arbitrario.
* <P>
* Este método SÓLO devuelve true si el objeto es una instancia de la
* clase base {@link SerialDate} y representa el mismo día que
* {@link SpreadsheetDate}.
*
* @param object el objeto que comparar (se permite <code>null</code>).
```

```
258   *
259   * @return Un valor booleano.
260   */
261   public boolean equals(final Object object) {
262
263   if (object instanceof SerialDate) {
264   final SerialDate s = (SerialDate) object;
265   return (s.toSerial() == this.toSerial());
266   }
267   else {
268   return false;
269   }
270
271   }
272
273   /**
274   * Devuelve un código hash para la instancia de este objeto.
275   *
276   * @return Un código hash.
277   */
278   public int hashCode() {
279   return toSerial();
280   }
281
282   /**
283   * Devuelve la diferencia (en días) entre esta fecha y la
284   * 'otra' fecha especificada.
285   *
286   * @param other la fecha con la que se compara.
287   *
288   * @return La diferencia (en días) entre esta fecha y la
289   * otra' fecha especificada.
290   */
291   public int compare(final SerialDate other) {
292   return this.serial – other.toSerial();
293   }
294
295   /**
296   * Implementa el método necesario para la interfaz Comparable.
297   *
298   * @param other el otro objeto (normalmente otro SerialDate).
299   *
300   * @return Un entero negativo, cero o un entero positivo si este objeto
301   * es menor que, igual o mayor que el objeto especificado.
302   */
303   public int compareTo(final Object other) {
304   return compare((SerialDate) other);
305   }
306
307   /**
308   * Devuelve true si esta SerialDate representa la misma fecha que la
309   * SerialDate especificada.
310   *
311   * @param other la fecha con la que se compara.
312   *
313   * @return <code>true</code> si esta SerialDate representa la misma fecha que
314   * la otra SerialDate especificada.
315   */
316   public boolean isOn(final SerialDate other) {
317   return (this.serial == other.toSerial());
318   }
319
320   /**
321   * Devuelve true si esta SerialDate representa una fecha anterior a
322   * la SerialDate especificada.
323   *
```

```
* @param other la fecha con la que se compara.
*
* @return <code>true</code> si esta SerialDate representa una fecha anterior a
* la SerialDate especificada.
*/
public boolean isBefore(final SerialDate other) {
return (this.serial < other.toSerial());
}

/**
* Devuelve true si esta SerialDate representa la misma fecha que la
* SerialDate especificada.
*
* @param other la fecha con la que se compara.
*
* @return <code>true</code> si esta SerialDate representa la misma fecha
* que la SerialDate especificada.
*/
public boolean isOnOrBefore(final SerialDate other) {
return (this.serial <= other.toSerial());
}

/**
* Devuelve true si esta SerialDate representa la misma fecha que la
* SerialDate especificada.
*
* @param other la fecha con la que se compara.
*
* @return <code>true</code> si esta SerialDate representa la misma fecha
* que la SerialDate especificada.
*/
public boolean isAfter(final SerialDate other) {
return (this.serial > other.toSerial());
}

/**
* Devuelve true si esta SerialDate representa la misma fecha que la
* SerialDate especificada.
*
* @param other la fecha con la que se compara.
*
* @return <code>true</code> si esta SerialDate representa la misma fecha
* que la SerialDate especificada.
*/
public boolean isOnOrAfter(final SerialDate other) {
return (this.serial >= other.toSerial());
}

/**
* Devuelve <code>true</code> si {@link SerialDate} se encuentra en el
* intervalo especificado (INCLUSIVE). El orden de fecha de d1 y d2 no es
* importante.
*
* @param d1 una fecha límite para el rango.
* @param d2 la otra fecha límite para el rango.
*
* @return Un valor booleano.
*/
public boolean isInRange(final SerialDate d1, final SerialDate d2) {
return isInRange(d1, d2, SerialDate.INCLUDE_BOTH);
}

/**
* Devuelve true si esta SerialDate se encuentra en el intervalo especificado
* (el invocador especifica si los puntos finales se incluyen o no). El orden
* de d1 y d2 no es importante.
```

```
390  *
391  * @param d1 una fecha límite para el rango.
392  * @param d2 la otra fecha límite para el rango.
393  * @param include un código que controla si la fecha inicial y final
394  * se incluyen en el intervalo.
395  *
396  * @return <code>true</code> si esta SerialDate se encuentra en el intervalo
397  * especificado.
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 start = 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 >= start && s <= end);
409  }
410  else if (include == SerialDate.INCLUDE_FIRST) {
411  return (s >= start && s < end);
412  }
413  else if (include == SerialDate.INCLUDE_SECOND) {
414  return (s > start && s <= end);
415  }
416  else {
417  return (s > start && s < end);
418  }
419  }
420
421  /**
422  * Calcular el número de serie a partir del día, mes y año.
423  * <P>
424  * 1-Ene-1900 = 2.
425  *
426  * @param d el día.
427  * @param m el mes.
428  * @param y el año.
429  *
430  * @return el número de serie a partir del día, mes y año.
431  */
432  private int calcSerial(final int d, final int m, final int y) {
433  final int 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) {
436  if (SerialDate.isLeapYear(y)) {
437  mm = mm + 1;
438  }
439  }
440  final int dd = d;
441  return yy + mm + dd + 1;
442  }
443
444  /**
445  * Calcular el día, mes y año a partir del número de serie.
446  */
447  private void calcDayMonthYear() {
448
449  // obtener el año a partir del número de serie de la fecha
450  final int days = this.serial - SERIAL_LOWER_BOUND;
451  // sobrevalorado ya que ignoramos los días bisiestos
452  final int overestimatedYYYY = 1900 + (days / 365);
453  final int leaps = SerialDate.leapYearCount(overestimatedYYYY);
454  final int nonleapdays = days - leaps;
455  // subestimado ya que sobrevaloramos los años
```

