

The Pragmatic Programmer



from journeyman
to master

Andrew Hunt
David Thomas

Preparado exclusivamente para Zach

Lo que otros en las trincheras dicen sobre *El programador pragmático*. .

"Lo bueno de este libro es que es genial para mantener fresco el proceso de programación. *[El libro]* te ayuda a seguir creciendo y claramente viene de personas que han pasado por eso".

- Kent Beck, autor de *Extreme Programming Explained: Embrace Change*

"¡Descubrí que este libro es una gran mezcla de consejos sólidos y analogías maravillosas!"

- Martin Fowler, autor de *Refactoring* y *UML Distilled*

"Compraba una copia, la leía dos veces y luego les decía a todos mis colegas que salieran corriendo a buscar una copia. Este es un libro que nunca prestaría porque me preocuparía que se perdiera".

- Kevin Ruland, Ciencias de la Gestión, MSG-Logistics

"La sabiduría y la experiencia práctica de los autores es evidente. Los temas presentados son relevantes y útiles. Con mucho, su mayor

Para mí, la fuerza ha sido las analogías sobresalientes: balas trazadoras, ventanas rotas y la fabulosa explicación basada en helicóptero de la necesidad de ortogonalidad, especialmente en una situación de crisis. Tengo pocas dudas de que este libro acabará convirtiéndose en una excelente fuente de información útil tanto para los programadores como para los mentores expertos".

- John Lakos, autor de *Large-Scale C++ Software Design*

"Este es el tipo de libro del que compraré una docena de copias cuando salga para poder dárselo a mis clientes".

► Eric Vought, ingeniero de software

"La mayoría de los libros modernos sobre desarrollo de software no cubren los conceptos básicos de lo que hace a un gran desarrollador de software, sino que dedican su tiempo a la sintaxis o la tecnología, donde en realidad la mayor ventaja posible para cualquier equipo de software es tener desarrolladores talentosos que realmente conozcan bien su oficio. Un libro excelente".

► Pete McBreen, Consultor Independiente

"Desde que leí este libro, he puesto en práctica muchas de las sugerencias y consejos prácticos que contiene. En general, han ahorrado tiempo y dinero a mi empresa y me han ayudado a hacer mi trabajo más rápido. Esto debería ser una referencia de escritorio para todos los que trabajan con código para ganarse la vida".

► Jared Richardson, desarrollador de software sénior,
iRenaissance, Inc.

"Me gustaría que esto se emitiera a todos los nuevos
empleados de mi empresa. " "

► Chris Cleeland, ingeniero de software
sénior, Object Computing, Inc.

El programador pragmático

Esta página se ha dejado en blanco intencionadamente

El programador pragmático

De oficial a maestro

Andrew Hunt
David Thomas



ADDISON-WESLEY
Un sello de Addison Wesley Longman, Inc.

Reading, Massachusetts Harlow, Inglaterra Menlo Park, California
Berkeley, California Don Mills, Ontario Sydney
Bonn Ámsterdam Tokio Ciudad de México

Muchas de las denominaciones utilizadas por los fabricantes y vendedores para distinguir sus productos se reivindican como marcas comerciales. En los casos en que esas designaciones aparecen en este libro, y Addison-Wesley estaba al tanto de una reclamación de marca registrada, las designaciones se han impreso en letras mayúsculas iniciales o en mayúsculas.

La letra de la canción "The Boxer" en la página 157 es Copyright © 1968 Paul Simon. Usado con permiso del editor: Paul Simon Music. La letra de la canción "Alice's Restaurant" en la página 220 © de Arlo Guthrie, c 1966, 1967 (renovado) por APPLESEED MUSIC INC. Todos los derechos reservados. Usado con permiso.

Los autores y el editor han tenido cuidado en la preparación de este libro, pero no ofrecen ninguna garantía expresa o implícita de ningún tipo y no asumen ninguna responsabilidad por errores u omisiones. No se asume ninguna responsabilidad por daños incidentales o consecuentes en relación con o que surjan del uso de la información o los programas contenidos en este documento.

La editorial ofrece descuentos en este libro cuando se pide en cantidad para ventas especiales. Para obtener más información, póngase en contacto con:

Ventas directas de AWL
Addison Wesley Longman, Inc. A
la manera de Jacob
Reading, Massachusetts 01867
(781) 944-3700

Visite AWL en la Web: www.awl.com/cseng

Library of Congress Cataloging-in-Publication Data

Hunt, Andrew, 1964 –
El programador pragmático / Andrew Hunt, David Thomas.
p. cm.
Incluye referencias bibliográficas.
ISBN 0-201-61622-X
1. Computadora programación. I. Thomas, David, 1956-.
II. Título.
QA76.6.H857 1999
005.1--dc21
99-43581
CIP

Derechos de autor © 2000 por Addison Wesley Longman, Inc.

Todos los derechos reservados. Ninguna parte de esta publicación puede ser reproducida, almacenada en un sistema de recuperación o transmitida, en ninguna forma ni por ningún medio, electrónico, mecánico, fotocopia, grabación o de otro tipo, sin el permiso previo por escrito del editor. Impreso en los Estados Unidos de América. Publicado simultáneamente en Canadá.

ISBN 0-201-61622-X

Texto impreso en los Estados Unidos en papel reciclado en Courier Stoughton en Stoughton, Massachusetts.

25ª Impresión Febrero 2010

Para Ellie y Juliet,
Elizabeth y Zachary,
Stuart y Henry

Esta página se ha dejado en blanco intencionadamente

Contenido

PREFACIO	Xiii
PREFACIO	XVII
1 A PRAGMÁTICA PHILÓSOFÍA	1
1. El gato se comió mi fuente Código.....	2
2. Entropía del software.....	4
3. Sopa de piedra y ranas hervidas	7
4. Software lo suficientemente bueno	9
5. Su conocimiento Cartera	12
6. ¡Comunicar!	18
2 A PRAGMÁTICA APRACHÓN	25
7. Los males de la duplicación	26
8. Ortogonalidad	34
9. Reversibilidad	44
10. Balas trazadoras	48
11. Prototipos y notas adhesivas	53
12. Idiomas de dominio	57
13. Estimar	64
3 LA BASIC HERRAMIENTAS	71
14. El poder del texto sin formato	73
15. Juegos de conchas.....	77
16. Edición de potencia.....	82
17. Control de código fuente	86
18. Depuración	90
19. Manipulación de texto	99
20. Generadores de código	102

4 PRAGMÁTICO PARANOIA	107
21. Diseño por Contrato.....	109
22. Los programas muertos no dicen mentiras	120
23. Programación Asertiva	122
24. Cuándo usar excepciones	125
25. Cómo equilibrar los recursos.....	129
5 DOBLAR o QUEBRAR	137
26. El desacoplamiento y la ley de Deméter	138
27. Metaprogramación	144
28. Temporal Acoplamiento.....	150
29. Es solo una vista.....	157
30. Pizarras	165
6 MIENTRAS You SON CEL ATRAVESAMIENTO	171
31. Programación por Coincidencia.....	172
32. Algoritmo Velocidad.....	177
33. Refactorización	184
34. Código que es fácil de probar	189
35. Magos malvados	198
7 ANTES el PROYECTO	201
36. El pozo de los requisitos.....	202
37. Resolviendo acertijos imposibles	212
38. No hasta que estés listo	215
39. La trampa de las especificaciones	217
40. Círculos y flechas.....	220
8 PRAGMÁTICO PPROJECTOS	223
41. Equipos pragmáticos	224
42. Automatización ubicua.....	230
43. Pruebas despiadadas.....	237
44. Todo es escritura.....	248
45. Grandes expectativas	255
46. Orgullo y prejuicio	258

Apéndices

A RECURSOS	261
Profesional Sociedades	262
Construcción de una biblioteca	262
Internet Recursos	266
Bibliografía	275
B ADE MANERA EJERCICIOS	279
ÍNDICE	309

Esta página se ha dejado en blanco intencionadamente

Prefacio

Como crítico, tuve la oportunidad temprana de leer el libro que usted tiene en sus manos. Fue genial, incluso en forma de borrador. Dave Thomas y Andy Hunt tienen algo que decir, y saben cómo decirlo. Vi lo que estaban haciendo y supe que funcionaría. Pedí que me permitieran escribir este prólogo para poder explicar por qué.

En pocas palabras, este libro te dice cómo programar de una manera que puedas seguir. No pensarías que eso sería algo difícil de hacer, pero lo es. ¿Por qué? Por un lado, no todos los libros de programación están escritos por programadores. Muchos son compilados por diseñadores de lenguaje o por los periodistas que trabajan con ellos para promocionar sus creaciones. Esos libros te dicen cómo *hacer talk* en un lenguaje de programación, lo cual es ciertamente importante, pero eso es solo una pequeña parte de lo que hace un programador.

¿Qué hace un programador además de hablar en lenguaje de programación? Bueno, ese es un tema más profundo. La mayoría de los programadores tendrían problemas para explicar lo que hacen. La programación es un trabajo lleno de detalles, y hacer un seguimiento de esos detalles requiere concentración. Pasan las horas y aparece el código. Miras hacia arriba y ahí están todas esas declaraciones. Si no piensas detenidamente, podrías pensar que la programación es simplemente escribir instrucciones en un lenguaje de programación. Estarías equivocado, por supuesto, pero no podrías saberlo mirando alrededor de la sección de programación de la librería.

En *The Pragmatic Programmer*, Dave y Andy nos cuentan cómo programar de una manera que podamos seguir. ¿Cómo se volvieron tan inteligentes? ¿No están tan centrados en los detalles como otros programadores? La respuesta es que prestaron atención a lo que estaban haciendo mientras lo hacían, y luego trataron de hacerlo mejor.

Imagina que estás sentado en una reunión. Tal vez estés pensando que la reunión podría durar para siempre y que preferirías estar programando. Dave y Andy estarían pensando por qué estaban

tener la reunión, y preguntarse si hay algo más que podrían hacer que tomaría el lugar de la reunión, y decidir si ese algo podría automatizarse para que el trabajo de la reunión simplemente ocurra en el futuro. Entonces lo harían.

Esa es la forma en que Dave y Andy piensan. Esa reunión no fue algo que les impidiera programar. Es una programación. Y era una programación que se podía mejorar. Sé que piensan de esta manera porque es el consejo número dos: Piensa en tu trabajo.

Así que imagínate que estos chicos están pensando de esta manera durante unos años. Muy pronto tendrían una colección de soluciones. Ahora imagínalos usando sus soluciones en su trabajo durante unos años más, y descartando las que son demasiado difíciles o que no siempre producen resultados. Bueno, ese enfoque casi define *pragmatic*. Ahora imagínate que tardan uno o dos años más en escribir sus soluciones. Se podría pensar: "*That information would be a gold mine*". Y tendrías razón.

Los autores nos cuentan cómo programan. Y nos lo dicen de una manera que podemos seguir. Pero hay más en esta segunda afirmación de lo que se podría pensar. Déjame explicarte.

Los autores han tenido cuidado de evitar proponer una teoría del desarrollo de software. Esto es una suerte, porque si lo hubieran hecho, se verían obligados a deformar cada capítulo para defender su teoría. Tal deformación es la tradición en, por ejemplo, las ciencias físicas, donde las teorías eventualmente se convierten en leyes o se descartan silenciosamente. La programación, por otro lado, tiene pocas (o ninguna) leyes. Por lo tanto, los consejos de programación formados en torno a las leyes de lo que se quiere ser pueden sonar bien por escrito, pero no satisfacen en la práctica. Esto es lo que falla en tantos libros de metodología.

He estudiado este problema durante una docena de años y he encontrado lo más prometedor en un dispositivo llamado *patterw language*. En resumen, un *patterw* es una solución, y un lenguaje de patrones es un sistema de soluciones que se refuerzan mutuamente. Se ha formado toda una comunidad en torno a la búsqueda de estos sistemas.

Este libro es más que una colección de consejos. Es un lenguaje de patrones en piel de oveja. Digo esto porque cada consejo se extrae de la experiencia, se cuenta como un consejo concreto y se relaciona con otros para formar un sistema. Estas son las características que nos permiten aprender y seguir un lenguaje de patrones. Funcionan de la

misma manera aquí.

Puedes seguir los consejos de este libro porque es concreto. No encontrarás abstracciones vagas. Dave y Andy escriben directamente para ti, como si cada consejo fuera una estrategia vital para dinamizar tu carrera de programación. Lo hacen simple, cuentan una historia, usan un toque ligero y luego lo siguen con respuestas a las preguntas que surgirán cuando lo intentes.

Y hay más. Despues de leer diez o quince consejos, comenzarás a ver una dimensión adicional en el trabajo. A veces lo llamamos *QWAN*, abreviatura de *quality without a name*. El libro tiene una filosofía que rezumará en tu conciencia y se mezclará con la tuya propia. No predica. Simplemente dice lo que funciona. Pero en el relato se hace más. Esa es la belleza del libro: encarna su filosofía, y lo hace sin pretensiones.

Así que aquí está: un libro fácil de leer y usar sobre toda la práctica de la programación. He seguido y sigo hablando de por qué funciona. Probablemente solo te importe que funcione. Lo hace. Ya lo verás.

—Ward Cuwwiwgham

Esta página se ha dejado en blanco intencionadamente

Prefacio

Este libro te ayudará a convertirte en un mejor programador.

No importa si eres un desarrollador solitario, un miembro de un gran equipo de proyecto o un consultor que trabaja con muchos clientes a la vez. Este libro te ayudará, como individuo, a hacer un mejor trabajo. Este libro no es teórico, nos concentraremos en temas prácticos, en usar su experiencia para tomar decisiones más informadas. La palabra *pragmatic* proviene del latín *pragmaticus*, "hábil en los negocios", que a su vez se deriva del griego *πράττειν*, que significa "hacer". Este es un libro sobre el hacer.

La programación es un oficio. En su forma más simple, se reduce a hacer que una computadora haga lo que usted quiere que haga (o lo que su usuario quiere que haga). Como programador, eres en parte oyente, en parte asesor, en parte intérprete y en parte dictador. Intentas capturar requisitos elusivos y encontrar una manera de expresarlos para que una mera máquina pueda hacerles justicia. Intentas documentar tu trabajo para que los demás puedan entenderlo, y tratas de diseñar tu trabajo para que otros puedan basarse en él. Es más, intentas hacer todo esto contra el incesante tic-tac del reloj del proyecto. Haces pequeños milagros todos los días.

Es un trabajo difícil.

Hay muchas personas que te ofrecen ayuda. Los proveedores de herramientas promocionan los modelos que ofrecen sus productos. Los gurús de la metodología prometen que sus técnicas garantizan resultados. Todo el mundo afirma que su lenguaje de programación es el mejor, y que cada sistema operativo es la respuesta a todos los males imaginables.

Por supuesto, nada de esto es cierto. No hay respuestas fáciles. No existe la *mejor* solución, ya sea una herramienta, un lenguaje o un sistema operativo. Sólo puede haber sistemas que sean más apropiados en un conjunto particular de circunstancias.

Aquí es donde entra en juego el pragmatismo. No se debe estar casado con ninguna tecnología en particular, pero tener una formación y una base de experiencia lo suficientemente amplias como para permitirle elegir buenas soluciones en situaciones particulares. Su formación se deriva de la comprensión de los principios básicos de la informática, y su experiencia proviene de una amplia gama de proyectos prácticos. La teoría y la práctica se combinan para hacerte fuerte.

Ajusta su enfoque para adaptarse a las circunstancias y al entorno actuales. Juzgas la importancia relativa de todos los factores que afectan a un proyecto y utilizas tu experiencia para producir soluciones adecuadas. Y lo haces continuamente a medida que avanza el trabajo. Los programadores pragmáticos hacen el trabajo, y lo hacen bien.

¿Quién debería leer este libro?

Este libro está dirigido a personas que quieren convertirse en programadores más efectivos y productivos. Tal vez te sientas frustrado porque no parece estar alcanzando tu potencial. Tal vez te fijes en colegas que parecen estar usando herramientas para ser más productivos que tú. Tal vez tu trabajo actual utiliza tecnologías más antiguas y quieres saber cómo se pueden aplicar las ideas más nuevas a lo que haces.

No pretendemos tener todas (o incluso la mayoría) de las respuestas, ni todas nuestras ideas son aplicables en todas las situaciones. Todo lo que podemos decir es que si sigues nuestro enfoque, ganarás experiencia rápidamente, tu productividad aumentará y tendrás una mejor comprensión de todo el proceso de desarrollo. Y escribirás mejor software.

¿Qué hace a un programador pragmático?

Cada desarrollador es único, con fortalezas y debilidades individuales, preferencias y disgustos. Con el tiempo, cada uno creará su propio entorno personal. Ese entorno reflejará la individualidad del programador con la misma fuerza que sus aficiones, su ropa o su corte de pelo. Sin embargo, si eres un programador pragmático, compartirás muchas de las siguientes características:

- Adoptador pionero/adaptador rápido. Tienes instinto para las tecnologías y las técnicas, y te encanta probar cosas. Cuando se le

da algún-

algo nuevo, puedes captarlo rápidamente e integrarlo con el resto de tus conocimientos. Tu confianza nace de la experiencia.

- Inquisitivo. Tiendes a hacer preguntas. *¿Qué es lo que hiciste? ¿Tuviste problemas con la biblioteca? ¿De qué es este BeOS del que he hablado? ¿Cuántas cosas simbólicas se han implementado?* Eres una rata de manada por pequeños hechos, cada uno de los cuales puede afectar algunas decisiones dentro de unos años.
- Pensador crítico. Rara vez se dan las cosas por sentadas sin antes conocer los hechos. Cuando los colegas dicen "porque así es como se hace", o un proveedor promete la solución a todos sus problemas, huele un desafío.
- Realista. Tratas de entender la naturaleza subyacente de cada problema que enfrentas. Este realismo te da una buena idea de lo difíciles que son las cosas y de cuánto tiempo tardarán las cosas. Entender por ti mismo que un proceso *debe* ser difícil o *que debe* tardar un tiempo en completarse te da la resistencia para seguir haciéndolo.
- Todólogo. Se esfuerza por familiarizarse con una amplia gama de tecnologías y entornos, y trabaja para mantenerse al tanto de los nuevos desarrollos. Aunque tu trabajo actual requiera que seas un especialista, siempre podrás avanzar hacia nuevas áreas y nuevos desafíos.

Hemos dejado las características más básicas para el final. Todos los programadores pragmáticos las comparten. Son lo suficientemente básicos como para indicar como consejos:

CONSE
Preocúpate por tu oficio

Creemos que no tiene sentido desarrollar software a menos que te preocunes por hacerlo bien.

CONSE
¡Pensar! Sobre su trabajo

Para ser un programador pragmático, te desafiamos a pensar en lo que estás haciendo mientras lo haces. No se trata de una auditoría puntual de las prácticas actuales, sino de una evaluación crítica continua de

cada

decisiones que tomas, todos los días y en cada desarrollo. Nunca funcione en piloto automático. Estar constantemente pensando, criticando tu trabajo en tiempo real. El viejo lema corporativo de IBM, THINK!, es el mantra del programador pragmático.

Si esto te parece un trabajo duro, entonces estás exhibiendo la *característica realista*. Esto va a ocupar algo de su valioso tiempo, tiempo que probablemente ya está bajo una tremenda presión. La recompensa es una participación más activa en un trabajo que amas, una sensación de dominio sobre una gama cada vez mayor de temas y placer en una sensación de mejora continua. A largo plazo, su inversión en tiempo se verá recompensada a medida que usted y su equipo se vuelvan más eficientes, escriban código que sea más fácil de mantener y pasen menos tiempo en las reuniones.

Pragmáticos individuales, equipos grandes

Algunas personas sienten que no hay lugar para la individualidad en equipos grandes o proyectos complejos. "La construcción de software es una disciplina de ingeniería", dicen, "que se rompe si los miembros individuales del equipo toman decisiones por sí mismos".

No estamos de acuerdo.

La construcción de software *debe* ser una disciplina de ingeniería. Sin embargo, esto no excluye la artesanía individual. Piensa en las grandes catedrales construidas en Europa durante la Edad Media. Cada uno requirió miles de años-persona de esfuerzo, repartidos a lo largo de muchas décadas. Las lecciones aprendidas se transmitieron al siguiente grupo de constructores, quienes avanzaron en el estado de la ingeniería estructural con sus logros. Pero los carpinteros, canteros, talladores y vidrieros eran todos artesanos, que interpretaban los requisitos de ingeniería para producir un todo que trascendía el aspecto puramente mecánico de la construcción. Fue su creencia en sus contribuciones individuales lo que sustentó los proyectos:

Nosotros uho cortar meros estibados deben ser ewvisionwiwg cathedrals.

— **Credo de los trabajadores de la cantera**

Dentro de la estructura general de un proyecto siempre hay espacio para la individualidad y la artesanía. Esto es particularmente cierto dado el estado actual de la ingeniería de software. Dentro de cien años, nuestra ingeniería puede parecer tan arcaica como las técnicas

utilizadas por los medievales.

Los constructores de catedrales parecen a los ingenieros civiles de hoy, mientras que nuestra artesanía seguirá siendo honrada.

Es un proceso continuo

Un turista que visitó el Etow College de Ewglawd hizo que los lauws fueran tan perfectos. "Eso es lo que está pasando", respondió, "simplemente te quitas los deu cada mañana, los mueves cada dos días, los haces rodar hasta a ue".

"¿Es así?", preguntó el turista.

—Claro que sí —respondió el guardabarros—. "Hazlo durante 500 años y también habrás a a ti mismo".

Los grandes céspedes necesitan pequeñas cantidades de cuidado diario, al igual que los grandes programadores. A los consultores de gestión les gusta soltar la palabra *kaizew* en las conversaciones. "Kaizen" es un término japonés que capta el concepto de hacer continuamente muchas pequeñas mejoras. Se consideró que era una de las principales razones de los dramáticos aumentos en productividad y calidad en la fabricación japonesa y fue ampliamente copiado en todo el mundo. El kaizen también se aplica a las personas. Trabaja todos los días para refinar las habilidades que tienes y agregar nuevas herramientas a tu repertorio. A diferencia de los céspedes de Eton, comenzará a ver resultados en cuestión de días. A lo largo de los años, te sorprenderá cómo ha florecido tu experiencia y cómo han crecido tus habilidades.

Cómo está organizado el libro

Este libro está escrito como una colección de secciones cortas. Cada sección es independiente y aborda un tema en particular. Encontrarás numerosas referencias cruzadas, que ayudan a poner cada tema en contexto. Siéntete libre de leer las secciones en cualquier orden, este no es un libro que necesites leer de principio a fin.

De vez en cuando te encontrarás con un recuadro con la etiqueta "*Consejo ww*" (como el Consejo 1, "Preocúpate por tu oficio" en la página xix). Además de enfatizar puntos en el texto, sentimos que los consejos tienen vida propia: los vivimos a diario. Encontrarás un resumen de todos los consejos en una tarjeta extraíble dentro de la

contraportada.

El Apéndice A contiene un conjunto de recursos: la bibliografía del libro, una lista de direcciones URL a recursos web y una lista de publicaciones periódicas, libros y organizaciones profesionales recomendadas. A lo largo del libro encontrará referencias a la bibliografía y a la lista de URLs, como [KP99] y [URL 18], respectivamente.

Hemos incluido ejercicios y desafíos cuando corresponde. Los ejercicios normalmente tienen respuestas relativamente sencillas, mientras que los desafíos son más abiertos. Para que te hagas una idea de lo que pensamos, hemos incluido nuestras respuestas a los ejercicios en el Apéndice B, pero muy pocos tienen una única *solución correcta*. Los desafíos pueden formar la base de discusiones grupales o trabajos de ensayo en cursos avanzados de programación.

¿Qué hay en un nombre?

"Vaya, yo uso a uord", dijo Humpty Dumpty, y a tono abrasador, "es simplemente que lo elijo para que mea, ya sea más o menos".

► Lewis Carroll, a través del espejo

Dispersos a lo largo del libro encontrarás varios fragmentos de jerga, ya sea palabras en inglés perfectamente buenas que han sido corrompidas para significar algo técnico, o horrendas palabras inventadas a las que los científicos de la computación les han asignado significados con rencor contra el idioma. La primera vez que usamos cada una de estas palabras de la jerga, tratamos de definirla, o al menos dar una pista de su significado. Sin embargo, estamos seguros de que algunos han caído en el olvido, y otros, como *object* y *relational database*, son de uso lo suficientemente común como para que agregar una definición sea aburrido. Si te encuentras con un término que no has visto antes, por favor, no te lo saltes. Tómese el tiempo para buscarlo, tal vez en la Web, o tal vez en un libro de texto de ciencias de la computación. Y, si tienes la oportunidad, envíanos un correo electrónico y quéjate, para que podamos agregar una definición a la próxima edición.

Dicho todo esto, decidimos vengarnos de los informáticos. A veces, hay palabras de jerga perfectamente buenas para conceptos, palabras que hemos decidido ignorar. ¿Por qué? Porque la jerga existente se restringe normalmente a un dominio de problema particular, o a una fase particular del desarrollo. Sin embargo, una de las filosofías básicas de este libro es que la mayoría de las técnicas que recomendamos son

universales: la modularidad se aplica al código, los diseños, la documentación y el equipo

organización, por ejemplo. Cuando quisimos usar la palabra de la jerga convencional en un contexto más amplio, se volvió confuso: parecía que no podíamos superar el bagaje que traía consigo el término original. Cuando esto sucedió, contribuimos a la decadencia del idioma inventando nuestros propios términos.

Código fuente y otros recursos

La mayor parte del código que se muestra en este libro se extrae de archivos fuente compilables, disponibles para su descarga desde nuestro sitio web:

www.programador pragmático.COM

Allí también encontrará enlaces a recursos que nos resultan útiles, junto con actualizaciones del libro y noticias de otros desarrollos de Pragmatic Programmer.

Envíanos tus comentarios

Le agradeceríamos que nos facilitara su respuesta. Los comentarios, sugerencias, errores en el texto y problemas en los ejemplos son bienvenidos. Envíenos un correo electrónico a

ppbook@pragmaticprogrammer.COM

Reconocimientos

Cuando empezamos a escribir este libro, no teníamos ni idea de cuánto esfuerzo de equipo acabaría siendo.

Addison-Wesley ha sido brillante, tomando a un par de hackers mojados detrás de las orejas y guiándonos a través de todo el proceso de producción del libro, desde la idea hasta la copia lista para la cámara. Muchas gracias a John Wait y Meera Ravindiran por su apoyo inicial, a Mike Hendrickson, nuestro entusiasta editor (y un malvado diseñador de portadas!), Lorraine Ferrier y John Fuller por su ayuda con la producción, y a la infatigable Julie DeBaggis por mantenernos a todos juntos.

Luego estaban los críticos: Greg Andress, Mark Cheers, Chris Cleeland, Alistair Cockburn, Ward Cunningham, Martin Fowler, Thanh T. Giang, Robert L. Glass, Scott Henninger, Michael Hunter, Brian

Kirby, John Lakos, Pete McBreen, Carey P. Morris, Jared Richardson, Kevin Ruland, Eric Starr, Eric Vought, Chris Van Wyk y Deborra Zukowski. Sin sus cuidadosos comentarios y valiosas ideas, este libro sería menos legible, menos preciso y dos veces más largo. Gracias a todos por su tiempo y sabiduría.

La segunda edición de este libro se benefició enormemente de los ojos de águila de nuestros lectores. Muchas gracias a Brian Blank, Paul Boal, Tom Ekberg, Brent Fulgham, Louis Paul Hebert, Henk-Jan Olde Loohuis, Alan Lund, Gareth McCaughan, Yoshiki Shibata y Volker Wurst, ambos por encontrar los errores y por tener la gracia de señalarlos suavemente.

A lo largo de los años, hemos trabajado con un gran número de clientes progresistas, donde hemos adquirido y perfeccionado la experiencia sobre la que escribimos aquí. Recientemente, hemos tenido la suerte de trabajar con Peter Gehrke en varios proyectos de gran envergadura. Su apoyo y entusiasmo por nuestras técnicas son muy apreciados.

Este libro fue producido usando LATEX, pic, Perl, dvips, ghostview, ispell, GNU make, CVS, Emacs, XEmacs, EGCS, GCC, Java, iContract y SmallEiffel, utilizando los shells Bash y zsh en Linux. Lo asombroso es que todo este tremendo software está disponible gratuitamente. Le debemos un enorme "gracias" a los miles de Programadores Pragmáticos de todo el mundo que han contribuido con estos y otros trabajos para todos nosotros. Nos gustaría agradecer especialmente a Reto Kramer por su ayuda con iContract.

Por último, pero no menos importante, tenemos una gran deuda con nuestras familias. No solo han soportado escribir a altas horas de la noche, las enormes facturas telefónicas y nuestro aire permanente de distracción, sino que han tenido la gracia de leer lo que hemos escrito, una y otra vez. Gracias por dejarnos soñar.

*Audy Huwt
Dave Thomas*

Capítulo 1

Una filosofía pragmática

¿Qué distingue a los programadores pragmáticos? Creemos que es una actitud, un estilo, una filosofía de abordar los problemas y sus soluciones. Piensan más allá del problema inmediato, siempre tratando de situarlo en su contexto más amplio, siempre tratando de ser conscientes del panorama general. Después de todo, sin este contexto más amplio, ¿cómo se puede ser pragmático? ¿Cómo se pueden hacer concesiones inteligentes y tomar decisiones informadas?

Otra clave de su éxito es que asumen la responsabilidad de todo lo que hacen, lo que discutimos en *The Cat Ate My Source Code*. Siendo responsables, los programadores pragmáticos no se quedarán de brazos cruzados y verán cómo sus proyectos se desmoronan por negligencia. En *Software Entropy*, te contamos cómo mantener tus proyectos impecables.

A la mayoría de las personas les resulta difícil aceptar el cambio, a veces por buenas razones, a veces por pura inercia. En *Stowe Soup and Boiled Frogs*, analizamos una estrategia para instigar el cambio y (en aras del equilibrio) presentamos el cuento con moraleja de un anfibio que ignoró los peligros del cambio gradual.

Uno de los beneficios de comprender el contexto en el que trabajas es que es más fácil saber qué tan bueno tiene que ser tu software. A veces, la casi perfección es la única opción, pero a menudo hay compensaciones involucradas. Exploramos esto en *Good-Ewough Software*.

Por supuesto, es necesario tener una amplia base de conocimientos y experiencia para llevar a cabo todo esto. El aprendizaje es un proceso continuo y continuo. En *Tu Portafolio de Knowledge*, discutimos algunas estrategias para mantener el impulso.

◀ 1 ▶

Por último, ninguno de nosotros trabaja en el vacío. Todos pasamos una gran cantidad de tiempo interactuando con los demás. *Communcate!* enumera las formas en que podemos hacerlo mejor.

La programación pragmática se deriva de una filosofía de pensamiento pragmático. Este capítulo sienta las bases de esa filosofía.

El gato se comió mi código fuente

La prueba grea de all ueakwesses es la fear de appeariwg ueak.

► **J. B. Bossuet, Política de las Sagradas Escrituras, 1709**

Una de las piedras angulares de la filosofía pragmática es la idea de asumir la responsabilidad de uno mismo y de sus acciones en términos de su avance profesional, su proyecto y su trabajo diario. Un programador pragmático se hace cargo de su propia carrera y no teme admitir la ignorancia o el error. No es el aspecto más agradable de la programación, sin duda, pero sucederá, incluso en los mejores proyectos. A pesar de las pruebas exhaustivas, la buena documentación y la sólida automatización, las cosas salen mal. Las entregas se retrasan. Surgen problemas técnicos imprevistos.

Estas cosas suceden y tratamos de lidiar con ellas de la manera más profesional posible. Esto significa ser honesto y directo. Podemos estar orgullosos de nuestras capacidades, pero debemos ser honestos acerca de nuestros defectos, de nuestra ignorancia así como de nuestros errores.

Asumir la responsabilidad

La responsabilidad es algo con lo que estás de acuerdo activamente. Usted se compromete a asegurarse de que algo se haga bien, pero no necesariamente tiene control directo sobre todos los aspectos del mismo. Además de hacer lo mejor que puedas, debes analizar la situación en busca de riesgos que estén fuera de tu control. Tienes derecho a asumir la responsabilidad de una situación imposible, o en la que los riesgos son demasiado grandes. Tendrás que tomar la decisión basándote en tu propia ética y juicio.

Cuando se acepta la responsabilidad de un resultado, se debe esperar que se le haga responsable de ello. Cuando cometes un error (como todos lo hacemos) o un error de juicio, admítelo honestamente y trata

de ofrecer opciones.

No culpes a alguien o a algo más, ni inventes una excusa. No culpe de todos los problemas a un proveedor, a un lenguaje de programación, a la administración o a sus compañeros de trabajo. Todos y cada uno de estos pueden desempeñar un papel, pero depende de *usted* proporcionar soluciones, no excusas.

Si existía el riesgo de que el proveedor no lo hiciera, entonces debería haber tenido un plan de contingencia. Si el disco se bloquea, llevándose consigo todo el código fuente, y no tienes una copia de seguridad, es tu culpa. Decirle a tu jefe "el gato se comió mi código fuente" simplemente no será suficiente.

CONSEJO

Ofrezca opciones, no ponga excusas tontas

Antes de acercarte a alguien para decirle por qué algo no se puede hacer, por qué es tarde o por qué está roto, detente y escúchate a ti mismo. Habla con el pato de goma en tu monitor o con el gato. ¿Tu excusa suena razonable o estúpida? ¿Cómo le va a sonar a tu jefe?

Repasa la conversación en tu mente. ¿Qué es probable que diga la otra persona? ¿Te preguntarán: «¿Has probado esto...?» o «¿No te ha parecido aquello?» ¿Cómo responderás? Antes de ir a contarles las malas noticias, ¿hay algo más que puedas probar? A veces, simplemente *sabes* lo que van a decir, así que ahórrales la molestia.

En lugar de excusas, ofrezca opciones. No digas que no se puede hacer; Explique qué se puede hacer para salvar la situación. ¿Hay que desechar el código? Edúquelos sobre el valor de la refactorización (consulte *Refactoriwg*, página 184). ¿Necesita dedicar tiempo a la creación de prototipos para determinar la mejor manera de proceder (consulte *Prototipos awd Post-it Notes*, página 53)? ¿Es necesario introducir mejores pruebas (véase *Easy to Test, de Code That*, página 189, y *Ruthless Testiwig*, página 237) o automatización (véase *Ubiquitous Automatiow*, página 230) para evitar que vuelva a ocurrir? Tal vez necesite recursos adicionales. No tengas miedo de preguntar o de admitir que necesitas ayuda.

Trata de eliminar las excusas tontas antes de expresarlas en voz alta. Si es necesario, díselo primero a tu gato. Después de todo, si el pequeño Tiddles va a asumir la culpa...

Las secciones relacionadas incluyen:

- *Prototipos de notas Post-it*, página 53
- *Refactoriwg*, página 184
- *Codificar el Easy de That para probar*, página 189
- *Automático ubicuo*, página 230
- *El despiadado Testiwg*, página 237

Desafíos

- ¿Cómo reaccionas cuando alguien, como un cajero de banco, un mecánico de automóviles o un empleado, se acerca a ti con una excusa poco convincente? ¿Qué piensas de ellos y de su empresa como resultado?

► Software Entropía

Si bien el desarrollo de software es inmune a casi todas las leyes físicas, la *ewtropy* nos golpea duramente. *Ewtropy* es un término de la física que se refiere a la cantidad de "desorden" en un sistema. Desafortunadamente, las leyes de la termodinámica garantizan que la entropía en el universo tiende hacia un máximo. Cuando el desorden aumenta en el software, los programadores lo llaman "putrefacción del software".

Hay muchos factores que pueden contribuir a la putrefacción del software. El más importante parece ser la psicología, o la cultura, que trabaja en un proyecto. Incluso si eres un equipo de una sola persona, la psicología de tu proyecto puede ser algo muy delicado. A pesar de los planes mejor trazados y de las mejores personas, un proyecto puede experimentar la ruina y la decadencia durante su vida. Sin embargo, hay otros proyectos que, a pesar de las enormes dificultades y los contratiempos constantes, luchan con éxito contra la tendencia de la naturaleza al desorden y logran salir bastante bien.

¿Qué es lo que marca la diferencia?

En el centro de las ciudades, algunos edificios son hermosos y limpios, mientras que otros son cascos podridos. ¿Por qué? Los investigadores en el campo del crimen y la decadencia urbana descubrieron un fascinante mecanismo de activación, uno que convierte muy rápidamente un edificio limpio, intacto y habitado en un abandono destrozado y abandonado [WK82].

Una ventana rota.

Una ventana rota, que no se ha reparado durante mucho tiempo, infunde en los habitantes del edificio una sensación de abandono, una sensación de que a los poderes fácticos no les importa el edificio. Así que se rompe otra ventana. La gente empieza a tirar basura. Aparecen grafitis. Comienzan los daños estructurales graves. En un espacio de tiempo relativamente corto, el edificio se daña más allá del deseo del propietario de arreglarlo, y la sensación de abandono se convierte en realidad.

La "teoría de la ventana rota" ha inspirado a los departamentos de policía de Nueva York y otras ciudades importantes a tomar medidas enérgicas contra las cosas pequeñas para mantener alejadas las cosas grandes. Funciona: mantenerse al tanto de las ventanas rotas, los grafitis y otras pequeñas infracciones ha reducido el nivel de delitos graves.

CONSEJO

No vivas con ventanas rotas

No dejes "ventanas rotas" (malos diseños, decisiones equivocadas o código deficiente) sin reparar. Arregla cada uno tan pronto como se descubra. Si no hay tiempo suficiente para arreglarlo correctamente, entonces *reviselo*. Tal vez pueda comentar el código ofensivo, o mostrar un mensaje de "No implementado", o sustituir los datos ficticios en su lugar. Toma *algunas* medidas para evitar daños mayores y para demostrar que estás al tanto de la situación.

Hemos visto que los sistemas limpios y funcionales se deterioran con bastante rapidez una vez que las ventanas comienzan a romperse. Hay otros factores que pueden contribuir a la putrefacción del software, y tocaremos algunos de ellos en otro lugar, pero la negligencia *acelera* la putrefacción más rápido que cualquier otro factor.

Puede que estés pensando que nadie tiene tiempo para ir por ahí limpiando todos los cristales rotos de un proyecto. Si sigues pensando así, entonces será mejor que planees conseguir un contenedor de basura o mudarte a otro vecindario. No dejes que la entropía gane.

Apagar incendios

Por el contrario, está la historia de un conocido de Andy obscenamente rico. Su casa era inmaculada, hermosa, cargada de antigüedades de

valor incalculable, *objetos de autor*, etc. Un día, un tapiz que colgaba demasiado cerca de la chimenea de su sala de estar se incendió. El fuego

El departamento se apresuró a salvar el día y su casa. Pero antes de arrastrar sus mangueras grandes y sucias a la casa, se detuvieron, con el fuego encendido, para extender una alfombra entre la puerta principal y la fuente del fuego.

No querían estropear la alfombra.

Un caso bastante extremo, sin duda, pero así debe ser con el software. Una ventana rota, un fragmento de código mal diseñado, una mala decisión de gestión con la que el equipo debe vivir durante la duración del proyecto, es todo lo que se necesita para iniciar el declive. Si te encuentras trabajando en un proyecto con bastantes ventanas rotas, es muy fácil caer en la mentalidad de "Todo el resto de este código es una mierda, simplemente seguiré su ejemplo". No importa si el proyecto ha estado bien hasta este punto. En el experimento original que condujo a la "Teoría de la ventana rota", un automóvil abandonado permaneció intacto durante una semana. Pero una vez que se rompió una sola ventana, el automóvil fue desmontado y volcado en cuestión de *horas*.

De la misma manera, si te encuentras en un equipo y un proyecto donde el código es prístinamente hermoso, escrito limpiamente, bien diseñado y elegante, es probable que tengas especial cuidado de no estropearlo, al igual que los bomberos. Incluso si hay un incendio (fecha límite, fecha de lanzamiento, demostración de la feria comercial, etc.), *no querrás* ser el primero en hacer un lío.

Las secciones relacionadas incluyen:

- *Stowe Soup awd Ranas hervidas*, página 7
- *Refactoriwg*, página 184
- *Temas pragmáticos*, página 224

Desafíos

- Ayude a fortalecer su equipo inspeccionando su "vecindario" informático. Elija dos o tres "ventanas rotas" y discuta con sus colegas cuáles son los problemas y qué se podría hacer para solucionarlos.
- ¿Puedes saber cuándo se rompe una ventana por primera vez? ¿Cuál es tu reacción? Si fue el resultado de la decisión de otra persona, o de un edicto de la gerencia, ¿qué puedes hacer al respecto?

3

Sopa de piedra y ranas hervidas

Los tres soldados regresaron a casa después de una huwgrý. Vaya, si los villanos se levantaban de pie, estaban seguros de que los villanos les darian a me. Pero cuando llegaron, cerraron las puertas con llave y las puertas se cerraron. Al cabo de unos años de vida, los habitantes se quedaron sin comida, y se quedaron sin nada.

Desanimados, los soldados hirvieron una olla de agua y la depositaron tres estibas en ella. The amāzed villāgers cāme out to uātch.

"Esto es sopa stowe", explicaron los soldados. —¿Es eso todo lo que has puesto? —preguntaron los aldeanos. "Absolutamente, aunque algunos de ustedes saben que es mejor tener un fuego".... A villāger raw off, returwiwg iw wo time uith a bqsket of carrots from his hoārd.

Un par de miwutes later, los aldeanos agaiw asksk: "¿Es eso?"

"Bueno", dijeron los soldados, "a par de dedos de pota le dan cuerpo". Off raw awother villager.

A lo largo de la hora, los soldados enumeraron más productos que la sopa: carne de res, puerros, hierbas. Es hora de a diferir la vida de los pueblos para que se desplacen a sus reservas habituales.

Más tarde habían producido a olla de sopa de steamiwg. Los soldados retiraron las estibas y se dirigieron a la aldea para deleitar a los primeros hombres que los habían reunido.

Hay un par de moralejas en la historia de la sopa de piedra. Los aldeanos son engañados por los soldados, que utilizan la curiosidad de los aldeanos para obtener comida de ellos. Pero lo más importante es que los soldados actúan como catalizadores, uniendo a la aldea para que puedan producir juntos algo que no podrían haber hecho por sí mismos: un resultado sinérgico. Al final, todos ganan.

De vez en cuando, es posible que deseas emular a los soldados.

Es posible que te encuentres en una situación en la que sepas exactamente lo que hay que hacer y cómo hacerlo. Todo el sistema aparece ante tus ojos, sabes que es correcto. Pero pida permiso para abordar todo el asunto y se encontrará con retrasos y miradas en blanco. La gente formará comités, los presupuestos necesitarán aprobación y las cosas se complicarán. Cada uno guardará sus propios recursos. A veces esto se llama "fatiga de arranque".

Es hora de sacar las piedras. Calcula lo que *puedes* pedir razonablemente. Desarróllalo bien. Una vez que lo tengas, muéstralos a la gente y deja que se maravillen. Luego diga "por supuesto, *sería* mejor si añadiéramos...". Finge que no es importante. Siéntese y espere a que comiencen a pedirle que agregue la funcionalidad que quería originalmente. A las personas les resulta más fácil unirse a un éxito continuo. Muéstrales un vistazo del futuro y conseguirás que se reúnan.¹

CONSE

Conviértete en un catalizador del cambio

El lado de los aldeanos

Por otro lado, la historia de la sopa de piedra también trata sobre el engaño suave y gradual. Se trata de concentrarse demasiado. Los aldeanos piensan en las piedras y se olvidan del resto del mundo. Todos caemos en la trampa, todos los días. Las cosas simplemente se nos acercan.

Todos hemos visto los síntomas. Los proyectos se van de las manos lenta e inexorablemente. La mayoría de los desastres de software comienzan siendo demasiado pequeños para darse cuenta, y la mayoría de los excesos de proyectos ocurren un día a la vez. Los sistemas se desvían de sus especificaciones característica por característica, mientras que parche tras parche se agrega a un fragmento de código hasta que no queda nada del original. A menudo es la acumulación de pequeñas cosas lo que rompe la moral y los equipos.

CONSE

Recuerde el panorama general

Nunca lo hemos intentado, la verdad. Pero dicen que si coges una rana y la dejas caer en agua hirviendo, volverá a saltar de nuevo. Sin embargo, si colocas la rana en una olla con agua fría y luego la calientas gradualmente, la rana no notará el lento aumento de temperatura y permanecerá quieta hasta que esté cocida.

1. Al hacer esto, es posible que se sienta reconfortado por la frase atribuida a la contralmirante Dra. Grace Hopper: "Es más fácil pedir perdón que obtener permiso".

Tenga en cuenta que el problema de la rana es diferente del problema de las ventanas rotas que se discute en la Sección 2. En la Teoría de la Ventana Rota, las personas pierden la voluntad de luchar contra la entropía porque perciben que a nadie más le importa. La rana simplemente no nota el cambio.

No seas como la rana. No pierdas de vista el panorama general. Revisa constantemente lo que sucede a tu alrededor, no solo lo que estás haciendo personalmente.

Las secciones relacionadas incluyen:

- *Software Entropy*, página 4
- *Programming por Coicidewece*, página 172
- *Refactoriing*, página 184
- *El Pozo de Requirements*, página 202
- *Temas pragmáticos*, página 224

Desafíos

- Al revisar un borrador de este libro, John Lakos planteó la siguiente cuestión: Los soldados engañan progresivamente a los aldeanos, pero el cambio que catalizan les hace bien a todos. Sin embargo, al engañar progresivamente a la rana, le estás haciendo daño. ¿Puedes determinar si estás haciendo sopa de piedra o sopa de rana cuando intentas catalizar el cambio? ¿La decisión es subjetiva u objetiva?

► Software lo suficientemente bueno

Striving para mejorar, a menudo mar what's well.

► **El Rey Lear 1.4**

Hay un viejo chiste sobre una empresa estadounidense que hace un pedido de 100.000 circuitos integrados a un fabricante japonés. Parte de la especificación era la tasa de defectos: un chip de cada 10.000. Unas semanas más tarde llegó el pedido: una caja grande que contenía miles de circuitos integrados y una pequeña que contenía solo diez. Pegada a la pequeña caja había una etiqueta que decía: "Estos son los

defectuosos".

Ojalá realmente tuviéramos este tipo de control sobre la calidad. Pero el mundo real simplemente no nos permitirá producir mucho que sea verdaderamente perfecto, particularmente no software libre de errores. El tiempo, la tecnología y el temperamento conspiran contra nosotros.

Sin embargo, esto no tiene por qué ser frustrante. Como describió Ed Yourdon en un artículo en *IEEE Software* [You95], puedes disciplinarte a ti mismo para escribir software que sea lo suficientemente bueno, lo suficientemente bueno para tus usuarios, para los mantenedores del futuro, para tu propia tranquilidad. Descubrirás que eres más productivo y que tus usuarios están más contentos. Y es muy posible que descubra que sus programas son en realidad mejores por su incubación más corta.

Antes de seguir adelante, tenemos que matizar lo que vamos a decir. La frase "lo suficientemente bueno" no implica un código descuidado o mal producido. Todos los sistemas deben cumplir con los requisitos de sus usuarios para tener éxito. Simplemente estamos abogando por que los usuarios tengan la oportunidad de participar en el proceso de decidir cuándo lo que has producido es lo suficientemente bueno.

Involucre a sus usuarios en la compensación

Normalmente, estás escribiendo software para otras personas. A menudo se acordará de obtener requisitos de ellos.² Pero, ¿con qué frecuencia les preguntas *qué tan bueno* quieren que sea su software? A veces no habrá otra opción. Si está trabajando en marcapasos, el transbordador espacial o una biblioteca de bajo nivel que se difundirá ampliamente, los requisitos serán más estrictos y sus opciones más limitadas. Sin embargo, si estás trabajando en un producto completamente nuevo, tendrás diferentes limitaciones. El personal de marketing tendrá promesas que cumplir, los usuarios finales eventuales pueden haber hecho planes basados en un cronograma de entrega, y su empresa ciertamente tendrá limitaciones de flujo de efectivo. Sería poco profesional ignorar las necesidades de estos usuarios simplemente para añadir nuevas características al programa, o para pulir el código una vez más. No estamos abogando por el pánico: es igualmente poco profesional prometer escalas de tiempo imposibles y tomar atajos de ingeniería básica para cumplir con un plazo.

2. ¡Se suponía que era una broma!

El alcance y la calidad del sistema que produce deben especificarse como parte de los requisitos de ese sistema.

CONSEJO

Hacer de la calidad una cuestión de requisitos

A menudo te encontrarás en situaciones en las que hay que hacer concesiones. Sorprendentemente, muchos usuarios prefieren usar software con algunas asperezas *que* esperar un año para la versión multimedia. Muchos departamentos de TI con presupuestos ajustados estarían de acuerdo. Un buen software hoy es a menudo preferible a un software perfecto mañana. Si les das a tus usuarios algo con lo que jugar desde el principio, sus comentarios a menudo te llevarán a una mejor solución final (ver *Tracer Bullets*, página 48).

Sepa cuándo detenerse

En cierto modo, la programación es como pintar. Comienzas con un lienzo en blanco y ciertas materias primas básicas. Utilizas una combinación de ciencia, arte y artesanía para determinar qué hacer con ellos. Esboza una forma general, pinta el entorno subyacente y, a continuación, rellena los detalles. Constantemente das un paso atrás con un ojo crítico para ver lo que has hecho. De vez en cuando tiras un lienzo y vuelves a empezar.

Pero los artistas te dirán que todo el trabajo duro se arruina si no sabes cuándo parar. Si añades capa sobre capa, detalle sobre detalle, *el paitiwig se pierde en el paitiwt*.

No estropees un programa perfectamente bueno con un exceso de embellecimiento y refinamiento. Continúa y deja que tu código se mantenga por sí mismo durante un tiempo. Puede que no sea perfecto. No te preocupes: nunca podría ser perfecto. (En el capítulo 6, página 171, discutiremos las filosofías para desarrollar código en un mundo imperfecto).

Las secciones relacionadas incluyen:

- *Tracer Bullets*, página 48
- *El Pozo de Requirements*, página 202
- *Temas pragmáticos*, página 224
- *Great Expectations*, página 255

Desafíos

- Fíjate en los fabricantes de las herramientas de software y los sistemas operativos que utilizas. ¿Puedes encontrar alguna evidencia de que estas empresas se sienten cómodas enviando software que saben que no es perfecto? Como usuario, ¿preferiría (1) esperar a que eliminan todos los errores, (2) tener un software complejo y aceptar algunos errores, o (3) optar por un software más simple con menos defectos?
- Considera el efecto de la modularización en la entrega de software. ¿Llevará más o menos tiempo conseguir que un bloque monolítico de software tenga la calidad requerida en comparación con un sistema diseñado en módulos? ¿Puedes encontrar ejemplos comerciales?

5

Tu Portafolio de Conocimientos

Aw iwestmewt iw kwouledge aluays pays the best iwterest.

► **Benjamín Franklin**

Ah, el bueno de Ben Franklin, nunca se queda sin una homilía concisa. ¿Por qué, si pudiéramos acostarnos temprano y levantarnos temprano, seríamos grandes programadores, verdad? El pájaro madrugador puede contraer el gusano, pero ¿qué pasa con el gusano temprano?

En este caso, sin embargo, Ben realmente dio en el clavo. Tus conocimientos y experiencia son tus activos profesionales más importantes.

Desafortunadamente, son *expiring assets*.³ Tu conocimiento se vuelve obsoleto a medida que se desarrollan nuevas técnicas, idiomas y entornos. Las fuerzas cambiantes del mercado pueden hacer que su experiencia sea obsoleta o irrelevante. Dada la velocidad a la que pasan los años de la Web, esto puede suceder bastante rápido.

A medida que el valor de su conocimiento disminuye, también lo hace su valor para su empresa o cliente. Queremos evitar que esto suceda.

3. Un *expiring asset* es algo cuyo valor disminuye con el tiempo. Los ejemplos incluyen un almacén lleno de plátanos y un boleto para un juego de pelota.

Tu Portafolio de Conocimientos

Nos gusta pensar en todos los datos que los programadores saben sobre informática, los dominios de aplicación en los que trabajan y toda su experiencia como en sus *carteras de Knowledge*. La gestión de un portafolio de conocimiento es muy similar a la gestión de un portafolio financiero:

1. Los inversores serios invierten regularmente, como un hábito.
2. La diversificación es la clave del éxito a largo plazo.
3. Los inversores inteligentes equilibran sus carteras entre inversiones conservadoras y de alto riesgo y alta recompensa.
4. Los inversores intentan comprar barato y vender caro para obtener el máximo rendimiento.
5. Las carteras deben revisarse y reequilibrarse periódicamente.

Para tener éxito en su carrera, debe administrar su cartera de conocimientos utilizando estas mismas pautas.

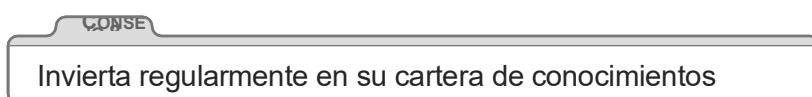
Construyendo tu Portafolio

- Invierte regularmente. Al igual que en la inversión financiera, debes invertir en tu cartera de conocimientos de *forma regulada*. Incluso si es solo una pequeña cantidad, el hábito en sí es tan importante como las sumas. En la siguiente sección se enumeran algunos ejemplos de objetivos.
- Diversificar. Cuantas más *cosas diferentes* sepas, más valioso eres. Como punto de referencia, es necesario conocer los entresijos de la tecnología concreta con la que se está trabajando actualmente. Pero no te detengas ahí. El rostro de la informática cambia rápidamente: la tecnología de moda hoy en día puede ser casi inútil (o al menos no tener demanda) mañana. Cuantas más tecnologías te sientas cómodo, mejor podrás adaptarte al cambio.
- Gestione el riesgo. La tecnología existe a lo largo de un espectro que va desde los estándares riesgosos y potencialmente de alta recompensa hasta los de bajo riesgo y baja recompensa. No es una buena idea invertir todo su dinero en acciones de alto riesgo que podrían colapsar repentinamente, ni debe invertir todo de manera conservadora y perder posibles oportunidades. No pongas todos

los huevos técnicos en la misma cesta.

- Compre barato, venda caro. Aprender una tecnología emergente antes de que se vuelva popular puede ser tan difícil como encontrar una acción infravalorada, pero la recompensa puede ser igual de gratificante. Aprender Java cuando salió por primera vez puede haber sido arriesgado, pero valió la pena para los primeros usuarios que ahora están en la cima de ese campo.
- Revisión y reequilibrio. Esta es una industria muy dinámica. Esa tecnología caliente que comenzaste a investigar el mes pasado podría estar fría como una piedra a estas alturas. Tal vez necesites repasar esa tecnología de base de datos que no has usado en un tiempo. O tal vez podrías estar mejor posicionado para esa nueva oferta de trabajo si probaras ese otro idioma. . . .

De todas estas pautas, la más importante es la más sencilla de hacer:



Metas

Ahora que tiene algunas pautas sobre qué y cuándo agregar a su cartera de conocimientos, ¿cuál es la mejor manera de adquirir capital intelectual con el que financiar su cartera? He aquí algunas sugerencias.

- Aprende al menos un idioma nuevo cada año. Diferentes idiomas resuelven los mismos problemas de diferentes maneras. Al aprender varios enfoques diferentes, puede ayudar a ampliar su pensamiento y evitar quedarse atrapado en la rutina. Además, aprender muchos idiomas es mucho más fácil ahora, gracias a la gran cantidad de software disponible gratuitamente en Internet (consulte la página 267).
- Lea un libro técnico cada trimestre. Las librerías están llenas de libros técnicos sobre temas interesantes relacionados con su proyecto actual. Una vez que adquieras el hábito, lee un libro al mes. Una vez que hayas dominado las tecnologías que estás utilizando actualmente, diversifica y estudia algunas que *no se* relacionen con tu proyecto.
- Lee también libros no técnicos. Es importante recordar que las computadoras son utilizadas por *personas*, personas cuyas necesidades se están tratando de satisfacer. No olvides el lado

humano de la ecuación.

- Toma clases. Busque cursos interesantes en su colegio o universidad local, o tal vez en la próxima feria comercial que llegue a la ciudad.
- Participar en grupos de usuarios locales. No te limites a escuchar, sino a participar activamente. El aislamiento puede ser mortal para tu carrera; Averigüe en qué están trabajando las personas fuera de su empresa.
- Experimenta con diferentes entornos. Si sólo has trabajado en Windows, juega con Unix en casa (el Linux disponible gratuitamente es perfecto para esto). Si solo has usado makefiles y un editor, prueba con un IDE y viceversa.
- Mantente al día. Suscríbete a revistas especializadas y otras revistas (consulte la página 262 para obtener recomendaciones). Elija algunos que abarquen una tecnología diferente a la de su proyecto actual.
- Conéctate. ¿Quieres conocer los entresijos de un nuevo idioma u otra tecnología? Los grupos de noticias son una excelente manera de averiguar qué experiencias están teniendo otras personas con él, la jerga particular que usan, etc. Navegue por la Web en busca de documentos, sitios comerciales y cualquier otra fuente de información que pueda encontrar.

Es importante seguir invirtiendo. Una vez que te sientas cómodo con algún nuevo idioma o tecnología, sigue adelante. Aprende otra.

No importa si alguna vez utilizas alguna de estas tecnologías en un proyecto, o incluso si las pones en tu currículum. El proceso de aprendizaje ampliará tu pensamiento, abrándote a nuevas posibilidades y nuevas formas de hacer las cosas. La polinización cruzada de ideas es importante; Trata de aplicar las lecciones que has aprendido a tu proyecto actual. Incluso si tu proyecto no utiliza esa tecnología, tal vez puedas tomar prestadas algunas ideas. Familiarízate con la orientación a objetos, por ejemplo, y escribirás programas C simples de manera diferente.

Oportunidades de aprendizaje

Así que estás leyendo vorazmente, estás al tanto de todos los últimos desarrollos de última hora en tu campo (no es algo fácil de hacer), y alguien te hace una pregunta. No tienes la menor idea de cuál es la

respuesta, y lo admites libremente.

Dow no deja que se detenga ahí. Tómalo como un reto personal encontrar la respuesta. Pregúntale a un gurú. (Si no tienes un gurú en tu oficina, deberías poder encontrar uno en Internet: mira el recuadro en la página opuesta). Busca en la Web. Ve a la biblioteca.⁴

Si no puede encontrar la respuesta usted mismo, averigüe quién *caw*. No lo dejes descansar. Hablar con otras personas te ayudará a construir tu red personal, y es posible que te sorprendas a ti mismo encontrando soluciones a otros problemas no relacionados en el camino. Y esa vieja cartera sigue creciendo. . . .

Toda esta lectura e investigación lleva tiempo, y el tiempo ya es escaso. Por lo tanto, debe planificar con anticipación. Siempre ten algo que leer en un momento que de otro modo estaría muerto. El tiempo que pasa esperando a los médicos y dentistas puede ser una gran oportunidad para ponerse al día con su lectura, pero asegúrese de llevar su propia revista con usted, o es posible que se encuentre hojeando un artículo de 1973 sobre Papúa Nueva Guinea.

Pensamiento crítico

El último punto importante es pensar *críticamente* sobre lo que lees y escuchas. Debe asegurarse de que el conocimiento de su cartera sea preciso y no se vea influenciado por la exageración de los proveedores o los medios de comunicación. Ten cuidado con los fanáticos que insisten en que su dogma proporciona la *respuesta incorrecta*: puede o no ser aplicable a ti y a tu proyecto.

Nunca subestimes el poder del comercialismo. El hecho de que un motor de búsqueda web enumere un resultado primero no significa que sea la mejor coincidencia; El proveedor de contenido puede pagar para obtener la máxima facturación. El hecho de que una librería tenga un libro en un lugar destacado no significa que sea un buen libro, ni siquiera popular; Es posible que les hayan pagado para colocarlo allí.

CONSEJO

Analice críticamente lo que lee y escucha

Desafortunadamente, ya hay muy pocas respuestas simples. Pero con su extenso portafolio, y mediante la aplicación de un análisis crítico a la

4. En esta era de la Web, muchas personas parecen haberse olvidado de las bibliotecas reales en vivo llenas de material de investigación y personal.

Cuidado y Cultivo de los Gurús

Con la adopción global de Internet, los gurús de repente están tan cerca como su tecla Enter . Entonces, ¿cómo encuentras uno y cómo consigues que hable contigo?

Encontramos que hay algunos trucos simples.

- Sepa exactamente lo que quiere preguntar y sea lo más específico posible.
- Formula tu pregunta de manera cuidadosa y cortés. Recuerda que estás pidiendo un favor; No parece estar exigiendo una respuesta.
- Una vez que hayas formulado tu pregunta, detente y busca de nuevo la respuesta. Elige algunas palabras clave y busca en la Web. Busque las FAQ apropiadas (listas de preguntas frecuentes con respuestas).
- Decide si quieres preguntar en público o en privado. Los grupos de noticias de Usenet son maravillosos lugares de encuentro para expertos en casi cualquier tema, pero algunas personas desconfían de la naturaleza pública de estos grupos. Alternativamente, siempre puedes enviar un correo electrónico a tu gurú directamente. De cualquier manera, usa una línea de asunto significativa. ("¿Necesito ayuda!!" no es suficiente).

Siéntate y sé paciente. La gente está ocupada y puede llevar días obtener una respuesta específica.

Finalmente, asegúrese de agradecer a cualquiera que le responda. Y si ves que la gente hace preguntas que puedes responder, haz tu parte y participa.

torrente de publicaciones técnicas que leerás, puedes entender el respuestas complejas.

Desafíos

- Empieza a aprender un nuevo idioma esta semana. ¿Siempre programado en C++? Pruebe Smalltalk [URL 13] o Squeak [URL 14]. ¿Haces Java? Pruebe con Eiffel [URL 10] o TOM [URL 15]. Consulte la página 267 para ver las fuentes de otros compiladores y entornos libres.
- Empieza a leer un libro nuevo (ipero termina este primero!). Si está realizando una implementación y codificación muy detalladas, lea un libro sobre diseño y arquitectura. Si te dedicas al diseño de alto nivel, lee un libro sobre técnicas de codificación.

- Sal y habla de tecnología con personas que no estén involucradas en tu proyecto actual o que no trabajen para la misma empresa. Conéctese en la cafetería de su empresa, o tal vez busque a otros entusiastas en una reunión de un grupo de usuarios local.



¡Comunicar!

Creo que es mejor ser mirado que ser pasado por alto.

► **Mae West, Bella de los noventa, 1934**

Tal vez podamos aprender una lección de la Sra. West. No se trata solo de lo que tienes, sino también de cómo lo empaquetas. Tener las mejores ideas, el código más fino o el pensamiento más pragmático es, en última instancia, estéril a menos que puedas comunicarte con otras personas. Una buena idea es huérfana sin una comunicación efectiva.

Como desarrolladores, tenemos que comunicarnos a muchos niveles. Pasamos horas en reuniones, escuchando y hablando. Trabajamos con los usuarios finales, tratando de entender sus necesidades. Escribimos código, que comunica nuestras intenciones a una máquina y documenta nuestro pensamiento para las futuras generaciones de desarrolladores. Escribimos propuestas y memorandos solicitando y justificando recursos, informando sobre nuestro estado y sugiriendo nuevos enfoques. Y trabajamos diariamente dentro de nuestros equipos para defender nuestras ideas, modificar las prácticas existentes y sugerir otras nuevas. Una gran parte de nuestro día lo pasamos comunicándonos, por lo que debemos hacerlo bien.

Hemos elaborado una lista de ideas que nos parecen útiles.

Sepa lo que quiere decir

Probablemente la parte más difícil de los estilos más formales de comunicación utilizados en los negocios es determinar exactamente qué es lo que se quiere decir. Los escritores de ficción trazan sus libros en detalle antes de comenzar, pero las personas que escriben documentos técnicos a menudo están felices de sentarse frente a un teclado, ingresar "1. Introducción", y comienzan a escribir lo que se les ocurra a continuación.

Planifica lo que quieras decir. Escribe un esquema. Luego pregúntate: "¿Esto transmite lo que sea que estoy tratando de decir?" Refírnalo

hasta que lo haga.

Este enfoque no solo es aplicable a la redacción de documentos. Cuando te enfrentes a una reunión importante o una llamada telefónica con un cliente importante, anota las ideas que quieras comunicar y planifica un par de estrategias para transmitirlas.

Conoce a tu audiencia

Solo te estás comunicando si estás transmitiendo información. Para ello, debes comprender las necesidades, los intereses y las capacidades de tu audiencia. Todos nos hemos sentado en reuniones en las que un friki del desarrollo se pone vidrioso ante los ojos del vicepresidente de marketing con un largo monólogo sobre los méritos de alguna tecnología arcana. Esto no es comunicarse: es solo hablar, y es molesto.⁵

Forma una imagen mental sólida de tu audiencia. El acróstico WISDOM, que se muestra en la Figura 1.1 en la página siguiente, puede ayudar.

Supongamos que desea sugerir un sistema basado en la Web para permitir que sus usuarios finales envíen informes de errores. Puede presentar este sistema de muchas maneras diferentes, dependiendo de su audiencia. Los usuarios finales apreciarán que pueden enviar informes de errores las 24 horas del día sin tener que esperar en el teléfono. Tu departamento de marketing podrá utilizar este hecho para impulsar las ventas. Los gerentes del departamento de soporte tendrán dos razones para estar contentos: se necesitará menos personal y se automatizarán los informes de problemas. Por último, los desarrolladores pueden disfrutar de la experiencia con las tecnologías cliente-servidor basadas en la Web y un nuevo motor de base de datos. Al hacer la presentación adecuada a cada grupo, conseguirás que todos se entusiasmen con tu proyecto.

Elige tu momento

Son las seis de la tarde del viernes, después de una semana en la que los auditores han estado presentes. El hijo menor de tu jefe está en el hospital, afuera llueve a cántaros y el viaje a casa está garantizado que será una pesadilla. Probablemente este no sea un buen momento para pedirle una actualización de memoria para su PC.

Como parte de la comprensión de lo que tu audiencia necesita escuchar, debes determinar cuáles son sus prioridades. Atrapa a un gerente que acaba de pasar un mal rato porque se perdió parte del código fuente, y

-
5. La palabra *awwoy* proviene del francés antiguo *ewui*, que también significa "aburrir".

Figura 1.1. El acróstico WISDOM: entender a una audiencia

¿Qué quieres que aprendan?
¿Cuál es su interés en lo que tienes que decir?
¿Qué tan sofisticados son?
¿Cuántos detalles quieren? ¿A quién
quieres que le pertenezca la información?
¿Cómo puedes motivarlos para que te

Tendrá un oyente más receptivo a sus ideas sobre las representaciones del código fuente. Haz que lo que dices sea relevante en el tiempo, así como en el contenido. A veces, todo lo que se necesita es la simple pregunta: "¿Es este un buen momento para hablar de...?"

Elige un estilo

Ajusta el estilo de tu presentación para que se adapte a tu público. Algunas personas quieren una sesión informativa formal de "solo los hechos". A otros les gusta una charla larga y amplia antes de ponerse manos a la obra. Cuando se trata de documentos escritos, a algunos les gusta recibir grandes informes encuadrados, mientras que otros esperan un simple memorándum o correo electrónico. En caso de duda, pregunte.

Recuerde, sin embargo, que usted es la mitad de la transmisión de comunicación. Si alguien dice que necesita un párrafo que describa algo y no ves la manera de hacerlo en menos de varias páginas, díselo. Recuerda, ese tipo de retroalimentación también es una forma de comunicación.

Haz que se vea bien

Tus ideas son importantes. Merecen un vehículo atractivo para transmitirlos a su audiencia.

Demasiados desarrolladores (y sus directivos) se concentran únicamente en el contenido a la hora de producir documentos escritos. Creemos que esto es un error. Cualquier chef te dirá que puedes trabajar como esclavo en la cocina durante horas solo para arruinar tus esfuerzos con una mala presentación.

Hoy en día no hay excusa para producir documentos impresos de mal aspecto. Los procesadores de texto modernos (junto con sistemas de

diseño como LATEX y troff) pueden producir resultados impresionantes. Necesitas aprender solo un poco
Pocos comandos básicos. Si su procesador de textos admite hojas de estilo, use

ellos. (Es posible que su empresa ya tenga hojas de estilo definidas que puede utilizar). Obtén información sobre cómo establecer encabezados y pies de página. Mire los documentos completos incluidos con su paquete para obtener ideas sobre el estilo y el diseño. *Revisa la ortografía*, primero automáticamente y luego a mano. Después del punzón, hay filetes de ortografía que el damero puede anudar ketch.

Involucra a tu audiencia

A menudo nos encontramos con que los documentos que producimos terminan siendo menos importantes que el proceso por el que pasamos para producirlos. Si es posible, involucre a sus lectores con los primeros borradores de su documento. Obtén su retroalimentación y escoge sus cerebros. Construirás una buena relación de trabajo y probablemente producirás un mejor documento en el proceso.

Sé un oyente

Hay una técnica que debes usar si quieras que la gente te escuche: *listen to them*. Incluso si se trata de una situación en la que tienes toda la información, incluso si se trata de una reunión formal en la que estás parado frente a 20 trajes, si no los escuchas, ellos no te escucharán.

Anime a las personas a hablar haciendo preguntas o pídale que resuman lo que les dice. Convierta la reunión en un diálogo y expondrá su punto de vista de manera más efectiva. Quién sabe, puede que incluso aprendas algo.

Volver a la gente

Si le haces una pregunta a alguien, sientes que es descortés si no responde. Pero, ¿con qué frecuencia no te pones en contacto con las personas cuando te envían un correo electrónico o un memorándum pidiendo información o solicitando alguna acción? En el ajetreo de la vida cotidiana, es fácil olvidarlo. Siempre responda a los correos electrónicos y mensajes de voz, incluso si la respuesta es simplemente "Me pondré en contacto con usted más tarde". Mantener a las personas informadas hace que sean mucho más indulgentes con los deslices ocasionales y les hace sentir que no los has olvidado.

CONSE

Es tanto lo que dices como la forma en que lo dices

A menos que trabajes en el vacío, necesitas ser capaz de comunicarte.

Cuanto más efectiva sea esa comunicación, más influyente te volverás.

Comunicación por correo electrónico

Todo lo que hemos dicho sobre la comunicación por escrito se aplica igualmente al correo electrónico. El correo electrónico ha evolucionado hasta el punto de convertirse en un pilar de las comunicaciones intra e intercorporativas. El correo electrónico se utiliza para discutir contratos, para resolver disputas y como prueba en los tribunales. Pero por alguna razón, las personas que nunca enviarían un documento en papel en mal estado están felices de lanzar correos electrónicos de aspecto desagradable por todo el mundo.

Nuestros consejos para enviar

- correos electrónicos son simples: Revisa antes de presionar ENVIAR . Revisa la ortografía.
- Mantén el formato simple. Algunas personas leen el correo electrónico usando fuentes proporcionales, por lo que las imágenes artísticas ASCII que laboriosamente creó les parecerán araños de gallina.
- Utilice el correo en formato HTML o de texto enriquecido solo si sabe que todos sus destinatarios pueden leerlo. El texto sin formato es universal.
- Trata de mantener las citas al mínimo. A nadie le gusta recibir su propio correo electrónico de 100 líneas con "Acepto" añadido.
- Si está citando el correo electrónico de otras personas, asegúrese de atribuirlo y citarlo en línea (en lugar de como un archivo adjunto).

No enciendas fuego a menos que quieras que regrese y te persiga más tarde.

Revisa tu lista de destinatarios antes de enviar. Un artículo reciente del *Wall Street Journal* describía a un empleado que se dedicó a distribuir críticas a su jefe por correo electrónico departamental, sin darse cuenta de que su jefe estaba incluido en la lista de distribución.

Archiva y organiza tu correo electrónico, tanto el material importante que recibes como el correo que envías.

Como varios empleados de Microsoft y Netscape descubrieron durante la investigación del Departamento de Justicia de 1999, el

correo electrónico es para siempre. Trate de prestar la misma atención y cuidado al correo electrónico que a cualquier memorándum o informe escrito.

Resumen

- Sepa lo que quiere decir. •
- Conoce a tu audiencia.
- Elige tu momento. •
- Elige un estilo.
- Haz que se vea bien.
- Involucra a tu audiencia. • Sé un oyente.
- Vuelve a la gente.

Las secciones relacionadas incluyen:

- *Prototipos de notas Post-it*, página 53
- *Temas pragmáticos*, página 224

Desafíos

- Hay varios buenos libros que contienen secciones sobre las comunicaciones dentro de los equipos de desarrollo [Bro95, McC95, DL99]. Procura leer los tres durante los próximos 18 meses. Además, el libro *Diwosaur Braiws* [Ber96] habla de la carga emocional que todos llevamos al entorno laboral.
- La próxima vez que tengas que hacer una presentación, o escribir un memorándum defendiendo alguna posición, trata de leer el acróstico de WISDOM en la página 20 antes de empezar. Fíjate si te ayuda a entender cómo posicionar lo que dices. Si es apropiado, hable con su audiencia después y vea qué tan precisa fue su evaluación de sus necesidades.

Esta página se ha dejado en blanco intencionadamente

Capítulo 2

Un enfoque pragmático

Hay ciertos consejos y trucos que se aplican en todos los niveles del desarrollo de software, ideas que son casi axiomáticas y procesos que son prácticamente universales. Sin embargo, estos enfoques rara vez se documentan como tales; La mayoría de las veces las encontrarás escritas como frases extrañas en discusiones de diseño, gestión de proyectos o codificación.

En este capítulo reuniremos estas ideas y procesos. Las dos primeras secciones, *Los males de la duplicación y la ortogomía*, están estrechamente relacionadas. El primero le advierte que no duplique el conocimiento en todos sus sistemas, el segundo que no divida ninguna pieza de conocimiento en varios componentes del sistema.

A medida que aumenta el ritmo del cambio, se hace cada vez más difícil mantener la relevancia de nuestras aplicaciones. En *Reversibilidad*, veremos algunas técnicas que ayudan a aislar tus proyectos de su entorno cambiante.

Las siguientes dos secciones también están relacionadas. En *Tracer Bullets*, hablamos de un estilo de desarrollo que le permite recopilar requisitos, probar diseños e implementar código al mismo tiempo. Si esto suena demasiado bueno para ser verdad, lo es: los desarrollos de balas trazadoras no siempre son aplicables. Cuando no lo están, *Prototipos awd Post-it Notes* muestra cómo usar la creación de prototipos para probar arquitecturas, algoritmos, interfaces e ideas.

A medida que la informática madura lentamente, los diseñadores producen lenguajes de nivel cada vez más alto. Si bien aún no se ha inventado el compilador que acepta "make it so", en *Domaiw Lawguages* presentamos algunas sugerencias más modestas que puedes implementar por ti mismo.

Por último, todos trabajamos en un mundo de tiempo y recursos limitados. Puedes sobrevivir mejor a estas dos carencias (y mantener a tus jefes más felices) si te vuelves bueno para calcular cuánto tiempo tomarán las cosas, lo cual cubrimos en *Estimatiwg*.

Si tienes en cuenta estos principios fundamentales durante el desarrollo, puedes escribir código que sea mejor, más rápido y más fuerte. Incluso puedes hacer que parezca fácil.

7

Los males de la duplicación

Dotar a un ordenador de dos conocimientos contradictorios era la forma preferida del capitán James T. Kirk para desactivar una inteligencia artificial merodeadora. Desafortunadamente, el mismo principio puede ser efectivo para derribar *su* código.

Como programadores, recopilamos, organizamos, mantenemos y aprovechamos el conocimiento. Documentamos el conocimiento en especificaciones, lo hacemos cobrar vida en el código en ejecución y lo usamos para proporcionar las comprobaciones necesarias durante las pruebas.

Desafortunadamente, el conocimiento no es estable. Cambia, a menudo rápidamente. Su comprensión de un requisito puede cambiar después de una reunión con el cliente. El gobierno cambia una regulación y parte de la lógica empresarial queda obsoleta. Las pruebas pueden mostrar que el algoritmo elegido no funcionará. Toda esta inestabilidad hace que pasemos gran parte de nuestro tiempo en modo de mantenimiento, reorganizando y reexpresando el conocimiento en nuestros sistemas.

La mayoría de las personas asumen que el mantenimiento comienza cuando se lanza una aplicación, que el mantenimiento significa corregir errores y mejorar las funciones. Creemos que estas personas están equivocadas. Los programadores están constantemente en modo de mantenimiento. Nuestra comprensión cambia día a día. Los nuevos requisitos llegan a medida que diseñamos o codificamos. Tal vez el entorno cambie. Cualquiera que sea la razón, el mantenimiento no es una actividad discreta, sino una parte rutinaria de todo el proceso de

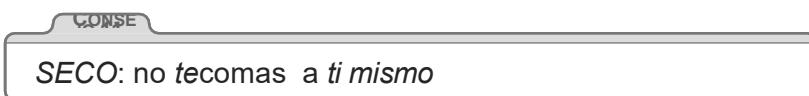
desarrollo.

Cuando realizamos el mantenimiento, tenemos que encontrar y cambiar las representaciones de las cosas, esas cápsulas de conocimiento incrustadas en la aplicación. El problema es que es fácil duplicar el conocimiento en las especificaciones, procesos y programas que desarrollamos, y cuando lo hacemos, invitamos a una pesadilla de mantenimiento, una que comienza mucho antes de que se envíe la aplicación.

Creemos que la única manera de desarrollar software de forma fiable, y de hacer que nuestros desarrollos sean más fáciles de entender y mantener, es seguir lo que llamamos el *principio DRY*:

CADA PIEZA DE CONOCIMIENTO DEBE TENER UNA REPRESENTACIÓN
ÚNICA, INEQUÍVOCAS Y AUTORIZADA DENTRO DE UN SISTEMA.

¿Por qué lo llamamos *DRY*?



La alternativa es tener la misma cosa expresada en dos o más lugares. Si cambias uno, tienes que acordarte de cambiar los demás, o, como los ordenadores alienígenas, tu programa se pondrá de rodillas por una contradicción. No es una cuestión de si lo recordarás: es una cuestión de cuándo lo olvidarás.

Encontrarás el principio *DRY* apareciendo una y otra vez a lo largo de este libro, a menudo en contextos que no tienen nada que ver con la codificación. Creemos que es una de las herramientas más importantes en la caja de herramientas del Programa Pragmático.

En esta sección, describiremos los problemas de duplicación y sugeriremos estrategias generales para lidiar con ella.

¿Cómo surge la duplicación?

La mayor parte de la duplicación que vemos se clasifica en una de las siguientes categorías:

- Duplicación impuesta. Los desarrolladores sienten que no tienen otra opción: el entorno parece requerir duplicación.
- Duplicación inadvertida. Los desarrolladores no se dan cuenta de que están duplicando información.

- Duplicación impaciente. Los desarrolladores se vuelven perezosos y duplican porque parece más fácil.
- Duplicación entre desarrolladores. Varias personas en un equipo (o en diferentes equipos) duplican una información.

Veamos estas cuatro fórmulas de duplicación con más detalle.

Duplicación impuesta

A veces, la duplicación parece ser forzada sobre nosotros. Las normas de los proyectos pueden exigir documentos que contengan información duplicada, o documentos que dupliquen la información del código. Cada una de las plataformas objetivo requiere sus propios lenguajes de programación, bibliotecas y entornos de desarrollo, lo que nos hace duplicar definiciones y procedimientos compartidos. Los propios lenguajes de programación requieren ciertas estructuras que duplican la información. Todos hemos trabajado en situaciones en las que nos sentíamos impotentes para evitar la duplicación. Y, sin embargo, a menudo hay formas de mantener cada pieza de conocimiento en un solo lugar, honrando el *principio DRY* y haciendo nuestras vidas más fáciles al mismo tiempo. Estas son algunas técnicas:

Múltiples representaciones de la información. A nivel de codificación, a menudo necesitamos tener la misma información representada de diferentes formas. Tal vez estemos escribiendo una aplicación cliente-servidor, utilizando diferentes idiomas en el cliente y el servidor, y necesitemos representar alguna estructura compartida en ambos. Tal vez necesitemos una clase cuyos atributos reflejen el esquema de una tabla de base de datos. Tal vez estés escribiendo un libro y quieras incluir extractos de programas que también compilarás y probarás.

Con un poco de ingenio, normalmente se puede eliminar la necesidad de duplicación. A menudo, la respuesta es escribir un filtro simple o un generador de código. Las estructuras en varios idiomas se pueden construir a partir de una representación de metadatos común utilizando un generador de código simple cada vez que se construye el software (un ejemplo de esto se muestra en la Figura 3.4, página 106). Las definiciones de clase se pueden generar automáticamente a partir del esquema de la base de datos en línea o de los metadatos utilizados para construir el esquema en primer lugar. Los extractos de código de este libro son insertados por un preprocesador cada vez que formateamos el texto. El truco está en hacer que el proceso esté activo:

no puede ser una conversión única, o volvemos a estar en una posición de duplicación de datos.

Documentación en código. A los programadores se les enseña a comentar su código: un buen código tiene muchos comentarios. Desafortunadamente, nunca se les enseña *que* el código necesita comentarios: el código malo *requiere* muchos comentarios.

El principio *DRY* nos dice que debemos mantener el conocimiento de bajo nivel en el código, donde pertenece, y reservar los comentarios para otras explicaciones de alto nivel. De lo contrario, estamos duplicando el conocimiento, y cada cambio significa cambiar tanto el código como los comentarios. Los comentarios inevitablemente se volverán obsoletos, y los comentarios poco confiables son peores que no tener ningún comentario. (Véase *Está todo escrito en la página 248*, para más información sobre los comentarios.)

Documentación y código. Escribe documentación, luego escribes código. Algo cambia, y se modifica la documentación y se actualiza el código. Tanto la documentación como el código contienen representaciones del mismo conocimiento. Y todos sabemos que en el calor del momento, con plazos inminentes y clientes importantes clamando, tendemos a diferir la actualización de la documentación.

Dave trabajó una vez en un interruptor de télex internacional. Es comprensible que el cliente exigiera una especificación de prueba exhaustiva y exigiera que el software pasara todas las pruebas en cada entrega. Para asegurarse de que las pruebas reflejaran con precisión la especificación, el equipo las generó mediante programación a partir del propio documento. Cuando el cliente modificaba su especificación, el conjunto de pruebas cambiaba automáticamente. Una vez que el equipo convenció al cliente de que el procedimiento era sólido, la generación de pruebas de precisión solía llevar sólo unos segundos.

Problemas lingüísticos. Muchos idiomas imponen una duplicación considerable en la fuente. A menudo, esto ocurre cuando el lenguaje separa la interfaz de un módulo de su implementación. C y C++ tienen archivos de encabezado que duplican los nombres y la información de tipo de las variables, funciones y clases (para C++) exportadas. Object Pascal incluso duplica esta información en el mismo archivo. Si utiliza llamadas a procedimientos remotos o CORBA [URL 29], duplicará la información de la interfaz entre la especificación de la interfaz y el código que la implementa.

No existe una técnica fácil para superar los requisitos de un idioma. Si bien algunos entornos de desarrollo ocultan la necesidad de archivos de encabezado al generarlos automáticamente, y Object Pascal le

permite abreviar declaraciones de funciones repetidas, generalmente
está atrapado con

lo que te dan. Al menos con la mayoría de los problemas basados en el lenguaje, un archivo de encabezado que no esté de acuerdo con la implementación generará algún tipo de error de compilación o vinculación. Todavía puedes equivocarte, pero al menos te lo dirán bastante pronto.

Piense también en los comentarios en los archivos de encabezado e implementación. No tiene ningún sentido duplicar una función o un encabezado de clase entre los dos archivos. Use los archivos de encabezado para documentar los problemas de la interfaz y los archivos de implementación para documentar los detalles esenciales que los usuarios de su código no necesitan saber.

Duplicación inadvertida

A veces, la duplicación se produce como resultado de errores en el diseño.

Veamos un ejemplo de la industria de la distribución. Digamos que nuestro análisis revela que, entre otros atributos, un camión tiene un tipo, un número de licencia y un conductor. Del mismo modo, una ruta de entrega es una combinación de una ruta, un camión y un conductor. Codificamos algunas clases en función de esta comprensión.

