O'REILLY® ANAYA MULTIMEDIA

¿Ordenar primero?

Un ejercicio personal en diseño de software empírico

Kent Beck

Prólogo de Larry Constantine

Agradecimientos

El "autor" de un libro es una ficción con fines contables. Yo he escrito las palabras, pero no estarían en tus manos sin la participación de un montón de personas. Estas son algunas de ellas.

Gracias por su pronta respuesta técnica a Anna Goodman, Matan Zruya, Jeff Carbonella, David Haley, Kelly Sutton y el resto de mis estudiantes de Gusto. Agradezco a Maude Lemaire, Rebecca Wirfs-Brock, Vlad Khononov y Oleksii Torunov su respuesta técnica al manuscrito. Doy las gracias a los suscriptores de pago de mi página `https://tidyfirst.substack.com/` por regalarme tiempo para escribir, y por sus comentarios sobre los capítulos a medida que los iba redactando.

Tengo mucho que agradecer al experto equipo de producción de O'Reilly, que me facilitó al máximo el proceso: Melissa Duffield, Michele Cronin y Louise Corrigan. Gracias a Tim O'Reilly por arriesgarse con un libro corto.

Gracias a Keith Adams y Pamela Vagata por las charlas técnicas, los ánimos y alguno que otro cóctel. Agradezco a Susan la mezcla perfecta de ánimo y pequeños empujoncitos. Gracias a mis hijos, Beth, Lincoln, Lindsey, Forrest y Joëlle.

Doy las gracias a mis mentores y colegas en el diseño de software: Ward Cunningham, Martin Fowler, Ron Jeffries, Erich Gamma, David Saff y Massimo Arnoldi.

Por último, agradezco a Ed Yourdon (honramos su memoria) y Larry Constantine por haber descubierto todo esto hace tanto tiempo.

Sobre el autor

Kent Beck es programador, creador de la programación extrema, pionero de los patrones de diseño de software, coautor de JUnit, redescubridor del desarrollo guiado por pruebas y observador de las 3X: explorar, expandir y extraer. Beck es también alfabéticamente el primer firmante del Manifiesto Ágil. Vive en San Francisco, California, y es científico jefe en Mechanical Orchard, donde se dedica a enseñar a *geeks* habilidades para que se sientan seguros en el mundo.

Los lectores pueden ponerse en contacto con él mediante estas opciones:

- Facebook: `https://www.facebook.com/kentlbeck`.
- Twitter: `https://twitter.com/KentBeck`.
- LinkedIn: `https://www.linkedin.com/in/kentbeck`.
- Medium: `https://medium.com/@kentbeck_7670`.
- Sitio web: `https://www.kentbeck.com`.

Índice de contenidos

Supongamos que tienes una función grande que contiene muchas líneas de código. Antes de cambiarla, lees el código para entender lo que está pasando. En el proceso, vislumbras la manera de dividir de una forma lógica el código en fragmentos más pequeños. Al extraer estas partes, estás limpiando el código. Otros preceptos de ordenación incluyen el uso de cláusulas de guarda, comentarios aclaratorios y funciones auxiliares.

En este libro se pone en práctica lo que en él se propone, es decir, presentar estas normas de limpieza en capítulos breves y sugerir cuándo y dónde se pueden aplicar. Así, en lugar de intentar dominar la ordenación del código de golpe, este libro permite probar algunos ejemplos que tienen sentido para el problema que se desea resolver. También empieza describiendo la teoría del diseño de software: acoplamiento, cohesión, flujos de fondos descontados y opcionalidad.

A quién está destinado este libro

Este libro está dirigido a programadores, desarrolladores, arquitectos de software y directores técnicos. No está vinculado a ningún lenguaje de programación, y todos los desarrolladores podrán aplicar los conceptos de este libro a sus propios proyectos. En el libro se da por sentado que el lector no es nuevo en la programación en general.

Qué aprenderás en este libro

Al final de la lectura del libro, el lector comprenderá:

- Las diferencias fundamentales entre los cambios en el comportamiento de un sistema y los cambios en su estructura.
- La magia propicia de alternar inversión en estructura e inversión en comportamiento, como un programador solitario que cambia código.
- La teoría básica del funcionamiento del diseño de software y las fuerzas que actúan sobre él.

También será capaz de lo siguiente:

- Mejorar su propia experiencia de programación ordenando unas veces primero y otras veces después.
- Empezar a realizar cambios de envergadura en pequeños pasos seguros.
- Prepararse para el diseño como una actividad humana con incentivos divergentes.

Estructura del libro

Este libro está dividido en una introducción y tres partes.

