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).

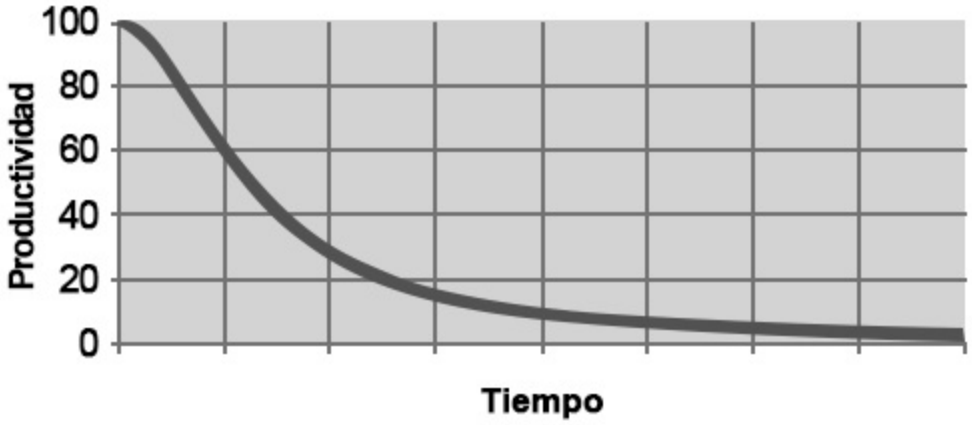


Figura 1.1. Productividad frente a tiempo.

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.

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.

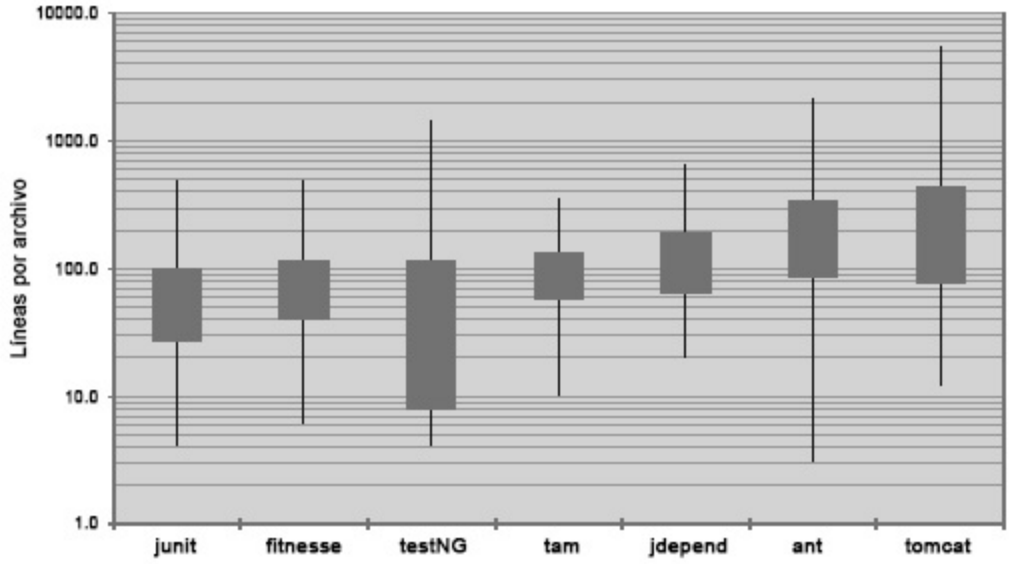


Figura 5.1. Escala LOG de distribuciones de longitud de archivos (altura del cuadro = sigma).

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.

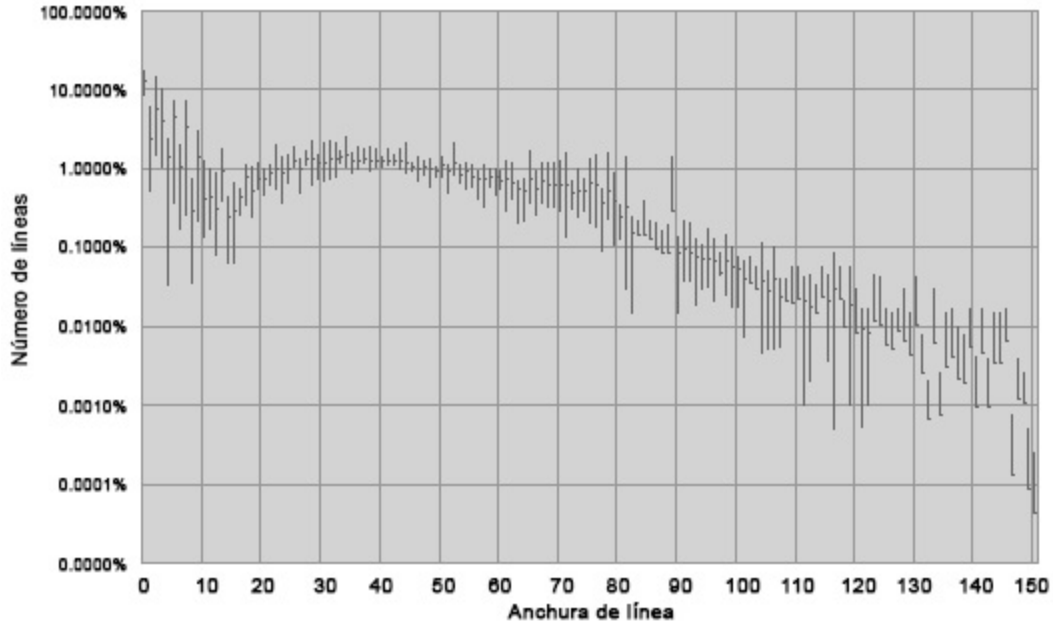


Figura 5.2. Distribución de anchura de líneas en Java.

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:

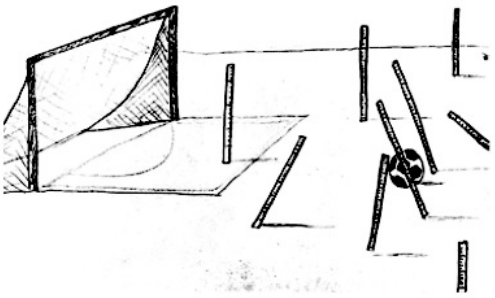
```
  public class CommentWidget extends TextWidget {
     public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";

     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;
  }
}
```