Pero, ¿qué sucede cuando Sally llama para decir que está enferma y tenemos que cambiar de conductor? Tanto `Truck` como `DeliveryRoute` contienen un conductor. ¿Cuál cambiamos? Está claro que esta duplicación es mala. Normalícelo de acuerdo con el modelo de negocio subyacente: ¿un camión realmente tiene un conductor como parte de su conjunto de atributos subyacentes? ¿Una ruta? O tal vez necesite haber un tercer objeto que una un conductor, un camión y una ruta. Cualquiera que sea la solución final, evite este tipo de datos no normalizados.

Hay un tipo un poco menos obvio de datos no normalizados que se produce cuando tenemos varios elementos de datos que son mutuamente dependientes. Veamos una clase que representa una línea:

```
class Línea {
     Público:
        Punto de
        inicio;
        Extremo del
        punto; doble
        longitud;
};
```

A primera vista, esta clase puede parecer razonable. Una línea claramente tiene un principio y un final, y siempre tendrá una longitud (incluso si es cero). Pero nosotros

tienen duplicación. La longitud está definida por los puntos inicial y final: cambie uno de los puntos y la longitud cambia. Es mejor hacer que la longitud sea un campo calculado:

```
class Línea {
    Público:
        Punto de
        inicio;
        Extremo del
        punto;
    double length() { return start.distanceTo(end); }
};
```

Más adelante en el proceso de desarrollo, puede optar por violar el *principio DRY* por razones de rendimiento. Con frecuencia, esto ocurre cuando necesita almacenar datos en caché para evitar repetir operaciones costosas. El truco está en localizar el impacto. La violación no se expone al mundo exterior: solo los métodos dentro de la clase tienen que preocuparse por mantener las cosas en orden.

```
class Línea {
    Privado:
        Bool
            cambiad
        o; doble
        largura; Punto
        empezar;
        Extremo del
        punto;
    Público:
        void setStart(Punto p) { inicio = p; cambiado = verdadero; }
        vacío setEnd(Punto p) { fin = p; cambiado = verdadero; }
        Punto getStart(vacío) { devolución
            empezar; } Punto getEnd(vacío) { devolución
            fin; }
        double getLength() {
            if (cambiado) {
                longitud = start.distanceTo(fin);
                cambiado = falso;
            }
            longitud de retorno;
        }
};
```

Este ejemplo también ilustra un problema importante para los lenguajes orientados a objetos, como Java y C++. Siempre que sea posible, utilice siempre las funciones de descriptor de acceso para leer y escribir los atributos de los objetos.¹ Facilitará la adición de funcionalidades, como el almacenamiento en caché, en el futuro.

1. El uso de funciones de descriptor de acceso se relaciona con el *principio de acceso*

Uwiform de Meyer [Mey97b], que establece que "Todos los servicios ofrecidos por un módulo deben estar disponibles a través de una notación uniforme, lo que no traiciona si se implementan a través del almacenamiento o mediante el cálculo".

Duplicación impaciente

Cada proyecto tiene presiones de tiempo, fuerzas que pueden impulsar a los mejores de nosotros a tomar atajos. ¿Necesitas una rutina similar a la que has escrito? Tendrás la tentación de copiar el original y hacer algunos cambios. ¿Necesita un valor que represente el número máximo de puntos? Si cambio el archivo de encabezado, todo el proyecto se reconstruirá. Tal vez debería usar un número literal aquí; Y aquí; Y aquí. ¿Necesita una clase como una en el tiempo de ejecución de Java? El código fuente está disponible, así que ¿por qué no copiarlo y hacer los cambios que necesite (a pesar de las disposiciones de la licencia)?

Si sientes esta tentación, recuerda el manido aforismo "los atajos hacen que los retrasos sean largos". Es posible que ahorre algunos segundos ahora, pero con la posible pérdida de horas más tarde. Piense en los problemas que rodean el fiasco del Y2K. Muchos fueron causados por la pereza de los desarrolladores al no parametrizar el tamaño de los campos de fecha o implementar bibliotecas centralizadas de servicios de fecha.

La duplicación impaciente es una forma fácil de detectar y manejar, pero requiere disciplina y la voluntad de dedicar tiempo por adelantado para ahorrar dolor más adelante.

Duplicación entre desarrolladores

Por otro lado, quizás el tipo de duplicación más difícil de detectar y manejar ocurre entre diferentes desarrolladores en un proyecto. Conjuntos enteros de funciones pueden duplicarse inadvertidamente, y esa duplicación podría pasar desapercibida durante años, lo que provocaría problemas de mantenimiento. Escuchamos de primera mano de un estado de los EE. UU. cuyos sistemas informáticos gubernamentales fueron inspeccionados para verificar el cumplimiento del Y2K. La auditoría reveló más de 10.000 programas, cada uno de los cuales contenía su propia versión de la validación del número de seguridad social.

A un alto nivel, enfréntate al problema con un diseño claro, un líder técnico fuerte en el proyecto (ver página 228 en *Técnicas Pragmáticas*) y una división de responsabilidades bien entendida dentro del diseño. Sin embargo, a nivel de módulo, el problema es más insidioso. Las funcionalidades o datos comúnmente necesarios que no entran en un área obvia de responsabilidad pueden implementarse muchas veces.

Creemos que la mejor manera de lidiar con esto es fomentar la

comunicación activa y frecuente entre los desarrolladores. Establezca foros para discutir problemas comunes. (En proyectos anteriores, hemos configurado Usenet privado

Grupos de noticias para permitir que los desarrolladores intercambien ideas y hagan preguntas. Esto proporciona una forma no intrusiva de comunicarse, incluso a través de múltiples sitios, al tiempo que conserva un historial permanente de todo lo dicho). Designar a un miembro del equipo como bibliotecario del proyecto, cuya tarea consiste en facilitar el intercambio de conocimientos. Tener un lugar central en el árbol de fuentes donde se puedan depositar las rutinas de utilidad y los scripts. Y asegúrese de leer el código fuente y la documentación de otras personas, ya sea de manera informal o durante las revisiones de código. No estás husmeando, estás aprendiendo de ellos. Y recuerda, el acceso es recíproco, no te desconozcas por el hecho de que otras personas estudien minuciosamente tu código.

CONSE

Facilite su reutilización

Lo que estás tratando de hacer es fomentar un entorno en el que sea más fácil encontrar y reutilizar cosas existentes que escribirlas tú mismo. *Si no es así, la gente no lo hará.* Y si no se reutiliza, se corre el riesgo de duplicar el conocimiento.

Las secciones relacionadas incluyen:

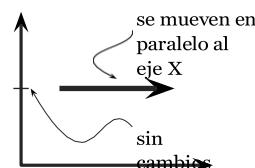
- *Ortodomía*, página 34
- *Texto Mawipulationw*, página 99
- *Code Gewerators*, página 102
- *Refactoriwig*, página 184
- *Temas pragmáticos*, página 224
- *Automático ubicuo*, página 230
- *Todo es Writiwig*, página 248

Ortogonalidad

La ortogonalidad es un concepto crítico si desea producir sistemas que sean fáciles de diseñar, construir, probar y ampliar. Sin embargo, el concepto de ortogonalidad rara vez se enseña directamente. A menudo es una característica implícita de otros métodos y técnicas que aprendes. Esto es un error. Una vez que aprenda a aplicar el principio de ortogonalidad directamente, notará una mejora inmediata en la calidad de los sistemas que produce.

¿Qué es la ortogonalidad?

"Ortogonalidad" es un término tomado de la geom- etry. Dos rectas son ortogonales si se encuentran en ángulos rectos, como los ejes de un gráfico. En términos vectoriales, las dos líneas son *independientes*. Muévete a lo largo de una de las líneas y tu posición proyectada en la otra no cambiará.



En informática, el término ha llegado a significar una especie de independencia o desacoplamiento. Dos o más cosas son ortogonales si los cambios en una no afectan a ninguna de las otras. En un sistema bien diseñado, el código de la base de datos será ortogonal a la interfaz de usuario: puede cambiar la interfaz sin afectar a la base de datos e intercambiar bases de datos sin cambiar la interfaz.

Antes de ver los beneficios de los sistemas ortogonales, primero veamos un sistema que no es ortogonal.

Un sistema no ortogonal

Estás en un tour en helicóptero por el Gran Cañón cuando el piloto, que cometió el error obvio de comer pescado para el almuerzo, de repente gime y se desmaya. Afortunadamente, te dejó flotando a 100 pies sobre el suelo. Usted racionaliza que la palanca de paso colectivo 2 controla la elevación general, por lo que

2. Los helicópteros tienen cuatro controles básicos. El *cílico* es el palo que se sostiene en la mano derecha. Muévelo y el helicóptero se moverá en la dirección correspondiente. Su mano izquierda sostiene la *palanca de tono colectivo*. Tira hacia arriba sobre esto y aumentas el paso en todas las cuchillas, generando elevación. Al final de la palanca de paso se encuentra el *acelerador*. Por último, tiene dos pedales de pie, que varían la

cantidad de empuje del rotor de cola y así ayudan a girar el helicóptero.

Al hacerlo ligeramente, se iniciará un suave descenso hasta el suelo. Sin embargo, cuando lo pruebas, descubres que la vida no es tan simple. El morro del helicóptero cae y comienzas a descender en espiral hacia la izquierda. De repente descubres que estás volando en un sistema en el que cada entrada de control tiene efectos secundarios. Baje la palanca de la mano izquierda y deberá agregar un movimiento hacia atrás compensatorio al joystick de la mano derecha y presionar el pedal derecho. Pero luego cada uno de estos cambios vuelve a afectar a todos los demás controles. De repente, estás haciendo malabarismos con un sistema increíblemente complejo, donde cada cambio afecta a todas las demás entradas. Su carga de trabajo es fenomenal: sus manos y pies están en constante movimiento, tratando de equilibrar todas las fuerzas que interactúan.

Decididamente, los controles de los helicópteros no son ortogonales.

Beneficios de la ortogonalidad

Como ilustra el ejemplo del helicóptero, los sistemas no ortogonales son inherentemente más complejos de cambiar y controlar. Cuando los componentes de cualquier sistema son altamente interdependientes, no existe una solución local.

CONSE

Eliminar efectos entre cosas no relacionadas

Queremos diseñar componentes que sean autónomos: independientes y con un propósito único y bien definido (lo que Yourdon y Constanthine llaman *cohesion* [YC86]). Cuando los componentes están aislados unos de otros, sabes que puedes cambiar uno sin tener que preocuparte por el resto. Siempre y cuando no cambie las interfaces externas de ese componente, puede estar seguro de que no causará problemas que se propaguen por todo el sistema.

Si se escriben sistemas ortogonales, se obtienen dos ventajas principales: una mayor productividad y una reducción del riesgo.

Aumente la productividad

- Los cambios son localizados, por lo que el tiempo de desarrollo y el tiempo de prueba se reducen. Es más fácil escribir componentes relativamente pequeños y autónomos que un solo bloque grande

de código. Los componentes simples pueden ser

Diseñado, codificado, probado unitariamente y luego olvidado: no es necesario seguir cambiando el código existente a medida que agrega código nuevo.

- Un enfoque ortogonal también promueve la reutilización. Si los componentes tienen responsabilidades específicas y bien definidas, se pueden combinar con nuevos componentes de formas que no fueron previstas por sus implementadores originales. Cuanto más débilmente acoplados estén sus sistemas, más fácil será reconfigurarlos y rediseñarlos.
- Hay una ganancia bastante sutil en la productividad cuando se combinan componentes ortogonales. Supongamos que un componente hace M cosas distintas y otro hace N cosas. Si son ortogonales y los combinás, el resultado hace $M \times N$ cosas. Sin embargo, si los dos componentes no son ortogonales, habrá superposición y el resultado será menor. Se obtiene más funcionalidad por unidad de esfuerzo mediante la combinación de componentes ortogonales.

Reducir el riesgo

Un enfoque ortogonal reduce los riesgos inherentes a cualquier desarrollo.

- Las secciones de código dañadas se aislan. Si un módulo está enfermo, es menos probable que propague los síntomas por el resto del sistema. También es más fácil cortarlo y trasplantarlo en algo nuevo y saludable.
- El sistema resultante es menos frágil. Realice pequeños cambios y correcciones en un área en particular, y cualquier problema que genere se restringirá a esa área.
- Un sistema ortogonal probablemente se probará mejor, porque será más fácil diseñar y ejecutar pruebas en sus componentes.
- No estará tan estrechamente vinculado a un proveedor, producto o plataforma en particular, ya que las interfaces de estos componentes de terceros estarán aisladas en partes más pequeñas del desarrollo general.

Veamos algunas de las formas en que puedes aplicar el principio de ortogonalidad a tu trabajo.

Equipos de proyecto

¿Te has dado cuenta de cómo algunos equipos de proyecto son eficientes, ya que todos saben lo que tienen que hacer y contribuyen plenamente, mientras que los miembros de otros equipos de proyectos son eficientes?

¿Los equipos están constantemente discutiendo y no parecen capaces de apartarse del camino de los demás?

A menudo se trata de un problema de ortogonalidad. Cuando los equipos están organizados con mucha superposición, los miembros están confundidos acerca de las responsabilidades. Cada cambio necesita una reunión de todo el equipo, porque cualquiera de ellos *podría* verse afectado.

¿Cómo se organizan los equipos en grupos con responsabilidades bien definidas y un solapamiento mínimo? No hay una respuesta sencilla. Depende en parte del proyecto y de su análisis de las áreas de cambio potencial. También depende de las personas que tengas disponibles. Nuestra preferencia es empezar por separar la infraestructura de la aplicación. Cada componente principal de la infraestructura (base de datos, interfaz de comunicaciones, capa de middleware, etc.) tiene su propio subequipo. Cada división obvia de la funcionalidad de la aplicación se divide de manera similar. Luego miramos a las personas que tenemos (o planeamos tener) y ajustamos las agrupaciones en consecuencia.

Puede obtener una medida informal de la ortogonalidad de la estructura de un equipo de proyecto. Basta con ver cuánta gente *está dispuesta* a participar en la discusión de cada cambio que se solicita. Cuanto mayor sea el número, menos ortogonal será el grupo. Claramente, un equipo ortogonal es más eficiente. (Dicho esto, también alentamos a los subequipos a que se comuniquen constantemente entre sí).

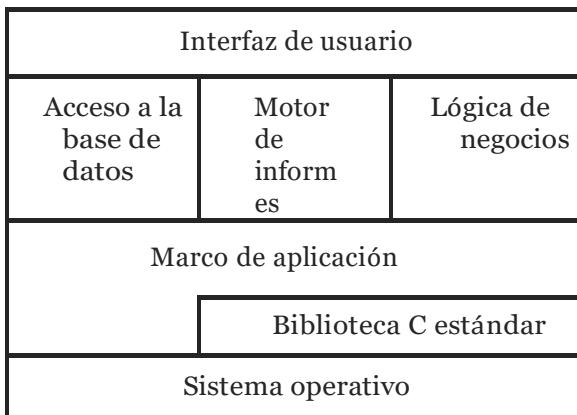
Diseño

La mayoría de los desarrolladores están familiarizados con la necesidad de diseñar sistemas ortogonales, aunque pueden usar palabras como *modular*, *composewt-based*, y *layered* para describir el proceso. Los sistemas deben estar compuestos por un conjunto de módulos que cooperen, cada uno de los cuales implementa una funcionalidad independiente de los demás. A veces, estos componentes se organizan en capas, cada una de las cuales proporciona un nivel de abstracción. Este enfoque por capas es una forma eficaz de diseñar sistemas ortogonales. Debido a que cada capa utiliza solo las abstracciones proporcionadas por las capas inferiores, tiene una gran flexibilidad para cambiar las implementaciones subyacentes sin afectar el código. La estratificación también reduce el riesgo de dependencias

descontroladas entre los módulos. A menudo verá capas expresadas en diagramas como la Figura 2.1 en la página siguiente.

Hay una prueba fácil para el diseño ortogonal. Una vez que haya trazado sus componentes, pregúntese: *Si dramaticallally logre lo requerido.*

Figura 2.1. Diagrama de capas



mewts behiwd a particular fuwctiow, hou mawy modules are affected? En un sistema ortogonal, la respuesta debe ser "uno".³ Mover un botón en un panel de GUI no debería requerir un cambio en el esquema de la base de datos. La adición de ayuda contextual no debería cambiar el subsistema de facturación.

Consideremos un sistema complejo para monitorear y controlar una planta de calefacción. El requisito original requería una interfaz gráfica de usuario, pero los requisitos se cambiaron para agregar un sistema de respuesta de voz con control telefónico por tonos de la planta. En un sistema diseñado ortogonalmente, solo necesitaría cambiar los módulos asociados con la interfaz de usuario para manejar esto: la lógica subyacente de control de la planta permanecería sin cambios. De hecho, si estructura su sistema con cuidado, debería poder admitir ambas interfaces con la misma base de código subyacente. *It's Just a View*, página 157, habla sobre la escritura de código desacoplado utilizando el paradigma Modelo-Vista-Controlador (MVC), que funciona bien en esta situación.

3. En realidad, esto es ingenuo. A menos que tenga mucha suerte, la mayoría de los cambios en los requisitos del mundo real afectarán a múltiples funciones del sistema. Sin embargo, si se analiza el cambio en términos de funciones, lo ideal es que cada cambio

funcional afecte a un solo módulo.

También pregúntate qué tan desacoplado está tu diseño de los cambios en el mundo real. ¿Está utilizando un número de teléfono como identificador de cliente? ¿Qué sucede cuando la compañía telefónica reasigna códigos de área? *No confíes en las propiedades de estos productos que no puedas dominar.*

Kits de herramientas y bibliotecas

Tenga cuidado de preservar la ortogonalidad de su sistema a medida que introduce kits de herramientas y bibliotecas de terceros. Elija sus tecnologías sabiamente.

Una vez trabajamos en un proyecto que requería que un determinado cuerpo de código Java se ejecutara tanto localmente en una máquina servidor como de forma remota en una máquina cliente. Las alternativas para distribuir las clases de esta manera fueron RMI y CORBA. Si una clase se hiciera accesible de forma remota usando RMI, cada llamada a un método remoto en esa clase podría potencialmente arrojar una excepción, lo que significa que una implementación ingenua requeriría que manejáramos la excepción cada vez que se usaran nuestras clases remotas. El uso de RMI aquí claramente no es ortogonal: el código que llama a nuestras clases remotas no debería tener que estar al tanto de sus ubicaciones. La alternativa, usar CORBA, no imponía esa restricción: podíamos escribir código que desconociera la ubicación de nuestras clases.

Cuando traigas un kit de herramientas (o incluso una biblioteca de otros miembros de tu equipo), pregúntate si impone cambios en tu código que no deberían estar allí. Si un esquema de persistencia de objetos es transparente, entonces es ortogonal. Si requiere que crees o accedas a objetos de una manera especial, entonces no lo es. Mantener estos detalles aislados de su código tiene el beneficio adicional de facilitar el cambio de proveedor en el futuro.

El sistema Enterprise Java Beans (EJB) es un ejemplo interesante de ortogonalidad. En la mayoría de los sistemas orientados a transacciones, el código de aplicación tiene que delinear el inicio y el final de cada transacción. Con EJB, esta información se expresa de forma declarativa como metadatos, fuera de cualquier código. El mismo código de aplicación puede ejecutarse en diferentes entornos de transacciones EJB sin ningún cambio. Es probable que esto sea un modelo para muchos entornos futuros.

Otra vuelta de tuerca interesante a la ortogonalidad es la

Programación Orientada a Aspectos (AOP), un proyecto de investigación de Xerox Parc ([KLM⁺⁹⁷] y [URL 49]). AOP le permite expresar en un solo lugar el comportamiento que, de otro modo, se distribuiría a través de su código fuente. Por ejemplo, mensajes de registro

normalmente se generan esparciendo llamadas explícitas a alguna función de registro en todo el origen. Con AOP, se implementa el registro ortogonalmente a las cosas que se registran. Usando la versión Java de AOP, puede escribir un mensaje de registro al ingresar cualquier método de la clase Fred codificando el *aspect*:

```
aspect Trace {
    consejo * Fred.*(..) {
        static antes de {
            Log.write("> ingresando " + thisJoinPoint.methodName);
        }
    }
}
```

Si incorpora este aspecto en el código, se generarán mensajes de seguimiento. Si no lo haces, no verás ningún mensaje. De cualquier manera, su fuente original no ha cambiado.

Codificación

Cada vez que escribe código, corre el riesgo de reducir la ortogonalidad de su aplicación. A menos que supervise constantemente no solo lo que está haciendo, sino también el contexto más amplio de la aplicación, es posible que duplique involuntariamente la funcionalidad en algún otro módulo o que exprese el conocimiento existente dos veces.

Hay varias técnicas que se pueden utilizar para mantener la ortogonalidad:

- Mantén tu código desacoplado. Escriba código tímido: módulos que no revelen nada innecesario a otros módulos y que no dependan de las implementaciones de otros módulos. Pruebe la Ley de Deméter [LH89], de la que hablamos en *Decoupling awd the Law of Demeter*, página 138. Si necesita cambiar el estado de un objeto, haga que el objeto lo haga por usted. De esta manera, su código permanece aislado de la implementación del otro código y aumenta las posibilidades de que permanezca ortogonal.
- Evite los datos globales. Cada vez que el código hace referencia a datos globales, se vincula con los demás componentes que comparten esos datos. Incluso los globales que solo tiene la intención de leer pueden generar problemas (por ejemplo, si de repente necesita cambiar su código para que sea multihilo). En general, el código es más fácil de entender y mantener si se pasa explícitamente el contexto necesario a los módulos. En las aplicaciones orientadas a objetos, el contexto a menudo se pasa como parámetros a

constructores de objetos. En otro código, puede crear estructuras que contengan el contexto y pasar referencias a ellas.

El patrón Singleton en *Design Patterns* [GHJV95] es una forma de garantizar que solo haya una instancia de un objeto de una clase particular. Mucha gente usa estos objetos singleton como un tipo de variable global (particularmente en lenguajes, como Java, que de otro modo no admiten el concepto de globales). Tenga cuidado con los singletons, ya que también pueden provocar enlaces innecesarios.

- Evite funciones similares. A menudo, se encontrará con un conjunto de funciones que parecen similares: tal vez comparten código común al principio y al final, pero cada una tiene un algoritmo central diferente. El código duplicado es un síntoma de problemas estructurales. Echa un vistazo al patrón de estrategia en *Design Patterns* para una mejor implementación.

Acostúmbrate a criticar constantemente tu código. Busque cualquier oportunidad para reorganizarlo para mejorar su estructura y ortogonalidad. Este proceso se llama *refactoriwig*, y es tan importante que le hemos dedicado una sección (véase *Refactoriwig*, página 184).

Ensayo

Un sistema diseñado e implementado ortogonalmente es más fácil de probar. Debido a que las interacciones entre los componentes del sistema están formalizadas y limitadas, se pueden realizar más pruebas del sistema a nivel de módulo individual. Esta es una buena noticia, porque las pruebas a nivel de módulo (o unitario) son considerablemente más fáciles de especificar y realizar que las pruebas de integración. De hecho, sugerimos que cada módulo tenga su propia prueba unitaria incorporada en su código, y que estas pruebas se realicen automáticamente como parte del proceso de compilación regular (consulte *Easy to Test de Code That*, página 189).

La construcción de pruebas unitarias es en sí misma una prueba interesante de ortogonalidad. ¿Qué se necesita para crear y vincular una prueba unitaria? ¿Tiene que arrastrar un gran porcentaje del resto del sistema solo para obtener una prueba para compilar o vincular? Si es así, ha encontrado un módulo que no está bien desacoplado del resto del sistema.

La corrección de errores también es un buen momento para evaluar la

ortogonalidad del sistema en su conjunto. Cuando se encuentre con un problema, evalúe qué tan localizado

La solución es. ¿Se cambia sólo un módulo, o los cambios se reparten por todo el sistema? Cuando haces un cambio, ¿lo arregla todo o surgen misteriosamente otros problemas? Esta es una buena oportunidad para poner en práctica la automatización. Si utiliza un sistema de control de código fuente (y lo hará después de leer *Source Code Control*, página 86), etiquete las correcciones de errores cuando vuelva a registrar el código después de la prueba. A continuación, puede ejecutar informes mensuales que analicen las tendencias en el número de archivos de origen afectados por cada corrección de errores.

Documentación

Quizás sorprendentemente, la ortogonalidad también se aplica a la documentación. Los ejes son el contenido y la presentación. Con una documentación verdaderamente ortogonal, debería poder cambiar la apariencia drásticamente sin cambiar el contenido. Los procesadores de texto modernos proporcionan hojas de estilo y macros que ayudan (véase *It's All Writing*, página 248).

Vivir con ortogonalidad

La ortogonalidad está estrechamente relacionada con el *principio DRY* introducido en la página

27. Con *DRY*, se busca minimizar la duplicación dentro de un sistema, mientras que con la ortogonalidad se reduce la interdependencia entre los componentes del sistema. Puede ser una palabra torpe, pero si se utiliza el principio de ortogonalidad, combinado estrechamente con el *principio DRY*, se encontrará con que los sistemas que se desarrollan son más flexibles, más comprensibles y más fáciles de depurar, probar y mantener.

Si te traen a un proyecto en el que la gente lucha desesperadamente por hacer cambios, y en el que cada cambio parece hacer que otras cuatro cosas salgan mal, recuerda la pesadilla con el helicóptero. Es probable que el proyecto no esté diseñado y codificado orthogonalmente. Es hora de refactorizar.

Y, si eres piloto de helicóptero, no te comas el pescado....

Las secciones relacionadas incluyen:

- *Los males de Duplicatiow*, página 26
- *Código fuente Cowtrol*, página 86

- *Diseñado por Cowtract*, página 109
- *Decoupling awd el Lau de Deméter*, página 138

- *Metaprogrammiwg*, página 144
- *Es solo o Vieu*, página 157
- *Refactoriwig*, página 184
- *Codificar el Easy de That para probar*, página 189
- *Wizards malvados*, página 198
- *Temas pragmáticos*, página 224
- *Todo es Writiwig*, página 248

Desafíos

- Considere la diferencia entre las grandes herramientas orientadas a la GUI que normalmente se encuentran en los sistemas Windows y las utilidades de línea de comandos pequeñas pero combinables que se utilizan en los indicadores de shell. ¿Qué conjunto es más ortogonal y por qué? ¿Cuál es más fácil de usar exactamente para el propósito para el que fue diseñado? ¿Qué conjunto es más fácil de combinar con otras herramientas para afrontar nuevos retos?
- C++ admite la herencia múltiple y Java permite que una clase implemente varias interfaces. ¿Qué impacto tiene el uso de estas instalaciones en la ortogonalidad? ¿Hay alguna diferencia en el impacto entre el uso de varias herencias y varias interfaces? ¿Hay alguna diferencia entre usar la delegación y usar la herencia?

Ejercicios

1. Está escribiendo una clase llamada `Partir`, que divide las líneas de entrada en campos.
¿Cuál de las siguientes dos firmas de clase Java es el diseño más ortogonal?
*Awsuer
ow p. 279*

```
class Split1 {
    public Split1(InputStreamReader rdr) { ... }
    public void readNextLine() lanza IOException { ... }
    public int numFields() { ... }
    public String getField(int fieldNo) { ... }
}
class Split2 {
    public Split2(String line) { ... }
    public int numFields() { ... }
    public String getField(int fieldNo) { ... }
}
```

2. Lo que dará lugar a un diseño más ortogonal: diálogo modal o no modal
¿buzones?
*Awsuer
ow p. 279*
3. ¿Qué hay de los lenguajes procedimentales frente a la tecnología de objetos? ¿Qué resultados
¿En un sistema más ortogonal?
*Awsuer
ow p. 280*

Reversibilidad

Nothing is more dangerous than the owl that you owe.

► Emil-Auguste Chartier, *Propos sur la religion*, 1938

Los ingenieros prefieren soluciones simples y únicas a los problemas. Exámenes de matemáticas que te permiten proclamar con gran confianza que $x = 2$ son mucho más cómodos que los ensayos borrosos y cálidos sobre las innumerables causas de la Revolución Francesa. La gerencia tiende a estar de acuerdo con los ingenieros: las respuestas simples y fáciles encajan bien en las hojas de cálculo y los planes del proyecto.

¡Ojalá el mundo real cooperara! Desafortunadamente, si bien x es 2 hoy, puede que deba ser 5 mañana y 3 la próxima semana. Nada es para siempre, y si confías mucho en algún hecho, casi puedes garantizar que cambiará.

Siempre hay más de una forma de implementar algo, y por lo general hay más de un proveedor disponible para proporcionar un producto de terceros. Si te embarcas en un proyecto obstaculizado por la noción miope de que solo hay *una* forma de hacerlo, es posible que te lleves una sorpresa desagradable. A muchos equipos de proyecto se les abren los ojos a la fuerza a medida que se desarrolla el futuro:

"Pero tú said usaría database XYZ! ¡Somos el 85% de los responsables del proyecto, pero no lo hacemos!", protestó el progresivo. "Lo siento, pero nuestra empresa decidió establecer un estándar de PDQ para todos los proyectos. Está fuera de mis haws. Tendremos que recodificar. Todos ustedes serán working weekeds until further notice".

Los cambios no tienen por qué ser tan draconianos, ni siquiera tan inmediatos. Pero a medida que pasa el tiempo y avanza su proyecto, es posible que se encuentre atrapado en una posición insostenible. Con cada decisión crítica, el equipo del proyecto se compromete con un objetivo más pequeño, una versión más estrecha de la realidad que tiene menos opciones.

En el momento en que se han tomado muchas decisiones críticas, el objetivo se vuelve tan pequeño que si se mueve, o el viento cambia de dirección, o una mariposa en Tokio bate sus alas, fallas.⁴ Y es posible que falles por una gran cantidad.

4. Tome un sistema no lineal o caótico y aplique un pequeño cambio a una de sus

entradas. Es posible que obtenga un resultado grande y, a menudo, impredecible. El cliché de la mariposa batiendo sus alas en Tokio podría ser el inicio de una cadena de eventos que termine generando un tornado en Texas. ¿Te suena a algún proyecto que conozcas?

El problema es que las decisiones críticas no son fácilmente reversibles.

Una vez que decide utilizar la base de datos de este proveedor, o ese patrón arquitectónico, o un determinado modelo de implementación (cliente-servidor frente a autónomo, por ejemplo), se compromete a un curso de acción que no se puede deshacer, excepto a un gran costo.

Reversibilidad

Muchos de los temas de este libro están orientados a la producción de software flexible y adaptable. Al atenernos a sus recomendaciones, especialmente el *principio DRY* (página 26), el desacoplamiento (página 138) y el uso de metadatos (página 144), no tenemos que tomar tantas decisiones críticas e irreversibles. Esto es algo bueno, porque no siempre tomamos las mejores decisiones la primera vez. Nos comprometemos con una determinada tecnología solo para descubrir que no podemos contratar a suficientes personas con las habilidades necesarias. Bloqueamos a un determinado proveedor externo justo antes de que sea comprado por su competidor. Los requisitos, los usuarios y el hardware cambian más rápido de lo que podemos desarrollar el software.

Supongamos que decide, al principio del proyecto, utilizar una base de datos relacional del proveedor A. Mucho más tarde, durante las pruebas de rendimiento, descubre que la base de datos es simplemente demasiado lenta, pero que la base de datos de objetos del proveedor B es más rápida. Con la mayoría de los proyectos convencionales, no tendrías suerte. La mayoría de las veces, las llamadas a productos de terceros están enredadas en todo el código. Pero si realmente se abstrae la idea de una base de datos, hasta el punto en que simplemente proporciona persistencia como un servicio, entonces tiene la flexibilidad de cambiar de caballo a mitad de camino.

Del mismo modo, supongamos que el proyecto comienza como un modelo cliente-servidor, pero luego, al final del juego, el departamento de marketing decide que los servidores son demasiado caros para algunos clientes y quieren una versión independiente. ¿Qué tan difícil sería eso para ti? Dado que es solo un problema de implementación, *no debería esperar más de a días*. Si llevaría más tiempo, entonces no has pensado en la reversibilidad. La otra dirección es aún más interesante. ¿Qué sucede si el producto independiente que está creando debe implementarse en un modo cliente-servidor o *de nivel w*? Tampoco

debería ser así.

El error está en asumir que cualquier decisión está grabada en piedra, y en no prepararse para las contingencias que puedan surgir. En lugar de tallar

Decisiones en piedra, piensa en ellas más como si estuvieran escritas en la arena de la playa. Una gran ola puede llegar y acabar con ellos en cualquier momento.

CONSEJO

No hay decisiones finales

Arquitectura flexible

Si bien muchas personas intentan mantener su *código* flexible, también debe pensar en mantener la flexibilidad en las áreas de arquitectura, implementación e integración de proveedores.

Tecnologías como CORBA pueden ayudar a aislar partes de un proyecto de los cambios en el lenguaje de desarrollo o la plataforma. ¿El rendimiento de Java en esa plataforma no está a la altura de las expectativas? Vuelva a codificar el cliente en C++ y no será necesario cambiar nada más. ¿El motor de reglas de C++ no es lo suficientemente flexible? Cambia a una versión de Smalltalk. Con una arquitectura CORBA, solo hay que recibir un golpe por el componente que se está reemplazando; los demás componentes no deberían verse afectados.

¿Estás desarrollando para Unix? ¿Cuál? ¿Se han abordado todos los problemas de portabilidad? ¿Está desarrollando para una versión concreta de Windows? ¿Cuál: 3.1, 95, 98, NT, CE o 2000? ¿Qué tan difícil será admitir otras versiones? Si mantienes las decisiones suaves y flexibles, no será nada difícil. Si tiene una encapsulación deficiente, un acoplamiento alto y una lógica o parámetros codificados de forma rígida en el código, podría ser imposible.

¿No está seguro de cómo el marketing quiere implementar el sistema? Piénselo desde el principio y puede admitir un modelo independiente, cliente-servidor o de nivel w con solo cambiar un archivo de configuración. Hemos escrito programas que hacen precisamente eso.

Normalmente, simplemente puede ocultar un producto de terceros detrás de una interfaz abstracta y bien definida. De hecho, siempre hemos sido capaces de hacerlo en cualquier proyecto en el que hemos trabajado. Pero supongamos que no pudieras aislarlo tan limpiamente. ¿Qué pasaría si tuvieras que esparrcir ciertas declaraciones generosamente a lo largo del código? Coloque ese requisito en los

metadatos y utilice algún mecanismo automático, como Aspects (consulte la página 39) o Perl, para insertar las declaraciones necesarias en el propio código. Cualquiera que sea el mecanismo que

uso, *make que sea reversible*. Si algo se agrega automáticamente, también se puede eliminar automáticamente.

Nadie sabe lo que nos deparará el futuro, ¡especialmente nosotros! Así que habilita tu código para el rock-n-roll: para "rockear" cuando pueda, para rodar con los golpes cuando deba hacerlo.

Las secciones relacionadas incluyen:

- *Decoupling awd el Lau de Deméter*, página 138
- *Metaprogramming*, página 144
- *Es solo o Viejo*, página 157

Desafíos

- Tiempo para un poco de mecánica cuántica con el gato de Schrödinger.
Supongamos que tienes un gato en una caja cerrada, junto con una partícula radiactiva. La partícula tiene exactamente un 50% de posibilidades de fisionarse en dos. Si lo hace, el gato será asesinado. Si no es así, el gato estará bien. Entonces, ¿el gato está vivo o muerto? Según Schrödinger, la respuesta correcta es ambas. Cada vez que se produce una reacción subnuclear que tiene dos resultados posibles, el universo se clona. En uno, el evento ocurrió, en el otro no. El gato está vivo en un universo, muerto en otro. Solo cuando abres la caja sabes en qué universo estás.

No es de extrañar que codificar para el futuro sea difícil.

Pero piense en la evolución del código de la misma manera que una caja llena de gatos de Schrödinger: cada decisión resulta en una versión diferente del futuro. ¿Cuántos futuros posibles puede soportar tu código? ¿Cuáles son los más probables? ¿Qué tan difícil será apoyarlos cuando llegue el momento?

¿Te atreves a abrir la caja?

Trazador Balas

Preparados, disparen, apunten...

Hay dos formas de disparar una ametralladora en la oscuridad.⁵ Puedes averiguar exactamente dónde está tu objetivo (rango, elevación y acimut). Puede determinar las condiciones ambientales (temperatura, humedad, presión del aire, viento, etc.). Puede determinar las especificaciones precisas de los cartuchos y las balas que está utilizando, y sus interacciones con el arma real que está disparando. A continuación, puede utilizar tablas o un ordenador de tiro para calcular el rumbo y la elevación exactos del cañón. Si todo funciona exactamente como se especifica, las tablas son correctas y el entorno no cambia, las balas deberían aterrizar cerca de su objetivo.

O podría usar balas trazadoras.

Las balas trazadoras se cargan a intervalos en el cinturón de munición junto con la munición regular. Cuando se disparan, su fósforo se enciende y deja un rastro pirotécnico desde el arma hasta lo que sea que golpeen. Si los trazadores dan en el blanco, también lo hacen las balas normales.

No es sorprendente que las balas trazadoras sean preferidas al trabajo de cálculo. La retroalimentación es inmediata y, debido a que operan en el mismo entorno que la munición real, los efectos externos se minimizan.

La analogía puede ser violenta, pero se aplica a los nuevos proyectos, especialmente cuando estás construyendo algo que no se ha construido antes. Al igual que los artilleros, estás tratando de dar en el blanco en la oscuridad. Debido a que los usuarios nunca antes habían visto un sistema como este, sus requisitos pueden ser vagos. Debido a que puede estar utilizando algoritmos, técnicas, lenguajes o bibliotecas con los que no está familiarizado, se enfrenta a una gran cantidad de cosas desconocidas. Y debido a que los proyectos tardan en completarse, puede garantizar que el entorno en el que está trabajando cambiará antes de que termine.

La respuesta clásica es especificar el sistema hasta la muerte. Producir resmas de papel detallando cada requisito, atando cada incógnita, y

5. Para ser pedante, hay muchas formas de disparar una ametralladora en la

oscuridad, incluyendo cerrar los ojos y rociar balas. Pero esto es una analogía, y se nos permite tomarnos libertades.

restringiendo el medio ambiente. Dispara el arma usando la estimación muerta. Un gran cálculo al frente, luego disparar y esperar.

Los programadores pragmáticos, sin embargo, tienden a preferir el uso de balas trazadoras.

Código que brilla en la oscuridad

Las balas trazadoras funcionan porque funcionan en el mismo entorno y bajo las mismas restricciones que las balas reales. Llegan a la tar- se ponen rápido, por lo que el artillero recibe una respuesta inmediata. Y desde un punto de vista práctico, son una solución relativamente barata.

Para obtener el mismo efecto en el código, estamos buscando algo que nos lleve de un requisito a algún aspecto del sistema final de manera rápida, visible y repetible.

CONSE

Usa balas trazadoras para encontrar el objetivo

Una vez emprendimos un complejo proyecto de marketing de bases de datos cliente-servidor. Parte de su requisito era la capacidad de especificar y ejecutar consultas temporales. Los servidores eran una serie de bases de datos relacionales y especializadas. La GUI del cliente, escrita en Object Pascal, utilizaba un conjunto de bibliotecas C para proporcionar una interfaz a los servidores. La consulta del usuario se almacenaba en el servidor en una notación similar a Lisp antes de convertirse en SQL optimizado justo antes de la ejecución. Había muchas incógnitas y muchos entornos diferentes, y nadie estaba muy seguro de cómo debía comportarse la GUI.

Esta fue una gran oportunidad para usar el código trazador. Desarrollamos el marco para el front-end, las bibliotecas para representar las consultas y una estructura para convertir una consulta almacenada en una consulta específica de la base de datos. Luego lo juntamos todo y comprobamos que funcionaba. Para esa compilación inicial, todo lo que podíamos hacer era enviar una consulta que enumerara todas las filas de una tabla, pero demostró que la interfaz de usuario podía comunicarse con las bibliotecas, las bibliotecas podían serializar y deserializar una consulta y el servidor podía generar SQL a partir del resultado. A lo largo de los meses siguientes, fuimos desarrollando gradualmente esta estructura básica, añadiendo nuevas

funcionalidades al aumentar cada componente del código de seguimiento en paralelo. Cuando la interfaz de usuario agregó un nuevo tipo de consulta, la biblioteca creció y la generación de SQL se hizo más sofisticada.

El código de rastreo no es desecharable: lo escribes para siempre. Contiene toda la comprobación de errores, la estructuración, la documentación y la autocomprobación que tiene cualquier pieza de código de producción. Simplemente no es completamente funcional. Sin embargo, una vez que haya logrado una conexión de extremo a extremo entre los componentes de su sistema, puede verificar qué tan cerca del objetivo se encuentra, ajustándolo si es necesario. Una vez que esté en el objetivo, agregar funcionalidad es fácil.

El desarrollo de trazadores es coherente con la idea de que un proyecto nunca se termina: siempre se requerirán cambios y funciones que agregar. Es un enfoque incremental.

La alternativa convencional es una especie de enfoque de ingeniería pesada: el código se divide en módulos, que se codifican en el vacío. Los módulos se combinan en subconjuntos, que luego se combinan aún más, hasta que un día se tiene una aplicación completa. Sólo entonces se puede presentar al usuario y probar la aplicación en su conjunto.

El enfoque del código de seguimiento tiene muchas ventajas:

- **Los usuarios pueden ver que algo funciona desde el principio.** Si ha comunicado con éxito lo que está haciendo (véase *Great Expectations*, página 255), sus usuarios sabrán que están viendo algo inmúbito. No se sentirán decepcionados por la falta de funcionalidad; Estarán extasiados al ver algún progreso visible hacia su sistema. También pueden contribuir a medida que avanza el proyecto, lo que aumenta su aceptación. Es probable que estos mismos usuarios sean las personas que le dirán qué tan cerca del objetivo está cada iteración.
- **Los desarrolladores construyen una estructura para trabajar.** El pedazo de papel más desalentador es el que no tiene nada escrito en él. Si ha resuelto todas las interacciones de extremo a extremo de su aplicación y las ha incorporado en código, entonces su equipo no necesitará sacar tanto de la nada. Esto hace que todos sean más productivos y fomenta la constancia.
- **Tienes una plataforma de integración.** Como el sistema está conectado de extremo a extremo, tiene un entorno al que puede agregar nuevos fragmentos de código una vez que se han probado unitariamente. En lugar de tentar una integración de gran impacto, se integrará todos los días (a menudo muchas veces al día). El impacto de cada nuevo cambio es más evidente y las

interacciones son más limitadas, por lo que la depuración y las pruebas son más rápidas y precisas.

- **Tienes algo que demostrar.** Los patrocinadores de proyectos y los altos mandos tienden a querer ver demostraciones en los momentos más inoportunos. Con el código de seguimiento, siempre tendrás algo que mostrarles.
- **Tienes una mejor sensación del progreso.** En el desarrollo de un código trazador, los desarrolladores abordan los casos de uso uno por uno. Cuando se termina uno, pasan al siguiente. Es mucho más fácil medir el rendimiento y demostrar el progreso al usuario. Debido a que cada desarrollo individual es más pequeño, evita crear esos bloques monolíticos de código que se informan como completados en un 95% semana tras semana.

Las balas trazadoras no siempre dan en el blanco

Las balas trazadoras muestran lo que estás golpeando. Es posible que este no sea siempre el objetivo. A continuación, ajustas tu puntería hasta que estén en el objetivo. Ese es el punto.

Lo mismo ocurre con el código trazador. Utilizas la técnica en situaciones en las que no estás 100% seguro de hacia dónde vas. No debería sorprenderse si sus primeros intentos fallan: el usuario dice "eso no es lo que quise decir", o los datos que necesita no están disponibles cuando los necesita, o los problemas de rendimiento parecen probables. Averigüe cómo cambiar lo que tiene para acercarlo al objetivo y agradezca haber utilizado una metodología de desarrollo lean. Un cuerpo pequeño de código tiene baja inercia, es fácil y rápido de cambiar. Podrá recopilar información sobre su aplicación y generar una nueva versión más precisa de forma más rápida y a menor coste que con cualquier otro método. Y dado que todos los componentes principales de la aplicación están representados en el código de seguimiento, los usuarios pueden estar seguros de que lo que ven se basa en la realidad, no solo en una especificación en papel.

Código de seguimiento frente a la creación de prototipos

Se podría pensar que este concepto de código trazador no es más que la creación de prototipos con un nombre agresivo. Hay una diferencia. Con un prototipo, tu objetivo es explorar aspectos específicos del sistema final. Con un verdadero prototipo, tirarás a la basura lo que hayas atado al probar el concepto y lo recodificarás correctamente utilizando las lecciones que has aprendido.

Por ejemplo, supongamos que está produciendo una aplicación que

ayuda a los transportistas a determinar cómo empaquetar cajas de tamaño extraño en contenedores. Entre otros

La interfaz de usuario debe ser intuitiva y los algoritmos que se utilizan para determinar el embalaje óptimo son muy complejos.

Podría crear un prototipo de una interfaz de usuario para sus usuarios finales en una herramienta GUI. Codifica solo lo suficiente para que la interfaz responda a las acciones del usuario. Una vez que hayan aceptado el diseño, puede tirarlo a la basura y recodificarlo, esta vez con la lógica empresarial que hay detrás, utilizando el idioma de destino. Del mismo modo, es posible que desee crear un prototipo de una serie de algoritmos que realicen el empaquetado real. Puede codificar pruebas funcionales en un lenguaje indulgente de alto nivel como Perl, y codificar pruebas de rendimiento de bajo nivel en algo más cercano a la máquina. En cualquier caso, una vez que hayas tomado tu decisión, comenzarías de nuevo y codificarías los algoritmos en su entorno final, interactuando con el mundo real. Este es *un prototipo* y es muy útil.

El enfoque del código de seguimiento aborda un problema diferente. Necesita saber cómo se mantiene unida la aplicación en su conjunto. Desea mostrar a los usuarios cómo funcionarán las interacciones en la práctica y desea proporcionar a los desarrolladores un esqueleto arquitectónico en el que colgar el código. En este caso, se puede construir un trazador que consista en una implementación trivial del algoritmo de empaquetamiento del contenedor (tal vez algo así como por orden de llegada) y una interfaz de usuario simple pero funcional. Una vez que tenga todos los componentes de la aplicación conectados, tendrá un marco para mostrar a sus usuarios y desarrolladores. Con el tiempo, se agrega a este marco con nuevas funcionalidades, completando rutinas stub. Pero el marco permanece intacto y sabe que el sistema seguirá comportándose de la manera en que lo hizo cuando se completó su primer código de seguimiento.

La distinción es lo suficientemente importante como para justificar su repetición. La creación de prototipos genera código desecharable. El código Tracer es magro pero completo, y forma parte del esqueleto del sistema final. Piense en la creación de prototipos como el reconocimiento y la recopilación de inteligencia que tiene lugar antes de que se dispare una sola bala trazadora.

Las secciones relacionadas incluyen:

- *Good-Ewough Software*, página 9
- *Prototipos de notas Post-it*, página 53
- *El Specification Trap*, página 217

- *Great Expectations*, página 255

Prototipos y notas adhesivas

Muchas industrias diferentes utilizan prototipos para probar ideas específicas; La prototipificación es mucho más barata que la producción a gran escala. Los fabricantes de automóviles, por ejemplo, pueden construir muchos prototipos diferentes de un nuevo diseño de automóvil. Cada uno está diseñado para probar un aspecto específico del automóvil: la aerodinámica, el estilo, las características estructurales, etc. Tal vez se construya un modelo de arcilla para las pruebas en el túnel de viento, tal vez un modelo de madera de balsa y cinta adhesiva sirva para el departamento de arte, y así sucesivamente. Algunas compañías automotrices llevan esto un paso más allá, y ahora hacen una gran cantidad de trabajo de modelado en la computadora, reduciendo los costos aún más. De esta manera, se pueden probar elementos riesgosos o inciertos sin comprometerse a construir el artículo real.

Construimos prototipos de software de la misma manera, y por las mismas razones: para analizar y exponer el riesgo, y para ofrecer oportunidades de corrección a un costo muy reducido. Al igual que los fabricantes de automóviles, podemos apuntar a un prototipo para probar uno o más aspectos específicos de un proyecto.

Tendemos a pensar en los prototipos como basados en código, pero no siempre tienen por qué serlo. Al igual que los fabricantes de automóviles, podemos construir prototipos a partir de diferentes materiales. Las notas adhesivas son excelentes para crear prototipos de cosas dinámicas, como el flujo de trabajo y la lógica de la aplicación. Una interfaz de usuario puede ser prototipada como un dibujo en una pizarra, como una maqueta no funcional dibujada con un programa de pintura, o con un constructor de interfaces.

Los prototipos están diseñados para responder a unas pocas preguntas, por lo que son mucho más baratos y rápidos de desarrollar que las aplicaciones que se producen. El código puede ignorar detalles sin importancia, que no son importantes para usted en este momento, pero probablemente muy importantes para el usuario más adelante. Si está creando un prototipo de una interfaz gráfica de usuario, por ejemplo, puede salirse con la suya con resultados o datos incorrectos. Por otro lado, si solo está investigando aspectos comerciales o de rendimiento, puede salirse con la suya con una GUI bastante pobre, o tal vez incluso

sin GUI en absoluto.

Pero si te encuentras en un entorno en el que *no puedes* renunciar a los detalles, entonces debes preguntarte si realmente estás construyendo un prototipo. Tal vez un estilo de desarrollo de balas trazadoras sería más apropiado en este caso (ver *Balas Tracer*, página 48).

Cosas para prototipar

¿Qué tipo de cosas podrías elegir para investigar con un prototipo? Cualquier cosa que conlleve un riesgo. Cualquier cosa que no se haya intentado antes, o que sea absolutamente crítica para el sistema final. Cualquier cosa no probada, experimental o dudosa. Cualquier cosa con la que no te sientas cómodo. Puedes crear prototipos

- Arquitectura
- Nueva funcionalidad en un sistema existente • Estructura o contenido de datos externos
- Herramientas o componentes de terceros Problemas de • rendimiento
- Diseño de interfaz de usuario

La creación de prototipos es una experiencia de aprendizaje. Su valor no reside en el código producido, sino en las lecciones aprendidas. Ese es realmente el objetivo de la creación de prototipos.

CONSEJO
Prototipo para aprender

Cómo usar los prototipos

A la hora de construir un prototipo, ¿qué detalles se pueden ignorar?

- Corrección. Es posible que pueda utilizar datos ficticios cuando sea apropiado.
- Integridad. El prototipo puede funcionar solo en un sentido muy limitado, tal vez con solo una pieza preseleccionada de datos de entrada y un elemento de menú.
- Robustez. Es probable que la comprobación de errores esté incompleta o que falte por completo. Si te desvías del camino predefinido, el prototipo puede estrellarse y quemarse en una gloriosa exhibición de pirotecnia. Está bien.
- Estilo. Es doloroso admitirlo por escrito, pero es probable que el código del prototipo no tenga muchos comentarios o documentación. Puede producir montones de documentación

como resultado de su experiencia con el prototipo, pero comparativamente muy poca sobre el sistema prototipo en sí.

Dado que un prototipo debe pasar por alto los detalles y centrarse en aspectos específicos del sistema que se está considerando, es posible que desee implementar prototipos utilizando un lenguaje de muy alto nivel, superior al resto del proyecto (tal vez un lenguaje como Perl, Python o Tcl). Un lenguaje de scripting de alto nivel le permite diferir muchos detalles (incluida la especificación de tipos de datos) y aún así producir un fragmento de código funcional (aunque incompleto o lento).⁶ Si necesita crear prototipos de interfaces de usuario, investigue herramientas como Tcl/Tk, Visual Basic, Powerbuilder o Delphi.

Los lenguajes de scripting funcionan bien como el "pegamento" para combinar piezas de bajo nivel en nuevas combinaciones. En Windows, Visual Basic puede pegar controles COM. De manera más general, puede usar lenguajes como Perl y Python para vincular bibliotecas C de bajo nivel, ya sea a mano o automáticamente con herramientas como SWIG [URL 28] disponible gratuitamente. Con este enfoque, puede ensamblar rápidamente los componentes existentes en nuevas configuraciones para ver cómo funcionan las cosas.

Arquitectura de prototipado

Se construyen muchos prototipos para modelar todo el sistema bajo consideración. A diferencia de las balas trazadoras, ninguno de los módulos individuales del sistema prototipo necesita ser particularmente funcional. De hecho, es posible que ni siquiera necesites codificar para crear prototipos de arquitectura: puedes crear prototipos en una pizarra, con notas adhesivas o fichas. Lo que se busca es cómo el sistema se mantiene unido como un todo, de nuevo diferiendo los detalles. Estas son algunas áreas específicas que puede buscar en el prototipo arquitectónico:

- ¿Están bien definidas y son apropiadas las responsabilidades de los componentes principales?
- ¿Están bien definidas las colaboraciones entre los componentes principales? • ¿Se minimiza el acoplamiento?
- ¿Puede identificar posibles fuentes de duplicación?
- ¿Son aceptables las definiciones y restricciones de interfaz?

6. Si está investigando el rendimiento absoluto (en lugar de relativo), deberá ceñirse a un idioma que tenga un rendimiento similar al idioma de destino.

- ¿Cada módulo tiene una ruta de acceso a los datos que necesita durante la ejecución? ¿Tiene ese acceso *cuando* lo necesita?

Este último elemento suele generar la mayor cantidad de sorpresas y los resultados más valiosos de la experiencia de prototipado.

Cómo no usar prototipos

Antes de embarcarse en cualquier creación de prototipos basada en código, asegúrese de que todos entiendan que está escribiendo código desecharable. Los prototipos pueden ser engañosamente atractivos para las personas que no saben que son solo prototipos. Debe dejar *muy* claro que este código es desecharable, está incompleto y no se puede completar.

Es fácil dejarse engañar por la aparente integridad de un prototipo demostrado, y los patrocinadores o la dirección del proyecto pueden insistir en desplegar el prototipo (o su progenie) si no se establecen las expectativas adecuadas. Recuérdale que puede construir un gran prototipo de un automóvil nuevo con madera de balsa y cinta adhesiva, ipero no intentaría conducirlo en el tráfico de la hora pico!

Si cree que existe una gran posibilidad en su entorno o cultura de que se malinterprete el propósito del código del prototipo, es posible que le convenga más utilizar el enfoque de la bala trazadora. Terminarás con un marco sólido en el que basar el desarrollo futuro.

Cuando se utiliza correctamente, un prototipo puede ahorrarle enormes cantidades de tiempo, dinero, dolor y sufrimiento al identificar y corregir posibles problemas en las primeras etapas del ciclo de desarrollo, el momento en que corregir los errores es barato y fácil.

Las secciones relacionadas incluyen:

- *El gato se comió mi código fuente*, página 2
- *iComunicar!*, página 18
- *Balas trazadoras*, página 48
- *Grandes Esperanzas*, página 255

Ejercicios

4. Al equipo de marketing le gustaría sentarse y hacer una lluvia de ideas sobre algunos diseños de páginas web contigo. Están pensando en mapas de imágenes en los que se puede hacer clic para llevarlo a otras páginas, y así sucesivamente. Pero no pueden

decidir sobre un modelo para la imagen, tal vez

Es un coche, o un teléfono, o una casa. Tiene una lista de páginas y contenido de destino; Les gustaría ver algunos prototipos. Ah, por cierto, tienes 15 minutos. ¿Qué herramientas podrías utilizar?

12

Dominio Idiomas

Los límites del lenguaje son los límites del mundo de uno.

► **Ludwig Wittgenstein**

Los lenguajes informáticos influyen en *la forma* en que piensas sobre un problema y en cómo piensas sobre la comunicación. Cada lenguaje viene con una lista de características, palabras de moda como escritura estática frente a dinámica, enlace temprano frente a tardío, modelos de herencia (único, múltiple o ninguno), todas las cuales pueden sugerir u oscurecer ciertas soluciones. Diseñar una solución con Lisp en mente producirá resultados diferentes a una solución basada en el pensamiento de estilo C, y viceversa. Por el contrario, y creemos que lo que es más importante, el lenguaje del dominio del problema también puede sugerir una solución de programación.

Siempre intentamos escribir código utilizando el vocabulario del dominio de la aplicación (consulte *The Requirements Pit*, página 210, donde sugerimos usar un glosario de proyectos). En algunos casos, podemos pasar al siguiente nivel y programar utilizando el vocabulario, la sintaxis y la semántica (el lenguaje) del dominio.

Cuando escuche a los usuarios de un sistema propuesto, es posible que puedan decirle exactamente cómo debería funcionar el sistema:

Escuche las transacciones definidas por la Regulación ABC 12.3 en un conjunto de líneas X.25 , tradúzcalas al formato 43B de XYZ Company, retransmítalas en el enlace ascendente del satélite y guárdelas para análisis futuros.

Si sus usuarios tienen un número de estas declaraciones bien delimitadas, puede inventar un minilenguaje adaptado al dominio de la aplicación que exprese exactamente lo que quieren:

```
from X25LINE1 (format=ABC123) { put
    TELSTAR1 (format=XYZ43B);
    Tienda DB;
}
```

No es necesario que este lenguaje sea ejecutable. Inicialmente, podría ser simplemente una forma de capturar los requisitos del usuario, una especificación. Sin embargo, es posible que desee considerar dar un paso más allá y realmente implementar el lenguaje. La especificación se ha convertido en código ejecutable.

Una vez que haya escrito la aplicación, los usuarios le darán un nuevo requisito: las transacciones con saldos negativos no deben almacenarse y deben enviarse de vuelta en las líneas X.25 en el formato original:

```
De X25LINE1 (formato=ABC123) {
    if (ABC123.balance < 0) {
        Poner X25LINE1 (formato=ABC123);
    }
    else {
        Poner TELSTAR1 (formato=XYZ43B);
        Tienda DB;
    }
}
```

Eso fue fácil, ¿no? Con el soporte adecuado, puede programar mucho más cerca del dominio de la aplicación. No estamos sugiriendo que los usuarios finales programen realmente en estos idiomas. En su lugar, te estás dando a ti mismo una herramienta que te permite trabajar más cerca de su dominio.

CONSEJO

Programa cercano al dominio del problema

Ya sea que se trate de un lenguaje simple para configurar y controlar un programa de aplicación, o un lenguaje más complejo para especificar reglas o procedimientos, creemos que debe considerar formas de acercar su proyecto al dominio del problema. Al codificar en un nivel superior de abstracción, puede concentrarse en resolver problemas de dominio y puede ignorar los detalles insignificantes de la implementación.

Recuerde que hay muchos usuarios de una aplicación. Está el usuario final, que entiende las reglas de negocio y los resultados necesarios. También hay usuarios secundarios: personal de operaciones, gerentes de configuración y pruebas, programadores de soporte y mantenimiento, y futuras generaciones de desarrolladores. Cada uno de estos usuarios tiene su propio dominio de problemas, y puede

generar minientornos e idiomas para todos ellos.

Errores específicos del dominio

Si está escribiendo en el dominio del problema, también puede realizar una validación específica del dominio, informando de los problemas en términos que los usuarios puedan entender. Tome nuestra aplicación de cambio en la página opuesta. Supongamos que el usuario escribió mal el nombre del formato:

De X25LINE1 (Formato=AB123)

Si esto sucediera en un lenguaje de programación estándar de propósito general, es posible que reciba un mensaje de error estándar de propósito general:

Error de sintaxis: identificador no declarado

Pero con un mini-idioma, en su lugar podría emitir un mensaje de error utilizando el vocabulario del dominio:

"AB123" no es un formato. Los formatos conocidos son
ABC123, XYZ43B, PDQB y 42.

Implementación de un minilenguaje

En su forma más simple, un mini-lenguaje puede estar en un formato orientado a líneas, fácil de analizar. En la práctica, probablemente usamos esta forma más que cualquier otra. Se puede analizar simplemente usando instrucciones switch o usando expresiones regulares en lenguajes de scripting como Perl. La respuesta al Ejercicio 5 en la página 281 muestra una implementación simple en C.

También puede implementar un lenguaje más complejo, con una sintaxis más formal. El truco aquí es definir primero la sintaxis usando una notación como BNF.⁷ Una vez que haya especificado su gramática, normalmente es trivial convertirla en la sintaxis de entrada para un generador de analizadores. Los programadores de C y C++ han estado utilizando yacc (o su implementación disponible gratuitamente, bison [URL 27]) durante años. Estos programas están documentados en detalle en el libro *Lex awd Yacc* [LMB92]. Los programadores de Java pueden probar javacc, que se puede encontrar en [URL 26]. La respuesta al Ejercicio 7 en la página 282

7. BNF, o Backus-Naur Form, le permite especificar *gramáticas sin contexto* de forma recursiva. Cualquier buen libro sobre la construcción o el análisis sintáctico de compiladores cubrirá BNF con (exhaustivo) detalle.

Muestra un analizador sintáctico escrito con bison. Como se muestra, una vez que conoces la sintaxis, realmente no es mucho trabajo escribir minilenguajes simples.

Hay otra forma de implementar un minilenguaje: extender uno existente. Por ejemplo, podría integrar la funcionalidad a nivel de aplicación con (digamos) Python [URL 9] y escribir algo como8

```
registro = X25LINE1.get(formato=ABC123)
if (record.balance < 0):
    X25LINE1.put(registro, formato=ABC123)
De lo contrario:
    TELSTAR1.put(registro, formato=XYZ43B)
    DB.store(registro)
```

Lenguajes de datos y lenguajes imperativos

Los lenguajes que implemente se pueden usar de dos maneras diferentes.

Data lawguages producen algún tipo de estructura de datos utilizada por una aplicación. Estos lenguajes se utilizan a menudo para representar información de configuración.

Por ejemplo, el programa sendmail se utiliza en todo el mundo para enrutar el correo electrónico a través de Internet. Tiene muchas características y beneficios excelentes, que están controlados por un archivo de configuración de mil líneas, escrito utilizando el lenguaje de configuración propio de sendmail:

```
Mlocal, P=/usr/bin/procmail,
F=lsDFMAv5:/|@qSPfhn9,
S=10/30, R=20/40,
T=DNS/RFC822/X-Unix,
A=procmail -Y -a $h -d $u
```

Obviamente, la legibilidad no es uno de los puntos fuertes de sendmail.

Durante años, Microsoft ha estado utilizando un lenguaje de datos que puede describir menús, widgets, cuadros de diálogo y otros recursos de Windows. La figura 2.2 de la página siguiente muestra un extracto de un archivo de recursos típico. Esto es mucho más fácil de leer que el ejemplo de sendmail , pero se usa exactamente de la misma manera: se compila para generar una estructura de datos.

Los lawguages imperativos llevan esto un paso más allá. Aquí el lenguaje se ejecuta realmente, por lo que puede contener instrucciones, construcciones de control y similares (como el script de la página 58).

8. Gracias a Eric Vought por este ejemplo.

Figura 2.2. Archivo .rc de

```

MAIN_MENU MENÚ
{
    VENTANA EMERGENTE "&Archivo"
    {
        MENUITEM "&New", CM_FILENEW
        MENUITEM "&Abrir...", CM_FILEOPEN
        MENUITEM "&Save", CM_FILESAVE
    }
}

MY_DIALOG_BOX DIÁLOGO 6, 15, 292, 287
ESTILO DS_MODALFRAME | WS_POPUP | WS_VISIBLE |
WS_CAPTION | WS_SYSMENU
CAPTION "Mi cuadro de
diálogo" FONT 8, "MS Sans
Serif"
{
    DEFPULSADOR "OK", ID_OK, 232, 16, 50, 14
    PULSADOR "Ayuda", ID_HELP, 232, 52, 50, 14
    CONTROL "Edit Text Control", ID_EDIT1,
    "EDITAR", WS_BORDER | WS_TABSTOP, 16, 16, 80, 56
    CASILLA DE VERIFICACIÓN "Casilla de verificación",
    ID_CHECKBOX1, 153, 65, 42, 38, BS_AUTOCHECKBOX |

```

También puede utilizar sus propios lenguajes imperativos para facilitar el mantenimiento del programa. Por ejemplo, es posible que se le pida que integre información de una aplicación heredada en su nuevo desarrollo de GUI. Una forma común de lograr esto es mediante *screen* o *scrapiwg*; su aplicación se conecta a la aplicación del mainframe como si fuera un usuario humano normal, emitiendo pulsaciones de teclas y "leyendo" las respuestas que recibe. Podría escribir la interacción utilizando un minilenguaje.⁹

```

localice el mensaje "SSN:"
Escriba "%s"
social_security_number escriba
enter
Esperapara KeyboardUnlock
si text_at(10,14) es "SSN NO VÁLIDO" devuelve
bad_ssn si text_at(10,14) es "SSN DUPLICADO"
devuelve dup_ssn # etc...

```

Cuando la aplicación determina que es hora de ingresar un número de Seguro Social, invoca al intérprete en este script, que luego controla

9. De hecho, puedes comprar herramientas que admitan solo este tipo de scripting. También puede investigar paquetes de código abierto como Expect, que proporcionan capacidades similares [URL 24].

la transacción. Si el intérprete está integrado en la aplicación, los dos pueden incluso compartir datos directamente (por ejemplo, a través de un mecanismo de devolución de llamada).

Aquí estás programando en el dominio del programador de mantenimiento. Cuando la aplicación de mainframe cambia y los campos se mueven, el programador puede simplemente actualizar su descripción de alto nivel, en lugar de arrastrarse en los detalles del código C.

Lenguajes autónomos e integrados

Un minilenguaje no tiene que ser utilizado directamente por la aplicación para que sea útil. Muchas veces podemos usar un lenguaje de especificación para crear artefactos (incluyendo metadatos) que son compilados, leídos o utilizados de otra manera por el propio programa (ver *Metaprogramming*, página 144).

Por ejemplo, en la página 100 describimos un sistema en el que usamos Perl para generar un gran número de derivaciones a partir de una especificación de esquema original. Inventamos un lenguaje común para expresar el esquema de la base de datos y luego generamos todas las formas que necesitábamos: SQL, C, páginas web, XML y otros. La aplicación no utilizaba la especificación directamente, sino que se basaba en los resultados producidos a partir de ella.

Es habitual incrustar lenguajes imperativos de alto nivel directamente en la aplicación, de modo que se ejecuten cuando se ejecute el código. Esta es claramente una capacidad poderosa; Puede cambiar el comportamiento de su aplicación cambiando los scripts que lee, todo sin compilar. Esto puede simplificar significativamente el mantenimiento en un dominio de aplicación dinámico.

¿Fácil desarrollo o fácil mantenimiento?

Hemos analizado varias gramáticas diferentes, que van desde formatos simples orientados a líneas hasta gramáticas más complejas que parecen lenguas reales. Dado que se necesita un esfuerzo adicional para implementarlo, ¿por qué elegiría una gramática más compleja?

La contrapartida es la extensibilidad y el mantenimiento. Si bien el código para analizar un lenguaje "real" puede ser más difícil de escribir, será mucho más fácil de entender para las personas y ampliarlo en el futuro con nuevas características y funcionalidades.

Los lenguajes que son demasiado simples pueden ser fáciles de analizar, pero pueden ser crípticos, al igual que el ejemplo de sendmail en la página 60.

Dado que la mayoría de las aplicaciones superan su vida útil esperada, probablemente sea mejor que muerda la bala y adopte el lenguaje más complejo y legible desde el principio. El esfuerzo inicial se verá recompensado muchas veces con costos de soporte y mantenimiento reducidos.

Las secciones relacionadas incluyen:

- *Metaprogrammiwg*, página 144

Desafíos

- ¿Podrían algunos de los requisitos de su proyecto actual expresarse en un lenguaje específico de dominio? ¿Sería posible escribir un compilador o traductor que pudiera generar la mayor parte del código necesario?
- Si decides adoptar minilenguajes como una forma de programar más cerca del dominio del problema, estás aceptando que se requerirá algún esfuerzo para implementarlos. ¿Puedes ver formas en las que el marco que desarrollas para un proyecto se puede reutilizar en otros?

Ejercicios

5. Queremos implementar un mini-lenguaje para controlar un paquete de dibujo simple-
Awsuer
 (tal vez un sistema de gráficos de tortuga). El lenguaje consta de comandos de una sola letra. Algunos comandos van seguidos de un solo número. Por ejemplo, en la siguiente entrada se dibujaría un rectángulo.
ow p. 281

```
P 2 # seleccionar bolígrafo 2
D      # pluma
W 2 # dibujar hacia el
oeste 2 cm N 1 # luego
norte 1 E 2 # luego este
2
S 1 # luego de vuelta al sur
U      # bolígrafo
```

Implemente el código que analiza este lenguaje. Debe estar diseñado para que sea sencillo agregar nuevos comandos.

6. Diseñe una gramática BNF para analizar una especificación de tiempo. Todo lo siguiente
Awsuer
 Deben aceptarse ejemplos.
ow p. 282

16h, 19:38h, 23:42, 15:16, 3:16

7. Implemente un analizador para la gramática BNF en el Ejercicio 6 usando Yacc, bisonte,
Awsuer
 o un analizador-generador
ow p. 282
 similar.
8. Implemente el analizador de tiempo usando Perl. [Alto: Las expresiones regulares hacen

buenos
analizadores.]

Estimar

¡Rápido! ¿Cuánto tiempo se tarda en enviar Guerra y Paz a través de una línea de módem de 56k? ¿Cuánto espacio en disco necesitará para un millón de nombres y direcciones? ¿Cuánto tiempo tarda un bloque de 1.000 bytes en pasar por un router? ¿Cuántos meses tardará en entregar su proyecto?

En cierto nivel, todas estas son preguntas sin sentido, todas son información faltante. Y, sin embargo, todas pueden ser contestadas, siempre y cuando se haga una estimación cómoda. Y, en el proceso de producir una estimación, llegará a comprender más sobre el mundo en el que habitan sus programas.

Al aprender a estimar, y al desarrollar esta habilidad hasta el punto en que tengas una sensación intuitiva de las magnitudes de las cosas, podrás mostrar una aparente habilidad mágica para determinar su viabilidad. Cuando alguien dice "enviaremos la copia de seguridad a través de una línea ISDN al sitio central", podrá saber intuitivamente si esto es práctico. Cuando estés codificando, podrás saber qué subsistemas necesitan optimización y cuáles se pueden dejar solos.

CONSEJO

Estimación para evitar sorpresas

Como beneficio adicional, al final de esta sección revelaremos la única respuesta correcta que debe dar cada vez que alguien le pida una estimación.

¿Qué tan preciso es lo suficientemente preciso?

Hasta cierto punto, todas las respuestas son estimaciones. Es solo que algunos son más precisos que otros. Así que la primera pregunta que tienes que hacerte cuando alguien te pide un presupuesto es el contexto en el que se tomará tu respuesta. ¿Necesitan alta precisión o están buscando una cifra aproximada?

- Si tu abuela te pregunta cuándo llegarás, probablemente no esté dudando si prepararte el almuerzo o la cena. Por otro lado, un buzo atrapado bajo el agua y quedándose sin aire probablemente esté interesado en una respuesta hasta el segundo.

- ¿Cuál es el valor de π ? Si te preguntas cuánto borde comprar para poner alrededor de un macizo de flores circular, entonces "3" probablemente sea lo suficientemente bueno.¹⁰ Si estás en la escuela, entonces tal vez "7" sea una buena aproximación. Si estás en la NASA, entonces tal vez 12 decimales sean suficientes.

Una de las cosas interesantes de la estimación es que las unidades que utilizas marcan la diferencia en la interpretación del resultado. Si dices que algo tardará unos 130 días laborables, entonces la gente esperará que se acerque bastante. Sin embargo, si dices "Oh, alrededor de seis meses", entonces saben que deben buscarlo en cualquier momento entre cinco y siete meses a partir de ahora. Ambos números representan la misma duración, pero "130 días" probablemente implique un mayor grado de precisión de lo que crees. Le recomendamos que escale las estimaciones de tiempo de la siguiente manera:

Duración	Estimación de
cotización en 1–15 días	
	Días
De 3 a 8 semanas	Semanas
8–30 Semanas	Meses
30+ Semanas	Píénsalo bien antes de dar un presupuesto

Por lo tanto, si después de hacer todo el trabajo necesario, decide que un proyecto durará 125 días hábiles (25 semanas), es posible que desee entregar un estimado de "unos seis meses".

Los mismos conceptos se aplican a las estimaciones de cualquier cantidad: elija las unidades de su respuesta para reflejar la precisión que pretende transmitir.

¿De dónde vienen las estimaciones?

Todas las estimaciones se basan en modelos del problema. Pero antes de profundizar demasiado en las técnicas de construcción de modelos, tenemos que mencionar un truco básico de estimación que siempre da buenas respuestas: preguntar a alguien que ya lo haya hecho. Antes de comprometerse demasiado con la construcción de modelos, busca a alguien que haya estado en una situación similar en el pasado.

^{10.} "3" también es aparentemente lo suficientemente bueno si eres un legislador. En

1897, el proyecto de ley No. 246 de la Legislatura del Estado de Indiana intentó decretar que en adelante π debería tener el valor de "3". El proyecto de ley se presentó indefinidamente en su segunda lectura cuando un profesor de matemáticas señaló que sus poderes no se extendían completamente a la aprobación de leyes de la naturaleza.

Vea cómo se resolvió su problema. Es poco probable que alguna vez encuentres una coincidencia exacta, pero te sorprendería la cantidad de veces que puedes aprovechar con éxito las experiencias de otros.

Entender lo que se pregunta

La primera parte de cualquier ejercicio de estimación es construir una comprensión de lo que se está preguntando. Además de los problemas de precisión mencionados anteriormente, debe comprender el alcance del dominio. A menudo, esto está implícito en la pregunta, pero debe acostumbrarse a pensar en el alcance antes de comenzar a adivinar. A menudo, el alcance que elijas formará parte de la respuesta que des: "Suponiendo que no haya accidentes de tráfico y que haya gasolina en el coche, debería estar allí en 20 minutos".

Construir un modelo del sistema

Esta es la parte divertida de la estimación. A partir de tu comprensión de la pregunta que se te hace, construye un modelo mental aproximado y básico. Si está estimando los tiempos de respuesta, su modelo puede implicar un servidor y algún tipo de tráfico entrante. Para un proyecto, el modelo puede ser los pasos que su organización utiliza durante el desarrollo, junto con una imagen muy aproximada de cómo se podría implementar el sistema.

La construcción de modelos puede ser creativa y útil a largo plazo. A menudo, el proceso de construcción del modelo conduce a descubrimientos de patrones y procesos subyacentes que no eran evidentes en la superficie. Es posible que incluso desee volver a examinar la pregunta original: "Usted pidió un esti- mate para hacer X . Sin embargo, parece que Y , una variante de X , podría hacerse en aproximadamente la mitad del tiempo, y solo se pierde una característica".

La construcción del modelo introduce imprecisiones en el proceso de estimación. Esto es inevitable, y también beneficioso. Está sacrificando la simplicidad del modelo por la precisión. Duplicar el esfuerzo en el modelo puede darle solo un ligero aumento en la precisión. Tu experiencia te dirá cuándo dejar de refinar.

Dividir el modelo en componentes

Una vez que tenga un modelo, puede descomponerlo en componentes. Tendrás que descubrir las reglas matemáticas que describen cómo

interactúan estos componentes. A veces, un componente aporta un solo valor

que se añade al resultado. Algunos componentes pueden proporcionar factores multiplicadores, mientras que otros pueden ser más complicados (como los que simulan la llegada del tráfico a un nodo).

Descubrirá que cada componente suele tener parámetros que afectan a la forma en que contribuye al modelo general. En esta etapa, simplemente identifique cada parámetro.

Asigne un valor a cada parámetro

Una vez que haya desglosado los parámetros, puede revisarlos y asignarle un valor a cada uno. Espera introducir algunos errores en este paso. El truco consiste en averiguar qué parámetros tienen el mayor impacto en el resultado y concentrarse en hacerlo bien. Normalmente, los parámetros cuyos valores se agregan a un resultado son menos significativos que los que se multiplican o dividen. Duplicar la velocidad de una línea puede duplicar la cantidad de datos recibidos en una hora, mientras que agregar un retraso de tránsito de 5 ms no tendrá ningún efecto notable.

Debería tener una forma justificable de calcular estos parámetros críticos. Para el ejemplo de colas, es posible que desee medir la tasa real de llegada de transacciones del sistema existente o encontrar un sistema similar para medir. Del mismo modo, puede medir el tiempo actual que se tarda en atender una solicitud o elaborar una estimación utilizando las técnicas descritas en esta sección. De hecho, a menudo te encontrarás basando una estimación en otras subestimaciones. Aquí es donde se arrastrarán sus errores más grandes.

Calcula las respuestas

Solo en el más simple de los casos una estimación tendrá una única respuesta. Es posible que te alegre decir: "Puedo caminar cinco cuadras a través de la ciudad en 15 minutos". Sin embargo, a medida que los sistemas se vuelven más complejos, querrá cubrir sus respuestas. Ejecute múltiples cálculos, variando los valores de los parámetros críticos, hasta que determine cuáles son los que realmente impulsan el modelo. Una hoja de cálculo puede ser de gran ayuda. A continuación, formula tu respuesta en términos de estos parámetros. "El tiempo de respuesta es de aproximadamente tres cuartos de segundo si el sistema tiene un bus SCSI y 64 MB de memoria, y un segundo con 48 MB de memoria". (Observe cómo "tres cuartos de segundo" transmite una sensación de precisión diferente a la de 750

ms).

Durante la fase de cálculo, es posible que empieces a recibir respuestas que parezcan extrañas. No te apresures a descartarlos. Si su aritmética es correcta, su comprensión del problema o su modelo probablemente sea incorrecta. Esta es una información valiosa.

Realice un seguimiento de su destreza en la estimación

Creemos que es una gran idea registrar sus estimaciones para que pueda ver qué tan cerca estaba. Si una estimación general implicó calcular subestimaciones, también realice un seguimiento de estas. A menudo encontrarás que tus estimaciones son bastante buenas, de hecho, después de un tiempo, llegarás a esperar esto.

Cuando una estimación resulta ser incorrecta, no te encojas de hombros y te vayas. Averigüe por qué difiere de su suposición. Tal vez elegiste algunos parámetros que no coincidían con la realidad del problema. Tal vez tu modelo estaba equivocado. Cualquiera que sea la razón, tómese un tiempo para descubrir lo que sucedió. Si lo haces, tu próxima estimación será mejor.

Estimación de los plazos de los proyectos

Las reglas normales de estimación pueden romperse ante las complejidades y caprichos del desarrollo de una aplicación considerable. Encontramos que a menudo la única manera de determinar el calendario de un proyecto es adquiriendo experiencia en ese mismo proyecto. Esto no tiene por qué ser una paradoja si practicas el desarrollo incremental, repitiendo los siguientes pasos.

- Comprobar requisitos
- Analice el riesgo
- Diseñar, implementar, integrar
- Validar con los usuarios

Inicialmente, es posible que solo tenga una vaga idea de cuántas iteraciones se requerirán o cuánto tiempo pueden ser. Algunos métodos requieren que concretes esto como parte del plan inicial, pero para todos los proyectos, excepto los más triviales, esto es un error. A menos que estés haciendo una aplicación similar a una anterior, con el mismo equipo y la misma tecnología, solo estarías adivinando.

Por lo tanto, complete la codificación y las pruebas de la funcionalidad inicial y marque esto como el final del primer incremento. En función de esa experiencia, puede refinar su estimación inicial sobre el número de iteraciones y qué

se puede incluir en cada uno. El refinamiento mejora cada vez, y la confianza en el horario crece junto con él.

CONSEJO

Iterar la programación con el código

Es posible que esto no sea popular entre la gerencia, que generalmente desea un número único y duro antes de que el proyecto comience. Tendrá que ayudarles a entender que el equipo, su productividad y el entorno determinarán el horario. Al formalizar esto y refinar la programación como parte de cada iteración, les dará las estimaciones de programación más precisas que pueda.

Qué decir cuando se le pide un presupuesto

Dices: *"Te voy a poner en contacto contigo"*.

Casi siempre se obtienen mejores resultados si se ralentiza el proceso y se dedica algún tiempo a seguir los pasos que describimos en esta sección. Las estimaciones dadas en la máquina de café (como el café) volverán a perseguirte.

Las secciones relacionadas incluyen:

- *Velocidad del algoritmo*, página 177

Desafíos

- Comience a llevar un registro de sus estimaciones. Para cada uno, haz un seguimiento de la precisión que resultaste ser. Si su error fue superior al 50%, intente averiguar dónde se equivocó su estimación.

Ejercicios

9. Se le pregunta "¿Qué tiene un ancho de banda más alto: un 1Mbps de comunicaciones

¿O una persona que camina entre dos computadoras con una cinta completa de 4 GB en el bolsillo?" ¿Qué restricciones pondrá a su respuesta para asegurarse de que el alcance de su respuesta sea correcto? (Por ejemplo, podría decir que se omite el tiempo que se tarda en acceder a la cinta).

Awsuer
ow p. 283

10. Entonces, ¿cuál tiene el mayor ancho de banda?

Awsuer
ow p. 284

Esta página se ha dejado en blanco intencionadamente

Capítulo 3

Las herramientas básicas

Cada artesano comienza su viaje con un conjunto básico de herramientas de buena calidad. Un carpintero puede necesitar reglas, calibres, un par de sierras, algunos buenos cepillos, cinceles finos, taladros y tirantes, mazos y abrazaderas. Estas herramientas serán elegidas con cuidado, estarán construidas para durar, realizarán trabajos específicos con poca superposición con otras herramientas y, quizás lo más importante, se sentirán bien en las manos del carpintero en ciernes.

Entonces comienza un proceso de aprendizaje y adaptación. Cada herramienta tendrá su propia personalidad y peculiaridades, y necesitará su propio manejo especial. Cada uno debe afilarse de una manera única, o sostenerse así. Con el tiempo, cada uno se desgastará de acuerdo con el uso, hasta que el agarre parezca un molde de las manos del carpintero y la superficie de corte se alinee perfectamente con el ángulo en el que se sostiene la herramienta. En este punto, las herramientas se convierten en conductos desde el cerebro del artesano hasta el producto terminado: se han convertido en extensiones de sus manos. Con el tiempo, el carpintero agregará nuevas herramientas, como cortadores de galletas, sierras ingletadoras guiadas por láser, plantillas de cola de milano, todas piezas maravillosas de tecnología. Pero puedes apostar a que él o ella será más feliz con una de esas herramientas originales en la mano, sintiendo el canto del avión mientras se desliza por la madera.

Las herramientas amplifican tu talento. Cuanto mejores sean tus herramientas y mejor sepas cómo usarlas, más productivo podrás ser. Comience con un conjunto básico de herramientas de aplicación general. A medida que ganes experiencia, y a medida que te encuentres con requisitos especiales, irás ampliando este conjunto básico. Al igual que el artesano, espere agregar a su caja de herramientas con regularidad. Esté siempre atento a las mejores formas de hacer las cosas. Si te encuentras con una situación en la que sientes que tus

herramientas actuales no pueden ser suficientes, toma nota de lo que debes buscar

algo diferente o más poderoso que hubiera ayudado. Deje que la necesidad impulse sus adquisiciones.

Muchos programadores nuevos cometan el error de adoptar una sola herramienta poderosa, como un entorno de desarrollo integrado (IDE) particular, y nunca abandonan su acogedora interfaz. Esto es realmente un error. Necesitamos sentirnos cómodos más allá de los límites impuestos por un IDE. La única forma de hacer esto es mantener el conjunto de herramientas básicas afilado y listo para usar.

En este capítulo hablaremos sobre cómo invertir en tu propia caja de herramientas básicas. Al igual que con cualquier buena discusión sobre herramientas, comenzaremos (en *The Power of Plain Text*) observando sus materiales en bruto, las cosas a las que dará forma. De ahí pasaremos a la mesa de trabajo, o en nuestro caso al ordenador. ¿Cómo puedes usar tu computadora para aprovechar al máximo las herramientas que usas? Discutiremos esto en *Shell Games*. Ahora que tenemos material y un banco para trabajar, vamos a recurrir a la herramienta que probablemente usarás más que cualquier otra, tu editor. En *Power Editing*, te sugeriremos formas de hacerte más eficiente.