- **Introducción:** Empiezo con una breve descripción de mis motivaciones para escribir este libro, cómo llegué a escribirlo, a quién va dirigido y qué puede esperar el lector. Después entramos en materia.
- **Parte I. "Preceptos de ordenación":** Una ordenación es como una pequeña refactorización en miniatura. Cada capítulo corto es una norma de ordenación. Si ves código así, entonces cámbialo por código asá. Después lo envías a producción.
- **Parte II. "Gestionar el orden":** Después hablamos de la gestión del proceso de ordenación. Parte de la filosofía de la limpieza de código es que nunca debe ser un gran problema. Nunca debe ser algo que deba ser informado, rastreado, planificado y programado. Hay que cambiar este código, y es difícil porque está desordenado, así que primero se ordena. Incluso como parte del trabajo diario, sigue siendo un proceso que mejora con la reflexión.
- **Parte III. "Teoría":** Aquí es donde por fin logro desplegar mis alas y profundizar en los temas que me emocionan. ¿Qué quiero decir con "el diseño de software es un ejercicio de relaciones humanas"? ¿Por qué el software cuesta tantísimo? ¿Qué podemos hacer al respecto (alerta de *spoiler*: diseño de software)? ¿Acoplamiento? ¿Cohesión? ¿Leyes potenciales?

Mi objetivo es que los lectores empiecen a leer por la mañana y diseñen mejor por la tarde y, a partir de entonces, que diseñen un poquito mejor cada día. Muy pronto, el diseño de software ya no será el eslabón más débil de la cadena de aportación de valor con ayuda de software.

¿Por qué diseño de software "empírico"?

Los debates más acalorados en diseño de software parecen girar en torno a qué diseñar:

- ¿Qué tamaño deben tener los servicios?
- ¿Qué tamaño deben tener los repositorios?
- Eventos frente a invocación explícita de servicios.
- Objetos frente a funciones frente a código imperativo.

Estos debates sobre el qué ocultan un desacuerdo más sustancial entre los diseñadores de software: ¿cuándo? Esta es una caricatura de los extremos de este desacuerdo:

- **Diseño especulativo:** Sabemos lo que queremos hacer a continuación, así que diseñemos hoy para conseguirlo. Será más barato diseñar ahora. Además, una vez que el software está en producción, nunca tendremos la oportunidad de diseñar, así que hagámoslo todo hoy.
- **Diseño reactivo:** Las funcionalidades son lo único que le importa a la gente, de modo que diseñemos lo menos posible hoy para poder volver a las funcionalidades. Solo cuando sea casi imposible añadir más, mejoraremos el diseño a regañadientes, y en ese momento haremos lo justo para volver a ellas.

Aspiro a responder a la pregunta de "¿cuándo?" con "en algún punto intermedio". Cuando observamos que una determinada clase de funcionalidades es difícil de agregar, diseñamos hasta que se alivia la presión. Empezamos con el diseño justo para poner en marcha los ciclos de retroalimentación:

- **Funcionalidades:** ¿Qué quieren los usuarios?
- **Diseño:** ¿Cuál es la mejor manera de ayudar a los programadores a ofrecer esas funcionalidades?

La respuesta del diseño de software a la pregunta "¿cuándo?" está supeditada. Diseña en el momento en que puedas sacar partido del diseño. Responder a esta pregunta requiere gusto, negociación y juicio. ¿Requerir gusto y juicio es una debilidad? Claro, pero es una debilidad inevitable. Los diseños especulativo y reactivo también requieren juicio, pero les dan a los diseñadores de software menos herramientas con las que trabajar.

Me gusta la palabra "empírico" para describir este estilo porque parece aclarar la distinción que estoy haciendo sincronizando los diseños especulativo y reactivo. "Basado en, preocupado por, o verificable por observación o experiencia más que por teoría o lógica pura". Suena bastante bien.

¿Cómo se me ocurrió escribir este libro?

Cuando era estudiante, asistí a un curso sobre diseño de software en el que se utilizaba el libro *Structured Design* de Ed Yourdon (*requiescat in pace*) y Larry Constantine. No entendí mucho del libro, en parte porque todavía no me había encontrado con los problemas que trataba.

Avancemos 25 años hasta 2005. Ya había diseñado un montón de aplicaciones para entonces. Tenía la sensación de dominar bastante bien el diseño. Stephen Fraser organizó una mesa redonda en la gran conferencia OOP-SLA sobre programación orientada a objetos para celebrar el 30 aniversario de la publicación del libro. Ed y Larry iban a estar allí, junto con Rebecca Wirfs-Brock, Grady Booch, Steve McConnell y Brian Henderson-Sellers.