```
int underestimatedYYYY = 1900 + (nonleapdays / 365);

if (underestimatedYYYY == overestimatedYYYY) {
this.year = underestimatedYYYY;
}
else {
int ss1 = calcSerial(1, 1, underestimatedYYYY);
while (ss1 <= this.serial) {
underestimatedYYYY = underestimatedYYYY + 1;
ss1 = calcSerial(1, 1, underestimatedYYYY);
}
this.year = underestimatedYYYY - 1;
}

final int ss2 = calcSerial(1, 1, this.year);

int[] daysToEndOfPrecedingMonth
= AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;

if (isLeapYear(this.year)) {
daysToEndOfPrecedingMonth
= LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
}

// get the month from the serial date
int mm = 1;
int sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
while (sss < this.serial) {
mm = mm + 1;
sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
}
this.month = mm - 1;

// el resto es d(+1);
this.day = this.serial - ss2
- daysToEndOfPrecedingMonth[this.month] + 1;

}

}
```

Listado B-6
RelativeDayOfWeekRule.java

```
/* ===========================================================
* JCommon : biblioteca gratuita de clases de propósito general para Java(tm)
* ===========================================================
*
* (C) Copyright 2000-2005, de Object Refinery Limited y colaboradores.
*
* Información del proyecto: http://www.jfree.org/jcommon/index.html
*
* Esta biblioteca es software gratuito; puede distribuirla y/o modificarla
* bajo las condiciones de la Licencia pública general GNU publicada por
* la Free Software Foundation; ya sea la versión 2.1 de la licencia, u
* otra versión posterior (de su elección).
*
* Esta biblioteca se distribuye con la intención de que sea útil, pero
* SIN GARANTÍA ALGUNA, incluida la garantía implícita de COMERCIABILIDAD
* e IDONEIDAD PARA UN DETERMINADO FIN. Consulte la Licencia pública general GNU
* si necesita más información al respecto.
*
* Debería haber recibido una copia de la Licencia pública general GNU
```

```
 * junto a esta biblioteca; en caso contrario, contacte con la Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
 * EE.UU.
 *
 * [Java es una marca comercial o marca comercial registrada de Sun
 * Microsystems, Inc. en Estados Unidos y otros países.]
 *
 * --------------------------
 * RelativeDayOfWeekRule.java
 * --------------------------
 * (C) Copyright 2000-2003, de Object Refinery Limited y colaboradores.
 *
 * Autor original: David Gilbert (por Object Refinery Limited);
 * Colaboradores(s): -;
 *
 * $Id: RelativeDayOfWeekRule.java,v 1.6 2005/11/16 15:58:40 taqua Exp $
 *
 * Cambios (26-Oct-2001)
 * --------------------------
 * 26-Oct-2001 : Se cambió el paquete por com.jrefinery.date.*;
 * 03-Oct-2002 : Se corrigen los errores detectados por Checkstyle (DG);
 *
 */

package org.jfree.date;

/**
 * Una regla de fechas anuales que devuelve una fecha por cada año en función de
 * (a) una regla de referencia; (b) un día de la semana y (c) un parámetro de
 * selección (SerialDate.PRECEDING, SerialDate.NEAREST, SerialDate.FOLLOWING).
 * <P>
 * Por ejemplo, el Viernes Santo se puede especificado ‘el Viernes ANTERIOR al
 * domingo de Resurrección.
 *
 * @author David Gilbert
 */
public class RelativeDayOfWeekRule extends AnnualDateRule {

/** Una referencia a la regla de fechas anuales sobre la que se basa esta regla. */
private AnnualDateRule subrule;

/**
 * El día de la semana (SerialDate.MONDAY, SerialDate.TUESDAY, etc).
 */
private int dayOfWeek;

/** Indica que día de la semana (PRECEDING, NEAREST o FOLLOWING). */
private int relative;

/**
 * Constructor predeterminado: genera una regla para el lunes siguiente al 1 de enero.
 */
public RelativeDayOfWeekRule() {
this(new DayAndMonthRule(), SerialDate.MONDAY, SerialDate.FOLLOWING);
}

/**
 * Constructor estándar: genera una regla en función de la subregla proporcionada.
 *
 * @param subrule la regla que determina la fecha de referencia.
 * @param dayOfWeek el día de la semana relativo a la fecha de referencia.
 * @param relative indica “qué” día de la semana (anterior, más próximo
 * o posterior).
 */
public RelativeDayOfWeekRule(final AnnualDateRule subrule,
final int dayOfWeek, final int relative) {
```

```
86   this.subrule = subrule;
87   this.dayOfWeek = dayOfWeek;
88   this.relative = relative;
89   }
90
91   /**
92   * Devuelve la subregla (también denominada regla de referencia).
93   *
94   * @return La regla de fechas anuales que determina la fecha de referencia para
95   * esta regla.
96   */
97   public AnnualDateRule getSubrule() {
98   return this.subrule;
99   }
100
101  /**
102  * Establece la subregla.
103  *
104  * @param subrule la regla de fechas anuales que determina la fecha de
105  * referencia para esta regla.
106  */
107  public void setSubrule(final AnnualDateRule subrule) {
108  this.subrule = subrule;
109  }
110
111  /**
112  * Devuelve el día de la semana de esta regla.
113  *
114  * @return el día de la semana de esta regla.
115  */
116  public int getDayOfWeek() {
117  return this.dayOfWeek;
118  }
119
120  /**
121  * Establece el día de la semana de esta regla.
122  *
123  * @param dayOfWeek el día de la semana de (SerialDate.MONDAY,
124  * SerialDate.TUESDAY, etc.).
125  */
126  public void setDayOfWeek(final int dayOfWeek) {
127  this.dayOfWeek = dayOfWeek;
128  }
129
130  /**
131  * Devuelve el atributo ‘relativo’ que determina “qué”
132  * día de la semana nos interesa (SerialDate.PRECEDING,
133  * SerialDate.NEAREST o SerialDate.FOLLOWING).
134  *
135  * @return El atributo ‘relativo’.
136  */
137  public int getRelative() {
138  return this.relative;
139  }
140
141  /**
142  * Establece el atributo ‘relativo’ (SerialDate.PRECEDING, SerialDate.NEAREST,
143  * SerialDate.FOLLOWING).
144  *
145  * @param relative determina “qué” día de la semana se selecciona con esta
146  * regla.
147  */
148  public void setRelative(final int relative) {
149  this.relative = relative;
150  }
151
```