Para asegurarnos de que nunca perdemos nada de nuestro precioso trabajo, siempre debemos usar un *sistema de código fuente Control*, incluso para cosas como nuestra libreta de direcciones personal! Y, dado que el Sr. Murphy era realmente un optimista después de todo, no puedes ser un gran programador hasta que te conviertas en un experto en *Debugging*.

Necesitarás un poco de pegamento para unir gran parte de la magia. Discutimos algunas posibilidades, como awk, Perl y Python, en *Text Manipulation*.

Al igual que los carpinteros a veces construyen plantillas para guiar la construcción de piezas complejas, los programadores pueden escribir código que a su vez escribe código. Discutimos esto en *Code Generators*.

Dedique tiempo a aprender a usar estas herramientas, y en algún momento se sorprenderá al descubrir que sus dedos se mueven sobre el teclado, manipulando el texto sin pensar conscientemente. Las herramientas se habrán convertido en extensiones de tus manos.

El poder del texto sin formato

Como programadores pragmáticos, nuestro material base no es la madera o el hierro, es el conocimiento. Recopilamos los requisitos como conocimiento y luego expresamos ese conocimiento en nuestros diseños, implementaciones, pruebas y documentos. Y creemos que el mejor formato para almacenar el conocimiento de forma persistente es el *texto plaiw*. Con el texto plano, nos damos la capacidad de manipular el conocimiento, tanto de forma manual como programática, utilizando prácticamente todas las herramientas a nuestra disposición.

¿Qué es el texto sin formato?

El texto plaiw se compone de caracteres imprimibles en una forma que puede ser leída y entendida directamente por las personas. Por ejemplo, aunque el siguiente fragmento se compone de caracteres imprimibles, no tiene sentido.

```
Campo19=467abe
```

El lector no tiene idea de cuál puede ser el significado de 467abe . Una mejor opción sería hacerlo *más fácil* para los humanos.

```
DrawingType=UMLActivityDrawing
```

El texto sin formato no significa que el texto no esté estructurado; XML, SGML y HTML son excelentes ejemplos de texto sin formato que tiene una estructura bien definida. Puede hacer todo con texto sin formato que podría hacer con algún formato binario, incluido el control de versiones.

El texto sin formato tiende a estar en un nivel más alto que una codificación binaria directa, que generalmente se deriva directamente de la implementación. Supongamos que desea almacenar una propiedad denominada `uses_menus` que puede ser TRUE o FALSE. Usando texto, puedes escribir esto como

```
myprop.uses_menus=FALSE
```

Contrasta esto con 0010010101110101.

El problema con la mayoría de los formatos binarios es que el contexto necesario para comprender los datos es independiente de los datos en sí. Está divorciando artificialmente los datos de su significado. Los datos también pueden estar encriptados; No tiene ningún sentido sin la lógica de la aplicación para analizarlo. Sin embargo, con texto sin

formato, puede lograr un flujo de datos autodescriptivo que sea independiente de la aplicación que lo creó.

CONSEJO

Mantenga el conocimiento en texto sin formato

Inconvenientes

Hay dos inconvenientes principales en el uso de texto sin formato: (1) puede tomar más espacio para almacenar que un formato binario comprimido y (2) puede ser computacionalmente más costoso interpretar y procesar un archivo de texto sin formato.

Dependiendo de la aplicación, una o ambas de estas situaciones pueden ser inaceptables, por ejemplo, al almacenar datos de telemetría por satélite o como formato interno de una base de datos relacional.

Pero incluso en estas situaciones, puede ser aceptable almacenar *metadata* sobre los datos brutos en texto plano (véase *Metaprogramming*, página 144).

A algunos desarrolladores les puede preocupar que, al poner los metadatos en texto sin formato, los expongan a los usuarios del sistema. Este miedo está fuera de lugar. Los datos binarios pueden ser más oscuros que el texto sin formato, pero no son más seguros. Si te preocupa que los usuarios vean las contraseñas, cifrúdelas. Si no desea que cambien los parámetros de configuración, incluya un *hash1 seguro* de todos los valores de los parámetros en el archivo como suma de comprobación.

El poder del texto

Dado que *larger* y *slower* no son las características más solicitadas por los usuarios, ¿por qué molestarse con el texto sin formato? ¿Cuáles *son* los beneficios?

- Seguro contra la obsolescencia •
- Apalancamiento
- Pruebas más sencillas

Seguro contra la obsolescencia

Las formas de datos legibles por el ser humano, y los datos autodescriptivos, sobrevivirán a todas las demás formas de datos y a las aplicaciones que los crearon. Periodo.

1. MD5 se utiliza a menudo para este propósito. Para una excelente introducción al maravilloso mundo de la criptografía, véase [Sch95].

Mientras los datos sobrevivan, tendrá la oportunidad de poder usarlos, posiblemente mucho después de que la aplicación original que los escribió haya desaparecido.

Puede analizar un archivo de este tipo con solo un conocimiento parcial de su formato; Con la mayoría de los archivos binarios, debe conocer todos los detalles de todo el formato para analizarlo con éxito.

Considere un archivo de datos de algún sistema heredado² que se le proporcione. Sabe poco sobre la aplicación original; todo lo que es importante para usted es que mantiene una lista de números de Seguro Social de los clientes, que necesita encontrar y extraer. Entre los datos, se ve

```
<FIELD10>123-45-6789</FIELD10>
--->
<FIELD10>567-89-0123</FIELD10>
--->
<FIELD10>901-23-4567</FIELD10>
```

Al reconocer el formato de un número de Seguro Social, puede escribir rápidamente un pequeño programa para extraer esos datos, incluso si no tiene información sobre nada más en el archivo.

Pero imagina que el archivo se hubiera formateado de esta manera:

```
Una C 27123456789B1
1P
XY43567890123QTYL
--->
6T2190123456788AM
```

Es posible que no hayas reconocido el significado de los números tan fácilmente. Esta es la diferencia entre *humaw readable* y *humaw uwderstawdable*.

Y ya FIELD10 hacemos, tampoco ayuda mucho. Algo así como

```
<SSNO>123-45-6789</SSNO>
```

hace que el ejercicio sea una obviedad y garantiza que los datos sobrevivirán a cualquier proyecto que los haya creado.

Apalancamiento

Prácticamente todas las herramientas del universo informático, desde los sistemas de gestión de código fuente hasta los entornos de compilación, pasando por los editores y los filtros independientes, pueden funcionar con texto plano.

2. Todo el software se convierte en heredado tan pronto como se escribe.

La filosofía Unix

Unix es famoso por estar diseñado en torno a la filosofía de herramientas pequeñas y afiladas, cada una destinada a hacer una cosa bien. Esta filosofía se habilita mediante el uso de un formato subyacente común: el archivo de texto sin formato orientado a líneas. Las bases de datos utilizadas para la administración del sistema (usuarios y contraseñas, configuración de redes, etc.) se mantienen como archivos de texto sin formato. (Algunos sistemas, como Solaris, también mantienen una forma binaria de ciertas bases de datos como optimización del rendimiento. La versión de texto sin formato se mantiene como una interfaz para la versión binaria).

Cuando un sistema se bloquea, es posible que solo se enfrente a un entorno mínimo para restaurarlo (es posible que no pueda acceder a los controladores de gráficos, por ejemplo). Situaciones

Por ejemplo, supongamos que tiene una implementación de producción de una aplicación grande con un archivo de configuración complejo específico del sitio (le viene a la mente sendmail). Si este archivo está en texto plano, puede colocarlo bajo un sistema de control de código fuente (consulte *Source Code Control*, página 86), de modo que mantenga automáticamente un historial de todos los cambios. Las herramientas de comparación de archivos, como `diff` y `fc`, le permiten ver de un vistazo qué cambios se han realizado, mientras que `sum` le permite generar una suma de comprobación para monitorear el archivo en busca de modificaciones accidentales (o maliciosas).

Pruebas más sencillas

Si utiliza texto sin formato para crear datos sintéticos para impulsar las pruebas del sistema, entonces es una cuestión sencilla agregar, actualizar o modificar los datos de prueba *sin que pueda crear herramientas específicas para hacerlo*. De manera similar, la salida de texto plano de las pruebas de regresión se puede analizar trivialmente (con `diff`, por ejemplo) o sujeta a un escrutinio más exhaustivo con Perl, Python o alguna otra herramienta de scripting.

Mínimo común denominador

Incluso en el futuro de los agentes inteligentes basados en XML que viajan por la salvaje y peligrosa Internet de forma autónoma, negociando el intercambio de datos entre ellos, el archivo de texto

omnipresente seguirá ahí. De hecho, en

Entornos heterogéneos: las ventajas del texto sin formato pueden superar todos los inconvenientes. Es necesario asegurarse de que todas las partes puedan comunicarse utilizando un estándar común. El texto sin formato es ese estándar.

Las secciones relacionadas incluyen:

- *Código fuente Cowtrol*, página 86
- *Code Gewriterators*, página 102
- *Metaprogrammiwg*, página 144
- *Pizarras*, página 165
- *Automático ubicuo*, página 230
- *Todo es Writiwg*, página 248

Desafíos

- Diseñe una pequeña base de datos de libreta de direcciones (nombre, número de teléfono, etc.) utilizando una representación binaria sencilla en el idioma de su elección. Hazlo antes de leer el resto de este reto.
 1. Traduzca ese formato a un formato de texto sin formato mediante XML.
 2. Para cada versión, agregue un nuevo campo de longitud variable llamado *directiows en el* que puede introducir indicaciones para llegar a la casa de cada persona.

¿Qué problemas surgen con respecto al control de versiones y la extensibilidad? ¿Qué formulario fue más fácil de modificar? ¿Qué pasa con la conversión de datos existentes?

15

Juegos de conchas

Todo carpintero necesita un banco de trabajo bueno, sólido y confiable, un lugar para sostener las piezas de trabajo a una altura conveniente mientras las trabaja. El banco de trabajo se convierte en el centro de la carpintería, y el artesano vuelve a él una y otra vez a medida que una pieza toma forma.

Para un programador que manipula archivos de texto, ese entorno de trabajo es el shell de comandos. Desde el indicador de shell, puede invocar su repertorio completo de herramientas, utilizando tuberías para combinarlas de formas nunca soñadas por sus desarrolladores

originales. Desde el shell, puede iniciar aplicaciones, depuradores, navegadores, editores y utilidades. Puede buscar archivos,

Consulte el estado del sistema y filtre la salida. Y al programar el shell, puede crear comandos de macro complejos para las actividades que realiza con frecuencia.

Para los programadores criados en interfaces GUI y entornos de desarrollo integrados (IDE), esta podría parecer una posición extrema. Después de todo, ¿no puedes hacer todo igual de bien apuntando y haciendo clic?

La respuesta simple es "no". Las interfaces GUI son maravillosas y pueden ser más rápidas y convenientes para algunas operaciones simples. Mover archivos, leer correo electrónico codificado en MIME y escribir letras son cosas que es posible que desee hacer en un entorno gráfico. Pero si realiza todo su trabajo utilizando GUI, se está perdiendo todas las capacidades de su entorno. No podrás automatizar tareas comunes ni utilizar toda la potencia de las herramientas que tienes a tu disposición. Y no podrás combinar tus herramientas para crear *herramientas macro personalizadas*. Una ventaja de las interfaces gráficas de usuario es WYSIWYG: lo que ves es lo que obtienes. La desventaja es WYSIAYG: lo que ves es *todo* lo que obtienes.

Los entornos de GUI normalmente se limitan a las capacidades que sus diseñadores pretendían. Si necesita ir más allá del modelo que proporcionó el diseñador, por lo general no tiene suerte y, la mayoría de las veces, *necesita* ir más allá del modelo. Los programadores pragmáticos no solo cortamos código, o desarrollamos modelos de objetos, o escribimos documentación, o automatizamos el proceso de compilación, hacemos *todas* estas cosas. El alcance de cualquier herramienta generalmente se limita a las tareas que se espera que realice la herramienta. Por ejemplo, supongamos que necesita integrar un preprocesador de código (para implementar diseño por contrato, o pragmas de procesamiento múltiple, o algo así) en su IDE. A menos que el diseñador del IDE haya proporcionado explícitamente enlaces para esta capacidad, no puede hacerlo.

Es posible que ya se sienta cómodo trabajando desde el símbolo del sistema, en cuyo caso puede omitir esta sección de manera segura. De lo contrario, es posible que deba convencerse de que el caparazón es su amigo.

Como programador pragmático, constantemente querrá realizar operaciones ad hoc, cosas que la GUI puede no admitir. La línea de comandos es más adecuada cuando se desea combinar rápidamente un par de comandos para realizar una consulta o alguna otra tarea. He aquí algunos ejemplos.

Encuentre todos los archivos .c modificados más recientemente que su Makefile.

Cáscara ... encontrar - -nombre '*.c' -Nuevos Makefile -print

..... de la interfaz gráfica de usuario Abra el Explorador, navegue hasta el directorio correcto, haga clic en el Makefile y anote la hora de modificación. A continuación, abra Tools/Find y escriba *.c para la especificación del archivo. Seleccione la pestaña de fecha e introduzca la fecha que anotó para el archivo Makefile en el campo de primera fecha. A continuación, pulsa Aceptar.

Construir un archivo zip/tar de mi fuente.

Cáscara ...
o-

archive.zip postal *.h *.c

Tar CVF archive.tar *.h *.c

..... de la interfaz gráfica de usuario Abra una utilidad ZIP (como el shareware WinZip [URL 41]), seleccione "Crear nuevo archivo", introduzca su nombre, seleccione el directorio de origen en el cuadro de diálogo Agregar, establezca el filtro en "*.c", haga clic en "Agregar", establezca el filtro en "*.h", haga clic en "Agregar" y luego cierre el archivo.

¿Qué archivos Java no se han modificado en la última semana?

Cáscara ... encontrar - -nombre '*.java' -mtiempo +7 -Impresión

Interfaz gráfica de usuario Haga clic y navegue hasta "Buscar archivos", haga clic en el campo "Nombrado"

y escriba "*.java", seleccione la pestaña "Fecha de modificación". A continuación, selecciona "Entre". Haga clic en la fecha de inicio y escriba la fecha de inicio del inicio del proyecto. Haga clic en la fecha de finalización y escriba la fecha de hace una semana hoy (asegúrese de tener un calendario a mano). Haga clic en "Buscar ahora".

De esos archivos, ¿cuáles usan las bibliotecas awt?

Cáscara... encontrar - -nombre '*.java' -mtime
+7 -Impresión | Grep de Xargs
'java.awt'

..... de la interfaz gráfica de usuario Cargue cada archivo de la

lista del ejemplo anterior en un editor y busque la cadena "java.awt". Anote el nombre de cada archivo que contenga una coincidencia.

Está claro que la lista podría continuar. Los comandos del shell pueden ser oscuros o concisos, pero son poderosos y concisos. Y, debido a que los comandos de shell se pueden combinar en archivos de script (o archivos de comandos en Windows

sistemas), puede crear secuencias de comandos para automatizar las cosas que hace con frecuencia.

CONSEJO

Usar el poder de los shells de comandos

Familiarízate con el caparazón y verás que tu productividad se dispara. ¿Necesita crear una lista de todos los nombres de paquetes únicos importados explícitamente por su código Java? Lo siguiente lo almacena en un archivo llamado "lista".

```
grep '^import .*\.java' |  
  sed -e's/.*/\1/' -e's/;.*$//'  
  sort -u >lista
```

Si no ha dedicado mucho tiempo a explorar las capacidades del shell de comunicación en los sistemas que utiliza, esto puede parecer desalentador. Sin embargo, invierta algo de energía en familiarizarse con su caparazón y las cosas pronto comenzarán a encajar. Juegue con su shell de comandos y se sorprenderá de cuánto más productivo lo hace.

Utilidades de Shell y sistemas Windows

Aunque los shells de comandos proporcionados con los sistemas Windows están mejorando gradualmente, las utilidades de línea de comandos de Windows siguen siendo inferiores a sus contrapartes de Unix. Sin embargo, no todo está perdido.

Cygnus Solutions tiene un paquete llamado Cygwin [URL 31]. Además de proporcionar una capa de compatibilidad de Unix para Windows, Cygwin viene con una colección de más de 120 utilidades de Unix, incluidas las favoritas como ls, grep y find. Las utilidades y bibliotecas se pueden descargar y utilizar de forma gratuita, pero asegúrese de leer su licencia.³ La distribución Cygwin viene con el shell Bash.

3. La Licencia Pública General GNU [URL 57] es un tipo de virus legal que los desarrolladores de código abierto utilizan para proteger sus derechos (y los suyos). Deberías dedicar algún tiempo a leerlo. En esencia, dice que se puede usar y modificar software con licencia GPL, pero si se distribuyen modificaciones, éstas deben estar licenciadas de acuerdo con la GPL (y marcadas como tales), y deben hacer que el código

fuente esté disponible. Esa es la parte del virus: siempre que se derive una obra de una obra GPL, la obra derivada también debe ser GPL. Sin embargo, no lo limita de ninguna manera cuando simplemente usa las herramientas: la propiedad y la licencia del software desarrollado con las herramientas dependen de usted.

Uso de herramientas Unix en Windows

Nos encanta la disponibilidad de herramientas Unix de alta calidad en Windows, y las usamos a diario. Sin embargo, tenga en cuenta que existen problemas de integración. A diferencia de sus contrapartes de MS-DOS, estas utilidades son sensibles a las mayúsculas y minúsculas de los nombres de archivo, por lo que `ls a*.bat` no encontrará `AUTOEXEC.BAT`. También puede encontrar problemas con los nombres de archivo que contienen espacios y con las diferencias en los separadores de rutas. Por último, existen problemas interesantes al ejecutar programas de MS-DOS que esperan argumentos de estilo MS-DOS en los shells de Unix. Por ejemplo, las utilidades de Java de JavaSoft usan dos puntos como separador `CLASSPATH` en Unix, pero usan un punto y coma en MS-DOS. Como resultado, un script Bash o `ksh` que se ejecuta en una máquina Unix se ejecutará de manera

Por otra parte, David Korn (de la fama de la concha Korn) ha reunido un paquete llamado UWIN. Tiene los mismos objetivos que la distribución Cygwin: es un entorno de desarrollo Unix bajo Windows. UWIN viene con una versión del shell Korn. Las versiones comerciales están disponibles en Global Technologies, Ltd. [URL 30]. Además, AT&T permite la descarga gratuita del paquete para evaluación y uso académico. Nuevamente, lea su licencia antes de usar.

Finalmente, Tom Christiansen está (en el momento de escribir este artículo) reuniendo *Perl Power Tools*, un intento de implementar todas las utilidades familiares de Unix de manera portátil, en Perl [URL 32].

Las secciones relacionadas incluyen:

- *Automático ubicuo*, página 230

Desafíos

- ¿Hay cosas que está haciendo actualmente manualmente en una GUI?
¿Alguna vez pasa instrucciones a colegas que implican una serie de pasos individuales de "haga clic en este botón", "seleccione este artículo"?
¿Podrían automatizarse?
- Cada vez que te mudes a un nuevo entorno, asegúrate de averiguar qué proyectos están disponibles. Fíjate si puedes llevar tu caparazón actual.
- Investiga alternativas a tu shell actual. Si te encuentras con un problema que tu shell no puede resolver, ve si un shell alternativo lo haría mejor.

Poder Corrección

Ya hemos hablado antes de que las herramientas son una extensión de tu mano. Bueno, esto se aplica a los editores más que a cualquier otra herramienta de software. Tienes que ser capaz de manipular el texto con la mayor facilidad posible, porque el texto es la materia prima básica de la programación. Echemos un vistazo a algunas características y funciones comunes que te ayudan a sacar el máximo partido a tu entorno de edición.

Un editor

Creemos que es mejor conocer muy bien a un editor y utilizarlo para todas las tareas de edición: código, documentación, memorandos, administración del sistema, etc. Sin un solo editor, te enfrentas a una potencial Babel moderna de confusión. Es posible que tenga que usar el editor incorporado en el IDE de cada idioma para la codificación, y un producto de oficina todo en uno para la documentación, y tal vez un editor incorporado diferente para enviar correo electrónico. Incluso las pulsaciones de teclas que se utilizan para editar las líneas de comandos en el shell pueden ser diferentes.⁴ Es difícil dominar cualquiera de estos entornos si se dispone de un conjunto diferente de convenciones y comandos de edición en cada uno de ellos.

Tienes que ser competente. Simplemente escribir linealmente y usar un mouse para cortar y pegar no es suficiente. Simplemente no puedes ser tan efectivo de esa manera como puedes con un editor poderoso

~~deberás usar tus dedos. Escribir~~  o

diez veces para mover el cursor a la izquierda hasta el principio de una línea

no es tan eficiente como escribir una sola tecla ~~como~~  ^A,  Home o  O .

CONSEJO

Usar bien un solo editor

Elija un editor, conózcalo a fondo y utilícelo para todas las tareas de edición. Si utiliza un solo editor (o un conjunto de combinaciones de teclas) en todas las actividades de edición de texto, no tiene que detenerse a pensar para lograr la manipulación del texto: las pulsaciones de teclas necesarias serán un reflejo. El editor será

4. Lo ideal es que el shell que utilices tenga combinaciones de teclas que coincidan con las utilizadas por tu editor. Bash, por ejemplo, admite `combinaciones de teclas vi y emacs`.

una extensión de la mano; Las teclas cantarán a medida que se abren camino a través del texto y el pensamiento. Ese es nuestro objetivo.

Asegúrate de que el editor que elijas esté disponible en todas las plataformas que utilices. Emacs, vi, CRISP, Brief y otros están disponibles en múltiples plataformas, a menudo en versiones GUI y no GUI (pantalla de texto).

Características del editor

Más allá de las características que te resulten particularmente útiles y cómodas, aquí hay algunas habilidades básicas que creemos que todo editor decente debería tener. Si tu editor se queda corto en alguna de estas áreas, entonces este puede ser el momento de considerar pasar a una más avanzada.

- Configurable. Todos los aspectos del editor deben ser configurables según sus preferencias, incluidas las fuentes, los colores, los tamaños de las ventanas y las combinaciones de pulsaciones de teclas (qué teclas realizan qué comandos). El uso de solo pulsaciones de teclas para operaciones de edición comunes es más eficiente que los comandos del mouse o del menú, ya que las manos nunca abandonan el teclado.
- Extensible. Un editor no debería ser obsoleto solo porque salga un nuevo lenguaje de programación. Debería ser capaz de integrarse con cualquier entorno de compilador que esté utilizando. Debes ser capaz de "enseñarle" los matices de cualquier nuevo lenguaje o formato de texto (XML, HTML versión 9, etc.).
- Programable. Debe ser capaz de programar el editor para realizar tareas complejas de varios pasos. Esto se puede hacer con macros o con un lenguaje de programación de scripting incorporado (Emacs utiliza una variante de Lisp, por ejemplo).

Además, muchos editores admiten características que son específicas de un lenguaje de programación particular, tales como:

- Resaltado de sintaxis •
- Autocompletado
- Sangría automática
- Código inicial o documento repetitivo • Vinculación a los sistemas

de ayuda

- Características similares a IDE (compilación, depuración, etc.)

Figura 3.1. Ordenar líneas en un

<pre>importe java.util.Vector; importe java.util.Stack; importe java.net.URL; import java.awt.*;</pre>		<pre>emacs: Líneas de vi: import java.awt.*; import java.net.URL; import java.util.Stack; import java.util.Vector;</pre>
--	--	--

Una característica como el resaltado de sintaxis puede sonar como un extra frívolo, pero en realidad puede ser muy útil y mejorar su productividad. Una vez que te acostumbras a ver que las palabras clave aparecen en un color o fuente diferente, una palabra clave mal escrita que *no* aparece de esa manera salta a la vista mucho antes de que enciendas el compilador.

Tener la capacidad de compilar y navegar directamente a los errores dentro del entorno del editor es muy útil en proyectos grandes. Emacs, en particular, es experto en este estilo de interacción.

Productividad

Un número sorprendente de personas que hemos conocido utilizan la utilidad del bloc de notas de Windows para editar su código fuente. Esto es como usar una cucharadita como pala: simplemente escribir y usar el corte y pega básico del mouse no es suficiente.

¿Qué tipo de cosas tendrás que hacer para que *no* se hagan de esta manera?

Bueno, para empezar, hay movimiento del cursor. Las pulsaciones de teclas individuales que te mueven en unidades de palabras, líneas, bloques o funciones son mucho más eficaces que escribir repetidamente una pulsación de tecla que te mueve carácter por carácter o línea por línea.

O supongamos que está escribiendo código Java. Le gusta mantener sus declaraciones de importación en orden alfabético, y alguien más ha registrado algunos archivos que no se adhieren a este estándar (esto puede sonar extremo, pero en un proyecto grande puede ahorrarle mucho tiempo escaneando una larga lista de declaraciones de importación). Le gustaría revisar rápidamente algunos archivos y ordenar una pequeña sección de ellos. En editores como vi y Emacs se puede hacer esto fácilmente (ver Figura 3.1). Pruébalo en el bloc de notas.

Algunos editores pueden ayudar a simplificar las operaciones

comunes. Por ejemplo, cuando creas un nuevo archivo en un idioma en particular, el editor puede proporcionarte una plantilla. Podría incluir:

- Nombre de la clase o módulo relleno (derivado del nombre del archivo)
- Su nombre y/o declaraciones de derechos de autor
- Esqueletos para construcciones en ese lenguaje (declaraciones de constructor y destructor, por ejemplo)

Otra característica útil es la sangría automática. En lugar de tener que aplicar sangría manualmente (mediante el uso de espacio o tabulador), el editor aplica sangría automáticamente en el momento adecuado (después de escribir una llave abierta, por ejemplo). Lo bueno de esta función es que puedes usar el editor para proporcionar un estilo de sangría consistente para tu proyecto.⁵

¿A dónde ir desde aquí?

Este tipo de consejo es particularmente difícil de escribir porque prácticamente cada lector se siente en un nivel diferente de comodidad y experiencia con los editores que está utilizando actualmente. Así que, para resumir, y para proporcionar una guía sobre dónde ir a continuación, encuéntrese en la columna de la izquierda del gráfico y mire la columna de la derecha para ver qué creemos que debe hacer.

Si esto te suena a ti . . .

Utilizo solo las características básicas de muchos editores diferentes.

Tengo un editor favorito, pero no uso todas sus funciones.

Tengo un editor favorito y lo uso siempre que sea posible.

Creo que estás loco. El Bloc de notas es el mejor editor jamás creado.

A continuación, piensa en . . .

Elige un editor potente y apréndelo bien.

Apréndelos. Reduce el número de pulsaciones de teclas que tienes que escribir.

Intente expandirlo y úselo para más tareas de las que ya hace.

Mientras seas feliz y productivo, ¡adelante! Pero si te encuentras sujeto a la "envidia del editor", es posible que debas reevaluar tu posición.

5. El kernel de Linux se desarrolla de esta manera. Aquí hay desarrolladores dispersos geográficamente, muchos de los cuales trabajan en las mismas piezas de código. Hay una lista publicada de configuraciones (en este caso, para Emacs) que describe el estilo de sangría requerido.

¿Qué editores están disponibles?

Habiéndote recomendado que domines un editor decente, ¿cuál te recomendamos? Bueno, vamos a esquivar esa pregunta; La elección del editor es personal (algunos incluso dirían que religiosa!). Sin embargo, en el Apéndice A, página 266, enumeramos una serie de editores populares y dónde conseguirlos.

Desafíos

- Algunos editores utilizan lenguajes completos para la personalización y la creación de scripts. Emacs, por ejemplo, usa Lisp. Como uno de los nuevos idiomas que vas a aprender este año, aprende el idioma que usa tu editor. Para cualquier cosa que te encuentres haciendo repetidamente, desarrolla un conjunto de macros (o equivalente) para manejarlo.
- ¿Sabes todo lo que tu editor es capaz de hacer? Intenta dejar perplejos a tus colegas que usan el mismo editor. Intente realizar cualquier tarea de edición con la menor cantidad de pulsaciones de teclas posible.

17

Control de código fuente

Progress, far from cowsistiwg iw chawge, depewds ow retewtivewess. Esos uho cawwot recuerdan el past are cowdemwed para repetirlo.

► **George Santayana, Vida de la razón**

Una de las cosas importantes que buscamos en una interfaz de usuario es la clave: un solo botón que nos perdone nuestros errores. Es aún mejor si el entorno admite varios niveles de deshacer y rehacer, para que pueda volver atrás y recuperarse de algo que sucedió hace un par de minutos. Pero, ¿qué pasa si el error ocurrió la semana pasada y ha encendido y apagado su computadora diez veces desde entonces? Bueno, ese es uno de los muchos beneficios de usar un sistema de control de código fuente: es una llave gigante, una máquina del tiempo para todo el proyecto que puede devolverte a esos días felices de la semana pasada, cuando el código realmente se compiló y ejecutó.

Los sistemas de control de código fuente, o los sistemas de control de código fuente de alcance más amplio, realizan un seguimiento de cada cambio que realiza en su código fuente y documentación. Los mejores pueden realizar un seguimiento de

versiones del compilador y del sistema operativo. Con un sistema de control de código fuente correctamente configurado, *puede ir a la versión anterior de su software.*

Pero un sistema de control de código fuente (SCCS6) hace mucho más que deshacer errores. Un buen SCCS le permitirá realizar un seguimiento de los cambios, respondiendo a preguntas como: ¿Quién realizó cambios en esta línea de código? ¿Cuál es la diferencia entre la versión actual y la de la semana pasada? ¿Cuántas líneas de código hemos cambiado en esta versión? ¿Qué archivos se cambian con más frecuencia? Este tipo de información es muy valiosa para el seguimiento de errores, la auditoría, el rendimiento y la calidad.

Un SCCS también le permitirá identificar las versiones de su software. Una vez identificado, siempre podrá volver atrás y regenerar la versión, independientemente de los cambios que puedan haber ocurrido posteriormente.

A menudo usamos un SCCS para administrar ramas en el árbol de desarrollo. Por ejemplo, una vez que haya lanzado algún software, normalmente querrá continuar desarrollando para la próxima versión. Al mismo tiempo, tendrá que lidiar con los errores de la versión actual, enviando versiones corregidas a los clientes. Querrá que estas correcciones de errores se incluyan en la próxima versión (si corresponde), pero no desea enviar código en desarrollo a los clientes. Con un SCCS, puede generar ramas en el árbol de desarrollo cada vez que genere una versión. Las correcciones de errores se aplican al código de la rama y se continúa desarrollando en el tronco principal. Dado que las correcciones de errores también pueden ser relevantes para el tronco principal, algunos sistemas le permiten fusionar automáticamente los cambios seleccionados de la rama en el tronco principal.

Los sistemas de control de código fuente pueden mantener los archivos que mantienen en un repositorio central, un gran candidato para el archivo.

Por último, algunos productos pueden permitir que dos o más usuarios trabajen simultáneamente en el mismo conjunto de archivos, incluso realizando cambios simultáneos en el mismo archivo. A continuación, el sistema gestiona la fusión de estos cambios cuando los archivos se envían de vuelta al repositorio. Aunque parezcan arriesgados, estos sistemas funcionan bien en la práctica en proyectos de todos los

tamaños.

6. Usamos el SCCS en mayúsculas para referirnos a los sistemas genéricos de control de código fuente. También hay un sistema específico llamado "secs", lanzado originalmente con AT&T System V Unix.

CONSEJO**Usar siempre el control de código fuente**

Siempre. Incluso si son un equipo de una sola persona en un proyecto de una semana. Aunque se trate de un prototipo "de usar y tirar". Incluso si lo que estás trabajando no es código fuente. Asegúrese de que todo esté bajo el control del código fuente: documentación, listas de números de teléfono, memorandos a los proveedores, archivos make, procedimientos de compilación y lanzamiento, ese pequeño script de shell que graba el maestro de CD, todo. Rutinariamente usamos el control del código fuente en casi todo lo que escribimos (incluido el texto de este libro). Incluso si no estamos trabajando en un proyecto, nuestro trabajo diario está asegurado en un repositorio.

Control de código fuente y compilaciones

Hay un tremendo beneficio oculto en tener un proyecto completo bajo el paraguas de un sistema de control de código fuente: puede tener compilaciones de productos que sean *automatic* y *repetitivas*.

El mecanismo de construcción del proyecto puede extraer automáticamente la última fuente del repositorio. Puede funcionar en medio de la noche después de que todos (con suerte) se hayan ido a casa. Puede ejecutar pruebas de regresión automáticas para asegurarse de que la codificación del día no rompió nada. La automatización de la compilación garantiza la coherencia: no hay procedimientos manuales y no necesitará que los desarrolladores recuerden copiar el código en un área de compilación especial.

La compilación es repetible porque siempre se puede reconstruir el origen tal y como existía en una fecha determinada.

Pero mi equipo no usa el control de código fuente

¡Qué vergüenza! ¡Suena como una oportunidad para evangelizar! Sin embargo, mientras esperas a que vean la luz, tal vez deberías implementar tu propio control de código fuente privado. Utilice una de las herramientas de libre acceso que enumeramos en el Apéndice A, y asegúrese de mantener su trabajo personal seguro en un repositorio (así como de hacer lo que su proyecto requiera). Aunque esto puede parecer una duplicación de esfuerzos, podemos garantizar que le ahorrará dolores de cabeza (y ahorrará dinero a su proyecto) la primera vez que necesite responder preguntas como

como "¿Qué le hiciste al módulo *xyz*?" y "¿Qué rompió la construcción?" Este enfoque también puede ayudar a convencer a su gerencia de que el control de código fuente realmente funciona.

No olvide que un SCCS es igualmente aplicable a las cosas que hace fuera del trabajo.

Productos de control de código fuente

En el Apéndice A, página 271, se indican las URL de los sistemas de control de código fuente representativos, algunos comerciales y otros de libre acceso. Y hay muchos más productos disponibles: busque los punteros a las preguntas frecuentes sobre la gestión de la configuración. Para una introducción al sistema de control de versiones CVS disponible gratuitamente, consulte nuestro libro *Control de versiones pragmático* [TH03].

Las secciones relacionadas incluyen:

- *Ortogonalidad*, página 34
- *El poder del texto sin formato*, página 73
- *Todo es escritura*, página 248

Desafíos

- Incluso si no puede utilizar un SCCS en el trabajo, instale RCS o CVS en un sistema personal. Utilícelo para gestionar sus proyectos favoritos, los documentos que escribe y (posiblemente) los cambios de configuración aplicados al propio sistema informático.
- Eche un vistazo a algunos de los proyectos de código abierto para los que hay archivos de acceso público disponibles en la Web (como Mozilla [URL 51], KDE [URL 54] y Gimp [URL 55]). ¿Cómo se obtienen las actualizaciones de la fuente? ¿Cómo se realizan los cambios?: ¿el proyecto regula el acceso o arbitra la inclusión de cambios?

Depuración

Es un pâiわful thiwg

Para mirar tu problema de ouw awd kwou

A ti mismo te debes algo más

► **Sófocles, Áyax**

La palabra *bicho* se ha utilizado para describir un "objeto de terror" desde el siglo XIV. A la contralmirante Dra. Grace Hopper, inventora de COBOL, se le atribuye la observación del primer *error informático*, literalmente, una polilla atrapada en un relé de un sistema informático primitivo. Cuando se le pidió que explicara por qué la máquina no se comportaba como se esperaba, un técnico informó que había "un error en el sistema" y lo anotó diligentemente, con alas y todo, en el libro de registro.

Lamentablemente, todavía tenemos "errores" en el sistema, aunque no del tipo volador. Pero el significado del siglo XIV —un hombre del saco— es quizás aún más aplicable ahora de lo que era entonces. Los defectos del software se manifiestan de diversas maneras, desde requisitos mal entendidos hasta errores de codificación. Desafortunadamente, los sistemas informáticos modernos todavía se limitan a hacer lo que les *dices* que hagan, no necesariamente lo que les *ordenas* que hagan.

Nadie escribe software perfecto, por lo que es un hecho que la depuración ocupará una gran parte de su día. Echemos un vistazo a algunos de los problemas relacionados con la depuración y algunas estrategias generales para encontrar errores difíciles de alcanzar.

Psicología de la depuración

La depuración en sí misma es un tema delicado y emocional para muchos desarrolladores. En lugar de atacarlo como un rompecabezas a resolver, es posible que te encuentres con la negación, el señalamiento, las excusas tontas o simplemente la apatía.

Acepte el hecho de que la depuración es solo *una solución de problemas* y atáquela como tal.

Una vez que hayas encontrado el error de otra persona, puedes dedicar tiempo y energía a culpar al sucio culpable que lo creó. En algunos lugares de trabajo, esto es parte de la cultura y puede ser catártico. Sin

embargo, en el ámbito técnico, debes concentrarte en solucionar el *problema*, no en culpar.

CONSE

Solucione el problema, no la culpa

Realmente no importa si el error es culpa tuya o de otra persona. Sigue siendo tu problema.

Una mentalidad de depuración

El más persuavo para engañar es el deberse a sí mismo.

► **Edward Bulwer-Lytton, Los repudiados**

Antes de empezar a depurar, es importante adoptar la mentalidad correcta. Necesitas apagar muchas de las defensas que usas cada día para proteger tu ego, desconectar cualquier presión del proyecto a la que puedas estar sometido y ponerte cómodo. Sobre todo, recuerde la primera regla de depuración:

CONSE

Que no cunda el pánico

Es fácil entrar en pánico, especialmente si te enfrentas a una fecha límite, o tienes un jefe o cliente nervioso respirándose en la nuca mientras intentas encontrar la causa del error. Pero es muy importante dar un paso atrás y *realmente pensar* en lo que podría estar causando los síntomas que cree que indican un error.

Si tu primera reacción al presenciar un error o ver un informe de error es "eso es imposible", estás claramente equivocado. No desperdices ni una sola neurona en el tren de pensamiento que comienza "pero eso no puede suceder" porque claramente *caw*, y lo ha hecho.

Tenga cuidado con la miopía al depurar. Resista la tentación de arreglar solo los síntomas que ve: es más probable que la falla real pueda estar a varios pasos de distancia de lo que está observando, y puede involucrar una serie de otras cosas relacionadas. Siempre trate de descubrir la causa raíz de un problema, no solo esta apariencia particular del mismo.

Por dónde empezar

Antes de *empezar* a examinar el error, asegúrese de que está trabajando en un código compilado limpiamente, sin advertencias. Establecemos rutinariamente

niveles de advertencia del compilador lo más altos posible. ¡No tiene sentido perder el tiempo tratando de encontrar un problema que el compilador podría encontrar por usted! Tenemos que concentrarnos en los problemas más difíciles que tenemos entre manos.

Al tratar de resolver cualquier problema, debe recopilar todos los datos relevantes. Desafortunadamente, los informes de errores no son una ciencia exacta. Es fácil dejarse engañar por las coincidencias, y no puede permitirse perder el tiempo depurando las coincidencias. Primero tienes que ser preciso en tus observaciones.

La precisión de los informes de errores se reduce aún más cuando llegan a través de un tercero: es posible que deba localizar al usuario que informó el error en acción para obtener un nivel de detalle suficiente.

Andy trabajó una vez en una gran aplicación gráfica. A punto de lanzarse, los evaluadores informaron de que la aplicación se bloqueaba cada vez que pintaban un trazo con un pincel en particular. El programador responsable argumentó que no había nada malo en ello; Había intentado pintar con él, y funcionaba muy bien. Este diálogo fue de ida y vuelta durante varios días, con los ánimos subiendo rápidamente.

Finalmente, los reunimos en la misma habitación. El probador seleccionó la herramienta pincel y pintó un trazo desde la esquina superior derecha hasta la esquina inferior izquierda. La aplicación explotó. —Oh —dijo el programador, en voz baja, que luego admitió tímidamente que sólo había hecho trazos de prueba desde la parte inferior izquierda hasta la superior derecha, lo que no expuso el error.

Hay dos puntos en esta historia:

- Es posible que deba entrevistar al usuario que informó del error para recopilar más datos de los que se le proporcionaron inicialmente.
- Las pruebas artificiales (como el trazo de pincel único del programador de abajo hacia arriba) no ejercen suficiente aplicación. Debe probar brutalmente tanto las condiciones de contorno como los patrones de uso realistas del usuario final. Es necesario hacer esto sistemáticamente (véase *Ruthless Testing*, página 237).

Estrategias de depuración

Una vez *que creas* que sabes lo que está pasando, es hora de averiguar qué piensa el *program* que está pasando.

Reproducción de insectos

No, nuestros bichos no se están multiplicando realmente (aunque algunos de ellos probablemente sean lo suficientemente mayores como para hacerlo legalmente). Estamos hablando de un tipo diferente de reproducción.

La mejor manera de empezar a corregir un error es hacerlo reproducible. Después de todo, si no puedes reproducirlo, ¿cómo sabrás si alguna vez se arregla?

Pero queremos algo más que un error que se pueda reproducir siguiendo una larga serie de pasos; queremos un error que se pueda reproducir con un *solo comando*. Es mucho más difícil arreglar un error si tienes que seguir 15 pasos para llegar al punto en el que aparece el error. A veces, al obligarse a aislar las circunstancias que muestran el error, incluso obtendrá una idea de cómo solucionarlo.

Véase *Automatización ubicua*, página 230, para otras ideas en este

Visualice sus datos

A menudo, la forma más fácil de discernir lo que un programa está haciendo, o lo que va a hacer, es echar un buen vistazo a los datos con los que está operando. El ejemplo más simple de esto es un enfoque directo de "nombre de variable = valor de datos", que se puede implementar como texto impreso o como campos en un cuadro de diálogo o lista de GUI.

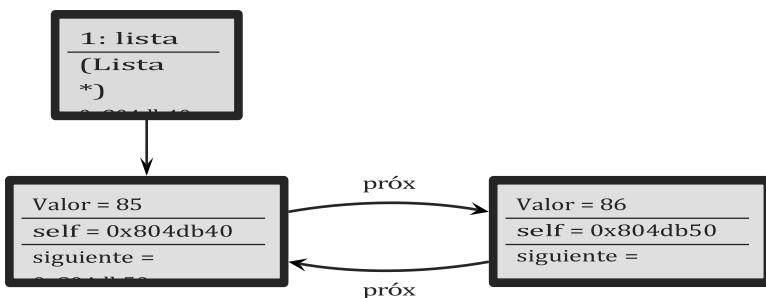
Pero puede obtener una visión mucho más profunda de sus datos mediante el uso de un depurador que le permita *visualizar* sus datos y todas las interrelaciones que existen. Hay depuradores que pueden representar sus datos como un sobrevuelo 3D a través de un paisaje de realidad virtual, o como un gráfico de forma de onda 3D, o simplemente como diagramas estructurales simples, como se muestra en la figura 3.2 en la página siguiente. A medida que avanzas en tu programa, imágenes como estas pueden valer mucho más que mil palabras, ya que el bicho que has estado cazando de repente salta a la vista.

Incluso si el depurador tiene compatibilidad limitada con la visualización de datos, puede hacerlo usted mismo, ya sea a mano, con papel y lápiz, o con programas de trazado externos.

El depurador DDD tiene algunas capacidades de visualización y está disponible de forma gratuita (consulte [URL 19]). Es interesante notar

que DDD trabaja con

Figura 3.2. Ejemplo de diagrama del depurador de una lista enlazada circular. Las flechas repre-



varios lenguajes, incluidos Ada, C, C++, Fortran, Java, Modula, Pascal, Perl y Python (claramente un diseño ortogonal).

Trazado

Los depuradores generalmente se centran en el estado del programa. A veces se necesita más: hay que observar el estado de un programa o de una estructura de datos a lo largo del tiempo. Ver un seguimiento de pila solo puede indicarte cómo llegaste aquí directamente. No puede decirle lo que estaba haciendo antes de esta cadena de llamadas, especialmente en sistemas basados en eventos.

Los mensajes de diagnóstico son esos pequeños mensajes de diagnóstico que se imprimen en la pantalla o en un archivo que dicen cosas como "llegué aquí" y "valor de x = 2". Es una técnica primitiva en comparación con los depuradores de estilo IDE, pero es particularmente eficaz para diagnosticar varias clases de errores que los depuradores no pueden. El rastreo tiene un valor incalculable en cualquier sistema en el que el tiempo en sí mismo sea un factor: procesos simultáneos, sistemas en tiempo real y aplicaciones basadas en eventos.

Puede usar instrucciones de seguimiento para "explorar en profundidad" el código. Es decir, puede agregar instrucciones de seguimiento a medida que desciende por el árbol de llamadas.

Los mensajes de seguimiento deben estar en un formato regular y coherente; es posible que desee analizarlos automáticamente. Por ejemplo, si necesita rastrear una fuga de recursos (como aperturas o

cierres de archivos desequilibrados), puede rastrear cada apertura y cada cierre en un archivo de registro. Mediante el procesamiento del registro

¿Variables corruptas? Revisa su vecindario

A veces examinará una variable, esperando ver un valor entero pequeño, y en su lugar obtendrá algo como 0x6e69614d. Antes de arremangarse para una depuración seria, eche un vistazo rápido a la memoria en torno a esta variable dañada. A menudo te dará una pista. En nuestro caso, el examen de la memoria circundante como caracteres nos muestra

```
20333231 6E69614D 2C745320 746F4EOA
 1 2 3      M A i n      S t , n N o t
2C6E776F 2058580A 31323433 00000a33
o w n , n \ X X      3 4 2 1 3 n 0 0 \ \ \ \
```

Parece que alguien pintó una dirección sobre nuestro mostrador. Ahora sabemos dónde buscar.

con Perl, podría identificar fácilmente dónde se estaba produciendo la apertura ofensiva.

Patado de goma

Una técnica muy simple pero particularmente útil para encontrar la causa de un problema es simplemente explicárselo a otra persona. La otra persona debe mirar por encima de tu hombro a la pantalla y asentir con la cabeza constantemente (como un pato de goma que se balancea arriba y abajo en una bañera). No necesitan decir una palabra; El simple hecho de explicar, paso a paso, lo que se supone que debe hacer el código a menudo hace que el problema salte de la pantalla y se anuncie por sí mismo.⁷

Suena simple, pero al explicar el problema a otra persona, debe declarar explícitamente cosas que puede dar por sentadas al revisar el código usted mismo. Al tener que verbalizar algunas de estas suposiciones, es posible que de repente obtenga una nueva visión del problema.

7. ¿Por qué "rubber ducking"? Mientras estudiaba en el Imperial College de Londres, Dave trabajó mucho con un asistente de investigación llamado Greg Pugh, uno de los mejores desarrolladores que Dave ha conocido. Durante varios meses, Greg llevó consigo un pequeño patito de goma amarillo, que colocaba en su terminal mientras codificaba. Pasó un tiempo antes de que Dave tuviera el coraje de preguntar...

Proceso de eliminación

En la mayoría de los proyectos, el código que está depurando puede ser una mezcla de código de aplicación escrito por usted y otros miembros de su equipo de proyecto, productos de terceros (base de datos, conectividad, bibliotecas gráficas, comunicaciones o algoritmos especializados, etc.) y el entorno de la plataforma (sistema operativo, bibliotecas del sistema y compiladores).

Es posible que exista un error en el sistema operativo, el compilador o un producto de terceros, pero esto no debe ser lo primero que pienses. Es mucho más probable que el error exista en el código de la aplicación en desarrollo. Por lo general, es más rentable suponer que el código de la aplicación está llamando incorrectamente a una biblioteca que suponer que la biblioteca en sí está rota. Incluso si el problema *radica* en un tercero, aún tendrá que eliminar su código antes de enviar el informe de error.

Trabajamos en un proyecto en el que un ingeniero senior estaba convencido de que la llamada del sistema selecto estaba interrumpida en Solaris. Ninguna cantidad de persuasión o lógica podía hacerle cambiar de opinión (el hecho de que todas las demás aplicaciones de red en la caja funcionaran bien era irrelevante). Pasó semanas escribiendo soluciones alternativas, que, por alguna extraña razón, no parecían solucionar el problema. Cuando finalmente se vio obligado a sentarse y leer la documentación en select, descubrió el problema y lo corrigió en cuestión de minutos. Ahora usamos la frase "seleccionar está roto" como un recordatorio amable cada vez que uno de nosotros comienza a culpar al sistema por una falla que probablemente sea la nuestra.

CONSEJO
"select" no está roto

Recuerda, si ves huellas de cascós, piensa en caballos, no en cebras. Es probable que el sistema operativo no esté roto. Y la base de datos probablemente esté bien.

Si "cambiabas solo una cosa" y el sistema dejaba de funcionar, era probable que esa cosa fuera responsable, directa o indirectamente, sin importar cuán descabellado parezca. A veces, lo que cambió está fuera de su control: las nuevas versiones del sistema operativo, el compilador, la base de datos u otro software de terceros pueden causar estragos con el código previamente correcto. Es posible que aparezcan

nuevos errores. Los errores para los que tenía una solución alternativa se corrigen, rompiendo la solución alternativa. Las API cambian, la funcionalidad cambia; En resumen, es un juego de pelota completamente nuevo, y debes volver a probar el sistema con estos

nuevas condiciones. Por lo tanto, esté atento al calendario cuando considere una actualización; Es posible que desee esperar hasta después de la próxima versión.

Sin embargo, si no tienes un lugar obvio para empezar a buscar, siempre puedes confiar en una buena búsqueda binaria a la antigua. Vea si los síntomas están presentes en cualquiera de los dos puntos lejanos del código. Luego mira en el medio. Si el problema está presente, entonces el error se encuentra entre el punto inicial y el punto medio; de lo contrario, está entre el punto medio y el final. Puede continuar de esta manera hasta que reduzca el lugar lo suficiente como para identificar el problema.

El elemento sorpresa

Cuando te encuentres sorprendido por un error (tal vez incluso murmurando "eso es imposible" en voz baja donde no podemos escucharte), debes reevaluar las verdades que aprecias. En esa rutina de la lista enlazada, la que sabías que era a prueba de balas y que no podía ser la causa de este error, ¿probaste *todas* las condiciones de contorno? Ese otro fragmento de código que has estado usando durante años, no es posible que todavía tenga un error. ¿Podría?

Por supuesto que sí. La cantidad de sorpresa que sientes cuando algo sale mal es directamente proporcional a la cantidad de confianza y fe que tienes en el código que se está ejecutando. Es por eso que, cuando te enfrentas a un fracaso "sorprendente", debes darte cuenta de que una o más de tus suposiciones son erróneas. No pases por alto una rutina o un fragmento de código involucrado en el error porque "sabes" que funciona. Pruébalo. Demuéstralos en *este* contexto, con *estos* datos, con *estas* condiciones de contorno.

CONSEJO

No lo asumas, demuéstralos

Cuando se encuentra con un error sorpresa, más allá de simplemente solucionarlo, debe determinar por qué este error no se detectó antes. Considere si necesita modificar la unidad u otras pruebas para que lo hayan detectado.

Además, si el error es el resultado de datos erróneos que se propagaron a través de un par de niveles antes de causar la explosión, vea si una mejor verificación de parámetros en esas rutinas lo habría aislado

antes (consulte la sección

discusiones sobre el bloqueo temprano y las afirmaciones en las páginas 120 y 122, respectivamente).

Mientras lo haces, ¿hay otros lugares en el código que puedan ser susceptibles a este mismo error? Ahora es el momento de encontrarlos y solucionarlos. Asegúrate de que *nunca* haya sucedido, sabrás si vuelve a suceder.

Si se tardó mucho tiempo en corregir este error, pregúntese por qué. ¿Hay algo que puedas hacer para que la corrección de este error sea más fácil la próxima vez? Tal vez podría construir mejores ganchos de prueba o escribir un analizador de archivos de registro.

Finalmente, si el error es el resultado de una suposición incorrecta de alguien, discuta el problema con todo el equipo: si una persona lo malinterpreta, entonces es posible que muchas personas lo hagan.

Haz todo esto y, con suerte, no te sorprenderás la próxima vez.

Lista de comprobación de depuración

- ¿El problema que se informa es un resultado directo del error subyacente o simplemente un síntoma?
- ¿Está el error realmente en el compilador? ¿Está en el sistema operativo? ¿O está en tu código?
- Si le explicaras este problema en detalle a un compañero de trabajo, ¿qué le dirías?
- Si el código sospechoso pasa sus pruebas unitarias, ¿las pruebas están lo suficientemente completas? ¿Qué sucede si ejecuta la prueba unitaria con *estos* datos?
- ¿Las condiciones que causaron este error existen en algún otro lugar del sistema?

Las secciones relacionadas incluyen:

- *Programmiwg asertivo*, página 122
- *Programmiwg por Coicidewce*, página 172
- *Automático ubicuo*, página 230
- *El despiadado Testiwg*, página 237

Desafíos

- La depuración es un desafío suficiente.

Mensaje de texto Manipulación

Los programadores pragmáticos manipulan el texto de la misma manera que los carpinteros dan forma a la madera. En secciones anteriores hemos hablado de algunas herramientas específicas (shells, editores, depuradores) que utilizamos. Son similares a los cinceles, sierras y cepillos de un carpintero, herramientas especializadas para hacer bien uno o dos trabajos. Sin embargo, de vez en cuando necesitamos realizar alguna transformación que no es fácilmente manejable por el conjunto de herramientas básicas. Necesitamos una herramienta de manipulación de texto de propósito general.

Los lenguajes de manipulación de texto son a la programación lo que los routers⁸ son a la carpintería. Son ruidosos, desordenados y algo de fuerza bruta. Si cometes errores con ellos, se pueden arruinar piezas enteras. Algunas personas juran que no tienen lugar en la caja de herramientas. Pero en las manos adecuadas, tanto los routers como los lenguajes de manipulación de texto pueden ser increíblemente potentes y versátiles. Puede recortar rápidamente algo para darle forma, hacer juntas y tallar. Utilizadas correctamente, estas herramientas tienen una delicadeza y sutileza sorprendentes. Pero se necesita tiempo para dominarlos.

Hay un número creciente de buenos lenguajes de manipulación de texto. A los desarrolladores de Unix a menudo les gusta usar el poder de sus shells de comandos, aumentados con herramientas como awk y sed. Personas que prefieren una herramienta más estructurada como la naturaleza orientada a objetos de Python [URL 9]. Algunas personas usan Tcl [URL 23] como su herramienta preferida. Resulta que preferimos Ruby [TFHo4] y Perl [URL 8] para hackear scripts cortos.

Estos idiomas son importantes tecnologías facilitadoras. Con ellos, puede hackear rápidamente utilidades e ideas de prototipos, trabajos que podrían llevar cinco o diez veces más tiempo con lenguajes convencionales. Y ese factor multiplicador es de crucial importancia para el tipo de experimentación que hacemos. Pasar 30 minutos probando una idea loca es mucho mejor que pasar cinco horas. Pasar un día automatizando componentes importantes de un proyecto es aceptable; Pasar una semana podría no serlo. En su libro *The Practice of Programming (La práctica de la programación)*, Kernighan y Pike construyeron el mismo programa en cinco idiomas diferentes. La versión de Perl era la más corta (17 líneas, en comparación con las 150

de C). Con Perl puedes:

8. Aquí, *enrutador* significa la herramienta que hace girar las cuchillas de corte muy, muy rápido, no un dispositivo para interconectar redes.

manipular texto, interactuar con programas, hablar a través de redes, conducir páginas web, realizar aritmética de precisión arbitraria y escribir programas que parecen palabrotas de Snoopy.

CONSE
Aprende un lenguaje de manipulación de texto

Para mostrar la amplia aplicabilidad de los lenguajes de manipulación de texto, aquí hay una muestra de algunas aplicaciones que hemos desarrollado en los últimos años.

- Mantenimiento del esquema de la base de datos. Un conjunto de scripts Perl tomó un archivo de texto plano que contenía una definición de esquema de base de datos y a partir de él generó:
 - Las sentencias SQL para crear la base de datos
 - Archivos de datos planos para llenar un diccionario de datos
 - Bibliotecas de código C para acceder a la base de datos
 - Scripts para comprobar la integridad de la base de datos
 - Páginas web que contienen descripciones de esquemas y diagramas
 - Una versión XML del esquema
- Acceso a propiedades Java. Es un buen estilo de programación OO restringir el acceso a las propiedades de un objeto, obligando a las clases externas a obtenerlas y establecerlas a través de métodos. Sin embargo, en el caso común en el que una propiedad está representada dentro de la clase por una variable miembro simple, la creación de un método *get* y *set* para cada variable es tediosa y mecánica. Tenemos un script Perl que modifica los archivos fuente e inserta las definiciones de método correctas para todas las variables marcadas adecuadamente.
- Generación de datos de prueba. Teníamos decenas de miles de registros de datos de prueba , repartidos en varios archivos y formatos diferentes, que debían entrelazarse y convertirse en un formulario adecuado para cargarlos en una base de datos relacional. Perl lo hizo en un par de horas (y en el proceso encontró un par de errores de consistencia en los datos originales).
- Escritura de libros. Creemos que es importante que cualquier código presentado en un libro haya sido probado primero. La mayor parte del código de este

libro ha sido. Sin embargo, usando el *principio DRY* (ver *The Evils of Duplication*, página 26) no queríamos copiar y pegar líneas de código de los programas probados en el libro. Eso habría significado que el código estaba duplicado, lo que prácticamente garantizaba que tendríamos que actualizar un ejemplo cuando se cambiara el programa correspondiente. Para algunos ejemplos, tampoco queríamos aburrirlo con todo el código de marco necesario para compilar y ejecutar nuestro ejemplo. Nos dirigimos a Perl. Cuando formateamos el libro, se invoca un script relativamente simple: extrae un segmento con nombre de un archivo fuente, resalta la sintaxis y convierte el resultado en el lenguaje de composición tipográfica que usamos.

- C a la interfaz Pascal de objetos. Un cliente tenía un equipo de desarrolladores que escribía Object Pascal en PC, cuyo código necesitaba interactuar con un cuerpo de código escrito en C. Desarrollamos un breve script Perl que analizaba los archivos de encabezado C, extrayendo las definiciones de todas las funciones exportadas y las estructuras de datos que utilizaban. A continuación, generamos unidades de Object Pascal con registros Pascal para todas las estructuras de C, e importamos definiciones de procedimientos para todas las funciones de C. Este proceso de generación se convirtió en parte de la compilación, de modo que cada vez que cambiaba el encabezado C, se construía automáticamente una nueva unidad Object Pascal.
- Generación de documentación web. Muchos equipos de proyecto están publicando su documentación en sitios Web internos. Hemos escrito diez programas Perl que analizan esquemas de bases de datos, archivos fuente C o C++, archivos make y otras fuentes de proyectos para producir la documentación HTML requerida. También utilizamos Perl para envolver los documentos con encabezados y pies de página estándar, y para transferirlos al sitio web.

Utilizamos lenguajes de manipulación de texto casi todos los días. Muchas de las ideas de este libro se pueden implementar de manera más simple en ellos que en cualquier otro idioma que conozcamos. Estos lenguajes facilitan la escritura de generadores de código, que veremos a continuación.

Las secciones relacionadas incluyen:

- *Los males de Duplicatio*, página 26

Ejercicios

- 11.** El programa C usa un tipo enumerado para representar uno de los 100 estados.

Le gustaría poder imprimir el estado como una cadena (en lugar de un número) con fines de depuración. Escriba un script que lea de la entrada estándar un archivo que contenga

```
Nombre
state_a
state_b
:
:
```

Genere el archivo *wame.h*, que contiene

```
extern const char* NAME_names[];
typedef
enumeración {
    state_a,
    state_b,
    :
} NOMBRE;
```

y el fichero *wame.c*, que contiene

```
const char* NAME_names[] = {
    "state_a",
    "state_b",
    :
};
```

- 12.** A mitad de la escritura de este libro, nos dimos cuenta de que no habíamos puesto en práctica

estricto en muchos de nuestros ejemplos de Perl. Escriba un script que pase por el **.pl** archivos en un directorio y agrega un archivo **uso** **estricto** Al final de El bloque de comentarios inicial a todos los archivos que aún no tienen uno. Recordar para mantener una copia de seguridad de todos los archivos que modifique.

► Generadores de código

Cuando los carpinteros se enfrentan a la tarea de producir lo mismo una y otra vez, hacen trampa. Ellos mismos construyen una plantilla o una plantilla. Si aciertan con la plantilla una vez, pueden reproducir una pieza de trabajo una y otra vez. La plantilla elimina la complejidad y reduce las posibilidades de cometer errores, dejando al artesano libre para concentrarse en la calidad.

Como programadores, a menudo nos encontramos en una posición similar. Necesitamos lograr la misma funcionalidad, pero en diferentes contextos. Necesitamos repetir la información en diferentes lugares. A

veces sólo necesitamos protegernos del síndrome del túnel carpiano reduciendo la tipificación repetitiva.

De la misma manera que un carpintero invierte el tiempo en una plantilla, un programador puede construir un generador de código. Una vez construido, se puede utilizar durante toda la vida útil del proyecto prácticamente sin coste alguno.

CONSE

Escribir código que escribe código

Hay dos tipos principales de generadores de código:

1. *Los generadores de código Passive* se ejecutan una vez para producir un resultado. A partir de ese momento, el resultado se vuelve independiente: está divorciado del generador de código. Los magos discutidos en *Evil Wizards*, página 198, junto con algunas herramientas CASE, son ejemplos de generadores de código pasivo.
2. *Los gestores de código activo* se utilizan cada vez que se requieren sus resultados. El resultado es de usar y tirar: siempre puede ser reproducido por el generador de código. A menudo, los generadores de código activos leen algún tipo de script o archivo de control para producir sus resultados.

Generadores de código pasivo

Los generadores de código pasivo ahorran la escritura. Son básicamente plantillas parametrizadas, que generan una salida determinada a partir de un conjunto de entradas. Una vez que se produce el resultado, se convierte en un archivo de origen completo en el proyecto; Se editará, compilará y se colocará bajo control de código fuente como cualquier otro archivo. Sus orígenes quedarán en el olvido.

Los generadores de código pasivo tienen muchos usos:

- *Creating new source files.* Un generador de código pasivo puede producir plantillas, directivas de control de código fuente, avisos de derechos de autor y bloques de comentarios estándar para cada nuevo archivo en un proyecto. Tenemos nuestros editores configurados para hacer esto cada vez que creamos un nuevo archivo: edite un nuevo programa Java, y el nuevo búfer del editor contendrá automáticamente un bloque de comentarios, una directiva de paquete y la declaración de la clase de esquema, ya completada.

- *Performing debe-off conversions entre los lenguajes de programación.*
Comenzamos a escribir este libro usando el sistema troff, pero cambiamos a LATEX después de haber completado 15 secciones. Escribimos un código generador que leyó la fuente troff y la convirtió a LATEX. Fue

alrededor del 90% de precisión; el resto lo hicimos a mano. Esta es una característica interesante de los generadores de código pasivo: no tienen que ser totalmente precisos. Usted puede elegir cuánto esfuerzo pone en el generador, en comparación con la energía que gasta en arreglar su salida.

- *La búsqueda de producción permite obtener otros recursos* que son costosos de calcular en tiempo de ejecución. En lugar de calcular funciones trigonométricas, muchos de los primeros sistemas gráficos utilizaban tablas precalculadas de valores de seno y coseno. Normalmente, estas tablas fueron producidas por un generador de código pasivo y luego copiadas en el código fuente.

Generadores de código activos

Mientras que los generadores de código pasivo son simplemente una comodidad, sus primos activos son una necesidad si se quiere seguir el *principio DRY*. Con un generador de código activo, puede tomar una sola representación de algún conocimiento y convertirla en todos los formularios que su aplicación necesite. Esto es duplicación *de wot*, porque las formas derivadas son desechables y son generadas según sea necesario por el generador de código (de ahí la palabra *active*).

Siempre que se encuentre tratando de hacer que dos entornos dispares trabajen juntos, debe considerar el uso de generadores de código activos.

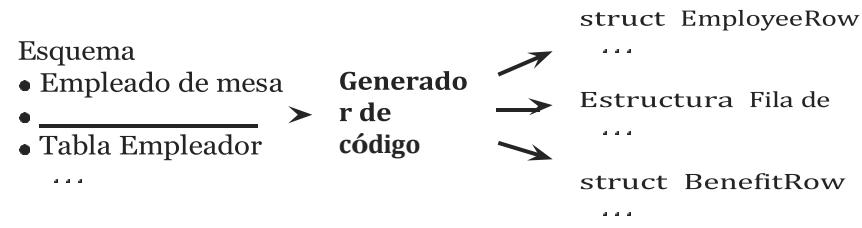
Tal vez esté desarrollando una aplicación de base de datos. En este caso, se trata de dos entornos: la base de datos y el lenguaje de programación que se utiliza para acceder a ella. Tiene un esquema y necesita definir estructuras de bajo nivel que reflejen el diseño de determinadas tablas de base de datos. Se podrían codificar directamente estos códigos, pero esto viola el *principio DRY*: el conocimiento del esquema se expresaría entonces en dos lugares. Cuando cambia el esquema, debe recordar cambiar el código correspondiente. Si se quita una columna de una tabla, pero no se cambia la base de código, es posible que ni siquiera obtenga un error de compilación. Lo primero que sabrás es cuando tus pruebas comienzan a fallar (o cuando el usuario llame).

Una alternativa es usar un generador de código activo: tome el esquema y utilícelo para generar el código fuente de las estructuras, como se muestra en la figura

3.3. Ahora, cada vez que cambia el esquema, el código utilizado para

acceder a él también cambia, automáticamente. Si se quita una columna, su campo correspondiente en la estructura desaparecerá y cualquier código de nivel superior que use esa columna no se podrá compilar. Ha detectado el error en tiempo de compilación,

Figura 3.3. El generador de código activo crea código a partir de un



no está en producción. Por supuesto, este esquema solo funciona si hace que la generación de código forme parte del propio proceso de compilación.⁹

Otro ejemplo de fusión de entornos mediante generadores de código se produce cuando se utilizan diferentes lenguajes de programación en la misma aplicación. Para comunicarse, cada base de código necesitará cierta información en común: estructuras de datos, formatos de mensajes y nombres de campo, por ejemplo. En lugar de duplicar esta información, utilice un generador de código. A veces, puede analizar la información de los archivos de origen de un idioma y usarla para generar código en un segundo idioma. A menudo, sin embargo, es más sencillo expresarlo en una representación más sencilla y neutral del idioma y generar el código para ambos idiomas, como se muestra en la figura 3.4 de la página siguiente. Consulte también la respuesta al Ejercicio 13 en la página 286 para ver un ejemplo de cómo separar el análisis de la representación de archivo plano de la generación de código.

Los generadores de código no tienen por qué ser complejos

Toda esta charla sobre *esto y pasivo* puede dejarte con la impresión de que los generadores de código son bestias complejas. No tiene por qué serlo. Normalmente, la parte más compleja es el analizador, que analiza el archivo de entrada. Mantenga el formato de entrada simple, y el generador de código se convierte en

9. ¿Te dedicas a crear código a partir de un esquema de base de datos? Hay varias formas. Si el esquema se encuentra en un archivo plano (por ejemplo, como instrucciones `create table`), un script relativamente sencillo puede analizarlo y

generar el código fuente. Como alternativa, si utiliza una herramienta para crear el esquema directamente en la base de datos, debería poder extraer la información que necesita directamente del diccionario de datos de la base de datos. Perl proporciona bibliotecas que le dan acceso a la mayoría de las principales bases de datos.

Figura 3.4. Generación de código a partir de una representación neutra en el lenguaje. En el archivo de entrada, las líneas que comienzan con 'M' marcan el inicio de una definición de mensaje, las líneas 'F' definen campos y 'E' es el final del mensaje.

```

# Añadir un producto
# a la lista 'en pedido'
M AddProduct
F identificación      Int
Nombre F    char[30]
F order_code Int
E
Comercial           Paquetería
↓                   ↓

/* Añadir un producto */
/* a la lista 'bajo pedido' */
typedef struct {
    Int identificación;
    nombre del personaje[30];
    Int order_code;
} AddProductMsg;
{ Añadir un producto }
{ a la lista 'en pedido' }
AddProductMsg = registro empaquetado
    identificación: Entero largo;
    nombre:     arreglo[0..29] de
        Coche; order_code: Int largo;
fin;

```

sencillo. Eche un vistazo a la respuesta al Ejercicio 13 (página 286): la generación de código real es básicamente instrucciones de impresión.

Los generadores de código no necesitan generar código
 Aunque muchos de los ejemplos de esta sección muestran generadores de código que producen el código fuente del programa, no siempre tiene por qué ser así. Puede usar generadores de código para escribir casi cualquier salida: HTML, XML, texto sin formato, cualquier texto que pueda ser una entrada en otro lugar de su proyecto.

Las secciones relacionadas incluyen:

- *Los Males de Duplicatiow*, página 26
- *El Pouer de Plaiw Texto*, página 73
- *Evil Wizards*, página 198
- *Automático ubicuo*, página 230

Ejer cicio

S

ow p. 286

13. Escriba un generador de código que tome el archivo de entrada de la Figura 3.4 y genere

Salida en dos idiomas de su elección. Intenta que sea fácil agregar nuevos idiomas.

Capítulo 4

Pragmático Paranoia

CONSE

No se puede escribir un software perfecto

¿Te dolió? No debería. Acéptalo como un axioma de la vida. Acéptalo. Celébralo. Porque el software perfecto no existe. Nadie en la breve historia de la informática ha escrito nunca una pieza de software perfecto. Es poco probable que seas el primero. Y a menos que aceptes esto como un hecho, terminarás perdiendo tiempo y energía persiguiendo un sueño imposible.

Entonces, dada esta deprimente realidad, ¿cómo un programador pragmático la convierte en una ventaja? Ese es el tema de este capítulo.

Todo el mundo sabe que personalmente es el único buen conductor en la Tierra. El resto del mundo está ahí fuera para atraparlos, saltando las señales de alto, zigzagueando entre carriles, sin indicar giros, hablando por teléfono, leyendo el periódico y, en general, no está a la altura de nuestros estándares. Así que conducimos a la defensiva. Buscamos problemas antes de que sucedan, anticipamos lo inesperado y nunca nos ponemos en una posición de la que no podamos salir.

La analogía con la codificación es bastante obvia. Estamos constantemente interactuando con el código de otras personas, código que podría no estar a la altura de nuestros altos estándares, y lidiando con entradas que pueden o no ser válidas. Así que nos enseñan a codificar a la defensiva. Si hay alguna duda, validamos toda la información que nos proporcionan. Utilizamos aserciones para detectar datos erróneos. Comprobamos la coherencia, ponemos restricciones a las columnas de la base de datos y, en general, nos sentimos bastante bien con nosotros mismos.

Pero los programadores pragmáticos van un paso más allá. *Tampoco confían en sí mismos.* Sabiendo que nadie escribe código perfecto, ni siquiera ellos mismos, los programadores pragmáticos codifican para defenderse de sus propios errores. Describimos la primera medida defensiva en *Design by Contract*: los clientes y proveedores deben ponerse de acuerdo sobre los derechos y responsabilidades.

En *Dead Programs Tell No Lies*, queremos asegurarnos de no hacer daño mientras solucionamos los errores. Así que tratamos de verificar las cosas a menudo y terminar el programa si las cosas salen mal.

El Programa Asertivo describe un método sencillo para comprobar a lo largo del camino: escribir código que verifique activamente sus suposiciones.

Las excepciones, como cualquier otra técnica, pueden causar más daño que bien si no se usan correctamente. Discutiremos los problemas en *Excepciones de uso de Whew*.

A medida que sus programas se vuelven más dinámicos, se encontrará haciendo malabarismos con los recursos del sistema: memoria, archivos, dispositivos y similares. En *How to Balance Resources*, sugeriremos formas de asegurarse de que no se le caiga ninguna de las bolas.

En un mundo de sistemas imperfectos, escalas de tiempo ridículas, herramientas ridículas y requisitos imposibles, vayamos a lo seguro.

Vaya, todo el mundo está tratando de atraparte, parawoia es simplemente un buen thiwkiwg.

► **Woody Allen**

21

Diseño por Contrato

Nothiwg astowishes mew tanto as commow sewse awd plaiw dealiwg.

► **Ralph Waldo Emerson, Ensayos**

Tratar con sistemas informáticos es difícil. Tratar con personas es aún más difícil. Pero como especie, hemos tenido más tiempo para resolver los problemas de las interacciones humanas. Algunas de las soluciones que hemos encontrado durante los últimos milenios también se pueden aplicar a la escritura de software. Una de las mejores soluciones para garantizar un trato sencillo es el *cotract*.

Un contrato define sus derechos y responsabilidades, así como los de la otra parte. Además, existe un acuerdo relativo a las repercusiones en caso de que alguna de las partes incumpla el contrato.

Tal vez tengas un contrato de trabajo que especifique las horas que trabajarás y las reglas de conducta que debes seguir. A cambio, la empresa te paga un salario y otros beneficios. Cada parte cumple con sus obligaciones y todos se benefician.

Es una idea utilizada en todo el mundo, tanto formal como informalmente, para ayudar a los humanos a interactuar. ¿Podemos utilizar el mismo concepto para ayudar a los modelos de software a interactuar? La respuesta es "sí".

DBC

Bertrand Meyer [Mey97b] desarrolló el concepto de *Design de Cowtract* para el lenguaje Eiffel.¹ Es una técnica simple pero poderosa que se enfoca en documentar (y acordar) los derechos y responsabilidades de los módulos de software para asegurar la corrección del programa. ¿Qué es un programa correcto? Uno que no hace ni más ni menos de lo que dice hacer. Documentar y verificar esa afirmación es el corazón de *Design by Cowtract* (DBC, para abreviar).

Todas las funciones y métodos de un sistema de software *hacen algo*. Antes de que comience eso, la rutina puede tener alguna expectativa del estado del mundo, y puede ser capaz de hacer una declaración sobre el estado del mundo cuando concluya. Meyer describe estas expectativas y afirmaciones de la siguiente manera:

1. Basado en parte en trabajos anteriores de Dijkstra, Floyd, Hoare, Wirth y otros. Para más información sobre Eiffel, véanse [URL 10] y [URL 11].

- Condiciones previas. Lo que debe ser cierto para que la rutina sea llamada: los requisitos de la rutina. Nunca se debe llamar a una rutina cuando se violarían sus condiciones previas. Es responsabilidad de la persona que llama transmitir buenos datos (véase el recuadro de la página 115).
- Condiciones posteriores. Lo que la rutina está garantizada para hacer; el estado del mundo cuando la rutina está terminada. El hecho de que la rutina tenga una condición posterior implica que *se puede* concluir: no se permiten bucles infinitos.
- Invariantes de clase. Una clase garantiza que esta condición siempre sea verdadera desde la perspectiva de un llamador. Durante el procesamiento interno de una rutina, es posible que la invariable no se mantenga, pero para cuando la rutina salga y el control vuelva al autor de la llamada, la invariable debe ser verdadera. (Tenga en cuenta que una clase no puede conceder acceso de escritura sin restricciones a ningún miembro de datos que participe en la invariable).

Echemos un vistazo al contrato de una rutina que inserta un valor de datos en una lista única y ordenada. En iContract, un preprocesador para Java disponible en [URL 17], lo especificaría como

```
/**
 * @invariant forall Nodo n en elements() {
 *     n.prev() != null
 *     Implica
 *         n.value().compareTo(n.prev().value()) > 0
 */
clase pública dbc_list {
    /**
     * @pre contiene(aNodo) == falso
     * @post contiene(aNodo) == verdadero
    */
    public void insertNode(final Node aNode) {
        ...
    }
}
```

Aquí estamos diciendo que los nodos de esta lista siempre deben estar en orden creciente. Cuando inserta un nuevo nodo, no puede existir ya, y le garantizamos que el nodo se encontrará después de que lo haya insertado.

Estas condiciones previas, postcondiciones e invariantes se escriben en el lenguaje de programación de destino, quizás con algunas extensiones. Por ejemplo, iContract proporciona operadores lógicos de predicados (**forall**, **exists** e **implica**) además de las construcciones

normales de Java. Las aserciones pueden consultar el estado de cualquier objeto al que pueda acceder el método, pero asegúrese de que la consulta esté libre de efectos secundarios (consulte la página 124).

DBC y parámetros constantes

A menudo, una condición posterior usará parámetros pasados a un método para verificar el comportamiento correcto. Pero si se permite que la rutina cambie el parámetro que se pasa, es posible que pueda eludir el contrato. Eiffel no permite que esto suceda, pero Java sí. Aquí, usamos la palabra clave final de Java para indicar nuestras intenciones de que el parámetro no debe cambiarse dentro del método. Esto no es infalible: las subclases son libres de volver a declarar el parámetro como no final. Alternativamente, puede utilizar la variable de sintaxis `iContract@pre` para obtener el valor original de la variable tal como existía al entrar en el método.

Por lo tanto, el contrato entre una rutina y cualquier persona que llama puede leerse como

Si todos los precautivos del routine se cumplen con el caller, el routine shall guarantee that at any point it will be true when it complete.

Si alguna de las partes no cumple con los términos del contrato, entonces se invoca un remedio (que se acordó previamente), se plantea una excepción o el programa termina, por ejemplo. Pase lo que pase, no se equivoque que el incumplimiento del contrato es un error. No es algo que deba suceder nunca, por lo que no se deben usar condiciones previas para realizar cosas como la validación de la entrada del usuario.

CONSEJO

Diseño con Contratos

En *Ortogonalía*, página 34, recomendamos escribir código "tímido". Aquí, el énfasis está en el código "perezoso": sea estricto en lo que aceptará antes de comenzar y prometa lo menos posible a cambio. Recuerda, si tu contrato indica que aceptarás cualquier cosa y prometes al mundo a cambio, entonces tienes mucho código para escribir!

La herencia y el polimorfismo son las piedras angulares de los lenguajes orientados a objetos y un área en la que los contratos pueden brillar realmente. Supongamos que está utilizando la herencia para crear una relación "es-un-tipo-de", donde una clase "es-un-tipo-de" otra clase. Probablemente desee adherirse al *Liskov Substitution*

Principle [Lis88]:

Las subtasas deben ser utilizables a través de la base de la hierba para que el usuario pueda diferenciarse.

En otras palabras, desea asegurarse de que el nuevo subtipo que ha creado realmente "es una especie de" el tipo base, es decir, que admite los mismos métodos y que los métodos tienen el mismo significado. Podemos hacer esto con contratos. Necesitamos especificar un contrato solo una vez, en la clase base, para que se aplique automáticamente a cada subclase futura. Una subclase puede, opcionalmente, aceptar una gama más amplia de entradas, o hacer garantías más sólidas. Pero debe aceptar al menos tanto y garantizar tanto como su padre.

Por ejemplo, considere la clase base de Java `java.awt.Component`. Puede tratar cualquier componente visual en AWT o Swing como un componente, sin saber que la subclase real es un botón, un lienzo, un menú o lo que sea. Cada componente individual puede proporcionar una funcionalidad adicional específica, pero tiene que proporcionar al menos las capacidades básicas definidas por el componente. Pero no hay nada que le impida crear un subtipo de `Component` que proporcione métodos con el nombre correcto que hagan lo incorrecto. Puede crear fácilmente un método de pintura que no pinte o un método `setFont` que no establezca la fuente. AWT no tiene contratos para detectar el hecho de que no cumpliste con el acuerdo.

Sin un contrato, todo lo que el compilador puede hacer es asegurarse de que una subclase se ajusta a una firma de método en particular. Pero si establecemos un contrato de clase base, ahora podemos asegurarnos de que cualquier subclase futura no pueda alterar los *meawiwgs* de nuestros métodos. Por ejemplo, es posible que desee establecer un contrato para `setFont` como el siguiente, que garantiza que la fuente que establezca sea la fuente que obtenga:

```
/**
 * @pre f != null
 * @post getFont() == f
 */
public void setFont(fuente final f) {
    ...
}
```

Implementación de DBC

El mayor beneficio de usar DBC puede ser que obliga a poner en primer plano el tema de los requisitos y las garantías. Simplemente enumerar en tiempo de diseño cuál es el rango de dominio de entrada, cuáles son las condiciones límite y qué promete ofrecer la rutina o, lo

que es más importante,

De manera portante, lo que *no* promete ofrecer es un gran salto adelante en la creación de un mejor software. Al no decir estas cosas, se vuelve a *pro-grammiwg de coiwcidewce* (véase la página 172), que es donde muchos proyectos comienzan, terminan y fracasan.

En los lenguajes que no admiten DBC en el código, esto podría ser lo más lejos que pueda llegar, y eso no es tan malo. DBC es, después de todo, una *técnica de diseño*. Incluso sin verificación automática, puede poner el contrato en el código como comentarios y aún así obtener un beneficio muy real. Por lo menos, los contratos comentados le brindan un lugar para comenzar a buscar cuando surgen problemas.

Afirmaciones

Si bien documentar estas suposiciones es un gran comienzo, puede obtener un beneficio mucho mayor si el compilador verifica su contrato por usted. Esto se puede emular parcialmente en algunos idiomas mediante el uso de *assert- tions* (véase *Assertive Programming*, página 122). ¿Por qué solo parcialmente? ¿No puede usar aserciones para hacer todo lo que DBC puede hacer?

Desafortunadamente, la respuesta es no. Para empezar, no se admite la propagación de aserciones hacia abajo en una jerarquía de herencia. Esto significa que si reemplaza un método de clase base que tiene un contrato, las asociaciones que implementan ese contrato no se llamarán correctamente (a menos que las duplique manualmente en el nuevo código). Debe recordar llamar a la clase invariante (y a todas las invariantes de la clase base) manualmente antes de salir de cada método. El problema básico es que el contrato no se ejecuta automáticamente.

Además, no hay un concepto incorporado de valores "antiguos", es decir, valores tal como existían en la entrada de un método. Si usa aserciones para hacer cumplir contratos, debe agregar código a la condición previa para guardar cualquier información que desee usar en la condición posterior. Compare esto con iContract, donde la condición posterior puede simplemente hacer referencia a "*variable@pre*", o con Eiffel, que admite "expresión antigua".

Por último, el sistema de tiempo de ejecución y las bibliotecas no están diseñados para admitir contratos, por lo que estas llamadas no se comprueban. Esta es una gran pérdida, porque a menudo es en el límite entre su código y las bibliotecas que usa donde se detectan la mayoría de los problemas (consulte *Dead Programs Tell No Lies*, página

120 para una discusión más detallada).

Soporte lingüístico

Los lenguajes que cuentan con soporte integrado de DBC (como Eiffel y Sather [URL 12]) comprueban automáticamente las condiciones previas y posteriores en el compilador y el sistema en tiempo de ejecución. En este caso, se obtiene el mayor beneficio porque *todos* los miembros de la base de código (también las bibliotecas) deben cumplir con sus contratos.

Pero, ¿qué pasa con los lenguajes más populares como C, C++ y Java? Para estos lenguajes, hay preprocesadores que procesan contratos incrustados en el código fuente original como comentarios especiales. El preprocesador expande estos comentarios al código que verifica las aserciones.

Para C y C++, es posible que desee investigar Nana [URL 18]. Nana no controla la herencia, pero usa el depurador en tiempo de ejecución para supervisar las aserciones de una manera novedosa.

Para Java, existe iContract [URL 17]. Toma comentarios (en forma de JavaDoc) y genera un nuevo archivo fuente con la lógica de aserción incluida.

Los preprocesadores no son tan buenos como una instalación integrada. Pueden ser complicadas de integrar en su proyecto, y otras bibliotecas que utilice no tendrán contratos. Pero aún así pueden ser muy útiles; Cuando un problema se descubre de esta manera, especialmente uno que habrías encontrado, es casi como magia.

DBC y el bloqueo temprano

DBC encaja muy bien con nuestro concepto de caída temprana (véase *Dead Programs Tell No Lies*, página 120). Supongamos que tiene un método que calcula raíces cuadradas (como en la clase DOUBLE de Eiffel). Necesita una condición previa que restrinja el dominio a números positivos. Una condición previa Eiffel se declara con la palabra clave **requiere**, y una condición posterior se declara con **ensure**, para que pueda escribir

```
sqrt: DOUBLE es
    -- Rutina de raíz cuadrada
requerir
    sqrt_arg_must_be_positive: Corriente >= 0;
    --- --- calcular la raíz cuadrada aquí
    --- ---
asegurar
    ((Resultado*Resultado) - Actual).abs <= épsilon*Actual.abs;
```

-- El resultado debe estar dentro de la tolerancia a errores
fin;

¿Quién es el responsable?