Si yo no quería salir disparado de allí, tenía trabajo que hacer, de modo que abrí mi copia del libro, cuyas hojas ya amarilleaban, y empecé a leer. Horas después, levanté la vista, absolutamente cautivado. Aquí estaban las leyes del movimiento de Newton, pero para diseño de software. Estaba todo muy claro cuando salió. ¿Cómo es que nosotros, como industria, hemos olvidado esa claridad?

Recuerdo que el debate fue bien. Un momento culminante de la conferencia fue el desayuno con Ed y Larry, dos tipos extremadamente brillantes que se sentían totalmente cómodos consigo mismos y con los demás. La figura P.1 muestra las firmas que dejaron en mi copia del libro en aquel entonces.

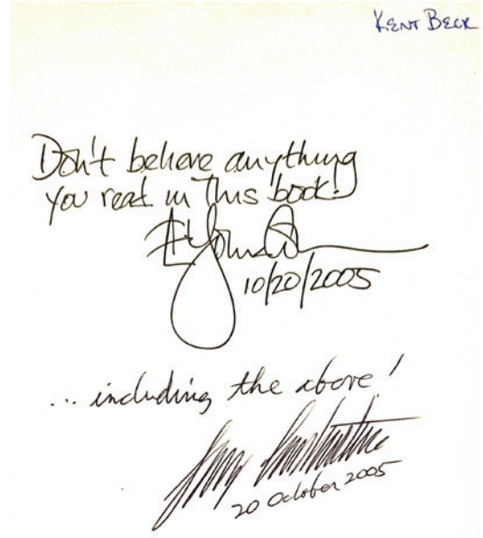

Figura P.1. Inscripciones de Ed Yourdon ("Don't believe anything you read in this book!", "¡No creas todo lo que leas en este libro!") y Larry Constantine ("...including the above!", "... ¡incluyendo lo de arriba!").

Por aquel entonces, el libro ya estaba anticuado. Los ejemplos con cinta perforada y magnética ya no eran relevantes. Ni lo era la discusión sobre el lenguaje ensamblador frente a los nuevos lenguajes de alto nivel. Sin embargo, los conceptos básicos eran correctos, así que hice la promesa de que pondría ese material al alcance del público actual.

Realicé varios intentos frustrados de escribir un libro sobre diseño de software en los años intermedios (busca "*Kent Beck Responsive Design*" si quieres ver en qué andaba). En 2019, inesperadamente, tuve dos semanas de tiempo sin nada planificado y, por lo tanto, decidí ver cuánto del libro podría escribir en esas dos semanas.

Diez mil palabras después, había aprendido una importante lección: no iba a ser capaz de abordar todo el diseño de software en un solo libro. Una situación que seguía apareciendo en lo que había redactado era este momento del diseño a pequeña escala: tengo código desordenado; ¿lo cambio o lo limpio primero?

Mi experiencia escribiendo libros ha sido siempre así. Elegir un tema que parece demasiado poco para un libro. Escribir. Descubrir que el tema es demasiado grande para un solo libro. Tomar una parte demasiado pequeña. Escribir. Descubrir que el fragmento es demasiado grande. Repetir.

CAPÍTULO 6

Orden de cohesión

Lees el código, te das cuenta de que para hacer un cambio de comportamiento vas a tener que cambiar varios puntos muy dispersos en el código, y te pones de mal humor. ¿Qué deberías hacer?

Reordena el código para que los elementos que necesitas cambiar sean adyacentes. El orden de cohesión funciona para las rutinas de un archivo: si dos rutinas están acopladas, colócalas una al lado de la otra. También funciona para archivos de directorios: si dos archivos están conectados, ponlos en el mismo directorio. Incluso funciona a través de repositorios: coloca el código acoplado en el mismo repositorio antes de cambiarlo.

Y ¿por qué no eliminar simplemente el acoplamiento? Si sabes cómo hacerlo, adelante. Es la mejor ordenación de todas, suponiendo que:

coste(desacoplamiento) + coste(cambio) < coste(acoplamiento) + coste(cambio)

Quizá no sea factible por distintas razones:

- El desacoplamiento puede ser un esfuerzo intelectual (no sabes cómo hacerlo).
- El desacoplamiento puede suponer un esfuerzo de tiempo y dinero (podrías hacerlo, pero justo ahora no puedes permitirte dedicarle ese tiempo).
- El desacoplamiento puede suponer también un esfuerzo para las relaciones (el equipo ya ha soportado todos los cambios que puede manejar).