```
152  /**
153  * Crea un clon de esta regla.
154  *
155  * @return un clon de esta regla.
156  *
157  * @throws CloneNotSupportedException nunca debe producirse.
158  */
159  public Object clone() throws CloneNotSupportedException {
160  final RelativeDayOfWeekRule duplicate
161  = (RelativeDayOfWeekRule) super.clone();
162  duplicate.subrule = (AnnualDateRule) duplicate.getSubrule().clone();
163  return duplicate;
164  }
165
166  /**
167  * Devuelve la fecha generada por esta regla, para el año especificado.
168  *
169  * @param year el año (1900 &lt;= year &lt;= 9999).
170  *
171  * @return La fecha generada por esta regla para un año concreto (posiblemente
172  * <code>null</code>).
173  */
174  public SerialDate getDate(final int year) {
175
176  // comprobar argumento…
177  if ((year < SerialDate.MINIMUM_YEAR_SUPPORTED)
178  || (year > SerialDate.MAXIMUM_YEAR_SUPPORTED)) {
179  throw new IllegalArgumentException(
180  “RelativeDayOfWeekRule.getDate(): year outside valid range.”);
181  }
182
183  // calcular la fecha…
184  SerialDate result = null;
185  final SerialDate base = this.subrule.getDate(year);
186
187  if (base != null) {
188  switch (this.relative) {
189  case(SerialDate.PRECEDING):
190  result = SerialDate.getPreviousDayOfWeek(this.dayOfWeek,
191  base);
192  break;
193  case(SerialDate.NEAREST):
194  result = SerialDate.getNearestDayOfWeek(this.dayOfWeek,
195  base);
196  break;
197  case(SerialDate.FOLLOWING):
198  result = SerialDate.getFollowingDayOfWeek(this.dayOfWeek,
199  base);
200  break;
201  default:
202  break;
203  }
204  }
205  return result;
206
207  }
208
209  }
```

Listado B-7
DayDate.java (Final)

```
1    /* ===========================================================
```

```
2     * JCommon: biblioteca gratuita de clases de propósito general para Java(tm)
3     * ============================================================
4     *
5     * (C) Copyright 2000-2005, de Object Refinery Limited y colaboradores.
…
36    */
37    package 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 asigna cada fecha a un entero que
45    * representa un número ordinal de días de un origen fijo.
46    *
47    * ¿Por qué no usar java.útil.Date? Lo haremos, cuando tenga sentido. En ocasiones,
48    * java.util.Date puede ser demasiado precisa; representa un instante en el tiempo,
49    * con una precisión de 1/1000 de segundo (y la fecha depende de la
50    * zona horaria). En ocasiones solo querremos representar un día concreto (como el 21
51    * de enero de 2015) sin preocuparnos de la hora del día, la
52    * zona horaria u otros aspectos. Para eso hemos definido SerialDate.
53    *
54    * Usar DayDateFactory.makeDate para crear una instancia.
55    *
56    * @author David Gilbert
57    * @author Robert C. Martin realizó gran parte de la refactorización.
58    */
59
60    public abstract class DayDate implements Comparable, Serializable {
61    public abstract int getOrdinalDay();
62    public abstract int getYear();
63    public abstract Month getMonth();
64    public abstract int getDayOfMonth();
65
66    protected abstract Day getDayOfWeekForOrdinalZero();
67
68    public DayDate plusDays(int days) {
69    return DayDateFactory.makeDate(getOrdinalDay() + days);
70    }
71
72    public DayDate plusMonths(int months) {
73    int thisMonthAsOrdinal = getMonth().toInt() - Month.JANUARY.toInt();
74    int thisMonthAndYearAsOrdinal = 12 * getYear() + thisMonthAsOrdinal;
75    int resultMonthAndYearAsOrdinal = thisMonthAndYearAsOrdinal + months;
76    int resultYear = resultMonthAndYearAsOrdinal / 12;
77    int resultMonthAsOrdinal = resultMonthAndYearAsOrdinal % 12 + Month.JANUARY.toInt();
78    Month resultMonth = Month.fromInt(resultMonthAsOrdinal);
79    int resultDay = correctLastDayOfMonth(getDayOfMonth(), resultMonth, resultYear);
80    return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);
81    }
82
83    public DayDate plusYears(int years) {
84    int resultYear = getYear() + years;
85    int resultDay = correctLastDayOfMonth(getDayOfMonth(), getMonth(), resultYear);
86    return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);
87    }
88
89    private int correctLastDayOfMonth(int day, Month month, int year) {
90    int lastDayOfMonth = DateUtil.lastDayOfMonth(month, year);
91    if (day > lastDayOfMonth)
92    day = lastDayOfMonth;
93    return day;
94    }
95
96    public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {
```