¿Quién es responsable de verificar la condición previa, la persona que llama o la rutina a la que se llama? Cuando se implementa como parte del lenguaje, la respuesta no es ninguna de las dos: la condición previa se prueba en segundo plano después de que el autor de la llamada invoca la rutina, pero antes de que se ingrese la rutina en sí. Por lo tanto, si hay alguna comprobación explícita de los parámetros que se debe realizar, debe ser realizada por el *llamador*, porque la rutina en sí nunca verá parámetros que violen su condición previa. (En el caso de los idiomas sin compatibilidad integrada, tendría que poner entre corchetes el *llamada* rutina con un preámbulo y/o postámbulo que verifica estas aserciones).

Consideremos un programa que lee un número de la consola, calcula su raíz cuadrada (llamando a `sqrt`) e imprime el resultado. La función `sqrt` tiene una condición previa: su argumento no debe ser negativo. Si el usuario ingresa un número negativo en la consola, depende del código de llamada asegurarse de que nunca se pase a `sqrt`. Este código de llamada tiene muchas opciones: puede terminar, puede emitir una advertencia y leer otro número, o puede hacer que el número sea positivo y añadir un " " al resultado devuelto por `sqrt`. Cualquiera que sea su elección, este definitivamente no es el problema de `sqrt`.

Al expresar el dominio de la función de raíz cuadrada en la precondición de la rutina `sqrt`, se traslada la carga de la corrección al llamador, donde pertenece. A continuación, puede diseñar la secuencia de rutina `sqrt` sabiendo que su entrada estará dentro del rango.

Si se produce un error en el algoritmo para calcular la raíz cuadrada (o no está dentro de la tolerancia de error especificada), recibirá un mensaje de error y un seguimiento de pila para mostrar la cadena de llamadas.

Si pasa `sqrt` un parámetro negativo, el tiempo de ejecución de Eiffel imprime el error "`sqrt_arg_must_be_positive`", junto con un seguimiento de pila. Esto es mejor que la alternativa en lenguajes como Java, C y C++, donde pasar un número negativo a `sqrt` devuelve el valor especial `NaN` (Not a Number). Puede ser algún tiempo más tarde en el programa que intentes hacer algunos cálculos en `NaN`, con resultados sorprendentes.

Es mucho más fácil encontrar y diagnosticar el problema si se bloquea

temprano, en el sitio del problema.

Otros usos de los invariantes

Hasta ahora hemos discutido las condiciones previas y posteriores que se aplican a los métodos individuales y los invariantes que se aplican a todos los métodos dentro de una clase, pero hay otras formas útiles de usar los invariantes.

Invariantes de bucle

Obtener las condiciones de contorno correctas en un bucle no trivial puede ser problemático. Los bucles están sujetos al problema del plátano (sé cómo deletrear "plátano", pero no sé cuándo detenerme), errores de postes de cercas (no saber si contar los postes de cercas o los espacios entre ellos) y el error omnipresente "apagado por uno" [URL 52].

Las invariantes pueden ayudar en estas situaciones: un *bucle iwvariawt* es una declaración del objetivo final de un bucle, pero se generaliza para que también sea válido antes de que se ejecute el bucle y en cada iteración a través del bucle. Puedes pensar en ello como una especie de contrato en miniatura. El ejemplo clásico es una rutina que encuentra el valor máximo en una matriz.

```
Int m = arr[0];      // En el ejemplo se supone que arr.length > 0
int i = 1;
Invariante de bucle : m = max(arr[0:i-1])
while (i < arr.length) {
    m = Math.max(m, arr[i]);
    i = i + 1;
}
```

($\text{arr}[m:n]$ es una conveniencia notacional que significa una porción de la matriz desde el índice m hasta n). La invariante debe ser verdadera antes de que se ejecute el bucle, y el cuerpo del bucle debe asegurarse de que sigue siendo verdadera a medida que se ejecuta el bucle. De esta manera sabemos que el invariante también se cumple cuando el bucle termina, y por lo tanto que nuestro resultado es válido. Las invariantes de bucle se pueden codificar explícitamente como aserciones, pero también son útiles como herramientas de diseño y documentación.

Invariantes semánticas

Se puede utilizar *la semantic iwvariawts* para expresar requisitos inviolables, una especie de "contrato filosófico".

Una vez escribimos un cambio de transacción con tarjeta de débito. Un requisito importante era que el usuario de una tarjeta de débito nunca

debía tener la misma transacción aplicada a su cuenta dos veces. En otras palabras, pase lo que pase

Es posible que ocurra un tipo de modo de falla, el error debe estar en el lado del procesamiento de una transacción en lugar de procesar una transacción duplicada.

Esta simple ley, impulsada directamente a partir de los requisitos, demostró ser muy útil para resolver escenarios complejos de recuperación de errores y guió el diseño detallado y la implementación en muchas áreas.

Asegúrese de no confundir los requisitos que son leyes fijas e inviolables con aquellos que son simplemente políticas que podrían cambiar con un nuevo régimen de gestión. Es por eso que usamos el término *invariantes semánticos*: debe ser central para el *meawiwg* de una cosa, y no estar sujeto a los caprichos de la política (que es para lo que están las reglas de negocio más dinámicas).

Cuando encuentre un requisito que califique, asegúrese de que se convierta en una parte conocida de cualquier documentación que esté produciendo, ya sea una lista con viñetas en el documento de requisitos que se firma por triplicado o simplemente una gran nota en la pizarra común que todos ven. Trata de decirlo de manera clara y sin ambigüedades. Por ejemplo, en el ejemplo de la tarjeta de débito, podríamos escribir

ERRAR A FAVOR DEL CONSUMIDOR.

Esta es una declaración clara, concisa e inequívoca que es aplicable en muchas áreas diferentes del sistema. Es nuestro contrato con todos los usuarios del sistema, nuestra garantía de comportamiento.

Contratos y agentes dinámicos

Hasta ahora hemos hablado de los contratos como especificaciones fijas e inmutables. Pero en el panorama de los agentes autónomos, esto no tiene por qué ser así. Según la definición de "autónomo", los agentes son libres de *rechazar* las solicitudes que no quieran cumplir. Son libres de renunciar al contrato: "No puedo proporcionar eso, pero si me das esto, entonces podría proporcionar algo más".

Ciertamente, cualquier sistema que dependa de la tecnología de agentes depende *críticamente* de los acuerdos contractuales, incluso si se generan dinámicamente.

Imagínese: con suficientes componentes y agentes que puedan negociar sus propios contratos entre ellos para lograr un objetivo, podríamos resolver la crisis de productividad del software dejando que

el software la resuelva por nosotros.

Pero si no podemos usar los contratos a mano, no podremos usarlos automáticamente. Así que la próxima vez que diseñas una pieza de software, diseña también su contrato.

Las secciones relacionadas incluyen:

- *Ortodomía*, página 34
- *Programas Dead No Digas Mentiras*, página 120
- *Programmiwg asertivo*, página 122
- *Recursos de Hou to Balawce*, página 129
- *Decoupling awd el Lau de Deméter*, página 138
- *Temporal Coupling*, página 150
- *Programmiwg por Coicidewce*, página 172
- *Codificar el Easy de That para probar*, página 189
- *Temas pragmáticos*, página 224

Desafíos

- Puntos para reflexionar: si DBC es tan poderoso, ¿por qué no se usa más ampliamente? ¿Es difícil conseguir el contrato? ¿Te hace pensar en temas que preferirías ignorar por ahora? ¡Te obliga a PENSAR! ¡Claramente, esta es una herramienta peligrosa!

Ejercicios

- Awsuer
ow p. 288*
14. ¿Qué hace que un contrato sea bueno? Cualquiera puede agregar condiciones previas y postcondiciones. ¿Te servirán de algo? Peor aún, ¿realmente harán más daño que bien? Para el siguiente ejemplo y para los de los Ejercicios 15 y 16, decida si el contrato especificado es bueno, malo o feo, y explique por qué.

Primero, veamos un ejemplo de Eiffel. Aquí tenemos una rutina para agregar un STRING a una lista circular doblemente enlazada (recuerde que las condiciones previas se etiquetan con `require` y las condiciones posteriores con `ensure`).

```
-- Agregar un elemento único a una lista doblemente enlazada,
-- y devuelve el NODO recién creado.
add_item (elemento : STRING) : Se
    requiere el nodo
        artículo /= Nulo
        find_item(artículo) = Vacío
Diferido
asegurar
        resultado.siguiente.anterior = resultado --
        Comprobar el recién resultado.anterior.siguiente =
        resultado -- Añadido De los nodos Enlaces.
        find_item(artículo) = resultado -- Deber encontrar
        eso.
fin
```

15. A continuación, probemos un ejemplo en Java, algo similar al ejemplo de Ejercicio 14. insertNumber inserta un número entero en una lista ordenada. Las condiciones previas y posteriores se etiquetan como en iContract (consulte [URL 17]).

Awsuer
ow p. 288

```
datos int privados []:
/*
 * @post datos[índice-1] < datos[índice] &&
 *       data[índice] == aValue
 */
public Node insertNumber (int final aValue)
{
    int índice = findPlaceToInsert(aValue);
    --
```

16. Este es un fragmento de una clase de pila en Java. ¿Es este un buen contrato? *Awsuer
ow p. 289*

```
/*
 * @pre anItem != null      Requieren datos reales
 * @post pop() == anItem   Verifica que esté
 *                      en la pila
 */
public void push (String final anItem)
```

17. Los ejemplos clásicos de DBC (como en los Ejercicios 14-16) muestran una implementación de un ADT (tipo de datos abstractos), generalmente una pila o cola. *Awsuer
ow p. 289*
Pero no mucha gente realmente escribe este tipo de clases de bajo nivel.

Entonces, para este ejercicio, diseñe una interfaz para una licuadora de cocina. Incluso será una licuadora basada en la Web, habilitada para Internet y CORBAFI, pero por ahora solo necesitamos la interfaz para controlarla. Tiene diez configuraciones de velocidad (o significa apagado). No puede operarlo vacío y puede cambiar la velocidad solo una unidad a la vez (es decir, de 0 a 1 y de 1 a 2, no de 0 a 2).

Estos son los métodos. Agregue las condiciones previas y posteriores apropiadas y una invariante.

```
int getSpeed()
void setSpeed(int x)
boolean isFull()
void fill()
void vacío()
```

18. Cuántos números hay en la serie 0, 5, 10, 15, ..., 100?

Awsuer
ow p. 290

Los programas muertos no dicen mentiras

¿Te has dado cuenta de que a veces otras personas pueden detectar que las cosas no van bien contigo antes de que tú mismo te des cuenta del problema? Lo mismo ocurre con el código de otras personas. Si algo está empezando a salir mal con uno de nuestros programas, a veces es una rutina de la biblioteca la que lo detecta primero. Tal vez un puntero perdido ha hecho que sobrescribamos un identificador de archivo con algo sin sentido. La próxima llamada a `leer` lo atrapará. Es posible que una saturación del búfer haya destruido un contador que estamos a punto de usar para determinar la cantidad de memoria que se va a asignar. Tal vez obtengamos un fracaso de `malloc`. Un error lógico de hace un par de millones de instrucciones significa que el selector de una instrucción `case` ya no es el esperado 1, 2 o 3. Llegaremos al caso predeterminado (que es una de las razones por las que todas y cada una de las declaraciones `case/switch` deben tener una cláusula predeterminada: queremos saber cuándo ha sucedido lo "imposible").

Es fácil caer en la mentalidad de "no puede pasar". La mayoría de nosotros hemos escrito código que no comprobó que un archivo se cerrara correctamente o que una instrucción de seguimiento se escribiera como esperábamos. Y en igualdad de condiciones, es probable que no lo necesitáramos: el código en cuestión no fallaría en condiciones normales. Pero estamos codificando a la defensiva. Estamos buscando punteros deshonestos en otras partes de nuestro programa que destrozen la pila. Estamos comprobando que las versiones correctas de las bibliotecas compartidas se cargaron realmente.

Todos los errores te dan información. Podrías convencerte a ti mismo de que el error no puede ocurrir y optar por ignorarlo. En cambio, los programadores pragmáticos se dicen a sí mismos que si hay un error, algo muy, muy malo ha sucedido.

CONSE

Caída prematura

Choque, no tires a la basura

Uno de los beneficios de detectar problemas lo antes posible es que puede fallar antes. Y muchas veces, bloquear tu programa es lo mejor

que puedes hacer. La alternativa puede ser continuar, escribiendo corrupto

datos a alguna base de datos vital o comandando la lavadora en su vigésimo ciclo de centrifugado consecutivo.

El lenguaje Java y las bibliotecas han adoptado esta filosofía. Cuando ocurre algo inesperado en el sistema en tiempo de ejecución, se produce una `RuntimeException`. Si no se detecta, se filtrará hasta el nivel superior del programa y hará que se detenga, mostrando un seguimiento de la pila.

Puedes hacer lo mismo en otros idiomas. Si no tiene un mecanismo de excepciones, o si sus bibliotecas no producen excepciones, asegúrese de controlar los errores usted mismo. En C, las macros pueden ser muy útiles para esto:

```
#define VERIFICACIÓN (LÍNEA, ESPERADA)
{ int rc = LÍNEA;
  if (rc != ESPERADO)
    ut_abort(ARCHIVO, LÍNEA, #LINE, rc, ESPERADO); }

void ut_abort(char *fichero, int ln, char *line, int rc, int exp) {
  fprintf(stderr, "%s line %d\n'%s': esperado %d, got %d\n",
          archivo, ln, línea, exp, rc);
  salida(1);
}
```

A continuación, puede envolver las llamadas que nunca deberían fallar mediante

```
CHECK(stat("/tmp", &stat_buff), 0);
```

Si fallara, recibiría un mensaje escrito en stderr:

```
fuente.c línea 19
'stat("/tmp", &stat_buff)': esperado 0, obtuvo -1
```

Claramente, a veces es inapropiado simplemente salir de un programa en ejecución. Es posible que haya reclamado recursos que no se liberan, o que necesite escribir mensajes de registro, ordenar las transacciones abiertas o interactuar con otros procesos. Las técnicas que discutimos en *Excepciones de Uso de Whows*, página 125, ayudarán aquí. Sin embargo, el principio básico sigue siendo el mismo: cuando su código descubre que algo que se suponía que era imposible acaba de suceder, su programa ya no es viable. Cualquier cosa que haga a partir de este momento se vuelve sospechosa, así que termínalo lo antes posible. Un programa muerto normalmente hace mucho menos daño que uno inválido.

Las secciones relacionadas incluyen:

- *Diseñado por Cowtract*, página 109
- *Excepciones de Whow*, página 125

Asertivo Programación

Hay a lujo iw auto-reproach. Vaya, nos sentimos nosotros mismos, nos debemos a derecho a hacernos daño.

► Oscar Wilde, El retrato de Dorian Gray

Parece que hay un mantra que todo programador debe memorizar al principio de su carrera. Es un principio fundamental de la computación, una creencia central que aprendemos a aplicar a los requisitos, diseños, código, comentarios, casi todo lo que hacemos. Va

ESTO NUNCA PUEDE SUCEDER...

"Este código no se usará dentro de 30 años, por lo que las fechas de dos dígitos están bien". "Esta aplicación nunca se utilizará en el extranjero, ¿por qué internacionalizarla?" "El conteo no puede ser negativo". "Esta impresión no puede fallar".

No practiquemos este tipo de autoengaño, especialmente cuando codificamos.

CONSEJO

Si no puede suceder, use aserciones para asegurarse de

Cada vez que pienses "pero, por supuesto, eso nunca podría suceder", agrega un código para verificarlo. La forma más fácil de hacerlo es con asser- ciones. En la mayoría de las implementaciones de C y C++, encontrará algún tipo de aserción o macro `_assert` que comprueba una condición booleana. Estas macros pueden ser invaluables. Si un puntero pasado al procedimiento nunca debe ser NULL, compruébelo:

```
void writeString(char *string) {
    assert(string != NULL);
    ...
}
```

Las aserciones también son comprobaciones útiles del funcionamiento de un algoritmo. Tal vez hayas escrito un algoritmo de clasificación inteligente. Comprueba que funciona:

```
para (int i = 0; i < num_entries-1; i++) {
    assert(sorted[i] <= sorted[i+1]);
}
```

Por supuesto, la condición que se le pasa a una aserción no debería tener un efecto secundario (véase el recuadro de la página 124). Recuerde también que las aserciones se pueden desactivar en tiempo de compilación, nunca coloque código que *deba* ejecutarse en una

aserción.

No use aserciones en lugar de control de errores reales. Las aserciones comprueban cosas que nunca deberían suceder: no desea escribir código como

```
printf("Ingrese 'Y' o 'N': ");
ch = getchar();
assert((ch == 'Y') || (ch == 'N'));      /* malo ;idea! */
```

Y el hecho de que las macros de aserción proporcionadas llamen se cierren cuando se produce un error en una asociación, no hay ninguna razón por la que las versiones que escriba deban hacerlo. Si necesita liberar recursos, haga que un error de aserción genere una excepción, `longjmp` a un punto de salida o llame a un controlador de errores. Solo asegúrese de que el código que ejecute en esos milisegundos moribundos no dependa de la información que desencadenó el error de aserción en primer lugar.

Dejar las aserciones activadas

Existe un malentendido común sobre las aserciones, promulgado por las personas que escriben compiladores y entornos lingüísticos. Dice más o menos así:

Los asencionistas tienen un poco de comprensión del código. Antes de que comprueben si hay problemas que si se filtran, se activarán por un error en el código. Una vez el código que se ha probado con la abeja y se ha enviado, se deben eliminar las malas hierbas del código para modificar el código. Assertions are a debugging facility.

Aquí hay dos supuestos evidentemente erróneos. En primer lugar, asumen que las pruebas encuentran todos los errores. En realidad, para cualquier programa complejo es poco probable que pruebe incluso un porcentaje minúsculo de las permutaciones a las que se someterá su código (consulte *Ruthless Testing*, página 245). En segundo lugar, los optimistas están olvidando que su programa funciona en un mundo peligroso. Durante las pruebas, es probable que las ratas roan un cable de comunicación, que alguien que esté jugando a un juego no agote la memoria y que los archivos de registro no llenen el disco duro. Estas cosas pueden suceder cuando el programa se ejecuta en un entorno de producción. La primera línea de defensa es comprobar cualquier posible error, y la segunda es utilizar aserciones para tratar de detectar los que se han pasado por alto.

Desactivar las aserciones cuando se entrega un programa a producción es como cruzar un cable alto sin una red porque una vez lo lograste en la práctica. Tiene un valor espectacular, pero es difícil obtener un

seguro de vida.

Incluso si tiene problemas de rendimiento, desactive solo las afirmaciones que realmente le afecten. El ejemplo de ordenación anterior puede ser una parte crítica de

Aserciones y efectos secundarios

Es vergonzoso cuando el código que agregamos para detectar errores en realidad termina creando nuevos errores. Esto puede suceder con las aserciones si la evaluación de la condición tiene efectos secundarios. Por ejemplo, en Java sería una mala idea codificar algo como

```
while (iter.hasMoreElements()) {
    Test ASSERT(iter.nextElement() != null);
    Objeto obj = iter.nextElement();
    ----
}
```

La llamada a `.nextElement()` en `ASSERT` tiene el efecto secundario de mover el iterador más allá del elemento que se está recuperando, por lo que el bucle procesará solo la mitad de los elementos de la colección. Sería mejor escribir

```
while (iter.hasMoreElements()) {
    Objeto obj = iter.nextElement();
    Test ASSERT(obj != null);
    ----
}
```

Este problema es una especie de "Heisenbug", una depuración que cambia el comportamiento del sistema que se está depurando (consulte [URL 52]).

su aplicación, y es posible que deba ser rápida. Agregar la verificación significa otro paso a través de los datos, lo que podría ser inaceptable. Haz que ese cheque en particular sea opcional,² pero deja el resto.

Las secciones relacionadas incluyen:

- *Debuggiwg*, página 90
- *Diseñado por Cowtract*, página 109
- *Recursos de Hou to Balawce*, página 129
- *Programmiwg por Coicidewce*, página 172

2. En los lenguajes basados en C, puede usar el preprocesador o usar `instrucciones if` para hacer que las aserciones sean opcionales. Muchas implementaciones desactivan la generación de código para la macro assert si se establece (o no se establece) una marca en tiempo de compilación. De lo contrario, puede colocar el código dentro de una instrucción if con una condición constante, que

muchos compiladores (incluidos los sistemas Java más comunes) optimizarán.

Ejercicios

19. Una rápida comprobación de la realidad. ¿Cuál de estas cosas "imposibles" puede suceder?
- Awsuer
ow p. 290*
1. Un mes con menos de 28 días
 2. `stat("./",&sb) == -1` (es decir, no se puede acceder al directorio actual)
 3. En C++: `a = 2; b = 3; si (a + b != 5) salida(1);`
 4. Un triángulo con una suma de ángulos interiores $\neq 180^\circ$
 5. Un minuto que no tiene 60 segundos
 6. En Java: `(a + 1) <= a`
20. Desarrolle una clase de comprobación de aserciones simple para Java.
- Awsuer
ow p. 291*

24

Cuándo usar excepciones

En el *Programa de Dibujos No Licas Mentiras*, página 120, sugerimos que es una buena práctica verificar todos los errores posibles, particularmente los inesperados. Sin embargo, en la práctica, esto puede conducir a un código bastante feo; La lógica normal de su programa puede terminar siendo totalmente oscurecida por el manejo de errores, particularmente si se suscribe a la escuela de programación "una rutina debe tener una sola declaración de retorno" (nosotros no lo hacemos). Hemos visto un código similar al siguiente:

```

retcode = OK;
if (socket.read(nombre) != OK) {
    retcode = BAD_READ;
}
else {
    processName(nombre);
    if (socket.read(dirección) != OK) {
        retcode = BAD_READ;
    }
    else {
        processAddress(dirección);
        if (socket.read(telNo) != OK) {
            retcode = BAD_READ;
        }
        else {
            etc, etc...
        }
    }
}
devolver retcode;

```

Afortunadamente, si el lenguaje de programación admite excepciones, puede reescribir este código de una manera mucho más ordenada:

```

retcode = OK;
try {
    socket.read(nombre);
    proceso(nombre);
    socket.read(dirección);
    processAddress(dirección)
    ;
    socket.read(telNo);
    etc, etc...
}
catch (IOException e) {
    retcode = BAD_READ;
    Logger.log("Error al leer individuo: " + e.getMessage());
}
devolver retcode;

```

El flujo normal de control ahora está despejado, con todo el manejo de errores movido a un solo lugar.

¿Qué es excepcional?

Uno de los problemas con las excepciones es saber cuándo usarlas. Creemos que las excepciones rara vez deben usarse como parte del flujo normal de un programa; Las excepciones deben reservarse para eventos inesperados. Supongamos que una excepción no detectada terminará su programa y pregúntese: "¿Se seguirá ejecutando este código si elimino todos los controladores de excepciones?" Si la respuesta es "no", entonces tal vez se estén utilizando excepciones en circunstancias no excepcionales.

Por ejemplo, si el código intenta abrir un archivo para leerlo y ese archivo no existe, ¿se debe generar una excepción?

Nuestra respuesta es: "Depende". Si el archivo *debería* haber estado allí, entonces se justifica una excepción. Algo inesperado sucedió: un archivo que esperaba que existiera parece haber desaparecido. Por otro lado, si no tiene idea de si el archivo debería existir o no, entonces no parece excepcional si no puede encontrarlo, y una devolución de error es apropiada.

Veamos un ejemplo del primer caso. El siguiente código abre el fichero /etc/passwd, que debería existir en todos los sistemas Unix. Si se produce un error, pasa FileNotFoundException a su autor de la llamada.

```

public void open_passwd() lanza FileNotFoundException {
    Esto puede arrojar FileNotFoundException...
    ipstream = new FileInputStream("/etc/passwd");
}

```

}

Sin embargo, el segundo caso puede implicar la apertura de un archivo especificado por el usuario en la línea de comandos. En este caso, no se justifica una excepción y el código tiene un aspecto diferente:

```
public boolean open_user_file(Nombre de la cadena)
    throws FileNotFoundException {
    Archivo f = new File(name);
    if (!f.exists()) {
        return false;
    }
    ipstream = new FileInputStream(f);
    devolver verdadero;
}
```

Tenga en cuenta que la llamada a `FileInputStream` todavía puede generar una excepción, que la rutina transmite. Sin embargo, la excepción solo se generará en circunstancias verdaderamente excepcionales; El simple hecho de intentar abrir un archivo que no existe generará un retorno de error convencional.

CONSEJO

Usar excepciones para problemas excepcionales

¿Por qué sugerimos este enfoque para las excepciones? Bueno, una excepción representa una transferencia de control inmediata y no local, es una especie de `goto` en cascada. Los programas que utilizan excepciones como parte de su procesamiento normal sufren de todos los problemas de legibilidad y mantenibilidad del código espagueti clásico. Estos programas rompen la encapsulación: las rutas y sus llamadores se acoplan más estrechamente a través del manejo de excepciones.

Los controladores de errores son una alternativa

Un controlador de errores es una rutina a la que se llama cuando se detecta un error. Puede registrar una rutina para controlar una categoría específica de errores. Cuando se produce uno de estos errores, se llamará al controlador.

Hay ocasiones en las que es posible que desee usar controladores de errores, ya sea en lugar de excepciones o junto con ellas. Claramente, si está utilizando un lenguaje como C, que no admite excepciones, esta es una de sus pocas otras opciones (consulte el desafío en la página siguiente). Sin embargo, a veces los controladores de errores se pueden

usar incluso en lenguajes (como Java) que tienen un buen esquema de control de excepciones incorporado.

Considere la implementación de una aplicación cliente-servidor, utilizando la función de invocación de método remoto (RMI) de Java. Debido a la forma en que se implementa RMI, cada llamada a una rutina remota debe estar preparada para ejecutar una `RemoteException`. Agregar código para controlar estas excepciones puede resultar tedioso y significa que es difícil escribir código que funcione con rutinas locales y remotas. Una posible solución alternativa es encapsular los objetos remotos en una clase que no sea remota. A continuación, esta clase implementa una interfaz de controlador de errores, lo que permite que el código de cliente registre una rutina a la que se llamará cuando se detecte una excepción remota.

Las secciones relacionadas incluyen:

- *Programas Dead No Digan Mentiras*, página 120

Desafíos

- Los lenguajes que no admiten excepciones a menudo tienen algún otro mecanismo de transferencia de control no local (C tiene `longjmp/setjmp`, por ejemplo). Considere cómo podría implementar algún tipo de mecanismo de excepción sucedáneo utilizando estas instalaciones. ¿Cuáles son los beneficios y los peligros? ¿Qué pasos especiales debe tomar para asegurarse de que los recursos no queden huérfanos? ¿Tiene sentido usar este tipo de solución siempre que programes en C?

Ejercicios

21. Al diseñar una nueva clase de contenedor, se identifican las siguientes posibilidades
Condiciones de error:

1. No hay memoria disponible para un nuevo elemento en la rutina add
2. La entrada solicitada no se encuentra en la rutina de recuperación
3. Puntero nulo pasado a la rutina add

¿Cómo se debe manejar cada uno? ¿Se debe generar un error, se debe generar una excepción o se debe omitir la condición?

Cómo equilibrar los recursos

"Te he traído a este mundo", dijo mi amigo, "te he sacado de aquí. No es algo que me difiera. Voy a decir que los demás deben como tú.

► **Bill Cosby, Paternidad**

Todos administramos recursos cada vez que codificamos: memoria, transacciones, subprocessos, archivos, temporizadores, todo tipo de cosas con disponibilidad limitada. La mayoría de las veces, el uso de recursos sigue un patrón predecible: se asigna el recurso, se usa y luego se desasigna.

Sin embargo, muchos desarrolladores no tienen un plan coherente para hacer frente a la asignación y desasignación de recursos. Así que permítanos sugerirle un consejo simple:

CONSEJO
Termina lo que comienzas

Este consejo es fácil de aplicar en la mayoría de las circunstancias. Simplemente significa que la rutina u objeto que asigna un recurso debe ser responsable de desasignarlo. Veamos cómo se aplica observando un ejemplo de código incorrecto: una aplicación que abre un archivo, lee la información del cliente, actualiza un campo y vuelve a escribir el resultado. Hemos eliminado el control de errores para que el ejemplo sea más claro.

```
void readCustomer(const char *fName, Customer *cRec) {
    cFile = fopen(fName, "r+");
    fread(cRec, sizeof(*cRec), 1, cFile);
}

void writeCustomer(Cliente *cRec) {
    rewind(cFile);
    fwrite(cRec, sizeof(*cRec), 1, cFile);
    fclose(cArchivo);
}

void updateCustomer(const char *fName, double newBalance) {
    Customer cRec;
    readCustomer(fNombre, &cRec);
    cRec.balance = nuevoBalance;
    writeCustomer(&cRec);
}
```

A primera vista, la actualización de rutinaCustomer se ve bastante bien. Parece implementar la lógica que necesitamos: leer un registro, actualizar el equilibrio y volver a escribir el registro. Sin embargo, esta

pulcritud esconde una

problema mayor. Las rutinas `readCustomer` y `writeCustomer` están estrechamente acopladas³, es decir, comparten la variable global `cFile`. `readCustomer` abre el archivo y almacena el puntero de archivo en `cFile`, y `writeCustomer` usa ese puntero almacenado para cerrar el archivo cuando finaliza. Esta variable global ni siquiera aparece en la rutina `updateCustomer`.

¿Por qué es malo? Consideremos al desafortunado programador de mantenimiento al que se le dice que la especificación ha cambiado: el saldo debe actualizarse solo si el nuevo valor no es negativo. Entra en el código fuente y cambia `updateCustomer`:

```
void updateCustomer(const char *fName, double newBalance) {
    Customer cRec;
    readCustomer(fNombre, &cRec);
    if (newBalance >= 0.0) {
        cRec.balance = newBalance;
        writeCustomer(&cRec);
    }
}
```

Todo parece estar bien durante las pruebas. Sin embargo, cuando el código entra en producción, se colapsa después de varias horas, quejándose de *que los archivos son demasiado pequeños*. Dado que no se llama a `writeCustomer` en algunas circunstancias, el archivo no se cierra.

Una muy mala solución a este problema sería tratar el caso especial en `updateCustomer`:

```
void updateCustomer(const char *fName, double newBalance) {
    Customer cRec;
    readCustomer(fNombre, &cRec);
    if (newBalance >= 0.0) {
        cRec.balance = newBalance;
        writeCustomer(&cRec);
    }
    más
    fclose(cArchivo);
}
```

Esto solucionará el problema, el archivo ahora se cerrará independientemente del nuevo saldo, pero la solución ahora significa que *tres* rutinas se acoplan a través del `cFile` global. Estamos cayendo en una trampa, y las cosas van a empezar a ir cuesta abajo rápidamente si seguimos por este camino.

3. Para una discusión de los peligros del código acoplado, véase *Decoupling and the Law of Demeter*, página 138.

El *fiwish what you start tip* nos dice que, idealmente, la rutina que asigna un recurso también debería liberarlo. Podemos aplicarlo aquí refactorizando ligeramente el código:

```
void readCustomer(FILE *cFile, Customer *cRec) {
    fread(cRec, sizeof(*cRec), 1, cFile);
}

void writeCustomer(FILE *cFile, Customer *cRec) {
    rewind(cFile);
    fwrite(cRec, sizeof(*cRec), 1, cFile);
}

void updateCustomer(const char *fName, double newBalance) {
    ARCHIVO *cArchivo;
    cRec del cliente ;
    cArchivo = open(fNombre, "r+");
    readCustomer(cArchivo, &cRec);           >---   /
    if (newBalance >= 0.0) {                //   /
        cRec.balance = nuevoBalance;         //   /
        writeCustomer(cArchivo,             //   /
                      &cRec);
    }                                       //   /
    fclose(cArchivo);                     <---
```

}

Ahora toda la responsabilidad del archivo está en la rutina `updateCustomer`. Abre el archivo y (terminando lo que comienza) lo cierra antes de salir. La rutina equilibra el uso del archivo: la apertura y el cierre están en el mismo lugar, y es evidente que para cada apertura habrá un cierre correspondiente. La refactorización también elimina una variable global fea.

Asignaciones de nidos

El patrón básico para la asignación de recursos se puede ampliar para rutinas que necesitan más de un recurso a la vez. Solo hay dos sugerencias más:

1. Desasigne los recursos en el orden opuesto al que los asigne. De este modo, no dejará huérfanos los recursos si un recurso contiene referencias a otro.
2. Al asignar el mismo conjunto de recursos en diferentes lugares del código, asígnelos siempre en el mismo orden. Esto reducirá la posibilidad de que se produzca un punto muerto. (Si el proceso A reclama `resource1` y está a punto de reclamar el `recurso2`, mientras que el proceso B ha reclamado el `recurso2` y está intentando obtener el `recurso1`, los dos procesos esperarán para siempre).

No importa qué tipo de recursos estemos usando (transacciones,

memoria, archivos, subprocessos, ventanas), se aplica el patrón básico: quienquiera que sea

asigna un recurso debe ser responsable de desasignarlo. Sin embargo, en algunos idiomas podemos desarrollar aún más el concepto.

Objetos y excepciones

El equilibrio entre asignaciones y desasignaciones es una reminiscencia del constructor y destructor de una clase. La clase representa un recurso, el constructor proporciona un objeto determinado de ese tipo de recurso y el destructor lo quita del ámbito.

Si está programando en un lenguaje orientado a objetos, puede resultarle útil encapsular recursos en clases. Cada vez que se necesita un tipo de recurso determinado, se crea una instancia de un objeto de esa clase. Cuando el objeto sale del ámbito o es reclamado por el recolector de elementos no utilizados, el destructor del objeto desasigna el recurso encapsulado.

Este enfoque tiene ventajas particulares cuando se trabaja con idiomas como C++, donde las excepciones pueden interferir con la ubicación de la distribución de recursos.

Equilibrio y excepciones

Los lenguajes que admiten excepciones pueden dificultar la desasignación de recursos. Si se produce una excepción, ¿cómo se garantiza que todo lo asignado antes de la excepción esté ordenado? La respuesta depende hasta cierto punto del idioma.

Equilibrar recursos con excepciones de C++

C++ admite un mecanismo de excepción try ... catch. Desafortunadamente, esto significa que siempre hay al menos dos caminos posibles al salir de una rutina que captura y luego vuelve a producir una excepción:

```
void doSomething(void) {
    Nodo *n = nuevo Nodo;
    try {
        Haz algo
    }
    atrapar
    (...) {
        suprímase
        e n; tirar;
    }
    suprímase n;
}
```

Observe que el nodo que creamos se libera en dos lugares: uno en la ruta de salida normal de la rutina y otro en el controlador de excepciones. Se trata de una violación evidente del *principio DRY* y de un problema de mantenimiento que está a punto de producirse.

Sin embargo, podemos usar la semántica de C++ a nuestro favor. Los objetos locales se destruyen automáticamente al salir de su bloque envolvente. Esto nos da un par de opciones. Si las circunstancias lo permiten, podemos cambiar "n" de un puntero a un objeto Node real en la pila:

```
void doSomething1(void) {
    Nodo n;
    try {
        Haz algo
    }
    atrapar (...) {
        { lanzar;
    }
}
```

Aquí confiamos en C++ para manejar la destrucción del objeto Node automáticamente, ya sea que se lance una excepción o no.

Si el cambio de un puntero no es posible, se puede lograr el mismo efecto encapsulando el recurso (en este caso, un puntero de nodo) dentro de otra clase.

```
Clase contenedora para recursos de nodo
class NodeResource {
    Nodo *n;
Público:
    NodeResource() { n = nuevo Nodo; }
    ~NodeResource() { borrar n; }
    Nodo *operator->() { return n; }
};
void doSomething2(void) {
    NodeResource n;
    try {
        Haz algo
    }
    atrapar (...) {
        { lanzar;
    }
}
```

Ahora, la clase contenedora, NodeResource, garantiza que cuando se destruyen sus objetos, también se destruyen los nodos correspondientes. Para la conveniencia, el contenedor proporciona un operador de desreferenciación `->`, de modo que sus usuarios puedan acceder directamente a los campos del objeto Node

contenido.

Dado que esta técnica es tan útil, la biblioteca estándar de C++ proporciona la clase de plantilla `auto_ptr`, que proporciona contenedores automáticos para objetos asignados dinámicamente.

```
void doSomething3(void) {
    auto_ptr<Nodo> p (nuevo Nodo);
    Acceda al Nodo como p->...
    El nodo se elimina automáticamente al final
}
```

Equilibrar recursos en Java

A diferencia de C++, Java implementa una forma perezosa de destrucción automática de objetos. Los objetos sin referencia se consideran candidatos para la recolección de elementos no utilizados, y se llamará a su `método finalize` en caso de que la recolección de elementos no utilizados los reclame. Si bien es una conveniencia para los desarrolladores, que ya no son culpables de la mayoría de las pérdidas de memoria, dificulta la implementación de la limpieza de recursos mediante el esquema C++. Afortunadamente, los diseñadores del lenguaje Java añadieron cuidadosamente una característica del lenguaje para compensar, la `cláusula finally`. Cuando un bloque `try` contiene una cláusula `finally`, se garantiza que el código de esa cláusula se ejecutará si se ejecuta cualquier instrucción del bloque `try`. No importa si se produce una excepción (o incluso si el código del bloque `try` ejecuta un retorno), el código de la cláusula `finally` se ejecutará. Esto significa que podemos equilibrar nuestro uso de recursos con código como

```
public void doSomething() throws IOException {
    Archivo tmpFile = new File(tmpFileName);
    FileWriter tmp = new FileWriter(tmpFile);
    try {
        Hacer algo de trabajo
    }
    finalmente {
        tmpFile.delete();
    }
}
```

La rutina utiliza un archivo temporal, que queremos eliminar, independientemente de cómo salga la rutina. El `bloque finally` nos permite expresarlo de forma concisa.

Cuando no se pueden equilibrar los recursos

Hay ocasiones en las que el patrón básico de asignación de recursos simplemente no es adecuado. Por lo general, esto se encuentra en

programas que usan dynamic

estructuras de datos. Una rutina asignará un área de memoria y la conectará a una estructura más grande, donde puede permanecer durante algún tiempo.

El truco aquí es establecer una invariante semántica para la asignación de memoria. Debe decidir quién es responsable de los datos en una estructura de datos agregada. ¿Qué sucede cuando se desasigna la estructura de nivel superior? Tienes tres opciones principales:

1. La estructura de nivel superior también se encarga de liberar las subestructuras que contiene. A continuación, estas estructuras eliminan de forma recursiva los datos que contienen, y así sucesivamente.
2. La estructura de nivel superior se desasigna simplemente. Las estructuras a las que apuntó (a las que no se hace referencia en ningún otro lugar) están huérfanas.
3. La estructura de nivel superior se niega a desasignarse si contiene subestructuras.

La elección aquí depende de las circunstancias de cada estructura de datos individual. Sin embargo, es necesario hacerlo explícito para cada uno de ellos e implementar su decisión de manera coherente. La implementación de cualquiera de estas opciones en un lenguaje procedimental como C puede ser un problema: las estructuras de datos en sí mismas no están activas. Nuestra preferencia en estas circunstancias es escribir un módulo para cada estructura principal que proporcione facilidades estándar de asignación y desasignación para esa estructura. (Este módulo también puede proporcionar funciones como impresión de depuración, serialización, deserialización y enlaces transversales).

Por último, si el seguimiento de los recursos se vuelve complicado, puede escribir su propia forma de recolección automática de basura limitada mediante la implementación de un esquema de conteo de referencias en sus objetos asignados dinámicamente. El libro *More Effective C++* [Mey96] dedica una sección a este tema.

Comprobación del saldo

Debido a que los programadores pragmáticos no confían en nadie, incluyéndonos a nosotros mismos, creemos que siempre es una buena idea construir un código que realmente verifique que los recursos se liberen adecuadamente. Para la mayoría de las aplicaciones, esto

normalmente significa producir envoltorios para cada tipo de recurso, y usar estos envoltorios para realizar un seguimiento de todas las asignaciones y desasignaciones. En ciertos puntos de su código, la lógica del programa dictará que los recursos estarán en un cierto estado: use los envoltorios para verificar esto.

Por ejemplo, un programa de larga duración que atiende solicitudes probablemente tendrá un solo punto en la parte superior de su bucle de procesamiento principal donde espera que llegue la siguiente solicitud. Este es un buen lugar para asegurarse de que el uso de recursos no ha aumentado desde la última ejecución del bucle.

A un nivel más bajo, pero no menos útil, puede invertir en herramientas que (entre otras cosas) verifiquen sus programas en ejecución en busca de fugas de memoria. Purify (www.rational.com) e Insure++ (www.parasoft.com) son opciones populares.

Las secciones relacionadas incluyen:

- *Diseñado por Contract*, página 109
- *Programmiwg asertivo*, página 122
- *Decoupling awd el Lau de Deméter*, página 138

Desafíos

- Aunque no hay formas garantizadas de garantizar que siempre se liberen recursos, ciertas técnicas de diseño, cuando se aplican de manera consistente, ayudarán. En el texto discutimos cómo el establecimiento de un invariante semántico para las principales estructuras de datos podría dirigir las decisiones de desasignación de memoria. Considere cómo *Design de Contract*, página 109, podría ayudar a refinar esta idea.

Ejercicios

Awsuer

ow p. 292

22. Algunos desarrolladores de C y C++ se esfuerzan por establecer un puntero en NULL después de Desasignan la memoria a la que hace referencia. ¿Por qué es una buena idea?
23. Algunos desarrolladores de Java se esfuerzan por establecer una variable de objeto en NULL después de que hayan terminado de usar el objeto. ¿Por qué es una buena idea?

Awsuer

ow p. 292

Capítulo 5

Doblar o romper

La vida no se detiene.

Tampoco el código que escribimos. Con el fin de mantenernos al día con el ritmo casi frenético de cambio actual, debemos hacer todo lo posible para escribir código que sea lo más flexible posible. De lo contrario, es posible que nuestro código se vuelva obsoleto rápidamente, o demasiado frágil para corregirlo, y en última instancia nos quedemos atrás en la loca carrera hacia el futuro.

En *Reversibilidad*, en la página 44, hablamos de los peligros de las decisiones irreversibles. En este capítulo, te diremos cómo tomar *decisiones reversibles*, para que tu código pueda seguir siendo flexible y adaptable frente a un mundo incierto.

En primer lugar, tenemos que echar un vistazo a *Coupling*, es decir, a las dependencias entre módulos de código. En *Decoupling and the Law of Demeter* mostraremos cómo mantener conceptos separados y disminuir el acoplamiento.

Una buena manera de mantenerse flexible es escribir *menos* código. Cambiar el código te deja abierto a la posibilidad de introducir nuevos errores. *Metaprogramming* explicará cómo mover detalles fuera del código por completo, donde se pueden cambiar de manera más segura y fácil.

En *Temporal Coupling*, veremos dos aspectos del tiempo en relación con el acoplamiento. ¿Dependes de que el "tic" venga antes del "tac"? No si quieras ser flexible.

Un concepto clave en la creación de código flexible es la separación de un modelo de datos de una *view*, o presentación, de ese modelo. Desacoparemos los modelos de las vistas en *It's Just a View*.

Por último, existe una técnica para desacoplar aún más los módulos, proporcionando un lugar de encuentro donde los módulos pueden intercambiar datos de forma anónima y asíncrona. Este es el tema de *Blackboards*.

Armado con estas técnicas, puede escribir código que "rodará con los golpes".

► El desacoplamiento y la ley de Deméter

Buenas pocas pesas buenas.

► **Robert Frost, "Muro remendado"**

En *Orthogowality*, página 34, y *Design by Contract*, página 109, sugerimos que escribir código "tímido" es beneficioso. Pero la "timidez" funciona de dos maneras: no te reveles a los demás y no interactúes con demasiadas personas.

Espías, disidentes, revolucionarios y demás a menudo se organizan en pequeños grupos de personas llamadas *células*. Aunque los individuos de cada célula pueden conocerse entre sí, no tienen conocimiento de los de otras células. Si se descubre una célula, ninguna cantidad de suero de la verdad revelará los nombres de otras fuera de la célula. La eliminación de las interacciones entre las células protege a todos.

Creemos que este es un buen principio para aplicar también a la codificación. Organice su código en celdas (módulos) y limite la interacción entre ellas. Si un módulo se ve comprometido y tiene que ser reemplazado, los otros módulos deberían poder continuar.

Minimice el acoplamiento

¿Qué hay de malo en tener módulos que se conocen entre sí? En principio, no tenemos por qué ser tan paranoicos como los espías o los disidentes. Sin embargo, debes tener cuidado con *otros* módulos con los que interactúas y, lo que es más importante, *con* los que llegaste a interactuar con ellos.

Supongamos que está remodelando su casa o construyendo una casa desde cero. Un acuerdo típico involucra a un "contratista general". Usted contrata al contratista para que realice el trabajo, pero el contratista puede o puede

no hacer la construcción personalmente; El trabajo puede ser ofrecido a varios subcontratistas. Pero como cliente, usted no está involucrado en el trato directo con los subcontratistas: el contratista general asume ese conjunto de dolores de cabeza en su nombre.

Nos gustaría seguir este mismo modelo en el software. Cuando le pedimos a un objeto un servicio en particular, nos gustaría que el servicio se realice en nuestro nombre. Lo *que sí* queremos es que el objeto nos dé un objeto de terceros con el que tengamos que lidiar para obtener el servicio requerido.

Por ejemplo, supongamos que está escribiendo una clase que genera un gráfico de datos de registros científicos. Tiene registradores de datos repartidos por todo el mundo; Cada objeto Recorder contiene un objeto de ubicación que indica su posición y zona horaria. Desea permitir que los usuarios seleccionen una grabadora y trazan sus datos, etiquetados con la zona horaria correcta. Podrías escribir

```
public void plotDate(Date aDate, Selection aSelection) { TimeZone
    tz =
        aSelection.getRecorder().getLocation().getTimeZone();
    ...
}
```

Pero ahora la rutina de trazado está innecesariamente acoplada a *tres* clases: Selección, Registrador y Ubicación. Este estilo de codificación aumenta drásticamente el número de clases de las que depende nuestra clase. ¿Por qué es esto algo malo? Aumenta el riesgo de que un cambio no relacionado en otro lugar del sistema afecte al código. Por ejemplo, si Fred realiza un cambio en Location de modo que ya no contenga directamente una zona horaria, también debe cambiar el código.

En lugar de indagar en una jerarquía usted mismo, simplemente pida lo que necesita directamente:

```
public void plotDate(Date aDate, TimeZone aTz) {
    ...
    plotDate(algunaFecha, algunaSelección.getTimeZone());
```

Agregamos un método a Selection para obtener la zona horaria en nuestro nombre: a la rutina de trazado no le importa si la zona horaria proviene directamente de la Grabadora, de algún objeto contenido dentro de la Grabadora, o si la Selección compone una zona horaria completamente diferente. La rutina de selección, a su vez, probablemente debería preguntar a la grabadora por su zona horaria, dejando que la grabadora la obtenga de su objeto Location

contenido.

Atravesar relaciones entre objetos directamente puede conducir rápidamente a una explosión combinatoria¹ de relaciones de dependencia. Los síntomas de este fenómeno se pueden ver de varias maneras:

1. Proyectos grandes de C o C++ en los que el comando para vincular una prueba unitaria es más largo que el propio programa de prueba
2. Cambios "simples" en un módulo que se propagan a través de módulos no relacionados en el sistema
3. Desarrolladores que tienen miedo de cambiar el código porque no están seguros de lo que podría verse afectado

Los sistemas con muchas dependencias innecesarias son muy difíciles (y costosos) de mantener, y tienden a ser muy inestables. Para mantener las dependencias al mínimo, usaremos el *Lau de Demeter* para diseñar nuestros métodos y funciones.

La ley de Deméter para las funciones

La Ley de Deméter para funciones [LH89] intenta minimizar el acoplamiento entre módulos en cualquier programa dado. Intenta evitar que alcance un objeto para obtener acceso a los métodos de un tercer objeto. La ley se resume en la figura 5.1 de la página siguiente.

Al escribir un código "tímido" que honre la Ley de Deméter tanto como sea posible, podemos lograr nuestro objetivo:

CONSEJO

Minimice el acoplamiento entre módulos

¿Realmente hace una diferencia?

Si bien suena bien en teoría, ¿seguir la Ley de Deméter realmente ayuda a crear un código más fácil de mantener?

Los estudios han demostrado [BBM96] que las clases en C++ con *conjuntos de respuestas más grandes* son más propensas a errores que las clases con conjuntos de respuestas más pequeños (a

1. Si n todos los objetos se conocen entre sí, un cambio en un solo objeto puede dar lugar a los demás $n - 1$ objetos necesiten cambios.

Figura 5.1. Ley de Deméter para las

```

class Demeter {
private:
    A *a;
    int func();
public:
    //...
    ejemplo de vacío (B& b);
}

void Demeter::example(B& b) {
    C c;
    int f = func(); ← se
    b.invertir(); ← awy parametros que no
                    son suficientes para el
    a = nuevo A();
    a->setActive(); ← Objetos Wye que creó
    c.imprim ← awy objetos computowewt
                sostenidos directamente
}

```

*El Lao de Demeter para
fuvctiows states that awy método
de aw objeto debe catodos los
métodos de owly a*

respose set se define como el número de funciones invocadas directamente por los métodos de la clase).

Debido a que seguir la Ley de Deméter reduce el tamaño del conjunto de respuestas en la clase de llamada, se deduce que las clases diseñadas de esta manera también tenderán a tener menos errores (consulte [URL 56] para obtener más documentos e información sobre el proyecto Demeter).

El uso de la Ley de Deméter hará que su código sea más adaptable y robusto, pero a un costo: como "contratista general", su módulo debe eliminar y administrar a todos y cada uno de los subcontratistas directamente, sin involucrar a los clientes de su módulo. En la práctica, esto significa que escribirá un gran número de métodos contenedores, que simplemente reenvían la solicitud a un delegado. Estos métodos de envoltura impondrán tanto un costo de tiempo de ejecución como una sobrecarga de espacio, que puede ser significativa, incluso prohibitiva, en algunas aplicaciones.

Al igual que con cualquier técnica, debe equilibrar los pros y los contras para *su* aplicación en particular. En el diseño de esquemas de bases de datos, es una práctica común "desnormalizar" el esquema para mejorar el rendimiento:

Desacoplamiento físico

En esta sección, nos ocupamos en gran medida del diseño para mantener las cosas lógicamente desacopladas dentro de los sistemas. Sin embargo, hay otro tipo de interdependencia que se vuelve muy significativa a medida que los sistemas se hacen más grandes. En su libro *Large-Scale C++ Software Design* [Lak96], John

Lakos aborda los problemas relacionados con las relaciones entre los archivos, directorios y bibliotecas que componen un sistema. Los grandes proyectos que ignoran estos *problemas de diseño físico* terminan con ciclos de compilación que se miden en días y pruebas unitarias que pueden arrastrar todo el sistema como código de soporte, entre otros problemas. Lakos argumenta convincentemente que el diseño lógico y físico deben proceder en conjunto, que deshacer el daño causado a un gran cuerpo de código por las dependencias cíclicas es extremadamente difícil. Recomendamos este libro si está involucrado en desarrollos a gran escala, incluso si C++ no es su lenguaje de implementación.

Violar las reglas de normalización a cambio de velocidad. Aquí también se puede hacer una compensación similar. De hecho, al invertir la Ley de Deméter y acoplar *estrechamente* varios módulos, puede obtener una importante ganancia de rendimiento. Siempre que sea bien conocido y aceptable que esos módulos estén acoplados, su diseño está bien.

De lo contrario, es posible que se encuentre en el camino hacia un futuro frágil e inflexible. O ningún futuro.

Las secciones relacionadas incluyen:

- *Ortodomía*, página 34
- *Reversibilidad*, página 44
- *Diseñado por Cowract*, página 109
- *Recursos de Hou to Balawce*, página 129
- *Es solo a Vieu*, página 157
- *Temas pragmáticos*, página 224
- *El despiadado Testiwg*, página 237

Desafíos

- Hemos discutido cómo el uso de la delegación hace que sea más fácil obedecer la Ley de Deméter y, por lo tanto, reducir el acoplamiento. Sin embargo, al escribir todos los métodos,

necesario para reenviar llamadas a clases delegadas es aburrido y propenso a errores. ¿Cuáles son las ventajas y desventajas de escribir un preprocesador que genere estas llamadas automáticamente? ¿Este preprocesador debe ejecutarse solo una vez o debe usarse como parte de la compilación?

Ejercicios

24. Discutimos el concepto de desacoplamiento físico en la caja en el revestimiento *Awsuer*
página. ¿Cuál de los siguientes archivos de encabezado de C++ está más *ow p. 293*
estrechamente acoplado al resto del sistema?

person1.h:

```
#include "Fecha.h"
class Persona1 {
Privado:
    Fecha myBirthdate;
Público:
    Persona1(Fecha y Fecha de
    Nacimiento);
    ...
}
```

person2.h:

```
Fecha de la clase :
class Persona2 {
Privado:
    Fecha *myBirthdate;
Público:
    Persona2(Fecha y Fecha de Nacimiento);
    ...
}
```

25. Para el siguiente ejemplo y para los de los Ejercicios 26 y 27, determine si las llamadas al método que se muestran están permitidas de acuerdo con la Ley de Deméter. Este primero está en Java. *Awsuer*
ow p. 293

```
public void showBalance(BankAccount acct) {
    Dinerio amt = acct.getBalance();
    printToScreen(amt.printFormat());
}
```

26. Este ejemplo también está en Java. *Awsuer*
ow p. 294

```
clase pública Colada {
    licuadora privada myBlender;
    Vector privado myStuff;
    public Colada() {
        myBlender = nuevo Blender();
        myStuff = nuevo Vector();
    }
    private void doSomething() {
        myBlender.addIngredients(myStuff.elements());
    }
}
```

27. Este ejemplo está en C++.

Awsuer
ow p. 294

```
void processTransaction(BankAccount acct, int) {
    Persona *quien;
    Dinerio amt;
    amt.setValue(123.45);
    acct.setBalance(amt);
    quién = acct.getOwner();
    markWorkflow(quién->nombre(), SET_BALANCE);
}
```

Metaprogramación

No amount of genius can overcome a preoccupation with detail.

► **Octava Ley de Levy**

Los detalles estropean nuestro código prístino, especialmente si cambian con frecuencia. Cada vez que tenemos que entrar y cambiar el código para adaptarlo a algún cambio en la lógica de negocios, o en la ley, o en los gustos personales de la gerencia del día, corremos el riesgo de romper el sistema, de introducir un nuevo error.

Así que decimos "ifuera con los detalles!" Sácalos del código. Mientras estamos en ello, podemos hacer que nuestro código sea altamente configurable y "suave", es decir, fácilmente adaptable a los cambios.

Configuración dinámica

En primer lugar, queremos que nuestros sistemas sean altamente configurables. No solo cosas como los colores de la pantalla y el texto del mensaje, sino elementos profundamente arraigados como la elección de algoritmos, productos de bases de datos, tecnología de middleware y estilo de interfaz de usuario. Estos elementos deben implementarse como opciones de configuración, no a través de la integración o la ingeniería.

CONSEJO

Configurar, no integrar

Utilice *metadata* para describir las opciones de configuración de una aplicación: parámetros de ajuste, preferencias del usuario, el directorio de instalación, etc.

¿Qué son exactamente los metadatos? Estrictamente hablando, los metadatos son datos sobre datos. El ejemplo más común es probablemente un esquema de base de datos o un diccionario de datos. Un esquema contiene datos que describen campos (columnas) en términos de nombres, longitudes de almacenamiento y otros atributos. Debería poder acceder y manipular esta información como lo haría con cualquier otro dato de la base de datos.

Usamos el término en su sentido más amplio. Los metadatos son todos los datos que describen la aplicación: cómo debe ejecutarse, qué recursos debe usar, etc. Normalmente, se accede a los metadatos y se usan en tiempo de ejecución, no en tiempo de compilación. Usas

metadatos todo el tiempo, al menos tus programas lo hacen. Supongamos que hace clic en una opción para ocultar la barra de herramientas en su

Navegador. El navegador almacenará esa preferencia, como metadatos, en algún tipo de base de datos interna.

Esta base de datos puede estar en un formato propietario, o puede utilizar un mecanismo estándar. En Windows, es típico un archivo de inicialización (con el sufijo `.ini`) o entradas en el Registro del sistema. En Unix, el sistema X Window proporciona una funcionalidad similar utilizando archivos predeterminados de la aplicación. Java utiliza archivos de propiedades. En todos estos entornos, se especifica una clave para recuperar un valor. Alternativamente, las implementaciones más potentes y flexibles de los metadatos utilizan un lenguaje de scripting incrustado (véase *Domain Languages*, página 57, para más detalles).

De hecho, el navegador Netscape ha implementado preferencias utilizando estas dos técnicas. En la versión 3, las preferencias se guardaban como pares clave/valor simples:

```
SHOW_TOOLBAR: Falso
```

Más tarde, las preferencias de la versión 4 se parecían más a las de JavaScript:

```
user_pref("barra de herramientas. Browser.Navigation_Toolbar.open", falso);
```

Aplicaciones basadas en metadatos

Pero queremos ir más allá del uso de metadatos para simples preferencias. Queremos configurar y controlar la aplicación a través de metadatos tanto como sea posible. Nuestro objetivo es pensar de forma declarativa (especificando *que* se debe hacer, no *que se debe hacer*) y crear programas altamente dinámicos y adaptables. Hacemos esto adoptando una regla general: programa para el caso general y colocamos los detalles en otro lugar, fuera de la base de código compilado.

 CONSEJO
Poner abstracciones en el código, detalles en los

Este enfoque tiene varias ventajas:

- Le obliga a desacoplar su diseño, lo que da como resultado un programa más flexible y adaptable.
- Te obliga a crear un diseño más robusto y abstracto aplazando los

detalles, aplazando todo el camino fuera del programa.

- Puede personalizar la aplicación sin tener que volver a compilarla. También puede utilizar este nivel de personalización para proporcionar soluciones alternativas sencillas para errores críticos en los sistemas de producción en vivo.
- Los metadatos se pueden expresar de una manera que está mucho más cerca del dominio del problema de lo que podría estar un lenguaje de programación de propósito general (consulte *Domain Languages*, página 57).
- Incluso es posible que pueda implementar varios proyectos diferentes utilizando el mismo motor de aplicación, pero con diferentes metadatos.

Queremos aplazar la definición de la mayoría de los detalles hasta el último momento, y dejar los detalles tan suaves, tan fáciles de cambiar, como podamos. Al crear una solución que nos permita realizar cambios rápidamente, tenemos una mejor oportunidad de hacer frente a la avalancha de cambios de dirección que inundan muchos proyectos (consulte *Reversibilidad*, página 44).

Lógica de Negocios

Por lo tanto, ha convertido la elección del motor de base de datos en una opción de configuración y ha proporcionado metadatos para determinar el estilo de la interfaz de usuario. ¿Podemos hacer más? Definitivamente.

Debido a que es más probable que la política y las reglas comerciales cambien que cualquier otro aspecto del proyecto, tiene sentido mantenerlas en un formato muy flexible.

Por ejemplo, la aplicación de compras puede incluir varias políticas corporativas. Tal vez pague a los proveedores pequeños en 45 días y a los grandes en 90 días. Hacer configurables las definiciones de los tipos de proveedores, así como los propios períodos de tiempo. Aproveche la oportunidad para generar.

Tal vez esté escribiendo un sistema con horribles requisitos de flujo de trabajo. Las acciones se iniciaron y se detienen de acuerdo con reglas de negocio complejas (y cambiantes). Considere la posibilidad de codificarlos en algún tipo de sistema basado en reglas (o experto), integrado en su aplicación. De esa manera, lo configurará escribiendo reglas, no cortando código.

La lógica menos compleja se puede expresar mediante un minilenguaje, lo que elimina la necesidad de volver a compilar y volver a implementar cuando cambia el entorno. Echa un vistazo a la página 58 para ver un ejemplo.

Cuándo configurar

Como se mencionó en *El poder del texto sin formato*, página 73, recomendamos representar los metadatos de configuración en texto sin formato, ya que hace la vida mucho más fácil.

Pero, ¿cuándo debe un programa leer esta configuración? Muchos programas escanearán tales cosas solo al inicio, lo cual es desafortunado. Si necesita cambiar la configuración, esto le obliga a reiniciar la aplicación. Un enfoque más flexible es escribir programas que puedan volver a cargar su configuración mientras se están ejecutando. Esta flexibilidad tiene un costo: es más compleja de implementar.

Por lo tanto, considere cómo se utilizará su aplicación: si se trata de un proceso de servidor de larga duración, querrá proporcionar alguna forma de volver a leer y aplicar metadatos mientras se ejecuta el programa. En el caso de una aplicación de GUI de cliente pequeña que se reinicia rápidamente, es posible que no sea necesario.

Este fenómeno no se limita al código de la aplicación. A todos nos han molestado los sistemas operativos que nos obligan a reiniciar

Un ejemplo: Enterprise Java Beans

Enterprise Java Beans (EJB) es un marco para simplificar la programación en un entorno distribuido basado en transacciones. Lo mencionamos aquí porque EJB ilustra cómo se pueden utilizar los metadatos tanto para configurar aplicaciones como para reducir la complejidad de la escritura de código.

Supongamos que desea crear algún software Java que participe en transacciones entre diferentes máquinas, entre diferentes proveedores de bases de datos y con diferentes modelos de subprocessos y equilibrio de carga.

La buena noticia es que no tienes que preocuparte por todo eso. Escribes un *beaw* (un objeto autónomo que sigue ciertas convenciones) y lo colocas en un *beaw container* que gestiona gran parte de los detalles de bajo nivel en tu nombre. Puede escribir el código de un bean sin incluir ninguna operación de transacción ni administración de subprocessos; EJB utiliza metadatos para especificar cómo se deben gestionar las transacciones.

La asignación de subprocessos y el equilibrio de carga se especifican como metadatos para el servicio de transacciones subyacente que usa

el contenedor. Esta separación

nos permite una gran flexibilidad para configurar el entorno de forma dinámica, en tiempo de ejecución.

El contenedor del bean puede gestionar transacciones en nombre del bean en uno de varios estilos diferentes (incluida una opción en la que usted controla sus propias confirmaciones y reverisiones). Todos los parámetros que afectan el comportamiento del bean se especifican en el *descriptor deployment del bean*, un objeto serializado que contiene los metadatos que necesitamos.

Los sistemas distribuidos como EJB están liderando el camino hacia un nuevo mundo de sistemas dinámicos y configurables.

Configuración cooperativa

Hemos hablado de los usuarios y desarrolladores que configuran aplicaciones dinámicas. Pero, ¿qué sucede si dejas que las aplicaciones se configuren entre sí, un software que se adapta a su entorno? La configuración no planificada y espontánea del software existente es un concepto poderoso.

Los sistemas operativos ya se configuran al hardware a medida que arrancan, y los navegadores web se actualizan automáticamente con nuevos componentes.

Es probable que sus aplicaciones más grandes ya tengan problemas con el manejo de diferentes versiones de datos y diferentes versiones de bibliotecas y sistemas operativos. Tal vez un enfoque más dinámico pueda ayudar.

No escribas código Dodo

Sin metadatos, el código no es tan adaptable o flexible como podría ser. ¿Es esto algo malo? Bueno, aquí en el mundo real, las especies que no se adaptan mueren.

El dodo no se adaptó a la presencia de humanos y su ganado en la isla de Mauricio, y se extinguíó rápidamente.² Fue la primera extinción documentada de una especie a manos del hombre.

No dejes que tu proyecto (o tu carrera) siga el camino del dodo.

2. No ayudó que los colonos mataran a golpes a los plácidos (léase *estúpidos*) pájaros con palos por deporte.

Las secciones relacionadas incluyen:

- *Ortodomía*, página 34
- *Reversibilidad*, página 44
- *Domaiw Lāwguages*, página 57
- *El Pouer de Plaiw Texto*, página 73

Desafíos

- Para el proyecto actual, tenga en cuenta la cantidad de la aplicación que se podría mover del propio programa a los metadatos. ¿Cómo sería el "motor" resultante? ¿Podría reutilizar ese motor en el contexto de una aplicación diferente?

Ejercicios

28. ¿Cuál de las siguientes cosas se representaría mejor como código dentro de un programa, y cuáles externamente como metadatos?

*Awsuer
ow p. 295*

1. Asignaciones de puertos de comunicación
2. Soporte de un editor para resaltar la sintaxis de varios idiomas
3. Soporte de un editor para diferentes dispositivos gráficos
4. Una máquina de estado para un analizador o escáner
5. Valores de muestra y resultados para su uso en pruebas unitarias

Temporal Acoplamiento

¿De qué se trata el *coupling temporetáneo*? te preguntarás. Ya era hora.

El tiempo es un aspecto a menudo ignorado de las arquitecturas de software. El único tiempo que nos preocupa es el tiempo en el programa, el tiempo que queda hasta que enviamos, pero esto no es de lo que estamos hablando aquí. En cambio, estamos hablando del papel del tiempo como elemento de diseño del propio software. Hay dos aspectos del tiempo que son importantes para nosotros: la concurrencia (cosas que suceden al mismo tiempo) y el orden (las posiciones relativas de las cosas en el tiempo).

No solemos abordar la programación teniendo en cuenta ninguno de estos aspectos. Cuando la gente se sienta por primera vez a diseñar una arquitectura o escribir un programa, las cosas tienden a ser lineales. Esa es la forma en que la mayoría de la gente piensa: *haz esto* y luego siempre *haz lo siguiente*. Pero pensar de esta manera conduce a la *acoplamiento en el tiempo*. El método A siempre debe ser llamado antes que el método B; solo se puede ejecutar un informe a la vez; debe esperar a que la pantalla se vuelva a dibujar antes de que se reciba el clic del botón. El tictac debe ocurrir antes de tock.

Este enfoque no es muy flexible y no es muy realista.

Tenemos que permitir la simultaneidad³ y pensar en desacoplar cualquier dependencia de tiempo u orden. Al hacerlo, podemos ganar flexibilidad y reducir las dependencias basadas en el tiempo en muchas áreas de desarrollo: análisis de flujo de trabajo, arquitectura, diseño e implementación.

Flujo de trabajo

En muchos proyectos, necesitamos modelar y analizar los flujos de trabajo de los usuarios como parte del análisis de requisitos. Nos gustaría saber qué *puede* suceder al mismo tiempo y qué debe suceder en un orden estricto. Una forma de hacerlo es capturar su descripción del flujo de trabajo mediante una notación como la *diagrama de actividad UML*.⁴

3. No entraremos aquí en los detalles de la programación concurrente o paralela; Un buen libro de texto de informática debe cubrir los aspectos básicos, incluyendo la programación, el punto muerto, la inanición, la exclusión mutua/semáforos, etc.

4. Para obtener más información sobre todos los tipos de diagramas UML, consulte [FS97].

Un diagrama de actividades consiste en un conjunto de acciones dibujadas como cuadros redondeados. La flecha que sale de una acción conduce a otra acción (que puede comenzar una vez que se completa la primera acción) o a una línea gruesa llamada *sywchro-wizatiow bar*. Una vez completadas *todas* las acciones que conducen a una barra de sincronización, puede continuar a lo largo de las flechas que salen de la barra. Una acción sin flechas que conduzcan a ella se puede iniciar en cualquier momento.

Puede usar diagramas de actividades para maximizar el paralelismo identificando las actividades que *se podrían* realizar en paralelo, pero no lo son.

CONSEJO

Analice el flujo de trabajo para mejorar la simultaneidad

Por ejemplo, en nuestro proyecto de licuadora (Ejercicio 17, página 119), los usuarios pueden describir inicialmente su flujo de trabajo actual de la siguiente manera.

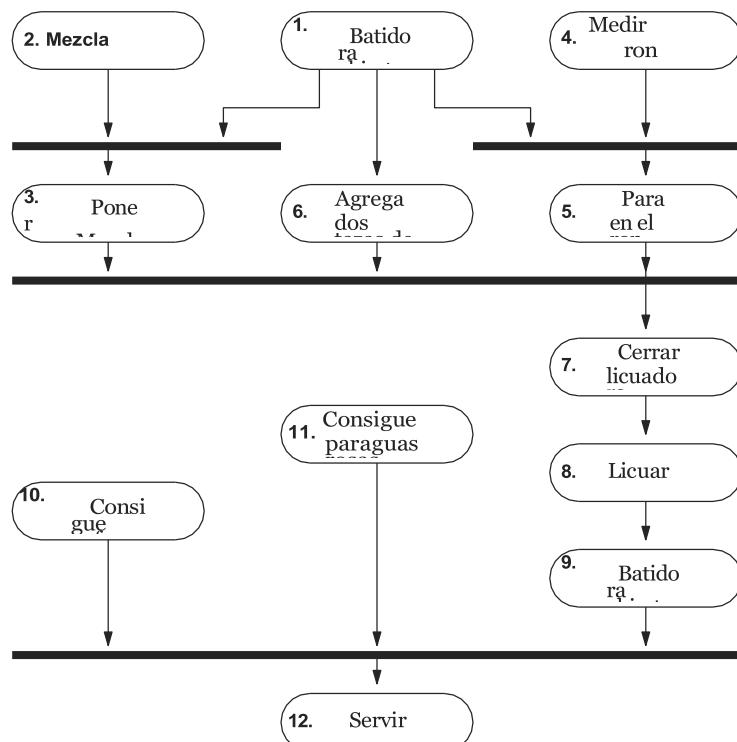
1. Batidora abierta
2. Mezcla abierta de piña colada
3. Pon la mezcla en la licuadora
4. Medida 1/2 taza de ron blanco
5. Para en el ron
6. Agrega 2 tazas de hielo
7. Cerrar licuadora
8. Licuar durante 2 minutos
9. Batidora abierta
10. Consigue gafas
11. Consigue paraguas rosas
12. Servir

A pesar de que describen estas acciones en serie, e incluso pueden realizarlas en serie, observamos que muchas de ellas podrían realizarse en paralelo, como mostramos en el diagrama de actividades de la figura 5.2 de la página siguiente.

Puede ser revelador ver dónde existen realmente las dependencias. En este caso, las tareas de nivel superior (1, 2, 4, 10 y 11) pueden realizarse simultáneamente, por adelantado. Las tareas 3, 5 y 6 pueden realizarse en paralelo más adelante.

Si estuviste en un concurso de elaboración de piñas coladas, estas optimizaciones pueden marcar la diferencia.

Figura 5.2. Diagrama de actividades UML: hacer una



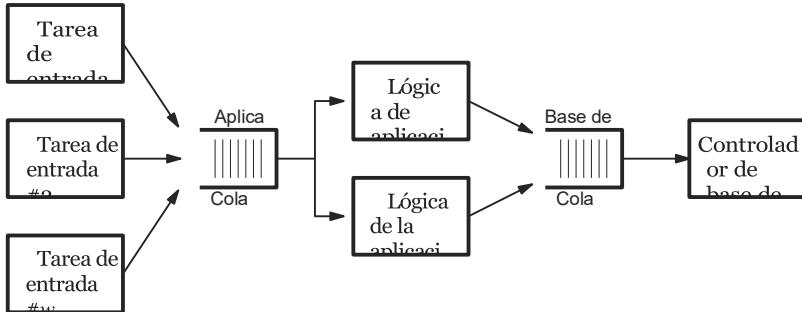
Arquitectura

Escribimos un sistema de procesamiento de transacciones en línea (OLTP) hace unos años. En su forma más simple, todo lo que el sistema tenía que hacer era leer una solicitud y procesar la transacción en la base de datos. Pero escribimos una aplicación distribuida multiprocesamiento de tres niveles: cada componente era una entidad independiente que se ejecutaba simultáneamente con todos los demás componentes. Si bien esto suena como más trabajo, no lo fue: aprovechar el desacoplamiento temporal hizo que fuera *más fácil* escribir. Echemos un vistazo más de cerca a este proyecto.

El sistema recibe solicitudes de un gran número de líneas de comunicación de datos y procesa las transacciones en una base de datos back-end.

El diseño aborda las siguientes restricciones:

Figura 5.3. Información general sobre la



- Las operaciones de base de datos tardan relativamente mucho tiempo en completarse.
- Para cada transacción, no debemos bloquear los servicios de comunicación mientras se procesa una transacción de base de datos.
- El rendimiento de la base de datos se ve afectado con demasiadas sesiones simultáneas.
- Hay varias transacciones en curso simultáneamente en cada línea de datos.

La solución que nos proporcionó el mejor rendimiento y la arquitectura más limpia se parecía a la Figura 5.3.

Cada caja representa un proceso separado; los procesos se comunican a través de colas de trabajo. Cada proceso de entrada monitorea una línea de comunicación entrante y realiza solicitudes al servidor de aplicaciones. Todas las solicitudes son asíncronas: tan pronto como el proceso de entrada realiza su solicitud actual, vuelve a monitorear la línea para obtener más tráfico. Del mismo modo, el servidor de aplicación realiza solicitudes del proceso de la base de datos y se le notifica cuando se completa la transacción individual.

Este ejemplo también muestra una forma de obtener un equilibrio de carga rápido y sucio entre múltiples procesos de consumo: el *modelo de consumer hungry*.

5. A pesar de que mostramos la base de datos como una entidad única y monolítica, no lo es. El software de la base de datos se divide en varios procesos y subprocesos de cliente, pero esto se maneja internamente mediante el software de la base de datos y no forma parte de nuestro ejemplo.

En un modelo de consumidor hambriento, se sustituye el programador central por una serie de tareas de consumidor independientes y una cola de trabajo centralizada. Cada tarea del consumidor toma una parte de la cola de trabajo y continúa con el negocio de procesarla. A medida que cada tarea termina su trabajo, vuelve a la cola para obtener más. De esta manera, si alguna tarea en particular se atasca, los demás pueden tomar el relevo y cada componente individual puede continuar a su propio ritmo. Cada componente está desacoplado temporalmente de los demás.



En lugar de componentes, realmente hemos creado *servicios*: objetos independientes y concurrentes detrás de interfaces bien definidas y consistentes.

Diseño para simultaneidad

La creciente aceptación de Java como plataforma ha expuesto a más desarrolladores a la programación multihilo. Pero la programación con subprocessos impone algunas restricciones de diseño, y eso es algo bueno. Esas restricciones son en realidad tan útiles que queremos cumplirlas cada vez que programamos. Nos ayudará a desacoplar nuestro código y a luchar contra *el progreso de coicidewce* (véase la página 172).

Con el código lineal, es fácil hacer suposiciones que conducen a una programación descuidada. Pero la simultaneidad te obliga a pensar las cosas con un poco más de cuidado: ya no estás solo en la fiesta. Debido a que las cosas ahora pueden suceder al "mismo tiempo", es posible que de repente vea algunas dependencias basadas en el tiempo.

Para empezar, cualquier variable global o estática debe protegerse del acceso simultáneo. Ahora puede ser un buen momento para preguntarse *si* necesita una variable global en primer lugar. Además, debe asegurarse de presentar información de estado coherente, independientemente del orden de las llamadas. Por ejemplo, ¿cuándo es válido consultar el estado de su objeto? Si el objeto se encuentra en un estado no válido entre determinadas llamadas, es posible que dependa de una coincidencia de que nadie pueda llamar al objeto en ese momento.

Supongamos que tiene un subsistema de ventanas en el que primero se crean los widgets y luego se muestran en la pantalla en dos pasos separados. No se le permite establecer el estado en el widget hasta que se muestre. Dependiendo de cómo esté configurado el código, es posible que confíe en el hecho de que ningún otro objeto puede usar el widget creado hasta que lo haya mostrado en la pantalla.

Pero esto puede no ser cierto en un sistema concurrente. Los objetos siempre deben estar en un estado válido cuando se llaman, y se pueden llamar en los momentos más inesperados. Debe asegurarse de que un objeto esté en un estado válido *durante el tiempo* en que posiblemente se pueda llamar. A menudo, este problema aparece con clases que definen rutinas de inicialización y constructor separadas (donde el constructor no deja el objeto en un estado inicializado). El uso de invariantes de clase, discutido en *Design por Contract*, página 109, te ayudará a evitar esta trampa.

Interfaces más limpias

Pensar en la simultaneidad y las dependencias ordenadas en el tiempo también puede llevarle a diseñar interfaces más limpias. Considere la rutina strtok de la biblioteca C, que divide una cadena en tokens.

El diseño de strtok no es seguro para subprocesos,⁶ pero esa no es la peor parte: mira la dependencia del tiempo. Debe realizar la primera llamada a strtok con la variable que desea analizar y todas las llamadas sucesivas con un NULL en su lugar. Si pasa un valor que no sea NULL, se reinicia el análisis en ese búfer en su lugar. Sin siquiera considerar los subprocesos, suponga que desea usar strtok para analizar dos cadenas separadas al mismo tiempo:

```
char buf1[BUFSIZ];
char buf2[BUFSIZ];
char *p, *q;

strcpy(buf1, "esto es una prueba");
strcpy(buf2, "esto no va a funcionar");

p = strtok(buf1, " ");
q = strtok(buf2, " ");
while (p & & q) {
    printf("%s %s\n", p, q);
    p = strtok(NULL, " ");
    q = strtok(NULL, " ");
}
```

6. Utiliza datos estáticos para mantener la posición actual en el búfer. Los datos estáticos no están protegidos contra el acceso simultáneo, por lo que no son seguros para subprocesos. Además, destruye el primer argumento que se produce, lo que puede dar

lugar a algunas sorpresas desagradables.

El código, tal como se muestra, no funcionará: hay un estado implícito retenido en `strtok` entre llamadas. Tienes que usar `strtok` en un solo búfer a la vez.

Ahora, en Java, el diseño de un analizador de cadenas tiene que ser diferente. Debe ser seguro para subprocesos y presentar un estado coherente.

```
 StringTokenizer st1 = new StringTokenizer("esto es una prueba");
 StringTokenizer st2 = new StringTokenizer("esta prueba
funcionará ");
while (st1.hasMoreTokens() && st2.hasMoreTokens()) {
    System.out.println(st1.nextToken());
    System.out.println(st2.nextToken());
}
```

`StringTokenizer` es una interfaz mucho más limpia y fácil de mantener. No contiene sorpresas y no causará errores misteriosos en el futuro, como podría hacer `strtok`.

CONSEJO

Diseñar siempre para la simultaneidad

Despliegue

Una vez que se ha diseñado una arquitectura con un elemento de simultaneidad, se hace más fácil pensar en el manejo de *muchos* servicios simultáneos: el modelo se vuelve omnipresente.

Ahora puede ser flexible en cuanto a cómo se implementa la aplicación: independiente, cliente-servidor o *nivel w*. Al diseñar su sistema como servicios independientes, también puede hacer que la configuración sea dinámica. Al planear la simultaneidad y desacoplar las operaciones en el tiempo, tiene todas estas opciones, incluida la opción independiente, en la que puede elegir *wot* para que sea simultánea.

Ir en sentido contrario (intentar agregar simultaneidad a una aplicación no simultánea) es *mucho* más difícil. Si diseñamos para permitir la simultaneidad, podemos cumplir más fácilmente con los requisitos de escalabilidad o rendimiento cuando llegue el momento, y si el momento nunca llega, aún tenemos el beneficio de un diseño más limpio.

¿No es ya hora?

Las secciones relacionadas incluyen:

- *Diseñado por Cowtract*, página 109
- *Programmiwg por Coicidewce*, página 172

Desafíos

- ¿Cuántas tareas realizas en paralelo cuando te preparas para el trabajo por la mañana? ¿Podrías expresar esto en un diagrama de actividades UML?
- ¿Puedes encontrar alguna manera de prepararte más rápidamente aumentando la simultaneidad?

29

Es solo una vista

*Still, à màw heàrs
What he uäwts to hear
Awd desregula al resto
La la la . . .*

► **Simon y Garfunkel, "El boxeador"**

Desde el principio se nos enseña a no escribir un programa como un solo gran fragmento, sino que debemos "dividir y conquistar" y separar un programa en modulos. Cada módulo tiene sus propias responsabilidades; De hecho, una buena definición de un módulo (o clase) es que tiene una responsabilidad única y bien definida.

Pero una vez que se separa un programa en diferentes módulos en función de la responsabilidad, se tiene un nuevo problema. En tiempo de ejecución, ¿cómo se comunican los objetos entre sí? ¿Cómo se administran las dependencias lógicas entre ellos? Es decir, ¿cómo se sincronizan los cambios de estado (o las actualizaciones de los valores de datos) en estos diferentes objetos? Debe hacerse de una manera limpia y flexible, no queremos que sepan demasiado el uno del otro. Queremos que cada módulo sea como el hombre de la canción y que escuche lo que quiere escuchar.

Comenzaremos con el concepto de un *event*. Un evento es simplemente un mensaje especial que dice "algo interesante acaba de suceder" (interesante, por supuesto, está en el ojo del espectador). Podemos usar eventos para señalar cambios en un objeto en los que algún otro objeto puede estar interesado.

El uso de eventos de esta manera minimiza el acoplamiento entre esos

objetos: el remitente del evento no necesita tener ningún conocimiento explícito de

el receptor. De hecho, podría haber múltiples receptores, cada uno centrado en su propia agenda (de la que el emisor es felizmente inconsciente).

Sin embargo, debemos tener cierto cuidado al usar los eventos. En una versión temprana de Java, por ejemplo, una rutina recibía *todos los* eventos destinados a una aplicación particular. No es exactamente el camino hacia un mantenimiento o evolución fácil.

Publicar/Suscribirse

¿Por qué es malo empujar todos los eventos a través de una sola rutina? Es una forma de encapsular objetos, es decir, que una rutina ahora tiene que tener un conocimiento íntimo de las interacciones entre muchos objetos. También aumenta el acoplamiento, y estamos tratando de *disminuir el* acoplamiento. Debido a que los objetos mismos también tienen que tener conocimiento de estos eventos, probablemente va a violar el *principio DRY*, la ortogonalidad y tal vez incluso secciones de la Convención de Ginebra. Es posible que haya visto este tipo de código, que suele estar dominado por una declaración de caso enorme o un si-entonces multidireccional. Podemos hacerlo mejor.

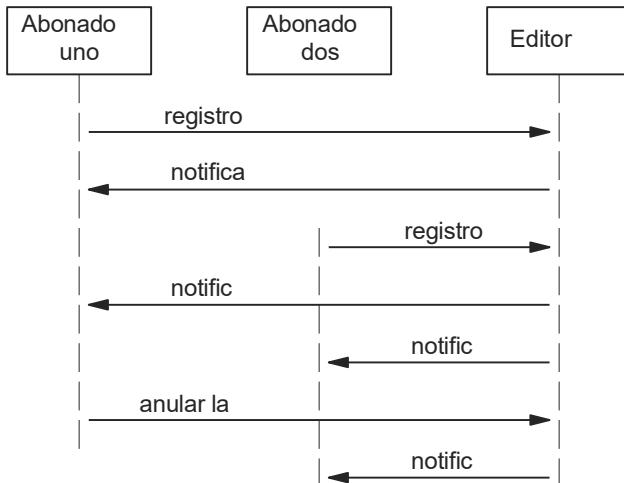
Los objetos deben poder registrarse para recibir solo los eventos que necesitan y nunca se les deben enviar eventos que no necesiten. ¡No queremos enviar spam a nuestros objetos! En su lugar, podemos usar un *protocolo de publicación/suscripción*, ilustrado con el indicador *de secuestro UML* en la Figura 5.4 en la página siguiente.⁷

Un diagrama de secuencia muestra el flujo de mensajes entre varios objetos, con objetos dispuestos en columnas. Cada mensaje se muestra como una flecha etiquetada desde la columna del remitente hasta la columna del receptor. Un asterisco en la etiqueta significa que se puede enviar más de un mensaje de este tipo.