No es que estés atascado con cambios del tipo modelo de queso suizo. Ordenar puede aumentar la cohesión lo suficiente como para facilitar los cambios de comportamiento. En ocasiones, la mayor claridad obtenida con una cohesión ligeramente mejor desbloquea lo que sea que te está impidiendo hacer el desacoplamiento. A veces una mejor cohesión ayuda a aceptar el acoplamiento.

CAPÍTULO 16

Separar las ordenaciones

Supondremos por el momento que utilizas un modelo de *pull request* (PR, solicitud de incorporación de cambios) o revisión de código (hablaremos más tarde de una alternativa). ¿En qué punto realizas las limpiezas?

Estamos ante un feo caso de bucle infinito:

1. Pongo mis ordenaciones junto con mis cambios de comportamiento.
2. Los colaboradores se quejan de que mis *pull requests* son demasiado largas.
3. Separo las ordenaciones en sus propias solicitudes de incorporación de cambios, antes (lo más probable) o después de los cambios de comportamiento.
4. Los colaboradores se quejan de que las solicitudes PR de las ordenaciones no tienen sentido.
5. Vuelvo al paso 1.

Las ordenaciones tienen que ir en algún lugar, en caso contrario no puedes ordenar. ¿Dónde puedes colocarlas? Resumiendo: en sus propias PR, con las mínimas ordenaciones por solicitud como sea posible.

Analicemos las ventajas y desventajas con más detenimiento. La gente que está aprendiendo a ordenar parece pasar por fases predecibles. En la primera fase solo estamos haciendo cambios, y empezamos con un conjunto no diferenciado de los mismos (figura 16.1).

En este momento estamos arreglando una sentencia `if`, nos damos cuenta de que un nombre está mal, lo corregimos y volvemos a la sentencia `if`. El cambio es el cambio.

Tras aprender a hacer ordenaciones, es como si la imagen que vemos en el microscopio se enfocara de repente. Algunos de esos cambios estaban modificando el comportamiento del programa y sus atributos, observados desde la ejecución del programa. Sin embargo, otros, que solamente pueden observarse mirando el código, estaban cambiando la estructura del programa: C=comportamiento, E=estructura (figura 16.2).

¿Ordenar primero?

El código desordenado es un fastidio. "Limpiar" el código, para que sea más legible, requiere dividirlo en fragmentos manejables. El autor de esta guía práctica, Kent Beck, creador de la programación extrema y pionero de los patrones de diseño de software, sugiere cuándo y dónde podemos aplicar limpiezas u ordenaciones para mejorar el código, teniendo en mente al mismo tiempo la estructura general del sistema.

En lugar de intentar dominar la ordenación del código de golpe, este libro permite probar algunos ejemplos que tienen sentido para el problema que se desee resolver.

Si disponemos de una función grande que contiene muchas líneas de código, aquí aprenderemos a dividirla de una manera lógica en fragmentos más pequeños. En paralelo aprenderemos la teoría del diseño de software: acoplamiento, cohesión, flujos de fondos descontados y opcionalidad.

Con este libro lograrás:

- Comprender la teoría básica del funcionamiento del diseño de software y las fuerzas que actúan sobre él.
- Explorar la diferencia entre los cambios en el comportamiento de un sistema y los cambios en su estructura.
- Mejorar la experiencia de programación ordenando a veces primero y a veces después.
- Aprender a realizar cambios de envergadura en pequeños pasos seguros.
- Abordar el diseño de software como un ejercicio de relaciones humanas.

Kent Beck, creador de la programación extrema, es pionero de los patrones de diseño de software, coautor de JUnit, redescubridor del desarrollo guiado por pruebas y observador de las 3X: explorar, expandir y extraer. Alfabéticamente, es el primer firmante del Manifiesto Ágil. Kent vive en San Francisco, California, y es científico jefe en Mechanical Orchard, donde se dedica a enseñar a *geeks* habilidades para que se sientan seguros en el mundo.

«El diseño trata de las formas que pintamos con nuestro código, y Kent nos ayuda a dibujar mejores formas. Es un libro importante sobre un tema importante».

—Dave Farley
Fundador y director de Continuous Delivery Ltd.

«Este libro ofrece consejos prácticos para ayudar a cualquier desarrollador a mejorar el código con el que trabaja».

—Sam Newman
Consultor independiente, tecnólogo y autor de *Building Microservices* y *Monolith to Microservices.*

«Las ideas de este libro son sencillas, pero te hacen preguntarte por qué no habías pensado en muchas de ellas antes. Recomendado para cualquiera al que le preocupe crear un código limpio y legible».

—Gergely Orosz
The Pragmatic Engineer

ANAYA MULTIMEDIA
O'REILLY®
www.anayamultimedia.es
ISBN 978-84-415-5015-5
2315262