

EDICIÓN ESPECIAL

El programador pragmático



Viaje a la maestría

DAVID THOMAS
ANDREW HUNT

Prologado por SARON YITBAREK

ANAYA
MULTIMEDIA

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

"Lo bueno de este libro es que es excelente para mantener fresco el proceso de programación.[*El libro*]te ayuda a seguir creciendo y claramente viene de gente que ha estado ahí."

- Kent Beck, autor de *Explicación de la programación extrema: aceptar el cambio*

"¡Encontré que este libro es una gran combinación de consejos sólidos y analogías maravillosas!"

- Martín Fowler, autor de *refactorización y UML destilado*

"Compraría una copia, la leería dos veces y luego les diría a todos mis colegas que salieran corriendo y tomaran una copia. Este es un libro que nunca prestaría porque me preocuparía que se perdiera".

- kevin ruland, Ciencias Administrativas, MSG-Logística

"La sabiduría y experiencia práctica de los autores es obvia. Los temas presentados son relevantes y útiles. . . . Con mucho, su mayor fortaleza para mí han sido las analogías sobresalientes: balas trazadoras, ventanas rotas y la fabulosa explicación basada en helicópteros de la necesidad de la ortogonalidad, especialmente en una situación de crisis. Tengo pocas dudas de que este libro eventualmente se convertirá en una excelente fuente de información útil para programadores principiantes y mentores expertos por igual".

- Juan Lacos, autor de *Diseño de software C++ a gran escala*

"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 conocen bien su oficio. . Un libro excelente.

► Pete McBreen, Consultor independiente

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

► Jared Richardson, Desarrollador senior de software,
iRenaissance, Inc.

"Me gustaría ver esto entregado a cada nuevo empleado de mi empresa. . . ."

► Chris Cleeland, Ingeniero de software senior,
Object Computing, Inc.

El programador pragmático

Esta página se dejó en blanco intencionalmente

El programador pragmático

De oficial a maestro

andres caza
David Tomás



ADDISON-WESLEY

Una huella de Addison Wesley Longman, Inc.

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

Muchas de las designaciones utilizadas por los fabricantes y vendedores para distinguir sus productos se reclaman como marcas comerciales. Donde esas designaciones aparecen en este libro, y Addison-Wesley estaba al tanto de un reclamo de marca registrada, las designaciones se han impreso en letras mayúsculas iniciales o en mayúsculas.

Las letras de la canción "The Boxer" en la página 157 son Copyright c 1968 Paul Simon. Usado con permiso del editor: Paul Simon Music. La letra de la canción "Alice's Restaurant" en la página 220 es de Arlo Guthrie, c 1966, 1967 (renovada) de APPLESEEDMETROUSICyOCAROLINA DEL NORTE. Reservados todos los derechos. Usado con permiso.

Los autores y el editor se han ocupado de la preparación de este libro, pero no ofrecen garantías expresas o implícitas de ningún tipo y no asumen responsabilidad alguna 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:

Ventas directas de AWL
Addison Wesley Longman, Inc. Una
forma de Jacob
Lectura, Massachusetts 01867
(781) 944-3700

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

Datos de catalogación en publicación de la Biblioteca del Congreso

Caza, Andrés, 1964 –
El programador pragmático / Andrew Hunt, David Tomás.
pags. cm.
Incluye referencias bibliográficas. ISBN
0-201-61622-X
I. Programación informática. I. Thomas, David, 1956–.
II. Título.
QA76.6.H857 1999
005.1--dc21 99-43581
CIP

Copyright © 2000 por Addison Wesley Longman, Inc.

Reservados todos los derechos. Ninguna parte de esta publicación puede ser reproducida, almacenada en un sistema de recuperación o transmitida, de ninguna forma o por ningún medio, ya sea electrónico, mecánico, fotocopiado, grabación o cualquier otro, 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.
Impresión del 25 de febrero de 2010

*Para Ellie y Juliet,
Elizabeth y Zachary,
estuardo y henry*

Esta página se dejó en blanco intencionalmente

Contenido

PRÓLOGO	XIII
PASAREVESTIR DE NUEVO	xvii
1 APAGMÁTICO PAGS FILOSOFIA	1
1. El gato se comió mi código fuente.....	2
2. Entropía del software.....	4
3. Sopa de piedra y ranas hervidas.....	7
4. Software suficientemente bueno.....	9
5. Su carpeta de conocimientos.....	12
6. ¡Comuníquese!.....	18
2 APAGMÁTICO ENFOQUE	25
7. Los males de la duplicación.....	26
8. Ortogonalidad.....	34
9. Reversibilidad.....	44
10. Balas trazadoras.....	48
11. Prototipos y Post-it Notes.....	53
12. Idiomas de dominio.....	57
13. Estimación.....	64
3 TÉLBASIC TOLIOS	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 PRAGMATICOS PAGSARANOIA	107
21. Diseño por contrato.....	109
22. Los programas muertos no cuentan mentiras.....	120
23. Programación asertiva.....	122
24. Cuándo utilizar excepciones.....	125
25. Cómo equilibrar los recursos.....	129
5 FINAL, O BREACCIONAR	137
26. El desacoplamiento y la Ley de Deméter.....	138
27. Metaprogramación.....	144
28. Acoplamiento temporal.....	150
29. Es solo una vista.....	157
30. Pizarrones.....	165
6 VARIOS HILE Y UNED ARE CODING	171
31. Programación por Coincidencia	172
32. Velocidad del algoritmo.....	177
33. Refactorización.....	184
34. Código fácil de probar.....	189
35. Magos malvados.....	198
7 BANTES DE LA PAGS PROYECTO	201
36. El pozo de 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 PRAGMATICOS PAGS PROYECTOS	223
41. Equipos pragmáticos.....	224
42. Automatización ubicua.....	230
43. Pruebas despiadadas.....	237
44. Todo es escritura.....	248
45. Grandes esperanzas	255
46. Orgullo y prejuicio	258

Apéndices

ArkansasRECURSOS	261
Sociedades profesionales	Construcción
de una biblioteca.	Recursos de
Internet	266
Bibliografía	275
licenciado en LetrasRESPUESTAS AMI EJERCICIOS	279
yoÍNDICE	309

Esta página se dejó en blanco intencionalmente

Prefacio

Como revisor, tuve la oportunidad temprana de leer el libro que 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 escribir este prólogo para poder explicar por qué.

En pocas palabras, este libro le dice cómo programar de una manera que pueda 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 recopilados por diseñadores de lenguajes o por los periodistas que trabajan con ellos para promocionar sus creaciones. Esos libros te dicen *cómo/hablarán* 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 piensa detenidamente, podría pensar que la programación es simplemente escribir declaraciones en un lenguaje de programación. Estaría equivocado, por supuesto, pero no podría saberlo mirando alrededor de la sección de programación de la librería.

En *El programador pragmático* Dave y Andy nos dicen 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 puedan hacer que tome el lugar de la reunión, y decidir si ese algo podría automatizarse para que el trabajo de la reunión simplemente suceda en el futuro. Entonces lo harían.

Así es como piensan Dave y Andy. Esa reunión no fue algo que les impidiera programar. Eso *estaba* programación. Y era la programación la que podía mejorarse. Sé que piensan de esta manera porque es el consejo número dos: piensa en tu trabajo.

Así que imagina que estos chicos están pensando de esta manera durante algunos años. Muy pronto tendrían una colección de soluciones. Ahora imagínelos usando sus soluciones en su trabajo durante algunos años más y descartando las que son demasiado difíciles o no siempre producen resultados. Bueno, ese enfoque casi define *pragmático*. Ahora imaginalos tomando uno o dos años más para escribir sus soluciones. Tú puedes pensar, *Esa información sería una mina de oro..* Y tú estarías bien.

Los autores nos cuentan cómo programan. Y nos lo dicen de una manera que podemos seguir. Pero hay más en esta segunda declaración de lo que podrías pensar. Dejame explicar.

Los autores han tenido cuidado de evitar proponer una teoría del desarrollo de software. Esto es una suerte, porque si lo hubieran hecho estarían obligados a deformar cada capítulo para defender su teoría. Tal deformación es la tradición en, digamos, las ciencias físicas, donde las teorías eventualmente se convierten en leyes o se descartan silenciosamente. La programación, por otro lado, tiene pocas leyes (si es que tiene alguna). Por lo tanto, los consejos de programación basados en aspirantes a leyes 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 encontré la mayor promesa en un dispositivo llamado *lenguaje de patrones*. En resumen, un *patrón* es una solución, y un lenguaje de patrones es un sistema de soluciones que se refuerzan entre sí. Toda una comunidad se ha formado en torno a la búsqueda de estos sistemas.

Este libro es más que una colección de consejos. Es un lenguaje de patrones con piel de cordero. Digo eso 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 son concretos. No encontrarás vagas abstracciones. Dave y Andy escriben directamente para usted, como si cada consejo fuera una estrategia vital para energizar su carrera como programador. Lo hacen simple, cuentan una historia, usan un toque ligero y luego continúan con respuestas a preguntas que surgirán cuando lo intente.

Y hay más. Despues de leer diez o quince consejos, comenzará a ver una dimensión adicional en el trabajo. A veces lo llamamos *QWAN*, abreviatura de *calidad sin nombre*. El libro tiene una filosofía que rezumará en tu conciencia y se mezclará con la tuya. No predica. Simplemente dice lo que funciona. Pero en la narración surge 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 seguido sobre por qué funciona. Probablemente solo te importe que funcione. Lo hace. Ya verás.

—Ward Cunningham

Esta página se dejó en blanco intencionalmente

Prefacio

Este libro te ayudará a convertirte en un mejor programador.

No importa si es un desarrollador solitario, un miembro de un gran equipo de proyecto o un consultor que trabaja con muchos clientes a la vez. Este libro le ayudará, como individuo, a hacer un mejor trabajo. Este libro no es teórico: nos concentramos en temas prácticos, en usar su experiencia para tomar decisiones más informadas. La palabra *pragmático* viene del latín *pragmático*—“experto en los negocios”—que a su vez se deriva del griego *πρᾶττειν*, que significa “hacer”. Este es un libro sobre hacer.

La programación es un oficio. En su forma más simple, se trata de 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. Intenta capturar requisitos elusivos y encontrar una manera de expresarlos para que una mera máquina pueda hacerles justicia. Intenta documentar su trabajo para que otros puedan entenderlo y trata de diseñar su trabajo para que otros puedan desarrollarlo. Lo que es más, intenta 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 vendedores de herramientas promocionan los milagros que realizan 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 cada sistema operativo es la respuesta a todos los males imaginables.

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

Aquí es donde entra en juego el pragmatismo. No debe estar atado a ninguna tecnología en particular, pero debe tener una formación y una base de experiencia lo suficientemente amplias que le permitan elegir buenas soluciones en situaciones particulares. Su formación se deriva de una 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.

Usted ajusta su enfoque para adaptarse a las circunstancias y el entorno actuales. Juzga la importancia relativa de todos los factores que afectan un proyecto y utiliza su experiencia para producir soluciones apropiadas. Y haces esto 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 se sienta frustrado porque no parece estar alcanzando su potencial. Tal vez mire a colegas que parecen estar usando herramientas para ser más productivos que usted. Tal vez su trabajo actual utiliza tecnologías más antiguas y desea saber cómo se pueden aplicar ideas más nuevas a lo que hace.

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 sigue nuestro enfoque, ganará experiencia rápidamente, su productividad aumentará y tendrá 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 aversiones. Con el tiempo, cada uno creará su propio entorno personal. Ese entorno reflejará la individualidad del programador con tanta fuerza como sus pasatiempos, vestimenta o corte de cabello. Sin embargo, si eres un programador pragmático, compartirás muchas de las siguientes características:

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

algo nuevo, puede comprenderlo rápidamente e integrarlo con el resto de su conocimiento. Su confianza nace de la experiencia.

- Inquisitivo.Tienes a hacer preguntas.*Eso es genial, ¿cómo hiciste eso? ¿Tuviste problemas con esa biblioteca? ¿Qué es este BeOS del que he oído hablar? ¿Cómo se implementan los enlaces simbólicos?* Eres un fanático de los pequeños hechos, cada uno de los cuales puede afectar alguna decisión dentro de unos años.
- Pensador crítico.Rara vez tomas las cosas como dadas sin obtener primero los hechos. Cuando los colegas dicen “porque así es como se hace”, o un proveedor promete la solución a todos sus problemas, usted huele un desafío.
- Realista.Intenta comprender la naturaleza subyacente de cada problema que enfrenta. Este realismo te da una buena idea de cuán difíciles son las cosas y cuánto tiempo llevarán. Comprender por sí mismo que un proceso de *debería ser* difícil o voluntad *tomar* un tiempo para completar te da la resistencia para seguir haciéndolo.
- Un mil usos.Se esfuerza por familiarizarse con una amplia gama de tecnologías y entornos, y trabaja para mantenerse al tanto de los nuevos desarrollos. Si bien su trabajo actual puede requerir que sea un especialista, siempre podrá avanzar hacia nuevas áreas y nuevos desafíos.

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

TIP1

Cuida tu oficio

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

TIP2

¡Pensar! Acerca de su trabajo

Para ser un programador pragmático, lo desafiamos a que piense en lo que está haciendo mientras lo hace. Esta no es una auditoría única de las prácticas actuales, es una evaluación crítica continua de cada

decisión que tomes, todos los días, y en cada desarrollo. Nunca corras en piloto automático. Esté constantemente pensando, criticando su trabajo en tiempo real. El viejo lema corporativo de IBM, ¡PENSAR!, es el mantra del programador pragmático.

Si esto le parece un trabajo duro, entonces está exhibiendo el *realista* característica. Esto va a consumir parte de su valioso tiempo, tiempo que probablemente ya esté bajo una tremenda presión. La recompensa es una participación más activa en un trabajo que amas, un sentimiento de dominio sobre una variedad cada vez mayor de temas y placer en un sentimiento de mejora continua. A largo plazo, su inversión de tiempo se verá recompensada a medida que usted y su equipo sean más eficientes, escriban código que sea más fácil de mantener y pasen menos tiempo en reuniones.

Pragmáticos individuales, grandes equipos

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

No estamos de acuerdo.

La construcción del software *debería 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 hicieron avanzar 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 sostuvo los proyectos:

Nosotros, que cortamos simples piedras, siempre debemos estar imaginando catedrales.

— **Credo del trabajador 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 visitaba el Eton College de Inglaterra le preguntó al jardinero cómo había conseguido que el césped fuera tan perfecto. "Eso es fácil", respondió, "simplemente quitas el rocío todas las mañanas, los cortas cada dos días y los enrollas una vez a la semana".

"¿Eso es todo?" preguntó el turista.

"Absolutamente", respondió el jardinero. "Haz eso durante 500 años y también tendrás un hermoso césped".

Los grandes céspedes necesitan pequeñas cantidades de cuidado diario, al igual que los grandes programadores. A los consultores de gestión les gusta correr la voz *kaizen* en conversaciones. "Kaizen" es un término japonés que capta el concepto de realizar continuamente muchas pequeñas mejoras. Se consideró que era una de las principales razones de las ganancias dramáticas en productividad y calidad en la fabricación japonesa y fue ampliamente copiado en todo el mundo. Kaizen también se aplica a las personas. Todos los días, trabaje para refinar las habilidades que tiene y agregue nuevas herramientas a su repertorio. A diferencia de los céspedes de Eton, comenzará a ver los resultados en cuestión de días. A lo largo de los años, se sorprenderá de cómo ha florecido su experiencia y cómo han crecido sus habilidades.

Cómo está organizado el libro

Este libro está escrito como una colección de secciones breves. Cada sección es independiente y aborda un tema en particular. Encontrará numerosas referencias cruzadas, que ayudan a poner cada tema en contexto. Siéntase libre de leer las secciones en cualquier orden; este no es un libro que necesita leer de adelante hacia atrás.

De vez en cuando te encontrarás con una caja etiquetada *Consejo nn* (como el Consejo 1, "Cuide su artesanía" en la página xix). Además de enfatizar puntos en el texto, creemos que los consejos tienen vida propia: los vivimos a diario. Encontrará 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 URL de recursos web y una lista de publicaciones periódicas, libros y organizaciones profesionales recomendadas. A lo largo del libro encontrará referencias a la bibliografía ya la lista de URL, como [KP99] y [URL 18], respectivamente.

Hemos incluido ejercicios y desafíos en su caso. Los ejercicios normalmente tienen respuestas relativamente sencillas, mientras que los desafíos son más abiertos. Para darle una idea de nuestro pensamiento, hemos incluido nuestras respuestas a los ejercicios en el Apéndice B, pero muy pocos tienen una sola *correcta* solución. Los desafíos pueden formar la base de discusiones grupales o trabajos de ensayo en cursos de programación avanzada.

¿Lo que hay en un nombre?

“Cuando uso una palabra”, dijo Humpty Dumpty, en un tono más bien desdeñoso, “significa exactamente lo que elijo que signifique, ni más ni menos”.