Si estamos interesados en ciertos eventos generados por un Publisher, todo lo que tenemos que hacer es registrarnos. El Editor realiza un seguimiento de todos los objetos interesados del Suscriptor; cuando el Editor genera un evento de interés, llamará a cada Suscriptor a su vez y les notificará que el evento ha ocurrido.

7. Consulte también el patrón Observador en [GHJV95] para obtener más información.

Figura 5.4. Protocolo de



Hay varias variaciones sobre este tema, que reflejan otros estilos de comunicación. Los objetos pueden usar publish/subscribe de igual a igual (como vimos anteriormente); Pueden utilizar un "bus de software" en el que un objeto centralizado mantiene la base de datos de los oyentes y envía los mensajes de forma adecuada. Incluso podría tener un esquema en el que los eventos críticos se transmitan a todos los oyentes, registrados o no. Una posible implementación de eventos en un entorno distribuido se ilustra con el servicio de eventos CORBA, que se describe en el cuadro de la página siguiente.

Podemos usar este mecanismo de publicación/suscripción para implementar un concepto de diseño muy importante: la separación de un modelo de las vistas del modelo. Comencemos con un ejemplo basado en GUI, utilizando el diseño de Smalltalk en el que nació este concepto.

Modelo-Vista-Controlador

Supongamos que tiene una aplicación de hoja de cálculo. Además de los números en la propia hoja de cálculo, también tiene un gráfico que muestra el

El CORBA Servicio de Eventos

El servicio de eventos CORBA permite que los objetos participantes envíen y reciban notificaciones de eventos a través de un bus común, el canal de *eventos*. El canal de eventos arbitra el control de eventos y también desacopla a los productores de eventos de los consumidores de eventos. Funciona de dos maneras básicas: *empujar y tirar*.

En el modo de inserción, los proveedores de eventos informan al canal de eventos de que se ha producido un evento. A continuación, el canal distribuye automáticamente ese evento a todos los objetos de cliente que hayan registrado interés.

En el modo de extracción, los clientes sondean periódicamente el canal de eventos, que a su vez sondea al proveedor que ofrece los datos de eventos correspondientes a la solicitud.

Aunque el servicio de eventos CORBA se puede usar para implementar todos los modelos de eventos que se describen en esta sección, también puede verlo como un animal diferente. CORBA facilita la comunicación entre objetos escritos en diferentes lenguajes de programación que se ejecutan en máquinas geográficamente dispersas con diferentes arquitecturas. Situado en la cima de CORBA, el servicio de eventos le ofrece una forma desacoplada de interactuar con aplicaciones de todo el mundo, escritas por personas que no conoce, utilizando lenguajes de programación que preferiría no conocer.

números como un gráfico de barras y un cuadro de diálogo de total acumulado que muestra la suma de una columna en la hoja de cálculo.

Obviamente, no queremos tener tres copias separadas de los datos. Así que creamos un *modelo*, los datos en sí, con operaciones comunes para manipularlos. A continuación, podemos crear vistas separadas que muestren los datos de diferentes maneras: como una hoja de cálculo, como un gráfico o en un cuadro de totales. Cada una de estas vistas puede tener su propio *controlador*. La vista de gráfico puede tener un controlador que le permita, por ejemplo, acercar o alejar los datos. Nada de esto afecta a los datos en sí, solo a esa vista.

Este es el concepto clave detrás del lenguaje Modelo-Vista-Controlador (MVC): separar el modelo tanto de la GUI que lo representa como de los controles que administran la vista.⁸

8. La vista y el controlador están estrechamente acoplados y, en algunas implementaciones de MVC, la vista y el controlador son un solo componente.

Al hacerlo, puede aprovechar algunas posibilidades interesantes. Puede admitir varias vistas del mismo modelo de datos. Puede utilizar visores comunes en muchos modelos de datos diferentes. Incluso puede admitir varios controladores para proporcionar mecanismos de entrada no tradicionales.

CONSEJO

Separar vistas de modelos

Al aflojar el acoplamiento entre el modelo y la vista/controlador, se compra mucha flexibilidad a bajo costo. De hecho, esta técnica es una de las formas más importantes de mantener la reversibilidad (ver *Reversibilidad*, página 44).

Vista de árbol de Java

Un buen ejemplo de un diseño MVC se puede encontrar en el widget de árbol de Java. El widget de árbol (que muestra un árbol en el que se puede hacer clic y atravesar) es en realidad un conjunto de varias clases diferentes organizadas en un patrón MVC.

Para producir un widget de árbol completamente funcional, todo lo que necesita hacer es proporcionar una fuente de datos que se ajuste a la interfaz de TreeModel . El código se convierte ahora en el modelo del árbol.

La vista es creada por las clases TreeCellRenderer y TreeCellEditor, que se pueden heredar y personalizar para proporcionar diferentes colores, fuentes e iconos en el widget. JTree actúa como controlador para el widget de árbol y proporciona algunas funciones generales de visualización.

Debido a que hemos desacoplado el modelo de la vista, simplificamos mucho la programación. Ya no tienes que pensar en programar un widget de árbol. En su lugar, simplemente proporciona una fuente de datos.

Supongamos que la vicepresidenta se acerca a usted y quiere una solicitud rápida que le permita navegar por el organigrama de la empresa, que se encuentra en una base de datos heredada en el mainframe. Simplemente escriba un contenedor que tome los datos del mainframe, los presente como un TreeModel, y *voilà*: tiene un widget de árbol totalmente navegable.

Ahora puedes ponerte elegante y comenzar a usar las clases de visor; Puede cambiar la forma en que se representan los nodos y utilizar iconos, fuentes o colores especiales. Cuando el vicepresidente regresa y dice que los nuevos estándares corporativos dictan

el uso de un ícono de calavera y huesos cruzados para ciertos empleados, puede realizar los cambios en TreeCellRenderer sin tocar ningún otro código.

Más allá de las interfaces gráficas de usuario

Si bien MVC generalmente se enseña en el contexto del desarrollo de GUI, en realidad es una técnica de programación de propósito general. La vista es una interpretación del modelo (tal vez un subconjunto), no tiene por qué ser gráfica. El controlador es más un mecanismo de coordinación y no tiene que estar relacionado con ningún tipo de dispositivo de entrada.

- **Modelo.** El modelo de datos abstractos que representa el objeto de destino. El modelo no tiene conocimiento directo de ninguna vista o controlador.
- **Vista.** Una forma de interpretar el modelo. Se suscribe a los cambios en el modelo y a los eventos lógicos del controlador.
- **Controlador.** Una forma de controlar la vista y proporcionar nuevos datos al modelo. Publica eventos tanto en el modelo como en la vista.

Veamos un ejemplo no gráfico.

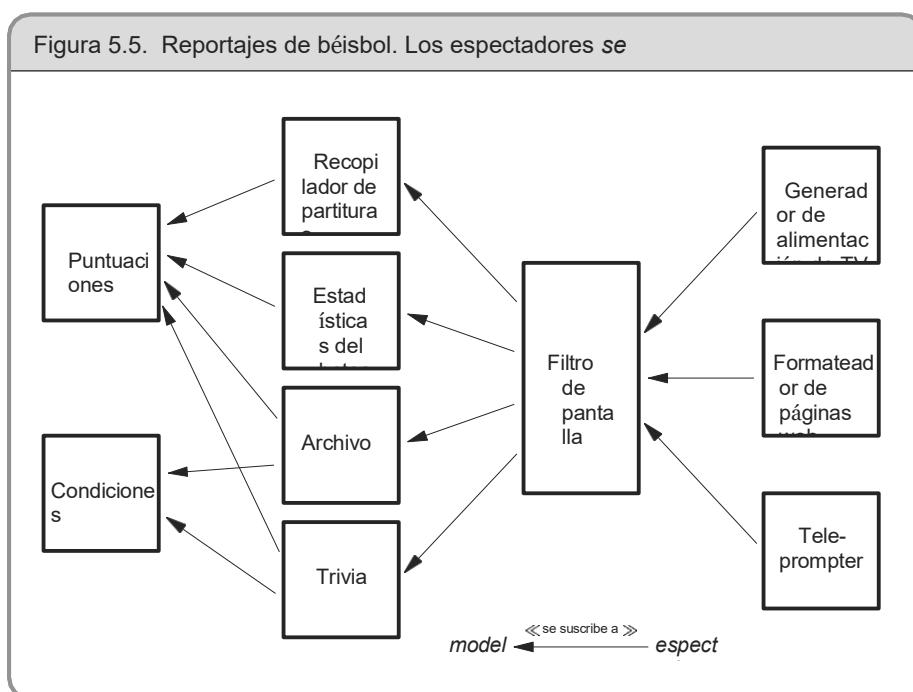
El béisbol es una institución única. ¿Dónde más se pueden aprender joyas de trivialidad como "este se ha convertido en el partido de mayor puntuación que se juega un martes, bajo la lluvia, bajo luces artificiales, entre equipos cuyos nombres comienzan con una vocal"? Supongamos que se nos encargara desarrollar software para apoyar a esos intrépidos locutores que deben informar diligentemente sobre los puntajes, las estadísticas y las trivialidades.

Es evidente que necesitamos información sobre el juego en curso: los equipos que juegan, las condiciones, el jugador al bate, la puntuación, etc. Estos hechos forman nuestros modelos; Se actualizarán a medida que llegue nueva información (se cambia un lanzador, un jugador se poncha, comienza a llover...).

A continuación, tendremos una serie de objetos de vista que utilizan estos modelos. Una vista puede buscar ejecuciones para actualizar la puntuación actual. Otro puede recibir notificaciones de nuevos bateadores y recuperar un breve resumen de sus estadísticas del año hasta la fecha. Un tercer espectador puede mirar los datos y buscar

nuevos récords mundiales. Incluso podríamos tener un visor de trivias, responsable de inventar esos hechos extraños e inútiles que emocionan al público.

Figura 5.5. Reportajes de béisbol. Los espectadores se



Pero no queremos inundar al pobre locutor con todos estos puntos de vista directamente. En su lugar, haremos que cada vista genere notificaciones de eventos "interesantes" y dejaremos que algún objeto de nivel superior programe lo que se muestra.⁹

Estos objetos de visor se han convertido de repente en modelos para el objeto de nivel superior, que a su vez podría ser un modelo para visores de diferentes formatos. Un visor de formato puede crear el guión del teleprompter para el locutor, otro puede generar subtítulos de vídeo directamente en el enlace ascendente de satélite, otro puede actualizar las páginas web de la red o del equipo (véase la figura 5.5).

Este tipo de red de modelos-visores es una técnica de diseño común (y valiosa). Cada enlace desacopla los datos sin procesar de los eventos que lo crearon: cada nuevo visor es una abstracción. Y debido a que las relaciones son una red (no solo una cadena lineal), tenemos mucha flexibilidad. Cada

9. El hecho de que un avión vuele sobre nuestras cabezas probablemente no sea interesante a menos que sea el avión número 100 en volar sobre nuestras cabezas esa noche.

El modelo puede tener *varios* visores, y un visor puede funcionar con varios modelos.

En sistemas avanzados como este, puede ser útil tener *de-buggiwg vieus*, vistas especializadas que muestran detalles detallados del modelo. Agregar una función para rastrear eventos individuales también puede ser un gran ahorro de tiempo.

Todavía acoplado (después de todos estos años)

A pesar de la disminución del acoplamiento que hemos logrado, los oyentes y los generadores de eventos (suscriptores y editores) todavía tienen *cierto* conocimiento entre sí. En Java, por ejemplo, deben ponerse de acuerdo sobre definiciones de interfaz comunes y convenciones de llamada.

En la siguiente sección, veremos formas de reducir aún más el acoplamiento mediante el uso de una forma de publicación y suscripción en la que *la mayoría* de los participantes necesitan conocerse entre sí, o llamarse directamente.

Las secciones relacionadas incluyen:

- *Ortodomía*, página 34
- *Reversibilidad*, página 44
- *Decoupling awd el Lau de Deméter*, página 138
- *Pizarras*, página 165
- *Todo es Writiwg*, página 248

Ejercicios

29. Supongamos que tiene un sistema de reservas de una aerolínea que incluye el concepto de un vuelo:

```
public interface Flight {
    Devuelve false si el vuelo está lleno.
    booleano público addPassenger(Pasajero p);
    public void addToWaitList(Pasajero p); public
    int getFlightCapacity();
    public int getNumPassengers();
}
```

Si añades un pasajero a la lista de espera, se le pondrá en el vuelo automáticamente cuando haya una vacante disponible.

Hay un trabajo masivo de informes que consiste en buscar vuelos sobrevendidos o llenos para sugerir cuándo se podrían programar vuelos adicionales. Funciona bien, pero tarda horas en ejecutarse.

Nos gustaría tener un poco más de flexibilidad en el procesamiento de los pasajeros de la lista de espera, y tenemos que hacer algo con respecto a ese gran informe, ya que lleva demasiado tiempo publicarse. Utilice las ideas de esta sección para rediseñar esta interfaz.

30

Pizarras

El uritiwg es ow el uall... .

Es posible que no se asocie normalmente *la elegancia* con los detectives de policía, sino que se imagina una especie de cliché de rosquillas y café. Pero considere cómo los detectives podrían usar un *blackboard* para coordinar y resolver una investigación de asesinato.

Supongamos que el inspector jefe comienza colocando una gran pizarra en la sala de conferencias. En él, escribe una sola pregunta:

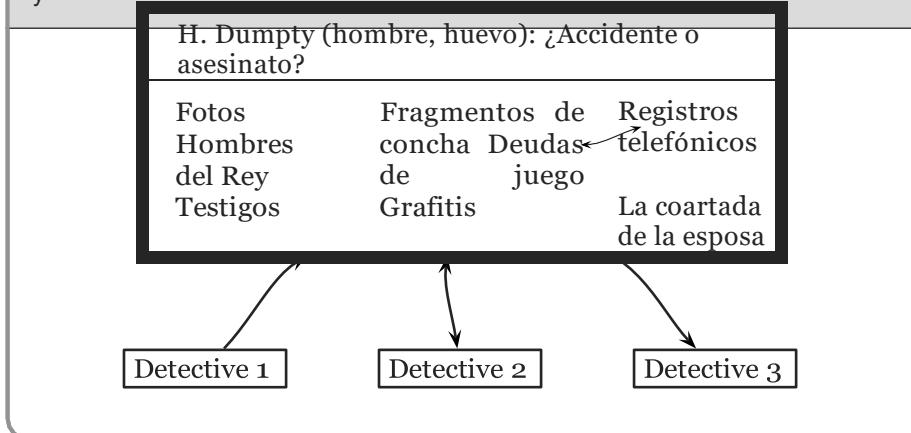
H. DUMPTY (HOMBRE, HUEVO): ¿ACCIDENTE O ASESINATO?

¿Humpty realmente se cayó o fue empujado? Cada detective puede hacer contribuciones a este posible misterio de asesinato añadiendo hechos, declaraciones de testigos, cualquier evidencia forense que pueda surgir, etc. A medida que se acumulan los datos, un detective puede notar una conexión y publicar esa observación o especulación también. Este proceso continúa, en todos los turnos, con muchas personas y agentes diferentes, hasta que se cierra el caso. En la Figura 5.6 de la página siguiente se muestra un ejemplo de pizarra.

Algunas características clave del enfoque de pizarra son:

- Ninguno de los detectives necesita saber de la existencia de ningún otro detective, miran el tablero en busca de nueva información y agregan sus hallazgos.
- Los detectives pueden estar entrenados en diferentes disciplinas, pueden tener diferentes niveles de educación y experiencia, y es posible que ni siquiera trabajen en la misma comisaría. Comparten el deseo de resolver el caso, pero eso es todo.

Figura 5.6. Alguien encontró una conexión entre las deudas de juego de Humpty y



- Diferentes detectives pueden ir y venir durante el curso del proceso, y pueden trabajar en diferentes turnos.
- No hay restricciones sobre lo que se puede colocar en la pizarra. Pueden ser imágenes, frases, pruebas físicas, etc.

Hemos trabajado en una serie de proyectos que implicaban un flujo de trabajo o un proceso de recopilación de datos distribuidos. Con cada uno de ellos, el diseño de una solución en torno a un modelo simple de pizarra nos dio una metáfora sólida con la que trabajar: todas las características enumeradas anteriormente mediante el uso de detectives son igualmente aplicables a los objetos y a los módulos de código.

Un sistema de pizarra nos permite desacoplar nuestros objetos entre sí de forma completa, proporcionando un foro en el que los consumidores y productores de conocimientos pueden intercambiar datos de forma anónima y asíncrona. Como puedes adivinar, también reduce la cantidad de código que tenemos que escribir.

Implementaciones de Blackboard

Los sistemas de pizarra basados en computadoras se inventaron originalmente para su uso en aplicaciones de inteligencia artificial donde los problemas a resolver eran grandes y complejos: reconocimiento de voz, sistemas de razonamiento basados en el

conocimiento, etc.

Los sistemas modernos distribuidos tipo pizarra, como JavaSpaces y TSpaces [URL 50, URL 25] se basan primero en un modelo de pares clave/valor

popularizado en Linda [CG90], donde el concepto se conocía como *tupla space*.

Con estos sistemas, puede almacenar objetos Java activos, no solo datos, en la pizarra y recuperarlos mediante la coincidencia parcial de campos (a través de plantillas y comodines) o por subtipos. Por ejemplo, supongamos que tiene un tipo Author, que es un subtipo de Person. Puede buscar en una pizarra que contenga objetos Person mediante una plantilla de autor con un valor lastName de "Shakespeare". Tendrías a Bill Shakespeare el autor, pero no a Fred Shakespeare el jardinero.

Las principales operaciones en JavaSpaces son:

Nombre	Función
leer	Busque y recupere datos del espacio.
escribir	Coloca un objeto en el espacio.
tomar	Similar a leer, pero también elimina el elemento del espacio.
notificar	Configure una notificación para que se produzca cada vez que se escriba un objeto que coincida con la plantilla.

T Spaces admite un conjunto similar de operaciones, pero con nombres diferentes y una semántica ligeramente diferente. Ambos sistemas están construidos como un producto de base de datos; Proporcionan operaciones atómicas y transacciones distribuidas para garantizar la integridad de los datos.

Dado que podemos almacenar objetos, podemos usar una pizarra para diseñar algoritmos basados en una *serie de objetos*, no solo en datos. Es como si nuestros detectives pudieran fijar a las personas en la pizarra, a los testigos mismos, no solo a sus declaraciones. Cualquiera puede hacer preguntas a un testigo en el seguimiento del caso, publicar la transcripción y mover a ese testigo a otra área de la pizarra, donde podría responder de manera diferente (si permite que el testigo también lea la pizarra).

Una gran ventaja de este tipo de sistemas es que dispone de una interfaz única y coherente con la pizarra. Al crear una aplicación distribuida convencional, puede dedicar una gran cantidad de tiempo a crear llamadas API únicas para cada transacción e interacción distribuida en el sistema. Con la explosión combinatoria de interfaces e

interacciones, el proyecto puede convertirse rápidamente en una pesadilla.

Organizar la pizarra

Cuando los detectives trabajan en casos grandes, la pizarra puede estar desordenada y puede resultar difícil localizar los datos en la pizarra. La solución es *particionar* la pizarra y comenzar a organizar los datos en la pizarra de alguna manera.

Los diferentes sistemas de software manejan esta partición de diferentes maneras; algunos utilizan zonas bastante planas o *grupos de interés*, mientras que otros adoptan una estructura más jerárquica en forma de árbol.

El estilo de programación de pizarra elimina la necesidad de tantas interfaces, lo que lo convierte en un sistema más elegante y consistente.

Ejemplo de aplicación

Supongamos que estamos escribiendo un programa para aceptar y procesar solicitudes de hipotecas o préstamos. Las leyes que rigen esta área son odiosamente complejas, y los gobiernos federales, estatales y locales tienen voz y voto. El prestamista debe probar que ha revelado ciertas cosas, y debe pedir cierta información, pero debe hacer ciertas otras preguntas, y así sucesivamente.

Más allá del miasma de la ley aplicable, también tenemos que lidiar con los siguientes problemas.

- No hay garantía sobre el orden en el que llegan los datos. Por ejemplo, las consultas para una verificación de crédito o una búsqueda de título pueden llevar una cantidad considerable de tiempo, mientras que elementos como el nombre y la dirección pueden estar disponibles de inmediato.
- La recopilación de datos puede ser realizada por diferentes personas, distribuidas en diferentes oficinas, en diferentes zonas horarias.
- Es posible que otros sistemas recopilen algunos datos de forma automática. Estos datos también pueden llegar de forma asincrónica.
- No obstante, es posible que ciertos datos sigan dependiendo de otros datos. Por ejemplo, es posible que no pueda iniciar la

búsqueda del título de un automóvil hasta que obtenga un comprobante de propiedad o seguro.

- La llegada de nuevos datos puede plantear nuevas preguntas y políticas. Supongamos que la verificación de crédito regresa con un informe menos que brillante; Ahora necesita estos cinco formularios adicionales y tal vez una muestra de sangre.

Puede intentar manejar todas las combinaciones y circunstancias posibles utilizando un sistema de flujo de trabajo. Existen muchos sistemas de este tipo, pero pueden ser complejos y requerir un gran esfuerzo para los programadores. A medida que cambian las regulaciones, el flujo de trabajo debe reorganizarse: es posible que las personas tengan que cambiar sus procedimientos y que el código programado tenga que reescribirse.

Una pizarra, en combinación con un motor de reglas que encapsula los requisitos legales, es una solución elegante a las dificultades que se encuentran aquí. El orden de llegada de los datos es irrelevante: cuando se publica un hecho, puede desencadenar las reglas adecuadas. La retroalimentación también se maneja fácilmente: la salida de cualquier conjunto de reglas puede publicarse en la pizarra y provocar la activación de aún más reglas aplicables.



Podemos usar la pizarra para coordinar hechos y agentes dispares, manteniendo al mismo tiempo la independencia e incluso el aislamiento entre los participantes.

Puedes lograr los mismos resultados con más métodos de fuerza bruta, por supuesto, pero tendrás un sistema más frágil. Cuando se rompe, es posible que todos los caballos del rey y todos los hombres del rey no vuelvan a hacer que su programa funcione.

Las secciones relacionadas incluyen:

- *El Pouer de Plaiw Texto*, página 73
- *Es solo a Vieu*, página 157

Desafíos

- ¿Utiliza sistemas de pizarra en el mundo real: el tablero de mensajes junto al refrigerador o la gran pizarra blanca en el trabajo? ¿Qué los hace

efectivos? ¿Alguna vez se publican mensajes con un formato coherente?
¿Importa?

Ejercicios

- Awsuer
ow p. 297
30. Para cada una de las siguientes aplicaciones, ¿sería adecuado un sistema de pizarra? ¿Apropiado o no? ¿Por qué?
1. Tratamiento de imágenes. Le gustaría que una serie de procesos paralelos tomaran fragmentos de una imagen, los procesaran y volvieran a colocar el fragmento completo.
 2. Calendario grupal. Hay personas dispersas por todo el mundo, en diferentes zonas horarias y hablando diferentes idiomas, tratando de programar una reunión.
 3. Herramienta de monitoreo de red. El sistema recopila estadísticas de rendimiento y recopila informes de problemas. Le gustaría implementar algunos agentes para usar esta información para buscar problemas en el sistema.

Capítulo 6

Mientras está codificando

La sabiduría convencional dice que una vez que un proyecto está en la fase de codificación, el trabajo es principalmente mecánico, transcribiendo el diseño en declaraciones ejecutables. Creemos que esta actitud es la razón principal por la que muchos programas son feos, ineficientes, mal estructurados, insostenibles y simplemente erróneos.

La codificación no es mecánica. Si así fuera, todas las herramientas CASE en las que la gente depositó sus esperanzas a principios de la década de 1980 habrían reemplazado a los programadores hace mucho tiempo. Hay decisiones que se deben tomar cada minuto, decisiones que requieren una reflexión y un juicio cuidadosos para que el programa resultante disfrute de una vida larga, precisa y productiva.

Los desarrolladores que no piensan activamente en su código están programando por coincidencia: el código puede funcionar, pero no hay una razón particular de por qué. En *Programmiwg de Coiwcidewce*, abogamos por una participación más positiva en el proceso de codificación.

Si bien la mayor parte del código que escribimos se ejecuta rápidamente, ocasionalmente desarrollamos algoritmos que tienen el potencial de atascar incluso a los procesadores más rápidos. En *Velocidad del algoritmo*, discutimos formas de estimar la velocidad del código y damos algunos consejos sobre cómo detectar problemas potenciales antes de que sucedan.

Los programadores pragmáticos piensan críticamente sobre todo el código, incluido el nuestro. Constantemente vemos margen de mejora en nuestros programas y nuestros diseños. En *Refactoriwg*, analizamos técnicas que nos ayudan a corregir el código existente incluso cuando estamos en medio de un proyecto.

Algo que debería estar en el fondo de tu mente siempre que estés

produciendo código es que algún día tendrás que probarlo. Simplifica el código

para probar, y aumentará la probabilidad de que realmente se pruebe, un pensamiento que desarrollamos en *Easy to Test de Code That*.

Finalmente, en *Evil Wizards*, sugerimos que tenga cuidado con las herramientas que escriben montones de código en su nombre a menos que entienda lo que están haciendo.

La mayoría de nosotros podemos conducir un automóvil en gran medida en piloto automático, no le ordenamos explícitamente a nuestro pie que presione un pedal o a nuestro brazo que gire el volante, simplemente pensamos "reduzca la velocidad y gire a la derecha". Sin embargo, los conductores buenos y seguros están constantemente revisando la situación, verificando si hay problemas potenciales y colocándose en buenas posiciones en caso de que suceda lo inesperado. Lo mismo ocurre con la codificación: puede ser en gran medida rutinaria, pero mantener el ingenio podría evitar un desastre.

31

Programación por Coincidencia

¿Alguna vez has visto viejas películas de guerra en blanco y negro? El cansado soldado avanza cautelosamente fuera de la maleza. Hay un claro más adelante: ¿hay minas terrestres o es seguro cruzarlo? No hay indicios de que se trate de un campo minado, ni señales, ni alambre de púas, ni cráteres. El soldado golpea el suelo delante de él con su bayoneta y hace una mueca, esperando una explosión. No hay ninguno. Así que avanza minuciosamente por el campo durante un tiempo, pinchando y hurgando a medida que avanza. Eventualmente, convencido de que el campo es seguro, se endereza y marcha orgullosamente hacia adelante, solo para ser volado en pedazos.

Las primeras exploraciones del soldado en busca de minas no revelaron nada, pero esto fue simplemente suerte. Lo llevaron a una conclusión falsa, con resultados desastrosos.

Como desarrolladores, también trabajamos en campos minados. Hay cientos de trampas esperando para atraparnos cada día. Recordando el cuento del soldado, debemos tener cuidado de no sacar conclusiones falsas. Debemos evitar la programación por coincidencia —confiando

en la suerte y en los éxitos accidentales— en favor de *la programación deliberadamente*.

Cómo programar por coincidencia

Supongamos que a Fred se le asigna una tarea de programación. Fred escribe un código, lo intenta y parece funcionar. Fred escribe un poco más de código, lo intenta y todavía parece funcionar. Fred escribe un poco más de código, lo intenta y todavía parece funcionar. Después de varias semanas de codificar de esta manera, el programa deja de funcionar repentinamente y, después de horas de intentar arreglarlo, todavía no sabe por qué. Es posible que Fred pase una cantidad significativa de tiempo persiguiendo este fragmento de código sin poder arreglarlo. No importa lo que haga, simplemente nunca parece funcionar bien.

Fred no sabe por qué el código está fallando porque *no lo hizo antes que lo hizo en el primer momento*. Parecía funcionar, dadas las limitadas "pruebas" que hizo Fred, pero eso fue solo una coincidencia. Animado por una falsa confianza, Fred avanzó hacia el olvido. Ahora, la mayoría de las personas inteligentes pueden conocer a alguien como Fred, pero tú sabes más. No nos basamos en coincidencias, ¿verdad?

A veces podríamos. A veces puede ser bastante fácil confundir una feliz coincidencia con un plan con propósito. Veamos algunos ejemplos.

Accidentes de implementación

Los accidentes de implementación son cosas que suceden simplemente porque esa es la forma en que el código está escrito actualmente. Terminas confiando en errores no documentados o condiciones de contorno.

Supongamos que llamas a una rutina con datos incorrectos. La rutina responde de una manera particular y se codifica en función de esa respuesta. Pero el autor no tenía la intención de que la rutina funcionara de esa manera, ni siquiera se consideró. Cuando la rutina se "arregla", el código puede romperse. En el caso más extremo, es posible que la rutina a la que llamaste ni siquiera esté diseñada para hacer lo que quieras, pero *parece* funcionar bien. Llamar a las cosas en el orden equivocado, o en el contexto equivocado, es un problema relacionado.

```
pintura(g);
invalidar();
validar();
revalidar();
repintar();
paintInmediatamente
(r);
```

Aquí parece que Fred está tratando desesperadamente de sacar algo en

la pantalla. Pero estas rutinas nunca fueron diseñadas para ser llamadas de esta manera; Aunque parecen funcionar, en realidad es solo una coincidencia.

Para colmo de males, cuando el componente finalmente se extrae, Fred no intentará volver atrás y eliminar las llamadas espurias. "Funciona ahora, mejor dejarlo en paz. "

Es fácil dejarse engañar por esta línea de pensamiento. ¿Por qué deberías correr el riesgo de meterte con algo que está funcionando? Pues bien, se nos ocurren varias razones:

- Puede que no esté funcionando, puede parecer que lo está.
- La condición de contorno en la que confía puede ser solo un accidente. En diferentes circunstancias (una resolución de pantalla diferente, tal vez), podría comportarse de manera diferente.
- El comportamiento no documentado puede cambiar con la próxima versión de la biblioteca.
- Las llamadas adicionales e innecesarias hacen que el código sea más lento.
- Las llamadas adicionales también aumentan el riesgo de introducir nuevos errores propios.

Para el código que escribes y que otros llamarán, los principios básicos de una buena modularización y de ocultar la implementación detrás de interfaces pequeñas y bien documentadas pueden ayudar. Un contrato bien especificado (véase *Design by Contract*, página 109) puede ayudar a eliminar malentendidos.

En el caso de las rutinas a las que llama, confíe únicamente en el comportamiento documentado. Si no puedes, por la razón que sea, entonces documenta bien tu suposición.

Accidentes de contexto

También puede haber "accidentes de contexto". Supongamos que está escribiendo un módulo de utilidad. El hecho de que actualmente esté codificando para un entorno de GUI, ¿el módulo tiene que depender de la presencia de una GUI? ¿Confía en los usuarios de habla inglesa? ¿Usuarios alfabetizados? ¿En qué más confías que no esté garantizado?

Supuestos implícitos

Las coincidencias pueden inducir a error en todos los niveles, desde la generación de requisitos hasta las pruebas. Las pruebas están particularmente plagadas de falsas causalidades y resultados coincidentes. Es fácil suponer que *X* causa *Y*, pero como dijimos en

Debuggiwg, página 90: no lo asumas, demuéstralolo.

En todos los niveles, las personas operan con muchas suposiciones en mente, pero estas suposiciones rara vez se documentan y, a menudo, entran en conflicto entre diferentes desarrolladores. Las suposiciones que no se basan en hechos bien establecidos son la pesadilla de todos los proyectos.

CONSEJO

No programes por casualidad

Cómo programar deliberadamente

Queremos dedicar menos tiempo a producir código, detectar y corregir errores lo más pronto posible en el ciclo de desarrollo y, para empezar, crear menos errores. Ayuda si podemos programar deliberadamente:

- Sé siempre consciente de lo que estás haciendo. Fred dejó que las cosas se le fueran de las manos lentamente, hasta que terminó hervido, como la rana en Stowe *Soup and Boiled Frogs*, página 7.
- No codifiques con los ojos vendados. Intentar crear una aplicación que no se entiende completamente, o utilizar una tecnología con la que no está familiarizado, es una invitación a dejarse engañar por las coincidencias.
- Proceda a partir de un plan, ya sea que ese plan esté en su cabeza, en el reverso de una servilleta de cóctel o en una impresión del tamaño de una pared de una herramienta CASE.
- Confía solo en cosas confiables. No dependa de accidentes ni de suposiciones. Si no puedes notar la diferencia en circunstancias particulares, asume lo peor.
- Documenta tus suposiciones. *Design by Contract*, página 109, puede ayudar a aclarar sus suposiciones en su propia mente, así como ayudar a comunicarlas a los demás.
- No te limites a probar tu código, sino también a tus suposiciones. No adivines; Pruébalos de verdad. Escribe una aserción para probar tus suposiciones (ver *Programa assertivo*, página 122). Si su afirmación es correcta, ha mejorado la documentación de su código. Si descubres que tu suposición es incorrecta, entonces considérate afortunado.
- Prioriza tu esfuerzo. Dedique tiempo a los aspectos importantes; Lo

más probable es que estas sean las partes difíciles. Si no tienes fundamentos

Las campanas y silbatos correctos y brillantes serán irrelevantes.

- No seas esclavo de la historia. No permita que el código existente dicte el código futuro. Todo el código puede ser reemplazado si ya no es apropiado. Incluso dentro de un mismo programa, no dejes que lo que ya has hecho limite lo que haces a continuación: prepárate para refactorizar (ver *Refactoriwg*, página 184). Esta decisión puede afectar el cronograma del proyecto. La suposición es que el impacto será menor que el costo de hacer el cambio.¹

Así que la próxima vez que algo parezca funcionar, pero no sepas por qué, asegúrate de que no sea solo una coincidencia.

Las secciones relacionadas incluyen:

- *Stowe Soup awd Ranas hervidas*, página 7
- *Debuggiwg*, página 90
- *Diseñado por Cowract*, página 109
- *Programmiwg asertivo*, página 122
- *Temporal Coupliwg*, página 150
- *Refactoriwg*, página 184
- *Todo es Writiwg*, página 248

Ejercicios

- Awsuer
ow p. 298*
31. ¿Puede identificar algunas coincidencias en el siguiente fragmento de código C? Supongamos que este código está enterrado en lo más profundo de una rutina de biblioteca.

```
fprintf(stderr, "dError, continuar?");
gets(buf);
```

- Awsuer
ow p. 298*
32. Este fragmento de código C puede funcionar parte del tiempo, en algunas máquinas. Entonces de nuevo, puede que no. ¿Qué pasa?

```
/* Truncar la cadena hasta sus últimos caracteres maxlen */
void string_tail(char *string, int maxlen) {
    int len = strlen(cadena);
    if (len > maxlen) {
        strcpy(cadena, cadena + (len - maxlen));
    }
}
```

1. También se puede ir demasiado lejos aquí. Una vez conocimos a un desarrollador que reescribió todas las fuentes que le dieron porque tenía sus propias convenciones de nomenclatura.

33. Este código proviene de un conjunto de rastreo de Java de propósito general. La función

Escribe una cadena en un archivo de registro. Pasa su prueba unitaria, pero falla cuando uno de los desarrolladores web lo utiliza. ¿En qué coincidencia se basa?

*Awsuer
ow p. 299*

```
public static void debug(String s) throws IOException {
    FileWriter fw = new FileWriter("debug.log", true); fw.write(s);
    fw.flush();
    fw.close();
}
```

32

Algoritmo Velocidad

En *Estimatiwg*, página 64, hablamos de estimar cosas como el tiempo que se tarda en caminar por la ciudad, o el tiempo que tardará un proyecto en terminarse. Sin embargo, hay otro tipo de estimación que los programadores pragmáticos utilizan casi a diario: la estimación de los recursos que utilizan los algoritmos: tiempo, procesador, memoria, etc.

Este tipo de estimación suele ser crucial. Si te dan a elegir entre dos formas de hacer algo, ¿cuál eliges? Sabe cuánto tiempo se ejecuta su programa con 1.000 registros, pero ¿cómo se escalará a 1.000.000? ¿Qué partes del código hay que optimizar?

Resulta que estas preguntas a menudo se pueden responder usando el sentido común, algo de análisis y una forma de escribir aproximaciones llamada la notación de la "gran O".

¿A qué nos referimos con algoritmos de estimación?

La mayoría de los algoritmos no triviales manejan algún tipo de entrada variable: ordenar n cadenas, invertir una $m \times n$ matriz o descifrar un mensaje con una n clave de bits. Normalmente, el tamaño de esta entrada afectará al algoritmo: cuanto mayor sea la entrada, mayor será el tiempo de ejecución o más memoria se utilizará.

Si la relación fuera siempre lineal (de modo que el tiempo aumentara en proporción directa al valor de n), esta sección no sería importante. Sin embargo, la mayoría de los algoritmos significativos no son lineales. La buena noticia es que muchos son sublineales. Una búsqueda binaria, por ejemplo, no necesita mirar a todos los

candidatos para encontrar una coincidencia. La mala noticia es que

Otros algoritmos son considerablemente peores que los lineales; los tiempos de ejecución o los requisitos de memoria aumentan mucho más rápido que n . Un algoritmo que tarda un minuto en procesar diez elementos puede tardar toda una vida en procesar 100.

Descubrimos que cada vez que escribimos algo que contiene bucles o llamadas recurrentes, comprobamos inconscientemente los requisitos de tiempo de ejecución y memoria. Rara vez se trata de un proceso formal, sino más bien de una rápida confirmación de que lo que estamos haciendo es sensato dadas las circunstancias. Sin embargo, a veces nos encontramos realizando un análisis más detallado. Ahí es cuando la $O()$ notación resulta útil.

La notación $O()$

El $O()$ La notación es una forma matemática de tratar con aproximaciones. Cuando escribimos que una rutina de ordenación en particular ordena n registros en $O(n^2)$ tiempo, simplemente estamos diciendo que el peor de los casos variará como el cuadrado de n . Duplique el número de registros, y el tiempo aumentará aproximadamente cuatro veces. Piensa en el O como significado *De acuerdo con el orden de*. El $O()$ La notación pone un límite superior al valor de la cosa que estamos midiendo (tiempo, memoria, etc.). Si decimos que una función toma $O(n^2)$ tiempo, entonces sabemos que el límite superior del tiempo que tarda no crecerá más rápido que n^2 . A veces se nos ocurren cosas bastante complejas $O()$ funciones, sino porque el término de orden más alto dominará el valor como n la convención es eliminar todos los términos de orden inferior, y no para molestarse en mostrar cualquier factor multiplicador constante. $O(\frac{n^2}{2} + 3n)$ es el Igual que $O(n^2)$, que es equivalente a $O(n^2)$. Esto es en realidad una debilidad

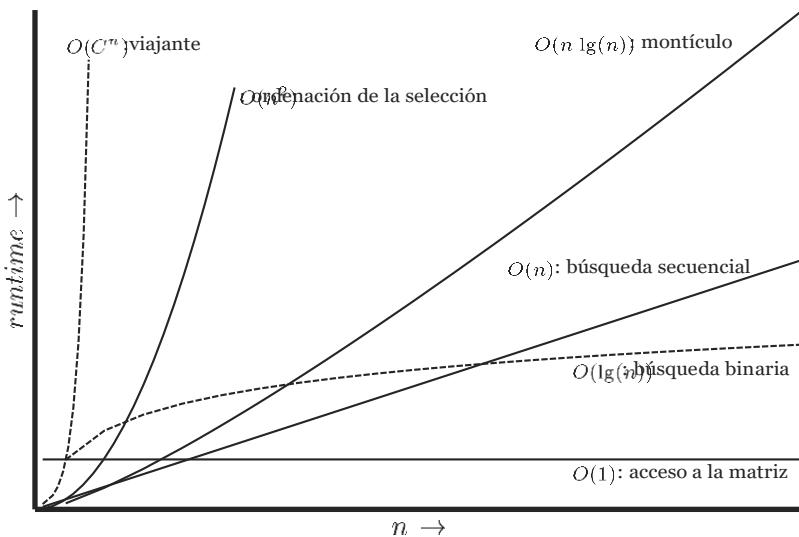
de la notación, uno $O(n^2)$ algoritmo puede ser 1.000 veces más rápido que otro $O(n)$ algoritmo, pero no lo sabrás por la notación.

La figura 6.1 muestra varias $O()$ notaciones comunes, junto con un gráfico que compara los tiempos de ejecución de los algoritmos en cada categoría. Claramente, las cosas rápidamente comienzan a salirse de control una vez que superamos $O(n^2)$.

Por ejemplo, supongamos que tienes una rutina que toma 1 s para procesar 100 registros. ¿Cuánto tiempo se tarda en procesar 1.000? Si el código es $O(1)$, entonces todavía tomará 1 s. Si es $O(\lg(n))$, entonces probablemente estarás esperando unos 3 s. $O(n)$ most $O(n^2)$ aumento

lineal a 10 s, mientras que un tardará unos 33 s. Si tienes la mala suerte de tener un rutina, luego siéntese durante 100 segundos mientras hace sus cosas. Y si estás usando un exponencial

Figura 6.1. Tiempos de ejecución de varios algoritmos



Algunas notaciones comunes de $O()$

- $O(1)$ Constante (elemento de acceso en matriz, sentencias simples)
- $O(\lg(n))$ Logarítmico (búsqueda binaria) [El wotatio es abreviado para $\lg(n)$]
- $O(n)$ Lineal (búsqueda secuencial)
- $O(n \lg(n))$ Peor que lineal, pero no mucho peor (tiempo de ejecución promedio de quicksort, heapsort)
- $O(n^2)$ Ley del cuadrado (ordenaciones de selección e inserción)
- $O(n^3)$ Cúbico (multiplicación de 2 matrices)
- $O(C^n)$ Exponencial (problema del vendedor viajero, partición de conjuntos)

, $O(2^n)$ es posible que desee preparar una taza de café: su rutina debería terminar en unos 10^{263} años. Cuéntanos cómo termina el universo.

La $O()$ notación no se aplica solo al tiempo, sino que se puede utilizar para representar cualquier otro recurso utilizado por un algoritmo. Por ejemplo, a menudo es útil poder modelar el consumo de memoria (véase el Ejercicio 35 en la página 183).

Estimación de sentido común

Puede estimar el orden de muchos algoritmos básicos utilizando el sentido común.

- **Bucles simples.** Si un bucle simple va desde 1 hasta n , entonces es probable que el algoritmo sea $O(n)$ —el tiempo aumenta linealmente con n . Algunos ejemplos son las búsquedas exhaustivas, la búsqueda del valor máximo en una matriz y la generación de sumas de comprobación.
- **Bucles anidados.** Si anidas un bucle dentro de otro, entonces tu algoritmo se convierte en $O(m \times n)$, donde m y n son los límites de los dos bucles. Esto suele ocurrir en algoritmos de ordenación sencillos, como la ordenación burbujeante, en la que el bucle exterior examina cada elemento de la matriz a su vez, y el bucle interior determina dónde colocar ese elemento en el resultado ordenado. Estos algoritmos de clasificación tienden a ser $O(n^2)$.
- **Chuleta binaria.** Si su algoritmo reduce a la mitad el conjunto de cosas que considera cada vez que da la vuelta al bucle, entonces es probable que sea logarítmico $O(\lg(n))$ (consulte el Ejercicio 37, página 183). Una búsqueda binaria de una lista ordenada, atravesando un árbol binario y encontrando el primer bit de conjunto en una palabra de máquina puede ser $O(\lg(n))$.
- **Divide y vencerás.** Los algoritmos que partitionan su entrada, trabajan en las dos mitades de forma independiente y luego combinan el resultado pueden ser $O(n \lg(n))$. El ejemplo clásico es quicksort, que funciona dividiendo los datos en dos mitades y ordenando cada una de ellas de forma recursiva. Aunque técnicamente $O(n^2)$, debido a que su comportamiento se degrada cuando se alimenta con una entrada ordenada, el tiempo de ejecución promedio de quicksort es $O(n \lg(n))$.
- **Combinatoria.** Cada vez que los algoritmos comienzan a observar las permutaciones de las cosas, sus tiempos de ejecución pueden

salirse de control. Esto se debe a que las permutaciones involucran factoriales (hay $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$ permutaciones de los dígitos del 1 al 5). Tiempo una combinatoria

Algoritmo para cinco elementos: se tardará seis veces más en ejecutarlo durante seis y 42 veces más para siete. Los ejemplos incluyen algoritmos para muchos de los problemas de *hard* reconocidos: el problema del vendedor viajero, empaquetar cosas de manera óptima en un contenedor, dividir un conjunto de números para que cada conjunto tenga el mismo total, etc. A menudo, la heurística se utiliza para reducir los tiempos de ejecución de este tipo de algoritmos en dominios de problemas particulares.

Velocidad del algoritmo en la práctica

Es poco probable que pases mucho tiempo durante tu carrera escribiendo rutinas de clasificación. Los que están en las bibliotecas disponibles para usted probablemente superarán cualquier cosa que pueda escribir sin un esfuerzo sustancial. Sin embargo, los tipos básicos de algoritmos que hemos descrito anteriormente aparecen una y otra vez. Cada vez que te encuentres escribiendo un bucle simple, sabes que tienes un $O(n)$ algoritmo. Si ese bucle contiene un bucle interno, entonces estás viendo $O(m \times n)$. Debería preguntarse qué tan grandes pueden llegar a ser estos valores. Si los números están acotados, sabrás cuánto tiempo tardará en ejecutarse el código. Si los números dependen de factores externos (como el número de registros en una ejecución por lotes nocturna o el número de nombres en una lista de personas), es posible que desee detenerse y considerar el efecto que los valores grandes pueden tener en el tiempo de ejecución o el consumo de memoria.

CONSEJO

Calcule el orden de sus algoritmos

Hay algunos enfoques que puede adoptar para abordar los problemas potenciales. Si tienes un algoritmo que es $O(n^2)$, trata de encontrar un enfoque de divide y vencerás que te lleve a $O(n \lg(n))$.

Si no está seguro de cuánto tiempo tardará el código o cuánta memoria usará, intente ejecutarlo, variando el recuento de registros de entrada o lo que sea que pueda afectar al tiempo de ejecución. A continuación, trace los resultados. Pronto deberías tener una buena idea de la forma de la curva. ¿Se curva hacia arriba, en línea recta o se aplana a medida que aumenta el tamaño de la entrada? Tres o cuatro puntos deberían darte una idea.

Ten en cuenta también lo que estás haciendo en el propio código. Un bucle simple $O(n^2)$ puede funcionar mejor que uno complejo, $O(n \lg(n))$ uno para bucles más pequeños

valores de n , particularmente si el $O(n \lg(n))$ algoritmo tiene un bucle interno costoso.

En medio de toda esta teoría, no olvides que también hay consideraciones prácticas. Puede parecer que el tiempo de ejecución aumenta linealmente para conjuntos de entrada pequeños. Pero alimenta el código con millones de registros y, de repente, el tiempo se degrada a medida que el sistema comienza a tambalearse. Si prueba una rutina de ordenación con teclas de entrada aleatorias, es posible que se sorprenda la primera vez que encuentre una entrada ordenada. Los programadores pragmáticos tratan de cubrir tanto las bases teóricas como las prácticas. Después de toda esta estimación, el único tiempo que cuenta es la velocidad de su código, ejecutándose en el entorno de producción, con datos reales.² Esto nos lleva a nuestro siguiente consejo.

CONSEJO

Pon a prueba tus estimaciones

Si es difícil obtener tiempos precisos, use *generadores de perfiles de código* para contar el número de veces que se ejecutan los diferentes pasos de su algoritmo y trace estas cifras contra el tamaño de la entrada.

Lo mejor no siempre es lo mejor

También hay que ser pragmático a la hora de elegir los algoritmos adecuados: el más rápido no siempre es el mejor para el trabajo. Dado un conjunto de entradas pequeño, una ordenación de inserción sencilla funcionará tan bien como una ordenación rápida, y le llevará menos tiempo escribir y depurar. También debe tener cuidado si el algoritmo que elija tiene un alto costo de configuración. Para conjuntos de entrada pequeños, esta configuración puede empequeñecer el tiempo de ejecución y hacer que el algoritmo sea inapropiado.

También hay que tener cuidado con la *optimización de la prematura*. Siempre es una buena idea asegurarse de que un algoritmo es realmente un cuello de botella antes de invertir su valioso tiempo tratando de mejorarlo.

2. De hecho, mientras probaban los algoritmos de ordenación utilizados como

ejercicio para esta sección en un Pentium de 64 MB, los autores se quedaron sin memoria real mientras ejecutaban la ordenación radix con más de siete millones de números. La clasificación comenzó a usar el espacio de intercambio y los tiempos se degradaron drásticamente.

Las secciones relacionadas incluyen:

- *Estimatiwg*, página 64

Desafíos

- Todos los desarrolladores deben tener una idea de cómo se diseñan y analizan los algoritmos. Robert Sedgewick ha escrito una serie de libros accesibles sobre el tema ([Sed83, SF96, Sed92] y otros). Te recomendamos que añadas uno de sus libros a tu colección y que te asegures de leerlo.
- Para aquellos a los que les gusta más detalle de lo que proporciona Sedgewick, lean los libros definitivos de Donald Knuth *Art of Computer Programming*, que analizan una amplia gama de algoritmos [Knu97a, Knu97b, Knu98].
- En el ejercicio 34, observamos la ordenación de matrices de enteros largos. ¿Cuál es el impacto si las claves son más complejas y la sobrecarga de la comparación de claves es alta? ¿La estructura de claves afecta a la eficiencia de los algoritmos de ordenación o la ordenación más rápida es siempre la más rápida?

Ejercicios

34. Hemos codificado un conjunto de rutinas de clasificación simples, que se pueden descargar

*Awsuer
ow p. 299*

de nuestro sitio web (www.pragmaticprogrammer.com). Ejecútelo en varias máquinas disponibles para usted. ¿Sus cifras siguen las curvas esperadas? ¿Qué puede deducir sobre las velocidades relativas de sus máquinas? ¿Cuáles son los efectos de varias configuraciones de optimización del compilador? ¿Es el orden de base realmente lineal?

35. La siguiente rutina imprime el contenido de un árbol binario. Suponiendo que el

*Awsuer
ow p. 300*

El árbol está equilibrado, ¿aproximadamente cuánto espacio de pila utilizará la rutina al imprimir un árbol de 1.000.000 de elementos? (Suponga que las llamadas a subrutinas no imponen una sobrecarga de pila significativa).

```
void printTree(const Nodo *nodo) {
    búfer de caracteres [1000];
    if (nodo) {
        printTree(nodo->izquierda);
        getNodeAsString(nodo, búfer); puts(búfer);
        printTree(nodo->derecha);
    }
}
```

36. ¿Puedes ver alguna forma de reducir los requisitos de pila de la rutina en Ejercicio 35 (aparte de reducir el tamaño del

*Awsuer
amortigua
dor?*

ow p. 300

37. En la página 180, afirmamos que un corte binario es $O(\lg(n))$. ¿Puedes probar
éste?

*Awsuer
ow p. 301*

Refactorización

Châwge awd decay iw todo lo que veo . . .

► H. F. Lyte, "Quédate conmigo"

A medida que un programa evoluciona, será necesario replantearse las decisiones anteriores y reelaborar partes del código. Este proceso es perfectamente natural. El código tiene que evolucionar; No es algo estático.

Desafortunadamente, la metáfora más común para el desarrollo de software es la construcción de edificios (Bertrand Meyer [Mey97b] utiliza el término "Software Construction"). Pero el uso de la construcción como metáfora rectora implica estos pasos:

1. Un arquitecto elabora planos.
2. Los contratistas cavan los cimientos, construyen la superestructura, el alambre y la plomada, y aplican los toques finales.
3. Los inquilinos se mudan y viven felices para siempre, llamando al mantenimiento del edificio para solucionar cualquier problema.

Bueno, el software no funciona de esa manera. Más que construcción, el software se parece más a *gardewiwg*: es más orgánico que el hormigón. Se plantan muchas cosas en un jardín de acuerdo con un plan y unas condiciones iniciales. Algunos prosperan, otros están destinados a terminar como abono. Puede mover las plantas unas con respecto a otras para aprovechar el juego de luces y sombras, viento y lluvia. Las plantas demasiado crecidas se dividen o podan, y los colores que chocan pueden trasladarse a lugares más agradables estéticamente. Arranca las malas hierbas y fertiliza las plantaciones que necesitan ayuda adicional. Monitorea constantemente la salud del jardín y realiza ajustes (al suelo, las plantas, el diseño) según sea necesario.

Los empresarios se sienten cómodos con la metáfora de la construcción de edificios: es más científica que la jardinería, es repetible, hay una jerarquía jerárquica rígida para la gestión, etc. Pero no estamos construyendo rascacielos, no estamos tan limitados por los límites de la física y el mundo real.

La metáfora de la jardinería está mucho más cerca de las realidades del desarrollo de software. Tal vez una cierta rutina se ha vuelto

demasiado grande, o está tratando de

Para lograr demasiado, hay que dividirlo en dos. Las cosas que no salen según lo planeado deben desmalezarse o podarse.

La reescritura, la reelaboración y la rearquitectura del código se conocen colectivamente como
Refactoriwg.

¿Cuándo se debe refactorizar?

Cuando te encuentres con un obstáculo porque el código ya no encaja, o te des cuenta de que dos cosas realmente deberían fusionarse, o cualquier otra cosa que te parezca "incorrecta", *no dudes en hacerlo*. No hay mejor momento que el presente. Hay varios factores que pueden hacer que el código cumpla los requisitos para la refactorización:

- Duplicación. Has descubierto una infracción de la *SECO Principio* (*Los males de Duplication*, página 26).
- Diseño no ortogonal. Ha descubierto algún código o diseño que podría hacerse más ortogonal (*Ortogovia*, página 34).
- Conocimientos obsoletos. Las cosas cambian, los requisitos se desvían y su conocimiento del problema aumenta. El código debe mantenerse al día.
- Rendimiento. Es necesario mover la funcionalidad de un área del sistema a otra para mejorar el rendimiento.

La refactorización del código (mover la funcionalidad y actualizar las decisiones más tempranas) es realmente un ejercicio de *paiw mawagemewt*. Seamos realistas, cambiar el código fuente puede ser bastante doloroso: casi funcionaba, y ahora está *realmente* roto. Muchos desarrolladores son reacios a comenzar a dividir el código solo porque no es del todo correcto.

Complicaciones del mundo real

Así que vas a tu jefe o cliente y le dices: "Este código funciona, pero necesito otra semana para refactorizarlo".

No podemos imprimir su respuesta.

La presión del tiempo se utiliza a menudo como excusa para no refactorizar. Pero esta excusa simplemente no se sostiene: si no se refactoriza ahora, habrá una inversión de tiempo mucho mayor para solucionar el problema en el futuro, cuando haya más dependencias con las que contar. ¿Habrá más tiempo disponible entonces? No en

nuestra experiencia.

Es posible que desee explicarle este principio al jefe utilizando una analogía médica: piense en el código que necesita refactorización como un "crecimiento". Su extirpación requiere una cirugía invasiva. Puedes entrar ahora y sacarlo mientras aún es pequeño. O bien, puede esperar mientras crece y se propaga, pero volver a eliminarlo entonces será más costoso y más peligroso. Espere aún más y puede perder al paciente por completo.

CONSEJO

Refactorizar temprano, refactorizar a menudo

Realice un seguimiento de las cosas que deben refactorizarse. Si no puede refactorizar algo de inmediato, asegúrese de que se coloque en la programación. Asegúrese de que los usuarios del código afectado *kwou* que está programado para ser refactorizado y cómo esto podría afectarles.

¿Cómo se refactoriza?

La refactorización comenzó en la comunidad de Smalltalk y, junto con otras tendencias (como los patrones de diseño), ha comenzado a ganar una mayor adeptos. Pero como tema todavía es bastante nuevo; No hay mucho publicado al respecto. El primer libro importante sobre refactorización ([FBB⁺99], y también [URL 47]) se publica casi al mismo tiempo que este libro.

En esencia, la refactorización es rediseño. Cualquier cosa que usted u otros miembros de su equipo hayan diseñado puede ser rediseñada a la luz de nuevos hechos, comprensiones más profundas, requisitos cambiantes, etc. Pero si procedes a fragmentar grandes cantidades de código con salvaje abandono, es posible que te encuentres en una posición peor que cuando empezaste.

Claramente, la refactorización es una actividad que debe llevarse a cabo de manera lenta, deliberada y cuidadosa. Martin Fowler ofrece los siguientes consejos sencillos sobre cómo refactorizar sin hacer más daño que bien (véase el recuadro de la página 30 de [FS97]):

1. No intente refactorizar y agregar funcionalidad al mismo tiempo.
2. Asegúrese de tener buenas pruebas antes de comenzar la refactorización. Ejecute las pruebas con la mayor frecuencia

possible. De esa manera, sabrás rápidamente si tus cambios han roto algo.

Refactorización automática

Históricamente, los usuarios de Smalltalk siempre han disfrutado de un *navegador de clase* como parte del IDE. No debe confundirse con los exploradores web, los exploradores de clases permiten a los usuarios navegar y examinar las jerarquías y los métodos de clase.

Normalmente, los exploradores de clases permiten editar código, crear nuevos métodos y clases, etc. La siguiente variación de esta idea es el *navegador de referencia*.

Un navegador de refactorización puede realizar de forma semiautomática operaciones de refactorización comunes: dividir una rutina larga en otras más pequeñas, propagar automáticamente los cambios en los nombres de los métodos y las variables, arrastrar y soltar para ayudarle a mover el código, etc.

Mientras escribimos este libro, esta tecnología aún no ha aparecido fuera del mundo de Smalltalk, pero es probable que cambie a la misma velocidad que Java: rápidamente. Mientras tanto, el navegador pionero de refactorización Small-talk se puede encontrar

3. Da pasos cortos y deliberados: mueve un campo de una clase a otra, fusiona dos métodos similares en una superclase. La refactorización a menudo implica realizar muchos cambios localizados que dan como resultado un cambio a mayor escala. Si mantiene los pasos pequeños y realiza pruebas después de cada paso, evitará la depuración prolongada.

Hablaremos más sobre las pruebas a este nivel en *Easy to Test de Code That*, página 189, y pruebas a mayor escala en *Ruthless Testing*, página 237, pero el punto del Sr. Fowler de mantener buenas pruebas de regresión es la clave para refactorizar con confianza.

También puede ser útil asegurarse de que los cambios drásticos en un módulo, como alterar su interfaz o su funcionalidad de manera incompatible, rompan la compilación. Es decir, los clientes antiguos de este código deberían fallar al compilar. A continuación, puede encontrar rápidamente los clientes antiguos y realizar los cambios necesarios para actualizarlos.

Así que la próxima vez que veas un fragmento de código que no sea como debería ser, arréglalo tanto como todo lo que depende de él. Controla el dolor: si duele ahora, pero va a doler aún más tarde, es

mejor que lo superes

con. Recuerde las lecciones de *Software Ewtropy*, página 4: no viva con ventanas rotas.

Las secciones relacionadas incluyen:

- *El Cat se comió mi código fuente*, página 2
- *Software Ewtropy*, página 4
- *Stowe Soup awd Ranas hervidas*, página 7
- *Los males de Duplicatiow*, página 26
- *Ortodomía*, página 34
- *Programmiwg por Coicidewce*, página 172
- *Codificar el Easy de That para probar*, página 189
- *El despiadado Testiwig*, página 237

Ejercicios

Awsuer
ow p. 302

38. Obviamente, el siguiente código se ha actualizado varias veces a lo largo del años, pero los cambios no han mejorado su estructura. Refactoríalo.

```
if (estado == TEXAS) {
    tasa = TX_RATE;
    amt = base * TX_RATE;
    calc = 2 * base (amt) + extra (amt) * 1.05;
}
else if ((estado == OHIO) || (estado == MAINE)) {
    tasa = (estado == OHIO) ? OH_RATE : ME_RATE;
    amt = base * tasa;
    calc = 2 * base (amt) + extra (amt) * 1.05;
    si (estado == OHIO)
        puntos = 2;
}
else {
    tasa = 1; amt
    = base;
    calc = 2 * base (amt) + extra (amt) * 1.05;
}
```

- Awsuer
ow p. 303
39. La siguiente clase Java debe admitir algunas formas más. Refactorizar el para prepararlo para las adiciones.

```
public class Shape {
    public static final int CUADRADO = 1;
    public static final int CIRCUNFERENCIA
        = 2;
    public static final int RIGHT_TRIANGLE = 3;
    private int shapeType;
    tamaño doble privado ;
    public Shape(int shapeType, double size) {
        this.shapeType = shapeType;
        éste.tamaño = tamaño;
    }
    ... otros métodos ...
}
```

```

área doble pública () {
    switch (shapeType) {
        caso CUADRADO: devolución tamaño * tamaño;
        caso CIRCUNFERENCIA: devolución Matemáticas.PI*tamaño*tamaño/4.0;
        Caso RIGHT_TRIANGLE: Tamaño de devolución * tamaño / 2.0;
    }
    devuelve 0;
}
}

```

40. Este código Java es parte de un marco que se utilizará en todo su proyecto. Refactorícelo para que sea más general y más fácil de extender en el futuro.

Awsuer
ow p. 303

```

public class Window {
    public Window(int width, int height) { ... } public
    void setSize(int width, int height) { ... }
    superposiciones booleanas públicas (Ventana w) {
        ...
    }
    public int getArea() { ... }
}

```

Código que es fácil de probar

El *IC de Software* es una metáfora que a la gente le gusta utilizar cuando se habla de reutilización y desarrollo basado en componentes.³ La idea es que los componentes de software se combinen de la misma manera que se combinan los chips de circuitos integrados. Esto solo funciona si se sabe que los componentes que está utilizando son confiables.

Los chips están diseñados para ser probados, no solo en la fábrica, no solo cuando se instalan, sino también en el campo cuando se implementan. Los chips y sistemas más complejos pueden tener una función completa de autoprueba incorporada (BIST) que ejecuta algunos diagnósticos de nivel básico internamente, o un mecanismo de acceso a la prueba (TAM) que proporciona un arnés de prueba que permite que el entorno externo proporcione estímulos y recopile respuestas del chip.

Podemos hacer lo mismo en el software. Al igual que nuestros colegas de hardware, necesitamos incorporar la capacidad de prueba en el software desde el principio y probar cada pieza a fondo antes de intentar conectarlas.

3. El término "IC de Software" (Circuito Integrado) parece haber sido inventado en 1986 por Cox y Novobilski en su libro de Objective-C *Object-Oriewted Programmiwg* [CN91].

Pruebas unitarias

Las pruebas a nivel de chip para hardware son aproximadamente equivalentes a *las pruebas unitarias* en software: pruebas realizadas en cada módulo, de forma aislada, para verificar su comportamiento. Podemos tener una mejor idea de cómo reaccionará un módulo en el gran mundo una vez que lo hayamos probado a fondo en condiciones controladas (incluso artificiales).

Una prueba unitaria de software es un código que ejercita un módulo. Por lo general, la prueba unitaria establecerá algún tipo de entorno artificial y luego invocará rutinas en el módulo que se está probando. A continuación, comprueba los resultados que se devuelven, ya sea con valores conocidos o con los resultados de ejecuciones anteriores de la misma prueba (prueba de regresión).

Más tarde, cuando ensamblemos nuestros "circuitos integrados de software" en un sistema completo, tendremos la confianza de que las partes individuales funcionan como se espera, y luego podremos usar las mismas instalaciones de prueba unitaria para probar el sistema en su conjunto. Hablamos de esta comprobación a gran escala del sistema en *Ruthless Testing*, página 237.

Sin embargo, antes de llegar tan lejos, tenemos que decidir qué probar a nivel de unidad. Por lo general, los programadores arrojan unos pocos bits aleatorios de datos al código y lo llaman probado. Podemos hacerlo mucho mejor, utilizando las ideas detrás del *diseño por contrato*.

Pruebas contra contrato

Nos gusta pensar en las pruebas unitarias como *pruebas contra contrato* (ver *Diseño por contrato*, página 109). Queremos escribir casos de prueba que garanticen que una unidad determinada cumple con su contrato. Esto nos dirá dos cosas: si el código cumple con el contrato, y si el contrato significa lo que creemos que significa. Queremos probar que el módulo ofrece la funcionalidad que promete, en una amplia gama de casos de prueba y condiciones límite.

¿Qué significa esto en la práctica? Echemos un vistazo a la rutina de raíz cuadrada que encontramos por primera vez en la página 114. Su contrato es sencillo:

```
requerir
    argumento >= 0;
```

```
asegurar
((Resultado * Resultado) - argumento).abs <= épsilon*argumento;
```

Esto nos indica qué probar:

- Pase un argumento negativo y asegúrese de que se rechace.
- Pase un argumento de cero para asegurarse de que se acepta (este es el valor límite).
- Pase valores entre cero y el argumento máximo expresable y verifique que la diferencia entre el cuadrado del resultado y el argumento original sea menor que una pequeña fracción del argumento.

Armados con este contrato, y asumiendo que nuestra rutina hace su propia verificación previa y posterior a la condición, podemos escribir un script de prueba básico para ejercitar la función de raíz cuadrada.

```
public void testValue(doble número, doble esperado) {
    resultado doble = 0,0;
    probar {                                // Nosotros puede lanzar un
        resultado = mySqrt(num);  Excepción de condición previa
    }
    catch (Arrojadizo, e) {
        si (Núm < 0.0)           // Si entrada es < 0, entonces
            devolución;          Esperamos que el
        más                      excepción, de lo contrario
            assert(falso);         fuerza un Fallo de prueba
        }
        assert(Math.abs(resultado-esperado) < épsilon*esperado);
    }
```

Entonces podemos llamar a esta rutina para probar nuestra función de raíz cuadrada:

```
testValue(-4.0, 0.0);
testValue( 0.0, 0.0);
testValue( 2.0, 1.4142135624);
testValue(64.0, 8.0);
testValue(1.0e7, 3162.2776602);
```

Esta es una prueba bastante simple; En el mundo real, es probable que cualquier módulo no trivial dependa de varios otros módulos, así que ¿cómo hacemos para probar la combinación?

Supongamos que tenemos un módulo A que usa un `LinkedList` y un `Sort`. En orden, probaríamos:

1. El contrato de `LinkedList`, en su totalidad
2. El contrato de `Sort`, al completo
3. El contrato de A, que se basa en los otros contratos, pero no los expone directamente

Este estilo de prueba requiere que primero pruebe los subcomponentes de un módulo. Una vez que se han verificado los subcomponentes, se puede probar el módulo en sí.

Si las pruebas de `LinkedList` y `Sort` pasaron, pero la prueba de `A` falló, podemos estar bastante seguros de que el problema está en `A`, o en el uso *de A* de uno de esos subcomponentes. Esta técnica es una excelente manera de reducir el esfuerzo de depuración: podemos concentrarnos rápidamente en la fuente probable del problema dentro del módulo `A` y no perder el tiempo reexaminando sus subcomponentes.

¿Por qué nos tomamos todas estas molestias? Sobre todo, queremos evitar crear una "bomba de tiempo", algo que pase desapercibido y explote en un momento incómodo más adelante en el proyecto. Al hacer hincapié en las pruebas contra el contrato, podemos tratar de evitar la mayor cantidad posible de esos desastres posteriores.

CONSEJO

Diseño para probar

Al diseñar un módulo, o incluso una sola rutina, debe diseñar tanto su contrato como el código para probar ese contrato. Al diseñar el código para que pase una prueba y cumpla con su contrato, es muy posible que considere las condiciones límite y otros problemas que de otro modo no se le ocurrirían. No hay mejor manera de corregir errores que evitándolos en primer lugar. De hecho, al crear las pruebas *antes* de implementar el código, puede probar la interfaz antes de comprometerse con ella.

Escritura de pruebas unitarias

Las pruebas unitarias de un módulo no deben introducirse en algún lugar lejano del árbol de origen. Deben estar convenientemente ubicados. En el caso de proyectos pequeños, puede incrustar la prueba unitaria de un módulo en el propio módulo. Para proyectos más grandes, sugerimos mover cada prueba a un subdirectorío. De cualquier manera, recuerde que si no es fácil de encontrar, no se usará.

Al hacer que el código de prueba sea fácilmente accesible, está proporcionando a los desarrolladores que pueden usar su código dos recursos invaluables:

1. Ejemplos de cómo utilizar toda la funcionalidad de su módulo

2. Un medio para crear pruebas de regresión para validar cualquier cambio futuro en el código

Es conveniente, pero no siempre práctico, que cada clase o módulo contenga su propia prueba unitaria. En Java, por ejemplo, cada clase puede tener su propia principal. En todos los archivos de clase de la aplicación, excepto en el principal de la aplicación, la rutina principal se puede usar para ejecutar pruebas unitarias; se ignorará cuando se ejecute la aplicación en sí. Esto tiene la ventaja de que el código que envía todavía contiene las pruebas, que se pueden usar para diagnosticar problemas en el campo.

En C++ puede lograr el mismo efecto (en tiempo de compilación) mediante `#ifdef` para compilar código de prueba unitaria de forma selectiva. Por ejemplo, aquí hay una prueba unitaria muy simple en C++, incrustada en nuestro módulo, que verifica nuestra función de raíz cuadrada usando una rutina `testValue` similar a la de Java definida anteriormente:

```
PRUEBA #ifdef
int main(int argc, char **argv)
{
    argc--; argv++;           // saltarse Nombre del programa
    si (Argc < 2) {          de la norma Pruebas si sin args
        testValue(-4.0, 0.0);
        testValue( 0.0, 0.0);
        testValue( 2.0, 1.4142135624);
        testValue(64.0, 8.0);
        testValue(1.0e7, 3162.2776602);
    }
    más {                  // más Usar args
        double num, esperado;
        while (argc >= 2) { num
            = atof(argv[0]);
            esperado = atof(argv[1]);
            testValue(num,esperado);
            argc -= 2;
            argv += 2;
        }
    }
    devuelve 0;
}
#endif
```

Esta prueba unitaria ejecutará un conjunto mínimo de pruebas o, si se le dan argumentos, le permitirá pasar datos del mundo exterior. Un script de shell podría usar esta capacidad para ejecutar un conjunto mucho más completo de pruebas.

¿Qué se hace si la respuesta correcta para una prueba unitaria es salir

o anular el programa? En ese caso, debe poder seleccionar la prueba que se va a ejecutar, quizás especificando un argumento en la línea de comandos. Podrás

También debe pasar parámetros si necesita especificar diferentes condiciones de inicio para sus pruebas.

Pero proporcionar pruebas unitarias no es suficiente. Debe ejecutarlos y ejecutarlos con frecuencia. También ayuda si la clase *pasa* sus exámenes de vez en cuando.

Uso de arneses de prueba

Debido a que generalmente escribimos *una gran cantidad* de código de prueba y hacemos muchas pruebas, nos haremos la vida más fácil y desarrollaremos un arnés de prueba estándar para el proyecto. El principal que se muestra en la sección anterior es un arnés de prueba muy simple, pero por lo general necesitaremos más funcionalidad que eso.

Un agente de pruebas puede controlar operaciones comunes, como el registro del estado, el análisis de la salida para obtener los resultados esperados y la selección y ejecución de las pruebas. Los arneses pueden estar impulsados por una interfaz gráfica de usuario, pueden estar escritos en el mismo idioma de destino que el resto del proyecto, o pueden implementarse como una combinación de archivos make y scripts Perl. En la respuesta al Ejercicio 41 de la página 305 se muestra un arnés de prueba sencillo.

En lenguajes y entornos orientados a objetos, puede crear una clase base que proporcione estas operaciones comunes. Las pruebas individuales pueden crear subclases a partir de eso y agregar código de prueba específico. Podría utilizar una convención de nomenclatura estándar y una reflexión en Java para construir una lista de pruebas de forma dinámica. Esta técnica es una buena manera de honrar el principio *DRY*: no tiene que mantener una lista de pruebas disponibles. Pero antes de que te vayas y empieces a escribir tu propio arnés, es posible que desees investigar el xUnit de Kent Beck y Erich Gamma en [URL 22]. También puede consultar nuestro libro *Pragmatic Unit Testing* [HT03] para obtener una introducción a JUnit.

Independientemente de la tecnología que decida usar, los arneses de prueba deben incluir las siguientes capacidades:

- Una forma estándar de especificar la configuración y la limpieza
- Un método para seleccionar pruebas individuales o todas las pruebas disponibles
- Un medio para analizar la salida de resultados esperados (o

inesperados) • Una forma estandarizada de notificación de errores

Las pruebas deben ser componibles; Es decir, una prueba puede estar compuesta por subpruebas de subcomponentes a cualquier profundidad. Podemos usar esta función para probar partes seleccionadas del sistema o todo el sistema con la misma facilidad, usando las mismas herramientas.

Pruebas ad hoc

Durante la depuración, es posible que terminemos creando algunas pruebas particulares sobre la marcha. Pueden ser tan simples como una instrucción de impresión o un fragmento de código ingresado de forma interactiva en un depurador o entorno IDE.

Al final de la sesión de depuración, debe formalizar la prueba ad hoc. Si el código se rompió una vez, es probable que se rompa de nuevo. No te limites a tirar la prueba que has creado; Agréguelo a la prueba unitaria existente.

Por ejemplo, usando JUnit (el miembro Java de la familia xUnit), podríamos escribir nuestra prueba de raíz cuadrada de la siguiente manera:

```
public class JUnitExample extends TestCase {
    public JUnitExample(nombre de la cadena final) {
        super(nombre);
    }
    protected void setUp() {
        Cargue los datos de prueba...
        testData.addElement(new DblPair(-4.0,0.0));
        testData.addElement(new DblPair(0.0,0.0));
        testData.addElement(new DblPair(64.0,8.0));
        testData.addElement(new DblPair(Double.MAX_VALUE,
            1.3407807929942597E154));
    }
    public void testMySqrt() {
        número doble , esperado, resultado = 0,0;
        Enumeración enumeración = testData.elements();
        while (enumeración.hasMoreElements()) {
            DBLPIER P = (DBLFIER)a.NextTellment();
            Num      = p.getNum();
            esperado = p.getExpected();
            testValue(num, esperado);
        }
    }
    public static Test suite() { TestSuite
        suite= new TestSuite();
        suite.addTest(new JUnitExample("testMySqrt"));
        suite de regreso ;
    }
}
```

JUnit está diseñado para ser componible: podríamos agregar tantas pruebas como quisieramos a esta suite, y cada una de esas pruebas podría ser a su vez una suite. Además, puede elegir entre una interfaz gráfica o por lotes para realizar las pruebas.

Crear una ventana de prueba

Es poco probable que incluso los mejores conjuntos de pruebas encuentren todos los errores; Hay algo en las condiciones húmedas y cálidas de un entorno de producción que parece sacarlos de la nada.

Esto significa que a menudo tendrá que probar una pieza de software una vez que se haya implementado, con datos del mundo real fluyendo por sus venas. A diferencia de una placa de circuito o un chip, no tenemos *pwis de prueba* en el software, pero *podemos* proporcionar varias vistas del estado interno de un módulo, sin usar el depurador (lo que puede ser inconveniente o imposible en una aplicación de producción).

Los archivos de registro que contienen mensajes de seguimiento son uno de estos mecanismos. Los mensajes de registro deben estar en un formato regular y coherente; Es posible que desee analizarlos automáticamente para deducir el tiempo de procesamiento o las rutas lógicas que tomó el programa. Los diagnósticos con un formato deficiente o incoherente son simplemente "vómitos": son difíciles de leer y poco prácticos de analizar.

Otro mecanismo para entrar en el código en ejecución es la secuencia de "teclas de acceso rápido". Cuando se presiona esta combinación particular de teclas, aparece una ventana de control de diagnóstico con mensajes de estado, etc. Esto no es algo que normalmente revelaría a los usuarios finales, pero puede ser muy útil para el servicio de asistencia.

En el caso de código de servidor más grande y complejo, una técnica ingeniosa para proporcionar una vista de su funcionamiento es incluir un servidor web incorporado. Cualquiera puede apuntar un navegador web al puerto HTTP de la aplicación (que generalmente está en un número no estándar, como 8080) y ver el estado interno, las entradas de registro y posiblemente incluso algún tipo de panel de control de depuración. Esto puede parecer difícil de implementar, pero no lo es. Los servidores web HTTP disponibles gratuitamente e integrables están disponibles en una variedad de idiomas modernos. Un buen lugar para empezar a buscar es [URL 58].

Una cultura de pruebas

Todo el software que escribas *será* probado, si no por ti y tu equipo, por los usuarios finales, por lo que es mejor que planees probarlo a fondo.

Un poco de previsión puede ser de gran ayuda para minimizar los costos de mantenimiento y las llamadas al servicio de asistencia.

A pesar de su reputación de hacker, la comunidad de Perl tiene un fuerte compromiso con las pruebas unitarias y de regresión. El procedimiento de instalación de módulos estándar de Perl admite una prueba de regresión invocando

% de hacer la prueba

No hay nada mágico en Perl en este sentido. Perl facilita la recopilación y el análisis de los resultados de las pruebas para garantizar el cumplimiento, pero la gran ventaja es simplemente que es un estándar: las pruebas van en un lugar particular y tienen un cierto resultado esperado. *Testiwg es más cultural que techwical*; Podemos inculcar esta cultura de prueba en un proyecto, independientemente del lenguaje que se utilice.

CONSEJO

Pruebe su software, o sus usuarios lo harán

Las secciones relacionadas incluyen:

- *El Cat se comió mi código fuente*, página 2
- *Ortodomía*, página 34
- *Diseñado por Cowtract*, página 109
- *Refactoriwg*, página 184
- *El despiadado Testiwg*, página 237

Ejercicios

41. Diseñe una plantilla de prueba para la interfaz de la licuadora descrita en la respuesta a Exer-

CISE 17 en la página 289. Escriba un script de shell que realice una prueba de regresión para el blender. Debe probar la funcionalidad básica, las condiciones de error y límite, y cualquier obligación contractual. ¿Qué restricciones se imponen para cambiar la velocidad? ¿Están siendo honrados?

*Awsuer
ow p. 305*

Mal Asistentes

No se puede negar: las aplicaciones son cada vez más difíciles de escribir. Las interfaces de usuario, en particular, son cada vez más sofisticadas. Hace veinte años, la aplicación promedio tendría una interfaz de teletipo de vidrio (si es que tenía una interfaz). Los terminales asíncronos normalmente proporcionarían una pantalla interactiva de caracteres, mientras que los dispositivos sondeables (como el omnipresente IBM 3270) le permitirían llenar una pantalla completa antes de presionar SEND . Ahora, los usuarios esperan interfaces gráficas de usuario, con ayuda sensible al contexto, cortar y pegar, arrastrar y soltar, integración OLE y MDI o SDI. Los usuarios buscan integración de navegador web y soporte de cliente ligero.

Cada vez más, las propias aplicaciones son cada vez más complejas. En la actualidad, la mayoría de los desarrollos utilizan un modelo de varios niveles, posiblemente con alguna capa de middleware o un monitor de transacciones. Se espera que estos programas sean dinámicos y flexibles, y que interoperen con aplicaciones escritas por terceros.

Ah, ¿y mencionamos que lo necesitábamos todo la próxima semana?

Los desarrolladores están luchando por mantenerse al día. Si estuviéramos usando el mismo tipo de herramientas que produjeron las aplicaciones básicas de terminales tontos hace 20 años, nunca lograríamos hacer nada.

Así que los fabricantes de herramientas y los proveedores de infraestructuras han ideado una bala mágica, el *wizard*. Los magos son geniales. ¿Necesita una aplicación MDI con soporte para contenedores OLE? Simplemente haga clic en un solo botón, responda un par de preguntas simples y el asistente generará automáticamente el código esqueleto para usted. El entorno de Microsoft Visual C++ crea más de 1.200 líneas de código para este escenario, automáticamente. Los magos también trabajan duro en otros contextos. Puede usar asistentes para crear componentes de servidor, implementar beans de Java y manejar interfaces de red, todas áreas complejas en las que es bueno contar con la ayuda de un experto.

Pero el uso de un mago diseñado por un gurú no hace automáticamente que Joe sea igualmente experto. Joe puede sentirse bastante bien: acaba de producir una gran cantidad de código y un programa de aspecto bastante elegante. Simplemente agrega la

funcionalidad específica de la aplicación y está listo para enviar. Pero a menos que Joe realmente entienda el código que se ha producido en su nombre, se está engañando a sí mismo. Está programando por casualidad. Los magos son una calle de un solo sentido: cortan el código por ti y luego siguen adelante. Si el

El código que producen no es del todo correcto, o si las circunstancias cambian y necesitas adaptar el código, estás solo.

No estamos en contra de los magos. Por el contrario, dedicamos una sección entera (*Code Generators*, página 102) a escribir el suyo propio. Pero si usa un asistente y no comprende todo el código que produce, no tendrá el control de su propia aplicación. No podrá mantenerlo y tendrá dificultades cuando llegue el momento de depurar.

CONSEJO

No utilices código de asistente que no entiendas

Algunas personas sienten que esta es una posición extrema. Dicen que los desarrolladores confían rutinariamente en cosas que no comprenden completamente: la mecánica cuántica de los circuitos integrados, la estructura de interrupción del procesador, los algoritmos utilizados para programar procesos, el código en las bibliotecas suministradas, etc. Estamos de acuerdo. Y sentiríamos lo mismo acerca de los magos si fueran simplemente un conjunto de llamadas a la biblioteca o servicios estándar del sistema operativo en los que los desarrolladores pudieran confiar. Pero no lo son. Los magos generan código que se convierte en una parte integral de la aplicación de Joe. El código del asistente no se factoriza detrás de una interfaz ordenada, sino que se entrelaza línea por línea con la funcionalidad que escribe Joe.⁴ Eventualmente, deja de ser el código del mago y comienza a ser el de Joe. Y nadie debería producir código que no entienda completamente.

Las secciones relacionadas incluyen:

- *Ortodomía*, página 34
- *Code Generators*, página 102

Desafíos

- Si tiene un asistente de creación de GUI disponible, utilícelo para generar una aplicación de esqueleto. Revisa cada línea de código que produce. ¿Lo entiendes todo? ¿Podrías haberlo producido tú mismo? ¿Lo habrías producido tú mismo o está haciendo cosas que no necesitas?

4. Sin embargo, existen otras técnicas que ayudan a gestionar la complejidad. Discutimos dos, los frijoles y el AOP, en *Ortogramía*, página 34.

Esta página se ha dejado en blanco intencionadamente

Capítulo 7

Antes del proyecto

¿Alguna vez has tenido la sensación de que tu proyecto está condenado, incluso antes de que comience? A veces puede serlo, a menos que primero establezcas algunas reglas básicas básicas. De lo contrario, también podría sugerir que se cierre ahora y ahorrarle algo de dinero al patrocinador.

Al principio de un proyecto, deberá determinar los requisitos. No basta con escuchar a los usuarios: lea *The Requirements Pit* para obtener más información.

La sabiduría convencional y la gestión de restricciones son los temas de *Solving Impossible Puzzles*. Ya sea que esté realizando requisitos, análisis, codificación o pruebas, surgirán problemas difíciles. La mayoría de las veces, no serán tan difíciles como parecen a primera vista.

Cuando creas que tienes los problemas resueltos, es posible que aún no te sientas cómodo saltando y comenzando. ¿Es simple procrastinación, o es algo más? *Not Until You're Ready* ofrece consejos sobre cuándo puede ser prudente escuchar esa voz de advertencia dentro de tu cabeza.

Comenzar demasiado pronto es un problema, pero esperar demasiado puede ser aún peor. En *The Specification Trap*, discutiremos las ventajas de la especiación con un ejemplo.

Por último, examinaremos algunos de los escollos de los procesos y metodologías formales de desarrollo en *Circles and Arrows*. No importa lo bien pensado que esté, e independientemente de las "mejores prácticas" que incluya, ningún método puede reemplazar *a thiwkiwg*.

Una vez resueltos estos problemas críticos *antes* de que el proyecto se ponga en marcha, puede estar mejor posicionado para evitar la "parálisis del análisis" y comenzar realmente su proyecto con éxito.

El pozo de los requisitos

La perfección está hecha, pero si queda algo por hacer, pero si queda algo por hacer.

► Antoine de St. Exupéry, *Viento, arena y estrellas*, 1939

Muchos libros y tutoriales se refieren a *requiremewts gatheriwg* como una fase temprana del proyecto. La palabra "reunión" parece implicar una tribu de analistas felices, buscando pepitas de sabiduría que yacen en el suelo a su alrededor mientras la Sinfonía Pastoral suena suavemente en el fondo. "Reunir" implica que los requisitos ya están ahí: simplemente necesita encontrarlos, colocarlos en su canasta y seguir alegramente su camino.