```
int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
if (offsetToTarget >= 0)
offsetToTarget -= 7;
return plusDays(offsetToTarget);
}

public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {
int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
if (offsetToTarget <= 0)
offsetToTarget += 7;
return plusDays(offsetToTarget);
}

public DayDate getNearestDayOfWeek(Day targetDayOfWeek) {
int offsetToThisWeeksTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
int offsetToPreviousTarget = offsetToFutureTarget - 7;

if (offsetToFutureTarget > 3)
return plusDays(offsetToPreviousTarget);
else
return plusDays(offsetToFutureTarget);
}

public DayDate getEndOfMonth() {
Month month = getMonth();
int year = getYear();
int lastDay = DateUtil.lastDayOfMonth(month, year);
return DayDateFactory.makeDate(lastDay, month, year);
}

public Date toDate() {
final Calendar calendar = Calendar.getInstance();
int ordinalMonth = getMonth().toInt() - Month.JANUARY.toInt();
calendar.set(getYear(), ordinalMonth, getDayOfMonth(), 0, 0, 0);
return calendar.getTime();
}

public String toString() {
return String.format("%02d-%s-%d", getDayOfMonth(), getMonth(), getYear());
}

public Day getDayOfWeek() {
Day startingDay = getDayOfWeekForOrdinalZero();
int startingOffset = startingDay.toInt() - Day.SUNDAY.toInt();
int ordinalOfDayOfWeek = (getOrdinalDay() + startingOffset) % 7;
return Day.fromInt(ordinalOfDayOfWeek + Day.SUNDAY.toInt());
}

public int daysSince(DayDate date) {
return getOrdinalDay() - date.getOrdinalDay();
}

public boolean isOn(DayDate other) {
return getOrdinalDay() == other.getOrdinalDay();
}

public boolean isBefore(DayDate other) {
return getOrdinalDay() < other.getOrdinalDay();
}

public boolean isOnOrBefore(DayDate other) {
return getOrdinalDay() <= other.getOrdinalDay();
}

public boolean isAfter(DayDate other) {
```

```
163  return getOrdinalDay() > other.getOrdinalDay();
164  }
165
166  public boolean isOnOrAfter(DayDate other) {
167  return getOrdinalDay() >= other.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 left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());
176  int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());
177  return interval.isIn(getOrdinalDay(), left, right);
178  }
179  }
```

Listado B-8
Month.java (Final)

```
1   package org.jfree.date;
2
3   import java.text.DateFormatSymbols;
4
5   public enum Month {
6   JANUARY(1), FEBRUARY(2), MARCH(3),
7   APRIL(4), MAY(5), JUNE(6),
8   JULY(7), AUGUST(8), SEPTEMBER(9),
9   OCTOBER(10),NOVEMBER(11),DECEMBER(12);
10  private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols();
11  private static final int[] LAST_DAY_OF_MONTH =
12  {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
13
14  private int index;
15
16  Month(int index) {
17  this.index = index;
18  }
19
20  public static Month fromInt(int monthIndex) {
21  for (Month m : Month.values()) {
22  if (m.index == monthIndex)
23  return m;
24  }
25  throw new IllegalArgumentException(“Invalid month index ” + monthIndex);
26  }
27
28  public int lastDay() {
29  return LAST_DAY_OF_MONTH[index];
30  }
31
32  public int quarter() {
33  return 1 + (index - 1) / 3;
34  }
35
36  public String toString() {
37  return dateFormatSymbols.getMonths()[index - 1];
38  }
39
40  public String toShortString() {
41  return dateFormatSymbols.getShortMonths()[index - 1];
42  }
```

```
43
44  public static Month parse(String s) {
45  s = s.trim();
46  for (Month m : Month.values())
47  if (m.matches(s))
48  return m;
49
50  try {
51  return fromInt(Integer.parseInt(s));
52  }
53  catch (NumberFormatException e) {}
54  throw new IllegalArgumentException(“Invalid month ” + s);
55  }
56
57  private boolean matches(String s) {
58  return s.equalsIgnoreCase(toString()) ||
59  s.equalsIgnoreCase(toShortString());
60  }
61
62  public int toInt() {
63  return index;
64  }
65  }
```

Listado B-9
Day.java (Final)

```
1   package org.jfree.date;
2
3   import java.util.Calendar;
4   import java.text.DateFormatSymbols;
5
6   public enum Day {
7   MONDAY(Calendar.MONDAY),
8   TUESDAY(Calendar.TUESDAY),
9   WEDNESDAY(Calendar.WEDNESDAY),
10  THURSDAY(Calendar.THURSDAY),
11  FRIDAY(Calendar.FRIDAY),
12  SATURDAY(Calendar.SATURDAY),
13  SUNDAY(Calendar.SUNDAY);
14
15  private final int index;
16  private static DateFormatSymbols dateSymbols = new DateFormatSymbols();
17
18  Day(int day) {
19  index = day;
20  }
21
22  public static Day fromInt(int index) throws IllegalArgumentException {
23  for (Day d : Day.values())
24  if (d.index == index)
25  return d;
26  throw new IllegalArgumentException(
27  String.format(“Illegal day index: %d.”, index));
28  }
29
30  public static Day parse(String s) throws IllegalArgumentException {
31  String[] shortWeekdayNames =
32  dateSymbols.getShortWeekdays();
33  String[] weekDayNames =
34  dateSymbols.getWeekdays();
35
36  s = s.trim();
```

```
for (Day day : Day.values()) {
if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
s.equalsIgnoreCase(weekDayNames[day.index])) {
return day;
}
}
throw new IllegalArgumentException(
String.format(“%s is not a valid weekday string”, s));
}

public String toString() {
return dateSymbols.getWeekdays()[index];
}

public int toInt() {
return index;
}
}
```

Listado B-10
DateInterval.java (Final)