► **Lewis Carroll, *Através del espejo***

Dispersos a lo largo del libro, encontrará varios fragmentos de jerga, ya sean palabras en inglés perfectamente buenas que se han corrompido para que signifiquen algo técnico, o palabras inventadas horrendas a las que los científicos informáticos les han asignado significados con rencor contra el idioma. La primera vez que usamos cada una de estas palabras de la jerga, tratamos de definirlas, o al menos dar una pista de su significado. Sin embargo, estamos seguros de que algunos han pasado desapercibidos y otros, como *objeto y base de datos relacional*, son de uso tan común que agregar una definición sería aburrido. Si usted *hace* encuentra un término que no ha visto antes, por favor no lo salte. 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 tiene la oportunidad, envíenos un correo electrónico y presente una queja para que podamos agregar una definición en 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 normalmente se restringe a un dominio de problema particular, o a una fase particular de 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 queríamos usar la palabra de la jerga convencional en un contexto más amplio, se volvió confuso: parecía que no podíamos superar el equipaje que traía el término original. Cuando esto sucedió, contribuimos al declive del idioma al inventar 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 descargar desde nuestro sitio web:

www.pragmaticprogrammer.com

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

Envianos tus comentarios

Apreciaríamos saber de usted. Comentarios, sugerencias, errores en el texto y problemas en los ejemplos son bienvenidos. Envíanos un email a

ppbook@pragmaticprogrammer.com

Expresiones de gratitud

Cuando comenzamos a escribir este libro, no teníamos idea de cuánto esfuerzo de equipo terminaría siendo.

Addison-Wesley ha sido brillante, tomando 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 editor entusiasta (¡y un diseñador de portadas mezquino!), a Lorraine Ferrier y John Fuller por su ayuda con la producción, y a la incansable Julie DeBaggis por mantenernos unidos.

Luego estaban los revisores: 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 sus valiosas ideas, este libro sería menos legible, menos preciso y el doble de largo. Gracias a todos por su tiempo y sabiduría.

La segunda impresión 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, tanto por encontrar los errores como por tener la gracia de señalar sacarlos suavemente.

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

Este libro fue producido usando L^AT_EX, 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 sorprendente es que todo este tremendo software está disponible gratuitamente. 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. En particular, nos gustaría agradecer 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 máquina hasta 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.

*andy caza
david thomas*

Capítulo 1

Una filosofía pragmática

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

Otra clave de su éxito es que asumen la responsabilidad de todo lo que hacen, de lo que hablaremos en *El gato se comió mi código fuente*. Al ser responsables, los programadores pragmáticos no se quedarán de brazos cruzados y verán cómo sus proyectos se desmoronan por negligencia. En *Entropía del software*, te decimos 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 simple inercia. En *Sopa de piedra y ranas hervidas*, analizamos una estrategia para instigar el cambio y (en aras del equilibrio) presentamos la historia con moraleja de un anfibio que ignoró los peligros del cambio gradual.

Uno de los beneficios de comprender el contexto en el que trabaja es que se vuelve más fácil saber qué tan bueno debe ser su software. A veces, casi la perfección es la única opción, pero a menudo hay compensaciones involucradas. Exploramos esto en *Software suficientemente bueno*.

Por supuesto, debe tener una amplia base de conocimientos y experiencia para lograr todo esto. El aprendizaje es un proceso continuo y permanente. En *Su carpeta de conocimientos*, discutimos algunas estrategias para mantener el impulso.

Finalmente, ninguno de nosotros trabaja en el vacío. Todos pasamos una gran cantidad de tiempo interactuando con los demás. *Comunicar* enumera las formas en que podemos hacer esto mejor.

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

1

El gato se comió mi código fuente

La mayor de todas las debilidades es el miedo a parecer débil.

► JB Bossuet, *Política de la Sagrada Escritura*, 1709

Uno de los pilares de la filosofía pragmática es la idea de asumir la responsabilidad de uno mismo y de sus acciones en términos de su desarrollo profesional, su proyecto y su trabajo diario. Un programador pragmático se hace cargo de su propia carrera y no tiene miedo de admitir la ignorancia o el error. Sin duda, no es el aspecto más agradable de la programación, 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 habilidades, pero debemos ser honestos acerca de nuestras deficiencias, nuestra ignorancia y 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 cada aspecto. Además de hacer su mejor esfuerzo personal, debe analizar la situación en busca de riesgos que estén fuera de su control. tienes derecho *no* a asumir la responsabilidad de una situación imposible, o en la que los riesgos son demasiado grandes. Tendrás que hacer la llamada en base a tu propia ética y criterio.

Cuando usted *hacer* acepta la responsabilidad de un resultado, debe esperar que se le haga responsable de ello. Cuando cometa un error (como todos lo hacemos) o un error de juicio, admítalo honestamente y trate de ofrecer opciones.

No culpes a alguien o algo más, ni inventes una excusa. No culpe de todos los problemas a un proveedor, un lenguaje de programación, la gerencia o sus compañeros de trabajo. Cualquiera y todos estos pueden desempeñar un papel, pero depende de *tú* para dar soluciones, no excusas.

Si existía el riesgo de que el proveedor no cumpliera con usted, entonces debería haber tenido un plan de contingencia. Si el disco falla, llevándose todo su código fuente, y no tiene una copia de seguridad, es su culpa. Decirle a su jefe "el gato se comió mi código fuente" simplemente no es suficiente.

TIP3

Proporcione opciones, no ponga excusas tontas

Antes de acercarse a alguien para decirle por qué algo no se puede hacer, está atrasado o no funciona, deténgase y escúchese a sí mismo. Habla con el patito de goma en tu monitor o con el gato. ¿Tu excusa suena razonable o estúpida? ¿Cómo le va a sonar a tu jefe?

Ejecute la conversación en su mente. ¿Qué es probable que diga la otra persona? ¿Preguntarán: "¿Has probado esto?" o "¿No consideraste eso?" ¿Cómo responderás? Antes de ir y decirles las malas noticias, ¿hay algo más que pueda probar? A veces, simplemente *saber* lo que van a decir, así que ahórrales la molestia.

En lugar de excusas, ofrezca opciones. No digas que no se puede hacer; explica que *pueden* hacerse para salvar la situación. ¿Hay que tirar el código? Edúquelos sobre el valor de la refactorización (ver *refactorización*, página 184). ¿Necesita dedicar tiempo a la creación de prototipos para determinar la mejor manera de proceder (ver *Prototipos y Post-it Notes*, página 53)? ¿Necesita introducir mejores pruebas (ver *Código que es fácil de probar*, página 189, y *Pruebas despiadadas*, página 237) o automatización (ver *Automatización ubicua*, página 230) para evitar que vuelva a suceder? Tal vez necesite recursos adicionales. No tenga miedo de preguntar o admitir que necesita ayuda.

Trate 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 cargar con la culpa....

Las secciones relacionadas incluyen:

- *Prototipos y Post-it Notes*, página 53
- *refactorización*, página 184
- *Código que es fácil de probar*, página 189
- *Automatización ubicua*, página 230
- *Pruebas despiadadas*, página 237

Desafíos

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

► Entropía del software

Si bien el desarrollo de software es inmune a casi todas las leyes físicas, *entropía* nos golpea fuerte. *entropía* 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 aumenta el desorden en el software, los programadores lo llaman "podredumbre del software".

Hay muchos factores que pueden contribuir a que el software se pudra. El más importante parece ser la psicología, o la cultura, en el trabajo de un proyecto. Incluso si es un equipo de uno, la psicología de su proyecto puede ser algo muy delicado. A pesar de los mejores planes y la mejor gente, un proyecto aún puede experimentar la ruina y el deterioro durante su vida útil. Sin embargo, hay otros proyectos que, a pesar de las enormes dificultades y los constantes contratiempos, combaten con éxito la tendencia de la naturaleza al desorden y logran salir bastante bien.

¿Qué hace la diferencia?

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



Una ventana rota.

Una ventana rota, que no se ha reparado durante un período de tiempo considerable, 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 otra ventana se rompe. La gente empieza a tirar basura. Aparece el grafiti. Comienzan serios daños estructurales. 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 hace realidad.

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

TIP4

No vivas con las ventanas rotas

No deje "ventanas rotas" (malos diseños, decisiones equivocadas o código deficiente) sin reparar. Repare cada uno tan pronto como se descubra. Si no hay tiempo suficiente para arreglarlo correctamente, entonces *subirlo a bordo*. Quizás pueda comentar el código ofensivo, o mostrar un mensaje "No implementado", o sustituir datos ficticios en su lugar. Tomar *alguna acción* para evitar más daños y demostrar que está al tanto de la situación.

Hemos visto sistemas limpios y funcionales que se deterioran con bastante rapidez una vez que las ventanas comienzan a romperse. Hay otros factores que pueden contribuir a que el software se pudra, y mencionaremos algunos de ellos en otro lugar, pero descuidemos *acelerarla* podredumbre más rápido que cualquier otro factor.

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

Apagando fuegos

Por el contrario, está la historia de un conocido obscenamente rico de Andy. Su casa estaba inmaculada, hermosa, llena de antigüedades de valor incalculable, *objetos de arte*, y así. Un día, un tapiz que colgaba demasiado cerca de la chimenea de su sala se incendió. El fuego

El departamento se apresuró a salvar el día y su casa. Pero antes de que arrastraran sus mangueras grandes y sucias dentro de la casa, se detuvieron, con el fuego furioso, 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, una pieza de código mal diseñada, 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 basura, simplemente haré lo mismo". 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 despojado y volcado dentro de *horas*.

De la misma manera, si se encuentra en un equipo y un proyecto donde el código es impecablemente hermoso (escrito de manera limpia, bien diseñado y elegante), es probable que tenga especial cuidado de no estropearlo, al igual que los bomberos. Incluso si hay un gran incendio (fecha límite, fecha de lanzamiento, demostración comercial, etc.), *tú* no quiero ser el primero en hacer un lío.

Las secciones relacionadas incluyen:

- *Sopa de piedra y ranas hervidas*, página 7
- *refactorización*, página 184 *Equipos*
- *pragmáticos*, página 224

Desafíos

- Ayude a fortalecer su equipo encuestando a 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 decir 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 puede hacer al respecto?

3

Sopa de piedra y ranas hervidas

Los tres soldados que regresaban a casa de la guerra tenían hambre. Cuando vieron el pueblo más adelante, se animaron: estaban seguros de que los aldeanos les darían de comer. Pero cuando llegaron allí, encontraron las puertas cerradas y las ventanas cerradas. Despues de muchos años de guerra, a los aldeanos les faltaba comida y atesoraban lo que tenían.

Sin inmutarse, los soldados hirvieron una olla de agua y cuidadosamente colocaron tres piedras en ella. Los asombrados aldeanos salieron a mirar.

"Esto es sopa de piedra", explicaron los soldados. "¿Eso es todo lo que pones?" preguntaron los aldeanos. "Absolutamente, aunque algunos dicen que sabe aún mejor con algunas zanahorias. Un aldeano salió corriendo y regresó al poco tiempo con una cesta de zanahorias de su tesoro.

Un par de minutos después, los aldeanos volvieron a preguntar "¿Es eso?"

"Bueno", dijeron los soldados, "un par de papas le dan cuerpo". Salió corriendo otro aldeano.

Durante la siguiente hora, los soldados enumeraron más ingredientes que mejorarían la sopa: carne de res, puerros, sal y hierbas. Cada vez, un aldeano diferente saldría corriendo para asaltar sus tiendas personales.

Finalmente, habían producido una gran olla de sopa humeante. Los soldados quitaron las piedras y se sentaron con todo el pueblo para disfrutar de la primera comida completa que habían comido en meses.

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

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

Es posible que se encuentre en una situación en la que sepa exactamente lo que necesita 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 cuidará sus propios recursos. A veces esto se llama "fatiga de puesta en marcha".

Es hora de sacar las piedras. averigua lo que *tupueden* pedir razonablemente. Desarróllalo bien. Una vez que lo tengas, muéstralos a la gente y deja que se maravillen. Luego diga “por supuesto, *haríasería* mejor si agregamos.. Finge que no es importante. Siéntese y espere a que comiencen a pedirle que agregue la funcionalidad que deseaba originalmente. A las personas les resulta más fácil unirse a un éxito continuo. Muéstrelas un atisbo del futuro y conseguirá que se reúnan.¹

TIP5

Sea un catalizador para el cambio

El lado de los aldeanos

Por otro lado, la historia de la sopa de piedras también se trata de un engaño suave y gradual. Se trata de enfocarse demasiado. Los aldeanos piensan en las piedras y se olvidan del resto del mundo. Todos caemos en eso, todos los días. Las cosas simplemente se nos escapan.

Todos hemos visto los síntomas. Los proyectos lenta e inexorablemente se salen totalmente de control. La mayoría de los desastres de software comienzan siendo demasiado pequeños como para notarlo, 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.

TIP6

Recuerde el panorama general

Nunca hemos probado esto, honesto. Pero dicen que si tomas una rana y la dejas caer en agua hirviendo, saltará de nuevo. Sin embargo, si colocas la rana en una cacerola con agua fría y luego la calientas gradualmente, la rana no notará el lento aumento de la temperatura y permanecerá en su lugar hasta que esté cocida.

1. Mientras hace esto, puede sentirse 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 discutido 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. Mantenga un ojo en el panorama general. Revisa constantemente lo que sucede a tu alrededor, no solo lo que estás haciendo personalmente.

Las secciones relacionadas incluyen:

- *Entropía del software*, página 4 *Programación por coincidencia*, página 172 *refactorización*, página 184
- *El pozo de requisitos*, página 202
- *Equipos pragmáticos*, página 224

Desafíos

- Mientras revisaba un borrador de este libro, John Lakos planteó el siguiente tema: 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 piedras o sopa de ranas cuando intentas catalizar el cambio? ¿La decisión es subjetiva u objetiva?

4

Software suficientemente bueno

Esforzándonos por mejorar, a menudo estropeamos lo que está bien.

► **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 en 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: "Estas son las defectuosas".

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 software libre de errores. El tiempo, la tecnología y el temperamento conspiran contra nosotros.

Sin embargo, esto no tiene por qué ser frustrante. Como Ed Yourdon describió en un artículo en *Software IEEE* [You95], puede disciplinarse para escribir software que sea lo suficientemente bueno, lo suficientemente bueno para sus usuarios, para futuros mantenedores, para su propia tranquilidad. Descubrirá que es más productivo y que sus usuarios están más contentos. Y es posible que descubras que tus programas son realmente mejores para su incubación más corta.

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

Involucre a sus usuarios en la compensación

Normalmente estás escribiendo software para otras personas. A menudo recordará obtener los requisitos de ellos.² Pero, ¿con qué frecuencia les preguntas *que bien 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 será ampliamente difundida, los requisitos serán más estrictos y sus opciones más limitadas. Sin embargo, si está trabajando en un producto completamente nuevo, tendrá diferentes limitaciones. El personal de marketing tendrá promesas que cumplir, los eventuales usuarios finales pueden haber hecho planes basados en un cronograma de entrega y su empresa seguramente tendrá restricciones de flujo de caja. Sería poco profesional ignorar los requisitos de estos usuarios simplemente para agregar nuevas funciones 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 básicos para cumplir con un plazo.

2. ¡Se suponía que era una broma!

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

TIP7

Haga de la calidad una cuestión de requisitos

A menudo, se encontrará en situaciones en las que hay que hacer concesiones.

Sorprendentemente, muchos usuarios prefieren usar software con algunas asperezas *Este Día* que esperar un año para la versión multimedia. Muchos departamentos de TI con presupuestos ajustados estarían de acuerdo. El gran software de hoy suele ser preferible al software perfecto de mañana. Si les da a sus usuarios algo con lo que jugar desde el principio, sus comentarios a menudo lo llevarán a una mejor solución eventual (consulte *Balas trazadoras*, página 48).

Sepa cuándo parar

De alguna manera, la programación es como pintar. Comienzas con un lienzo en blanco y ciertas materias primas básicas. Utiliza una combinación de ciencia, arte y artesanía para determinar qué hacer con ellos. Usted esboza una forma general, pinta el entorno subyacente y luego completa los detalles. Constantemente retrocedes con ojo crítico para ver lo que has hecho. De vez en cuando tirarás un lienzo y empezarás de nuevo.

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

No eche a perder un programa perfectamente bueno con adornos y refinamientos excesivos. Continúe y deje que su código se sostenga por sí mismo por 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:

- *Balas trazadoras*, página 48 *El pozo de*
- *requisitos*, página 202 *Equipos*
- *pragmáticos*, página 224 *Grandes*
- *expectativas*, página 255

Desafíos

- Fíjese en los fabricantes de las herramientas de software y los sistemas operativos que utiliza. ¿Puede encontrar alguna evidencia de que estas empresas se sientan 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 u (3) optar por un software más simple con menos defectos?
- Considere el efecto de la modularización en la entrega de software. ¿Tomará más o menos tiempo lograr que un bloque monolítico de software tenga la calidad requerida en comparación con un sistema diseñado en módulos? ¿Puedes encontrar ejemplos comerciales?

5

Su carpeta de conocimientos

Una inversión en conocimiento siempre paga el mejor interés.