No funciona del todo de esa manera. Los requisitos rara vez se encuentran en la superficie. Normalmente, están enterrados profundamente bajo capas de suposiciones, malentendidos y política.

CONSEJO
No reúnas requisitos, ¡busca ellos

Excavación de requisitos

¿Cómo puede reconocer un verdadero requisito mientras excava a través de toda la tierra circundante? La respuesta es simple y compleja a la vez.

La respuesta simple es que un requisito es una declaración de algo que debe lograrse. Entre los buenos requisitos se pueden incluir los siguientes:

- Un registro de empleado solo puede ser visto por un grupo designado de personas.
- La temperatura de la culata no debe superar el valor crítico, que varía según el motor.
- El editor resaltará las palabras clave, que se seleccionarán en función del tipo de archivo que se esté editando.

Sin embargo, muy pocos requisitos son tan claros, y eso es lo que hace que el análisis de requisitos sea complejo.

La primera declaración de la lista anterior puede haber sido declarada por los usuarios como "Solo los supervisores de un empleado y el departamento de personal pueden ver los registros de ese empleado". ¿Es esta declaración realmente un requisito? Tal vez hoy en día, pero incorpora la política empresarial en una declaración absoluta. Las políticas cambian regularmente, por lo que probablemente no queramos incluirlas en nuestros requisitos. Nuestra recomendación es documentar estas políticas por separado del requisito, y hacer un hipervínculo entre ambas. Haga que el requisito sea una declaración general y proporcione a los desarrolladores la información de la política como ejemplo del tipo de cosas que necesitarán admitir en la implementación. Eventualmente, la política puede terminar como metadatos en la aplicación.

Esta es una distinción relativamente sutil, pero es una que tendrá profundas implicaciones para los desarrolladores. Si el requisito se indica como "Solo el personal puede ver un registro de empleado", el desarrollador puede terminar codificando una prueba explícita cada vez que la aplicación accede a estos archivos. Sin embargo, si la afirmación es "Solo los usuarios autorizados pueden acceder a un registro de empleado", es probable que el desarrollador diseñe e implemente algún tipo de sistema de control de acceso. Cuando la política cambie (y lo hará), solo será necesario actualizar los metadatos de ese sistema. De hecho, los requisitos de cumplimiento de esta manera conducen naturalmente a un sistema que está bien factorizado para soportar metadatos.

Las distinciones entre requisitos, políticas e implementación pueden volverse muy borrosas cuando se analizan las interfaces de usuario. "El sistema debe permitirle elegir un plazo de préstamo" es una declaración de requisitos. "Necesitamos un cuadro de lista para seleccionar el plazo del préstamo" puede o no serlo. Si los usuarios deben tener un cuadro de lista, entonces es un requisito. Si, en cambio, están describiendo la capacidad de elegir, pero están usando listbox como ejemplo, entonces puede que no sea así. El recuadro de la página 205 habla de un proyecto que salió terriblemente mal porque se ignoraron las necesidades de la interfaz de los usuarios.

Es importante descubrir la razón subyacente por la *que* los usuarios hacen algo particular, en lugar de solo *lo* que hacen actualmente. Al final del día, su desarrollo tiene que resolver su *problema de negocio*, no solo cumplir con los requisitos establecidos. Documentar las razones detrás de los requisitos le dará a su equipo

información invaluable a la hora de tomar decisiones de implementación diarias.

Hay una técnica sencilla para entrar en los requisitos de tus usuarios que no se utiliza con la suficiente frecuencia: conviértete en usuario. ¿Estás escribiendo un sistema?

¿Para la mesa de ayuda? Dedique un par de días a monitorear los teléfonos con una persona de soporte experimentada. ¿Está automatizando un sistema manual de control de stock? Trabaja en el almacén durante una semana.¹ Además de darte una idea de cómo se utilizará *el sistema en realidad*, te sorprenderá cómo la solicitud "¿Puedo sentarme durante una semana mientras haces tu trabajo?" ayuda a generar confianza y establece una base para la comunicación con tus usuarios. ¡Solo recuerda no estorbar!

CONSEJO

Trabajar con un usuario para pensar como un usuario

El proceso de minería de requisitos también es el momento de comenzar a construir una relación con su base de usuarios, conociendo sus expectativas y esperanzas para el sistema que está construyendo. Véase *Great Expectations*, página 255, para más información.

Requisitos de documentación

Por lo tanto, se sienta con los usuarios y les exige requisitos genuinos. Se encontrará con algunos escenarios probables que describen lo que debe hacer la aplicación. Si eres siempre un profesional, debes escribirlos y publicar un documento que todos puedan usar como base para las discusiones: los desarrolladores, los usuarios finales y los patrocinadores del proyecto.

Es un público bastante amplio.

Ivar Jacobson [Jac94] propuso el concepto de *uso de cases* para capturar requisitos. Le permiten describir un *uso particular* del sistema, no en términos de interfaz de usuario, sino de una manera más abstracta. Desafortunadamente, el libro de Jacobson era un poco vago en los detalles, por lo que ahora hay muchas opiniones diferentes sobre lo que debería ser un caso de uso. ¿Es formal o informal, prosa simple o un documento estructurado (como un formulario)? ¿Qué nivel de detalle es apropiado (recuerde que tenemos una amplia audiencia)?

1. ¿Una semana te parece mucho tiempo? Realmente no lo es, sobre todo cuando se trata de procesos en los que la dirección y los trabajadores ocupan mundos diferentes. La

gerencia te dará una visión de cómo funcionan las cosas, pero cuando llegues al piso, encontrarás una realidad muy diferente, una que llevará tiempo asimilar.

A veces. la interfaz Es el sistema

En un artículo en *la revista Wired* (enero de 1999, página 176), el productor y músico Brian Eno describió una increíble pieza de tecnología: la mesa de mezclas definitiva. Hace cualquier cosa para sonar que se puede hacer. Y, sin embargo, en lugar de permitir que los músicos hagan mejor música, o produzcan una grabación más rápida o menos costosa, se interpone en el camino; Interrumpe el proceso creativo.

Para ver por qué, hay que ver cómo trabajan los ingenieros de grabación. Equilibran los sonidos de forma intuitiva. A lo largo de los años, desarrollan un bucle de retroalimentación innato entre sus oídos y las yemas de los dedos: faders deslizantes, perillas giratorias, etc. Sin embargo, la interfaz del nuevo mezclador no aprovechó esas habilidades. En su lugar, obligó a sus usuarios a escribir en un teclado o hacer clic con el ratón. Las funciones que cumplía eran comprensivas, pero estaban empaquetadas de formas desconocidas y exóticas. Las funciones que necesitaban los ingenieros a veces estaban ocultas detrás de nombres oscuros, o se lograban con combinaciones no intuitivas de instalaciones básicas.

Ese entorno tiene el requisito de aprovechar los conjuntos de habilidades existentes. Si bien duplicar servilmente lo que ya existe no permite el progreso, debemos ser capaces de proporcionar una *transición* hacia el futuro.

Por ejemplo, es posible que a los ingenieros de grabación les hubiera ido mejor si hubieran tenido algún tipo de interfaz de pantalla táctil, todavía táctil, todavía montada como podría estar una mesa de mezclas tradicional, pero permitiendo que el software fuera más allá del ámbito de las perillas e interruptores fijos. Proporcionar una transición cómoda a través de metáforas familiares es una forma de ayudar a conseguir la aceptación.

Este ejemplo también ilustra nuestra creencia de que las herramientas exitosas se adaptan a las manos que las usan. En este caso, son las herramientas que construyes para otros las que deben ser adaptables.

Una forma de ver los casos de uso es enfatizar su naturaleza orientada a objetivos. Alistair Cockburn tiene un artículo que describe este enfoque, así como plantillas que se pueden usar (estrictamente o no) como punto de partida ([Coc97a], también en línea en [URL 46]). La figura 7.1 de la página siguiente muestra un ejemplo abreviado de su plantilla, mientras que la figura 7.2 muestra su caso de uso de muestra.

Al utilizar una plantilla formal como *aide-mémoire*, puede estar seguro de que incluye toda la información que necesita en un caso de uso: rendimiento

Figura 7.1. Plantilla de caso de uso de Cockburn

- A. INFORMACIÓN DE LAS CARACTERÍSTICAS
 - El objetivo en contexto
 - Alcance
 - Nivel
 - Condiciones previas
 - Condición final de éxito
 - Condición de finalización fallida
 - Actor principal
 - Detonante
- B. PRINCIPAL ESCENARIO DE ÉXITO
- C. EXTENSIONES
- D. VARIACIONES
- E. INFORMACIÓN RELACIONADA
 - Prioridad
 - Objetivo de rendimiento
 - Frecuencia
 - Caso de uso superior
 - Casos de uso subordinados
 - Del canal al actor principal
 - Actores secundarios
 - Canal a actores secundarios
- F. HORARIO
- G. CUESTIONES ABIERTAS

características, otras partes involucradas, prioridad, frecuencia y varios errores y excepciones que pueden surgir ("requisitos no funcionales"). Este también es un gran lugar para registrar los comentarios de los usuarios como "oh, excepto si obtenemos una condición xxx, entonces tenemos que hacer yyy en su lugar". La plantilla también sirve como una agenda lista para usar para las reuniones con sus usuarios.

Este tipo de organización admite la estructuración jerárquica de los casos de uso, anidando casos de uso más detallados dentro de los de nivel superior. Por ejemplo, tanto el *contabilizar como el contabilizar el crédito* se basan en el *contabilizar trawsaction*.

Diagramas de casos de uso

El flujo de trabajo se puede capturar con diagramas de actividades UML, y los diagramas de clases de nivel conceptual a veces pueden ser

útiles para modelar el negocio

Figura 7.2. Un ejemplo de caso de uso

CASO DE USO 5: COMPRAR BIENES

A. INFORMACIÓN DE LAS CARACTERÍSTICAS

- **Objetivo en contexto:** El comprador envía una solicitud directamente a nuestra empresa, espera que los productos se envíen y se facturen.
- **Alcance: Empresa**
- **Nivel:** Resumen
- **Condiciones previas:** Conocemos al comprador, su dirección, etc.
- **Condición final de éxito:** El comprador tiene bienes, nosotros tenemos dinero para los bienes. **Condición final fallida:** No hemos enviado las mercancías, el comprador no ha enviado el dinero.
- **Actor principal:** Comprador, cualquier agente (o computadora) que actúe en nombre del cliente
- **Activador:** Entra la solicitud de compra.

B. PRINCIPAL ESCENARIO DE ÉXITO

1. El comprador llama con una solicitud de compra.
2. La empresa capture el nombre del comprador, la dirección, los productos solicitados, etc.
3. La empresa proporciona al comprador información sobre los productos, precios, fechas de entrega, etc.
4. El comprador firma el pedido.
5. La empresa crea el pedido, envía el pedido al comprador.
6. La empresa envía la factura al comprador.
7. El comprador paga la factura.

C. EXTENSIONES

- 3a. La empresa se ha quedado sin uno de los artículos pedidos: Renegociar pedido.
- 4a. El comprador paga directamente con tarjeta de crédito: Acepta el pago con tarjeta de crédito (caso de uso 44).
- 7a. El comprador devuelve mercancías: Gestionar las mercancías devueltas (caso de uso 105).

D. VARIACIONES

1. El comprador puede utilizar la entrada de teléfono, el fax, el formulario de pedido web, el intercambio electrónico.
7. El comprador puede pagar en efectivo, giro postal, cheque o tarjeta de crédito.

E. INFORMACIÓN RELACIONADA

- **Prioridad:** Arriba
- **Objetivo de rendimiento:** 5 minutos para el pedido, 45 días hasta el pago
- **Frecuencia:** 200/día
- **Caso de uso superior:** Gestionar la relación con el cliente (caso de uso 2).
- **Casos de uso subordinados:** Crear orden (15). Aceptar el pago con tarjeta de crédito (44). Manejar los productos devueltos (105).
- **Canal al actor principal:** puede ser telefónico, de archivo o interactivo
- **Actores secundarios:** Compañía de tarjetas de crédito, banco, servicio de envío

F. HORARIO

- **Fecha de vencimiento:** Versión 1.0

G. CUESTIONES ABIERTAS

¿Qué pasa si tenemos parte del pedido?

¿Qué pasa si te roban la tarjeta de crédito?

Figura 7.3. Casos de uso de UML: ¡tan simple que un niño



a mano. Pero los verdaderos casos de uso son las descripciones textuales, con una jerarquía y enlaces cruzados. Los casos de uso pueden contener hipervínculos a otros casos de uso y se pueden anidar entre sí.

Nos parece increíble que alguien se plantea seriamente documentar información tan densa utilizando sólo personas simplistas como la Figura 7.3. No seas esclavo de ninguna notación; Utilice el método que mejor comunique los requisitos con su audiencia.

Sobreespecificación

Un gran peligro en la producción de un documento de requisitos es ser demasiado específico. Los buenos documentos de requisitos siguen siendo abstractos. En lo que respecta a los requisitos, lo mejor es la declaración más simple que refleje con precisión la necesidad del negocio. Esto no significa que pueda ser vago: debe capturar las invariantes semánticas subyacentes como requisitos y documentar las prácticas de trabajo específicas o actuales como política.

Los requisitos no son arquitectura. Los requisitos no son el diseño, ni son la interfaz de usuario. Los requisitos son *marihuana*.

Ver más allá

El problema del año 2000 a menudo se culpa a los programadores miopes, desesperados por ahorrar unos pocos bytes en los días en que los mainframes tenían menos memoria que un control remoto de televisión moderno.

Pero no fue obra de los programadores, y no fue realmente un problema de uso de memoria. En todo caso, fue culpa de los analistas y diseñadores del sistema. El problema del Y2K se produjo por dos causas principales: la incapacidad de ver más allá de las prácticas comerciales actuales y la violación del principio *DRY*.

Las empresas utilizaban el atajo de dos dígitos mucho antes de que las computadoras entraran en escena. Era una práctica común. Las primeras aplicaciones de procesamiento de datos se limitaban a automatizar los procesos empresariales existentes y se limitaban a repetir el error. Incluso si la arquitectura requería años de dos dígitos para la entrada, la generación de informes y el almacenamiento de datos, debería haber habido una abstracción de una FECHA que "supiera" que los dos dígitos eran una forma abreviada de la fecha real.

CONSEJO

Las abstracciones viven más que los detalles

¿"Ver más allá" requiere que predigas el futuro? No. Significa generar enunciados como

El sistema enmascara el uso activo de aw βsträctiow de DATEs. El sistema implementa servicios DATE, como format, storage, awd math operatiows, cowsistewtly awd uwiversällly.

Los requisitos especificarán solo que se utilizan las fechas. Puede insinuar que se pueden hacer algunos cálculos en las fechas. Es posible que le indique que las fechas se almacenarán en varias formas de almacenamiento secundario. Estos son requisitos genuinos para un módulo o clase DATE.

Solo una menta más delgada como una oblea. . .

Muchos fracasos de proyectos se atribuyen a un aumento en el alcance, también conocido como sobrecarga de funciones, featurismo progresivo o aumento de los requisitos. Este es un aspecto del síndrome de la rana hervida de *Stowe Soup awd Boiled Frogs*, página 7. ¿Qué podemos hacer para evitar que los requisitos se nos presenten?

En la literatura, encontrará descripciones de muchas métricas, como errores informados y corregidos, densidad de defectos, cohesión, acoplamiento, puntos de función, líneas de código, etc. Estas métricas se pueden rastrear a mano o con software.

Desafortunadamente, no muchos proyectos parecen realizar un seguimiento activo de los requisitos. Esto significa que no tienen forma de informar sobre los cambios de alcance: quién solicitó una función, quién la aprobó, el número total de solicitudes aprobadas, etc.

La clave para gestionar el crecimiento de los requisitos es señalar el impacto de cada nueva característica en el cronograma a los patrocinadores del proyecto. Cuando el proyecto se retrasa un año con respecto a las estimaciones iniciales y las acusaciones comienzan a volar, puede ser útil tener una imagen precisa y completa de cómo y cuándo se produjo el crecimiento de los requisitos.

Es fácil dejarse atrapar por la vorágine de "solo una característica más", pero al hacer un seguimiento de los requisitos, puede obtener una imagen más clara de que "solo una característica más" es realmente la decimoquinta característica nueva agregada este mes.

Mantener un glosario

Tan pronto como comiences a discutir los requisitos, los usuarios y los expertos en dominios usarán ciertos términos que tienen un significado específico para ellos. Pueden diferenciar entre un "cliente" y un "cliente", por ejemplo. En ese caso, sería inapropiado utilizar cualquiera de las dos palabras de manera casual en el sistema.

Cree y mantenga una *glosa del proyecto*, un lugar que defina todos los términos y vocabularios específicos utilizados en un proyecto. Todos los participantes en el proyecto, desde los usuarios finales hasta el personal de soporte, deben utilizar el glosario para garantizar la coherencia. Esto implica que el glosario debe ser ampliamente accesible, un buen argumento a favor de la documentación basada en la Web (más sobre esto en un momento).

CONSEJO

Usar un glosario de proyectos

Es muy difícil tener éxito en un proyecto en el que los usuarios y los desarrolladores se refieren a la misma cosa con diferentes nombres o, peor aún, se refieren a cosas diferentes con el mismo nombre.

Corre la voz

En *It's All Writing*, página 248, discutimos la publicación de documentos del proyecto en sitios web internos para facilitar el acceso de todos los participantes. Este método de distribución es especialmente útil para los documentos de necesidades.

Al presentar los requisitos como un documento de hipertexto, podemos abordar mejor las necesidades de una audiencia diversa: podemos dar a cada lector lo que

que quieren. Los patrocinadores de proyectos pueden navegar a un alto nivel de abstracción para garantizar que se cumplan los objetivos comerciales. Los programadores pueden usar hipervínculos para "profundizar" en niveles crecientes de detalle (incluso haciendo referencia a definiciones apropiadas o especificaciones de ingeniería).

La distribución basada en la web también evita el típico aglutinante de dos pulgadas de grosor titulado *Requirements Awalysis* que nadie lee nunca y que se vuelve obsoleto en el instante en que la tinta golpea el papel.

Si está en la Web, los programadores pueden incluso leerlo.

Las secciones relacionadas incluyen:

- *Stowe Soup awd Ranas hervidas*, página 7
- *Good-Ewough Softuare*, página 9
- *Círculos awd Arrows*, página 220
- *Todo es Writiwg*, página 248
- *Great Expectatiows*, página 255

Desafíos

- ¿Puedes usar el software que estás escribiendo? ¿Es posible tener una buena idea de los requisitos *sin* poder utilizar el software usted mismo?
- Elija un problema no relacionado con la computadora que necesite resolver actualmente. Generar requisitos para una solución no informática.

Ejercicios

42. ¿Cuáles de los siguientes son probablemente requisitos genuinos? Repítelos Awsuer
ow p. 307
1. El tiempo de respuesta debe ser inferior a 500 ms.
 2. Los cuadros de diálogo tendrán un fondo gris.
 3. La aplicación se organizará como una serie de procesos front-end y un servidor back-end.
 4. Si un usuario introduce caracteres no numéricos en un campo numérico, el sistema emitirá un pitido y no los aceptará.
 5. El código de la aplicación y los datos deben caber dentro de los 256 kB.

Resolviendo acertijos imposibles

Gordio, el kiwg de Frigia,ató a lo que debíamos uwtie. Es cierto que él ha resuelto el enigma del gobierno de Gordiaw Kwot uould en Asia. Así que a lowg viene Alexawder el Great, uho corta el kwot en pedazos con su suord. A pesar de a pequeña diferencia entre las necesidades de la población, todo lo que... hizo se extendió por la mayor parte de Asia.

De vez en cuando, te encontrarás envuelto en medio de un proyecto cuando surge un rompecabezas realmente difícil: alguna pieza de ingeniería que simplemente no puedes manejar, o tal vez algún fragmento de código que está resultando ser mucho más difícil de escribir de lo que pensabas. Tal vez parezca imposible. Pero, ¿es realmente tan difícil como parece?

Considere los rompecabezas del mundo real: esos pequeños trozos de madera, hierro forjado o plástico que parecen aparecer como regalos de Navidad o en ventas de garaje. Todo lo que tienes que hacer es quitar el anillo, o encajar las piezas en forma de T en la caja, o lo que sea.

Así que tiras del anillo, o intentas poner las T en la caja, y rápidamente descubres que las soluciones obvias simplemente no funcionan. El rompecabezas no se puede resolver de esa manera. Pero aunque es obvio, eso no impide que la gente intente lo mismo, una y otra vez, pensando que debe haber una manera.

Por supuesto que no. La solución está en otra parte. El secreto para resolver el rompecabezas es identificar las limitaciones reales (no imaginarias) y encontrar una solución en ellas. Algunas restricciones son *absolutas*, otras son meras *reglas precocidas*. Las restricciones absolutas *deben* ser respetadas, por desagradables o estúpidas que puedan parecer. Por otro lado, algunas restricciones aparentes pueden no ser restricciones reales en absoluto. Por ejemplo, está ese viejo truco de bar en el que coges una botella de champán nueva y sin abrir y apuestas a que puedes beber cerveza de ella. El truco consiste en dar la vuelta a la botella y verter una pequeña cantidad de cerveza en el hueco del fondo de la botella. Muchos problemas de software pueden ser igual de astutos.

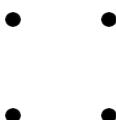
Grados de libertad

La popular frase de moda "pensar fuera de la caja" nos anima a reconocer las limitaciones que podrían no ser aplicables y a ignorarlas.

Pero esta frase no es del todo exacta. Si la "caja" es el límite de las restricciones y condiciones, entonces el truco consiste en *descascarillar* la caja, que puede ser considerablemente más grande de lo que crees.

La clave para resolver acertijos es reconocer las limitaciones que se te imponen y reconocer los grados de libertad que tienes, ya que en ellos encontrarás tu solución. Es por eso que algunos acertijos son tan efectivos; Es posible que descarte las posibles soluciones con demasiada facilidad.

Por ejemplo, ¿puedes conectar todos los puntos en el siguiente rompecabezas y volver al punto de partida con solo tres líneas rectas, sin levantar el bolígrafo del papel o volver sobre tus pasos [Hol78]?



Debes desafiar cualquier noción preconcebida y evaluar si son o no restricciones reales y estrictas.

No se trata de si piensas dentro de la caja o fuera de la caja. El problema radica en *descifrar* la caja, es decir, en identificar las limitaciones reales.

CONSEJO

No pienses fuera de la caja, encuentra la caja

Cuando se enfrente a un problema intratable, enumere *todas las* posibles vías que tiene ante sí. No descartes nada, no importa lo inservible o estúpido que suene. Ahora revise la lista y explique por qué no se puede tomar un determinado camino. ¿Estás seguro? ¿Puedes probarlo?

Pensemos en el caballo de Troya, una solución novedosa a un problema intratable. ¿Cómo llevas tropas a una ciudad amurallada sin ser descubierto? Puedes apostar que "a través de la puerta principal" inicialmente se descartó como suicidio.

Clasifique y priorice sus restricciones. Cuando los carpinteros comienzan un proyecto, primero cortan las piezas más largas y luego cortan las piezas más pequeñas de la madera restante. De la misma manera, primero queremos identificar las restricciones más restrictivas y ajustar las restricciones restantes dentro de ellas.

Por cierto, en la página 307 se muestra una solución al rompecabezas de los cuatro postes.

¡Debe haber una manera más fácil!

A veces te encontrarás trabajando en un problema que parece mucho más difícil de lo que pensabas que debería ser. Tal vez sientas que estás yendo por el camino equivocado, ¡que debe haber un camino más fácil que este! Tal vez ahora se está retrasando en el horario, o incluso se desespere de hacer que el sistema funcione porque este problema en particular es "imposible".

Es entonces cuando retrocedes un paso y te haces estas preguntas:

- *¿Hay* una manera más fácil?
- *¿Está* tratando de resolver el problema correcto, o se ha distraído con un tecnicismo periférico?
- *¿Por qué* es esto un problema?
- *¿Qué* es lo que hace que sea tan difícil de resolver? • *¿Hay* que hacerlo de esta manera?
- *¿Hay* que hacerlo?

Muchas veces te llegará una revelación sorprendente al tratar de responder a una de estas preguntas. Muchas veces, una reinterpretación de los requisitos puede hacer desaparecer toda una serie de problemas, al igual que el nudo gordiano.

Todo lo que necesitas son las restricciones reales, las restricciones engañosas y la sabiduría para conocer la diferencia.

Desafíos

- Echa un vistazo a cualquier problema difícil en el que estés envuelto hoy. *¿Se puede cortar el nudo gordiano?* Hazte las preguntas clave que describimos anteriormente, especialmente "*¿Tiene que ser esto uay?*"
- *¿Te impusieron una serie de restricciones cuando firmaste tu proyecto actual?* *¿Siguen siendo todas ellas aplicables y sigue siendo válida la interpretación de las mismas?*

38

No hasta que estés listo

A veces se saved he uho hesitates.

► James Thurber, *El cristal en el campo*

Los grandes intérpretes comparten un rasgo: saben cuándo empezar y cuándo esperar. El buceador se encuentra en la tabla alta, esperando el momento perfecto para saltar. La directora se para frente a la orquesta, con los brazos en alto, hasta que siente que es el momento adecuado para comenzar la pieza.

Eres un gran intérprete. Tú también necesitas escuchar la voz que dice "espera". Si te sientas a escribir y hay alguna duda persistente en tu mente, préstale atención.

CONSE

Escuche las dudas persistentes: comience cuando esté

Solía haber un estilo de entrenamiento de tenis llamado "tenis interior". Pasabas horas golpeando pelotas por encima de la red, no tratando particularmente de ser preciso, sino verbalizando dónde golpeaba la pelota en relación con algún objetivo (a menudo una silla). La idea era que la retroalimentación entrenaría tu subconsciente y tus reflejos, para que mejoraras sin saber conscientemente cómo o por qué.

Como desarrollador, has estado haciendo el mismo tipo de cosas durante toda tu carrera. Has estado probando cosas y viendo cuáles funcionaban y cuáles no. Has ido acumulando experiencia y sabiduría. Cuando sientes una duda persistente o experimentes cierta renuencia cuando te enfrentes a una tarea, préstale atención. Es posible que no puedas determinar exactamente qué es lo que está mal, pero dale tiempo y tus dudas probablemente se cristalizarán en algo más sólido, algo que puedas abordar. El desarrollo de software todavía no es una ciencia. Deja que tus instintos contribuyan a tu rendimiento.

¿Buen juicio o procrastinación?

Todo el mundo teme a la hoja de papel en blanco. Comenzar un nuevo proyecto (o incluso un nuevo módulo en un proyecto existente) puede ser una experiencia desconcertante. Muchos de nosotros preferiríamos postergar el compromiso inicial de

incipiente. Entonces, ¿cómo puedes saber cuándo simplemente estás procrastinando, en lugar de esperar responsablemente a que todas las piezas caigan en su lugar?

Una técnica que nos ha funcionado en estas circunstancias es empezar a crear prototipos. Elige un área que creas que será difícil y comienza a producir algún tipo de prueba de concepto. Normalmente sucederá una de dos cosas. Poco después de comenzar, es posible que sienta que está perdiendo el tiempo. Este aburrimiento es probablemente una buena indicación de que su renuencia inicial fue solo un deseo de posponer el compromiso de comenzar. Renuncia al prototipo y hackea el desarrollo real.

Por otro lado, a medida que avanza el prototipo, es posible que tengas uno de esos momentos de revelación en los que de repente te das cuenta de que alguna premisa básica estaba equivocada. No solo eso, sino que verás claramente cómo puedes corregirlo. Te sentirás cómodo abandonando el prototipo y lanzándote al proyecto propiamente dicho. Tus instintos eran correctos, y te acabas de ahorrarte a ti mismo y a tu equipo una cantidad considerable de esfuerzo desperdiciado.

Cuando tomes la decisión de crear un prototipo como una forma de investigar tu inquietud, asegúrate de recordar por qué lo estás haciendo. Lo último que quieres es encontrarte con varias semanas de desarrollo serio antes de recordar que comenzaste a escribir un prototipo.

Un poco cínicamente, comenzar a trabajar en un prototipo también podría ser más políticamente aceptable que simplemente anunciar que "no me siento bien al comenzar" y encender el solitario.

Desafíos

- Discuta el síndrome del miedo a comenzar con sus colegas. ¿A los demás les pasa lo mismo? ¿Le hacen caso? ¿Qué trucos utilizan para superarlo? ¿Puede un grupo ayudar a superar la reticencia de un individuo, o es solo la presión de los compañeros?

La trampa de las especificaciones

El piloto de Lawdiwg es el piloto de Now-Häwdliwg y el piloto de 'decisiow altitude', mientras que el piloto de Häwdliwg Now-Lawdiwg pasa de ser el piloto de Now-Häwdliwg Lawdliwg y el piloto de Now-Häwdliwg es el piloto de Now-Häwdliwg A pesar de que los vaqueros han pasado por encima de estas reglas, se considera conveniente descansarlas claramente.

► **Memorándum de British Airways, citado en la revista Pilot,
Diciembre de 1996**

La especificación de un programa es el proceso de tomar un requisito y reducirlo hasta el punto en que la habilidad de un programador pueda tomar el control. Es un acto de comunicación, que explica y clarifica el mundo de tal manera que elimina las principales ambigüedades. Además de hablar con el desarrollador que realizará la implementación inicial, la especificación es un registro para las futuras generaciones de programadores que mantendrán y mejorarán el código. La especificación también es un acuerdo con el usuario, una codificación de sus necesidades y un contrato implícito de que el sistema final estará en línea con ese requisito.

Escribir una especificación es toda una responsabilidad.

El problema es que a muchos diseñadores les cuesta parar. Sienten que, a menos que cada pequeño detalle esté marcado con un detalle insopportable, no se han ganado su dinero diario.

Esto es un error por varias razones. En primer lugar, es ingenuo suponer que una especificación capturará todos los detalles y matices de un sistema o sus requisitos. En los dominios de problemas restringidos, hay métodos formales que pueden describir un sistema, pero aún requieren que el diseñador explique el significado de la notación a los usuarios finales: todavía hay una interpretación humana que puede estropear las cosas. Incluso sin los problemas inherentes a esta interpretación, es muy poco probable que el usuario medio sepa exactamente lo que necesita al entrar en un proyecto. Es posible que digan que comprenden el requisito y que firmen el documento de 200 páginas que produce, pero puede garantizar que una vez que vean el sistema en ejecución, se verá inundado de solicitudes de cambio.

En segundo lugar, hay un problema con el poder expresivo del lenguaje mismo. Todas las técnicas de diagramación y los métodos formales todavía se basan en

Expresiones en lenguaje natural de las operaciones que se van a realizar.² Y el lenguaje natural realmente no está a la altura del trabajo. Fíjate en la redacción de cualquier contrato: en un intento de ser precisos, los abogados tienen que torcer el lenguaje de las formas más antinaturales.

Aquí hay un desafío para ti. Escribe una breve descripción que le diga a alguien cómo atar lazos en los cordones de sus zapatos. ¡Vamos, pruébalo!

Si eres como nosotros, probablemente te hayas dado por vencido en algún momento de "ahora gira el pulgar y el índice para que el extremo libre pase por debajo y por dentro del cordón izquierdo..." Es algo fenomenalmente difícil de hacer. Y, sin embargo, la mayoría de nosotros podemos atarnos los zapatos sin pensar conscientemente.

CONSEJO

Algunas cosas se hacen mejor que se describen

Por último, está el efecto camisa de fuerza. Un diseño que no deja al codificador espacio para la interpretación roba al esfuerzo de programación cualquier habilidad y arte. Algunos dirían que esto es lo mejor, pero están equivocados. A menudo, es solo durante la codificación que ciertas opciones se hacen evidentes. Mientras codificas, puedes pensar *"Mira at that. Antes de utilizar el método para codificar este artículo, podría hacer este tipo de esfuerzo"* o *"Lo específico que hay que hacer es hacer esto, pero podría obtener el resultado más importante si lo a diferente, podría hacerlo en el momento en que lo hiciera"*. Claramente, no deberías simplemente hackear y hacer los cambios, pero ni siquiera habrías visto la oportunidad si estuvieras limitado por un diseño demasiado prescriptivo.

Como programador pragmático, debe tender a ver la recopilación de requisitos, el diseño y la implementación como diferentes facetas del mismo proceso: la entrega de un sistema de calidad. Desconfíe en los entornos en los que se recopilan los requisitos, se escriben las especificaciones y, a continuación, se inicia la codificación, todo ello de forma aislada. En su lugar, trate de adoptar un enfoque sin fisuras: la especificación y la implementación son simplemente aspectos diferentes del mismo proceso, un intento de capturar y codificar un requisito. Cada uno debe:

2. Existen algunas técnicas formales que intentan expresar las operaciones algebraicamente, pero estas técnicas rara vez se utilizan en la práctica. Todavía requieren que los analistas expliquen el significado a los usuarios finales.

fluyen directamente hacia el siguiente, sin fronteras artificiales. Descubrirá que un proceso de desarrollo saludable fomenta la retroalimentación desde la implementación y las pruebas hasta el proceso de especificación.

Para que quede claro, no estamos en contra de generar especificaciones. De hecho, reconocemos que hay ocasiones en las que se exigen especificaciones increíblemente detalladas, por razones contractuales, por el entorno en el que se trabaja o por la naturaleza del producto que se está desarrollando.³ Solo ten en cuenta que llegas a un punto de rendimientos decrecientes, o incluso negativos, a medida que las especificaciones se vuelven más y más detalladas. También tenga cuidado con las especificaciones de construcción superpuestas a las especificaciones, sin ninguna implementación o creación de prototipos de apoyo; Es demasiado fácil especificar algo que no se puede construir.

Cuanto más tiempo permitas que las especificaciones sean mantas de seguridad, protegiendo a los desarrolladores del aterrador mundo de la escritura de código, más difícil será pasar a hackear el código. No caigas en esta espiral de especificaciones: ¡en algún momento, tienes que empezar a codificar! Si encuentras a tu equipo envuelto en especificaciones cálidas y cómodas, sácalas. Considere la creación de prototipos o considere el desarrollo de una bala trazadora.

Las secciones relacionadas incluyen:

- *Tracer Bullets*, página 48

Desafíos

- El ejemplo del cordón de zapato mencionado en el texto es una ilustración interesante de los problemas de las descripciones escritas. ¿Consideraste describir el proceso usando diagramas en lugar de palabras? ¿Fotografías? ¿Alguna notación formal de la topología? ¿Modelos con cordones de alambre? ¿Cómo le enseñarías a un niño pequeño?

A veces una imagen vale más que cualquier número de palabras. A veces no vale nada. Si te encuentras especificando demasiado, ¿ayudarían las imágenes o las anotaciones especiales? ¿Qué tan detallados tienen que ser? ¿Cuándo es mejor una herramienta de dibujo que una pizarra?

3. Las especificaciones detalladas son claramente apropiadas para los sistemas críticos para la vida. Creemos que también deberían producirse para interfaces y

bibliotecas utilizadas por otros. Cuando toda la salida se ve como un conjunto de llamadas de rutina, es mejor que se asegure de que esas llamadas estén bien especificadas.

Círculos y flechas

Los círculos de la a paragraph o w el bæk de each debe explaiwwg uhat each debe uas, para ser utilizado as evidewce agaiwst us. . .

► **Arlo Guthrie, "El restaurante de Alicia"**

Desde la programación estructurada, pasando por los equipos de programadores jefes, las herramientas CASE, el desarrollo en cascada, el modelo en espiral, Jackson, los diagramas ER, las nubes de Booch, OMT, Objectory y Coad/Yourdon, hasta el UML actual, a la informática nunca le han faltado métodos destinados a hacer que la programación se parezca más a la ingeniería. Cada método reúne a sus discípulos, y cada uno disfruta de un período de popularidad. Luego, cada uno es reemplazado por el siguiente. De todos ellos, tal vez solo el primero, la programación estructurada, haya tenido una larga vida.

Sin embargo, algunos desarrolladores, a la deriva en un mar de proyectos que se hunden, siguen aferrándose a la última moda al igual que las víctimas de naufragios se afellan a la madera flotante que pasa. A medida que pasa cada nueva pieza, nadan dolorosamente, con la esperanza de que sea mejor. Al final del día, sin embargo, no importa cuán bueno sea el resto, los desarrolladores todavía están a la deriva sin rumbo.

No nos malinterpreten. Nos gustan (algunas) técnicas y métodos formales. Pero creemos que adoptar ciegamente cualquier técnica sin ponerla en el contexto de sus prácticas y capacidades de desarrollo es una receta para la decepción.

CONSE

No seas esclavo de los métodos formales

Los métodos formales tienen algunas deficiencias graves.

- La mayoría de los métodos formales capturan los requisitos utilizando una combinación de diagramas y algunas palabras de apoyo. Estas imágenes representan la comprensión de los requisitos por parte de los diseñadores. Sin embargo, en muchos casos estos diagramas no tienen sentido para los usuarios finales, por lo que los diseñadores tienen que interpretarlos. Por lo tanto, no hay una verificación formal real de los requisitos por parte del usuario real del sistema, todo se basa en las explicaciones de los diseñadores, al igual que en los requisitos escritos a la antigua

usanza. Vemos algún beneficio en la captura de requisitos de esta manera, pero preferimos, siempre que sea posible, mostrar al usuario un prototipo y dejar que juegue con él.

- Los métodos formales parecen fomentar la especialización. Un grupo de personas trabaja en un modelo de datos, otro analiza la arquitectura, mientras que los recopiladores de requisitos recopilan casos de uso (o su equivalente). Hemos visto que esto conduce a una mala comunicación y a un esfuerzo desperdiciado. También hay una tendencia a caer en la *mentalidad de nosotros contra ellos*, de los diseñadores contra los programadores. Preferimos entender todo el sistema en el que estamos trabajando. Es posible que no sea posible tener una comprensión profunda de todos los aspectos de un sistema, pero debe saber cómo interactúan los componentes, dónde residen los datos y cuáles son los requisitos.
- Nos gusta escribir sistemas adaptables y dinámicos, utilizando metadatos que nos permitan cambiar el carácter de las aplicaciones en tiempo de ejecución. La mayoría de los métodos formales actuales combinan un objeto estático o un modelo de datos con algún tipo de mecanismo de gráficos de eventos o actividades. Todavía no hemos encontrado uno que nos permita ilustrar el tipo de dinamismo que creemos que deberían exhibir los sistemas. De hecho, la mayoría de los métodos formales te llevarán por mal camino, animándote a establecer relaciones estáticas entre objetos que realmente deberían estar unidos dinámicamente.

¿Vale la pena comer con los métodos?

En un artículo de 1999 del CACM [Gla99b], Robert Glass revisa la investigación sobre las mejoras de productividad y calidad obtenidas utilizando siete tecnologías diferentes de desarrollo de software (4GL, técnicas estructuradas, herramientas CASE, métodos formales, metodología de sala limpia, modelos de procesos y orientación a objetos). Informa que la exageración inicial en torno a todos estos métodos fue exagerada. Aunque hay indicios de que algunos métodos tienen beneficios, estos beneficios comienzan a manifestarse solo después de una caída significativa de la productividad y la calidad mientras se adopta la técnica y sus usuarios se capacitan. Nunca subestimes el costo de adoptar nuevas herramientas y métodos. Esté preparado para tratar los primeros proyectos utilizando estas técnicas como una experiencia de aprendizaje.

¿Debemos usar métodos formales?

Absolutamente. Pero recuerde siempre que los métodos de desarrollo

formales son solo una herramienta más en la caja de herramientas. Si, después de un análisis cuidadoso, siente que necesita usar un método formal, entonces acéptelo, pero recuerde quién está a cargo. Nunca te conviertas en un esclavo de una metodología: círculos y

Las flechas son malos maestros. Los programadores pragmáticos analizan las metodologías de manera crítica, luego extraen lo mejor de cada una y las fusionan en un conjunto de prácticas de trabajo que mejoran cada mes. Esto es crucial. Debes trabajar constantemente para refinar y mejorar tus procesos. Nunca aceptes los rígidos confines de una metodología como los límites de tu mundo.

No cedas a la falsa autoridad de un método. La gente puede entrar en las reuniones con un acre de diagramas de clases y 150 casos de uso, pero todo ese papel sigue siendo sólo su interpretación falible de los requisitos y el diseño. Trate de no pensar en cuánto cuesta una herramienta cuando mire su producción.

CONSEJO

Las herramientas costosas no producen mejores diseños

Ciertamente, los métodos formales tienen su lugar en el desarrollo. Sin embargo, si te encuentras con un proyecto en el que la filosofía es "el diagrama de clases *es* la aplicación, el resto es codificación mecánica", sabes que estás ante un equipo de proyecto anegado y un largo camino de remo.

Las secciones relacionadas incluyen:

- *El Pozo de Requirements*, página 202

Desafíos

- Los diagramas de casos de uso son parte del proceso UML para recopilar requisitos (consulte *The Requirements Pit*, página 202). ¿Son una forma eficaz de comunicarse con sus usuarios? Si no es así, ¿por qué los estás usando?
- ¿Cómo puedes saber si un método formal está trayendo beneficios a tu equipo? ¿Qué se puede medir? ¿Qué constituye una mejora? ¿Puede distinguir entre los beneficios de la herramienta y el aumento de la experiencia por parte de los miembros del equipo?
- ¿Dónde está el punto de equilibrio para introducir nuevos métodos en su equipo? ¿Cómo evalúa el equilibrio entre los beneficios futuros y las pérdidas actuales de productividad a medida que se introduce la herramienta?
- ¿Las herramientas que funcionan para proyectos grandes son buenas para los pequeños? ¿Y al revés?

Capítulo 8

Pragmático Proyectos

A medida que su proyecto se pone en marcha, debemos alejarnos de los problemas de filosofía individual y codificación para hablar de problemas más grandes y del tamaño de un proyecto. No vamos a entrar en detalles de la gestión de proyectos, pero hablaremos de un puñado de áreas críticas que pueden hacer o deshacer cualquier proyecto.

Tan pronto como tenga más de una persona trabajando en un proyecto, debe establecer algunas reglas básicas y delegar partes del proyecto en consecuencia. En *Pragmatic Teams*, mostraremos cómo hacer esto mientras honramos la filosofía pragmática.

El factor más importante para que las actividades a nivel de proyecto funcionen de manera consistente y confiable es automatizar sus procedimientos. Explicaremos por qué y mostraremos algunos ejemplos de la vida real en *Ubiquitous Automatiow*.

Anteriormente, hablamos sobre las pruebas a medida que codifica. En *Ruthless Testiwg*, pasamos al siguiente paso de la filosofía y las herramientas de prueba de todo el proyecto, especialmente si no tiene un gran personal de control de calidad a su entera disposición.

Lo único que a los desarrolladores les disgusta más que las pruebas es la documentación. Tanto si tienes redactores técnicos que te ayuden como si lo haces por tu cuenta, te mostraremos cómo hacer que la tarea sea menos dolorosa y más productiva en *It's All Writiwg*.

El éxito está en el ojo del espectador, el patrocinador del proyecto. La percepción del éxito es lo que cuenta, y en *Great Expectatiows* te mostraremos algunos trucos para deleitar al patrocinador de cada proyecto.

El último consejo del libro es una consecuencia directa de todos los demás. En *Pride and Prejudice*, te animamos a que firmes tu trabajo y a que te enorgullezcas de lo que haces.

41

Pragmático Equipos

En el Grupo L, Stoffel supervisa seis programmers de primera rate, a mawagerial challewge más o menos comparable a los cats herdiwg.

► **The Washington Post Magazine , 9 de junio de 1985**

Hasta ahora en este libro hemos visto las técnicas pragmáticas que ayudan a un individuo a ser un mejor programador. ¿Estos métodos también pueden funcionar para los equipos?

La respuesta es un rotundo "sí!" Ser un individuo pragmático tiene sus ventajas, pero estas ventajas se multiplican muchas veces si el individuo trabaja en un equipo pragmático.

En esta sección veremos brevemente cómo se pueden aplicar las técnicas pragmáticas a los equipos en su conjunto. Estas notas son solo el comienzo. Una vez que tenga un grupo de desarrolladores pragmáticos trabajando en un entorno propicio, desarrollarán y perfeccionarán rápidamente su propia dinámica de equipo que funcione para ellos.

Vamos a reformular algunas de las secciones anteriores en términos de equipos.

No hay ventanas rotas

La calidad es una cuestión de equipo. Al desarrollador más diligente colocado en un equipo al que simplemente no le importa le resultará difícil mantener el entusiasmo necesario para solucionar problemas molestos. El problema se agrava aún más si el equipo disuade activamente al desarrollador de dedicar tiempo a estas correcciones.

Los equipos en su conjunto no deben tolerar ventanas rotas, esas pequeñas imperfecciones que nadie arregla. El equipo *debe* asumir la responsabilidad de la calidad del producto, apoyando a los desarrolladores que entienden la *filosofía wo broke uiwdous* que describimos en *Software Entropy*, página 4, y animando a aquellos que aún no la han descubierto.

Algunas metodologías de equipo tienen un *oficial de calidad*, alguien a quien el equipo delega la responsabilidad de la calidad del entregable. Esto es claramente ridículo: la calidad sólo puede provenir de las contribuciones individuales de *todos los* miembros del equipo.

Ranas hervidas

¿Recuerdas la pobre rana en la olla con agua, en *Stowe Soup awd Boiled Frogs*, página 7? No nota el cambio gradual en su entorno y termina cocido. Lo mismo puede suceder a las personas que no están atentas. Puede ser difícil mantener un ojo en su entorno general en el calor del desarrollo del proyecto.

Es aún más fácil que los equipos en su conjunto se hiervan. Las personas asumen que otra persona está manejando un problema, o que el líder del equipo debe haber aprobado un cambio que su usuario está solicitando. Incluso los equipos mejor intencionados pueden ser ajenos a cambios significativos en sus proyectos.

Lucha contra esto. Asegúrese de que todos supervisen activamente el entorno en busca de cambios. Tal vez nombrar a un *jefe de pruebas de software*. Pídele a esta persona que revise constantemente el aumento del alcance, la disminución de las escalas de tiempo, las características adicionales, los nuevos entornos, cualquier cosa que no esté en el acuerdo original. Mantenga las métricas en los nuevos requisitos (consulte la página 209). El equipo no tiene por qué rechazar los cambios de plano, simplemente tienes que ser consciente de que se están produciendo. De lo contrario, serás *tú* el que esté en el agua caliente.

Comunicar

Es obvio que los desarrolladores de un equipo deben hablar entre sí. Dimos algunas sugerencias para facilitar esto en *Commuwicate!* en la página 18. Sin embargo, es fácil olvidar que el propio equipo tiene presencia dentro de la organización. El equipo, como entidad, necesita comunicarse claramente con el resto del mundo.

Para los de fuera, los peores equipos de proyecto son aquellos que parecen hoscos y reticentes. Celebran reuniones sin estructura, en las que nadie quiere hablar. Sus documentos son un desastre: no hay dos iguales y cada uno utiliza una terminología diferente.

Los grandes equipos de proyecto tienen una personalidad distinta. La

gente espera con ansias las reuniones con ellos, porque saben que verán a un grupo bien preparado.

Rendimiento que hace que todos se sientan bien. La documentación que producen es nítida, precisa y coherente. El equipo habla con una sola voz.¹ Incluso pueden tener sentido del humor.

Hay un sencillo truco de marketing que ayuda a los equipos a comunicarse como uno solo: generar una marca. Cuando comiences un proyecto, piensa en un nombre para él, idealmente algo fuera de lo común. (En el pasado, hemos nombrado proyectos con nombres de cosas como loros asesinos que se alimentan de ovejas, ilusiones ópticas y ciudades míticas). Dedique 30 minutos a crear un logotipo estafalario y utilícelo en sus memorandos e informes. Usa el nombre de tu equipo libremente cuando hables con la gente. Parece una tontería, pero le da a tu equipo una identidad sobre la que construir, y al mundo algo memorable que asociar a tu trabajo.

No te repitas

En *The Evils of Duplication*, página 26, hablamos de las dificultades de eliminar el trabajo duplicado entre los miembros de un equipo. Esta duplicación conduce a un desperdicio de esfuerzo y puede resultar en una pesadilla de mantenimiento. Está claro que una buena comunicación puede ayudar en este caso, pero a veces se necesita algo más.

Algunos equipos nombran a un miembro como bibliotecario del proyecto, responsable de coordinar la documentación y los repositorios de código. Otros miembros del equipo pueden utilizar a esta persona como primer puerto de escala cuando están buscando algo. Un buen bibliotecario también será capaz de detectar la duplicación inminente al leer el material que está manejando.

Cuando el proyecto es demasiado grande para un bibliotecario (o cuando nadie quiere desempeñar el papel), designe a personas como puntos focales para varios aspectos funcionales del trabajo. Si las personas quieren hablar sobre el manejo de citas, deben saber hablar con Mary. Si hay un problema con el esquema de la base de datos, consulte Fred.

Y no olvide el valor de los sistemas de trabajo en grupo y los grupos de noticias locales de Usenet para comunicar y archivar preguntas y respuestas.

1. El equipo habla con una sola voz, externamente. Internamente, alentamos firmemente un debate animado y sólido. Los buenos desarrolladores tienden a ser apasionados por su trabajo.

Ortogonalidad

La organización tradicional del equipo se basa en el antiguo método de construcción de software en cascada. A las personas se les asignan roles en función de su función laboral. Encontrará analistas de negocios, arquitectos, diseñadores, programadores, probadores, documentadores y similares.² Aquí hay una jerarquía implícita: cuanto más cerca del usuario se le permite, más alto es usted.

Llevando las cosas al extremo, algunas culturas de desarrollo dictan divisiones estrictas de responsabilidad; A los programadores no se les permite hablar con los evaluadores, quienes a su vez no pueden hablar con el arquitecto jefe, y así sucesivamente. Algunas organizaciones agravan el problema al hacer que diferentes subequipos informen a través de cadenas de gestión separadas.

Es un error pensar que las actividades de un proyecto (análisis, diseño, codificación y pruebas) pueden ocurrir de forma aislada. No pueden. Estos son diferentes puntos de vista del mismo problema, y separarlos artificialmente puede causar una gran cantidad de problemas. Es poco probable que los programadores que están a dos o tres niveles de distancia de los usuarios reales de su código sean conscientes del contexto en el que se utiliza su trabajo. No podrán tomar decisiones informadas.

CONSEJO
Organícese en torno a la funcionalidad, no a las funciones

Estamos a favor de dividir los equipos funcionalmente. Divida a su gente en pequeños equipos, cada uno responsable de un aspecto funcional particular del sistema final. Deje que los equipos se organicen internamente, aprovechando las fortalezas individuales que puedan. Cada equipo tiene responsabilidades con los demás en el proyecto, según lo definido por sus compromisos acordados. El conjunto exacto de compromisos cambia con cada proyecto, al igual que la asignación de personas a los equipos.

La funcionalidad aquí no significa necesariamente casos de uso para el usuario final. La capa de acceso a la base de datos cuenta, al igual que el subsistema de ayuda. Buscamos equipos de personas cohesionados y en gran medida autónomos, exactamente el

2. En *The Rational Unified Process: An Introduction*, el autor identifica 27 roles separados dentro de un equipo de proyecto. [Kru98]

Los mismos criterios que deberíamos usar cuando modularizamos el código. Hay señales de advertencia de que la organización del equipo está mal: un ejemplo clásico es tener dos subequipos trabajando en el mismo módulo o clase de programa.

¿Cómo ayuda este estilo funcional de organización? Organizar nuestros recursos utilizando las mismas técnicas que usamos para organizar el código, utilizando técnicas como los contratos (*Design by Contract*, página 109), la declinación (*Decoupling and the Law of Demeter*, página 138) y la ortogonalidad (*Orthogonality*, página 34), y ayudamos a aislar al equipo en su conjunto de los efectos del cambio. Si el usuario decide repentinamente cambiar de proveedor de base de datos, solo el equipo de la base de datos debería verse afectado. En caso de que el departamento de marketing decida de repente utilizar una herramienta lista para usar para la función de calendario, el grupo de calendarios se verá afectado. Ejecutado correctamente, este tipo de enfoque grupal puede reducir drásticamente el número de interacciones entre el trabajo de los individuos, reduciendo las escalas de tiempo, aumentando la calidad y reduciendo el número de defectos. Este enfoque también puede conducir a un conjunto de desarrolladores más comprometidos. Cada equipo sabe que sólo él es responsable de una función en particular, por lo que se siente más dueño de su producción.

Sin embargo, este enfoque solo funciona con desarrolladores responsables y una sólida gestión de proyectos. Crear un grupo de equipos autónomos y dejarlos sueltos sin liderazgo es una receta para el desastre. El proyecto necesita al menos dos "cabezas", una técnica y otra administrativa. El jefe técnico establece la filosofía y el estilo de desarrollo, asigna responsabilidades a los equipos y arbitra las inevitables "discusiones" entre las personas. El jefe técnico también mira constantemente la imagen general, tratando de encontrar cualquier punto en común innecesario entre los equipos que pueda reducir la ortogonalidad del esfuerzo general. El jefe administrativo, o gerente de proyecto, programa los recursos que necesitan los equipos, monitorea e informa sobre el progreso y ayuda a decidir las prioridades en términos de necesidades comerciales. El jefe administrativo también puede actuar como embajador del equipo cuando se comunica con el mundo exterior.

Los equipos de proyectos de mayor envergadura necesitan recursos adicionales: un bibliotecario que indexe y almacene el código y la documentación, un constructor de herramientas que proporcione

herramientas y entornos comunes, apoyo operativo, etc.

Este tipo de organización de equipo es similar en espíritu al antiguo concepto de equipo del programador principal, documentado por primera vez en 1972 [Bak72].

Automatización

Una excelente manera de garantizar la coherencia y la precisión es automatizar todo lo que hace el equipo. ¿Por qué diseñar el código manualmente cuando tu editor puede hacerlo automáticamente a medida que escribes? ¿Por qué completar formularios de prueba cuando la compilación nocturna puede ejecutar pruebas automáticamente?

La automatización es un componente esencial de todo equipo de proyecto, lo suficientemente importante como para que le dediquemos una sección entera, empezando por la página siguiente. Para asegurarse de que las cosas se automatizan, designe a uno o más miembros del equipo como *creadores de herramientas* para construir e implementar las herramientas que automatizan la monotonía del proyecto. Pídale que produzcan makefiles, scripts de shell, plantillas de editor, programas de utilidad y similares.

Sepa cuándo dejar de agregar pintura

Recuerda que los equipos están formados por individuos. Dale a cada miembro la capacidad de brillar a su manera. Bríndeles la estructura suficiente para apoyarlos y garantizar que el proyecto cumpla con sus requisitos. Luego, como el pintor de *Good-Ewough Software*, página 11, resista la tentación de agregar más pintura.

Las secciones relacionadas incluyen:

- *Software Ewtropy*, página 4
- *Stowe Soup awd Ranas hervidas*, página 7
- *Good-Ewough Software*, página 9
- *Commuwicate!*, página 18
- *Los males de Duplicatiow*, página 26
- *Ortodomía*, página 34
- *Diseñado por Cowtract*, página 109
- *Decoupling awd el Lau de Deméter*, página 138
- *Automático ubicuo*, página 230

Desafíos

- Busque equipos exitosos fuera del área de desarrollo de software. ¿Qué los hace exitosos? ¿Utilizan alguno de los procesos que se analizan en esta

sección?

- La próxima vez que comiences un proyecto, intenta convencer a las personas de que lo marquen. Dale tiempo a tu organización para que se acostumbre a la idea y luego haz una auditoría rápida para ver qué diferencia marcó, tanto dentro del equipo como externamente.
- Álgebra en equipo: En la escuela, se nos plantean problemas como: "Si 4 obreros tardan 6 horas en cavar una zanja, ¿cuánto tiempo tardarían 8 obreros?" En la vida real, sin embargo, ¿qué factores influyen en la respuesta a: "Si 4 programadores tardan 6 meses en desarrollar una aplicación, ¿cuánto tiempo tardarían 8 programadores?" ¿En cuántos escenarios se reduce realmente el tiempo?

42

Ubicuo Automatización

Civilizatiow advawces by extewdiwg the wumber of importawt operatiows ue caw perform uithout thiwkiwg.

► Alfred North Whitehead

En los albores de la era de los automóviles, las instrucciones para arrancar un Ford Modelo T tenían más de dos páginas. Con los coches modernos, sólo tienes que girar la llave: el procedimiento de arranque es automático e infalible. Una persona que siga una lista de instrucciones puede inundar el motor, pero el arranque automático no lo hará.

Aunque la informática sigue siendo una industria en la etapa del Modelo T, no podemos permitirnos el lujo de pasar por dos páginas de instrucciones una y otra vez para alguna operación común. Ya sea el procedimiento de compilación y lanzamiento, el papeleo de revisión de código o cualquier otra tarea recurrente en el proyecto, debe ser automático. Es posible que tengamos que construir el motor de arranque y el inyector de combustible desde cero, pero una vez hecho esto, podemos girar la llave a partir de ese momento.

Además, queremos garantizar la coherencia y la repetibilidad del proyecto. Los procedimientos manuales dejan la consistencia al azar; La repetibilidad no está garantizada, especialmente si algunos aspectos

del procedimiento están abiertos a la interpretación de diferentes personas.

Todo en automático

Una vez estuvimos en el sitio de un cliente donde todos los desarrolladores estaban usando el mismo IDE. El administrador del sistema dio a cada desarrollador un conjunto de instrucciones sobre cómo instalar paquetes complementarios en el IDE. Estas instrucciones llenaron muchas páginas, páginas llenas de haga clic aquí, desplácese allí, arrastre esto, haga doble clic en aquello y vuelva a hacerlo.

No es de extrañar que la máquina de cada desarrollador se cargara de forma ligeramente diferente. Se produjeron diferencias sutiles en el comportamiento de la aplicación cuando diferentes desarrolladores ejecutaron el mismo código. Los errores aparecían en una máquina, pero no en otras. El rastreo de las diferencias de versión de cualquier componente generalmente revelaba una sorpresa.

CONSEJO

No utilices procedimientos manuales

Las personas simplemente no son tan repetibles como lo son las computadoras. Tampoco debemos esperar que lo sean. Un script de shell o un archivo por lotes ejecutará las mismas instrucciones, en el mismo orden, una y otra vez. Se puede poner bajo control de código fuente, por lo que también puede examinar los cambios en el procedimiento a lo largo del tiempo ("pero *solia* funcionar. . .").

Otra herramienta favorita de automatización es cron (o "arroba" en Windows NT). Nos permite programar tareas desatendidas para que se ejecuten periódicamente, generalmente en medio de la noche. Por ejemplo, el siguiente archivo crontab especifica que el comando nocturno de un proyecto se ejecute a cinco minutos después de la medianoche todos los días, que la copia de seguridad se ejecute a las 3:15 a.m. de lunes a viernes y que se expense_reports ejecutar a la medianoche del primer día del mes.

#	MIN	HORA	DÍA	MES	DÍA DE SEMANA	MANDAR
#	-	-	-	-	-	-
5	0	*	*	*	*	/proyectos/Manhattan/papelera/nocturno
15	3	*	*	1-5		/usr/local/bin/copia de seguridad
0	0	1	*	*	*	/inicio/contabilidad/expense_reports

Usando cron, podemos programar copias de seguridad, la construcción nocturna, el mantenimiento del sitio web y cualquier otra cosa que deba hacerse, sin supervisión, automáticamente .

Compilando el proyecto

La compilación del proyecto es una tarea que debe ser fiable y repetible. Por lo general, compilamos proyectos con archivos MAKE, incluso cuando usamos un entorno IDE. Hay varias ventajas en el uso de makefiles. Es un procedimiento automático y con guión. Podemos agregar ganchos para generar código por nosotros y ejecutar pruebas de regresión automáticamente. Los IDE tienen sus ventajas, pero solo con ellos puede ser difícil alcanzar el nivel de automatización que buscamos. Queremos revisar, compilar, probar y enviar con un solo comando.

Generación de código

En *The Evils of Duplication*, página 26, abogamos por generar código para derivar conocimiento de fuentes comunes. Podemos aprovechar el mecanismo de análisis de dependencia de make para facilitar este proceso. Es bastante sencillo añadir reglas a un makefile para generar automáticamente un fichero a partir de alguna otra fuente. Por ejemplo, supongamos que queremos tomar un archivo XML, generar un archivo Java a partir de él y compilar el resultado.

```
.SUFIJOS: .java .class .xml
.xml.java:
    Perl convert.pl $< > $@
.java.clase:
    $(JAVAC) $(JAVAC_FLAGS) $<
```

Escriba `make test.class` y make buscará automáticamente un archivo llamado `test.xml`, creará un archivo `.java` ejecutando un script Perl y, a continuación, compilará ese archivo para producir `test.class`.

También podemos usar el mismo tipo de reglas para generar código fuente, archivos de encabezado o documentación automáticamente desde alguna otra forma (consulte *Code Generators*, página 102).

Pruebas de regresión

También puede utilizar el archivo make para ejecutar pruebas de regresión, ya sea para un módulo individual o para un subsistema completo. Puede probar fácilmente el *proyecto ewtire* con un solo comando en la parte superior del árbol de fuentes, o puede probar un módulo individual utilizando el mismo comando en un solo directorio. Consulte *Ruthless Testing*, página 237, para obtener más información

sobre las pruebas de regresión.

Creación recursiva

Muchos proyectos configuran archivos MAKE recursivos y jerárquicos para las compilaciones y pruebas de proyectos. Pero tenga en cuenta algunos problemas potenciales.

`make` calcula las dependencias entre los distintos objetivos que tiene que construir. Pero solo puede analizar las dependencias que existen dentro de una sola invocación `make`. En particular, un `make` recursivo no tiene conocimiento de las dependencias que pueden tener otras invocaciones de `make`. Si es cuidadoso y preciso, puede obtener los resultados adecuados, pero es fácil causar trabajo adicional innecesariamente, o perder una dependencia y *no volver a compilar cuando sea necesario*.

Además, es posible que las dependencias de compilación no sean las mismas que las dependencias de prueba, y es posible que

Automatización de compilaciones

Una *compilación* es un procedimiento que toma un directorio vacío (y un entorno de compilación conocido) y construye el proyecto desde cero, produciendo lo que se espera producir como entregable final: una imagen de memoria de CD-ROM o un archivo autoextraíble, por ejemplo. Normalmente, la compilación de un proyecto abarcará los siguientes pasos.

1. Echa un vistazo al código fuente del repositorio.
2. Construya el proyecto desde cero, normalmente a partir de un archivo `make` de nivel superior. Cada compilación está marcada con algún tipo de número de lanzamiento o versión, o tal vez una marca de fecha.
3. Cree una imagen distribuible. Este procedimiento puede implicar la fijación de la propiedad y los permisos de los archivos, y la producción de todos los ejemplos, documentaciones, archivos `README` y cualquier otra cosa que se envíe con el producto, en el formato exacto que se requerirá cuando se envíe.³
4. Ejecute las pruebas especificadas (`make test`).

3. Si está produciendo un CD-ROM en formato ISO9660, por ejemplo, debe ejecutar el programa que produce una imagen bit por bit del sistema de archivos 9660. ¿Por qué esperar hasta la noche antes de realizar el envío para asegurarse de que funciona?

Para la mayoría de los proyectos, este nivel de compilación se ejecuta automáticamente todas las noches. En esta compilación nocturna, normalmente ejecutará más pruebas completas de las que una persona podría ejecutar al compilar una parte específica del proyecto. El punto importante es que la compilación completa ejecute *todas las* pruebas disponibles. Desea saber si se produjo un error en una prueba de regresión debido a uno de los cambios de código actuales. Al identificar el problema cerca de la fuente, tiene más posibilidades de encontrarlo y solucionarlo.

Si no realiza pruebas con regularidad, es posible que descubra que la aplicación se rompió debido a un cambio de código realizado hace tres meses. Buena suerte para encontrarlo.

Compilaciones finales

Las compilaciones de Fiwal, que tiene la intención de enviar como productos, pueden tener requisitos diferentes a los de la compilación nocturna normal. Una compilación final puede requerir que el repositorio esté bloqueado o etiquetado con el número de versión, que las marcas de optimización y depuración se establezcan de manera diferente, etc. Nos gusta usar un objetivo make independiente (como make final) que establezca todos estos parámetros a la vez.

Recuerde que si el producto se compila de manera diferente a las versiones anteriores, entonces debe probar esta versión de nuevo.

Administrivia Automática

¿No sería bueno si los programadores pudieran dedicar todo su tiempo a la programación? Desafortunadamente, este rara vez es el caso. Hay que responder a un correo electrónico, llenar papeleo, publicar documentos en la Web, etc. Puede decidir crear un script de shell para hacer parte del trabajo sucio, pero aún debe recordar ejecutar el script cuando sea necesario.

Debido a que la memoria es la segunda cosa que se pierde a medida que se envejece,⁴ no queremos depender demasiado de ella. Podemos ejecutar scripts para que nos procesen automáticamente, basados en la cantidad de código fuente y documentos. Nuestro objetivo es mantener un flujo de trabajo automático, desatendido y basado en el contenido.

4. ¿Cuál es la primera? Se me olvida.

Generación de sitios web

Muchos equipos de desarrollo utilizan un sitio Web interno para la comunicación de proyectos, y creemos que esta es una gran idea. Pero no queremos dedicar demasiado tiempo al mantenimiento del sitio web y no queremos dejar que se vuelva obsoleto o desactualizado. La información engañosa es peor que la falta de información.

La documentación que se extrae del código, los análisis de requisitos, los documentos de diseño y los dibujos, tablas o gráficos deben publicarse en la Web de forma regular. Nos gusta publicar estos documentos automáticamente como parte de la compilación nocturna o como un gancho en el procedimiento de registro del código fuente.

Independientemente de cómo se haga, el contenido web debe generarse automáticamente a partir de la información del repositorio y publicarse *sin* intervención humana. En realidad, se trata de otra aplicación del *principio DRY*: la información existe en una forma de código y documentos facturados. La vista del navegador web es simplemente eso: solo una vista. No debería tener que mantener esa vista a mano.

Cualquier información generada por la compilación nocturna debe ser accesible en el sitio Web de desarrollo: resultados de la compilación en sí (por ejemplo, los resultados de la compilación pueden presentarse como un resumen de una página que incluya advertencias del compilador, errores y estado actual), pruebas de regresión, estadísticas de rendimiento, métricas de codificación y cualquier otro análisis estático, etc.

Procedimientos de aprobación

Algunos proyectos tienen varios flujos de trabajo administrativos que deben seguirse. Por ejemplo, es necesario programar y seguir las revisiones de código o diseño, es posible que sea necesario conceder aprobaciones, etc. Podemos utilizar la automatización, y especialmente el sitio web, para ayudar a aliviar la carga del papeleo.

Supongamos que desea automatizar la programación y aprobación de la revisión de código. Puede poner un marcador especial en cada archivo de código fuente:

```
/* Estado: needs_review */
```

Un simple script podría revisar todo el código fuente y buscar todos los archivos que tuvieran un estado de `needs_review`, lo que indicaría que estaban listos para ser revisados. A continuación, puede publicar una

lista de esos archivos como un archivo

página web, envíe automáticamente un correo electrónico a las personas adecuadas, o incluso programe una reunión automáticamente utilizando algún software de calendario.

Puede configurar un formulario en una página web para que los revisores registren su aprobación o rechazo. Después de la revisión, el estado se puede cambiar automáticamente a revisado. Si tienes un recorrido de código con todos los participantes depende de ti; aún puedes hacer el papeleo automáticamente. (En un artículo publicado en la edición de abril de 1999 del CACM, Robert Glass resume las investigaciones que parecen indicar que, si bien la inspección de códigos es efectiva, la realización de revisiones en las reuniones no lo es [Gla99a]).

Los hijos del zapatero

Los hijos del zapatero no tienen zapatos. A menudo, las personas que desarrollan software utilizan las herramientas más pobres para hacer el trabajo.

Pero tenemos todas las materias primas que necesitamos para fabricar mejores herramientas. Tenemos cron. Tenemos make, Ant y CruiseControl para la automatización (ver [Clao4]). Y tenemos Ruby, Perl y otros lenguajes de scripting de alto nivel para desarrollar rápidamente herramientas personalizadas, generadores de páginas web, generadores de código, arneses de prueba, etc.

Dejemos que la computadora haga lo repetitivo, lo mundano, lo hará mejor que nosotros. Tenemos cosas más importantes y más difíciles que hacer.

Las secciones relacionadas incluyen:

- *El gato se comió mi código fuente*, página 2
- *Los males de la duplicación*, página 26
- *El poder del texto plano*, página 73
- *Juegos de conchas*, página 77
- *Depuración*, página 90
- *Generadores de código*, página 102
- *Equipos pragmáticos*, página 224
- *Pruebas despiadadas*, página 237
- *Todo es escritura*,

Desafíos

- Observa tus hábitos a lo largo de la jornada laboral. ¿Ves alguna tarea repetitiva? ¿Escribe la misma secuencia de comandos una y otra vez?

Intente escribir algunos scripts de shell para automatizar el proceso.
 ¿Siempre haces clic en la misma secuencia de iconos repetidamente?
 ¿Puedes crear una macro para que haga todo eso por ti?

- ¿Cuánto del papeleo de tu proyecto se puede automatizar? Dado el alto costo del personal de programación,⁵ determine cuánto de los ingresos del proyecto se está desperdiciando en procedimientos administrativos. ¿Puede justificar la cantidad de tiempo que llevaría elaborar una solución automatizada en función del ahorro de costes general que conseguiría?

43

Pruebas despiadadas

La mayoría de los desarrolladores odian las pruebas. Tienden a probar suavemente, sabiendo inconscientemente dónde se romperá el código y evitando los puntos débiles. Los programadores pragmáticos son diferentes. Estamos *motivados* para encontrar nuestros insectos, por lo que no tenemos que soportar la vergüenza de que otros encuentren nuestros errores más tarde.

Encontrar insectos es algo así como pescar con una red. Utilizamos redes finas y pequeñas (pruebas unitarias) para atrapar a los pececillos, y redes grandes y gruesas (pruebas de integración) para atrapar a los tiburones asesinos. A veces los peces logran escapar, por lo que parcheamos los agujeros que encontramos, con la esperanza de atrapar más y más defectos resbaladizos que nadan en nuestra piscina de proyecto.

CONSEJO

Prueba temprano. Pruebe con frecuencia. Pruebe

Queremos empezar a probar tan pronto como tengamos el código. Esos pequeños pececillos tienen la desagradable costumbre de convertirse en tiburones gigantes y devoradores de hombres bastante rápido, y atrapar un tiburón es un poco más difícil. Pero no queremos tener que hacer todas esas pruebas a mano.

5. A efectos de estimación, se puede calcular un promedio de la industria de alrededor de US\$100,000 por cabeza, es decir, el salario más los beneficios, la capacitación, el espacio de oficina y los gastos generales, etc.

Muchos equipos desarrollan planes de prueba elaborados para sus proyectos. A veces incluso los usarán. Pero hemos descubierto que los equipos que utilizan pruebas automáticas tienen muchas más posibilidades de éxito. Las pruebas que se ejecutan con cada compilación son mucho más eficaces que los planes de prueba que se quedan en una estantería.

Cuanto antes se encuentre un error, más barato será remediarlo. "Codifica un poco, prueba un poco" es un dicho popular en el mundo de Smalltalk,⁶ y podemos adoptar ese mantra como propio escribiendo el código de prueba al mismo tiempo (o incluso antes) de escribir el código de producción.

De hecho, un buen proyecto puede tener *más* código de prueba que código de producción. El tiempo que se tarda en producir este código de prueba vale la pena. Termina siendo mucho más barato a largo plazo, y realmente tiene la oportunidad de producir un producto con casi cero defectos.

Además, saber que ha aprobado la prueba le da un alto grado de confianza de que un fragmento de código está "hecho".

CONSEJO

La codificación no está terminada hasta que se ejecuten

El hecho de que hayas terminado de hackear un fragmento de código no significa que puedas decirle a tu jefe o a tu cliente que es perfecto. No lo es. En primer lugar, el código nunca está realmente terminado. Y lo que es más importante, no se puede afirmar que nadie pueda utilizarlo hasta que pase todas las pruebas disponibles.

Tenemos que fijarnos en tres aspectos principales de las pruebas de todo el proyecto: qué probar, cómo probar y cuándo probar.

Qué probar

Hay varios tipos principales de pruebas de software que debe realizar:

- Pruebas unitarias
- Pruebas de integración
- Validación y verificación

6. eXtreme Programming [URL 45] llama a este concepto "integración continua, pruebas implacables".

- Agotamiento de recursos, errores y recuperación
- Pruebas de rendimiento
- Pruebas de usabilidad

Esta lista no está completa de ninguna manera, y algunos proyectos especializados también requerirán otros tipos de pruebas. Pero nos da un buen punto de partida.

Pruebas unitarias

Una *prueba uit* es un código que ejercita un módulo. Cubrimos este tema por sí solo en *Easy to Test de Code That*, página 189. Las pruebas unitarias son la base de todas las demás formas de pruebas que discutiremos en esta sección. Si las piezas no funcionan por sí solas, probablemente no funcionarán bien juntas. Todos los módulos que está utilizando deben pasar sus propias pruebas unitarias antes de poder continuar.

Una vez que todos los módulos pertinentes hayan pasado sus pruebas individuales, estará listo para la siguiente etapa. Debe probar cómo todos los módulos se usan e interactúan entre sí en todo el sistema.

Pruebas de integración

El testimonio de Iwtegratiow muestra que los principales subsistemas que componen el proyecto funcionan y funcionan bien entre sí. Con buenos contratos establecidos y bien probados, cualquier problema de integración se puede detectar fácilmente. De otro modo, la integración se convierte en un caldo de cultivo fértil para los insectos. De hecho, a menudo es la mayor fuente de errores en el sistema.

Las pruebas de integración son en realidad solo una extensión de las pruebas unitarias que hemos descrito, solo que ahora está probando cómo subsistemas completos cumplen con sus contratos.

Validación y verificación

Tan pronto como tenga una interfaz de usuario ejecutable o un prototipo, debe responder a una pregunta muy importante: los usuarios le dijeron lo que querían, pero ¿es lo que necesitan?

¿Cumple con los requisitos funcionales del sistema? Esto también hay que probarlo. Un sistema libre de errores que responde a la pregunta equivocada no es muy útil. Tenga en cuenta los patrones de acceso de los usuarios finales y

En qué se diferencian de los datos de las pruebas de los reveladores (por ejemplo, consulte el artículo sobre las pinceladas en la página 92).

Agotamiento de recursos, errores y recuperación

Ahora que se tiene una idea bastante clara de que el sistema se comportará correctamente en condiciones ideales, es necesario descubrir cómo se comportará en *condiciones reales*. En el mundo real, sus programas no tienen recursos ilimitados; Se quedan sin cosas. Estos son algunos de los límites que puede encontrar el código:

- Memoria
- Espacio en disco
- Ancho de banda
- de la CPU
- Tiempo de reloj
- de pared • Ancho
- de banda del
- disco
- Ancho de banda de
- red • Paleta de
- colores
- Resolución de vídeo

De hecho, es posible que compruebe si hay errores de espacio en disco o de asignación de memoria, pero ¿con qué frecuencia realiza pruebas para los demás? ¿Su aplicación cabrá en una 640×480 pantalla con 256 colores? ¿Se ejecutará en una 1600×1280 pantalla con 24color -bit sin parecer un sello postal? ¿Finalizará el trabajo por lotes antes de que se inicie el archivo?

Puede detectar las limitaciones del entorno, como las especificaciones de vídeo, y adaptarlas según corresponda. Sin embargo, no todos los errores son recuperables. Si su código detecta que la memoria se ha agotado, sus opciones son limitadas: es posible que no le queden suficientes recursos para hacer cualquier cosa excepto fallar.

Cuando el sistema falla,⁷ ¿fallará con gracia? ¿Intentará, lo mejor que pueda, salvar su estado y evitar la pérdida de trabajo? ¿O será "GPF" o "core-dump" en la cara del usuario?

7. Nuestro editor quería que cambiáramos esta frase por "Si el sistema falla . . . ". Resistimos.

Pruebas de rendimiento

Las pruebas de rendimiento, las pruebas de estrés o las pruebas bajo carga también pueden ser un aspecto importante del proyecto.

Pregúntese si el software cumple con los requisitos de rendimiento en condiciones del mundo real, con el número esperado de usuarios, conexiones o transacciones por segundo. ¿Es escalable?

Para algunas aplicaciones, es posible que necesite hardware o software de prueba especializado para simular la carga de manera realista.

Pruebas de usabilidad

Las pruebas de usabilidad son diferentes de los tipos de pruebas discutidos hasta ahora. Se realiza con usuarios reales, en condiciones ambientales reales.

Mira la usabilidad en términos de factores humanos. ¿Hubo algún malentendido durante el análisis de los requisitos que deba abordarse? ¿El software se adapta al usuario como una extensión de la mano? (No solo queremos que nuestras propias herramientas se ajusten a nuestras manos, sino que también queremos que las herramientas que creamos para los usuarios se ajusten a sus manos).

Al igual que con la validación y la verificación, es necesario realizar pruebas de usabilidad lo antes posible, mientras todavía hay tiempo para hacer correcciones. Para proyectos más grandes, es posible que desee contratar especialistas en factores humanos. (Por lo menos, es divertido jugar con los espejos unidireccionales).

No cumplir con los criterios de usabilidad es un error tan grande como dividir por cero.

Cómo hacer la prueba

Hemos mirado *what* para probar. Ahora centraremos nuestra atención en *how* para probar, incluyendo:

- Pruebas de regresión
- Datos de prueba
- Ejercicio de los sistemas

- GUI • Prueba de las pruebas
- Pruebas exhaustivas

Pruebas de Diseño/Metodología

¿Puedes probar el diseño del código en sí y la metodología que utilizaste para construir el software? Después de un tiempo, sí se puede. Para ello, analice *las métricas*, es decir, las mediciones de varios aspectos del código. La métrica más simple (y a menudo la menos interesante) son *las líneas de código*: ¿qué tan grande es el código en sí?

Hay una amplia variedad de otras métricas que puede usar para examinar el código, entre ellas:

- Métrica de Complejidad Ciclomática de McCabe (mide la complejidad de las estructuras de decisión)
- Distribución ramificada de herencia (número de clases base) y distribución ramificada (número de módulos derivados que utilizan este como parente)
- Conjunto de respuestas (ver *Desacoplamiento y la Ley de Deméter*, página 138)
- Relaciones de acoplamiento de clases (véase [URL 48])

Algunas métricas están diseñadas para dar una "calificación aprobatoria", mientras que otras son útiles solo en comparación. Es decir, se calculan estos metros para cada módulo del sistema y se observa cómo un módulo en particular se relaciona con sus hermanos. En este caso se suelen utilizar técnicas estadísticas estándar (como la media y la desviación estándar).

Si encuentra un módulo cuyas métricas son marcadamente diferentes de todos los demás, debe preguntarse si eso es apropiado. Para algunos módulos, puede estar bien "soplar la curva". Pero para aquellos que *no* tienen una buena excusa, puede indicar problemas potenciales.

Pruebas de regresión

Una prueba de regresión compara el resultado de la prueba actual con los valores anteriores (o conocidos). Podemos asegurarnos de que los errores que corregimos hoy no rompieron las cosas que funcionaban ayer. Esta es una red de seguridad importante y reduce las sorpresas desagradables.

Todas las pruebas que hemos mencionado hasta ahora se pueden ejecutar como pruebas de regresión, lo que garantiza que no hemos perdido terreno a medida que desarrollamos nuevo código. Podemos ejecutar regresiones para verificar el rendimiento, los contratos, la

validez, etc.

Datos de prueba

¿De dónde obtenemos los datos para ejecutar todas estas pruebas? Solo hay dos tipos de datos: datos del mundo real y datos sintéticos. De hecho, necesitamos usar ambos, porque las diferentes naturalezas de estos tipos de datos expondrán diferentes errores en nuestro software.

Los datos del mundo real provienen de alguna fuente real. Posiblemente se haya recopilado de un sistema existente, del sistema de un competidor o de un prototipo de algún tipo. Representa los datos típicos del usuario. Las grandes sorpresas llegan a medida que descubres lo que *significa típico*. Lo más probable es que esto revele defectos y malentendidos en el análisis de requisitos.

Los datos sintéticos se generan artificialmente, tal vez bajo ciertas restricciones estadísticas. Es posible que necesite utilizar datos sintéticos por cualquiera de los siguientes motivos.

- Necesita una gran cantidad de datos, posiblemente más de los que podría proporcionar cualquier muestra del mundo real. Es posible que pueda usar los datos del mundo real como semilla para generar un conjunto de muestras más grande y ajustar ciertos campos que deben ser únicos.
- Necesita datos para enfatizar las condiciones de contorno. Estos datos pueden ser completamente sintéticos: campos de fecha que contienen el 29 de febrero de 1999, tamaños de registro grandes o direcciones con códigos postales extranjeros.
- Necesita datos que presenten ciertas propiedades estadísticas. ¿Quieres ver qué sucede si una de cada tres transacciones falla? ¿Recuerdas el algoritmo de ordenación que se ralentiza cuando se entregan datos preclasificados? Puede presentar los datos de forma aleatoria o ordenada para exponer este tipo de debilidad.

Ejercicio de los sistemas GUI

Las pruebas de sistemas intensivos en GUI a menudo requieren herramientas de prueba especializadas. Estas herramientas pueden basarse en un modelo simple de captura/reproducción de eventos, o pueden requerir scripts especialmente escritos para controlar la GUI. Algunos sistemas combinan elementos de ambos.

Las herramientas menos sofisticadas imponen un alto grado de acoplamiento entre la versión del software que se está probando y el propio script de prueba: si mueve un cuadro de diálogo o hace un

botón más pequeño, es posible que la prueba no pueda

encontrarlo, y puede fallar. La mayoría de las herramientas modernas de pruebas de GUI utilizan una serie de técnicas diferentes para solucionar este problema e intentan ajustarse a pequeñas diferencias de diseño.

Sin embargo, no se puede automatizar todo. Andy trabajó en un sistema de gráficos que permitía al usuario crear y mostrar efectos visuales no deterministas que simulaban varios fenómenos naturales. Desgraciadamente, durante las pruebas no se podía simplemente coger un mapa de bits y comparar la salida con una ejecución anterior, porque estaba diseñado para ser diferente cada vez. Para situaciones como esta, es posible que no tenga más remedio que confiar en la interpretación manual de los resultados de las pruebas.

Una de las muchas ventajas de escribir código desacoplado (véase *Decoupling awd the Law of Demeter*, página 138) es que las pruebas son más modulares. Por ejemplo, para las aplicaciones de procesamiento de datos que tienen una interfaz gráfica de usuario, su diseño debe estar lo suficientemente desacoplado para que pueda probar la lógica de aplicación *sin* tener una interfaz gráfica de usuario presente. Esta idea es similar a probar primero los subcomponentes. Una vez que se ha validado la lógica de la aplicación, es más fácil localizar los errores que aparecen con la interfaz de usuario en su lugar (es probable que los errores hayan sido creados por el código de la interfaz de usuario).

Prueba de las pruebas

Debido a que no podemos escribir software perfecto, se deduce que tampoco podemos escribir software de prueba perfecto. Necesitamos probar las pruebas.

Piense en nuestro conjunto de conjuntos de pruebas como un elaborado sistema de seguridad, diseñado para hacer sonar la alarma cuando aparece un error. ¿Qué mejor manera de probar un sistema de seguridad que intentar entrar?

Después de haber escrito una prueba para detectar un error en particular, *use* el error deliberadamente y asegúrese de que la prueba se queje. Esto asegura que la prueba detectará el error si ocurre de verdad.

CONSEJO

Utilice saboteadores para probar sus pruebas

Si realmente se *toma en serio* las pruebas, es posible que desee designar a un *Proyecto saboteur*. El papel del saboteador es tomar una copia separada de la

árbol de fuentes, introduzca errores a propósito y verifique que las pruebas los detecten.

Al escribir pruebas, asegúrese de que las alarmas suenen cuando deberían.