```
package org.jfree.date;

public enum DateInterval {
OPEN {
public boolean isIn(int d, int left, int right) {
return d > left && d < right;
}
},
CLOSED_LEFT {
public boolean isIn(int d, int left, int right) {
return d >= left && d < right;
}
},
CLOSED_RIGHT {
public boolean isIn(int d, int left, int right) {
return d > left && d <= right;
}
},
CLOSED {
public boolean isIn(int d, int left, int right) {
return d >= left && d <= right;
}
};

public abstract boolean isIn(int d, int left, int right);
}
```

Listado B-11
WeekInMonth.java (Final)

```
package org.jfree.date;

public enum WeekInMonth {
FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);
private final int index;

WeekInMonth(int index) {
```

```
  this.index = index;
  }

  public int toInt() {
  return index;
  }
  }
```

Listado B-12
WeekdayRange.java (Final)

```
package org.jfree.date;

public enum WeekdayRange {
LAST, NEAREST, NEXT
}
```

Listado B-13
DateUtil.java (Final)

```
  package org.jfree.date;

  import java.text.DateFormatSymbols;

  public class DateUtil {
  private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols();

  public static String[] getMonthNames() {
  return dateFormatSymbols.getMonths();
  }

  public static boolean isLeapYear(int year) {
  boolean fourth = year % 4 == 0;
  boolean hundredth = year % 100 == 0;
  boolean fourHundredth = year % 400 == 0;
  return fourth && (!hundredth || fourHundredth);
  }

  public static int lastDayOfMonth(Month month, int year) {
  if (month == Month.FEBRUARY && isLeapYear(year))
  return month.lastDay() + 1;
  else
  return month.lastDay();
  }

  public static int leapYearCount(int year) {
  int leap4 = (year - 1896) / 4;
  int leap100 = (year - 1800) / 100;
  int leap400 = (year - 1600) / 400;
  return leap4 - leap100 + leap400;
  }
  }
```

Listado B-14
DayDateFactory.java (Final)

```
package org.jfree.date;

public abstract class DayDateFactory {
private static DayDateFactory factory = new SpreadsheetDateFactory();
public static void setInstance(DayDateFactory factory) {
DayDateFactory.factory = factory;
}

protected abstract DayDate _makeDate(int ordinal);
protected abstract DayDate _makeDate(int day, Month month, int year);
protected abstract DayDate _makeDate(int day, int month, int year);
protected abstract DayDate _makeDate(java.util.Date date);
protected abstract int _getMinimumYear();
protected abstract int _getMaximumYear();

public static DayDate makeDate(int ordinal) {
return factory._makeDate(ordinal);
}

public static DayDate makeDate(int day, Month month, int year) {
return factory._makeDate(day, month, year);
}

public static DayDate makeDate(int day, int month, int year) {
return factory._makeDate(day, month, year);
}

public static DayDate makeDate(java.util.Date date) {
return factory._makeDate(date);
}

public static int getMinimumYear() {
return factory._getMinimumYear();
}

public static int getMaximumYear() {
return factory._getMaximumYear();
}
}
```

Listado B-15
SpreadsheetDateFactory.java (Final)

```
package org.jfree.date;

import java.util.*;

public class SpreadsheetDateFactory extends DayDateFactory {
public DayDate _makeDate(int ordinal) {
return new SpreadsheetDate(ordinal);
}

public DayDate _makeDate(int day, Month month, int year) {
return new SpreadsheetDate(day, month, year);
}

public DayDate _makeDate(int day, int month, int year) {
return new SpreadsheetDate(day, month, year);
}

public DayDate _makeDate(Date date) {
```

```
final GregorianCalendar calendar = new GregorianCalendar();
calendar.setTime(date);
return new SpreadsheetDate(
calendar.get(Calendar.DATE),
Month.fromInt(calendar.get(Calendar.MONTH) + 1),
calendar.get(Calendar.YEAR));
}

protected int _getMinimumYear() {
return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
}

protected int _getMaximumYear() {
return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
}
}
```

Listado B-16
SpreadsheetDate.java (Final)

```
/* ===========================================================
* JCommon: biblioteca gratuita de clases de propósito general para Java(tm)
* ===========================================================
*
* (C) Copyright 2000-2005, de Object Refinery Limited y Colaboradores.
*
…
*
*/

package org.jfree.date;

import static org.jfree.date.Month.FEBRUARY;

import java.util.*;

/**
* Representa una fecha con un entero, de forma similar a la
* implementación en Microsoft Excel. El intervalo de fechas admitido es
* del 1-Ene-1900 al 31-Dic-9999.
* <p/>
* Recuerde que Excel tiene un error que reconoce el año
* 1900 como bisiesto cuando en realidad no lo es. Encontrará más
* información en el sitio de Microsoft, en el artículo Q181370:
* <p/>
* http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
* <p/>
* Excel usa como convención que el 1-Ene-1900 = 1. Esta clase usa la
* convención de que el 1-Ene-1900 = 2.
* Como resultado, el número de día de esta clase será diferente al de
* Excel para enero y febrero de 1900… pero Excel añade un día
* más (29-Feb-1900 que en realidad no existe) y a partir de ahí
* los números de los días coinciden.
*
* @author David Gilbert
*/
public class SpreadsheetDate extends DayDate {
public static final int EARLIEST_DATE_ORDINAL = 2; // 1/1/1900
public static final int LATEST_DATE_ORDINAL = 2958465; // 12/31/9999
public static final int MINIMUM_YEAR_SUPPORTED = 1900;
public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
{0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
```