► **Benjamin Franklin**

Ah, el bueno de Ben Franklin, nunca le falta una homilía concisa. ¿Por qué, si pudiéramos acostarnos temprano y levantarnos temprano, seríamos grandes programadores, verdad? El pájaro madrugador puede conseguir el gusano, pero ¿qué sucede con el gusano madrugador?

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

Desafortunadamente, son *activos que expiran*.³ Su conocimiento se vuelve obsoleto a medida que se desarrollan nuevas técnicas, lenguajes y entornos. Cambiar las fuerzas del mercado puede hacer que su experiencia sea obsoleta o irrelevante. Dada la velocidad a la que vuelan los años de la Web, esto puede suceder bastante rápido.

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

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

Su carpeta de conocimientos

Nos gusta pensar en todos los hechos que los programadores conocen sobre computación, los dominios de aplicación en los que trabajan y toda su experiencia como su *Portafolios de conocimiento*. La gestión de una cartera de conocimientos es muy similar a la gestión de una cartera financiera:

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 su cartera

- Invierta regularmente. Al igual que en la inversión financiera, debe invertir en su cartera de conocimientos *regularmente*. Incluso si es solo una pequeña cantidad, el hábito en sí mismo es tan importante como las sumas. Algunos objetivos de muestra se enumeran en la siguiente sección.
- Diversificar. Cuanto más *diferentes* cosas que sabes, más valioso eres. Como línea de base, necesita conocer los entresijos de la tecnología particular con la que está trabajando actualmente. Pero no te detengas allí. La cara de la informática cambia rápidamente: la tecnología de hoy puede ser casi inútil (o al menos no tener demanda) mañana. Cuantas más tecnologías te resulten cómodas, mejor podrás adaptarte al cambio.
- Gestionar el riesgo. La tecnología existe a lo largo de un espectro desde arriesgado, potencialmente de alta recompensa a estándares 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 invertirlo todo de manera conservadora y perder posibles oportunidades. No ponga todos sus huevos técnicos en una canasta.

- Compra barato, vende 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.
- Revisar y reequilibrar.Esta es una industria muy dinámica. Esa tecnología candente que comenzaste a investigar el mes pasado podría estar fría como una piedra ahora. Tal vez necesite repasar esa tecnología de base de datos que no ha usado en mucho tiempo. O tal vez podría estar mejor posicionado para esa nueva oferta de trabajo si probara ese otro idioma. . . .

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

TIP8

Invierta regularmente en su cartera de conocimientos

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? Aquí hay 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 estancado en la rutina. Además, ahora es mucho más fácil aprender muchos idiomas, 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 tu proyecto actual. Una vez que tenga el hábito, lea un libro al mes. Una vez que haya dominado las tecnologías que está utilizando actualmente, diversifique y estudie algunas que no estén relacionadas con su proyecto.
- Lee también libros no técnicos.Es importante recordar que las computadoras son utilizadas por gente—personas cuyas necesidades intentas satisfacer. No olvides el lado humano de la ecuación.

- Tomar clases. Busque cursos interesantes en su colegio comunitario o universidad local, o tal vez en la próxima feria comercial que se realice en la ciudad.
- Participar en grupos de usuarios locales. No se limite a ir y escuchar, sino a participar activamente. El aislamiento puede ser mortal para tu carrera; averigüe en qué está trabajando la gente fuera de su empresa.
- Experimenta con diferentes entornos. Si ha trabajado solo en Windows, juegue con Unix en casa (el Linux disponible gratuitamente es perfecto para esto). Si has usado solo archivo MAKEs y un editor, pruebe con un IDE y viceversa.
- Estar al día. Suscríbase a revistas especializadas y otros diarios (consulte la página 262 para obtener recomendaciones). Elija algunos que cubran una tecnología diferente a la de su proyecto actual.
- Conéctate. ¿Quiere conocer los entresijos de un nuevo idioma u otra tecnología? Los grupos de noticias son una excelente manera de averiguar qué experiencias tienen otras personas con ellos, la jerga particular que usan, etc. Navegue por Internet 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 un nuevo idioma o un poco de tecnología, sigue adelante. Aprende otro.

No importa si alguna vez usó alguna de estas tecnologías en un proyecto, o incluso si las incluyó en su currículum. El proceso de aprendizaje expandirá tu pensamiento, abriéndote a nuevas posibilidades y nuevas formas de hacer las cosas. La polinización cruzada de ideas es importante; trate de aplicar las lecciones que ha aprendido a su proyecto actual. Incluso si su proyecto no usa esa tecnología, tal vez pueda tomar prestadas algunas ideas. Familiarícese con la orientación a objetos, por ejemplo, y escribirá programas simples en C de manera diferente.

Oportunidades de aprendizaje

Así que estás leyendo vorazmente, estás al tanto de los últimos avances 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.

No dejes que se detenga allí. Tómalo como un desafío personal para encontrar la respuesta. Pregúntale a un gurú. (Si no tiene un gurú en su oficina, debería poder encontrar uno en Internet: vea el cuadro en la página opuesta). Busque en la Web. Ir a la biblioteca.⁴

Si no puede encontrar la respuesta usted mismo, averigüe quién *pueden*. No dejes que descance. Hablar con otras personas ayudará a construir su red personal, y puede sorprenderse al encontrar 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 escasea. Así que necesitas planificar con anticipación. Siempre tenga algo para leer en un momento 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 traer su propia revista, o puede encontrarse 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 en su cartera sea preciso y no se deje influenciar por el proveedor o la exageración de los medios. Cuidado con los fanáticos que insisten en que su dogma proporciona la sola *verdadera* respuesta: puede o no ser aplicable a usted y a su proyecto.

Nunca subestimes el poder del comercialismo. El hecho de que un motor de búsqueda en la Web incluya una coincidencia en primer lugar no significa que sea la mejor coincidencia; el proveedor de contenido puede pagar para obtener la mejor facturación. El hecho de que una librería presente un libro de manera destacada no significa que sea un buen libro, o incluso popular; es posible que les hayan pagado para colocarlo allí.

TIP9

Analice críticamente lo que lee y escucha

Desafortunadamente, ya hay muy pocas respuestas simples. Pero con su extensa cartera, y aplicando un poco de análisis crítico a la

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

Cuidado y Cultivo de Gurús

Con la adopción global de Internet, los gurús de repente están tan cerca como tuIngresarllave. Entonces, ¿cómo encuentra uno y cómo logra que uno hable con usted?

Encontramos que hay algunos trucos simples.

- Sepa exactamente lo que quiere preguntar y sea lo más específico posible.
- Formule su pregunta con cuidado y cortesía. Recuerda que estás pidiendo un favor; no parece estar exigiendo una respuesta.
- Una vez que haya formulado su pregunta, deténgase y busque nuevamente la respuesta. Elija algunas palabras clave y busque en la Web. Busque las preguntas frecuentes apropiadas (listas de preguntas frecuentes con respuestas).
- Decide si quieras 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 puede enviar un correo electrónico a su gurú directamente. De cualquier manera, use una línea de asunto significativa. ("¡¡¡Necesitas ayuda!!!"
no lo corta.)
- Siéntese y sea 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 gente haciendo preguntas tú puedes responder, hacer tu parte y participar.

torrente de publicaciones técnicas que leerás, puedes entender el *complejo* respuestas

Desafíos

- Comienza a aprender un nuevo idioma esta semana. ¿Siempre programado en C++? Prueba Smalltalk [URL 13] o Squeak [URL 14]. ¿Haciendo Java? Pruebe Eiffel [URL 10] o TOM [URL 15]. Consulte la página 267 para conocer las fuentes de otros compiladores y entornos gratuitos.
- Comienza a leer un libro nuevo (¡pero termina este primero!). Si está haciendo una implementación y codificación muy detallada, lea un libro sobre diseño y arquitectura. Si está haciendo diseño de alto nivel, lea 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. Red en la cafetería de su empresa, o tal vez busque a otros entusiastas en una reunión del grupo de usuarios local.

6

¡Comunicar!

Creo que es mejor ser mirado por encima de lo que es pasado por alto.

► **mae oeste, Bella de los noventa, 1934**

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

Como desarrolladores, tenemos que comunicarnos en 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 futuras generaciones de desarrolladores. Redactamos 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. Pasamos gran parte de nuestro día comunicándonos, por lo que debemos hacerlo bien.

Hemos reunido una lista de ideas que encontramos útiles.

Sepa lo que quiere decir

Probablemente, la parte más difícil de los estilos de comunicación más formales que se usan en los negocios es saber exactamente qué es lo que quieras 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 empiezan a escribir lo que les venga a la cabeza a continuación.

Planifica lo que quieras decir. Escribe un esquema. Luego pregúntese: "¿Esto transmite lo que estoy tratando de decir?" Refina hasta que lo haga.

Este enfoque no solo es aplicable a la redacción de documentos. Cuando se enfrente a una reunión importante o una llamada telefónica con un cliente importante, anote las ideas que desea comunicar y planifique un par de estrategias para transmitirlas.

Conozca a su audiencia

Te estás comunicando sólo si estás transmitiendo información. Para hacerlo, debe comprender las necesidades, los intereses y las capacidades de su audiencia. Todos nos hemos sentado en reuniones en las que un geek del desarrollo deslumbra al vicepresidente de marketing con un largo monólogo sobre los méritos de alguna tecnología arcana. Esto no es comunicar: es solo hablar, y es molesto.⁵

Forme una imagen mental fuerte de su audiencia. el acrósticosABIDURÍA, 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 poder enviar informes de errores las 24 horas del día sin tener que esperar al teléfono. Su departamento de marketing podrá utilizar este hecho para impulsar las ventas. Los gerentes del departamento de soporte tendrán dos motivos para estar contentos: se necesitará menos personal y se automatizará el informe de problemas. Finalmente, los desarrolladores pueden disfrutar adquiriendo experiencia con las tecnologías cliente-servidor basadas en Web y un nuevo motor de base de datos. Al hacer el lanzamiento apropiado para cada grupo, logrará que todos se entusiasmen con su 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 su jefe está en el hospital, afuera está lloviendo a cántaros y el viaje a casa seguramente 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 su audiencia necesita escuchar, debe determinar cuáles son sus prioridades. Atrapa a un gerente a quien su jefe le acaba de hacer pasar un mal rato porque se perdió un código fuente, y

5. La palabra *enojarse* proviene del francés antiguo *enui*, que también significa "aburrir".

Figura 1.1. la SABIDURÍAacróstico—comprensión de una audiencia

WoQué quieres que aprendan?
 ¿Cuál es su interés en lo que tienes que decir?
 Cómo**s**on sofisticados?
 Cuánto**d**etalles quieren? a quien
 quieres**o**Tienes la información?
 Como puedes**metr**o¿Los motivas a que te escuchen?

tendrá un oyente más receptivo a sus ideas sobre los repositorios de 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

Ajuste el estilo de su entrega para adaptarse a su audiencia. Algunas personas quieren una sesión informativa formal de "solo los hechos". A otros les gusta una conversación 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 memorando o correo electrónico. Si tienes dudas pregunta.

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

Haz que se vea bien

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

Demasiados desarrolladores (y sus gerentes) se concentran únicamente en el contenido cuando producen documentos escritos. Creemos que esto es un error. Cualquier chef te dirá que puedes esclavizarte 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 mala apariencia. Procesadores de texto modernos (junto con sistemas de diseño como LATEX y troff) pueden producir resultados sorprendentes. Necesitas aprender solo algunos comandos básicos. Si su procesador de textos admite hojas de estilo, utilice

a ellos. (Es posible que su empresa ya tenga hojas de estilo definidas que puede usar). Aprenda a configurar encabezados y pies de página. Mire los documentos de muestra incluidos con su paquete para obtener ideas sobre estilo y diseño. *Revisar la ortografía*, si primero automáticamente y luego a mano. Después del punzón, hay filetes de señorita de ortografía que el corrector puede anudar ketch.

Involucra a tu audiencia

A menudo encontramos 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. Obtenga sus comentarios y escoja sus cerebros. Construirá una buena relación de trabajo y probablemente producirá un mejor documento en el proceso.

ser un oyente

Hay una técnica que debes usar si quieras que la gente te escuche: *escúchalo*s. Incluso si esta es una situación en la que tiene toda la información, incluso si se trata de una reunión formal con usted parado frente a 20 trajes, si no los escucha, ellos no lo escucharán.

Anime a las personas a hablar haciéndoles preguntas o pídale que resuman lo que les dice. Convierta la reunión en un diálogo y expresará su punto de manera más efectiva. Quién sabe, tal vez 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 responde a las personas cuando le envían un correo electrónico o un memorando solicitando información o solicitando alguna acción? En el ajetreo de la vida cotidiana, es fácil olvidar. Responda siempre 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 las hace mucho más indulgentes con los deslices ocasionales y les hace sentir que no las ha olvidado.

TIP10

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

A menos que trabaje en un vacío, debe poder comunicarse. Cuanto más efectiva sea esa comunicación, más influyente serás.

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 interempresariales. El correo electrónico se utiliza para discutir contratos, 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 enviar correos electrónicos de aspecto desagradable alrededor del mundo.

Nuestros consejos por correo electrónico son simples:

- Revisa antes de golpear ENVIAR.
- Revisar la ortografía.
- Mantenga el formato simple. Algunas personas leen el correo electrónico utilizando fuentes proporcionales, por lo que las imágenes de arte ASCII que ha creado laboriosamente les parecerán rayas de gallina.
- Use correo con texto enriquecido o formato HTML solo si sabe que todos sus destinatarios pueden leerlo. El texto sin formato es universal.
- Trate de mantener las cotizaciones al mínimo. A nadie le gusta recibir su propio correo electrónico de 100 líneas con el mensaje "Estoy de acuerdo".
- Si está citando el correo electrónico de otras personas, asegúrese de atribuirlo y citarlo en línea (en lugar de como un archivo adjunto).
- No llames a menos que quieras que vuelva y te persiga más tarde.
- Revisa tu lista de destinatarios antes de enviar. Un reciente *Wall Street Journal* El artículo describía a un empleado que se dedicaba a distribuir críticas a su jefe por correo electrónico del departamento, sin darse cuenta de que su jefe estaba incluido en la lista de distribución.
- Archive y organice su correo electrónico, tanto el material importante que recibe como el correo que envía.

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 como lo haría con cualquier nota o informe escrito.

Resumen

- Sepa lo que quiere decir.
- Conozca a su audiencia.
- Elige tu momento.
- Elige un estilo.
- Haz que se vea bien.
- Involucra a tu audiencia.
- Sea un oyente.
- Vuelve a la gente.

Las secciones relacionadas incluyen:

- *Prototipos y Post-it Notes*, página 53
- *Equipos pragmáticos*, página 224

Desafíos

- Hay varios buenos libros que contienen secciones sobre comunicaciones dentro de los equipos de desarrollo [Bro95, McC95, DL99]. Asegúrese de tratar de leer los tres durante los próximos 18 meses. Además, el libro *Cerebros de dinosaurio* [Ber96] analiza el equipaje emocional que todos traemos al entorno laboral.
- La próxima vez que tenga que hacer una presentación, o escribir un memorándum defendiendo alguna posición, trate de trabajar con el SABIDURÍAcróstico en la página 20 antes de empezar. A ver si te ayuda a entender cómo posicionar lo que dices. Si corresponde, hable con su audiencia después y vea qué tan precisa fue su evaluación de sus necesidades.

Esta página se dejó en blanco intencionalmente

Capítulo 2

Un enfoque pragmático

Hay ciertos consejos y trucos que se aplican en todos los niveles de desarrollo de software, ideas que son casi axiomáticas y procesos que son prácticamente universales. Sin embargo, estos enfoques rara vez se documentan como tales; en su mayoría, los encontrará escritos como oraciones extrañas en discusiones sobre diseño, gestión de proyectos o codificación.

En este capítulo reuniremos estas ideas y procesos. Las dos primeras secciones, *Los males de la duplicación y ortogonalidad*, están estrechamente relacionados. El primero le advierte que no duplique el conocimiento en todos sus sistemas, el segundo que no divida ningún 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 sus proyectos de su entorno cambiante.

Las siguientes dos secciones también están relacionadas. En *Balas trazadoras*, 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 las balas trazadoras no siempre son aplicables. cuando no lo son, *Prototipos y Post-it Notes* muestra cómo usar la creación de prototipos para probar arquitecturas, algoritmos, interfaces e ideas.

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

Finalmente, todos trabajamos en un mundo de tiempo y recursos limitados. Puede sobrevivir mejor a estas dos carencias (y mantener a sus jefes más felices) si se le da bien calcular cuánto tiempo tomarán las cosas, que cubrimos en *Estimación*.

Si tiene en cuenta estos principios fundamentales durante el desarrollo, puede escribir un código mejor, más rápido y más sólido. Incluso puedes hacer que parezca fácil.

7

Los males de la duplicación

Darle a una computadora dos piezas de conocimiento contradictorias era la forma preferida del Capitán James T. Kirk de desactivar una inteligencia artificial merodeadora. Desafortunadamente, el mismo principio puede ser efectivo para derribar *sucódigo*.

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 alguna lógica comercial se vuelve obsoleta. Las pruebas pueden mostrar que el algoritmo elegido no funcionará. Toda esta inestabilidad hace que pasemos gran parte de nuestro tiempo en modo mantenimiento, reorganizando y reexpresando el conocimiento en nuestros sistemas.