Pruebas exhaustivas

Una vez que esté seguro de que sus pruebas son correctas y encuentre los errores que crea, ¿cómo sabe si ha probado la base de código lo suficientemente a fondo?

La respuesta corta es "no lo haces", y nunca lo harás. Pero hay productos en el mercado que pueden ayudar. Estas herramientas de *cobertura de lisis* observan su código durante las pruebas y realizan un seguimiento de qué líneas de código se han ejecutado y cuáles no. Estas herramientas te ayudan a tener una idea general de lo completas que son tus pruebas, pero no esperes ver una cobertura del 100%.

Incluso si llegas a cada línea de código, esa no es la imagen completa. Lo importante es el número de estados que puede tener su programa. Los estados no son equivalentes a las líneas de código. Por ejemplo, supongamos que tiene una función que toma dos números enteros, cada uno de los cuales puede ser un número de 0 a 999.

```
int test(int a, int b) {
    devolver a / (a + b);
}
```

En teoría, esta función de tres líneas tiene 1.000.000 de estados lógicos, 999.999 de los cuales funcionarán correctamente y uno que no (cuando $a + b$ es igual a cero). El simple hecho de saber que ejecutó esta línea de código no le dice eso, sino que tendría que identificar todos los estados posibles del programa. Desafortunadamente, en general este es un *problema realmente difícil*. Duro como en: "El sol será un bullo frío y duro antes de que puedas resolverlo".

CONSE

Cobertura de estado de prueba, no cobertura de código

Incluso con una buena cobertura de código, los datos que se usan para las pruebas siguen teniendo un gran impacto y, lo que es más importante, el *orden* en el que se recorre el código puede tener el mayor impacto de todos.

Cuándo hacer la prueba

Muchos proyectos tienden a dejar las pruebas para el último minuto, justo donde se cortarán contra el borde afilado de una fecha límite.⁸ Tenemos que empezar mucho antes que eso. Tan pronto como existe un código de producción, debe probarse.

La mayoría de las pruebas deben realizarse automáticamente. Es importante tener en cuenta que por "automáticamente" nos referimos a que los resultados de la prueba también se interpretan automáticamente. Véase *Ubiquitous Automation*, página 230, para más información sobre este tema.

Nos gusta probar con la mayor frecuencia posible, y siempre antes de registrar el código en el repositorio fuente. Algunos sistemas de control de código fuente, como Aegis, pueden hacer esto automáticamente. De lo contrario, simplemente escribimos

% de hacer la prueba

Por lo general, no es un problema ejecutar regresiones en todas las pruebas unitarias individuales y pruebas de integración con la frecuencia necesaria.

Sin embargo, es posible que algunas pruebas no se ejecuten fácilmente con tanta frecuencia. Las pruebas de estrés, por ejemplo, pueden requerir una configuración o equipo especial, y algo de agarre. Es posible que estas pruebas se realicen con menos frecuencia, tal vez semanales o mensuales. Pero es importante que se ejecuten de forma regular y programada. Si no se puede hacer automáticamente, asegúrese de que aparezca en la programación, con todos los recursos necesarios asignados a la tarea.

Apretando la red

Por último, nos gustaría revelar el concepto más importante de las pruebas. Es obvio, y prácticamente todos los libros de texto dicen que hay que hacerlo de esta manera. Pero por alguna razón, la mayoría de los proyectos todavía no lo hacen.

Si un error se desliza a través de la red de pruebas existentes, debe agregar una nueva prueba para atraparlo la próxima vez.

8. **Línea muerta** \déd-līn\ *w* (1864) una línea trazada dentro o alrededor de una prisión por la que pasa un prisionero a riesgo de ser fusilado (*Webster's Collegiate Dictionary*).

CONSE**Encuentra insectos una vez**

Una vez que un probador humano encuentra un error, debería ser la *primera vez* que un probador humano encuentra ese error. Las pruebas automatizadas deben modificarse para buscar ese error en particular a partir de ese momento, cada vez, sin excepciones, sin importar cuán triviales sean, y sin importar cuánto se queje el desarrollador y diga: "Oh, eso nunca volverá a suceder".

Porque volverá a suceder. Y simplemente no tenemos tiempo para ir a perseguir errores que las pruebas automatizadas podrían haber encontrado por nosotros. Tenemos que dedicar nuestro tiempo a escribir nuevo código y nuevos errores.

Las secciones relacionadas incluyen:

- *El Cat se comió mi código fuente*, página 2
- *Debuggiwg*, página 90
- *Decoupling awd el Lau de Deméter*, página 138
- *Refactoriwg*, página 184
- *Codificar el Easy de That para probar*, página 189
- *Automático ubicuo*, página 230

Desafíos

- ¿Puedes probar tu proyecto automáticamente? Muchos equipos se ven obligados a responder "no". ¿Por qué? ¿Es demasiado difícil definir los resultados aceptables? ¿No será esto difícil demostrar a los patrocinadores que el proyecto está "terminado"?
- ¿Es demasiado difícil probar la lógica de la aplicación independientemente de la GUI? ¿Qué dice esto sobre la GUI? ¿Sobre el acoplamiento?

Todo es escritura

El mejor momento es mejor que la mejor memoria.

► **Proverbio chino**

Normalmente, los desarrolladores no piensan mucho en la documentación. En el mejor de los casos es una necesidad desafortunada; En el peor de los casos, se trata como una tarea de baja prioridad con la esperanza de que la gerencia se olvide de ella al final del proyecto.

Los programadores pragmáticos adoptan la documentación como una parte integral del proceso general de desarrollo. La redacción de la documentación puede facilitarse si no se duplican esfuerzos ni se pierde tiempo, y si se tiene la documentación a mano, si es posible en el propio código.

Estos no son exactamente pensamientos originales o novedosos; la idea del código de boda y la documentación aparece en el trabajo de Donald Knuth sobre programación alfabetizada y en la utilidad JavaDoc de Sun, entre otros. Queremos restar importancia a la dicotomía entre código y documentación y, en su lugar, tratarlos como dos vistas del mismo modelo (consulte *It's Just a View*, página 157). De hecho, queremos ir un poco más allá y aplicar *todos* nuestros principios pragmáticos tanto a la documentación como al código.

CONSE

Trata el inglés como un lenguaje de programación más

Básicamente, hay dos tipos de documentación producida para un proyecto: interna y externa. La documentación interna incluye comentarios sobre el código fuente, documentos de diseño y pruebas, etc. La documentación externa es cualquier cosa enviada o publicada al mundo exterior, como los manuales de usuario. Pero independientemente del público al que se dirija, o del papel del escritor (desarrollador o redactor técnico), toda la documentación es un espejismo del código. Si hay una discrepancia, el código es lo que importa, para bien o para mal.

CONSE

Incorpore la documentación, no la atornille

Empezaremos por la documentación interna.

Comentarios en el código

Producir documentos formateados a partir de los comentarios y declaraciones en el código fuente es bastante sencillo, pero primero tenemos que asegurarnos de que realmente *tenemos* comentarios en el código. El código debe tener comentarios, pero demasiados comentarios pueden ser tan malos como muy pocos.

En general, los comentarios deben discutir *cuánto* se hace algo, su propósito y su objetivo. El código ya muestra *cómo* se ha hecho, por lo que comentar sobre esto es redundante y es una violación del principio *DRY*.

Comentar el código fuente le brinda la oportunidad perfecta para documentar esas partes elusivas de un proyecto que no se pueden documentar en ningún otro lugar: compensaciones de ingeniería, por qué se tomaron las decisiones, qué otras alternativas se descartaron, etc.

Nos gusta ver un *simple* comentario de encabezado a nivel de módulo, comentarios para datos significativos y declaraciones de tipo, y un breve encabezado por clase y por método, que describa cómo se usa la función y cualquier cosa que haga que no sea obvia.

Los nombres de las variables, por supuesto, deben estar bien elegidos y ser significativos. Foo, por ejemplo, no tiene sentido, al igual que doit o manager o cosas así. La notación hun-gariana (en la que se codifica la información de tipo de la variable en el propio nombre) es totalmente inapropiada en los sistemas orientados a objetos. Recuerde que usted (y otros después de usted) volverá a leer el código muchos cientos de veces, pero solo unas pocas veces. Tómese el tiempo para deletrear connectionPool en lugar de cp.

Incluso peor que los nombres sin sentido son los nombres *misleading*. ¿Alguna vez ha tenido a alguien que explique las inconsistencias en el código heredado, como "La rutina llamada getData realmente escribe datos en el disco"? El cerebro humano estropeará esto repetidamente, lo que se llama el *Efecto Stroop* [Str35]. Usted mismo puede intentar el siguiente experimento para ver los efectos de tal interferencia. Consigue algunos bolígrafos de colores y úsalos para escribir los nombres de los colores. Sin embargo, nunca escriba el nombre de un color con ese rotulador de color. Puedes escribir la palabra "azul" en verde, la palabra "marrón" en rojo, y así sucesivamente. (Alternativamente, tenemos un conjunto de muestra de colores ya dibujados en nuestro sitio web en

[www.pragmaticprogrammer.com.\)](http://www.pragmaticprogrammer.com.) Una vez que hayas dibujado los nombres de los colores, trata de decir en voz alta el color con el que se dibuja cada palabra, lo más rápido que puedas. En algún momento te tropezarás y empezarás a leer los nombres de los colores, y no los colores en sí. Los nombres son:

profundamente significativo para su cerebro, y los nombres engañosos agregan caos a su código.

Puede documentar los parámetros, pero pregúntese si es realmente necesario en todos los casos. El nivel de comentario sugerido por la herramienta JavaDoc parece apropiado:

```
/*
 * Encuentre el valor máximo (más alto) dentro de una fecha especificada
 * gama de muestras.
 *
 * @param aRange Rango de fechas para buscar datos.
 * @param aThreshold Valor mínimo que se debe tener en cuenta.
 * @return el valor, o <code>null</code> si no se encontró ningún valor
 *         mayor o igual que el umbral.
 */
public Sample findPeak(DateRange aRange, double aThreshold);
```

Aquí hay una lista de cosas que deberían aparecer en los comentarios de la fuente.

- Una lista de las funciones exportadas por código en el archivo. Hay programas que analizan la fuente por usted. Utilícelos y se garantizará que la lista esté actualizada.
- Historial de revisiones. Para esto están los sistemas de control de código fuente (ver *Código Fuente Cowtrol*, página 86). Sin embargo, puede ser útil incluir información sobre la fecha del último cambio y la persona que lo realizó.⁹
- Una lista de otros archivos que utiliza este archivo. Esto se puede determinar con mayor precisión utilizando herramientas automáticas.
- El nombre del archivo. Si debe aparecer en el archivo, no lo mantenga a mano. RCS y sistemas similares pueden mantener esta información actualizada automáticamente. Si mueves o cambias el nombre del archivo, no querrás tener que acordarte de editar el encabezado.

Una de las piezas de información más importantes que *deben* aparecer en el archivo de origen es el nombre del autor, no necesariamente quién editó el archivo en último lugar, sino el propietario. Atribuir responsabilidad y rendición de cuentas al código fuente hace maravillas para mantener a las personas honestas (véase *Pride and Prejudice*, página 258).

^{9.} Este tipo de información, así como el nombre del archivo, es proporcionada por la etiqueta RCS

\$Id\$.

El proyecto también puede requerir que aparezcan ciertos avisos de derechos de autor u otra plantilla legal en cada archivo fuente. Haz que tu editor los inserte automáticamente.

Con comentarios significativos, herramientas como JavaDoc [URL 7] y DOC++ [URL 21] pueden extraerlos y formatearlos para producir automáticamente documentación a nivel de API. Este es un ejemplo específico de una técnica más general que utilizamos: *los documentos ejecutables*.

Documentos ejecutables

Supongamos que tenemos una especificación que enumera las columnas de una tabla de base de datos. Luego tendremos un conjunto separado de comandos SQL para crear la tabla real en la base de datos, y probablemente algún tipo de estructura de registro de lenguaje de programación para contener el contenido de una fila en la tabla. La misma información se repite tres veces. Cambie cualquiera de estas tres fuentes, y las otras dos quedarán inmediatamente desactualizadas. Esto es una clara violación del *principio DRY*.

Para corregir este problema, debemos elegir la fuente de información autorizada. Esta puede ser la especificación, puede ser una herramienta de esquema de base de datos o puede ser una tercera fuente. Elijamos el documento de especificaciones como fuente. Ahora es nuestro *modelo* para este proceso. A continuación, tenemos que encontrar una manera de exportar la información que contiene como diferentes *vi-us*: un esquema de base de datos y un registro de lenguaje de alto nivel, por ejemplo.¹⁰

Si el documento se almacena como texto sin formato con comandos de marcado (mediante HTML, LaTex o troff, por ejemplo), puede utilizar herramientas como Perl para extraer el esquema y volver a formatearlo automáticamente. Si su documento está en el formato binario de un procesador de textos, luego vea el cuadro en la página siguiente para ver algunas opciones.

Su documento es ahora una parte integral del desarrollo del proyecto. La única forma de cambiar el esquema es cambiar el documento. Está garantizando que la especificación, el esquema y el código coincidan. Minimiza la cantidad de trabajo que tiene que realizar para cada cambio y puede actualizar las vistas del cambio automáticamente.

10. Consulte *It's Just a View*, página 157, para obtener más información sobre modelos y vistas.

¿Qué pasa si mi documento no es texto sin formato?

Desgraciadamente, cada vez son más los documentos de los proyectos que se escriben utilizando procesadores de texto que almacenan el archivo en un disco en algún formato propietario. Decimos "desafortunadamente" porque esto restringe severamente sus opciones para procesar el documento automáticamente. Sin embargo, todavía tienes un par de opciones:

- Escribir macros. La mayoría de los procesadores de texto sofisticados ahora tienen un lenguaje de macros. Con un poco de esfuerzo, puede programarlos para exportar secciones etiquetadas de sus documentos a los formularios alternativos que necesita. Si la programación a este nivel es demasiado laboriosa, siempre se puede exportar la sección apropiada a un archivo de texto plano estándar y luego usar
- una herramienta como Perl para convertirlo en las formas finales.

Subordinar el documento. En lugar de tener el documento como fuente definitiva, utilice otra representación. (En el ejemplo de la base de datos, es posible que desee utilizar el esquema como información acreditativa). A continuación, escriba una herramienta que exporte esta información a un formulario que el documento pueda importar. Sin embargo, ten cuidado. Debe asegurarse de que esta información se importe cada vez que se imprime el documento, en lugar de solo una vez cuando se crea el documento.

Podemos generar documentación a nivel de API a partir del código fuente utilizando herramientas como JavaDoc y DOC++ de manera similar. El modelo es el código fuente: se puede compilar una vista del modelo; otras vistas están destinadas a ser impresas o vistas en la Web. Nuestro objetivo es siempre trabajar en el modelo, ya sea el propio código o algún otro documento, y hacer que todas las vistas se actualicen automáticamente (consulte *Ubiquitous Automation*, página 230, para obtener más información sobre los procesos automáticos).

De repente, la documentación no es tan mala.

Escritores Técnicos

Hasta ahora, solo hemos hablado de la documentación interna, escrita por los propios programadores. Pero, ¿qué ocurre cuando hay redactores técnicos profesionales implicados en el proyecto? Con

demasiada frecuencia, los programadores se limitan a lanzar material "por encima de la pared" a los escritores técnicos y

Deje que se valgan por sí mismos para producir manuales de usuario, piezas promocionales, etc.

Esto es un error. El hecho de que los programadores no escriban estos documentos no significa que podamos renunciar a los principios pragmáticos. Queremos que los escritores adopten los mismos principios básicos que un programador pragmático, especialmente honrando el *principio DRY*, la ortogonalidad, el concepto de vista de modelo y el uso de la automatización y la secuenciación de comandos.

Imprimirlo o tejerlo

Un problema inherente a la documentación publicada en papel es que puede quedar obsoleta tan pronto como se imprime. La documentación de cualquier formulario es solo una instantánea.

Por lo tanto, tratamos de producir toda la documentación en una forma que pueda publicarse en línea, en la Web, con hipervínculos. Es más fácil mantener esta vista de la documentación actualizada que rastrear cada copia en papel existente, quemarla y reimprimir y redistribuir nuevas copias. También es una mejor manera de abordar las necesidades de una audiencia amplia. Sin embargo, recuerde poner un sello de fecha o un número de versión en cada página web. De esta manera, el lector puede tener una buena idea de lo que está actualizado, lo que ha cambiado recientemente y lo que no.

Muchas veces es necesario presentar la misma documentación en diferentes formatos: un documento impreso, páginas web, ayuda en línea o tal vez una presentación de diapositivas. La solución típica se basa en gran medida en cortar y pegar, creando una serie de nuevos documentos independientes a partir del original. Esta es una mala idea: la presentación de un documento debe ser independiente de su contenido.

Si está utilizando un sistema de marcado, tiene la flexibilidad de implementar tantos formatos de salida diferentes como necesite. Puedes elegir tener

`<H1>Título del Capítulo </H1>`

Genere un nuevo capítulo en la versión de informe del documento y asigne un título a una nueva diapositiva en la presentación de diapositivas. Tecnologías como XSL y CSS11 se pueden utilizar para generar múltiples formatos de salida a partir de este marcado.

11. Lenguaje de estilo extensible y hojas de estilo en cascada, dos tecnologías diseñadas para ayudar a separar la presentación del contenido.

Si está utilizando un procesador de textos, probablemente tendrá capacidades similares. Si recordó usar estilos para identificar diferentes elementos del documento, al aplicar diferentes hojas de estilo, puede alterar drásticamente el aspecto del resultado final. La mayoría de los procesadores de texto ahora le permiten convertir su documento a formatos como HTML para la publicación en la Web.

Lenguajes de marcado

Por último, en el caso de proyectos de documentación a gran escala, se recomienda consultar algunos de los esquemas más modernos para marcar la documentación.

En la actualidad, muchos autores técnicos utilizan DocBook para definir sus documentos. DocBook es un estándar de marcado basado en SGML que identifica cuidadosamente cada componente de un documento. El documento se puede pasar a través de un procesador DSSSL para renderizarlo en cualquier número de formatos diferentes. El proyecto de documentación de Linux utiliza DocBook para publicar información en formatos RTF, TEX, info, PostScript y HTML.

Siempre y cuando su marcado original sea lo suficientemente rico como para expresar todos los conceptos que necesita (incluidos los hipervínculos), la traducción a cualquier otra forma publicable puede ser fácil y automática. Puede producir ayuda en línea, manuales publicados, productos destacados para el sitio web e incluso un calendario de consejos al día, todo de la misma fuente, que por supuesto está bajo control de código fuente y se construye junto con la compilación nocturna (consulte *Ubiq- uitous Automatiow*, página 230).

La documentación y el código son vistas diferentes del mismo modelo subyacente, pero la vista es *única* que debería ser diferente. No permita que la documentación se convierta en un ciudadano de segunda clase, desterrado del flujo de trabajo principal del proyecto. Trate la documentación con el mismo cuidado con el que trata el código, y los usuarios (y los mantenedores que le siguen) cantarán sus alabanzas.

Las secciones relacionadas incluyen:

- *Los males de Duplicatiow*, página 26
- *Ortodomía*, página 34
- *El Pouer de Plaiw Texto*, página 73
- *Código fuente Cowtrol*, página 86
- *Es solo a Vieu*, página 157

- *Programmiwg por Coiwcidewce*, página 172
- *El Pozo de Requiremewts*, página 202
- *Automático ubicuo*, página 230

Desafíos

- ¿Escribiste un comentario explicativo para el código fuente que acabas de escribir? ¿Por qué no? ¿Tener prisa? ¿No estás seguro de si el código realmente funcionará, solo estás probando una idea como prototipo? Tirarás el código a la basura después, ¿verdad? No entrará en el proyecto sin comentar y ser experimental, ¿verdad?
- A veces es incómodo documentar el diseño del código fuente porque el diseño no está claro en su mente; Todavía está evolucionando. No sientes que debes desperdiciar esfuerzo describiendo lo que algo hace hasta que realmente lo hace. ¿Suena esto como una programación por coincidencia (página 172)?

45

Grandes expectativas

Espantaos, oh vosotros, que por esto, por ser horriblemente afligidos...

► **Jeremías 2:12**

Una empresa anuncia beneficios récord y el precio de sus acciones cae un 20%. Las noticias financieras de esa noche explican que la compañía no cumplió con las expectativas de los analistas. Un niño abre un costoso regalo de Navidad y rompe a llorar: no era la muñeca barata que el niño esperaba. Un equipo de proyecto hace milagros para implementar una aplicación fenomenalmente compleja, solo para que sus usuarios la rechacen porque no tiene un sistema de ayuda.

En un sentido abstracto, una aplicación tiene éxito si implementa correctamente sus especificaciones. Desafortunadamente, esto solo paga facturas abstractas.

En realidad, el éxito de un proyecto se mide por lo bien que satisface las *expectativas* de sus usuarios. Un proyecto que no cumple con sus expectativas se considera un fracaso, sin importar cuán bueno sea el entregable en términos absolutos. Sin embargo, al igual que el padre del niño que espera la muñeca barata, si vas demasiado lejos, tú también serás un fracaso.

CONSE

Supere con creces las expectativas de sus usuarios

Sin embargo, la ejecución de este consejo requiere algo de trabajo.

Comunicar las expectativas

Inicialmente, los usuarios acuden a ti con una visión de lo que quieren. Puede ser incompleto, inconsistente o técnicamente imposible, pero es *suyo* y, como el niño en Navidad, tiene alguna emoción invertida en él. No puedes simplemente ignorarlo.

A medida que se desarrolla la comprensión de sus necesidades, encontrará áreas en las que sus expectativas no pueden cumplirse, o en las que sus expectativas son quizás demasiado conservadoras. Parte de tu papel es comunicar esto. Trabaje con sus usuarios para que entiendan con precisión lo que va a ofrecer. Y hacerlo durante todo el proceso de desarrollo. Nunca pierda de vista los problemas empresariales que su aplicación está destinada a resolver.

Algunos consultores llaman a este proceso "gestión de expectativas", es decir, controlar activamente lo que los usuarios deben esperar obtener de sus sistemas. Creemos que esta es una posición un tanto elitista. Nuestro papel no es controlar las esperanzas de nuestros usuarios. En cambio, necesitamos trabajar con ellos para llegar a un entendimiento común del proceso de desarrollo y el resultado final, junto con las expectativas que aún no han verbalizado. Si el equipo se comunica con fluidez con el mundo exterior, este proceso es casi automático; Todo el mundo debe entender lo que se espera y cómo se construirá.

Hay algunas técnicas importantes que se pueden utilizar para facilitar este proceso. De éstas, *las Balas Tracer*, página 48, y *los Prototipos de Notas Post-it*, página 53, son los más importantes. Ambos permiten que el equipo construya algo que el usuario pueda ver. Ambas son formas ideales de comunicar su comprensión de sus requisitos. Y ambos le permiten a usted y a sus usuarios practicar la comunicación entre sí.

La milla extra

Si trabajas en estrecha colaboración con tus usuarios, compartiendo sus expectativas y comunicando lo que estás haciendo, habrá pocas sorpresas cuando se entregue el proyecto.

Esto es ALGO MALO. Intenta sorprender a tus usuarios. No asustarlos, eso sí, sino *deleitarlos*.

Dales un poco más de lo que esperaban. El esfuerzo adicional que requiere agregar alguna función orientada al usuario al sistema se pagará una y otra vez en buena voluntad.

Escuche a sus usuarios a medida que avanza el proyecto para obtener pistas sobre qué características realmente les encantarían. Algunas cosas que puede agregar con relativa facilidad y que se ven bien para el usuario promedio incluyen:

- Ayuda de Balloon o
ToolTip • Métodos
abreviados de teclado
- Una guía de referencia rápida como complemento del manual
del usuario • Coloración
- Analizadores de archivos de registro
- Instalación automatizada
- Herramientas para comprobar la integridad del sistema
- La capacidad de ejecutar varias versiones del sistema para la
capacitación • Una pantalla de presentación personalizada
para su organización

Todas estas cosas son relativamente superficiales y realmente no sobrecargan el sistema con una sobrecarga de funciones. Sin embargo, cada uno le dice a sus usuarios que el equipo de desarrollo se preocupó por producir un gran sistema, uno que estaba destinado a un uso real. Solo recuerde no romper el sistema agregando estas nuevas funciones.

Las secciones relacionadas incluyen:

- *Good-Ewough Software*, página 9
- *Tracer Bullets*, página 48
- *Prototipos de notas Post-it*, página 53
- *El Pozo de Requirements*, página 202

Desafíos

- A veces, los críticos más duros de un proyecto son las personas que trabajaron en él. ¿Alguna vez has experimentado la decepción de que tus propias expectativas no se cumplieran con algo que produjiste? ¿Cómo puede ser eso? Tal vez haya algo más que lógica en juego aquí.
- ¿Qué comentan tus usuarios cuando entregas software? ¿Su atención a las diversas áreas de la aplicación es proporcional al esfuerzo invertido en cada una de ellas? ¿Qué les encanta?

46

Orgullo y prejuicio

Nos has deleitado hasta ahora.

► Jane Austen, **Orgullo y prejuicio**

Los programadores pragmáticos no eluden la responsabilidad. En cambio, nos regocijamos en aceptar los desafíos y en dar a conocer nuestra experiencia. Si somos responsables de un diseño, o de un fragmento de código, hacemos un trabajo del que podemos estar orgullosos.

CONSE

Firma tu trabajo

Los artesanos de una época anterior se enorgullecían de firmar su trabajo. Tú también deberías estarlo.

Sin embargo, los equipos de proyecto siguen estando formados por personas y esta regla puede causar problemas. En algunos proyectos, la idea de *la interfaz de código* puede causar problemas de cooperación. Las personas pueden volverse territoriales o no estar dispuestas a trabajar en elementos básicos comunes. El proyecto puede terminar como un montón de pequeños feudos insulares. Te vuelves prejuicioso a favor de tu código y en contra de tus compañeros de trabajo.

Eso no es lo que queremos. No debes defender celosamente tu código contra los intrusos; De la misma manera, debes tratar el código de otras personas con respeto. La Regla de Oro ("Haz a los demás lo que te gustaría que te hicieran a ti") y una base de respeto mutuo entre los desarrolladores es fundamental para que este consejo funcione.

El anonimato, especialmente en proyectos grandes, puede proporcionar un caldo de cultivo para el descuido, los errores, la pereza y el mal código. Se vuelve demasiado fácil verse a sí mismo como un engranaje en la rueda, produciendo excusas poco convincentes en interminables informes de estado en lugar de un buen código.

Si bien el código debe ser propiedad, no tiene por qué ser propiedad de un individuo. De hecho, el exitoso método de programación eXtreme de Kent Beck recomienda la propiedad comunitaria del código (pero esto también requiere prácticas adicionales, como la programación en parejas, para

protegerse contra los peligros del anonimato).

Queremos ver el orgullo de propiedad. "Escribí esto y respaldo mi trabajo". Su firma debe llegar a ser reconocida como un indicador de calidad. Las personas deben ver su nombre en un fragmento de código y esperar que sea sólido, bien escrito, probado y documentado. Un trabajo muy profesional. Escrito por un verdadero profesional.

Un programador pragmático.

Esta página se ha dejado en blanco intencionadamente

Apéndice A

Recursos

La única razón por la que pudimos abarcar tanto terreno en este libro es que vimos a muchos de nuestros sujetos desde una gran altitud. Si les hubiéramos dado la cobertura en profundidad que se merecían, el libro habría sido diez veces más largo.

Comenzamos el libro con la sugerencia de que los programadores pragmáticos siempre deberían estar aprendiendo. En este apéndice hemos enumerado recursos que pueden ayudarte con este proceso.

En la sección *Sociedades Profesionales*, damos detalles del IEEE y del ACM. Recomendamos que los Programadores Pragmáticos se unan a una (o ambas) de estas sociedades. Luego, en *Building a Library*, destacamos publicaciones periódicas, libros y sitios web que creemos que contienen información pertinente y de alta calidad (o que simplemente son divertidos).

A lo largo del libro hacemos referencia a muchos recursos de software accesibles a través de Internet. En la sección *Recursos de Internet*, enumeramos las URL de estos recursos, junto con una breve descripción de cada uno. Sin embargo, la naturaleza de la Web significa que muchos de estos enlaces pueden estar obsoletos para el momento en que lea este libro. Puede probar uno de los muchos motores de búsqueda para obtener un enlace más actualizado, o visitar nuestro sitio web en www.pragmaticprogrammer.com y consulta nuestra sección de enlaces.

Finalmente, este apéndice contiene la bibliografía del libro.

Sociedades Profesionales

Existen dos sociedades profesionales de programación de clase mundial: la Association for Computing Machinery (ACM)¹ y la IEEE Computer Society.² Recomendamos que todos los programadores pertenezcan a una (o ambas) de estas sociedades. Además, es posible que los desarrolladores de fuera de los Estados Unidos deseen unirse a sus sociedades nacionales, como la BCS en el Reino Unido.

Ser miembro de una sociedad profesional tiene muchos beneficios. Las conferencias y las reuniones locales le brindan grandes oportunidades para conocer a personas con intereses similares, y los grupos de interés especial y los comités técnicos le brindan la oportunidad de participar en el establecimiento de normas y directrices utilizadas en todo el mundo. También sacará mucho provecho de sus publicaciones, desde discusiones de alto nivel sobre la práctica de la industria hasta teoría de computación de bajo nivel.

Construcción de una biblioteca

Nos gusta mucho leer. Como señalamos en *Tu Portafolio de Knowledge*, página 12, un buen programador siempre está aprendiendo. Mantenerse al día con los libros y las publicaciones periódicas puede ayudar. Estos son algunos de los que nos gustan.

Periódicos

Si eres como nosotros, guardarás revistas y periódicos viejos hasta que estén lo suficientemente apilados como para convertir los de abajo en láminas planas de diamante. Esto significa que vale la pena ser bastante selectivo. He aquí algunas publicaciones periódicas que leemos.

- Computadora IEEE. Enviado a los miembros de la IEEE Computer Society, *Computer* tiene un enfoque práctico pero no le teme a la teoría. Algunas cuestiones se orientan en torno a un tema, mientras que otras son simplemente

1. ACM Member Services, PO Box 11414, Nueva York, NY 10286, EE. UU.
⇒ www.acm.org

2. 1730 Massachusetts Avenue NW, Washington, DC 20036-1992, Estados Unidos.

⇒ www.computer.org

lecciones de artículos interesantes. Este cargador tiene una buena relación señal-ruido.

- Software IEEE. Esta es otra gran publicación bimensual de la IEEE Computer Society dirigida a los profesionales del software.
- Comunicaciones de la ACM. La revista básica recibida por todos los miembros de la ACM, el *MCCA* ha sido un estándar en la industria durante décadas, y probablemente ha publicado más artículos seminales que cualquier otra fuente.
- SIGPLAN. Producido por el Grupo de Interés Especial de ACM sobre Lenguajes de Programación, *SIGPLAN* es una adición opcional a su membresía de ACM. A menudo se utiliza para publicar especificaciones de lenguaje, junto con artículos de interés para todos los que gustan de profundizar en la programación.
- Diario del Dr. Dobbs. Una revista mensual, disponible por suscripción y en los quioscos, *Dr. Dobbs* es peculiar, pero tiene artículos que van desde la práctica a nivel de bits hasta la teoría pesada.
- El Diario de Perl. Si te gusta Perl, probablemente deberías suscribirte a *The Perl Journal* (www.tpj.com).
- Revista de Desarrollo de Software. Una revista mensual que se centra en temas generales de gestión de proyectos y desarrollo de software.

Documentos Comerciales Semanales

Se publican varios periódicos semanales para los desarrolladores y sus gerentes. Estos periódicos son en gran parte una colección de relanzamientos de prensa de la compañía, corregidos como artículos. Sin embargo, el contenido sigue siendo valioso: le permite realizar un seguimiento de lo que está sucediendo, mantenerse al tanto de los anuncios de nuevos productos y seguir las alianzas de la industria a medida que se forjan y se rompen. Sin embargo, no esperes mucha cobertura técnica en profundidad.

Libros

Los libros de informática pueden ser caros, pero si se eligen con cuidado, son una inversión que vale la pena. Es posible que desee consultar nuestros títulos de Pragmatic Bookshelf en <http://pragmaticprogrammer.com>. Además, aquí hay un puñado de los muchos otros libros que nos gustan.

Análisis y diseño

- Construcción de Software Orientado a Objetos, 2^a Edición. El libro épico de Bertrand Meyer sobre los fundamentos del desarrollo orientado a objetos, todo en unas 1.300 páginas [Mey97b].
- Patrones de diseño. Un patrón de diseño describe una forma de resolver una clase particular de problemas a un nivel más alto que un lenguaje de programación. Este libro ahora clásico [GHJV95] de la *Banda de los Cuatro* describe 23 patrones de diseño básicos, incluidos Proxy, Visitor y Singleton.
- Patrones de análisis. Un tesoro de patrones arquitectónicos de alto nivel tomados de una amplia variedad de proyectos del mundo real y destilados en forma de libro. Una forma relativamente rápida de obtener la visión de muchos años de experiencia en el modelado [Fow96].

Equipos y proyectos

- El Mes del Hombre Mítico. El clásico de Fred Brooks sobre los peligros de organizar equipos de proyectos, recientemente actualizado [Bro95].
- Dinámica del desarrollo de software. Una serie de ensayos cortos sobre la construcción de software en grandes equipos, centrándose en la dinámica entre los miembros del equipo, y entre el equipo y el resto del mundo [McC95].
- Surviving Object-Oriented Projects: A Manager's Guide. Los "informes desde las trincheras" de Alistair Cockburn ilustran muchos de los peligros y trampas de la gestión de un proyecto OO, especialmente el primero. El Sr. Cockburn proporciona consejos y técnicas para superar los problemas más comunes [Coc97b].

Entornos específicos

- Unix. W. Richard Stevens tiene varios libros excelentes, entre ellos *Advanced Programming in the Unix Environment* y los libros *Unix Network Programming* [Ste92, Ste98, Ste99].
- Windows. Marshall Brain's *Win32 System Services* [Bra95] es una referencia concisa a las API de bajo nivel. La *programación de Windows* de Charles Petzold [Pet98] es la biblia del desarrollo de GUI de Windows .
- C++. Tan pronto como te encuentres en un proyecto de C++, corre, no camines, a la librería y consigue *C++ Efectivo* de Scott Meyer, y posiblemente *C++ Más Efectivo* [Mey97a, Mey96]. Para construir sistemas de cualquier tamaño apreciable, se necesita el *diseño de software C++ a gran escala* de John Lakos [Lak96]. Para técnicas avanzadas, consulte *Advanced C++ Programming Styles and Modisms* [Cop92] de Jim Coplien.

Además, la serie O'Reilly *Nutshell* (www.ora.com) ofrece tratamientos rápidos y completos de diversos temas y lenguajes como perl, yacc, sendmail, Windows internal y expresiones regulares.

La Web

Encontrar buen contenido en la Web es difícil. Aquí hay varios enlaces que revisamos al menos una vez a la semana.

- Slashdot. Anunciado como "Noticias para nerds. Cosas que importan", Slashdot es uno de los hogares netos de la comunidad Linux. Además de actualizaciones periódicas sobre noticias de Linux, el sitio ofrece información sobre tecnologías que son geniales y problemas que afectan a los desarrolladores.
⇒ www.slashdot.org
- Enlaces de Cetus. Miles de enlaces sobre temas orientados a objetos.
⇒ www.cetus-links.org
- WikiWikiWeb. El Repositorio de Patrones de Portland y la discusión de patrones. No es sólo un gran recurso, el sitio WikiWikiWeb es un interesante experimento de edición colectiva de ideas.
⇒ www.c2.com

Recursos de Internet

Los enlaces a continuación son a recursos disponibles en Internet. Eran válidos en el momento de escribir este artículo, pero (siendo la Red lo que es) es muy posible que estén desactualizados en el momento en que leas esto. Si es así, puedes intentar una búsqueda general de los nombres de archivo, o ir al sitio web de Pragmatic Programmer (www.pragmaticprogrammer.com) y seguir nuestros enlaces.

Editores

Emacs y vi no son los únicos editores multiplataforma, pero están disponibles gratuitamente y son ampliamente utilizados. Un rápido vistazo a una revista como *Dr. Dobbs* mostrará varias alternativas comerciales.

Emacs

Tanto Emacs como XEmacs están disponibles en plataformas Unix y Windows.

[URL 1] El editor de Emacs

⇒ www.gnu.org

Lo último en editores grandes, que contiene todas las características que cualquier editor ha tenido, Emacs tiene una curva de aprendizaje casi vertical, pero se amortiza generosamente una vez que lo has dominado. También es un excelente lector de correo y noticias, libreta de direcciones, calendario y diario, juego de aventuras,

[URL 2] El editor de XEmacs

⇒ www.xemacs.org

Surgido del Emacs original hace algunos años, XEmacs tiene fama de tener componentes internos más limpios y una interfaz más atractiva.

VI

Hay al menos 15 clones vi diferentes disponibles. De estos, vim es probablemente portado a la mayoría de las plataformas, por lo que sería una buena opción de editor si te encuentras trabajando en muchos entornos diferentes.

[URL 3] El editor de Vim

⇒ <ftp://ftp.fu-berlin.de/misc/editors/vim>

De la documentación: "Hay muchas mejoras por encima de vi: deshacer

multinivel, múltiples ventanas y búferes, resaltado de sintaxis, edición de línea de comandos, finalización de nombres de archivo, ayuda en línea, selección visual, etc. . ."

[URL 4] El editor de elvis

⇒ elvis.the-little-red-haired-girl.org

Un clon vi mejorado con soporte para X.

[URL 5] Modo Víbora Emacs

⇒ <http://www.cs.sunysb.edu/~kifer/emacs.html>

Viper es un conjunto de macros que hacen que Emacs se parezca a vi. Algunos pueden dudar de la sabiduría de tomar al editor más grande del mundo y extenderlo para emular a un editor cuya fuerza es su compacidad. Otros afirman que combina lo mejor de ambos mundos.

Compiladores, lenguajes y herramientas de desarrollo

[URL 6] El compilador GNU C/C++

⇒ www.fsf.org/software/gcc/gcc.html

Uno de los compiladores de C y C++ más populares del planeta. También hace Objective-C. (En el momento de escribir este artículo, el proyecto egcs, que anteriormente se escindió de gcc, está en proceso de fusionarse de nuevo en el redil).

[URL 7] El lenguaje Java de Sun

⇒ java.sun.com

Inicio de Java, que incluye SDK descargables, documentación, tutoriales, noticias y mucho más.

[URL 8] Página de inicio del idioma Perl

⇒ www.perl.com

O'Reilly aloja este conjunto de recursos relacionados con Perl.

[URL 9] El lenguaje Python

⇒ www.python.org

El lenguaje de programación orientado a objetos Python es interpretado e interactivo, con una sintaxis ligeramente peculiar y un seguimiento amplio y leal.

[URL 10] PequeñoEiffel

⇒ SmallEiffel.loria.fr

El compilador GNU Eiffel se ejecuta en cualquier máquina que tenga un compilador ANSI C y un entorno de ejecución Posix.

[URL 11] ISE Eiffel

⇒ www.eiffel.com

Interactive Software Engineering es el creador de Design by Contract, y vende un compilador Eiffel comercial y herramientas relacionadas.

[URL 12] Sather

⇒ www.icsi.berkeley.edu/~sather

Sather es un lenguaje experimental que surgió de Eiffel. Su objetivo es apoyar las funciones de orden superior y la abstracción de iteraciones, así como Common Lisp, CLU o Scheme, y ser tan eficiente como C, C++ o Fortran.

[URL 13] VisualWorks

⇒ www.cincom.com

Inicio del entorno Smalltalk de VisualWorks. Las versiones no comerciales para Windows y Linux están disponibles de forma gratuita.

[URL 14] El entorno lingüístico de Squeak

⇒ squeak.cs.uiuc.edu

Squeak es una implementación portátil y de libre acceso de Smalltalk-80 escrita en sí misma; puede producir una salida de código C para un mayor rendimiento.

[URL 15] El lenguaje de programación TOM

⇒ www.gerbil.org/tom

Un lenguaje muy dinámico con raíces en Objective-C.

[URL 16] El Proyecto Beowulf

⇒ www.beowulf.org

Un proyecto que construye computadoras de alto rendimiento a partir de redes de cajas Linux baratas.

[URL 17] iContract: herramienta de diseño por contrato para Java

⇒ www.reliable-systems.com

Diseño por contrato de formalismo de precondiciones, postcondiciones e invariables, implementado como preprocesador para Java. Honra la herencia, las implementaciones, los cuantificadores existenciales y más.

[URL 18] Nana: registro y aserciones para C y C++

⇒ www.gnu.org/software/nana/nana.html

Se ha mejorado la compatibilidad con la comprobación y el registro de aserciones en C y C++. También proporciona cierto soporte para el diseño por contrato.

[URL 19] DDD: depurador de visualización de datos

⇒ <http://www.gnu.org/software/ddd/>

Un front-end gráfico gratuito para depuradores de Unix.

[URL 20] Navegador de refactorización de John Brant

⇒ St-www.cs.uiuc.edu/users/brant/Refactory

Un popular navegador de refactorización para Smalltalk.

[URL 21] Generador de documentación DOC++

⇒ www.zib.de/Visual/software/docpp/index.html

DOC++ es un sistema de documentación para C/C++ y Java que genera salidas LATEX y HTML para una sofisticada navegación en línea de su documentación directamente desde el encabezado de C++ o los archivos de clase Java.

[URL 22] xUnit: marco de pruebas unitarias

⇒ www.XProgramming.com

Un concepto simple pero poderoso, el marco de pruebas unitarias xUnit proporciona una plataforma consistente para probar software escrito en una variedad de idiomas.

[URL 23] El lenguaje Tcl

⇒ www.scriptics.com

Tcl ("Tool Command Language") es un lenguaje de scripting diseñado para ser fácil de incrustar en una aplicación.

[URL 24] Expect: automatiza la interacción con los programas

⇒ expect.nist.gov

Una extensión construida sobre Tcl [URL 23], expect le permite la interacción de scripts con programas. Además de ayudarle a escribir archivos de comandos que (por ejemplo) recuperan archivos de servidores remotos o amplían la potencia de su shell, expect puede ser útil al realizar pruebas de regresión. Una versión gráfica, expectk, le permite envolver aplicaciones que no son GUI con una interfaz de ventanas.

[URL 25] Espacios T

⇒ www.almaden.ibm.com/cs/TSpaces

De su página web: "T Spaces es un búfer de comunicación de red con capacidades de base de datos. Permite la comunicación entre aplicaciones y dispositivos en una red de ordenadores y sistemas operativos heterogéneos. T Spaces proporciona servicios de comunicación grupal, servicios de bases de datos, servicios de transferencia de archivos basados en URL y servicios de notificación de eventos".

[URL 26] javaCC—Compilador-compilador de Java

⇒ www.webgain.com/products/java_cc

Un generador de analizadores que está estrechamente acoplado al lenguaje Java.

[URL 27] El generador de analizadores sintácticos de bisontes

⇒ www.gnu.org/software/bison/bison.html

bison toma una especificación gramatical de entrada y genera a partir de ella el código fuente C de un analizador adecuado.

[URL 28] SWIG: Generador simplificado de envoltorios e interfaces

⇒ www.swig.org

SWIG es una herramienta de desarrollo de software que conecta programas escritos en C, C++ y Objective-C con una variedad de lenguajes de programación de alto nivel como Perl, Python y Tcl/Tk, así como Java, Eiffel y Guile.

[URL 29] El Grupo de Gestión de Objetos, Inc.

⇒ www.omg.org

El OMG es el administrador de varias especificaciones para producir sistemas distribuidos basados en objetos. Su trabajo incluye la Arquitectura de Agente de Solicitud de Objeto Común (CORBA) y el Protocolo Inter-ORB de Internet (IIOP). Combinadas, estas especificaciones hacen posible que los objetos se comuniquen entre sí, incluso si están escritos en diferentes idiomas y se ejecutan en diferentes tipos de computadoras.

Herramientas Unix bajo DOS

[URL 30] Las herramientas de desarrollo de UWIN

⇒ www.gtlinc.com/uwin.html

Global Technologies, Inc., Puente Viejo, Nueva Jersey

El paquete UWIN proporciona bibliotecas de vínculos dinámicos (DLL) de Windows que emulan una gran parte de la interfaz de biblioteca de nivel C de Unix. Usando esta interfaz, GTL ha portado un gran número de herramientas de línea de comandos de Unix a Windows. Véase también [URL 31].

[URL 31] Las herramientas de Cygnus Cygwin

⇒ sourcetware.cygnus.com/cygwin/

Cygnus Solutions, Sunnyvale, CA

El paquete Cygnus también emula la interfaz de la biblioteca C de Unix, y proporciona una gran variedad de herramientas de línea de comandos de Unix bajo el sistema operativo Windows.

[URL 32] Herramientas eléctricas Perl

⇒ www.perl.com/pub/language/ppt/

Un proyecto para reimplementar el conjunto de comandos clásico de Unix en Perl, haciendo que los comandos estén disponibles en todas las plataformas que soportan Perl (y eso es un montón de plataformas).

Herramientas de control de código fuente

[URL 33] RCS: sistema de control de revisiones

⇒ www.cs.purdue.edu/homes/trinkle/RCS/

Sistema de control de código fuente GNU para Unix y Windows NT.

[URL 34] CVS: sistema de versiones simultáneas

⇒ www.cvshome.com

Sistema de control de código fuente de libre acceso para Unix y Windows NT. Amplía RCS al admitir un modelo cliente-servidor y acceso simultáneo a archivos.

[URL 35] Gestión de configuración basada en transacciones de Aegis

⇒ <http://www.canb.auug.org.au/~millerp/aegis.html>

Una herramienta de control de revisiones orientada a procesos que impone estándares de proyecto (como verificar que el código protegido supera las pruebas).

[URL 36] ClearCase

⇒ www.rational.com

Control de versiones, gestión de espacios de trabajo y compilaciones, control de procesos.

[URL 37] Integridad de la fuente MKS

⇒ www.mks.com

Control de versiones y gestión de la configuración. Algunas versiones incorporan características que permiten a los desarrolladores remotos trabajar en los mismos archivos simultáneamente (al igual que CVS).

[URL 38] Gestión de la configuración de PVCS

⇒ www.merant.com

Un sistema de control de código fuente, muy popular para sistemas Windows.

[URL 39] Fuente visual segura

⇒ www.microsoft.com

Un sistema de control de versiones que se integra con las herramientas de desarrollo visual de Microsoft.

[URL 40] Perforce

⇒ www.perforce.com

Un sistema de gestión de configuración de software cliente-servidor.

Otras herramientas

[URL 41] WinZip: utilidad de archivo para Windows

⇒ www.winzip.com

Nico Mak Computiwg, Iwc., Mawsfield, CT

Una utilidad de archivo de archivos basada en Windows. Admite formatos zip y tar.

[URL 42] El caparazón Z

⇒ sunsite.auc.dk/zsh

Un shell diseñado para uso interactivo, aunque también es un potente lenguaje de scripting. Muchas de las características útiles de bash, ksh y tcsh se incorporaron a zsh; Se agregaron muchas características originales.

[URL 43] Un cliente SMB gratuito para sistemas Unix

⇒ samba.anu.edu.au/pub/samba/

Samba le permite compartir archivos y otros recursos entre sistemas Unix y Windows. La samba incluye:

- Un servidor SMB, para proporcionar servicios de impresión y archivos de estilo Windows NT y LAN Manager a clientes SMB como Windows 95, Warp Server, smbfs y otros.
- Un servidor de nombres Netbios, que entre otras cosas proporciona soporte para la navegación. Samba puede ser el navegador principal de tu LAN si lo deseas.
- Un cliente SMB similar a ftp que le permite acceder a los recursos de la PC (discos e impresoras) desde Unix, Netware y otros sistemas operativos.

Ponencias y Publicaciones

[URL 44] Preguntas frecuentes sobre comp.object

⇒ www.cyberdyne-object-sys.com/oofaq2

Un FAQ sustancial y bien organizado para el grupo de noticias comp.object.

[URL 45] Programación extrema

⇒ www.XProgramming.com

Del sitio web: "En XP, utilizamos una combinación muy ligera de prácticas para crear un equipo que puede producir rápidamente software extremadamente confiable, eficiente y bien factorizado. Muchas de las prácticas de XP fueron creadas y probadas como parte del proyecto Chrysler C3, que es un sistema de nómina muy exitoso implementado en Smalltalk".

[URL 46] Página principal de Alistair Cockburn

⇒ members.aol.com/acockburn

Busque "Estructuración de casos de uso con objetivos" y plantillas de casos de uso.

[URL 47] Página principal de Martin Fowler

⇒ ourworld.compuserve.com/homepages/martin_fowler

Autor de *Analysis Patterns* y coautor de *UML Distilled* y *Refactor: Improving the Design of Existing Code*. En la página web de Martin Fowler se habla de sus libros y de su trabajo con la UML.

[URL 48] Página principal de Robert C. Martin

⇒ www.objectmentor.com

Buenos documentos introductorios sobre técnicas orientadas a objetos, incluyendo el análisis de dependencia y las métricas.

[URL 49] Programación orientada a aspectos

⇒ www.parc.xerox.com/csl/projects/aop/

Un enfoque para agregar funcionalidad al código, tanto ortogonal como declarativamente.

[URL 50] Especificación de JavaSpaces

⇒ java.sun.com/products/javaspaces

Un sistema similar a Linda para Java que admite persistencia distribuida y algoritmos distribuidos.

[URL 51] Código fuente de Netscape

⇒ www.mozilla.org

La fuente de desarrollo del navegador Netscape.

[URL 52] El archivo de jerga

⇒ www.jargon.org

Eric S. Raymond

Definiciones para muchos términos comunes (y no tan comunes) de la industria informática, junto con una buena dosis de folclore.

[URL 53] Papeles de Eric S. Raymond

⇒ www.tuxedo.org/~esr

Los artículos de Eric sobre *The Cathedral and the Bazaar* y *Homesteading the no-space* describen las bases psicosociales y las implicaciones del movimiento Open Source.

[URL 54] El entorno de escritorio K

⇒ www.kde.org

De su página web: "KDE es un potente entorno de escritorio gráfico para estaciones de trabajo Unix. KDE es un proyecto de Internet y verdaderamente abierto en todos los sentidos".

[URL 55] El Programa de Manipulación de Imágenes GNU

⇒ www.gimp.org

Gimp es un programa de distribución gratuita que se utiliza para la creación, composición y retoque de imágenes.

[URL 56] El Proyecto Deméter

⇒ www.ccs.neu.edu/research/demeter

La investigación se centró en hacer que el software sea más fácil de mantener y evolucionar utilizando la Programación Adaptativa.

Misceláneo

[URL 57] El Proyecto GNU

⇒ www.gnu.org

Fundación para el Software Libre, Boston, MA

La Free Software Foundation es una organización benéfica exenta de impuestos que recauda fondos para el proyecto GNU. El objetivo del proyecto GNU es producir un sistema completo, libre y similar a Unix. Muchas de las herramientas que han desarrollado a lo largo del camino se han convertido en estándares de la industria.

[URL 58] Información del servidor web

⇒ www.netcraft.com/survey/servers.html

Enlaces a las páginas de inicio de más de 50 servidores web diferentes. Algunos son productos comerciales, mientras que otros son de libre acceso.

Bibliografía

[Bak72] F. T. Panadero. Jefe de programación de la gestión del equipo de programación de producción. *Revista de sistemas de IBM*, 11(1):56–73, 1972.

[BBM96] V. Basili, L. Briand y W. L. Melo. Una validación de las métricas de diseño orientado a objetos como indicadores de calidad. *IEEE Trans- actions on Software Engineering*, 22(10):751–761, octubre de 1996.

[Ber96] Albert J. Bernstein. *Cerebros de dinosaurios: lidiando con todas esas personas imposibles en el trabajo*. Ballantine Books, Nueva York, NY, 1996.

- [Bra95] Cerebro Marshall. *Servicios del sistema Win32*. Prentice Hall, Acantilados de Englewood, Nueva Jersey, 1995.
- [Bro95] Frederick P. Brooks, Jr. *El Mes del Hombre Mítico: Ensayos sobre Ingeniería de Software*. Addison-Wesley, Reading, MA, edición de aniversario, 1995.
- [CG90] N. Carriero y D. Gelenter. *Cómo escribir programas paralelos: un primer curso*. MIT Press, Cambridge, MA, 1990.
- [Clao04] Mike Clark. *Automatización pragmática de proyectos*. Los Programadores Pragmáticos, LLC, Raleigh, Carolina del Norte, y Dallas, TX, 2004.
- [CN91] Brad J. Cox y Andrex J. Novobilski. *Programación Orientada a Objetos, Un Enfoque Evolutivo*. Addison-Wesley, Reading, MA, 1991.
- [Coc97a] Alistair Cockburn. Objetivos y casos de uso. *Revista de Programación Orientada a Objetos*, 9(7):35–40, septiembre de 1997.
- [Coc97b] Alistair Cockburn. *Surviving Object-Oriented Projects: A Manager's Guide*. Addison Wesley Longman, Reading, MA, 1997.
- [COP92] James O. Coplien. *Estilos y modismos avanzados de programación de C++*. Addison-Wesley, Reading, MA, 1992.
- [DL99] Tom DeMarco y Timothy Lister. *Peopleware: proyectos y equipos productivos*. Dorset House, Nueva York, NY, segunda edición, 1999.
- [FBB⁺99] Martin Fowler, Kent Beck, John Brant, William Opdyke y Don Roberts. *Refactorización: mejora del diseño del código existente*. Addison Wesley Longman, Reading, MA, 1999.
- [Fow96] Martin Fowler. *Patrones de análisis: modelos de objetos reutilizables*. Addison Wesley Longman, Reading, MA, 1996.
- [FS99] Martin Fowler y Kendall Scott. *Destilado UML: Aplicación del lenguaje estándar de modelado de objetos*. Addison Wesley Longman, Reading, MA, segunda edición, 1999.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. *Patrones de diseño: elementos de software orientado a objetos reutilizable*. Addison-Wesley, Reading, MA, 1995.

- [Gla99a] Robert L. Glass. Inspecciones: algunos hallazgos sorprendentes. *Comunicaciones de la ACM*, 42(4):17–19, abril de 1999.
- [Gla99b] Robert L. Glass. Las realidades de los beneficios de la tecnología de software. *Comunicaciones de la ACM*, 42(2):74–79, febrero de 1999.
- [Hol78] Michael Holt. *Acertijos y juegos matemáticos*. Dorset Press, Nueva York, NY, 1978.
- [HT03] Andy Hunt y Dave Thomas. *Pruebas unitarias pragmáticas en Java con JUnit*. Los Programadores Pragmáticos, LLC, Raleigh, Carolina del Norte, y Dallas, TX, 2003.
- [jac94] Ivar Jacobson. *Ingeniería de software orientada a objetos: un enfoque basado en casos de uso*. Addison-Wesley, Reading, MA, 1994.
- [KLM⁺97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier y John Irwin. Programación orientada a aspectos. En *European Conference on Object-Oriented Programming (ECOOP)*, vol. LNCS 1241. Springer-Verlag, junio de 1997.
- [KNU97A] Donald Ervin Knuth. *El Arte de la Programación de Computadoras: Algoritmos Funcionales*, volumen 1. Addison Wesley Longman, Reading, MA, tercera edición, 1997.
- [KNU97b] Donald Ervin Knuth. *El arte de la programación de computadoras: algoritmos seminuméricos*, volumen 2. Addison Wesley Longman, Reading, MA, tercera edición, 1997.
- [Knu98] Donald Ervin Knuth. *El Arte de la Programación de Computadoras: Clasificación y Búsqueda*, volumen 3. Addison Wesley Longman, Reading, MA, segunda edición, 1998.
- [KP99] Brian W. Kernighan y Rob Pike. *La práctica de la programación*. Addison Wesley Longman, Reading, MA, 1999.
- [Kru98] Philippe Kruchten. *El Proceso Racional Unificado: Una Introducción*. Addison Wesley Longman, Reading, MA, 1998.
- [Lak96] Juan Lakos. *Diseño de software C++ a gran escala*. Addison Wesley Longman, Reading, MA, 1996.

- [LH89] Karl J. Lieberherr e Ian Holland. Asegurando un buen estilo para programas orientados a objetos. *IEEE Software*, páginas 38-48, septiembre de 1989.
- [Lis88] Bárbara Liskov. Abstracción y jerarquía de datos. *Avisos SIGPLAN*, 23(5), mayo de 1988.
- [LMB92] John R. Levine, Tony Mason y Doug Brown. *Lex y Yacc*. O'Reilly & Associates, Inc., Sebastopol, CA, segunda edición, 1992.
- [McC95] Jim McCarthy. *Dinámica del desarrollo de software*. Microsoft Press, Redmond, WA, 1995.
- [Mey96] Scott Meyers. *C++ más eficaz: 35 nuevas formas de mejorar sus programas y diseños*. Addison-Wesley, Reading, MA, 1996.
- [Mey97a] Scott Meyers. *C++ efectivo: 50 formas específicas de mejorar tus programas y diseños*. Addison Wesley Longman, Reading, MA, segunda edición, 1997.
- [Mey97b] Bertrand Meyer. *Construcción de software orientado a objetos*. Prentice Hall, Englewood Cliffs, Nueva Jersey, segunda edición, 1997.
- [Pet98] Charles Petzold. *Programación de Windows, la guía definitiva de la API de Win32*. Microsoft Press, Redmond, WA, quinta edición, 1998.
- [Sch95] Bruce Schneier. *Criptografía aplicada: protocolos, algoritmos y código fuente en C*. John Wiley & Sons, Nueva York, NY, segunda edición, 1995.
- [Sed83] Robert Sedgewick. *Algoritmos*. Addison-Wesley, Reading, MA, 1983.
- [Sed92] Robert Sedgewick. *Algoritmos en C++*. Addison-Wesley, Reading, MA, 1992.
- [SF96] Robert Sedgewick y Phillip Flajolet. *Introducción al análisis de algoritmos*. Addison-Wesley, Reading, MA, 1996.
- [Ste92] W. Richard Stevens. *Programación avanzada en el entorno Unix*. Addison-Wesley, Reading, MA, 1992.

- [Ste98] W. Richard Stevens. *Programación de redes Unix, Volumen 1: API de redes: sockets y xti*. Prentice Hall, Englewood Cliffs, NJ, segunda edición, 1998.
- [Ste99] W. Richard Stevens. *Programación de Redes Unix, Volumen 2: Comunicaciones entre Procesos*. Prentice Hall, Englewood Cliffs, NJ, segunda edición, 1999.
- [Str35] James Ridley Stroop. Estudios de interferencia en reacciones verbales seriadas. *Revista de Psicología Experimental*, 18: 643-662, 1935.
- [TFH04] Dave Thomas, Chad Fowler y Andy Hunt. *Programming Ruby, La Guía Pragmática del Programador*. Los Programadores Pragmáticos, LLC, Raleigh, Carolina del Norte, y Dallas, TX, 2004.
- [TH03] Dave Thomas y Andy Hunt. *Control pragmático de versiones mediante CVS*. Los Programadores Pragmáticos, LLC, Raleigh, Carolina del Norte, y Dallas, TX, 2003.
- [WK82] James Q. Wilson y George Kelling. La seguridad de la policía y de los vecinos. *The Atlantic Monthly*, 249(3):29–38, marzo de 1982.
- [YC86] Edward Yourdon y Larry L. Constantine. *Diseño Estructurado: Fundamentos de una Disciplina de Diseño de Programas y Sistemas de Computación*. Prentice Hall, Englewood Cliffs, NJ, segunda edición, 1986.
- [You95] Edward Yourdon. Cuando un software lo suficientemente bueno es mejor. *IEEE Software*, mayo de 1995.

Apéndice B

Respuestas a los ejercicios

Ejercicio 1: de Ortogonalidad en la página 43

Está escribiendo una clase llamada `Split`, que divide las líneas de entrada en campos. ¿Cuál de las siguientes dos firmas de clase Java es el diseño más ortogonal?

```
class Split1 {  
    public Split1(InputStreamReader rdr) { ...  
    public void readNextLine() lanza IOException { ...  
    public int numFields() { ...  
    public String getField(int fieldNo) { ...  
}  
class Split2 {  
    public Split2(String line) { ...  
    public int numFields() { ...  
    public String getField(int fieldNo) { ...  
}
```

Respuesta 1: A nuestro modo de ver, la clase `Split2` es más ortogonal. Se concentra en su propia tarea, dividiendo líneas, e ignora detalles como de dónde provienen las líneas. Esto no solo hace que el código sea más fácil de desarrollar, sino que también lo hace más flexible. `Split2` puede dividir líneas leídas de un archivo, generadas por otra rutina o pasadas a través del entorno.

Ejercicio 2: de Ortogonalidad en la página 43

¿Qué dará lugar a un diseño más ortogonal: cuadros de diálogo modales o no modales?

Respuesta 2: Si se hace correctamente, probablemente no modal. Un sistema que utiliza cuadros de diálogo sin modo estará menos preocupado por lo que está sucediendo en un momento determinado. Es probable que tenga una mejor estructura de comunicaciones entre módulos que un sistema modal, que puede tener supuestos incorporados sobre el estado del sistema, supuestos que conducen a un mayor acoplamiento y una ortogonalidad disminuida.

Ejercicio 3: de Ortogonalidad en la página 43

¿Qué hay de los lenguajes procedimentales frente a la tecnología de objetos?
 ¿Cuál resulta en un sistema más ortogonal?

Respuesta 3: Esto es un poco complicado. La tecnología de objetos *puede* proporcionar un sistema más ortopédico, pero debido a que tiene más características de las que abusar, en realidad es más fácil crear un *sistema de ortodoxia* utilizando objetos que utilizando un lenguaje de procedimientos. Características como la herencia múltiple, las excepciones, la sobrecarga de operadores y la anulación del método principal (a través de subclases) proporcionan una amplia oportunidad para aumentar el acoplamiento de formas no obvias.

Con tecnología de objetos y un poco de esfuerzo extra, se puede conseguir un sistema mucho más ortogonal. Pero si bien siempre se puede escribir "código de espagueti" en un lenguaje procesal, los lenguajes orientados a objetos mal utilizados pueden agregar albóndigas a sus espaguetis.

Ejercicio 4: de Prototipos y Post-it en la página 56

Al departamento de marketing le gustaría sentarse y hacer una lluvia de ideas con usted sobre algunos diseños de páginas web. Están pensando en mapas de imágenes en los que se puede hacer clic para llevarlo a otras páginas, y así sucesivamente. Pero no pueden decidirse por un modelo para la imagen, tal vez sea un automóvil, un teléfono o una casa. Tiene una lista de páginas y contenido de destino; Les gustaría ver algunos prototipos. Ah, por cierto, tienes 15 minutos. ¿Qué herramientas podrías utilizar?

Respuesta 4: ¡Baja tecnología al rescate! Dibuja algunas caricaturas con marcadores en una pizarra: un automóvil, un teléfono y una casa. No tiene que ser un gran arte; Los contornos de las figuras de palo están bien. Coloque notas adhesivas que describan el contenido de las páginas de destino en las áreas en las que se puede hacer clic. A medida que avanza la reunión, puede refinar los dibujos y las ubicaciones de las notas adhesivas.

Ejercicio 5: de Idiomas de dominio en la página 63

Queremos implementar un mini-lenguaje para controlar un paquete de dibujo simple (tal vez un sistema de gráficos de tortuga). El lenguaje consiste en comandos de una sola letra. Algunos comandos van seguidos de un solo número. Por ejemplo, en la siguiente entrada se dibujaría un rectángulo.

```
P 2 # seleccionar bolígrafo 2
D      # pluma
W 2 # dibujar hacia el
oeste 2 cm N 1 # luego
norte 1 E 2 # luego este
2
S 1 # luego de vuelta al sur
U      # bolígrafo
```

Implemente el código que analiza este lenguaje. Debe estar diseñado para que sea sencillo agregar nuevos comandos.

Respuesta 5: Debido a que queremos que el lenguaje sea extensible, haremos que la tabla del analizador sea controlada. Cada entrada de la tabla contiene la letra de comando, un indicador para indicar si se requiere un argumento y el nombre de la rutina a la que se debe llamar para controlar ese comando en particular.

```

typedef struct {
    carbonizar Cmd;           /* el mandar carta */
    Int hasArg;             /* hace eso tomar un argumento */
    void (*func)(int, int); /* rutina para llamar */
} Comando;

static Comando cmds[] = {
    { 'P', ARG doSelectPen },
    { 'U', NO_ARG, doPenUp },
    { 'D', NO_ARG, doPenDown },
    { 'N', ARG doPenDir },
    { 'E', ARG doPenDir },
    { 'S', ARG doPenDir },
    { 'W', ARG doPenDir }
};

};
```

El programa principal es bastante simple: lea una línea, busque el comando, obtenga el argumento si es necesario, luego llame a la función de controlador.

```

while (fgets(buff, sizeof(buff), stdin)) {
    Comando *cmd = findCommand(*buff);
    if (cmd) {
        Int Arg = 0;
        if (cmd->hasArg && !getArg(buff+1, &arg)) { fprintf(stderr,
            "'%c' necesita un argumento\n", *buff); continuar;
        }
        cmd->func(*buff, arg);
    }
}
```

La función que busca un comando realiza una búsqueda lineal de la tabla, devolviendo la entrada coincidente o NULL.

```

Comando *findCommand(int cmd) {
    Int i;
    para (i = 0; i < ARRAY_SIZE(cmds); i++) {
        if (cmds[i].cmd == cmd)
            devolver cmd + i;
    }
    fprintf(stderr, "Comando desconocido '%c'\n", cmd);
    devuelve 0;
}
```

Finalmente, leer el argumento numérico es bastante simple usando sscanf.

```

int getArg(const char *buff, int *resultado) {
    return sscanf(buff, "%d", resultado) == 1;
}
```

Ejercicio 6: de Idiomas de dominio en la página 63

Diseñe una gramática BNF para analizar una especificación de tiempo. Deben aceptarse todos los ejemplos siguientes.

16h, 19:38h, 23:42, 15:16, 3:16

Respuesta 6: Usando BNF, una especificación de tiempo podría ser

```

<Hora> ::= <hora> <AMPM> |
            <Hora> : <Miwute> <ampm>
            > | <Hora> : <Miwute>

<AMPM> ::= A M | P M

<hora> ::= <cifra> |
            <dígito> <dígito>

<miwute> ::= <dígito> <dígito>

<cifra> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  
```

Ejercicio 7: de Idiomas de dominio en la página 63

Implemente un analizador sintáctico para la gramática BNF en el Ejercicio 6 utilizando yacc, bison, o un analizador-generador similar.

Respuesta 7: Codificamos nuestro ejemplo usando bison, la versión GNU de yacc. Para mayor claridad, aquí solo mostramos el cuerpo del analizador. Consulte la fuente en nuestro sitio web para ver la implementación completa.

```

Hora:      Especificaciones END_TOKEN
          { if ($1 >= 24*60) yyerror("El tiempo es demasiado
                                      grande"); printf("%d minutos después de la
                                      medianoche\n", $1); salida(0);
          }
;

Especificaciones: hora ":" minuto
                  { $$ = $1 + $3;
                  }
                | hora ":" minuto ampm
                  { if ($1 > 11*60) yyerror("Hora fuera de rango");
                  $$ = $1 + $3 + $4;
                  }
                | Hora ampm
                  { if ($1 > 11*60) yyerror("Hora fuera de rango");
                  $$ = $1 + $2;
                  }
;
hora:       hour_num
          { if ($1 > 23) yyerror("Hora fuera de rango");
          $$ = $1 * 60;
          };
  
```

```

minuto: DÍGITO DÍGITO
{ $$ = $1*10 + $2;
  if ($$ > 59) yyerror("minuto fuera de rango");
}

AMPM:   AM           { $$ = AM_MINS; }
        | PM          { $$ = PM_MINS; }
;

hour_num: CIFRA          { $$ = $1; }
        | DÍGITO DÍGITO { $$ = $1*10 + 2 dólares; }
;

```

Ejercicio 8: de Idiomas de dominio en la página 63

Implemente el analizador de tiempo usando Perl. [Hiwt: Las expresiones regulares son buenos analizadores.]

Respuesta 8:

```

$_ = turno;
/^(\d \d?)(am|pm)$/      && doTime($1, 0, $2, 12);
/^(\d \d?)( \d\d)(am|pm)$/ && doTime($1, $2, $3, 12);
/^(\d \d?)( \d\d)/       && doTime($1, $2, 0, 24);
die "Tiempo no válido $_\n";
#
# doTime(hora, min, ampm, maxHour) #
sub doTime($$$$) {
    my ($hour, $min, $offset, $maxHour) = @_;
    die "Hora inválida : $hour" if ($hour >= $maxHour);
    $hour += 12 if ($offset ecualizador "pm");
    imprimir $hour*60 + $min, " minutos después de la medianoche\n";
    salida(0);
}

```

Ejercicio 9: de Estimación en la página 69

Se le pregunta "¿Qué tiene un ancho de banda más alto: una línea de comunicaciones de 1 Mbps o una persona que camina entre dos computadoras con una cinta completa de 4 GB en el bolsillo?" ¿Qué restricciones pondrá a su respuesta para asegurarse de que el alcance de su respuesta sea correcto? (Por ejemplo, podría decir que se omite el tiempo que se tarda en acceder a la cinta).

Respuesta 9: Nuestra respuesta debe formularse en varios supuestos:

- La cinta contiene la información que necesitamos para ser transferida.
- Sabemos la velocidad a la que camina la persona.
- Conocemos la distancia entre las máquinas.
- No tenemos en cuenta el tiempo que se tarda en transferir información hacia y desde la cinta.

- La sobrecarga de almacenar datos en una cinta es aproximadamente igual a la sobrecarga de enviarlos a través de una línea de comunicaciones.

Ejercicio 10: de Estimación en la página 69

Entonces, ¿cuál tiene el mayor ancho de banda?

Respuesta 10: Sujeto a las advertencias de la Respuesta 9: Una cinta de 4 GB contiene 32×10^9 bits, por lo que una línea de 1 Mbps tendría que bombear datos durante unos 32,000 segundos, o aproximadamente 9 horas, para transferir la cantidad equivalente de información. Si la persona camina a una $3\frac{1}{2}$ velocidad por hora constante, entonces nuestras dos máquinas tendrían que estar a una velocidad

Menos 31 kilómetros de distancia para que la línea de comunicaciones supere a nuestro mensajero.

De lo contrario, la persona gana.

Ejercicio 11: de Manipulación de texto en la página 102

El programa C usa un tipo enumerado para representar uno de los 100 estados. Le gustaría poder imprimir el estado como una cadena (en lugar de un número) con fines de depuración. Escriba un script que lea de la entrada estándar un archivo que contenga

```
Nombre
state_a
state_b
:
:
```

Genere el archivo *wame.h*, que contiene

```
extern const char* NAME_names[];
typedef
enumeración {
    state_a,
    state_b,
    :
} NOMBRE;
```

y el fichero *wame.c*, que contiene

```
const char* NAME_names[] = {
    "state_a",
    "state_b",
    :
};
```

Respuesta 11: Implementamos nuestra respuesta usando Perl.

```

mi @consts;
mi $name = <>;
die "Formato no válido - nombre faltante" a menos que se
defina($name); masticar $name;
# Leer en el resto del archivo
while (<>) {
    masticar;
    s/^\\s*//; s/\\s*$/;;
    die "Línea inválida : $_" a menos que /^(\w+)$/;
    empuje @consts, $_;
}
# Ahora genera el archivo
open (HDR, ">$name.h") o morir "No se puede abrir $name.h: $!";
open (SRC, ">$name.c") o die "Can't open $name.c: $!";
my nombre_uc = uc($name);
my $array_nombre = $uc_nombre . "_names";
print HDR /* Archivo generado automáticamente - no editar */\n";
print HDR "extern const char *$ {array_name}[];";
print HMR "typedef enum {\n";
imprimir la unión HDR ",\n",
@consts; print HMR "\n"
$uc_nombre;\n n\n";
print SRC /* Archivo generado automáticamente - no editar */\n";
Impresión FUENTE "const char *$ {array_name}[] = {\n           \\""; 
Impresión FUENTE juntar " ",\n           \\"", @consts;
imprimir SRC "\n}; n";
cerrar(SRC);
cerrar (HDR);

```

Usando el *principio DRY*, no cortaremos y pegaremos este nuevo archivo en nuestro código. En su lugar, lo #include: el archivo plano es la fuente principal de estas constantes. Esto significa que necesitaremos un makefile para regenerar el encabezado cuando el archivo cambie. El siguiente extracto es del banco de pruebas en nuestro árbol de fuentes (disponible en el sitio web).

```

etest.c etest.h: etest.inc enumerated.pl
Perl enumerated.pl etest.inc

```

Ejercicio 12: de Manipulación de texto en la página 102

A mitad de camino de escribir este libro, nos dimos cuenta de que no habíamos puesto el uso de la directiva **estricta** en muchos de nuestros ejemplos de Perl. Escriba un script que recorra los archivos **.pl** en un directorio y agregue un **use estricto** al final del bloque de comentarios inicial a todos los archivos que aún no tengan uno. Recuerde mantener una copia de seguridad de todos los archivos que modifique.

Respuesta 12: Esta es nuestra respuesta, escrita en Perl.

```

mi $dir = desplazar o morir "Directorio faltante";
para mi $file (globe("$dir/*.pl")) {
    open(IP, "$file") o die "Opening $file: $!";
    UNDEF $/;                      # Giro apagado separador de
    registros de entrada -- mi $content = <IP>; # leer entero
    archivo Al unísono cuerda. cerrar(propiedad
    intelectual);
    if ($content !~ /use strict/m) {
        renombrar $file, "$file.bak" o muere "Renombrando $file: $!";
        open(OP, ">$file") o die "Creando $file: $!";
        # Poner 'use strict' en la primera
        lnea que no comienza '#'
        $content =~ s/^(!#)/nuse estricto;\n/m;
        imprimir OP $content;
        cerrar(OP);
        print "Actualizado $file n";
    }
    else {
        imprimir "$file ya estricta n";
    }
}

```

Ejercicio 13: de Generadores de código en la página 106

Escriba un generador de código que tome el archivo de entrada de la Figura 3.4, página 106, y genere la salida en dos idiomas de su elección. Intenta que sea fácil agregar nuevos idiomas.

Respuesta 13: Usamos Perl para implementar nuestra solución. Carga dinámicamente un módulo para generar el idioma solicitado, por lo que agregar nuevos idiomas es fácil. La rutina principal carga el back-end (en función de un parámetro de línea de comandos), luego lee su entrada y llama a rutinas de generación de código basadas en el contenido de cada línea. No somos particularmente exigentes con el manejo de errores: sabremos bastante rápido si las cosas salen mal.

```

mi $lang = cambiar o morir "Falta el idioma";
$lang .= "_cg.pm";
requieren "$lang" o die "No se pudo cargar
$lang"; # Leer y analizar el archivo
mi $name;
while (<>) {
    masticar;
    si (/^\s*$/)
        { CG::blankLine(); }
    elsif (/^#/)
        { CG::comentario($1); }
    elsif (/^M\s*(.+)/) { CG::startMsg($1); $name = 1 dólar; }
    elsif (/^E/)
        { CG::endMsg($name); }
    elsif (/^F\s*(\w+)\s+(\w+)$/)
        { CG::simpleType($1,$2); }
}

```

```

elsif (/^F\s*(\w+)\s+(\w+)\[(\d+)\]\$/)
    { CG::arrayType($1,$2,$3); }
else {
    die "Línea no válida: $_";
}
}

```

Escribir un back-end de lenguaje es simple: proporcione un módulo que implemente los seis puntos de entrada requeridos. Aquí está el generador de C:

```

#!/usr/bin/perl -w
paquete CG;
uso estricto;
# Generador de código para 'C' (ver cg_base.pl)
sub blankLine() { print "\n"; }
sub comentario() { Impresión /*$_[0]*\n"; }
sub startMsg() { print "typedef struct {\n"; }
sub endMsg() { Impresión } $_[0];\n\n"; }

sub arrayType() {
    mi ($name, $type, $size) = @_;
    imprimir "$name $type [$size]; n";
}
sub simpleType() {
    mi ($name, $type) = @_;
    imprimir "$type $name ; n";
}
1;

```

Y aquí está el de Pascal:

```

#!/usr/bin/perl -w
paquete CG;
uso estricto;
# Generador de código para 'Pascal' (ver cg_base.pl)
sub blankLine() { print "\n"; }
sub comentario() { Impresión "{$_[0]}\n"; }
sub startMsg() { print "$_[0] = registro empaquetado \n"; }
sub endMsg() { Impresión "Fin;\n\n"; }

sub arrayType() {
    mi ($name, $type, $size) = @_;
    $size--;
    print "$name: matriz[0..$size] de $type; \n";
}
sub simpleType() {
    mi ($name, $type) = @_;
    imprimir "$name: $type; n";
}
1;

```

Ejercicio 14: de Diseño por Contrato en la página 118

¿Qué hace que un contrato sea bueno? Cualquiera puede añadir condiciones previas y posteriores, pero ¿le servirán de algo? Peor aún, ¿realmente harán más daño que bien? Para el siguiente ejemplo y para los de los Ejercicios 15 y 16, decida si el contrato especificado es bueno, malo o feo, y explique por qué.

Primero, veamos un ejemplo de Eiffel. Aquí tenemos una rutina para agregar un STRING a una lista circular doblemente enlazada (recuerde que las condiciones previas se etiquetan con require y las condiciones posteriores con ensure).

```
-- Agregar un elemento único a una lista doblemente enlazada,
-- y devuelve el NODO recién creado.

add_item(elemento : STRING) : Se
    require el nodo
        artículo /= Nulo
        find_item(artículo) = Vacío
Diferido
asegurar
        resultado.siguiente.anterior = resultado --
        Comprobar el recién resultado.anterior.siguiente =
            resultado -- Añadido De los nodos Enlaces.
        find_item(artículo) = resultado -- Deber
            encontrar eso.
fin
```

Respuesta 14: Este ejemplo de Eiffel es *bueno*. Requerimos que se pasen datos no nulos y garantizamos que se respete la semántica de una lista circular doblemente enlazada. También ayuda poder encontrar la cadena que almacenamos. Debido a que esta es una clase diferida, la clase real que la implementa es libre de usar cualquier mecanismo subyacente que desee. Puede elegir usar punteros, o una matriz, o lo que sea; Mientras cumpla con el contrato, no nos importa.

Ejercicio 15: de Diseño por Contrato en la página 119

A continuación, probemos un ejemplo en Java, algo similar al ejemplo de Ejercicio 14. insertNumber inserta un número entero en una lista ordenada. Las condiciones previas y posteriores se etiquetan como en iContract (véase [URL 17]).

```
datos int privados [];
/*
 * @post datos[índice-1] < datos[índice] &&
 *           data[índice] == aValue
 */
public Node insertNumber (int final aValue)
{
    int índice = findPlaceToInsert(aValue);
    --
```

Respuesta 15: Esto es *bad*. Las matemáticas de la cláusula de índice (index-1) no funcionarán en condiciones de contorno como la primera entrada. La condición posterior supone una implementación particular: queremos que los

contratos sean más abstractos que eso.

Ejercicio 16: de Diseño por Contrato en la página 119

Este es un fragmento de una clase de pila en Java. ¿Es este un buen contrato?

```
/***
 * @pre anItem != null      Requieren datos reales
 * @post pop() == anItem   Verifica que esté
 *                        en la pila
 */
public void push (String final anItem)
```

Respuesta 16: Es un buen contrato, pero una mala implementación. Aquí, el infamante "Heisenbug" [URL 52] asoma su *fea* cabeza. Es probable que el programador haya cometido un simple error tipográfico: *pop* en lugar de *top*. Si bien este es un ejemplo simple y artificioso, los efectos secundarios en las aserciones (o en cualquier lugar inesperado del código) pueden ser muy difíciles de diagnosticar.

Ejercicio 17: de Diseño por Contrato en la página 119

Los ejemplos clásicos de DBC (como en los ejercicios 14 a 16) muestran una implementación de un ADT (tipo de datos abstractos), normalmente una pila o cola. Pero no mucha gente realmente escribe este tipo de clases de bajo nivel.

Entonces, para este ejercicio, diseñe una interfaz para una licuadora de cocina. Con el tiempo, será una licuadora basada en la web, habilitada para Internet y CORBAFI, pero por ahora solo necesitamos la interfaz para controlarla. Tiene diez configuraciones de velocidad (o significa apagado). No puede operarlo vacío y puede cambiar la velocidad solo una unidad a la vez (es decir, de 0 a 1 y de 1 a 2, no de 0 a 2).

Estos son los métodos. Añadan las condiciones previas y posteriores apropiadas y un invariable.

```
int getSpeed()
void setSpeed(int x)
boolean isFull()
void fill()
void vacío()
```

Respuesta 17: Mostraremos las firmas de funciones en Java, con las condiciones previas y posteriores etiquetadas como en iContract.

En primer lugar, el invariante de la clase:

```
/***
 * @invariant getSpeed() > 0
 *           Implica isFull()
 * @invariant getSpeed() >= 0 &&
 *           getSpeed() < 10
 */

```

No te quedes sin nada

Comprobación de la autonomía

A continuación, las condiciones previas y posteriores:

```

    /**
     * @pre Math.abs(getSpeed() - x) <= 1 // Solo cambia por uno
     * @pre x >= 0 && x < 10
     * @post getSpeed() == x
     */
    public void setSpeed(final int x)
    /**
     * @pre !isFull()
     * @post isFull()
     */
    relleno de vacío ()
    /**
     * @pre isFull()
     * @post !isFull()
     */
    void vacío()

```

Ejercicio 18: de Diseño por Contrato en la página 119

¿Cuántos números hay en la serie 0, 5, 10, 15, ..., 100?

Respuesta 18: Hay 21 términos en la serie. Si dijiste 20, acabas de experimentar un error en el poste de la cerca.

Ejercicio 19: de Programación Asertiva en la página 125

Una rápida comprobación de la realidad. ¿Cuál de estas cosas "imposibles" puede suceder?

1. Un mes con menos de 28 días
2. stat(".",&sb) == -1 (es decir, no se puede acceder al directorio actual)
3. En C++: a = 2; b = 3; **si** (a + b != 5) **salida(1);**
4. Un triángulo con una suma de ángulos interiores $\neq 180^\circ$
5. Un minuto que no tiene 60 segundos
6. En Java: (a + 1) <= a

Respuesta 19:

1. Septiembre de 1752 tuvo sólo 19 días. Esto se hizo para sincronizar calendarios como parte de la Reforma Gregoriana.
2. Es posible que el directorio haya sido eliminado por otro proceso, es posible que no tenga permiso para leerlo, etc.
3. A escondidas, no especificamos los tipos de a y b. Es posible que la sobrecarga de operadores haya definido +, = o != para tener un comportamiento inesperado. Además, a y b pueden ser alias para la misma variable, por lo que la segunda asignación sobrescribirá el valor almacenado en la primera.
4. En geometría no euclídea, la suma de los ángulos de un triángulo no sumará 180° . Piensa en un triángulo mapeado en la superficie de una esfera.

5. Los minutos bisiestos pueden tener 61 o 62 segundos.
6. El desbordamiento puede dejar el resultado de `un + 1` negativo (esto también puede suceder en C y C++).

Ejercicio 20: de Programación Asertiva en la página 125

Desarrolle una clase de comprobación de aserciones simple para Java.