```
  static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
  {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};

  private int ordinalDay;
  private int day;
  private Month month;
  private int year;

  public SpreadsheetDate(int day, Month month, int year) {
  if (year < MINIMUM_YEAR_SUPPORTED || year > MAXIMUM_YEAR_SUPPORTED)
  throw new IllegalArgumentException(
  “The ‘year’ argument must be in range ” +
  MINIMUM_YEAR_SUPPORTED + “ to ” + MAXIMUM_YEAR_SUPPORTED + “.”);
  if (day < 1 || day > DateUtil.lastDayOfMonth(month, year))
  throw new IllegalArgumentException(“Invalid ‘day’ argument.”);

  this.year = year;
  this.month = month;
  this.day = day;
  ordinalDay = calcOrdinal(day, month, year);
  }

  public SpreadsheetDate(int day, int month, int year) {
  this(day, Month.fromInt(month), year);
  }

  public SpreadsheetDate(int serial) {
  if (serial < EARLIEST_DATE_ORDINAL || serial > LATEST_DATE_ORDINAL)
  throw new IllegalArgumentException(
  “SpreadsheetDate: Serial must be in range 2 to 2958465.”);

  ordinalDay = serial;
  calcDayMonthYear();
  }

  public int getOrdinalDay() {
  return ordinalDay;
  }

  public int getYear() {
  return year;
  }

  public Month getMonth() {
  return month;
  }

  public int getDayOfMonth() {
  return day;
  }

  protected Day getDayOfWeekForOrdinalZero() {return Day.SATURDAY;}

  public boolean equals(Object object) {
  if (!(object instanceof DayDate))
  return false;

  DayDate date = (DayDate) object;
  return date.getOrdinalDay() == getOrdinalDay();
  }

  public int hashCode() {
  return getOrdinalDay();
  }

  public int compareTo(Object other) {
```

```
return daysSince((DayDate) other);
}

private int calcOrdinal(int day, Month month, int year) {
int leapDaysForYear = DateUtil.leapYearCount(year - 1);
int daysUpToYear = (year - MINIMUM_YEAR_SUPPORTED) * 365 + leapDaysForYear;
int daysUpToMonth = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[month.toInt()];
if (DateUtil.isLeapYear(year) && month.toInt() > FEBRUARY.toInt())
daysUpToMonth++;
int daysInMonth = day - 1;
return daysUpToYear + daysUpToMonth + daysInMonth + EARLIEST_DATE_ORDINAL;
}

private void calcDayMonthYear() {
int days = ordinalDay - EARLIEST_DATE_ORDINAL;
int overestimatedYear = MINIMUM_YEAR_SUPPORTED + days / 365;
int nonleapdays = days - DateUtil.leapYearCount(overestimatedYear);
int underestimatedYear = MINIMUM_YEAR_SUPPORTED + nonleapdays / 365;

year = huntForYearContaining(ordinalDay, underestimatedYear);
int firstOrdinalOfYear = firstOrdinalOfYear(year);
month = huntForMonthContaining(ordinalDay, firstOrdinalOfYear);
day = ordinalDay - firstOrdinalOfYear - daysBeforeThisMonth(month.toInt());
}

private Month huntForMonthContaining(int anOrdinal, int firstOrdinalOfYear) {
int daysIntoThisYear = anOrdinal - firstOrdinalOfYear;
int aMonth = 1;
while (daysBeforeThisMonth(aMonth) < daysIntoThisYear)
aMonth++;

return Month.fromInt(aMonth - 1);
}

private int daysBeforeThisMonth(int aMonth) {
if (DateUtil.isLeapYear(year))
return LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
else
return AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
}

private int huntForYearContaining(int anOrdinalDay, int startingYear) {
int aYear = startingYear;
while (firstOrdinalOfYear(aYear) <= anOrdinalDay)
aYear++;

return aYear - 1;
}

private int firstOrdinalOfYear(int year) {
return calcOrdinal(1, Month.JANUARY, year);
}

public static DayDate createInstance(Date date) {
GregorianCalendar calendar = new GregorianCalendar();
calendar.setTime(date);
return new SpreadsheetDate(calendar.get(Calendar.DATE),
Month.fromInt(calendar.get(Calendar.MONTH) + 1),
calendar.get(Calendar.YEAR));