La mayoría de la gente asume 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. Nuestro entendimiento cambia día a día. Llegan nuevos requisitos 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 integradas en la aplicación. El problema es que es fácil duplicar el conocimiento en las especificaciones, los procesos y los programas que desarrollamos, y cuando lo hacemos, invitamos a una pesadilla de mantenimiento, que comienza mucho antes de que se envíe la aplicación.

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

MUY CONOCIMIENTO DEBE TENER UN SOLO, UNAMBIGUOS, REPRESENTACIÓN AUTORIZADA DENTRO DE UN SISTEMA.

¿Por qué lo llamamos *SECO*?

TIP11

SECO—Dno Repetir Ynosotros mismos

La alternativa es tener la misma cosa expresada en dos o más lugares. Si cambias uno, tienes que acordarte de cambiar los otros, o, como las computadoras extraterrestres, tu programa caerá de rodillas por una contradicción. No se trata de si recordarás: se trata de cuándo olvidarás.

Encontrarás el *SECO* principio que aparece 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 programador pragmático.

En esta sección, describiremos los problemas de la duplicación y sugeriremos estrategias generales para solucionarlos.

¿Cómo surge la duplicación?

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

- Duplicación impuesta. Los desarrolladores sienten que no tienen opción: el ambiente parece requerir duplicación.
- Duplicación inadvertida. Los desarrolladores no se dan cuenta de que son duplicación de 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 estos cuatro/s de duplicación con más detalle.

Duplicación impuesta

A veces, la duplicación parece ser forzada en nosotros. Los estándares del proyecto pueden requerir documentos que contengan información duplicada o documentos que dupliquen información en el código. Varias plataformas de destino requieren 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 sentimos impotentes para evitar la duplicación. Y, sin embargo, a menudo hay formas de mantener cada pieza de conocimiento en un solo lugar, honrando el *SECOy* al mismo tiempo facilitarnos la vida. Aquí hay algunas técnicas:

Múltiples representaciones de la información. A nivel de codificación, a menudo necesitamos tener la misma información representada en diferentes formas. Tal vez estamos escribiendo una aplicación cliente-servidor, usando diferentes lenguajes en el cliente y el servidor, y necesitamos representar alguna estructura compartida en ambos. Quizás necesitemos una clase cuyos atributos reflejen el esquema de una tabla de base de datos. Tal vez esté escribiendo un libro y quiera incluir extractos de programas que también compilará y probará.

Con un poco de ingenio, normalmente 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 usando 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 desde el esquema de la base de datos en línea o desde 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 es hacer que el proceso esté activo: esto no puede ser una conversión de una sola vez, o estamos de vuelta 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 *por qué* el código necesita comentarios: código incorrecto *requiere* muchos comentarios

los *SECOE* El principio nos dice que mantengamos el conocimiento de bajo nivel en el código, donde pertenece, y reservemos los comentarios para otras explicaciones de alto nivel. De lo contrario, estamos duplicando conocimiento, y cada cambio significa cambiar tanto el código como los comentarios. Los comentarios inevitablemente quedarán desactualizados, y los comentarios poco confiables son peores que ningún comentario. (*Vertodo es escritura*, página 248, para obtener más información sobre los comentarios).

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

Dave trabajó una vez en un conmutador 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 garantizar que las pruebas reflejaran con precisión la especificación, el equipo las generó mediante programación a partir del propio documento. Cuando el cliente modificó su especificación, el conjunto de pruebas cambió automáticamente. Una vez que el equipo convenció al cliente de que el procedimiento era sólido, la generación de pruebas de aceptación generalmente tomó solo unos segundos.

Cuestiones de idioma. 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 escriben información de variables, funciones y (para C++) clases 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. Mientras que algunos entornos de desarrollo ocultan la necesidad de archivos de encabezado al generarlos automáticamente, y Object Pascal le permite abbreviar declaraciones de funciones repetidas, generalmente se queda atascado con

lo que te dan Al menos con la mayoría de los problemas relacionados con el idioma, un archivo de encabezado que no esté de acuerdo con la implementación generará algún tipo de error de compilación o vinculación. Todavía puede equivocarse, pero al menos se lo informarán bastante pronto.

Piense también en los comentarios en los archivos de encabezado e implementación. No tiene absolutamente ningún sentido duplicar un comentario de encabezado de función o clase entre los dos archivos. Utilice 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 distribución. Digamos que nuestro análisis revela que, entre otros atributos, un camión tiene un tipo, un número de licencia y un conductor. De manera similar, una ruta de entrega es una combinación de una ruta, un camión y un conductor. Codificamos algunas clases basándonos en este entendimiento.

Pero, ¿qué sucede cuando Sally se reporta enferma y tenemos que cambiar de conductor? Ambas cosasCamiónyRuta de entregacontener un controlador. ¿Cuál cambiamos? Claramente 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 subyacente? ¿Una ruta? O tal vez debe haber un tercer objeto que une a un conductor, un camión y una ruta. Cualquiera que sea la solución final, evite este tipo de datos no normalizados.

Hay un tipo de datos no normalizados un poco menos obvio que ocurre cuando tenemos múltiples elementos de datos que son mutuamente dependientes.

Veamos una clase que representa una línea:

```
claseLínea {
    público:
        Punto comienzo;
        Punto final;
        doble longitud;
};
```

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

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

```
claseLínea {
    público:
        Punto comienzo;
        Punto final;
        doble longitud() {devolver inicio.distanciaHasta(fin); }
    };
}
```

Más adelante en el proceso de desarrollo, puede optar por violar el *SECO* principio por razones de rendimiento. Con frecuencia, esto ocurre cuando necesita almacenar datos en caché para evitar repetir operaciones costosas. El truco es 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.

```
claseLínea {
    privado:
        bool cambió;
        doble longitud;
        Punto comienzo;
        Punto final;

    público:
        vacío setStart(Punto p) { inicio = p; cambiado = verdadero; } vacío
        establecerFinal(Punto p) { final = p; cambiado = verdadero; }
        punto getStart(vacío) Punto {devolver comienzo; }
        obtenerFin(vacío) {devolver final; }

        doble obtenerLongitud() {
            si(cambió) {
                longitud = inicio.distanciaHasta(fin);
                cambió = falso;
            }
            devolver longitud;
        };
}
```

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

1. El uso de funciones de acceso se relaciona con Meyer's *Principio de acceso uniforme*[Mey97b], que establece que "Todos los servicios ofrecidos por un módulo deben estar disponibles a través de una notación uniforme, que no traicione si se implementan mediante almacenamiento o mediante computación".

Duplicación impaciente

Cada proyecto tiene presiones de tiempo, fuerzas que pueden llevar a lo mejor de nosotros a tomar atajos. ¿Necesitas una rutina similar a la que has escrito? Tendrá la tentación de copiar el original y hacer algunos cambios. ¿Necesita un valor para representar el número máximo de puntos? Si cambio el archivo de encabezado, se reconstruirá todo el proyecto. Tal vez debería usar un número literal aquí; y aquí; y aquí. ¿Necesita una clase como una en el tiempo de ejecución de Java? La fuente está disponible, así que ¿por qué no simplemente copiarla y hacer los cambios necesarios (a pesar de las disposiciones de la licencia)?

Si siente esta tentación, recuerde el trillado aforismo "los atajos provocan largas demoras". Es posible que ahore algunos segundos ahora, pero con la posibilidad de perder horas más tarde. Piense en los problemas que rodean el fiasco 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 al principio 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 completos de funciones pueden duplicarse inadvertidamente, y esa duplicación podría pasar desapercibida durante años, lo que generaría problemas de mantenimiento. Escuchamos de primera mano de un estado de EE. UU. cuyos sistemas informáticos gubernamentales fueron inspeccionados para el cumplimiento de Y2K. La auditoría arrojó más de 10,000 programas, cada uno con su propia versión de validación del número de Seguro Social.

A un alto nivel, aborde el problema con un diseño claro, un líder de proyecto técnico sólido (consulte la página 228 en *Equipos pragmáticos*), y una división de responsabilidades bien entendida dentro del diseño. Sin embargo, a nivel de módulo, el problema es más insidioso. La funcionalidad o los datos comúnmente necesarios que no caen en un área de responsabilidad obvia pueden implementarse muchas veces.

Creemos que la mejor manera de lidiar con esto es fomentar una comunicación activa y frecuente entre los desarrolladores. Establecer foros para discutir problemas comunes. (En proyectos anteriores, hemos configurado Usenet privado

grupos de noticias para permitir a los desarrolladores intercambiar ideas y hacer preguntas. Esto proporciona una forma no intrusiva de comunicación, incluso entre varios sitios, al tiempo que conserva un historial permanente de todo lo dicho). Designe a un miembro del equipo como bibliotecario del proyecto, cuyo trabajo es facilitar el intercambio de conocimientos. Tenga un lugar central en el árbol de fuentes donde se puedan depositar las secuencias de comandos y las rutinas de utilidad. Y asegúrese de leer el código fuente y la documentación de otras personas, ya sea de manera informal o durante las revisiones del código. No estás husmeando, estás aprendiendo de ellos. Y recuerda, el acceso es recíproco: no te engañes con otras personas que están examinando (*¿pateando?*) su código, tampoco.

TIP12

Que sea fácil de reutilizar

Lo que está tratando de hacer es fomentar un entorno en el que sea más fácil encontrar y reutilizar material existente que escribirlo usted mismo. *Si no es fácil, la gente no lo hará.* Y si no lo reutiliza, corre el riesgo de duplicar el conocimiento.

Las secciones relacionadas incluyen:

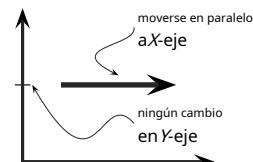
- *ortogonalidad*, página 34 *Manipulación de texto*,
- página 99 *Generadores de código*,
- página 102 *refactorización*, página 184
- *Equipos pragmáticos*, página 224
- *Automatización ubicua*, página 230 *todo*
- *es escritura*, página 248
-

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 varios otros métodos y técnicas que aprende. 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 geometría. Dos rectas son ortogonales si forman ángulos rectos, como los ejes de un gráfico. En términos vectoriales, las dos líneas son *independiente*. Muévase a lo largo de una de las líneas y su posición proyectada sobre la otra no cambia.



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 la base de datos e intercambiar bases de datos sin cambiar la interfaz.

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

Un sistema no ortogonal

Estás en un recorrido 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 colectivo² controla la elevación general, por lo que bajar-

2. Los helicópteros tienen cuatro controles básicos. Los *cílicos* es el palo que sostienes en tu mano derecha. Muévelo, y el helicóptero se mueve en la dirección correspondiente. Su mano izquierda sostiene el *palanca de paso colectivo*. Tire hacia arriba de esto y aumentará el paso en todas las palas, generando sustentación. Al final de la palanca de paso está el *acelerador*. Por fin tienes dos *piespedales*, que varían la cantidad de empuje del rotor de cola y ayudan a girar el helicóptero.

girándolo ligeramente comenzará un suave descenso hacia el suelo. Sin embargo, cuando lo pruebas, descubres que la vida no es tan sencilla. El morro del helicóptero cae y comienzas a descender en espiral hacia la izquierda. De repente, descubres que está pilotando un sistema en el que cada entrada de control tiene efectos secundarios. Baje la palanca de la izquierda y deberá agregar un movimiento hacia atrás de compensación a la palanca de la derecha y pisar el pedal derecho. Pero luego, cada uno de estos cambios afecta de nuevo a todos los demás controles. De repente, está haciendo malabarismos con un sistema increíblemente complejo, donde cada cambio afecta a todas las demás entradas. Tu carga de trabajo es fenomenal: tus manos y pies se mueven constantemente, tratando de equilibrar todas las fuerzas que interactúan.

Los controles de los helicópteros definitivamente 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.

TIP13

Eliminar efectos entre cosas no relacionadas

Queremos diseñar componentes que sean autónomos: independientes y con un solo propósito bien definido (lo que Yourdon y Constantine llaman *cohesión*[YC86]). Cuando los componentes están aislados unos de otros, sabe que puede cambiar uno sin tener que preocuparse por el resto. Mientras no cambie las interfaces externas de ese componente, puede estar seguro de que no causará problemas que afectarán a todo el sistema.

Obtiene dos beneficios principales si escribe sistemas ortogonales: mayor productividad y menor riesgo.

Gane productividad

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

Diseñado, codificado, probado en unidades 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 maneras que no fueron previstas por sus implementadores originales. Cuanto más débilmente estén acoplados sus sistemas, más fáciles serán de reconfigurar y rediseñar.
- Hay una ganancia bastante sutil en la productividad cuando combina componentes ortogonales. Suponga que un componente hace cosas distintas y otro hace cosas. Si son ortogonales y los combinás, el resultado no es $M \times N$ cosas. Sin embargo, si los dos componentes no son ortogonales, habrá superposición y el resultado será menor. Obtiene más funcionalidad por unidad de esfuerzo al combinar componentes ortogonales.

Reducir el riesgo

Un enfoque ortogonal reduce los riesgos inherentes a cualquier desarrollo.

- Las secciones de código enfermas están aisladas. 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 arreglos en un área en particular, y cualquier problema que genere se limitará a esa área.
- Probablemente se probará mejor un sistema ortogonal, porque será más fácil diseñar y ejecutar pruebas en sus componentes.
- No estará tan atado a un proveedor, producto o plataforma en particular, porque las interfaces con estos componentes de terceros estarán aisladas en partes más pequeñas del desarrollo general.

Veamos algunas de las formas en que puede aplicar el principio de ortogonalidad a su trabajo.

Equipos de proyecto

¿Ha notado cómo algunos equipos de proyecto son eficientes, todos saben qué hacer y contribuyen plenamente, mientras que los miembros de otros

¿Los equipos están discutiendo constantemente y no parecen poder apartarse unos de otros?

A menudo se trata de un problema de ortogonalidad. Cuando los equipos se organizan con mucha superposición, los miembros se confunden acerca de las responsabilidades. Todo cambio necesita una reunión de todo el equipo, porque cualquiera de ellos *puede querer* ser afectado.

¿Cómo organiza equipos en grupos con responsabilidades bien definidas y superposición mínima? 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 comenzar 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.

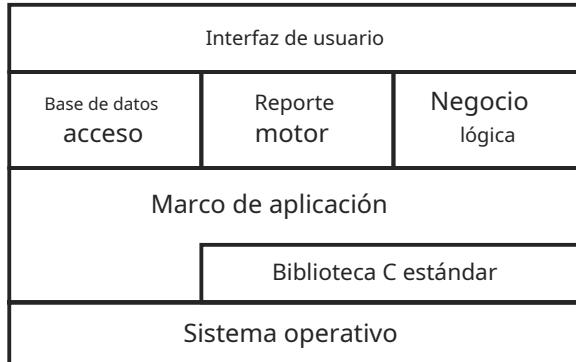
Puede obtener una medida informal de la ortogonalidad de la estructura de un equipo de proyecto. Simplemente vea cuántas personas *necesitar* participar en la discusión de cada cambio que se solicite. Cuanto mayor sea el número, menos ortogonal será el grupo. Claramente, un equipo ortogonal es más eficiente. (Habiendo dicho esto, también alentamos a los subequipos a comunicarse constantemente entre sí).

Diseño

La mayoría de los desarrolladores están familiarizados con la necesidad de diseñar sistemas ortogonales, aunque pueden usar palabras como *modular, basado en componentes, y en capas* para describir el proceso. Los sistemas deben estar compuestos por un conjunto de módulos cooperantes, 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 en capas es una forma poderosa de diseñar sistemas ortogonales. Debido a que cada capa usa solo las abstracciones proporcionadas por las capas debajo de ella, tiene una gran flexibilidad para cambiar las implementaciones subyacentes sin afectar el código. La estratificación también reduce el riesgo de dependencias fuera de control entre módulos. A menudo verá capas expresadas en diagramas como el de la Figura 2.1 en la página siguiente.

Hay una prueba fácil para el diseño ortogonal. Una vez que haya mapeado sus componentes, pregúntese: *Si cambio drásticamente el requerimiento*

Figura 2.1. Diagrama de capa típico



mentos detrás de una función en particular, ¿cuántos módulos se ven afectados? En un sistema ortogonal, la respuesta debe ser "uno".³ Mover un botón en un panel GUI no debería requerir un cambio en el esquema de la base de datos. Agregar ayuda sensible al contexto 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 de tonos de la planta. En un sistema diseñado ortogonalmente, necesitaría cambiar solo aquellos 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.*es solo una vista,* página 157, habla sobre la escritura de código desacoplado utilizando el paradigma Model-View-Controller (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 en el sistema. Sin embargo, si analiza el cambio en términos de funciones, lo ideal sería que cada cambio funcional afectara solo a un módulo.

También pregúntese qué tan desvinculado está su 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 las cosas que no puedes controlar.*

Juegos de herramientas y bibliotecas

Tenga cuidado de preservar la ortogonalidad de su sistema cuando presente juegos de herramientas y bibliotecas de terceros. Elija sabiamente sus tecnologías.