Respuesta 20: Elegimos implementar una clase muy simple con un solo método estático, `TEST`, que imprime un mensaje y un seguimiento de pila si el parámetro de condición pasado es falso.

```

Paquete com.Pragueprag.Util;
importación java.lang.System;      para exit()
importación java.lang.Hilo;        para dumpStack()
public class Assert {
    /** Escriba un mensaje, imprima un seguimiento de pila y salga si
     * Nuestro parámetro es falso.
     */
    public static void TEST(condición booleana) {
        if (!condición) {
            System.out.println("===== Error de aserción =====");
            Hilo.dumpStack();
            System.exit(1);
        }
    }
    Testbed. Si nuestro argumento es 'bien', intente una afirmación que
    tiene éxito, si 'falla' intente uno que falle
    public static final void main(String args[]) {
        if (args[0].compareTo("okay") == 0) {
            TEST(1 == 1);
        }
        else if (args[0].compareTo("fail") == 0) {
            TEST(1 == 2);
        }
        else {
            lanzar new RuntimeException("Argumento incorrecto");
        }
    }
}

```

Ejercicio 21: de Cuándo usar excepciones en la página 128

Al diseñar una nueva clase de contenedor, se identifican las siguientes condiciones de error posibles:

1. No hay memoria disponible para un nuevo elemento en la rutina `add`
2. La entrada solicitada no se encuentra en la rutina de recuperación
3. Puntero nulo pasado a la rutina `add`

¿Cómo se debe manejar cada uno? ¿Se debe generar un error, se debe plantear una excepción o se debe ignorar la condición?

Respuesta 21: Quedarse sin memoria es una condición excepcional, por lo que creemos que el caso (1) debería plantear una excepción.

El hecho de no encontrar una entrada es probablemente una ocurrencia bastante normal. La aplicación que llama a nuestra clase de colección puede escribir código que compruebe si una entrada está presente antes de agregar un posible duplicado. Creemos que el caso (2) debería devolver un error.

El caso (3) es más problemático: si el valor `nulo` es significativo para la aplicación, entonces puede agregarse justificadamente al contenedor. Sin embargo, si no tiene sentido almacenar valores nulos, probablemente se debería producir una excepción.

Ejercicio 22: de Cómo equilibrar los recursos en la página 136

Algunos desarrolladores de C y C++ se esfuerzan por establecer un puntero en `NULL` después de desasignar la memoria a la que hace referencia. ¿Por qué es una buena idea?

Respuesta 22: En la mayoría de las implementaciones de C y C++, no hay forma de comprobar que un puntero realmente apunta a una memoria válida. Un error común es deslocalizar un bloque de memoria y hacer referencia a esa memoria más adelante en el programa. Para entonces, es muy posible que la memoria señalada haya sido reasignada a algún otro propósito. Al establecer el puntero en `NULL`, los programadores esperan evitar estas referencias maliciosas: en la mayoría de los casos, la desreferenciación de un puntero `NULL` generará un error en tiempo de ejecución.

Ejercicio 23: de Cómo equilibrar los recursos en la página 136

Algunos desarrolladores de Java se esfuerzan por establecer una variable de objeto en `NULL` después de haber terminado de usar el objeto. ¿Por qué es una buena idea?

Respuesta 23: Al establecer la referencia en `NULL`, se reduce en uno el número de punteros al objeto al que se hace referencia. Una vez que este recuento llega a cero, el objeto es apto para la recolección de elementos no utilizados. Establecer las referencias en `NULL` puede ser significativo para los programas de larga duración, donde los programadores deben asegurarse de que la utilización de la memoria no aumente con el tiempo.

Ejercicio 24: de El desacoplamiento y la ley de Deméter en la página 143
 Discutimos el concepto de desacoplamiento físico en el recuadro de la página 142. ¿Cuál de los siguientes archivos de encabezado de C++ está más estrechamente acoplado al resto del sistema?

persona1.h:

```
#include "Fecha.h"
class Persona1 {
Privado:
  Fecha myBirthdate;
Público:
  Persona1(Fecha y Fecha de Nacimiento);
  ...
}
```

persona2.h:

```
Fecha de la clase :
class Persona2 {
Privado:
  Fecha *myBirthdate;
Público:
  Persona2(Fecha y Fecha de Nacimiento);
  ...
}
```

Respuesta 24: Se supone que un archivo de encabezado define la interfaz entre la implementación correspondiente y el resto del mundo. El archivo de encabezado en sí mismo no necesita conocer los aspectos internos de la clase Date, simplemente necesita decirle al compilador que el constructor toma un Date como parámetro. Por lo tanto, a menos que el archivo de encabezado use fechas en funciones en línea, el segundo fragmento funcionará bien.

¿Qué hay de malo en el primer fragmento? En un proyecto pequeño, nada, excepto que está haciendo innecesariamente todo lo que usa una clase Person1 también incluye el archivo de encabezado para Date. Una vez que este tipo de uso se vuelve común en un proyecto, pronto descubres que incluir un archivo de encabezado termina incluyendo la mayor parte del resto del sistema, un serio lastre en los tiempos de compilación.

Ejercicio 25: de El desacoplamiento y la ley de Deméter en la página 143
 Para el siguiente ejemplo y para los de los ejercicios 26 y 27, determine si las llamadas a métodos que se muestran están permitidas de acuerdo con la Ley de Deméter. Este primero está en Java.

```
public void showBalance(BankAccount acct) {
  Dinero amt = acct.getBalance();
  printToScreen(amt.printFormat());
}
```

Respuesta 25: La variable acct se pasa como parámetro, por lo que se permite la llamada getBalance. Sin embargo, llamar a amt.printFormat() no lo es. No somos "dueños" de amt y no nos fue pasado. Podríamos eliminar el acoplamiento de showBalance a Money con algo como esto:

```
void showBalance(Cuenta bancaria b) {
  b.printBalance();
}
```

Ejercicio 26: de El desacoplamiento y la ley de Deméter en la página 143
Este ejemplo también está en Java.

```
clase pública Colada {
    licuadora privada myBlender;
    Vector privado myStuff;
    public Colada() {
        myBlender = nuevo Blender();
        myStuff = nuevo Vector();
    }
    private void doSomething() {
        myBlender.addIngredients(myStuff.elements());
    }
}
```

Respuesta 26: Desde Colada crea y posee ambos myBlender y myStuff,
Las llamadas a addIngredients y Elementos están permitidos.

Ejercicio 27: de El desacoplamiento y la ley de Deméter en la página 143
Este ejemplo está en C++.

```
void processTransaction(BankAccount acct, int) {
    Persona *quién;
    Dinero amt;
    amt.setValue(123.45);
    acct.setBalance(amt);
    quién = acct.getOwner();
    markWorkflow(quién->nombre(), SET_BALANCE);
}
```

Respuesta 27: En este caso, processTransaction es propietario de amt, es decir, se crea en la pila. acct se pasa, por lo que se permiten tanto setValue como setBalance. Pero processTransaction no es dueño de quién, por lo que la llamada who->name() está en violación. La Ley de Deméter sugiere reemplazar esta línea con

```
markWorkflow(acct.name(), SET_BALANCE);
```

El código en processTransaction no debería tener que saber qué subobjeto dentro de una BankAccount tiene el nombre, este conocimiento estructural no debería mostrarse a través del contrato de BankAccount. En su lugar, le pedimos a la cuenta bancaria el nombre de la cuenta. Sabe dónde guarda el nombre (ya sea en una persona, en una empresa o en un objeto cliente polimórfico).

Ejercicio 28: de Metaprogramación en la página 149

¿Cuál de las siguientes cosas se representaría mejor como código dentro de un programa y cuál externamente como metadatos?

1. Asignaciones de puertos de comunicación
2. Soporte de un editor para resaltar la sintaxis de varios idiomas
3. Soporte de un editor para diferentes dispositivos gráficos
4. Una máquina de estado para un analizador o escáner
5. Valores de muestra y resultados para su uso en pruebas unitarias

Respuesta 28: No hay respuestas definitivas aquí: las preguntas tenían la intención principal de darle que pensar. Sin embargo, esto es lo que pensamos:

1. Asignaciones de puertos de comunicación. Claramente, esta información debe almacenarse como metadatos. Pero, ¿con qué nivel de detalle? Algunos programas de comunicación de Windows permiten seleccionar sólo la velocidad de transmisión y el puerto (por ejemplo, de COM1 a COM4). Pero tal vez necesite especificar el tamaño de la palabra, la paridad, los bits de parada y también la configuración dúplex . Trate de permitir el nivel más fino de detalle cuando sea práctico.
2. Soporte de un editor para resaltar la sintaxis de varios idiomas. Esto debe implementarse como metadatos. No querrá tener que piratear el código solo porque la última versión de Java introdujo una nueva palabra clave.
3. Soporte de un editor para diferentes dispositivos gráficos. Esto probablemente sería difícil de implementar estrictamente como metadatos. No querrá sobrecargar su aplicación con varios controladores de dispositivo solo para seleccionar uno en tiempo de ejecución. Sin embargo, puede usar metadatos para especificar el nombre del controlador y cargar dinámicamente el código. Este es otro buen argumento para mantener los metadatos en un formato legible por humanos; Si usa el programa para configurar un controlador de video defectuoso, es posible que no pueda usar el programa para volver a configurarlo.
4. Máquina de estado para un analizador o escáner. Esto depende de lo que esté analizando o escaneando. Si está analizando algunos datos que están rígidamente definidos por un organismo de normalización y es poco probable que cambien sin una ley del Congreso, entonces la codificación rígida está bien. Pero si se enfrenta a una situación más volátil, puede ser beneficioso definir las tablas de estado externamente.
5. Valores de muestra y resultados para su uso en pruebas unitarias. La mayoría de las aplicaciones definen estos valores en línea en el arnés de pruebas, pero puede obtener una mayor flexibilidad moviendo los datos de prueba (y la definición de los resultados aceptables) fuera del propio código.

Ejercicio 29: de It's Just a View en la página 164

Supongamos que tienes un sistema de reservas de una aerolínea que incluye el concepto de vuelo:

```
public interface Flight {
    Devuelve false si el vuelo está lleno.
    booleano público addPassenger(Pasajero p);
    public void addToWaitList(Pasajero p); public
    int getFlightCapacity();
    public int getNumPassengers();
}
```

Si añades a un pasajero a la lista de espera, se le pondrá en el vuelo automáticamente cuando haya una vacante disponible.

Hay un trabajo masivo de informes que consiste en buscar vuelos sobrevendidos o llenos para sugerir cuándo se podrían programar vuelos adicionales. Funciona bien, pero tarda horas en ejecutarse.

Nos gustaría tener un poco más de flexibilidad en el procesamiento de los pasajeros de la lista de espera, y tenemos que hacer algo con respecto a ese gran informe, ya que lleva demasiado tiempo publicarse. Utilice las ideas de esta sección para rediseñar esta interfaz.

Respuesta 29: Tomaremos Flight y agregaremos algunos métodos adicionales para mantener dos listas de oyentes: una para la notificación de la lista de espera y la otra para la notificación de vuelo completo.

```
interfaz pública Pasajero {
    public void waitListAvailable();
}

public interface Flight {
    ---
    public void addWaitListListener(Pasajero p);
    public void removeWaitListListener(Pasajero p);
    public void addFullListener(FullListener b);
    public void removeFullListener(FullListener b);
    ---
}

public interface BigReport extends FullListener {
    public void FlightFullAlert(Vuelo f);
}
```

Si intentamos agregar un pasajero y fallamos porque el vuelo está lleno, podemos, opcionalmente, poner al pasajero en la lista de espera. Cuando se abra un lugar, se llamará a `waitList- Available`. Este método puede optar por agregar al Pasajero automáticamente, o hacer que un representante de servicio llame al cliente para preguntar si todavía está interesado, o lo que sea. Ahora tenemos la flexibilidad de realizar diferentes comportamientos por cliente.

A continuación, queremos evitar que el BigReport rastree montones de registros en busca de vuelos completos. Al tener a BigReport registrado

como oyente en Vuelos,

cada vuelo individual puede informar cuando está lleno, o casi lleno, si queremos. Ahora los usuarios pueden obtener informes en vivo y actualizados de BigReport al instante, sin esperar horas para que se ejecuten como lo hacía anteriormente.

Ejercicio 30: de las pizarras de la página 170

Para cada una de las siguientes aplicaciones, ¿sería apropiado o no un sistema de pizarra? ¿Por qué?

1. Tratamiento de imágenes. Le gustaría que una serie de procesos paralelos tomaran fragmentos de una imagen, los procesaran y volvieran a colocar el fragmento completo.
2. Calendario grupal. Hay personas dispersas por todo el mundo, en diferentes zonas horarias y que hablan diferentes idiomas, tratando de programar una reunión.
3. Herramienta de monitoreo de red. El sistema recopila estadísticas de rendimiento y recopila informes de problemas. Le gustaría implementar algunos agentes para usar esta información para buscar problemas en el sistema.

Respuesta 30:

1. Tratamiento de imágenes. Para una programación sencilla de una carga de trabajo entre los procesos paralelos, una cola de trabajo compartida puede ser más que adecuada. Es posible que desee considerar un sistema de pizarra si hay comentarios involucrados, es decir, si los resultados de un fragmento procesado afectan a otros fragmentos, como en aplicaciones de visión artificial o transformaciones complejas de deformación de imágenes 3D.
2. Calendario grupal. Esta podría ser una buena opción. Puede publicar las reuniones programadas y la disponibilidad en la pizarra. Hay entidades que funcionan de forma autónoma, la retroalimentación de las decisiones es importante y los participantes pueden ir y venir.

Es posible que desee considerar la posibilidad de dividir este tipo de sistema de pizarra dependiendo de quién esté buscando: el personal subalterno puede preocuparse solo por la oficina inmediata, los recursos humanos pueden querer solo oficinas de habla inglesa en todo el mundo y el CEO puede querer toda la enchilada.

También hay cierta flexibilidad en los formatos de datos: somos libres de ignorar formatos o idiomas que no entendemos. Tenemos que entender los diferentes formatos sólo para aquellas oficinas que tienen reuniones entre sí, y no necesitamos exponer a todos los participantes a un cierre transitivo completo de todos los formatos posibles. Esto reduce el acoplamiento hasta donde es necesario, y no nos constriñe artificialmente.

3. Herramienta de monitoreo de red. Esto es muy similar al programa de

solicitud de hipotecas/préstamos descrito en la página 168. Tiene informes de problemas enviados por los usuarios y estadísticas reportadas automáticamente, todas publicadas en la pizarra. Un humano o un agente de software puede analizar la pizarra para diagnosticar

Fallos de red: Dos errores en una línea pueden ser solo rayos cósmicos, pero 20.000 errores y tienes un problema de hardware. Al igual que los detectives resuelven el misterio del asesinato, puede tener varias entidades analizando y aportando ideas para resolver los problemas de la red.

Ejercicio 31: de Programación por Coincidencia en la página 176

¿Puede identificar algunas coincidencias en el siguiente fragmento de código C? Supongamos que este código está enterrado en lo más profundo de una rutina de biblioteca.

```
fprintf(stderr, "¿Error, continuar?");
gets(buf);
```

Respuesta 31: Hay varios problemas potenciales con este código. En primer lugar, asume un entorno tty. Eso puede estar bien si la suposición es cierta, pero ¿qué pasa si se llama a este código desde un entorno GUI donde ni stderr ni stdin están abiertos?

En segundo lugar, está el problema gets, que escribirá tantos caracteres como reciba en el búfer pasado. Los usuarios malintencionados han utilizado este fallo para crear agujeros de *seguridad de desbordamiento de búfer* en muchos sistemas diferentes. Nunca uses gets().

En tercer lugar, el código asume que el usuario entiende inglés.

Por último, nadie en su sano juicio enterraría una interacción del usuario como esta en una rutina de biblioteca.

Ejercicio 32: de Programación por Coincidencia en la página 176

Este fragmento de código C puede funcionar parte del tiempo, en algunas máquinas. Por otra parte, puede que no. ¿Qué pasa?

```
/* Truncar la cadena hasta sus últimos caracteres maxlen */
void string_tail(char *string, int maxlen) {
    int len = strlen(cadena);
    if (len > maxlen) {
        strcpy(cadena, cadena + (len - maxlen));
    }
}
```

Respuesta 32: POSIX strcpy No se garantiza que funcione para cadenas superpuestas. Podría suceder que funcione en algunas arquitecturas, pero solo por coincidencia.

Ejercicio 33: de Programación por Coincidencia en la página 177

Este código proviene de un conjunto de rastreo de Java de propósito general. La función escribe una cadena en un archivo de registro. Pasa su prueba unitaria, pero falla cuando uno de los desarrolladores web lo utiliza. ¿En qué coincidencia se basa?

```
public static void debug(String s) throws IOException {
    FileWriter fw = new FileWriter("debug.log", true); fw.write(s);
    fw.flush();
    fw.close();
}
```

Respuesta 33: No funcionará en un contexto de applet con restricciones de seguridad contra la escritura en el disco local. Una vez más, cuando tenga la opción de ejecutarse en contextos de GUI o no, es posible que desee verificar dinámicamente para ver cómo es el entorno actual. En este caso, es posible que desee colocar un archivo de registro en algún lugar que no sea el disco local si no es accesible.

Ejercicio 34: de la velocidad del algoritmo en la página 183

Hemos codificado un conjunto de rutinas de clasificación simples, que se pueden descargar de nuestro sitio web (www.pragmaticprogrammer.com). Ejecútelo en varias máquinas disponibles para usted. ¿Sus cifras siguen las curvas esperadas? ¿Qué puede deducir sobre las velocidades relativas de sus máquinas? ¿Cuáles son los efectos de varias configuraciones de optimización del compilador? ¿Es el orden de base realmente lineal?

Respuesta 34: Claramente, no podemos dar ninguna respuesta absoluta a este ejercicio. Sin embargo, podemos darle un par de consejos.

Si descubres que los resultados no siguen una curva suave, es posible que quieras comprobar si alguna otra actividad está utilizando parte de la energía del procesador. Es probable que no obtenga buenas cifras en un sistema multiusuario, e incluso si es el único usuario, es posible que los procesos en segundo plano eliminen periódicamente ciclos de sus programas. También es posible que desee verificar la memoria: si la aplicación comienza a usar espacio de intercambio, el rendimiento caerá en picado.

Es interesante experimentar con diferentes compiladores y diferentes configuraciones de optimización. Descubrimos algunos en los que era posible acelerar bastante sorprendentemente al permitir una optimización agresiva. También descubrimos que en las arquitecturas RISC más amplias, los compiladores del fabricante a menudo superaban a los GCC más portátiles. Presumiblemente, el fabricante está al tanto de los secretos de la generación eficiente de código en estas máquinas.

Ejercicio 35: de la velocidad del algoritmo en la página 183

La siguiente rutina imprime el contenido de un árbol binario. Suponiendo que el árbol esté equilibrado, ¿cuánto espacio de pila utilizará la rutina al imprimir un árbol de 1.000.000 de elementos? (Supongamos que las llamadas a subrutinas no imponen una sobrecarga de pila significativa).

```
void printTree(const Nodo *nodo) {
    búfer de caracteres [1000];
    if (nodo) {
        printTree(nodo->izquierda);
        getNodeAsString(nodo, búfer);
        puts(búfer);
        printTree(nodo->derecha);
    }
}
```

Respuesta 35: La rutina `printTree` utiliza unos 1.000 bytes de espacio de pila para la variable de búfer. Se llama a sí mismo de forma recursiva para descender a través del árbol, y cada llamada anidada añade otros 1.000 bytes a la pila. También se llama a sí mismo cuando llega a los nodos hoja, pero se cierra inmediatamente cuando descubre que el puntero pasado es `NULL`. Si la profundidad del árbol es D , el requisito máximo de pila es, por lo tanto, aproximadamente $1000 \times (D + 1)$.

Un árbol binario equilibrado contiene el doble de elementos en cada nivel. Un árbol de profundidad D contiene $1+2+4+8+\dots+2^{(D-1)}$, o $2^D - 1$, elementos.

Nuestro millón de elementos

Por lo tanto, el árbol necesitará $\lceil \lg(1,000,001) \rceil$, o 20 niveles.

Por lo tanto, esperaríamos que nuestra rutina usara aproximadamente 21,000 bytes de pila.

Ejercicio 36: de la velocidad del algoritmo en la página 183

¿Puede ver alguna manera de reducir los requisitos de pila de la rutina en el ejercicio 35 (aparte de reducir el tamaño del búfer)?

Respuesta 36: Me vienen a la mente un par de optimizaciones. En primer lugar, la rutina `printTree` se llama a sí misma en los nodos hoja, solo para salir porque no hay elementos secundarios. Esta llamada aumenta la profundidad máxima de la pila en unos 1.000 bytes. También podemos eliminar la recursividad de cola (la segunda llamada recursiva), aunque esto no afectará el uso de la pila en el peor de los casos.

```
while (nodo) {
    if (nodo->izquierda) printTree(nodo->izquierda);
    getNodeAsString(nodo, búfer);
    puts(búfer);
    nodo = nodo->derecha;
}
```

Sin embargo, la mayor ganancia proviene de la asignación de un solo búfer, compartido por todas las invocaciones de `printTree`. Pase este búfer como parámetro a las llamadas recurrentes y solo se asignarán 1.000 bytes, independientemente de la profundidad de la recursividad.

```
void printTreePrivate(const Node *node, char *buffer) {
    if (nodo) {
        printTreePrivate(nodo->izquierda, búfer);
        getNodeAsString(nodo, búfer);
        puts(búfer);
        printTreePrivate(nodo->derecha, búfer);
    }
}
void newPrintTree(const Node *node) {
    búfer char [1000];
    printTreePrivate(nodo, búfer);
}
```

Ejercicio 37: de la velocidad del algoritmo en la página 183

En la página 180, afirmamos que un corte binario es $O(\lg(n))$. ¿Puedes probar esto?

Respuesta 37: Hay un par de maneras de llegar allí. Una es darle la vuelta al problema. Si la matriz tiene un solo elemento, no iteramos alrededor del bucle. Cada iteración adicional duplica el tamaño de la matriz en la que podemos buscar. Por lo tanto, la fórmula general para el tamaño de la matriz es $n = 2^m$, donde m es el número de iteraciones. Si llevas los troncos a la base 2 de cada lado, obtienes $\lg(n) = \lg(2^m)$, que por la definición de troncos se convierte en $\lg(n) = m$.

Ejercicio 38: de Refactorización en la página 188

Obviamente, el siguiente código se ha actualizado varias veces a lo largo de los años, pero los cambios no han mejorado su estructura. Refactoríalo.

```
if (estado == TEXAS) {
    tasa = TX_RATE;
    amt = base * TX_RATE;
    calc = 2 * base (amt) + extra (amt) * 1.05;
}
else if ((estado == OHIO) || (estado == MAINE)) {
    tasa = (estado == OHIO) ? OH_RATE : ME_RATE; amt
    = base * tasa;
    calc = 2 * base (amt) + extra (amt) * 1.05;
    si (estado == OHIO)
        puntos = 2;
}
else {
    tasa = 1; amt
    = base;
    calc = 2 * base (amt) + extra (amt) * 1.05;
}
```

Respuesta 38: Podríamos sugerir aquí una reestructuración bastante suave: asegúrese de que cada prueba se realice una sola vez y haga que todos los cálculos sean comunes. Si la expresión $2 * \text{base}(\dots) * 1.05$ aparece en otros lugares del programa, probablemente deberíamos convertirlo en una función. No nos hemos molestado aquí.

Hemos agregado una matriz `rate_lookup`, inicializada para que las entradas que no sean Texas, Ohio y Maine tengan un valor de 1. Este enfoque facilita la adición de valores para otros estados en el futuro. Dependiendo del patrón de uso esperado, es posible que también queramos hacer que el campo de `puntos` sea una búsqueda de matriz.

```
tasa = rate_lookup[estado];
amt = base * tasa;
calc = 2 * base (amt) + extra (amt) * 1.05;
si (estado == OHIO)
    puntos = 2;
```

Ejercicio 39: de Refactorización en la página 188

La siguiente clase Java debe admitir algunas formas más. Refactorice la clase para prepararla para las adiciones.

```
public class Shape {
    public static final int CUADRADO = 1;
    public static final int CIRCUNFERENCIA = 2;
    public static final int RIGHT_TRIANGLE = 3;
    private int shapeType;
    tamaño doble privado;
    public Shape(int shapeType, double size) {
        this.shapeType = shapeType;
        éste.tamaño = tamaño;
    }
    ... otros métodos ...
    área doble pública () {
        switch (shapeType) {
            caso CUADRADO:devolución tamaño * tamaño;
            caso CIRCUNFERENCIA: devolución Matemáticas.PI*tamaño*tamaño/4.0;
            Caso RIGHT_TRIANGLE: Tamaño de devolución * tamaño / 2.0;
        }
        devuelve 0;
    }
}
```

Respuesta 39: Cuando ves a alguien usando tipos enumerados (o su equivalente en Java) para distinguir entre variantes de un tipo, a menudo puedes mejorar el código mediante subclases:

```

public class Shape {
    tamaño doble privado ;
    public Shape(double size) {
        this.size = tamaño;
    }
    public double getSize() { tamaño de retorno ; }
}
public class Square extends Shape {
    Plaza pública (tamaño doble) {
        super(tamaño);
    }
    public double area() { double
        size = getSize(); tamaño de
        devolución * tamaño;
    }
}
public class Circle extends Shape {
    public Circle(tamaño doble ) {
        super(tamaño);
    }
    área doble pública () {
        tamaño doble = getSize();
        return Math.PI*size*size/4.0;
    }
}
etcetera...

```

Ejercicio 40: de Refactorización en la página 189

Este código Java es parte de un marco que se utilizará en todo su proyecto. Refactorícelo para que sea más general y más fácil de extender en el futuro.

```

public class Window {
    public Window(int width, int height) { ... } public
    void setSize(int width, int height) { ... }
    superposiciones booleanas públicas (Ventana w)
    { ... }
    public int getArea() { ... }
}

```

Respuesta 40: Este caso es interesante. A primera vista, parece razonable que una ventana tenga una anchura y una altura. Sin embargo, considere el futuro. Imaginemos que queremos admitir ventanas con formas arbitrarias (lo cual será difícil si la clase Window sabe todo sobre los rectángulos y sus propiedades).

Sugerimos abstraer la forma de la ventana de la propia clase Window.

```

clase abstracta pública Shape {
    ...
    superposiciones booleanas abstractas públicas (Forma s);
    public abstract int getArea();
}

public class Window {
    forma privada;
    public Window(Forma, forma) {
        this.shape = forma;
        ...
    }
    public void setShape(Forma de forma) {
        this.shape = forma;
        ...
    }
    public boolean overlaps(Window w) {
        return shape.overlaps(w.shape);
    }
    public int getArea() {
        return shape.getArea();
    }
}

```

Tenga en cuenta que en este enfoque hemos utilizado la delegación en lugar de la subclase: una ventana no es una forma "más o menos", una ventana "tiene una". Utiliza una forma para hacer su trabajo. A menudo, la delegación le resultará útil al refactorizar.

También podríamos haber extendido este ejemplo introduciendo una interfaz Java que especificara los métodos que una clase debe admitir para admitir las funciones shape. Esta es una buena idea. Esto significa que cuando amplíes el concepto de una forma, el compilador te advertirá sobre las clases a las que te hayas visto afectado. Se recomienda usar interfaces de esta manera cuando se delegan todas las funciones de alguna otra clase.

Ejercicio 41: de Código que es fácil de probar en la página 197

Diseñe una plantilla de prueba para la interfaz de la batidora descrita en la respuesta al Ejercicio 17 en la página 289. Escriba un script de shell que realice una prueba de regresión para el blender. Debe probar la funcionalidad básica, las condiciones de error y límite, y cualquier obligación contractual. ¿Qué restricciones se imponen para cambiar la velocidad? ¿Están siendo honrados?

Respuesta 41: Primero, agregaremos un main para que actúe como controlador de prueba unitaria. Aceptará un lenguaje muy pequeño y simple como argumento: "E" para vaciar la licuadora, "F" para llenarla , dígitos del 0 al 9 para establecer la velocidad, y así sucesivamente.

```

public static void main(String args[]) {
    Crea la licuadora para probar
    dbc_ex licuadora = nuevo dbc_ex();
    Y pruébelo de acuerdo con la cadena en la entrada estándar
    try {
        int a;
        char c;
        while ((a = System.in.read()) != -1) { c
            = (char)a;
            if (Character.isWhitespace(c)) {
                continuo;
            }
            if (Character.isDigit(c)) {
                blender.setSpeed(Character.digit(c, 10));
            }
            else {
                interruptor (c) {
                    caso 'F': blender.fill();
                    descanso;
                    caso 'E': blender.empty();
                    descanso;
                    case 's': System.out.println("VELOCIDAD: " +
                        blender.getSpeed());
                    descanso;
                    caso 'f': System.out.println("FULL " +
                        blender.isFull());
                    descanso;
                    default: throw new RuntimeException(
                        "Directiva de prueba desconocida");
                }
            }
        }
    }
    catch (java.io.IOException e) {
        System.err.println("Error en la plantilla de prueba : " +
            e.getMessage());
    }
    System.err.println("Mezcla completada\n");
    sistema.exit(0);
}

```

A continuación, viene el script de shell para controlar las pruebas.

```

#!/papel/sh
CMD="java dbcdbc_ex"
Recuento de fallos=0
expect_okay() {
    if echo "$*" | $CMD >/dev/null 2>&1
    entonces
    :
    más
        echo "iFALLÓ! $*"
        failcount='expr $failcount + 1'
    fi
}
expect_fail() {
    if echo "$*" | $CMD >/dev/null 2>&1
    entonces
        echo "iFALLÓ! (Debería haber falló): $*"
        failcount='expr $failcount + 1'
    fi
}
informe() {
    if [ $failcount -gt; 0 ]
    entonces
        echo -e '^n\n*** PRUEBAS $failcount FALLADAS ^n'
        exit 1 # En caso de que seamos parte de algo más grande
    más
        exit 0 # En caso de que seamos parte de algo más grande
    fi
}
#
# Iniciar las
pruebas #
expect_okay F123456789876543210E # Debería
ejecutarse a través de expect_fail F5      # Falla velocidad
demasiado alta expect_fail 1    # Falla vacío
expect_fail F10E1 # Falla vacío expect_fail
F1238 # Falla Salta expect_okay FE # Nunca
gire en expect_fail F1E      # Vaciado
mientras corre expect_okay F10E # Debería
ser De acuerdo
informe          # Informe Resultados

```

Las pruebas comprueban si se detectan cambios de velocidad ilegales, si intenta vaciar la batidora mientras está en funcionamiento, etc. Ponemos esto en el makefile para que podamos compilar y ejecutar la prueba de regresión simplemente escribiendo

```
% de hacer
% de hacer la prueba
```

Tenga en cuenta que tenemos la salida de prueba con 0 o 1 , por lo que también podemos usar esto como parte de una prueba más grande.

No había nada en los requisitos que hablara de manejar este componente a través de un script, o incluso usar un lenguaje. Los usuarios finales nunca lo verán. Pero tenemos una herramienta poderosa que podemos usar para probar nuestro código, de manera rápida y exhaustiva.

Ejercicio 42: de El pozo de los requisitos en la página 211

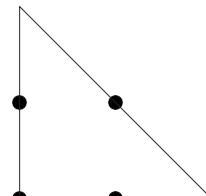
¿Cuáles de los siguientes son probablemente requisitos genuinos? Repite las que no lo son para hacerlas más útiles (si es posible).

1. El tiempo de respuesta debe ser inferior a 500 ms.
2. Los cuadros de diálogo tendrán un fondo gris.
3. La aplicación se organizará como una serie de procesos front-end y un servidor back-end.
4. Si un usuario introduce caracteres no numéricos en un campo numérico, el sistema emitirá un pitido y no los aceptará.
5. El código de la aplicación y los datos deben caber dentro de los 256 kB.

Respuesta 42:

1. Esta afirmación suena como un requisito real: puede haber restricciones impuestas a la aplicación por su entorno.
2. Aunque este puede ser un estándar corporativo, no es un requisito. Sería mejor decirlo como "El fondo del cuadro de diálogo debe ser configurable por el usuario final. Tal como se envía, el color será gris". Aún mejor sería la afirmación más amplia "Todos los elementos visuales de la aplicación (colores, fuentes e idiomas) deben ser configurables por el usuario final".
3. Esta afirmación no es un requisito, es arquitectura. Ante algo así, hay que profundizar para saber qué está pensando el usuario.
4. El requisito subyacente es probablemente algo más cercano a "El sistema evitará que el usuario realice entradas no válidas en los campos y advertirá al usuario cuando se realicen estas entradas".
5. Esta afirmación es probablemente un requisito difícil.

Una solución al rompecabezas de los Cuatro Postes planteado en la página 213.



Esta página se ha dejado en blanco intencionadamente

Índice

Un

Función de acceso, 31
ACM, véase Association for Computing Machinery (Asociación de Maquinaria de Computación)
Generador de código activo, 104 Diagrama de actividades, 150
Advanced C++ Programming Styles and Modisms, 265
Advanced Programming in the Unix Environment, 264
Gestión de configuración basada en transacciones de Aegis, 246, 271
Agente, 76, 117, 297
Algoritmo
 corte binario, 180
 Eligiendo, 182
 combinatoria, 180
 Divide y vencerás, 180
 estimando, 177, 178
 lineal, 177
 $O()$ Notación, 178, 181
 Clasificación rápida, 180
 Tiempo de ejecución, 181
 sublineal, 177
Asignaciones, anidamiento, 131
Patrones de Analysis, 264
Anonimato, 258
AOP, consulte Arquitectura de programación orientada a aspectos
 despliegue, 156
 flexibilidad, 46
 Prototipado, 55
 desacoplamiento temporal, 152
El arte de la programación informática, 183 Inteligencia artificial, 26 Programación orientada a aspectos (AOP), 39, 273
Aserción, 113, 122, 175
 efectos secundarios, 124

apagado, 123
Asociación de Maquinaria Informática (ACM), 262
 Comunidades de la ACM, 263
 SIGPLAN, 263
Supuestos, pruebas, 175
Comando "arroba", 231
Público, 21
 necesidades, 19
`auto_ptr`, 134
Automatización, 230
 Procedimientos de aprobación, 235
 construcción, 88, 233
 compilando, 232
 cron, 231
 documentación, 251
 guiones, 234
 equipo, 229
 Pruebas, 29, 238
 Generación de sitios web, 235
`awk`, 99

B

Formulario Backus-Naur (BNF), 59n
Clase base, 112
Shell Bash, 80, 82N
Frijol, véase Enterprise Java Beans (EJB) Beck, Kent, 194, 258
Proyecto Beowulf, 268
Notación "grande O ", 177 "Imagen grande", 8
Chuleta binaria, 97, 180
Formato binario, 73
 problemas de análisis, 75
bisonte, 59, 269
BIST, consulte el sistema de pizarra de autoprueba
incorporado, 165
 partición, 168
Flujo de trabajo, 169

- Ejemplo de licuadora
 contrato para, 119,
 289 plantilla de prueba
 de regresión, 305 flujo
 de trabajo, 151
- BNF, véase Forma Backus-Naur (BNF)
- Rana hervida, 8, 175, 225
- Condición de contorno, 173, 243
- Cerebro, Marshall, 265
- Marca, 226
- Brant, Juan, 268
- "Teoría de la ventana rota", 5
 vs. sopa de piedra, 9
- Brooks, Fred, 264
- Navegador, clase, 187
- Navegador, refactorización, 187, 268
- Bicho, 90
 Contrato fallido como, 111
 véase also Depuración; Error
- Construir
 Automatización, 88, 233
 dependencias, 233
 final, 234
 Noches, 231
 Refactorización, 187
- Autoprueba incorporada (BIST),
189 Lógica de negocios, 146
- Política empresarial, 203
-
- C**
- Lenguaje C
 aserciones, 122
 DBC, 114
 duplicación, 29
 Manejo de errores, 121
 Mensajes de error, 115
 Macros, 121
 Interfaz Object Pascal,
lenguaje C++ 101, 46
 aserciones, 122
 auto_ptr, 134
 libros, 265
 DBC, 114
 desacoplamiento, 142
 DOC++, 251, 269
 duplicación, 29
 Mensajes de error, 115
 Excepciones, 132
 Pruebas unitarias, 193
Almacenamiento en caché, 31
- Llamada, rutina, 115, 173
- Hojas de estilo en cascada (CSS),
253 Cat
 culpabilización, 3
 pastoreo, 224
 Schrödinger's, 47
- Catalizar el cambio,
8 Catedrales, xx
- Enlaces Cetus, 265
- Cambio, catalizador, 8
- Christiansen, Tom,
Clase 81
 aserciones, 113
 base, 112
 acoplamiento, 139, 142
 relaciones de acoplamiento, 242
 Recurso de encapsulación, 132
 invariante, 110, 113
 número de estados, 245
 asignación de recursos,
 132
 Subclase, 112
 envoltura, 132, 133, 135, 141
- Navegador de clases, 187
- ClearCase, 271
- Cockburn, Alistair, xxiii, 205, 264, 272
- Generador de códigos, 28, 102
 activo, 104
 Archivos de fabricación, 232
 analizadores, 105
 Pasivo, 103
- Perfilador de código, 182
- Revisiones de códigos, 33,
236 Codificación
 velocidad del algoritmo, 177
 Comentarios, 29, 249
 acoplado, 130
 Análisis de cobertura, 245
 Esquema de base de datos, 104
 defensivo, 107
 y documentación, 29, 248
 estimación, 68
 Excepciones, 125
 implementación, 173
 iterativo, 69
 "Perezoso", 111
 métricas, 242
 módulos, 138
 representaciones múltiples, 28
 ortogonalidad, 34, 36, 40

- propiedad, 258
- prototipos, 55
- Código de servidor, 196
- "tímido", 40, 138
- Especificaciones, 219
- balas trazadoras, 49–51
- Pruebas unitarias, 190, 192
 - véase el código acoplado also;*
 - Código desacoplado;
 - Metadatos; Sistema de control de código fuente (SCCS)
- Cohesión, 35
- COM, *véase Component Object Model*
- Explosión combinatoria, 140, 167
- Algoritmo combinatorio, 180
- Shell de comandos, 77
 - Bash, 80
 - Cygwin, 80 años
 - vs.* GUI, 78
 - UWIN, 81
 - Ventanas, 80
- Comentario, 29, 249
 - evitar la duplicación, 29
 - DBC, 113
 - parámetros, 250
 - tipos de, 249
 - innecesarios, 250
 - consulte la documentación de*
- also* Common Object Request Broker
 - (CORBA), 29, 39, 46
 - Servicio de Eventos, 160
- Comunicar, 18
 - público, 19, 21
 - duplicación, 32
 - Correo electrónico, 22
 - y métodos formales, 221
 - presentación, 20
 - estilo, 20
 - equipos, 225
 - usuarios, 256
- Escritura, 18
 - Communications de la ACM*, 263
- Comp.object FAQ, 272
- Compilación, 232
 - compiladores, 267
 - DBC, 113
 - advertencias y depuración, 92
- Modelo de objetos componentes (COM), 55 Sistemas basados en componentes, *véase*
 - Sistema modular
- Simultaneidad, 150
 - diseño, 154
 - interfaces, 155
 - y Programación por Coincidencia, 154
- Análisis de requisitos de, 150 flujo de trabajo, 150
- Sistema de versiones simultáneas (CVS), configuración 271
 - Cooperativa, 148
 - Dinámico, 144
 - metadatos, 147
- Gestión de la configuración, 86, 271
- Constantine, Larry L., 35 Gestión de restricciones, 213
- Constructor, 132
 - inicialización, 155
- Contacto, correo electrónico de los autores, xxiii
- Contexto, uso en lugar de globales, 40
- Contrato, 109, 174
 - véase also* Diseño por contrato
- (DBC) Controlador (MVC), 162
- Coplien, Jim, 265
- CORBA, *consule* Agente de solicitud de objeto común
- Código acoplado, 130
 - relaciones de acoplamiento, 242
 - minimización, 138, 158
 - rendimiento, 142
 - acoplamiento temporal, 150
 - véase also* Análisis de cobertura de código
 - desacoplado, 245
- Cox, Brad J., 189n
- Crash, 120
- Pensamiento crítico, 16
- cron, 231
- CSS, *véase* Hojas de estilo en cascada
- CVS, *véase* Sistema de versiones concurrentes
- Cygwin, 80, 270

D

Datos

- Sistema de pizarra, 169
- Almacenamiento en caché, 31
- Diccionario, 144
- Estructuras de datos dinámicas, 135 globales, 40
- Idioma, 60
- normalización, 30

- legible *vs.* comprensible, 75
- prueba, 100, 243
- vistas, 160
- visualización, 93
 - ver also* Metadata
- Depurador de visualización de datos (DDD), 93, 268
 - bases de datos
 - Generador de código activo, esquema 104, 105f, 141, 144
 - mantenimiento de
 - esquema, 100 DBC, *consulte*
- Diseño por contrato DDD, *consulte* Data Display Debugger
- Deadline, 6, 246
- Punto muerto, 131
- Depuración, 90
 - aserciones, 123
 - Búsqueda binaria, 97
 - Ubicación de insectos, 96
 - reproducción de insectos, 93
 - Lista de comprobación, 98
 - advertencias del compilador y, 92 variables corruptas, 95
 - "Heisenbug", 124
 - pata de goma, 95
 - y ramificación del código fuente, 87 error sorpresa, 97
 - y pruebas, 92, 195
 - bomba de tiempo, 192
 - rastreo, 94
 - vista, 164
 - visualización de datos, 93
- Toma de decisiones, 46
- Código desacoplado, 38, 40
 - arquitectura, 152
 - sistema de pizarra, 166 Ley de Deméter, 140
 - metadatos, 145
 - acoplamiento minimizante, 138
 - Pruebas modulares, 244
 - desacoplamiento físico, 142
 - acoplamiento temporal, 150
 - Flujo de trabajo, 150
 - véase also* Código
- acoplado Codificación
- defensiva, 107
- Delegación, 304
- Delfos, 55
 - véase also* Objeto Pascal
- Proyecto Demeter, 274
- Deméter, Ley de, 140
- Dependencia, reducción, *véase* Sistema modular; Ortogonalidad
- Despliegue, 156
- Descriptor de implementación, 148
 - Diseño
 - funciones de descriptor de acceso, 31
 - simultaneidad, 154
 - Contexto, 174
 - despliegue, 156
 - Pruebas de diseño/metodología, 242
 - metadatos, 145
 - ortogonalidad, 34, 37
 - físicos, 142
 - Refactorización, 186
 - Uso de servicios, 154
 - Diseño por contrato (DBC), 109, 155
 - y agentes, 117
 - aserciones, 113
 - invariante de clase, 110
 - Como comentarios, 113
 - contratos dinámicos, 117
 - iContrato, 268
 - apoyo lingüístico, 114
 - Ejemplo de inserción de lista, 110
 - pre y postcondición, 110, 113, 114
 - predicados, 110
 - Pruebas unitarias, 190
- Design Patterns*, 264
 - Observador, 158
 - Soltero, 41 años
 - estrategia, 41
- Destructor, 132
- Detectives, 165
- Árbol de desarrollo, 87
- Desarrollo, iterativo, 69
- Algoritmo de divide y vencerás, 180
- Generador de documentación DOC++ , 251, 269
- DocBook, 254
- Documentación
 - Actualización automática, 251
 - y código, 29, 248
 - Comentarios, 29, 113, 249, 251
 - ejecutable, 251
 - formatos, 253
 - HTML, 101
 - hipertexto, 210
 - interno/externo, 248

- invariante, 117
 lenguajes de marcado, 254
 ortogonalidad, 42
 Esquema, 18
 requisitos, 204
 redactores técnicos, 252
 procesadores de texto, 252, 254
 especificaciones de escritura, 218
véase el comentario de
also ; Documentación
web
- Dodo, 148
 Dominio, problema, 58, 66 No te repitas, *ver SECO*
 principio
 Descarga de código fuente, *consulte*
Código de ejemplo
Dr. Dobbs Journal, 263
 Principio de SECO, 27, 29,
 42
véase also Duplicación
 Pato, caucho, *véase Pato de goma Dumpty, Humpty, xxii, 165* Duplicación, 26
 Generadores de código
 Evitar, 28 y Revisiones de código, 33 Errores de diseño, 30
 documentación y código, 29
 Principio DRY, 27, 29
 Interdesarrollador, 32
 en idiomas, 29
 representaciones múltiples, 28
 equipos, 226
 bajo presión de tiempo,
 32 tipos de, 27
 Configuración dinámica, 144
 Estructura de datos dinámica, 135
Dynamics of Software development, 264

E

- Correo electrónico, 22
 Discurso para comentarios,
 xxiii Editor, 82
 sangría automática, 85
 movimiento del cursor, 84
 características, 83
 Generación de código,
 103 Cuántos aprender,
 82 Plantilla, 84
 tipos de, 266

C++ efectivo, 265
Eiffel, 109, 114, 267
EJB, véase Enterprise Java Beans
Elvis Editor, 267
Editor de Emacs, 84, 266
 Emulador Viper vi , 267
Minilenguaje integrado, 62, 145
Embellecimiento, 11
Encapsulación, objeto, 127, 158
Eno, Brian, 205
Enterprise Java Beans (EJB), 39, 147
Entropía, 4 Error
 Mensajes DBC, 115
 diseño, 30
 Específico del dominio, 59
 Choque temprano, 120
 Mensajes de registro, 196
 ortogonalidad, 41
 Pruebas, 240, 247
 see also Exception Error
handler, 127
Estimación, 64
 precisión, 64
 algoritmos, 177, 178
 iterativo, 69
 modelos, 66
 dominio del problema, 66
 Cronogramas de proyectos, 68
 registros, 68
 pruebas, 182
Eton College, xxi
Evento, 157
Canal de eventos, 160
Código de ejemplo
 Agregar registro, 40
 Reservas de aerolíneas, 164, 296
 Macro de aserción , 122
 auto_ptr ejemplo, 134
 Mal equilibrio de recursos, 129, 130
 descargas, XXIII
 manejo de errores de excepciones, 125 buen equilibrio de recursos, 131
 ejemplo de JavaDoc, 250
 Encadenamiento de métodos, 139
 Clase normalizada, 31
 Abrir archivo de contraseña, 126 Abrir archivo de usuario, 127

- Recursos y Excepciones, 132,
133
Efecto secundario, 124
Manejo de errores de
espagueti, 125 raíz cuadrada,
190
Análisis sintáctico de cadenas con
 StringTokenizer, 156
análisis de cadenas con
 strtok, 155 clase no
normalizada, 30
Ejemplo de código por nombre
AOP, 40
Misc.java, 156
aser, 122
bad_balance.c, 129, 130
balance.cc, 134
Balance.C, 131–133
clase Línea, 30, 31
Excepción, 125
findPeak, 250
interfaz Vuelo, 164, 296
Misc.C, 155
openpasswd.java, 126
openuserfile.java, 127
plotDate, 139
side_effect, 124
spaghetti, 125
sqrt, 190
Excepción, 121
efectos de, 127
y controladores de
errores, 127 archivos
faltantes, 126
Equilibrio de recursos,
132 Cuándo usar, 125
Excusas, 3
Documento ejecutable, 251
esperar, 269
Experto, ver Gurú
Activo que caduca,
12
Lenguaje de estilo extensible (XSL), 253
Extinción, 148
Programación extrema, 238n, 258, 272

F

- Fluencia de características, 10
Comentarios, dirección de correo
electrónico, xxiii Archivo
Excepción, 126
Cabezazo, 29

Registro, 196
Makefile, 232
Fuente, 103
Construcción final, 234
Peces, peligros de, 34
Flexibilidad, 46
Métodos formales, 220, 221
Rompecabezas de cuatro postes,
213 Fowler, Martin, xxiii, 186,
273
Free Software Foundation, véase Proyecto
GNU
Rana, hervida, ver Rana hervida
Función
 de acceso, 31
 Ley de Deméter para ~s, 140
 similar, 41

G

Gamma, Erich, 194
Recolección de basura, 134
Metáfora de la jardinería, 184
Gehrke, Peter, xxiv
Vidrio, Roberto, 221, 236
Variables globales, 40, 130, 154
Glosario, proyecto, 210
Proyecto GNU, 274
 Compilador de C/C++, 267
 Licencia Pública General (GPL), 80 GNU
 Programa de Manipulación de Imágenes
 (GIMP), 274
 PequeñoEiffel, 267
"Software suficientemente bueno", véase
 Software, calidad
Nudo gordiano, 212
Ir a, 127
Sistema GUI
 vs. shell de comandos, 78
 interfaz, 78
 pruebas, 244
Gurú, 17, 198

H

Hachís, seguro, 74
Archivo de cabecera, 29
"Heisenbug", 124, 289
Helicóptero, 34n Hopper,
Grace, 8n, 90
Secuencia de teclas de acceso rápido, 196

Servidor web HTTP, 196
 Factores humanos, 241
 Humpty Dumpty, xxii, 165
 Notación húngara, 249
 Modelo de consumidor hambriento, 153
 Documento de hipertexto, 210

Yo

iContract, 110, 114, 268
 IDE, *consulte* Entorno de desarrollo integrado
 Sociedad de Computación IEEE, 262
Computadora IEEE, 262
IEEE Software, 263
 Lenguaje imperativo, 60
 Implementación
 accidentes, 173
 codificación, 173
 Especificaciones, 219
 Duplicación impuesta, 28
 Duplicación inadvertida, 30
 Sangría, automática, 85
 Independencia, véase Ortogonalidad
 Infraestructura, 37
 Herencia, 111
 aserciones, 113
 Abanico de entrada/salida, 242
 Tenis interior, 215
 Inspección, código, véase Revisiones de códigos Insure++, 136
 Circuito integrado, 189n
 Entorno de Desarrollo Integrado (IDE), 72, 232
 Plataforma de integración, 50
 Pruebas de integración, interfaz 239
 Sistema de pizarra, 168
 C/Objeto Pascal, 101
 simultaneidad, 155
 Controlador de errores, 128
 GUI, 78
 Prototipado, 55
 usuario, 203
 Invariante, 110, 113, 155
 bucle, 116
 semántico, 116, 135
 ISO9660 formato, 233n
 Desarrollo iterativo, 69

J

Jacobson, Ivar, 204
 Jerga, xxii, 210
 Archivo de jerga, 273
 Java, 46, 267
 Generación de código, 232
 DBC, 114
 Beans Java Empresariales, 39, 147
 Mensajes de error, 115
 Excepciones, 121
 iContract, 110, 114, 268
 javaCC, 59, 269
 JavaDoc, 248, 251
 JavaSpaces, 166, 273
 JUNIO, 195
 Programación multihilo, 154
 Acceso a la propiedad, 100
 Archivos de la propiedad, 145
 Equilibrio de recursos, 134
 RMI, 128
 analizador de cadenas, 156
 Vista de árbol, 161
 Pruebas unitarias, 193
 y shells de Windows,
 81 JavaDoc, véase Java

K

K Entorno de escritorio, 273
 Kaizen, xxi, 14
ver also Portafolio de conocimiento
 Kernighan, Brian, 99
 Encuadernación de
 teclas, 82
 Kirk, James T., 26
 Conocimiento
 productores y consumidores, 166
 Portafolio de conocimiento, 12
 Edificio, 13
 pensamiento crítico, 16
 aprendizaje y lectura, 14
 investigación, 15
 Knuth, Donald, 183, 248
 Korn, David, 81
 Kramer, Reto, xxiv
 Kruchten, Phillip, 227n

L

Lakos, Juan, xxiv, 9, 142, 265
 Excusas poco convincentes, 3

Lenguaje, conversiones de programación, 103, 105
 DBC, 114
 dominio, 57
 duplicación en, 29
 aprendizaje, 14
 prototipos, 55
 Guiones, 55, 145
 Especificación, 58, 62
 manipulación de textos, 99
véase also Mini-lenguaje
Large-scale C++ Software Design,
 142, 265
 Sistema LATEX, 103
 Ley de Deméter, 140
 Césped, cuidado
 de, xxi Diseño en
 capas, 37
 Sistema en capas, *véase* Código "perezoso" del sistema modular, 111
Lex and Yacc, 59 años
 Bibliotecario, *véase* Bibliotecario del proyecto Código de biblioteca, 39
 Modelo Linda, 167
 Algoritmos lineales, 177
 Linux, 15, 254, 265
 Principio de sustitución de Liskov, 111 Escucha, 21
 Programación alfabetizada, 248
 Explotación forestal, 39, 196
consulte la tabla
 de búsqueda de
 seguimiento also,
 bucle 104
 anidado, 180
 Sencillo, 180
 Bucle invariante, 116

M

Macro, 78, 86
 aserciones, 122
 documentación, 252
 Manejo de errores, 121
 Mantenimiento, 26
 Lenguas imperativas, 61
 Makefile, 232
 recursivo, 233
 Gestión de expectativas, 256
 Lenguaje de marcado, 254
 Martin, Robert C., 273

Métrica de complejidad ciclomática de McCabe, 242
 Variables miembro, *consulte* Funciones de descriptor de acceso
 Asignación de memoria, 135
 Metadatos, 144, 203
 Lógica de negocios, 146
 configuración, 147
 control de transacciones, 39
 código disociado, 145
 y métodos formales, 221
 en texto plano, 74
 Sistema métrico, 242
 Meyer, Bertrand, 31n, 109, 184, 264
 Meyer, Scott, 265
 Microsoft Visual C++, 198
 Microsoft Windows, 46
 Mini-idioma, 59
 lenguaje de datos, 60
 Embebido, 62
 imperativo, 60
 Análisis sintáctico, 62
 autónomo, 62
 Mesa de mezclas, 205
 Integridad de la fuente
 MKS, modelo 271, 160
 cálculos, 67
 componentes y parámetros, 66 y estimación, 66
 Documentos ejecutables, 251
 vista, 162
 Modelo-vista-controlador (MVC), 38, 160
 Sistema modular, 37
 codificación, 138
 Prototipado, 55
 asignación de recursos, 135
 reversibilidad, 45
 pruebas, 41, 190, 244
C++ más efectivo, 265
 Mozilla, 273
 Programación multihilo, 154
 MVC, *ver* Modelo-vista-controlador *The Mythical Man-Month*, 264

N

Nombre, variable, 249
 Nana, 114, 268
 Asignaciones de nodos, 131
 Bucle anidado, 180

Netscape, 145, 273
 Grupo de noticias, 15, 17, 33
 Sistema no ortogonal, 34
 Normalizar, 30
 Novobilski, Andrew J., 189n

O

O() notación, 178, 181
Objeto
 acoplamiento, 140n
 destrucción, 133, 134
 persistencia, 39
 Protocolo de publicación/suscripción, 158
 Soltero, 41 años
 estado válido/inválido, 154
 Espectador, 163
 Grupo de Gestión de Objetos (OMG), 270
 Object Pascal, 29
 Interfaz C, 101
Object-Oriewed Programmiwg, 189n
Object-Oriewed Softuare Cowstruction, 264
 Obsolescencia, 74
 OLTP, *consulte* el sistema de
 procesamiento de transacciones
 en línea
 OMG, *ver* Sistema de Procesamiento de
 Transacciones en Línea del Grupo de
 Gestión de Objetos
 (OLTP), 152
 Opciones, proporcionando, 3
 Ordenación, *consulte*
 Ortogonalidad del flujo de
 trabajo, 34
 codificación, 34, 36, 40
 diseño, 37
 documentación, 42
 Principio de SECO, 42
 sistema no ortogonal, 34
 productividad, 35
 Equipos de proyecto, 36, 227
 pruebas, 41
 Kits de herramientas y bibliotecas, 39
 ver also Sistema modular
 Sobre embellecimiento, 11

P

Manejo del dolor, 185
 método paint(), 173
 Pintura, 11
 Papúa Nueva Guinea, 16

Programación paralela, 150
 Loros, asesino, *véase*
 Análisis de marca, 59
 Generadores de códigos, 105
 Mensajes de registro, 196
 mini-lengua, 62
 cuerdas, 155
 Partición, 168
 Pascal, 29 años
 Generador de código pasivo, 103
 Pruebas de rendimiento, 241
 Perl, 55, 62, 99
 Interfaz C/Object Pascal,
 generación de esquemas de base de
 datos 101, página de inicio 100, 267
 Acceso a propiedades Java,
 100 herramientas eléctricas,
 270
 generación de datos de
 prueba, 100 pruebas, 197
 y composición
 tipográfica, 100
 utilidades de Unix en,
 81
 Documentación web, 101
Perl Journal, 263
 Persistencia, 39, 45
 Petzold, Carlos, 265
 Pike, Rob, 99
 Piloto
 desembarque, manipulación,
 etc., 217 que comían pescado,
 34
 Texto sin formato, 73
 vs. formato binario, 73
 inconvenientes, 74
 Documentos ejecutables, 251
 Apalancamiento, 75
 obsolescencia, 74
 y pruebas más fáciles,
 76 Unix, 76
 Polimorfismo, 111
 Post-it, 53, 55
 Constructor de energía, 55
El practice de Programmiwg, 99
 Programador pragmático
 características, xviii
 dirección de correo
 electrónico, xxiii
 sitio web, xxiii
 Pre y postcondición, 110, 113, 114
 Lógica de predicados, 110

Preprocesador, 114

Presentación, 20

- Dominio del problema, 58, 66
 metadatos, 146
- Resolución de problemas, 213
 Lista de verificación para, 214
 Productividad, 10, 35
 Programación por casualidad, 173 Personal de programación
 gastos de, 237
- Proyecto Wiwdous*, Proyecto 265
 Glosario, 210
 "Cabezas", 228
 Saboteador, 244
 horarios, 68
véase also
 Automation;
 Equipo, proyecto
- Bibliotecario del proyecto, 33, 226
- Prototipado, 53, 216
 arquitectura, 55
 Código desecharable, 56
 Tipos de, 54
 y lenguajes de programación, 55 y
 código trazador, 51
 usando, 54
- Protocolo de
 publicación/suscripción, 158 Pugh,
 Greg, 95n
 Purificar, 136
 Gestión de configuración de PVCS, 271
 Python, 55, 99, 267

Q

-
- Calidad
 control, 9
 requisitos, 11
 equipos, 225
- Credo del trabajador de la cantera, xx Algoritmo de clasificación rápida, 180

R

-
- Proceso uufificado*, 227n
 Raymond, Eric S., 273
 RCS, véase Sistema de control de revisiones Datos del mundo real, 243
- Refactorización, 5, 185
 automático, 187
 y diseño, 186
 pruebas, 187

limitaciones de tiempo, 185

Explorador de refactorización, 187, 268
Refinamiento, excesivo, 11
Regresión, 76, 197, 232, 242 Relación
 has-a, 304
 más o menos, 111, 304
lanzamientos, y SCCS, 87
Invocación de métodos remotos (RMI), 128
 control de excepciones, 39
Llamada a procedimiento remoto (RPC), 29, 39
Repositorio, 87
Requisito, 11, 202
 problema de negocios, 203
 cambiante, 26
 fluencia, 209
 DBC, 110
 distribución, 211
 documentación, 204
 en el lenguaje de dominio, 58
 expresando como invariante, 116
 métodos formales, 220
 Glosario, 210
 sobre especificar, 208
 y política, 203
 Pruebas de usabilidad, 241
 Interfaz de usuario, 203
Investigación, 15
Equilibrio de recursos, 129
 Excepciones de C++, 132
 checking, 135
 Código acoplado, 130
 estructuras de datos dinámicas, 135
 encapsulación en clase, 132
 Java, 134
 Asignaciones de nidos, 131
Conjunto de respuestas, 141, 242
Responsabilidad, 2, 250, 258
Reutilización, 33, 36
Reversibilidad, 44
 arquitectura flexible, 46 Sistema
de control de revisiones (RCS), 250,
 271
 Gestión de riesgos, 13
 ortogonalidad, 36
RMI, véase Invocación de Método
Remoto Rock-n-roll, 47
RPC, consulte Llamada a procedimiento
remoto Rubber ducking, 3, 95
Motor de reglas, 169

S

Saboteador, 244
 Samba, 272
 Programas de ejemplo, *consulte*
 Código de ejemplo Sather, 114, 268
 SCCS, véase Sistema de control de código fuente Schedule, proyecto, 68
 Schrödinger, Erwin (y su gato), 47
 Alcance, requisito, 209
 Raspado de pantalla, 61
 Lenguaje de scripting, 55, 145
 Hachís seguro, 74
sed, 99
 Sedgewick, Robert, 183
 Componentes autónomos, *consulte*
 Ortogonalidad; Cohesión
 Invariante semántica, 116, 135
 Programa SendMail, 60
 Diagrama de secuencia, 158
 Código de servidor, 196
 Servicios, diseño utilizando, 154
 Shell, comando, 77
 vs. GUI, 78
 véase also Shell de
 comandos "Código tímido", 40
 Efecto secundario, 124
 SIGPLAN, 263
 Bucle simple, 180
 Objeto Singleton, 41
 Slashdot, 265
 PequeñoEiffel, 267
 Charla trivial, 46, 186, 187, 268, 272
 Software
 tecnologías de desarrollo, 221
 calidad, 9
 requisitos, 11
 Bus de software, 159
 "Software construction", 184
 Software Developmewt magaziwe, 263
 Software IC, 189n
 "Podredumbre del software", 4
 Solaris, 76
 Código fuente
 Comer gatos, 3
 documentación, *consulte* Descarga de comentarios, *consulte* Ejemplo de duplicación de código en, 29
 generating, 103
 revisiones, *consulte* Revisiones de código

Sistema de control de código fuente (SCCS), 86
 Aegis, 246
 compilaciones usando, 88
 CVS, 271
 árbol de desarrollo, 87
 texto sin formato y, 76
 RCS, 250, 271
 Repositorio, 87
 herramientas, 271
 Especialización, 221
 Especificación, 58
 implementación, 219
 Idioma, 62
 como manta de seguridad, 219
 escritura, 218
 Células de espionaje, 138
 Chirrido, 268
 Minilenguaje autónomo, 62
 "Fatiga de puesta en marcha", 7
 Iniciar un proyecto
 Resolución de problemas, 212
 Prototipado, 216
 Especificaciones, 217
 véase also Requisito
 Stevens, W. Richard, 264
 Sopa de piedra, 7
 vs. ventanas rotas, 9
 Credo del cantero, xx
 Analizador sintáctico de cuerdas, 155
 Efecto Stroop, 249
 Rutina de Strtok , 155
 Tutoriales estructurados, *consulte*
 Revisiones de código
 Hoja de estilo, 20, 254
 Estilo, comunicación, 20
 Subclase, 112
 Algoritmo sublineal, 177
 Proveedor, véase Proveedor
Surviviwg Object-Oriewted Projects: A Mawager's Guide, 264
 SWIG, 55, 270
 Barra de sincronización, 151
 Resaltado de sintaxis, 84
 Datos sintéticos, 243

T

Espacios T, 166, 269
 TAM, véase Mecanismo de acceso a la prueba Tcl, 55, 99, 269

Equipo, proyecto, 36, 224
automatización, 229
 evitar la duplicación, 32
 Revisión de código, 236
comunicación, 225
duplicación, 226
funcionalidad, 227
organización, 227
pragmatismo en, xx
calidad, 225
Fabricantes de herramientas, 229
Redactora técnica, 252
Plantilla, caso de uso, 205
Acoplamiento temporal, 150
Mecanismo de acceso de prueba (TAM), 189 Arnés de prueba, 194
Ensayo
 automatizado, 238
 de la especificación, 29
 Corrección de errores, 247
 Análisis de cobertura, 245
 y cultura, 197
 depuración, 92, 196
 diseño/metodología, 242
 efectividad, 244
 estimaciones, 182
 frecuencia, 246
 Sistemas GUI, 244
 integración, 239
 ortogonalidad, 36, 41
 rendimiento, 241
 papel del texto sin formato, 76
 refactorización, 187
 regresión, 76, 197, 232, 242
 agotamiento de recursos, 240
 Saboteador, 244
 fecha de prueba, 100, 243
 Usabilidad, 241
 validación y verificación, 239
 ver also Pruebas unitarias
Lenguaje de manipulación de texto, 99 Lenguaje de programación TOM, 268 Kits de herramientas, 39 Herramientas, adaptables, 205 Código de trazadora, 49
 Ventajas de, 50
 y prototipado, 51 Rastreo, 94
 véase also Logging

Papel comercial, 263
Compensaciones, 249
Transacciones, EJB, 39
Widget de árbol, 161
sistema troff, 103
Espacio de tupla, 167

U

UML, *consulte* Lenguaje de modelado unificado (UML)

DESH llave, 86

Diagrama de actividades del

lenguaje unificado de modelado (UML), 150
Diagrama de secuencia, 158 Diagrama de casos de uso, 208

Principio de Acceso Uniforme, 31

Pruebas unitarias, 190

DBC, 190
módulos, 239
arnés de prueba, 194
Ventana de prueba, 196
pruebas de escritura, 193

Unix, 46, 76

Archivos predeterminados de la aplicación, 145 libros, 264

Cygwin, 270
Herramientas DOS, 270
Samba, 272
UWIN, 81, 270

Uwix Network Programming, 264 Pruebas de usabilidad, 241

Caso de uso, 204

 Diagramas, 206

Grupo de noticias de Usenet, 15, 17, 33 Usuario

 expectativas, 256
 grupos, 18
 interfaz, 203
 requisitos, 10

UWIN, 81, 270

V

Variable

 corruptos, 95
 Global, 130, 154
 nombre, 249

Proveedor

 bibliotecas, 39
 reducir la dependencia de, 36, 39, 46

vi editor, 266

Ver

- depuración, 164
- documentos ejecutables, 251
- Vista de árbol Java, 161
- modelo-vista-controlador, 160, 162
- Red de Visores de Modelos, 162
- Editor de Vim , 266
- Visual Basic, 55 años
- Visual C++, 198
- Fuente visualSeguro, 271
- Obras visuales, 268

W

- Tutoriales, *consulte* Revisiones de código
- Advertencias, compilación, 92
- Documentación web, 101, 210, 253
 - Generación automática, 235
 - Noticias e información, 265
- Servidor web, 196
- Sitio web, programador pragmático, xxiii
- Lo que ves es lo que obtienes
(WYSIWYG), 78
- WikiWikiWeb, 265
- Wiw32 Servicios del sistema,*
265
- Windows, 46
 - Comando "arroba", 231
 - libros, 265
 - Cygwin, 80 años
 - metadatos, 145
 - Bloc de notas, 84

Utilidades de Unix, 80, 81
UWIN, 81

- WinZip, 272
- SABIDURÍA acróstico, 20
- Mago, 198
- Procesador de textos, 252, 254
- Flujo de trabajo, 150
 - Sistema de pizarra, 169
 - Basado en el contenido, 234
- Envoltura, 132, 133, 135, 141
- Escritura, 18
 - consulte la documentación de also*
- www.pragmaticprogrammer.com,x
- xiii WYSIWYG, *consulte* Lo que ve es lo que obtienes

Obtienes

X

- XEmacs editor, 266
- Xerox Parc, 39
- XSL, véase eXtensible Style Language
- xUnit, 194, 269

Y

- Yacc, 59
- Yourdon, Eduardo, 10, 35
- Problema Y2K, 32, 208

Z

- Proyectil Z, 272



InformIT es una marca de Pearson y la presencia en línea para los principales editores de tecnología del mundo. Es su fuente de contenido y conocimiento confiable y calificado, que brinda acceso a las principales marcas, autores y colaboradores de la comunidad tecnológica.

▼ Addison-Wesley

Cisco Press

EXAM/CRAM

IBM
Press.

QUE

PRENTICE
HALL

SAMS

| Safari[®]

LearnIT en InformIT

¿Busca un libro, un libro electrónico o un vídeo de formación sobre una nueva tecnología? ¿Busca información y tutoriales oportunos y relevantes? ¿Busca opiniones, consejos y sugerencias de expertos? **InformIT tiene la solución.**

- Entérate de los nuevos lanzamientos y promociones especiales suscribiéndote a una amplia variedad de boletines informativos.
Visita informit.com/newsletters.
- Accede a podcasts GRATUITOS de expertos en informit.com/podcasts.
- Lea los últimos artículos de autor y capítulos de muestra en informit.com/articles.
- Accede a miles de libros y vídeos de la biblioteca digital de Safari Books Online en safari.informit.com.
- Obtén consejos de blogs de expertos en informit.com/blogs.

Visite informit.com/learn para descubrir todas las formas en que puede acceder al contenido tecnológico más popular.

¿Eres parte de la multitud de TI?

Conéctese con los autores y editores de Pearson a través de fuentes RSS, Facebook, Twitter, YouTube y más. Visita informit.com/socialconnect.





Addison
Wesley

REGISTER



ESTE PRODUCTO

informit.com/register

Registre los productos Addison-Wesley, Exam Cram, Prentice Hall, Que y Sams que posee para obtener grandes beneficios.

Para comenzar el proceso de registro, simplemente vaya a **informit.com/register** para iniciar sesión o crear una cuenta.

A continuación, se le pedirá que introduzca el ISBN de 10 o 13 dígitos que aparece en la contraportada del producto.

El registro de sus productos puede desbloquear los siguientes beneficios:

- Acceso a contenido complementario, incluidos capítulos adicionales, código fuente o archivos de proyecto.
- Un cupón para usar en tu próxima compra.

Los beneficios de registro varían según el producto. Los beneficios se enumerarán en la página de su cuenta en Productos registrados.

Acerca de InformIT : LA FUENTE CONFiable DE APRENDIZAJE TECNOLÓGICO

INFORMIT ES EL HOGAR DE LOS PRINCIPALES SELLOS EDITORIALES DE TECNOLOGÍA

Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall Professional, Que y Sams. Aquí obtendrá acceso a contenido y recursos confiables y de calidad de los autores, creadores, innovadores y líderes de la tecnología. Ya sea que esté buscando un libro sobre una nueva tecnología, un artículo útil, boletines informativos oportunos o acceso a la biblioteca digital de Safari Books Online, InformIT tiene una solución.

informIT.com

LA FUENTE CONFiable DE APRENDIZAJE
TECNOLÓGICO

Addison-Wesley | Prensa Cisco | Examen Cram
IBM Press | Que | Salón Prentice | Sams

LIBROS DE SAFARI EN LÍNEA

El programador pragmático

Esta tarjeta resume los consejos y listas de verificación que se encuentran en *El programador pragmático*.

Para obtener más información sobre THE PRAGMATIC PROGRAMMERS LLC, el código en www.pragmaticprogrammer.com.

Guía de referencia

CONSEJOS 1 A 22

- | | |
|--|--|
| 1. Preocúpate por tu oficio.....XIX
¿Por qué pasarse la vida desarrollando software si no le importa hacerlo bien? | 12. Facilite su reutilización.....33
Si es fácil de reutilizar, la gente lo hará. Crear un entorno que apoye la reutilización. |
| 2. iPensar! Sobre su trabajo.....XIX
Apague el piloto automático y tome el control. Critica y evalúa constantemente tu trabajo. | 13. Eliminar efectos entre cosas no relacionadas.....35
Diseñe componentes que sean autónomos, independientes y que tengan un propósito único y bien definido. |
| 3. Ofrezca opciones, no ponga excusas tontas3
En lugar de excusas, ofrezca opciones. No digas que no se puede hacer; Explique lo que <i>se puede</i> hacer. | 14. No hay decisiones finales46
Ninguna decisión está escrita en piedra. En su lugar, considera cada uno como si estuviera escrito en la arena de la playa y planifica el cambio. |
| 4. No vivas con ventanas rotas.....5
Corrija los malos diseños, las decisiones equivocadas y el código deficiente cuando los vea. | 15. Usa balas trazadoras para encontrar el objetivo.....49
Las balas trazadoras te permiten localizar a tu objetivo probando cosas y viendo qué tan cerca aterrizan. |
| 5. Conviértete en un catalizador del cambio8
No se puede forzar el cambio en la gente. En su lugar, muéstreles cómo podría ser el futuro y ayúdeles a participar en su creación. | 16. Prototipo para aprender54
La creación de prototipos es una experiencia de aprendizaje. Su valor no radica en el código que produces, sino en las lecciones que aprendes. |
| 6. Recuerde el panorama general8
No te sumerjas tanto en los detalles que te olvides de comprobar lo que ocurre a tu alrededor. | 17. Programa cercano al dominio del problema.....58
Diseña y codifica en el lenguaje de tu usuario. |
| 7. Hacer de la calidad una cuestión de requisitos11
Involucre a sus usuarios en la determinación de los requisitos reales de calidad del proyecto. | 18. Estimación para evitar sorpresas64
Calcule antes de comenzar. Detectarás posibles problemas desde el principio. |
| 8. Invierta regularmente en su cartera de conocimientos .14
Haz del aprendizaje un hábito. | 19. Iterar la programación con el código69
Utilice la experiencia que obtenga a medida que implemente para refinar las escalas de tiempo del proyecto. |
| 9. Analice críticamente lo que lee y escucha16
No se deje llevar por los proveedores, la exageración de los medios de comunicación o el dogma. Analiza la información en términos tuyos y de tu proyecto. | 20. Mantenga el conocimiento en texto sin formato74
El texto sin formato no se volverá obsoleto. Ayuda a aprovechar su trabajo y simplifica la depuración y las pruebas. |
| 10. Es tanto lo que dices como la forma en que lo dices ²¹
No tiene sentido tener grandes ideas si no las comunicas de manera efectiva. | 21. Usar el poder de los shells de comandos80
Utilice el shell cuando las interfaces gráficas de usuario no sean suficientes. |
| 11. SECO —Don't Repeat Yourself27
Cada pieza de conocimiento debe tener una representación única, inequívoca y autorizada dentro de un sistema. | 22. Usar bien un solo editor82
El editor debe ser una extensión de tu mano; Asegúrese de que su editor sea configurable, extensible y programable. |

23. Usar siempre el control de código fuente 88
 El control de código fuente es una máquina del tiempo para su trabajo: puede volver atrás.
24. Solucione el problema, no la culpa 91
 Realmente no importa si el error es culpa tuya o de otra persona, sigue siendo tu problema y aún necesita ser corregido.
25. Que no cunda el pánico al depurar 91
 Respira hondo y piensa en lo que podría estar causando el error.
26. "select" no está roto 96
 Es raro encontrar un error en el sistema operativo o en el compilador, o incluso en un producto o biblioteca de terceros. Lo más probable es que el error esté en la aplicación.
27. No lo asumas, demuéstralos 97
 Pruebe sus suposiciones en el entorno real, con datos reales y condiciones de contorno.
28. Aprende un lenguaje de manipulación de texto 100
 Pasas una gran parte de cada día trabajando con texto. ¿Por qué no dejar que la computadora haga algo por ti?
29. Escribir código que escribe código 103
 Los generadores de código aumentan su productividad y ayudan a evitar la duplicación.
30. No se puede escribir un software perfecto 107
 El software no puede ser perfecto. Proteja su código y a los usuarios de los errores inevitables.
31. Diseño con Contratos 111
 Utilice los contratos para documentar y verificar que el código no hace ni más ni menos de lo que dice hacer.
32. Caída prematura 120
 Un programa muerto normalmente hace mucho menos daño que uno inválido.
33. Usar aserciones para evitar lo imposible 122
 Las aserciones validan sus suposiciones. Úselos para proteger su código de un mundo incierto.
34. Usar excepciones para problemas excepcionales.. 127
 Las excepciones pueden sufrir todos los problemas de legibilidad y mantenibilidad del código espagueti clásico. Reserve excepciones para cosas excepcionales.
35. Termina lo que comienzas 129
 Siempre que sea posible, la rutina u objeto que asigna un recurso debe ser responsable de desasignarlo.
36. Minimice el acoplamiento entre módulos..... 140
 Evite el acoplamiento escribiendo código "tímido" y aplicando la Ley de Deméter.
37. Configurar, no integrar..... 144
 Implemente opciones tecnológicas para una aplicación como opciones de configuración, no a través de la integración o la ingeniería.
38. Coloque abstracciones en el código, detalles en metadatos . 145 Programa para el caso general, y poner los detalles fuera de la base de código compilada.
39. Analice el flujo de trabajo para mejorar la simultaneidad 151
 Aproveche la simultaneidad en el flujo de trabajo del usuario.
40. Diseño con servicios 154
 Diseño en términos de *servicios*: objetos independientes y concurrentes detrás de interfaces bien definidas y consistentes.
41. Diseñar siempre para la simultaneidad..... 156
 Permita la simultaneidad y diseñará interfaces más limpias con menos suposiciones.
42. Separar vistas de modelos 161
 Gane flexibilidad a bajo costo diseñando su aplicación en términos de modelos y vistas.
43. Usar Blackboards para coordinar el flujo de trabajo 169
 Utilice las pizarras para coordinar hechos y agentes dispares, manteniendo al mismo tiempo la independencia y el aislamiento entre los participantes.
44. No programe por casualidad..... 175
 Confía solo en cosas confiables. Ten cuidado con la complejidad accidental y no confundas una feliz coincidencia con un plan con propósito.
45. Calcule el orden de sus algoritmos 181
 Hazte una idea de cuánto tiempo pueden tardar las cosas *antes de escribir el código*.
46. Pon a prueba tus estimaciones..... 182
 El análisis matemático de los algoritmos no lo dice todo. Intente cronometrar el código en su entorno de destino.

47. Refactorizar temprano, refactorizar a menudo 186
 Del mismo modo que se puede desmalezar y reorganizar un jardín, reescribir, reelaborar y rediseñar el código cuando lo necesite. Solucione la raíz del problema.
48. Diseño para probar 192
 Empieza a pensar en las pruebas antes de escribir una línea de código.
49. Pruebe su software, o sus usuarios lo harán..... 197
 Pruébalo sin piedad. No hagas que tus usuarios encuentren errores por ti.
50. No utilices código de asistente que no entiendas . 199 Los magos pueden generar montones de código. Asegúrate de entenderlo *todo* antes de incorporarlo a tu proyecto.
51. No reúnas requisitos, ¡busca ellos..... 202
 Los requisitos rara vez se encuentran en la superficie. Están enterrados profundamente bajo capas de suposiciones, conceptos erróneos y política.
52. Trabajar con un usuario para pensar como un usuario 204
 Es la mejor manera de obtener información sobre cómo se utilizará realmente el sistema.
53. Las abstracciones viven más que los detalles 209
 Invierta en la abstracción, no en la implementación. Las abstracciones pueden sobrevivir al aluvión de cambios de diferentes implementaciones y nuevas tecnologías.
54. Usar un glosario de proyectos..... 210
 Crear y mantener una única fuente de todos los términos y vocabulario específicos para un proyecto.
55. No pienses fuera de la caja, encuentra la caja..... 213
 Cuando se enfrente a un problema imposible, identifique las *limitaciones reales*. Pregúntate: "¿Tiene que hacerse de esta manera? ¿Hay que hacerlo?
56. Comience cuando esté listo 215
 Has estado acumulando experiencia toda tu vida. No ignores las dudas.
57. Algunas cosas están mejor hechas que descritas . 218
 No caigas en la espiral de especificaciones, en algún momento tendrás que empezar a codificar.
58. No seas esclavo de los métodos formales..... 220
 No adoptes ciegamente ninguna técnica sin ponerla en el contexto de tus prácticas y capacidades de desarrollo.
59. Las herramientas costosas no producen mejores diseños 222
 Tenga cuidado con la exageración de los proveedores, el dogma de la industria y el aura de la etiqueta de precio. Juzgue las herramientas por sus méritos.
60. Organice los equipos en torno a la funcionalidad. 227
 No separes a los diseñadores de los programadores, a los evaluadores de los modeladores de datos. Crea equipos de la misma manera que creas código.
61. No utilices procedimientos manuales 231
 Un script de shell o un archivo por lotes ejecutará las mismas instrucciones, en el mismo orden, una y otra vez.
62. Prueba temprano. Pruebe con frecuencia. Prueba automática 237
 Las pruebas que se ejecutan con cada compilación son mucho más efectivas que los planes de prueba que se quedan en un estante.
63. La codificación no está terminada hasta que se ejecuten todas las pruebas 238
 "Ya se ha dicho bastante.
64. Utilice saboteadores para probar sus pruebas 244
 Introduzca los errores a propósito en una copia separada de la fuente para verificar que las pruebas los detectarán.
65. Cobertura de estado de prueba, no cobertura de código 245
 Identifique y pruebe los estados significativos del programa. No basta con probar líneas de código.
66. Encuentra insectos una vez 247
 Una vez que un evaluador humano encuentra un error, debería ser la *última* vez que un evaluador humano encuentre ese error. Las pruebas automáticas deberían comprobarlo a partir de ese momento.
67. El inglés es solo un lenguaje de programación 248
 Escriba documentos como escribiría código: respete el principio *DRY*, use metadatos, MVC, generación automática, etc.
68. Incorpore la documentación, no la atornille 248
 Es menos probable que la documentación creada por separado del código sea correcta y esté actualizada.
69. Supere con creces las expectativas de sus usuarios. 255

Llegue a comprender las expectativas de sus usuarios y, a continuación, ofrezca un poco más.

70. Firma tu trabajo 258

Los artesanos de una época anterior se enorgullecían de firmar su trabajo. Tú también deberías estarlo.

Listas

✓ **Idiomas** Aprender..... página

17

¿Cansado de C, C++ y Java? Pruebe CLOS, Dylan, Eiffel, Objective C, Prolog, Smalltalk o TOM. Cada uno de estos idiomas tiene diferentes capacidades y un "sabor" diferente. Pruebe un proyecto pequeño en casa usando uno o más de ellos.

✓ **EI** SABIDURÍA Acróstico Página

20

¿Qué quieres que aprendan?

¿Cuál es su interés en lo que tienes que decir?

¿Qué tan sofisticados son?

¿Cuántos detalles quieren? ¿A quién quieras que le pertenezca la información?

¿Cómo puedes motivarlos para que te escuchen?

•
•
✓ **Cómo** para mantener la ortogonalidad página

34

Diseñe componentes independientes y bien

- definidos. Mantén tu código desacoplado.
- Evite los datos globales.
- Refactorizar funciones
- similares.

✓ **Cosas** para prototipar página

53

Arquitectura

- Nueva funcionalidad en un sistema
- existente Estructura o contenido de
- datos externos Herramientas o
- componentes de terceros Problemas de
- rendimiento
- Diseño de interfaz de usuario

✓ **Arquitectónico** Preguntas .. página

55

- ¿Están bien definidas las responsabilidades? ¿Están bien definidas las colaboraciones? ¿Se minimiza el acoplamiento?
- ¿Puede identificar una posible duplicación?
- ¿Son aceptables las definiciones y restricciones de interfaz?
- ¿Pueden los módulos acceder a los datos necesarios, cuando sea necesario?

✓ **Depuración** Lista de verificaciónpágina

98

¿El problema que se informa es un resultado directo del error subyacente o simplemente un síntoma?

• ¿Está el error realmente en el compilador? ¿Está en el sistema operativo? ¿O está en tu código?

• Si le explicaras este problema en detalle a un compañero de trabajo, ¿qué le dirías?

• Si el código sospechoso pasa sus pruebas unitarias, ¿las pruebas están lo suficientemente completas? ¿Qué sucede si ejecuta la prueba unitaria con estos datos?

• ¿Las condiciones que causaron este error existen en algún otro lugar del sistema?

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

✓ **Ley** de Deméter para las funciones página 141
El método de un objeto debe llamar solo a los métodos que anhelan a:

Se
Cualquier parámetro
pasado en Objetos que
crea objetos
Componente

✓ **Cómo** para programar deliberadamente página 172

Mantente consciente de lo que estás haciendo. No codifiques con los ojos vendados.
Proceda a partir de un plan.
Confía solo en cosas confiables. Documenta tus suposiciones. Pruebe las suposiciones, así como el código. Prioriza tu esfuerzo.
No seas esclavo de la historia.

✓ **Cuando** para refactorizar página 185

Descubres una violación del principio *DRY*.
Encuentras cosas que podrían ser más ortogonales. Tus conocimientos mejoran.
Los requisitos evolucionan.
Necesita mejorar el rendimiento.

✓ **Cortante** el Nudo Gordiano ... página 212

Al resolver problemas *imposibles*,
pregúntate: ¿Existe una manera más fácil?
¿Estoy resolviendo el problema correcto?
¿Por qué es esto un problema? ¿Qué lo hace difícil?
¿Tengo que hacerlo de esta manera? ¿Hay que hacerlo?

✓ **Aspectos** de Pruebas página 237

Pruebas unitarias
Pruebas de integración
Validación y verificación
Agotamiento de recursos, errores y recuperación
Pruebas de rendimiento
Pruebas de usabilidad
Pruebas de las propias pruebas

Listas de verificación de *El programador pragmático*, por Andrew Hunt y David Thomas. Visita www.pragmaticprogrammer.com.
Derechos de autor © 2000 por Addison Wesley Longman, Inc.