}
}
```

Epílogo

En 2005, mientras asistía a la conferencia Agile en Denver (EE.UU.), Elisabeth Hedrickson[125] me dio una pulsera verde parecida a la que Lance Armstrong popularizó hace unos años. En ésta se leía *Test Obsessed* (Obsesionado por las pruebas). Me la puse y la lucí con orgullo. Desde que aprendí el TDD de Kent Beck en 1999, sin duda el desarrollo controlado por pruebas me ha obsesionado.

Pero sucedió algo extraño. No me podía quitar la pulsera. No porque se hubiera quedado físicamente pegada, sino porque estaba moralmente pegada.

La pulsera resumía mi ética profesional.

Era un indicador visible de mi compromiso por crear el mejor código posible. Si me la hubiera quitado habría traicionado a esa ética y a ese compromiso.

Y todavía la llevo en la muñeca.

Cuando escribo código, la veo ahí. Es un recordatorio constante de la promesa que me hice de escribir código limpio.

Robert Cecil “Uncle Bob” Martin (Palo Alto California, Estados Unidos, 1952). Es un prestigioso desarrollador de *software* desde 1970 y consultor internacional desde 1990. Es fundador y presidente de Object Mentor, Inc., un equipo de experimentados consultores que ayudan a clientes de todo el mundo en diferentes campos de la programación como C++, Java, C#, Ruby, Programación orientada a objetos (POO), patrones de diseño, UML, metodologías ágiles y programación eXtreme.

Notas

[1] [Beck07]. <<

[2] Cuando Ignaz Semmelweis recomendó en 1847 que los médicos se lavaran las manos, su propuesta fue rechazada aludiendo que los doctores estaban demasiado ocupados para hacerlo entre paciente y paciente. <<

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

[4] [Knuth92]. <<

[5] Es una adaptación del mensaje de despedida de Robert Stephenson Smyth Baden-Powell a los Scouts: «Intentad dejar este mundo un poco mejor de como os lo encontrasteis…». <<

[6] Como veremos más adelante, aunque un contenedor *sea* una `lista`, no conviene codificar el tipo de contenedor en el nombre. <<

[7] Imagine que se crea una variable con el nombre `klass` sólo porque el nombre `class` se ha usado en otro elemento. <<

[8] Uncle Bob solía hacerlo en C++ pero ha abandonado esta práctica ya que no es necesario en los IDE modernos. <<

[9] http://java.sun.com/products/javabeans/docs/spec.html. <<

[10] Una herramienta de pruebas de código abierto (www.fitnese.org). <<

[11] Una herramienta de código abierto para probar unidades para Java (www.junit.org). <<

[12] Le pregunté a Kent si todavía conservaba una copia, pero no la encontró. Busqué en mis viejos ordenadores, pero nada. Solamente se conserva el recuerdo de aquél programa. <<

[13] El lenguaje LOGO usaba la palabra clave `TO` al igual que Ruby y Python usaban `def`. Por tanto, todas las funciones comenzaban por `TO`, lo que tenía un efecto interesante en cómo se diseñaban. <<

[14] [KP78], p. 37. <<

[15] Y, por supuesto, se incluyen cadenas if/else. <<

[16] a. http://en.wikipedia.org/wiki/Single_responsibility_principle

b. http://www.objectmentor.com/resources/articles/srp.pdf <<

[17] a. http://en.wikipedia.org/wiki/Open/closed_principle

b. http://www.objectmentor.com/resources/articles/ocp.pdf <<

[18] [GOF]. <<

[19] Terminé la refactorización de un módulo que usaba la forma dinámica. Conseguí convertir el módulo `outputStream` en un campo de la clase y las invocaciones de `writeField` a formato monódico. El resultado fue mucho más limpio. <<

[20] Existen algunos que creen que pueden evitar volver a compilar e implementar, y nos hemos encargado de ellos. <<

[21] Ejemplo de principio abierto/cerrado (OCP) [PPP02]. <<

[22] El principio DRY. [PRAG]. <<

[23] [SP72]. <<

[24] [KP78], p. 144. <<

[25] La tendencia actual de los IDE de comprobar la ortografía de los comentarios será un bálsamo para los que tenemos que leer gran cantidad de código. <<

[26] El cuadro muestra sigma/2 por encima y debajo de la media. Asumo que la distribución de la longitud de archivos no es normal, por lo que la desviación estándar no es matemáticamente precisa. Pero aquí el objetivo no es la precisión, sino la sensación. <<

[27] Es lo contrario a lo que sucede en lenguajes como Pascal, C y C++ que obligan a definir, o al menos a declarar, las funciones antes de usarlas. <<

[28] ¿A quien voy a engañar? Sigo siendo programador de lenguajes de ensamblado. En este caso, el hábito sí hace al monje. <<

[29] Siempre existe una solución conocida por los diseñadores orientados a objetos con experiencia: VISITOR o entrega dual, por ejemplo. Pero son técnicas costosas y suelen devolver la estructura de un programa por procedimientos. <<

[30] http://es.wikipedia.org/wiki/Ley_de_Demeter. <<

[31] De la estructura Apache. <<

[32] En ocasiones se denomina Feature Envy (Envidia de las características), de [Refactoring]. <<

[33] [Martin]. <<

[34] [BeckTDD], pp. 136-137. <<

[35] Véase el patrón del adaptador en [GOF]. <<

[36] Más información al respecto en [WELC]. <<

[37] *Professionalism and Test-Driven Development*, Robert C. Martin, Object Mentor, IEEE Software, Mayo/Junio 2007 (Vol. 24, No. 3) pp. 32-36 http://doi.ieeecomputersociety.org/10.1109/MS.2007.85 <<

[38] http://fitnesse.org/FitNesse.AcceptanceTestPatterns. <<

[39] Véase el apartado sobre asignaciones mentales del capítulo 2. <<

[40] Véase la entrada de Dave Astel: http://www.artima.com/weblogs/viewpost.jsp?thread=35578 <<

[41] [RSpec]. <<

[42] [GOF]. <<

[43] ¡Cíñase al código! <<

[44] Materiales de formación de Object Mentor. <<

[45] [RDD]. <<

[46] Encontrará más Información sobre este principio en [PPP]. <<

[47] [Knuth92]. <<

[48] [PPP]. <<

[49] [PPP]. <<

[50] [Mezzaros07]. <<

[51] [GOF]. <<

[52] Véase, por ejemplo, [Fowler]. <<

[53] Véase [Spring], También existe una estructura Spring.NET. <<