Una vez trabajamos en un proyecto que requería que cierto cuerpo de código Java se ejecutara localmente en una máquina servidor y 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 mediante RMI, cada llamada a un método remoto en esa clase podría generar una excepción, lo que significa que una implementación ingenua requeriría que manejemos la excepción cada vez que se usen nuestras clases remotas. El uso de RMI aquí claramente no es ortogonal: el código que llama a nuestras clases remotas no debería tener que conocer sus ubicaciones. La alternativa, usar CORBA, no imponía esa restricción: podíamos escribir código que desconocía las ubicaciones de nuestras clases.

Cuando traiga un conjunto de herramientas (o incluso una biblioteca de otros miembros de su equipo), pregúntese si impone cambios en su código que no deberían estar allí. Si un esquema de persistencia de objetos es transparente, entonces es ortogonal. Si requiere que cree o acceda a objetos de una manera especial, entonces no lo es. Mantener dichos detalles aislados de su código tiene el beneficio adicional de facilitar el cambio de proveedores 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 la aplicación tiene que delimitar el inicio y el final de cada transacción. Con EJB, esta información se expresa declarativamente como metadatos, fuera de cualquier código. El mismo código de aplicación puede ejecutarse en diferentes entornos de transacciones EJB sin cambios. Es probable que este sea un modelo para muchos entornos futuros.

Otro giro interesante en la ortogonalidad es la programación orientada a aspectos (AOP), un proyecto de investigación en 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, registrar mensajes

normalmente se generan rociando llamadas explícitas a alguna función de registro en toda su fuente. Con AOP, implementa el registro de forma ortogonal a las cosas que se registran. Usando la versión Java de AOP, puede escribir un mensaje de registro al ingresar cualquier método de clase Fred codificando el *aspecto*:

```
aspettoRastro {
    consejo*Fred.*(..) {
        estática antes{
            Registro.escribir("> Entrando"+thisJoinPoint.methodName);
        }
    }
}
```

Si usted *tejido* este aspecto en su código, se generarán mensajes de seguimiento. Si no lo hace, no verá 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 controle constantemente no solo lo que está haciendo sino también el contexto más amplio de la aplicación, es posible que involuntariamente duplique la funcionalidad en algún otro módulo o exprese el conocimiento existente dos veces.

Hay varias técnicas que puede utilizar para mantener la ortogonalidad:

- Mantenga su 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], que discutimos en *El desacoplamiento y la Ley de Deméter*, 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 su código hace referencia a datos globales, se vincula a sí mismo con los otros 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 multiproceso). En general, su código es más fácil de entender y mantener si pasa explícitamente cualquier contexto requerido a sus módulos. En 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 pasarles referencias.

El patrón Singleton en *Patrones de diseño*[GHJV95] es una forma de garantizar que solo haya una instancia de un objeto de una clase en particular. Mucha gente usa estos objetos singleton como una especie de variable global (particularmente en lenguajes, como Java, que de otro modo no son compatibles con el concepto de variables globales). Tenga cuidado con los singletons, también pueden conducir a enlaces innecesarios.

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

Adquiera el hábito de ser constantemente crítico con su código. Busque cualquier oportunidad de reorganizarlo para mejorar su estructura y ortogonalidad. Este proceso se llama *refactorización*, y es tan importante que le hemos dedicado una sección (ver *refactorización*, página 184).

Pruebas

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 de nivel de módulo (o unidad) son considerablemente más fáciles de especificar y realizar que las pruebas de integración. De hecho, sugerimos que cada módulo tenga su propia prueba unitaria integrada en su código, y que estas pruebas se realicen automáticamente como parte del proceso de construcción normal (ver *Código que es fácil de probar*, página 189).

La construcción de pruebas unitarias es en sí misma una interesante prueba de ortogonalidad. ¿Qué se necesita para construir 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 encuentre un problema, evalúe qué tan localizado

la solución es. ¿Cambia solo un módulo o los cambios están dispersos en todo el sistema? Cuando haces un cambio, ¿lo soluciona todo o surgen misteriosamente otros problemas? Esta es una buena oportunidad para aplicar la automatización. Si utiliza un sistema de control de código fuente (y lo hará después de leer *Control de código fuente*, página 86), etiquete las correcciones de errores cuando vuelva a ingresar el código después de la prueba. A continuación, puede ejecutar informes mensuales que analicen las tendencias en la cantidad de archivos de origen afectados por cada corrección de errores.

Documentación

Quizás sorprendentemente, la ortogonalidad también se aplica a la documentación. Los ejes son contenido y presentación. Con 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 (*ver todo es escritura*, página 248).

Vivir con la ortogonalidad

La ortogonalidad está estrechamente relacionada con la *SECO* principio introducido en la página 27. Con *SECO*, busca minimizar la duplicación dentro de un sistema, mientras que con la ortogonalidad reduce la interdependencia entre los componentes del sistema. Puede ser una palabra torpe, pero si usa el principio de ortogonalidad, combinado estrechamente con el *SECO* En principio, encontrará que los sistemas que desarrolla son más flexibles, más comprensibles y más fáciles de depurar, probar y mantener.

Si te involucran en un proyecto en el que las personas luchan desesperadamente por hacer cambios y en el que cada cambio parece causar que otras cuatro cosas salgan mal, recuerda la pesadilla del helicóptero. El proyecto probablemente 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 la duplicación*, página 26
- *Control de código fuente*, página 86
- *Diseño por contrato*, página 109
- *El desacoplamiento y la Ley de Deméter*, página 138

- *Metaprogramación*, página 144 es
- *solo una vista*, página 157
- *refactorización*, página 184
- *Código que es fácil de probar*, página 189
- *Magos malvados*, página 198 *Equipos*
- *pragmáticos*, página 224 *todo es escritura*,
- página 248

Desafíos

- Considere la diferencia entre las grandes herramientas orientadas a GUI que normalmente se encuentran en los sistemas Windows y las utilidades de línea de comando pequeñas pero combinables que se usan 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 enfrentar nuevos desafíos?
- C++ admite herencia múltiple y Java permite que una clase implemente múltiples interfaces. ¿Qué impacto tiene el uso de estas instalaciones en la ortogonalidad? ¿Hay alguna diferencia en el impacto entre usar herencia múltiple y múltiples interfaces? ¿Hay alguna diferencia entre usar delegación y usar herencia?

Ejercicios

1. Estás escribiendo una clase llamada `Separar`, que divide las líneas de entrada en campos. ¿Cuál de las siguientes dos firmas de clase Java es el diseño más ortogonal?

Responder
en P. 279

```
claseDividir1 {
    públicoSplit(InputStreamReader rdr) { ... vacío público leerLíneaSiguiente() lanza
    IOExcepción { ... public int numFields() { ...
    públicoCadena getField(En tcampoNo) { ...
}

claseDividir2 {
    públicoDividir2 (línea de cadena) { ... public int
    numFields() { ...
    públicoCadena getField(En tcampoNo) { ...
}
```

2. ¿Qué conducirá a un diseño más ortogonal: cuadros de diálogo modales o no modales?

Responder
en P. 279

3. ¿Qué hay de los lenguajes procedimentales frente a la tecnología de objetos? ¿Cuál da como resultado un sistema más ortogonal?

Responder
en P. 280

Reversibilidad

Nada es más peligroso que una idea si es la única que tienes.

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

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

¡Si tan solo el mundo real cooperara! Desafortunadamente, si bien es ~~2~~ hoy, es posible que deba ~~sér~~ mañana y la ~~próx~~ima semana. Nada es para siempre, y si confía mucho en algún hecho, casi puede garantizar que *voluntad* cambio.

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 se lleve una sorpresa desagradable. Muchos equipos de proyecto tienen los ojos abiertos a la fuerza a medida que se desarrolla el futuro:

“Pero dijiste que usaríamos la base de datos XYZ! Ya hemos terminado de codificar el proyecto en un 85 %, ¡no podemos cambiarlo ahora!”. protestó el programador. “Lo sentimos, pero nuestra empresa decidió estandarizar la base de datos PDQ en su lugar, para todos los proyectos. Está fuera de mis manos. Solo tendremos que recodificar. Todos ustedes trabajarán los fines de semana hasta nuevo aviso”.

Los cambios no tienen que ser tan draconianos, ni siquiera tan inmediatos. Pero a medida que pasa el tiempo y su proyecto avanza, 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 limitada de la realidad que tiene menos opciones.

Cuando se han tomado muchas decisiones críticas, el objetivo se vuelve tan pequeño que si se mueve, si el viento cambia de dirección o si una mariposa en Tokio agita sus alas, fallas.⁴ Y puede fallar por una gran cantidad.

4. Tome un sistema no lineal o caótico y aplique un pequeño cambio a una de sus entradas. Puede obtener un resultado grande y, a menudo, impredecible. El cliché de la mariposa batiendo sus alas en Tokio podría ser el comienzo de una cadena de eventos que termine generando un tornado en Texas. ¿Se parece a algún proyecto que conozcas?

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

Una vez que decida usar la base de datos de este proveedor, o ese patrón arquitectónico, o un determinado modelo de implementación (cliente-servidor versus independiente, por ejemplo), está comprometido con 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 producir software flexible y adaptable. Al apegarse a sus recomendaciones, especialmente las *SECO*principio (página 26), desacoplamiento (página 138) y 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 cierta tecnología solo para descubrir que no podemos contratar suficientes personas con las habilidades necesarias. Aseguramos a cierto proveedor externo justo antes de que su competidor los compre. Los requisitos, los usuarios y el hardware cambian más rápido de lo que podemos desarrollar el software.

Suponga 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 tendría suerte. La mayoría de las veces, las llamadas a productos de terceros se enredan en todo el código. pero si tu *De Verdad*abstraigo 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.

De manera similar, suponga que el proyecto comienza como un modelo cliente-servidor, pero luego, al final del juego, marketing decide que los servidores son demasiado caros para algunos clientes y quieren una versión independiente. ¿Qué tan difícil sería para ti? Dado que es solo un problema de implementación, *no debería tomar más de unos pocos días*. Si tomaría más tiempo, entonces no ha pensado en la reversibilidad. La otra dirección es aún más interesante. ¿Qué sucede si el producto independiente que está creando debe implementarse en un cliente-servidor *on the go*-moda de nivel? *Eso tampoco debería ser difícil.*

El error está en asumir que cualquier decisión es inamovible 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 venir y acabar con ellos en cualquier momento.

TIP14

No hay decisiones finales

Arquitectura flexible

Mientras que muchas personas tratan de mantener su código Florida exigible, 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 o la plataforma de desarrollo. ¿El rendimiento de Java en esa plataforma no está a la altura de las expectativas? Vuelva a codificar el cliente en C++, y nada más necesita cambiar. ¿El motor de reglas en C++ no es lo suficientemente flexible? Cambia a una versión de Smalltalk. Con una arquitectura CORBA, debe recibir un golpe solo para el componente que está reemplazando; los otros componentes no deberían verse afectados.

¿Estás desarrollando para Unix? ¿Cuál? ¿Tiene todas las preocupaciones de portabilidad abordadas? ¿Está desarrollando para una versión particular de Windows? ¿Cuál: 3.1, 95, 98, NT, CE o 2000? ¿Qué tan difícil será admitir otras versiones? Si mantiene las decisiones suaves y flexibles, no será difícil en absoluto. Si tiene una encapsulación deficiente, un alto acoplamiento y una lógica o parámetros codificados en el código, podría ser imposible.

¿No está seguro de cómo marketing quiere implementar el sistema? Piénselo desde el principio y podrá admitir un cliente-servidor independiente o *norte*-modelo de nivel simplemente cambiando un archivo de configuración. Hemos escrito programas que hacen precisamente eso.

Normalmente, simplemente puede ocultar un producto de terceros detrás de una interfaz abstracta bien definida. De hecho, siempre hemos podido hacerlo en cualquier proyecto en el que hemos trabajado. Pero suponga que no pudiera aislarlo tan limpiamente. ¿Qué pasaría si tuviera que esparcir ciertas declaraciones generosamente a lo largo del código? Ponga ese requisito en los metadatos y use algún mecanismo automático, como Aspects (consulte la página 39) o Perl, para insertar las declaraciones necesarias en el código mismo. Sea cual sea el mecanismo que Ud.

usar, *hacerlo reversible*. Si algo se agrega automáticamente, también se puede quitar automáticamente.

¡Nadie sabe lo que puede deparar el futuro, especialmente nosotros! Así que habilite su código para rock-n-roll: para "rockear" cuando pueda, para rodar con los golpes cuando deba hacerlo.

Las secciones relacionadas incluyen:

- *El desacoplamiento y la Ley de Deméter*, página 138
- *Metaprogramación*, página 144 *es solo una vista*,
- página 157

Desafíos

- Hora de un poco de mecánica cuántica con el gato de Schrödinger. Suponga que tiene 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 morirá. Si no es así, el gato estará bien. Entonces, ¿el gato está vivo o muerto? Según Schrödinger, la respuesta correcta *es ambas cosas*. Cada vez que se produce una reacción subnuclear que tiene dos posibles resultados, se clona el universo. En uno ocurrió el evento, en el otro no. El gato está vivo en un universo, muerto en otro. Solo cuando abres la caja sabes qué universo tú estás adentro

No es de extrañar que la codificación 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 da como resultado una versión diferente del futuro. ¿Cuántos futuros posibles puede soportar su código? ¿Cuáles son más probables? ¿Qué tan difícil será apoyarlos cuando llegue el momento?

¿Te atreves a abrir la caja?

10

Balas trazadoras

Listo, fuego, apunta ...

Hay dos formas de disparar una ametralladora en la oscuridad.⁵ Puede averiguar exactamente dónde está su objetivo (alcance, elevación y azimut). 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. Luego puede usar tablas o una computadora de disparo para calcular el rumbo y la elevación exactos del cañón. Si todo funciona exactamente como se especifica, sus tablas son correctas y el entorno no cambia, sus balas deberían caer cerca de su objetivo.

O podrías usar balas trazadoras.

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

No es sorprendente que se prefieran las balas trazadoras a la labor 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 proyectos nuevos, especialmente cuando se está construyendo algo que no se ha construido antes. Al igual que los artilleros, estás tratando de alcanzar un objetivo en la oscuridad. Debido a que sus usuarios nunca antes han 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 incógnitas. 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 saciedad. Producir resmas de papel detallando cada requerimiento, atando cada incógnita y

5. Para ser pedante, hay muchas formas de disparar una ametralladora en la oscuridad, incluso cerrar los ojos y disparar balas. Pero esto es una analogía, y se nos permite tomarnos libertades.

restringiendo el entorno. Dispara el arma usando la navegación a estima. Un gran cálculo por adelantado, luego dispara y espera.

Los programadores pragmáticos, sin embargo, tienden a preferir usar viñetas trazadoras.

Código que brilla en la oscuridad

Las balas trazadoras funcionan porque operan en el mismo entorno y bajo las mismas restricciones que las balas reales. Llegan al objetivo rápidamente, 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, buscamos algo que nos lleve de un requisito a algún aspecto del sistema final de forma rápida, visible y repetible.

TIP 15

Usa balas trazadoras para encontrar el objetivo

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

Esta fue una gran oportunidad para usar el código de seguimiento. Desarrollamos el marco para el front-end, bibliotecas para representar las consultas y una estructura para convertir una consulta almacenada en una consulta específica de la base de datos. Luego lo montamos 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 hablar con las bibliotecas, las bibliotecas podían serializar y deserializar una consulta, y el servidor podía generar SQL desde el resultado. Durante los meses siguientes, desarrollamos gradualmente esta estructura básica, agregando nuevas funciones 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 rastreador 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 está del objetivo, ajustándolo si es necesario. Una vez que esté en el objetivo, agregar funcionalidad es fácil.

El desarrollo de Tracer es consistente con la idea de que un proyecto nunca está terminado: siempre habrá cambios necesarios y funciones para 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 subensamblajes, que luego se combinan aún más, hasta que un día tenga una aplicación completa. Solo entonces se puede presentar la aplicación como un todo al usuario y probarla.

El enfoque del código trazador tiene muchas ventajas:

- **Los usuarios pueden ver algo funcionando temprano.** Si ha comunicado con éxito lo que está haciendo (ver *Grandes expectativas*, página 255), sus usuarios sabrán que están viendo algo inmaduro. No se sentirán decepcionados por la falta de funcionalidad; estarán encantados de ver algún progreso visible hacia su sistema. También pueden contribuir a medida que avanza el proyecto, aumentando su participación. Estos mismos usuarios probablemente serán 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. Si ha resuelto todas las interacciones de extremo a extremo de su aplicación y las ha incorporado en el código, entonces su equipo no necesitará sacar tanto de la nada. Esto hace que todos sean más productivos y fomenta la consistencia.
- **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 hayan probado por unidad. En lugar de intentar una integración a lo grande, 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 inconvenientes. Con el código rastreador, siempre tendrás algo que mostrarles.
- **Tienes una mejor idea del progreso.** En un desarrollo de código rastreador, los desarrolladores abordan los casos de uso uno por uno. Cuando uno termina, pasan al siguiente. Es mucho más fácil medir el rendimiento y demostrar el progreso a su usuario. Debido a que cada desarrollo individual es más pequeño, evita crear esos bloques de código monolíticos que se informan como completos en un 95 % semana tras semana.

Las balas trazadoras no siempre dan en el blanco

Las balas trazadoras muestran lo que estás golpeando. Este puede no ser siempre el objetivo. Luego ajusta su puntería hasta que estén en el objetivo. Ese es el punto.

Es lo mismo con el código rastreador. Utiliza la técnica en situaciones en las que no está 100% seguro de hacia dónde se dirige. 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 parece probable que haya problemas de rendimiento. Averigüe cómo cambiar lo que tiene para acercarlo al objetivo y agradezca haber utilizado una metodología de desarrollo esbelto. Un pequeño cuerpo de código tiene baja inercia, es fácil y rápido de cambiar. Podrá recopilar comentarios sobre su aplicación y generar una versión nueva y más precisa más rápido y a menor costo que con cualquier otro método. Y debido a que todos los componentes principales de la aplicación están representados en su código de rastreo, sus usuarios pueden estar seguros de que lo que están viendo se basa en la realidad, no solo en una especificación en papel.

Código trazador frente a creación de prototipos

Podrías pensar que este concepto de código rastreador no es más que un prototipo bajo un nombre agresivo. Hay una diferencia. Con un prototipo, su objetivo es explorar aspectos específicos del sistema final. Con un verdadero prototipo, desecharás todo lo que hayas juntado al probar el concepto y lo recodificarás correctamente usando las lecciones que has aprendido.

Por ejemplo, supongamos que está produciendo una aplicación que ayuda a los transportistas a determinar cómo empacar cajas de tamaños extraños en contenedores. Entre otros

problemas, la interfaz de usuario debe ser intuitiva y los algoritmos que utiliza para determinar el empaque óptimo son muy complejos.

Podría crear un prototipo de una interfaz de usuario para sus usuarios finales en una herramienta GUI. Solo codifica lo suficiente para que la interfaz responda a las acciones del usuario. Una vez que hayan aceptado el diseño, puede desecharlo y recodificarlo, esta vez con la lógica comercial detrás de él, utilizando el idioma de destino. De manera similar, es posible que desee crear un prototipo de una serie de algoritmos que realicen el empaque 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 haya tomado su decisión, comenzaría de nuevo y codificaría los algoritmos en su entorno final, interactuando con el mundo real. Esto es *creación de prototipos*, y es muy útil.

El enfoque del código rastreador aborda un problema diferente. Necesita saber cómo funciona la aplicación como un todo. Desea mostrar a sus usuarios cómo funcionarán las interacciones en la práctica y desea brindarles a sus desarrolladores un esqueleto arquitectónico en el que colgar el código. En este caso, puede construir un rastreador que consista en una implementación trivial del algoritmo de empaquetado de contenedores (tal vez algo así como primero en llegar, primero en ser atendido) y una interfaz de usuario simple pero funcional. Una vez que haya conectado todos los componentes de la aplicación, tendrá un marco para mostrar a sus usuarios y desarrolladores. Con el tiempo, agrega nuevas funcionalidades a este marco, completando rutinas stub. Pero el marco permanece intacto y usted sabe que el sistema seguirá comportándose como 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 simple 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:

- *Software suficientemente bueno*, página 9
- *Prototipos y Post-it Notes*, página 53 *La trampa*
- *de las especificaciones*, página 217 *Grandes expectativas*, página 255

11

Prototipos y Post-it Notes

Muchas industrias diferentes usan prototipos para probar ideas específicas; La creación de prototipos 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 van un paso más allá y ahora realizan una gran cantidad de trabajo de modelado en la computadora, lo que reduce aún más los costos. De esta forma, se pueden probar elementos arriesgados o inciertos sin comprometerse a construir el elemento real.

Construimos prototipos de software de la misma manera y por las mismas razones: para analizar y exponer el riesgo, y para ofrecer posibilidades 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 que ser así. Al igual que los fabricantes de automóviles, podemos construir prototipos con diferentes materiales. Las notas post-it son excelentes para crear prototipos de cosas dinámicas, como el flujo de trabajo y la lógica de la aplicación. Se puede crear un prototipo de una interfaz de usuario como un dibujo en una pizarra, como una maqueta no funcional dibujada con un programa de pintura o con un generador de interfaz.

Los prototipos están diseñados para responder solo unas pocas preguntas, por lo que son mucho más baratos y rápidos de desarrollar que las aplicaciones que entran en producción. El código puede ignorar detalles sin importancia, sin importancia para usted en este momento, pero probablemente muy importantes para el usuario más adelante. Si está creando un prototipo de una GUI, por ejemplo, puede salirse con la suya con resultados o datos incorrectos. Por otro lado, si solo está investigando aspectos computacionales o de rendimiento, puede salirse con la suya con una GUI bastante pobre, o tal vez incluso sin GUI.

Pero si te encuentras en un entorno en el que no puedes renunciar a los detalles, entonces debe preguntarse si realmente está construyendo un prototipo. Tal vez un estilo de desarrollo de bala trazadora sería más apropiado en este caso (ver *Balas trazadoras*, página 48).

Cosas para Prototipar

¿Qué tipo de cosas podría elegir investigar con un prototipo? Cualquier cosa que conlleve riesgo. Cualquier cosa que no se haya probado 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 hacer un prototipo

- 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 radica en el código producido, sino en las lecciones aprendidas. Ese es realmente el objetivo de la creación de prototipos.



Cómo usar prototipos

Al construir un prototipo, ¿qué detalles puedes ignorar?

- **Exactitud.** Es posible que pueda utilizar datos ficticios cuando corresponda.
- **Lo completo.** El prototipo puede funcionar solo en un sentido muy limitado, quizás con solo una pieza preseleccionada de datos de entrada y un elemento de menú.
- **Robustez.** Es probable que la verificación 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. Esta bien.
- **Estilo.** Es doloroso admitir esto en forma impresa, pero el código prototipo probablemente no tenga muchos comentarios o documentación. Puede producir una gran cantidad 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, más alto que el resto del proyecto (tal vez un lenguaje como Perl, Python , o Tcl). Un lenguaje de secuencias de comandos de alto nivel le permite aplazar muchos detalles (incluida la especificación de tipos de datos) y aún producir una pieza de código funcional (aunque incompleta o lenta).⁶ Si necesita crear prototipos de interfaces de usuario, investigue herramientas como Tcl/Tk, Visual Basic, Powerbuilder o Delphi.

Los lenguajes de secuencias de comandos funcionan bien como el "pegamento" para combinar piezas de bajo nivel en nuevas combinaciones. Bajo Windows, Visual Basic puede unir controles COM. En términos más generales, 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 prototipos

Muchos prototipos se construyen 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 necesite codificar para crear prototipos de arquitectura: puede crear prototipos en una pizarra, con notas Post-it o fichas. Lo que está buscando es cómo el sistema se mantiene unido como un todo, nuevamente aplazando los detalles. Aquí hay algunas áreas específicas que puede querer buscar en el prototipo arquitectónico:

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

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

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

Este último elemento tiende a generar la mayor cantidad de sorpresas y los resultados más valiosos de la experiencia de creación de prototipos.

Cómo *Nous*ar 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. debes hacerlo *muy* claro que este código es descartable, incompleto y no se puede completar.

Es fácil dejarse engañar por la aparente integridad de un prototipo demostrado, y los patrocinadores o la gerencia del proyecto pueden insistir en implementar el prototipo (o su descendencia) si no establece las expectativas correctas.

Recuérdale que puede construir un gran prototipo de un automóvil nuevo con madera de balsa y cinta adhesiva, pero no trataría de conducirlo en el tráfico de la hora pico!

Si cree que existe una gran posibilidad en su entorno o cultura de que el propósito del código prototipo se malinterprete, es posible que le vaya mejor con el enfoque de viñeta trazadora. Terminará con un marco sólido en el que basar el desarrollo futuro.

Cuando se usa correctamente, un prototipo puede ahorrarle una gran cantidad de tiempo, dinero, dolor y sufrimiento al identificar y corregir posibles puntos problemáticos al principio del ciclo de desarrollo, el momento en que corregir errores es barato y fácil.

Las secciones relacionadas incluyen:

- *El gato se comió mi código fuente*, página 2
- *¡Comunicar!*, página 18 *Balas trazadoras*,
- página 48 *Grandes expectativas*, página 255
-

Ejercicios

4. A 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 decidir sobre un modelo para la imagen, tal vez

es un coche, o un teléfono, o una casa. Tiene una lista de páginas de destino y contenido; les gustaría ver algunos prototipos. Oh, por cierto, tienes 15 minutos. ¿Qué herramientas podría usar?

12

Idiomas de dominio

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

► Luis Wittgenstein

Influencia de los lenguajes informáticos cómo piensas acerca de un problema, y cómo piensas acerca de la comunicación. Cada idioma viene con una lista de características (palabras de moda, como tipeo estático versus dinámico, vinculación temprana versus tardía, modelos de herencia (único, múltiple o ninguno), todo lo cual puede sugerir u ocultar ciertas soluciones. Diseñar una solución con Lisp en mente producirá resultados diferentes a los de una solución basada en el pensamiento de estilo C, y viceversa. Por el contrario, y creemos que es más importante, el lenguaje del dominio del problema también puede sugerir una solución de programación.

Siempre tratamos de escribir código usando el vocabulario del dominio de la aplicación (ver *El pozo de requisitos*, página 210, donde sugerimos utilizar un glosario del proyecto). En algunos casos, podemos pasar al siguiente nivel y realmente programar usando el vocabulario, la sintaxis y la semántica, el lenguaje del dominio.

Cuando escucha 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 satelital y guárde las para futuros análisis.

Si sus usuarios tienen una serie de afirmaciones bien delimitadas, puede inventar un minilenguaje adaptado al dominio de la aplicación que exprese exactamente lo que quieren:

```
Desde X25LINE1 (Formato=ABC123) {  
    Ponga TELSTAR1 (Formato=XYZ43B); Almacenar  
    base de datos;  
}
```

Este lenguaje no necesita ser ejecutable. Inicialmente, podría ser simplemente una forma de capturar los requisitos del usuario: una especificación. Sin embargo, es posible que desee considerar llevar esto un paso más allá e implementar el lenguaje. Su especificación se ha convertido en código ejecutable.

Después de que haya escrito la aplicación, los usuarios le dan un nuevo requisito: las transacciones con saldos negativos no deben almacenarse y deben devolverse en las líneas X.25 en el formato original:

```
Desde X25LINE1 (Formato=ABC123) {
    si (ABC123.saldo < 0) {
        Poner X25LINE1      (Formato=ABC123);
    }
    más {
        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 sus usuarios finales realmente programen en estos lenguajes. En cambio, se está dando a sí mismo una herramienta que le permite trabajar más cerca de su dominio.

TIP17

Programa cerca del 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 a un nivel más alto de abstracción, puede concentrarse en resolver problemas de dominio y puede ignorar los pequeños detalles de implementación.

Recuerda que hay muchos usuarios de una aplicación. Está el usuario final, que comprende las reglas comerciales y los resultados requeridos. También hay usuarios secundarios: personal de operaciones, administradores 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 los problemas en términos que sus usuarios puedan entender. Tome nuestra aplicación de cambio en la página opuesta. Supongamos que el usuario escribió mal el nombre del formato:

Desde X25LINE1 (Formato=AB123)

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

Error de sintaxis: identificador no declarado

Pero con un mini-idioma, en cambio, 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.

Implementando un Mini-Lenguaje

En su forma más simple, un mini-lenguaje puede estar en un formato orientado a líneas y fácilmente analizable. En la práctica, probablemente usamos este formulario más que cualquier otro. Se puede analizar simplemente usando cambiar declaraciones, o el uso de expresiones regulares en lenguajes de secuencias de comandos 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 analizador. Los programadores de C y C++ han estado usando yacc (o su implementación libremente disponible, bison) [URL 27]) durante años. Estos programas están documentados en detalle en el libro *Lex y 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 *sin contexto* gramáticas recursivamente. Cualquier buen libro sobre construcción o análisis de compiladores cubrirá BNF en detalle (exhaustivo).

muestra un analizador escrito usando bison. Como muestra, una vez que conoce la sintaxis, realmente no es mucho trabajo escribir mini-lenguajes 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) si
registro.saldo < 0:
    X25LINE1.put(registro,           formato=ABC123)
más:
    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.

Idiomas de datos producir alguna forma de estructura de datos utilizada por una aplicación. Estos lenguajes se utilizan a menudo para representar información de configuración.

por ejemplo, el enviar correoEl programa 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 usando enviar correo'Lenguaje de configuración propio:

```
mlocal,      P=/usr/bin/procmail,
              F=lsDFMAw5:/|@qSPfhn9,
              S=10/30, R=20/40,
              T=DNS/RFC822/X-Unix,
              A=procmail -Y -a $h -d $u
```

Obviamente, la legibilidad no es una de enviar correo'fortalezas de s.

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

Idiomas imperativos lleva 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 Windows

```

MENÚ PRINCIPAL_MENU
{
    EMERGENTE "&Archivo"
    {
        OPCIÓN DEL MENÚ "&Nuevo", CM_FILENEUEVO
        OPCIÓN DEL MENÚ "&Abierto...", CM_FILEOPEN
        OPCIÓN DEL MENÚ "&Guardar", CM_FILES救人
    }
}

MY_DIALOG_BOX DIALOG 6, 15, 292, 287 ESTILO DS_MODALFRAME |
WS_POPUP | WS_VISIBLE |
WS_CAPTION | WS_SYSMENU
TÍTULO "Mi cuadro de diálogo"
FUENTE 8, "MS Sans Serif" {

    DEFBUTTON "OK", ID_OK, 232, 16, 50, 14 PULSADOR "Ayuda",
    ID_HELP, 232, 52, 50, 14 CONTROL "Editar control de texto",
    ID_EDIT1,
        "EDITAR", WS_BORDER | WS_TABSTOP, 16, 16, 80, 56 CHECKBOX "Casilla
        de verificación", ID_CHECKBOX1, 153, 65, 42, 38,
        BS_AUTOCHECKBOX | WS_TABSTOP
}

```

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

```

localizar mensaje "SSN:"
escribe "%s" social_security_number ingresar
escribe
esperar desbloqueo del teclado
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, puede comprar herramientas que admitan este tipo de secuencias de comandos. También puede investigar paquetes de código abierto como Expect, que brindan 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 del mainframe cambia y los campos se mueven, el programador puede simplemente actualizar su descripción de alto nivel, en lugar de arrastrarse por los detalles del código C.

Idiomas autónomos e integrados

Un minilenguaje no tiene que ser utilizado directamente por la aplicación para ser útil. Muchas veces podemos usar un lenguaje de especificación para crear artefactos (incluidos los metadatos) que el propio programa compila, lee o utiliza de otro modo (ver *Metaprogramación*, página 144).

Por ejemplo, en la página 100 describimos un sistema en el que usamos Perl para generar una gran cantidad 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 otras. La aplicación no usó la especificación directamente, pero se basó en el resultado producido a partir de ella.

Es común incrustar lenguajes imperativos de alto nivel directamente en su aplicación, para que se ejecuten cuando se ejecuta su 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 sencillos orientados a líneas hasta gramáticas más complejas que parecen lenguajes reales. Dado que la implementación requiere un esfuerzo adicional, ¿por qué elegiría una gramática más compleja?

La compensación 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 se ampliará en el futuro con nuevas características y funciones. Los lenguajes que son demasiado simples pueden ser fáciles de analizar, pero pueden ser críticos, al igual que el enviar correo ejemplo en la página 60.

Dado que la mayoría de las aplicaciones exceden su vida útil esperada, es probable que 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 en la reducción de los costos de soporte y mantenimiento.

Las secciones relacionadas incluyen:

- *Metaprogramación*, página 144

Desafíos

- ¿Podrían expresarse algunos de los requisitos de su proyecto actual en un lenguaje específico de dominio? ¿Sería posible escribir un compilador o traductor que pudiera generar la mayor parte del código requerido?
- Si decide adoptar mini-lenguajes como una forma de programación más cercana al dominio del problema, está aceptando que se requerirá algún esfuerzo para implementarlos. ¿Puedes ver formas en las que el marco que desarrollas para un proyecto puede reutilizarse en otros?

Ejercicios

5. Queremos implementar un mini-lenguaje para controlar un paquete de dibujo simple (quizás un sistema de gráficos de tortugas). El lenguaje consta de comandos de una sola letra. Algunos comandos van seguidos de un solo número. Por ejemplo, la siguiente entrada dibujaría un rectángulo.

*Responder
en P. 281*

```

PAQS # Seleccione lápiz 2
D   # lápiz abajo
W 2 # dibujar Oeste 2cm
norte 1 # después norte 1
mi 2 # después este 2
S 1 # después espalda SUR
tu   # lápiz arriba

```

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

6. Diseñe una gramática BNF para analizar una especificación de tiempo. Todos los siguientes ejemplos deben ser aceptados.

*Responder
en P. 282*

4pm, 7:38pm, 23:42, 3:16, 3:16am

7. Implemente un analizador para la gramática BNF en el Ejercicio 6 usando yacc, bisonte, o un analizador-generador similar.