[54] No olvide que la creación de instancias/evaluación tardía es sólo una optimización, puede que prematura. <<

[55] Sistema de administración de base de datos. <<

[56] Consulte [AOSD] si necesita información general sobre aspectos y [AspectJ] y [Colyer] para Información concreta de AspectJ. <<

[57] No se necesita la modificación manual del código fuente de destino. <<

[58] Véase [CGLIB], [ASM] y [Javassist]. <<

[59] Si necesita ejemplos más detallados de la API Proxy y ejemplos de uso, consulte [Goetz]. <<

[60] AOP se suele confundir con técnicas empleadas para implementarlo, como la intercepción y envoltorio de métodos a través de `proxies`. El verdadero valor de un sistema AOP es la capacidad para especificar comportamientos del sistema de forma concisa y modular. <<

[61] Véase [Spring] y [JBoss]. Java puro significa sin AspectJ. <<

[62] Adaptado de www.theserverside.com/tt/articles/article.tss?l=IntrotoSpring25. <<

[63] [GOF]. <<

[64] El ejemplo se puede simplificar mediante mecanismos que usen convenciones y anotaciones de Java 5 para reducir la cantidad necesaria de lógica de conexión explícita. <<

[65] Adaptado de http://www.onjava.com/pub/a/onjava/2006/05/17/standardizing-with-ejb3-java-persistence-api.html. <<

[66] Véase [AspectJ] y [Colyer]. <<

[67] No confundir con la práctica de diseño anticipado. BDUF es la práctica de diseñar todo por adelantado antes de implementar nada. <<

[68] Existe una cantidad significativa de exploración iterativa y detalles de análisis, incluso una vez iniciada la construcción. <<

[69] El término fue empleado por primera vez por [Kolence]. <<

[70] El trabajo de [Alexander] ha sido una gran influencia para la comunidad de *software*. <<

[71] Véase, por ejemplo, [DSL]. [JMock] es un buen ejemplo de API de Java que crea un DSL. <<

[72] [XPE]. <<

[73] [GOF]. <<

[74] Correspondencia privada. <<

[75] Rayos cósmicos, repeticiones, etc. <<

[76] Véase el apéndice A. <<

[77] Véase el apéndice A. <<

[78] [PPP]. <<

[79] Véase el apéndice A. <<

[80] [PRAG]. <<

[81] [Lea99]. <<

[82] http://en.wikipedia.org/wiki/Producer-consumer. <<

[83] http://en.wikipedia.org/wiki/Readers-writers_problem. <<

[84] http://es.wikipedia.org/wiki/Problema_de_la_cena_de_los_filósofos. <<

[85] Véase el apéndice A. <<

[86] Una sección crítica es cualquier sección de código que debe protegerse de usos simultáneos por parte del programa para que sea correcta. <<

[87] Véase el apéndice A. <<

[88] Véase el apéndice A. <<

[89] ¿Sabía que el modelo de procesos de Java no garantiza el procesamiento preventivo? Los SO modernos sí lo hacen, de modo que lo obtiene de forma gratuita. No obstante, la MVJ no lo garantiza. <<

[90] No es estrictamente el caso. Como la MVJ no garantiza los procesos preventivos, un determinado algoritmo puede que siempre funcione en un SO que no prevea los procesos. Lo contrario también es posible, pero por distintos motivos. <<

[91]

https://www.ibm.com/developerworks/community/groups/service/html/communityview?lang=es&communityUuid=18d10b14-e2c8-4780-bace-9af1fc463cc0. <<

[92] Hace poco modifiqué este módulo para Ruby. Tenía una séptima parte del tamaño original y una mejor estructura. <<

[93] Para evitar este tipo de sorpresas, añadí una nueva prueba de unidad que invocaba todas las pruebas de FitNesse. <<

[94] *JUnit Pocket Guide*, Kent Beck, O'Reilly. 2004. p. 43. <<

[95] Véase el capítulo 1. <<

[96] Una solución mejor sería que el Javadoc presentara todos los comentarios con formato previo, para que tengan el mismo aspecto en el código y en el documento. <<

[97] Algunos revisores de este texto no comparten esta decisión. Sostienen que en una estructura de código abierto es más recomendable ejercer control manual sobre el ID de serie para que los cambios mínimos del *software* no invaliden las fechas señalizadas antiguas. Me parece justo. Sin embargo, al menos el fallo, aunque sea inconveniente, tiene un motivo evidente. Por otra parte, si el autor de la clase se olvida de actualizar el ID, el modo de fallo será indefinido y puede que silencioso. Creo que la moraleja es que no debe esperar a deserializar entre versiones. <<

[98] [GOF]. <<

[99] Ibid. <<

[100] Ibid. <<

[101] [Simmons04], p. 73. <<

[102] [Refactoring]. <<

[103] [Beck97]. <<

[104] [Refactoring]. <<

[105] http://es.wikipedia.org/wiki/Principio_de_la_Mínima_Sorpresa <<

[106] [PRAG]. <<

[107] [GOF]. <<

[108] [GOF]. <<

[109] [Refactoring]. <<

[110] En concreto, el Principio de Responsabilidad Única, el Principio Abierto/Cerrado y el Principio de Cierre Común. Véase [PPP]. <<

[111] [Beck97], p. 108. <<

[112] [Beck07]. <<

[113] Es distinto saber cómo funciona el código y saber si el algoritmo se encargará de realizar la tarea para la que se necesita. Es habitual desconocer si un algoritmo es el adecuado. Desconocer lo que hace el código es indolencia. <<

[114] O mejor todavía, una clase Money que use enteros. <<

[115] [PRAG]. p. 138. <<

[116] Véase la cita de Ward Cunningham del capítulo 1. <<

[117] [DDD]. <<

[118] Puede comprobar personalmente el código antes y después, y revisar las versiones con y sin subprocesos, que veremos en un apartado posterior. <<

[119] Es una comparación simplificada, pero para los objetivos de este ejercicio es un modelo válido. <<

[120] De hecho, la interfaz `Iterator` es incompatible con subprocesos por naturaleza. No se diseñó para usar varios subprocesos, de modo que no debería sorprenderle. <<

[121] Por ejemplo, alguien añade un resultado de depuración y el problema desaparece. El código de depuración corrige el problema, pero permanece en el sistema. <<

[122] Siglas de *There ain’t no such thing as a free lunch* (Todo tiene un precio). <<

[123] http://www.haifa.ibm.com/projects/verification/contest/index.html <<

[124] Véase [Lea99] p. 191. <<

[125] http://www.qualitytree.com/ <<

Table of Contents