*Responder
en P. 282*

8. Implemente el analizador de tiempo usando Perl. [Insinuación: Las expresiones regulares son buenos analizadores.]

*Responder
en P. 283*

13

Estimación

¡Rápido! ¿Cuánto tiempo se tarda en enviar *Guerra y paza* través de una línea de módem de 56k? ¿Cuánto espacio en disco necesitará para un millón de nombres y direcciones? ¿Cuánto tarda un bloque de 1000 bytes en pasar por un enrutador? ¿Cuántos meses tardará en entregar su proyecto?

En un nivel, todas estas son preguntas sin sentido: todas son información faltante. Y, sin embargo, todas pueden responderse, siempre que se sienta cómodo estimando. Y, en el proceso de producir una estimación, comprenderá más sobre el mundo en el que habitan sus programas.

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

TIP18

Estimar 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 un presupuesto.

¿Qué tan preciso es lo suficientemente preciso?

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

- Si tu abuela te pregunta cuándo llegarás, probablemente se esté preguntando 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? Si se pregunta cuántos bordes comprar para poner alrededor de un macizo de flores circular, entonces "3" probablemente sea lo suficientemente bueno.¹⁰ Si estás en la escuela, quizás²² " sea una buena aproximación. Si estás en la NASA, quizás 12 decimales sean suficientes.

Una de las cosas interesantes de estimar es que las unidades que usa marcan la diferencia en la interpretación del resultado. Si dice que algo tomará alrededor de 130 días hábiles, 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 implica un mayor grado de precisión de lo que cree. Le recomendamos que escale las estimaciones de tiempo de la siguiente manera:

Duración	Presupuesto presupuestado en
1–15 días	días
3–8 semanas	semanas
8–30 semanas	meses
30+ semanas	piénsalo bien antes de dar un presupuesto

Entonces, si después de hacer todo el trabajo necesario, decide que un proyecto tomará 125 días hábiles (25 semanas), es posible que desee entregar una estimación de "alrededor de 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, debemos mencionar un truco básico de estimación que siempre da buenas respuestas: pregúntele a alguien que ya lo haya hecho. Antes de comprometerse demasiado con la construcción de modelos, busque a alguien que haya estado en una situación similar en el pasado.

10. "3" aparentemente también es lo suficientemente bueno si eres legislador. En 1897, el Proyecto de Ley No. 246 de la Cámara de la Legislatura del Estado de Indiana intentó decretar que de ahora en adelante debería tener el valor de "3". El proyecto de ley se pospuso indefinidamente en su segunda lectura cuando un profesor de matemáticas señaló que sus poderes no se extendían del todo a la aprobación de leyes de la naturaleza.

Vea cómo se resolvió su problema. Es poco probable que alguna vez encuentre una coincidencia exacta, pero se sorprendería de cuántas veces puede aprovechar con éxito las experiencias de otros.

Comprender lo que se pregunta

La primera parte de cualquier ejercicio de estimación es desarrollar una comprensión de lo que se pregunta. Además de los problemas de precisión discutidos 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 elija formará parte de la respuesta que dé: "Suponiendo que no haya accidentes de tráfico y que haya gasolina en el automóvil, debería estar allí en 20 minutos".

Construya un modelo del sistema

Esta es la parte divertida de estimar. A partir de su comprensión de la pregunta que se le hace, construya un modelo mental básico, aproximado y listo. Si está estimando los tiempos de respuesta, su modelo puede involucrar 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 podría implementarse el sistema.

La construcción de modelos puede ser tanto creativa como útil a largo plazo. A menudo, el proceso de construcción del modelo conduce a descubrimientos de patrones y procesos subyacentes que no eran aparentes en la superficie. Incluso es posible que desee volver a examinar la pregunta original: "Usted pidió un presupuesto para hacer X . Sin embargo, parece que Y , una variante de X , se puede hacer en aproximadamente la mitad del tiempo y solo se pierde una función".

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

Dividir el modelo en componentes

Una vez que tenga un modelo, puede descomponerlo en componentes. Deberá descubrir las reglas matemáticas que describen cómo interactúan estos componentes. A veces, un componente contribuye con un solo valor

que se suma 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 normalmente tendrá parámetros que afectan la forma en que contribuye al modelo general. En esta etapa, simplemente identifique cada parámetro.

Dar a cada parámetro un valor

Una vez que haya desglosado los parámetros, puede revisarlos y asignarles un valor a cada uno. Espera introducir algunos errores en este paso. El truco consiste en determinar qué parámetros tienen el mayor impacto en el resultado y concentrarse en hacerlos bien. Normalmente, los parámetros cuyos valores se suman a un resultado son menos significativos que aquellos 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 un efecto notable.

Debe tener una forma justificable de calcular estos parámetros críticos. Para el ejemplo de la cola, es posible que desee medir la tasa de llegada de transacciones real 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 realizar una estimación utilizando las técnicas descritas en esta sección. De hecho, a menudo se encontrará basando una estimación en otras subestimaciones. Aquí es donde aparecerán sus mayores errores.

Calcular las respuestas

Solo en el más simple de los casos una estimación tendrá una única respuesta. Es posible que esté feliz de decir "Puedo caminar cinco cuadras cruzando la ciudad en 15 minutos". Sin embargo, a medida que los sistemas se vuelven más complejos, querrá proteger sus respuestas. Ejecute múltiples cálculos, variando los valores de los parámetros críticos, hasta que determine cuáles realmente impulsan el modelo. Una hoja de cálculo puede ser de gran ayuda. Luego exponga su 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 una memoria de 64 MB, y un segundo con una memoria de 48 MB". (Observe cómo "tres cuartos de segundo" transmite una sensación de precisión diferente a 750 ms).

Durante la fase de cálculo, puede comenzar a obtener respuestas que parecen extrañas. No se apresure a descartarlos. Si su aritmética es correcta, su comprensión del problema o su modelo probablemente sea incorrecto. Esta es información valiosa.

Lleve un registro de su destreza de estimación

Creemos que es una gran idea registrar sus estimaciones para que pueda ver qué tan cerca estuvo. Si una estimación general involucró el cálculo de subestimaciones, también realice un seguimiento de estas. A menudo encontrará que sus estimaciones son bastante buenas; de hecho, después de un tiempo, llegará a esperar esto.

Cuando una estimación resulta incorrecta, no se encoja de hombros y se vaya. Averigüe por qué difiere de su suposición. Tal vez eligió 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 hace, su próxima estimación será mejor.

Estimación de cronogramas de proyectos

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

- Consultar requisitos
- Analizar 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 durar. Algunos métodos requieren que usted fije esto como parte del plan inicial, pero para todos los proyectos, excepto los más triviales, esto es un error. A menos que esté haciendo una aplicación similar a una anterior, con el mismo equipo y la misma tecnología, solo estaría adivinando.

Entonces completa la codificación y prueba de la funcionalidad inicial y marca esto como el final del primer incremento. Con base en esa experiencia, puede refinar su suposición inicial sobre el número de iteraciones y qué

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

TIP19

Iterar el horario con el código

Esto puede no ser popular entre la gerencia, que normalmente quiere un número único, duro y rápido incluso antes de que comience el proyecto. Deberá ayudarlos a comprender que el equipo, su productividad y el entorno determinarán el cronograma. Al formalizar esto y refinar la programación como parte de cada iteración, les proporcionará las estimaciones de programación más precisas que pueda.

Qué decir cuando se le pide un presupuesto

Tu dices “*Volvere a ti.*”

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

Las secciones relacionadas incluyen:

- *Velocidad del algoritmo*, página 177

Desafíos

- Comience a llevar un registro de sus estimaciones. Para cada uno, haga un seguimiento de qué tan preciso resultó ser. Si su error fue superior al 50 %, intente averiguar en qué se equivocó su estimación.

Ejercicios

9. Se le pregunta "¿Qué tiene un mayor ancho de banda: 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 en su respuesta para asegurarse de que el alcance de su respuesta sea correcto? (Por ejemplo, podría decir que se ignora el tiempo necesario para acceder a la cinta).

*Responder
en P. 283*

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

*Responder
en P. 284*

Esta página se dejó en blanco intencionalmente

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 abrazaderas, mazos y abrazaderas. Estas herramientas se elegirán con amor, se construirán 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 mantenerse así. Con el tiempo, cada uno se desgastará según el uso, hasta que la empuñadura parezca un molde de las manos de un carpintero y la superficie de corte se alinee perfectamente con el ángulo en el que se sujetaba 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 de inglete guiadas por láser, caladoras de cola de milano, todas maravillosas piezas 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 a través de la madera.

Las herramientas amplifican tu talento. Cuanto mejores sean sus herramientas y mejor sepa cómo usarlas, más productivo podrá ser. Comience con un conjunto básico de herramientas de aplicación general. A medida que gane experiencia y encuentre requisitos especiales, agregará a este conjunto básico. Al igual que el artesano, espere agregar a su caja de herramientas regularmente. Esté siempre atento a mejores formas de hacer las cosas. Si se encuentra con una situación en la que siente que sus herramientas actuales no pueden cortarlo, tome nota para 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 realmente es un error. Necesitamos sentirnos cómodos más allá de los límites impuestos por un IDE. La única forma de hacer esto es mantener el juego de herramientas básico afilado y listo para usar.

En este capítulo hablaremos sobre invertir en su propia caja de herramientas básica. Como con cualquier buena discusión sobre herramientas, comenzaremos (*en El poder del texto sin formato*) al observar sus materias primas, las cosas a las que dará forma. Desde allí nos moveremos al banco de trabajo, o en nuestro caso a la computadora. ¿Cómo puedes usar tu computadora para aprovechar al máximo las herramientas que usas? Discutiremos esto *en Juegos de conchas*. Ahora que tenemos material y un banco para trabajar, pasaremos a la herramienta que probablemente usará más que cualquier otra, su editor. En *Edición de potencia*, le sugeriremos formas de hacerlo más eficiente.

Para asegurarnos de que nunca perdemos nada de nuestro precioso trabajo, siempre debemos usar un *Control de código fuente* sistema, ¡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 vuelves muy hábil en *depuración*.

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

Así como 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 *Generadores de código*.

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 pensarlo 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 requisitos como conocimiento y luego expresamos ese conocimiento en nuestros diseños, implementaciones, pruebas y documentos. Y creemos que el mejor formato para almacenar conocimiento de forma persistente es *Texto sin formato*. Con el texto sin formato, 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?

Texto sin formato se compone de caracteres imprimibles en un formato que las personas pueden leer y comprender directamente. Por ejemplo, aunque el siguiente fragmento se compone de caracteres imprimibles, no tiene sentido.

```
Campo19=467abe
```

El lector no tiene idea de cuál es el significado de 467abe quizás. Una mejor opción sería hacerlo *comprendible* a humanos.

```
DrawingType=UMLActivityDrawing
```

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. Suponga que desea almacenar una propiedad llamada `utiliza_menus` que puede ser `CIERTO` o `FALSO`. Usando texto, puede escribir esto como

```
myprop.usas_menus=FALSO
```

Contrasta esto con 0010010101110101.

El problema con la mayoría de los formatos binarios es que el contexto necesario para comprender los datos está separado de los datos mismos. Estás 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 es independiente de la aplicación que lo creó.

TIP20

Mantener 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 su aplicación, cualquiera de estas situaciones o ambas pueden ser inaceptables, por ejemplo, cuando se almacenan datos de telemetría satelital o como el formato interno de una base de datos relacional.

Pero incluso en estas situaciones, puede ser aceptable almacenar *metadatos* sobre los datos sin procesar en texto sin formato (ver *Metaprogramación*, página 144).

Algunos desarrolladores pueden preocuparse de que al poner los metadatos en texto sin formato, los estén exponiendo 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 le preocupa que los usuarios vean las contraseñas, cifíquelas. Si no quiere que cambien los parámetros de configuración, incluya un *hash seguro*¹ de todos los valores de los parámetros en el archivo como una suma de control.

El poder del texto

Ya que *más grande* y *Más lento* son las funciones más solicitadas por los usuarios, ¿por qué molestarse con el texto sin formato? Qué *son* los beneficios?

- Seguro contra obsolescencia
- Apalancamiento
- Pruebas más fáciles

Seguro Contra Obsolescencia

Las formas de datos legibles por humanos y los datos autodescriptivos sobrevivirán a todas las demás formas de datos y las aplicaciones que las crearon. Período.

1. MD5 se usa a menudo para este propósito. Para una excelente introducción al maravilloso mundo de la criptografía, consulte [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 dicho archivo con solo un conocimiento parcial de su formato; Con la mayoría de los archivos binarios, debe conocer todos los detalles del formato completo para poder analizarlo correctamente.

Considere un archivo de datos de algún sistema heredado que te dan. Sabes poco sobre la aplicación original; todo lo que es importante para usted es que mantuvo una lista de los 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 imagine si el archivo hubiera sido formateado de esta manera:

```
AC27123456789B11P
...
XY43567890123QTYL
...
6T2190123456788AM
```

Es posible que no haya reconocido el significado de los números tan fácilmente. Esta es la diferencia entre *legible por humanos* y *humano comprensible*.

Mientras estamos en eso, CAMPO10 tampoco ayuda mucho. Algo como

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

hace que el ejercicio sea sencillo y asegura que los datos sobrevivirán a cualquier proyecto que los haya creado.

Aprovechar

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

2. Todo el software se convierte en legado 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 bien una cosa. 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 red, etc.) se mantienen como archivos de texto sin formato. (Algunos sistemas, como Solaris, también mantienen una forma binaria de ciertas bases de datos como una optimización del rendimiento. La versión de texto sin formato se mantiene como una interfaz para la versión binaria).

Cuando un sistema falla, es posible que se enfrente a un entorno mínimo para restaurarlo (es posible que no pueda acceder a los controladores de gráficos, por ejemplo). Situaciones como esta realmente pueden hacerte apreciar la simplicidad del texto sin formato.

Por ejemplo, suponga que tiene una implementación de producción de una aplicación grande con un archivo de configuración complejo específico del sitio (enviar correo me viene a la mente). Si este archivo está en texto sin formato, puede colocarlo bajo un sistema de control de código fuente (ver *Control de código fuente*, página 86), para que mantenga automáticamente un historial de todos los cambios. Herramientas de comparación de archivos como `diff` y `cvs` le permite ver de un vistazo qué cambios se han realizado, mientras `sum` le permite generar una suma de verificación para monitorear el archivo en busca de modificaciones accidentales (o maliciosas).

Pruebas más fáciles

Si usa texto sin formato para crear datos sintéticos para impulsar las pruebas del sistema, entonces es una cuestión simple agregar, actualizar o modificar los datos de prueba. *sin tener que crear ninguna herramienta especial para hacerlo.* De manera similar, la salida de texto sin formato de las pruebas de regresión se puede analizar de forma trivial (`diff`, por ejemplo) o sometido a un escrutinio más exhaustivo con Perl, Python o alguna otra herramienta de secuencias de comandos.

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 omnipresente archivo de texto seguirá estando ahí. de hecho, en

entornos heterogéneos, las ventajas del texto sin formato pueden superar todos los inconvenientes. Debe asegurarse de que todas las partes puedan comunicarse utilizando un estándar común. El texto sin formato es ese estándar.

Las secciones relacionadas incluyen:

- *Control de código fuente*, página 86
- *Generadores de código*, página 102
- *Metaprogramación*, página 144 *pizarras*
- , página 165 *Automatización ubicua*,
- página 230 *todo es escritura*, página
- 248

Desafíos

- Diseñe una pequeña base de datos de la libreta de direcciones (nombre, número de teléfono, etc.) utilizando una representación binaria sencilla en el idioma de su elección. Haz esto antes de leer el resto de este desafío.
 1. Traduzca ese formato a un formato de texto sin formato usando XML.
 2. Para cada versión, agregue un nuevo campo de longitud variable llamado *direcciones* en el que puede ingresar direcciones 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 una mesa de trabajo buena, sólida y confiable, en algún lugar para sostener las piezas de trabajo a una altura conveniente mientras las trabaja. El banco de trabajo se convierte en el centro del taller de madera, el artesano regresa a él una y otra vez a medida que una pieza toma forma.

Para un programador que manipula archivos de texto, ese banco de trabajo es el shell de comandos. Desde el indicador de shell, puede invocar su repertorio completo de herramientas, utilizando canalizaciones para combinarlas en formas que nunca soñaron sus desarrolladores originales. Desde el shell, puede iniciar aplicaciones, depuradores, navegadores, editores y utilidades. Puede buscar archivos,

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

Para los programadores criados en interfaces GUI y entornos de desarrollo integrados (IDE), esta puede 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 cartas son todas las cosas que quizás desee hacer en un entorno gráfico. Pero si hace todo su trabajo utilizando GUI, se está perdiendo todas las capacidades de su entorno. No podrá automatizar tareas comunes ni utilizar toda la potencia de las herramientas disponibles. Y no podrá combinar sus herramientas para crear *herramientas de macros*. Una ventaja de las GUI es WYSIWYG: lo que ve es lo que obtiene. La desventaja es WYSIAYG: lo que ves es *todo* lo que obtiene.

Los entornos de GUI normalmente se limitan a las capacidades que sus diseñadores pretendían. Si necesita ir más allá del modelo proporcionado por el diseñador, por lo general no tiene suerte, y la mayoría de las veces, *hacer* hay que ir más allá del modelo. Los programadores pragmáticos no solo cortan código, desarrollan modelos de objetos, escriben documentación o automatizan el proceso de compilación, lo hacen *todos* de estas cosas. El alcance de cualquier herramienta por lo general se limita a las tareas que se espera que realice la herramienta. Por ejemplo, suponga que necesita integrar un preprocesador de código (para implementar pragmas de diseño por contrato o multiprocesamiento, o algo así) en su IDE. A menos que el diseñador del IDE proporcione explícitamente ganchos 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, querrá realizar constantemente operaciones ad hoc, cosas que la GUI puede no admitir. La línea de comandos es más adecuada cuando desea combinar rápidamente un par de comandos para realizar una consulta o alguna otra tarea. Aquí están algunos ejemplos.

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

Caparazón ... encontrar . -nombre '*.c' -nuevo Makefile -imprimir

interfaz gráfica de usuario Abra el Explorador, navegue hasta el directorio correcto, haga clic en Makefile y anote la hora de modificación. Luego abra Herramientas/ Buscar e ingrese *.Cpara la especificación del archivo. Seleccione la pestaña de fecha e ingrese la fecha que anotó para el Makefile en el primer campo de fecha. Luego presione Aceptar.

Construya un archivo zip/tar de mi fuente.

Caparazón ... zip archivo.zip *.h *.c tar cvf archivo.tar *.h
*.c - O -

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

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

Caparazón ... encontrar . -nombre '*.java' -mtime +7 -imprimir

interfaz gráfica de usuario Haga clic y navegue hasta "Buscar archivos", haga clic en el campo "Nombre" y escriba "*.java", seleccione la pestaña "Fecha de modificación". Luego seleccione "Entre". Haga clic en la fecha de inicio y escriba la fecha de 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, que utilizan elawtbibliotecas?

Caparazón ... encontrar . -nombre '*.java' -mtime +7 -imprimir |
xargs grep 'java.awt'

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.

Claramente, la lista podría continuar. Los comandos de 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 secuencias de comandos (o archivos de comandos en Windows

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

TIP21

Usa el poder de los proyectiles de comando

Familiarícese con el caparazón y notará que su 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 '^importar' *.java |  
    sed -e's/.*/import * //i' sort -u >lista      - e's/;.*$//' |
```

Si no ha pasado mucho tiempo explorando las capacidades del shell de comandos en los sistemas que usa, 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 comando y se sorprenderá de lo mucho más productivo que lo hace.

Utilidades de Shell y sistemas Windows

Aunque los shells de comandos provistos con los sistemas Windows están mejorando gradualmente, las utilidades de línea de comandos de Windows aún son 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 con Unix para Windows, Cygwin viene con una colección de más de 120 utilidades de Unix, incluidas las favoritas como ls, grep, y encontrar. Las utilidades y bibliotecas se pueden descargar y usar de forma gratuita, pero asegúrese de leer su licencia.³ La distribución Cygwin viene con el shell Bash.

3. La Licencia pública general de GNU [URL 57] es un tipo de virus legal que los desarrolladores de código abierto utilizan para proteger sus derechos (y los tuyos). Deberías pasar un tiempo leyéndolo. En esencia, dice que puedes usar y modificar el software bajo GPL, pero si distribuyes cualquier modificación, debe tener una licencia de acuerdo con la GPL (y debe estar marcada como tal), y debe hacer que la fuente esté disponible. Esa es la parte del virus: cada vez que deriva un trabajo de un trabajo bajo GPL, su trabajo derivado también debe tener 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 las herramientas de 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 hay 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 al murciélagos encontraré AUTOEXEC.BAT. También puede encontrar problemas con los nombres de archivo que contienen espacios y con las diferencias en los separadores de ruta. Finalmente, existen problemas interesantes cuando se ejecutan programas de MS-DOS que esperan argumentos al estilo de MS-DOS bajo los shells de Unix. Por ejemplo, las utilidades Java de JavaSoft usan dos puntos como separador CLASSPATH en Unix, pero usan un punto y coma en MS-DOS. Como resultado, una secuencia de comandos Bash o ksh que se ejecuta en un cuadro de Unix se ejecutará de manera idéntica en Windows, pero la línea de comando que pasa a Java se interpretará incorrectamente.

Alternativamente, David Korn (de la fama de Korn shell) ha creado 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 (al momento de escribir este artículo) armando *Herramientas eléctricas Perl*, un intento de implementar todas las utilidades conocidas de Unix de forma portátil, en Perl [URL 32].

Las secciones relacionadas incluyen:

- *Automatización ubicua*, página 230

Desafíos

- ¿Hay cosas que actualmente está haciendo 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 elemento"? ¿Se podrían automatizar?
- Cada vez que se mude a un nuevo entorno, asegúrese de averiguar qué shells están disponibles. Vea si puede traer su caparazón actual con usted.
- Investigue alternativas a su caparazón actual. Si se encuentra con un problema que su shell no puede resolver, vea si un shell alternativo lo resolvería mejor.



Edición de potencia

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. Debe poder manipular el texto con la mayor facilidad posible, porque el texto es la materia prima básica de la programación. Veamos algunas características y funciones comunes que lo ayudan a aprovechar al máximo su entorno de edición.

un editor

Creemos que es mejor conocer muy bien un editor y usarlo para todas las tareas de edición: código, documentación, notas, administración del sistema, etc. Sin un solo editor, te enfrentas a una posible Babel moderna de confusión. Es posible que deba 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 usa para editar líneas de comando en el shell pueden ser diferentes.⁴ Es difícil dominar cualquiera de estos entornos si tiene un conjunto diferente de convenciones y comandos de edición en cada uno.

Necesitas ser competente. Simplemente escribir linealmente y usar un mouse para cortar y pegar no es suficiente. Simplemente no puede ser tan efectivo de esa manera como con un poderoso editor bajo sus dedos. escribiendo o ↵ [RETROCESO] diez veces para mover el cursor hacia la izquierda hasta el comienzo de una línea no es tan eficiente como escribir una sola tecla como , A [Hogar], o . 0

TIP22

Use un pozo de un solo editor

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

4. Idealmente, el shell que use debe tener combinaciones de teclas que coincidan con las que usa su editor. Bash, por ejemplo, es compatible con ambos vi y emacs tajos de teclado.

una extensión de tu mano; las teclas cantarán mientras se abren camino a través del texto y el pensamiento. Ese es nuestro objetivo.

Asegúrese de que el editor que elija esté disponible en todas las plataformas que utilice. Emacs, vi, CRISP, Brief y otros están disponibles en varias plataformas, a menudo en versiones GUI y no GUI (pantalla de texto).

Características del editor

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

- **Configurable.**Todos los aspectos del editor deben ser configurables según sus preferencias, incluidas las fuentes, los colores, los tamaños de ventana y las combinaciones de teclas (qué teclas ejecutan qué comandos). Usar solo pulsaciones de teclas para operaciones de edición comunes es más eficiente que el mouse o los comandos controlados por menús, porque sus manos nunca dejan el teclado.
- **Extensible.**Un editor no debería quedar obsoleto solo porque sale un nuevo lenguaje de programación. Debería poder integrarse con cualquier entorno de compilación que esté utilizando. Debería poder "enseñarle" los matices de cualquier nuevo idioma o formato de texto (XML, HTML versión 9, etc.).
- **Programable.**Debería poder programar el editor para realizar tareas complejas de varios pasos. Esto se puede hacer con macros o con un lenguaje de programación de secuencias de comandos incorporado (Emacs usa una variante de Lisp, por ejemplo).

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

- Resaltado de sintaxis
- Autocompletar
- sangría automática
- Código inicial o documento repetitivo
- Vinculación a sistemas de ayuda
- Funciones similares a IDE (compilación, depuración, etc.)

Figura 3.1. Ordenar líneas en un editor

```
importar java.util.Vector;
importar java.util.Stack;
importar java.net.URL;
importar java.awt.*;
```

emacs:M-xsort-lines
vi:...+3! ordenar

```
importar java.awt.*;
importar java.net.URL;
importar java.util.Stack;
importar java.util.Vector;
```

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 se acostumbre a ver que las palabras clave aparecen en un color o fuente diferente, una palabra clave mal escrita que no aparecer de esa manera te salta a la vista mucho antes de que inicies 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

Una sorprendente cantidad de personas que hemos conocido usan Windowsbloc utilidad para editar su código fuente. Esto es como usar una cucharilla como una pala: simplemente escribir y usar las funciones básicas de cortar y pegar con el mouse no es suficiente.

¿Qué tipo de cosas necesitarás para hacer eso? *no poderse* hace de esta manera?

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

O suponga que está escribiendo código Java. Te gusta mantener tu importar declaraciones 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 a través de una larga lista de importar declaraciones). Le gustaría revisar rápidamente algunos archivos y ordenar una pequeña sección de ellos. En editores como vi y Emacs puede hacerlo fácilmente (vea la Figura 3.1). Probar que en bloc.

Algunos editores pueden ayudar a optimizar las operaciones comunes. Por ejemplo, cuando crea un nuevo archivo en un idioma en particular, el editor puede proporcionarle 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 idioma (declaraciones de constructor y destructor, por ejemplo)

Otra característica útil es la sangría automática. En lugar de tener que sangrar manualmente (mediante espacio o tabulación), el editor automáticamente sangra en el momento apropiado (después de escribir una llave abierta, por ejemplo). Lo bueno de esta característica es que puede usar el editor para proporcionar un estilo de sangría consistente para su proyecto.⁵

A dónde ir desde aquí

Este tipo de consejo es particularmente difícil de escribir porque prácticamente todos los lectores se encuentran en un nivel diferente de comodidad y experiencia con los editores que están utilizando actualmente. Por lo tanto, para resumir y brindar orientación sobre adónde ir a continuación, búsquese en la columna de la izquierda del gráfico y mire la columna de la derecha para ver lo que creemos que debe hacer.

Si esto te suena

...

Entonces piensa en ...

Solo uso funciones básicas de muchos editores diferentes.

Elige un editor potente y apréndelo bien.

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

Aprendelos. Reduzca el número de pulsaciones de teclas que necesita escribir.

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

Intenta expandirlo y usarlo para más tareas de las que ya haces.

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

Mientras seas feliz y productivo, ¡adelante! Pero si se encuentra sujeto a la "envidia del editor", es posible que deba reevaluar su posición.

5. El kernel de Linux se desarrolla de esta manera. Aquí tienes desarrolladores dispersos geográficamente, muchos trabajando 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?

Habiendo recomendado que domines un editor decente, ¿cuál recomendamos? Bueno, vamos a esquivar esa pregunta; tu elección de editor es personal (¡algunos incluso dirían que es 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 secuencias de comandos. Emacs, por ejemplo, usa Lisp. Como uno de los nuevos idiomas que aprenderá este año, aprenda el idioma que usa su 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? Trate de confundir a sus colegas que usan el mismo editor. Intente realizar cualquier tarea de edición determinada con la menor cantidad de pulsaciones de teclas posible.

17

Control de código fuente

El progreso, lejos de consistir en un cambio, depende de la retención. Quien no puede recordar el pasado está condenado a repetirlo.

► **Jorge Santayana, vida de la razon**

Una de las cosas importantes que buscamos en una interfaz de usuario es la clave ver nota solo botón que nos perdone nuestros errores. Es aún mejor si el entorno admite varios niveles de deshacer y rehacer, de modo que pueda regresar y recuperarse de algo que sucedió hace un par de minutos. Pero, ¿qué sucede 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 clave gigante, una máquina del tiempo para todo el proyecto ver nota que puede devolverlo a esos días felices de la semana pasada, cuando el código realmente se compiló y ejecutó.

Sistemas de control de código fuente, o los más amplios *gestión de la configuración* sistemas, realice un seguimiento de cada cambio que realice en su código fuente y documentación. Los mejores pueden hacer un seguimiento de

compilador y versiones del sistema operativo también. Con un sistema de control de código fuente correctamente configurado, *siempre puede volver a una 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 preguntas como: ¿Quién hizo los cambios en esta línea de código? ¿Cuál es la diferencia entre la versión actual y la de la semana pasada? ¿Cuántas líneas de código cambiamos en esta versión? ¿Qué archivos se modifican con más frecuencia? Este tipo de información es invaluable para fines de seguimiento de errores, auditoría, rendimiento y calidad.

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

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, deberá solucionar los errores de la versión actual y enviar versiones corregidas a los clientes. Querrá que estas correcciones de errores se incluyan en la próxima versión (si corresponde), pero no desea enviar el código en desarrollo a los clientes. Con un SCCS puede generar ramas en el árbol de desarrollo cada vez que genera una versión. Aplica correcciones de errores al código en la rama y 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 los cambios seleccionados de la rama nuevamente en el tronco principal automáticamente.

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

Finalmente, algunos productos pueden permitir que dos o más usuarios trabajen al mismo tiempo en el mismo conjunto de archivos, incluso haciendo cambios al mismo tiempo en el mismo archivo. Luego, 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 SCCS en mayúsculas para referirnos a sistemas genéricos de control de código fuente. También hay un sistema específico llamado "sccs", lanzado originalmente con AT&T System V Unix.

TIP23

Utilice siempre el control de código fuente

Siempre. Incluso si eres un equipo de una sola persona en un proyecto de una semana. Incluso si es un prototipo "desechable". Incluso si lo que está trabajando no es código fuente. Asegúrate de *es todo* está bajo el control del código fuente: documentación, listas de números de teléfono, memorandos a los proveedores, archivos MAKE, procedimientos de compilación y publicación, ese pequeño script de shell que graba el CD maestro, todo. Habitualmente utilizamos 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 protegido 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 son *automático* y *repetible*.

El mecanismo de construcción del proyecto puede sacar la fuente más reciente del repositorio automáticamente. Puede ejecutarse 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 rompa nada. La automatización de la compilación garantiza la consistencia: 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 puede reconstruir la fuente tal como existía en una fecha determinada.

Pero mi equipo no usa el control de código fuente

¡Me avergüenzo de ellos! ¡Suena como una oportunidad para hacer un poco de evangelización! Sin embargo, mientras espera que vean la luz, tal vez debería implementar su propio control de fuente privado. Utilice una de las herramientas disponibles gratuitamente que enumeramos en el Apéndice A y asegúrese de mantener su trabajo personal guardado de forma segura en un repositorio (además de hacer lo que requiera su proyecto). Si bien esto puede parecer una duplicación de esfuerzos, podemos garantizarle que le ahorrará molestias (y le ahorrará dinero a su proyecto) la primera vez que necesite responder preguntas como

como "¿Qué le hiciste al xyz módulo?" y "¿Qué rompió la construcción?" Este enfoque también puede ayudar a convencer a su gerencia de que el control del 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

El Apéndice A, página 271, proporciona direcciones URL para sistemas de control de código fuente representativos, algunos comerciales y otros disponibles gratuitamente. Y hay muchos más productos disponibles: busque los punteros a las preguntas frecuentes sobre administración de la configuración. Para obtener 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 usar un SCCS en el trabajo, instale RCS o CVS en un sistema personal. Úselo para administrar 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 cuales los archivos de acceso público están disponibles en la Web (como Mozilla [URL 51], KDE [URL 54] y Gimp [URL 55]). ¿Cómo se obtienen actualizaciones de la fuente? ¿Cómo se realizan los cambios? ¿El proyecto regula el acceso o arbitra la inclusión de cambios?

depuración

es algo doloroso

Mirar tu propio problema y saber que tú mismo y nadie más lo ha hecho.

► **Sófocles, Ajax**

La palabra *insecto* se ha utilizado para describir un "objeto de terror" desde el siglo XIV. A la contraalmirante Dra. Grace Hopper, la inventora de COBOL, se le atribuye la observación de la primera *error informático*—literalmente, una polilla atrapada en un relé en uno de los primeros sistemas informáticos. Cuando se le pidió que explicara por qué la máquina no se estaba comportando como se esperaba, un técnico informó que había "un error en el sistema" y debidamente grabó las 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 que entonces. Los defectos del software se manifiestan de diversas formas, desde requisitos mal entendidos hasta errores de codificación. Desafortunadamente, los sistemas informáticos modernos todavía se limitan a hacer lo que *dicen* que hagan, no necesariamente lo que tú *deseas* que hagan para hacer.

Nadie escribe software perfecto, por lo que es un hecho que la depuración ocupará la mayor parte de su día. Veamos algunos de los problemas involucrados en la depuración y algunas estrategias generales para encontrar errores esquivos.

Psicología de la depuración

La depuración en sí misma es un tema sensible y emotivo para muchos desarrolladores. En lugar de atacarlo como un rompecabezas que debe resolverse, es posible que encuentre negación, señalamientos con el dedo, excusas tontas o simplemente apatía.

Acepta el hecho de que la depuración es solo *resolución de problemas*, y atacarlo como tal.

Después de haber encontrado el error de otra persona, puede dedicar tiempo y energía a culpar al asqueroso 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, desea concentrarse en arreglar el *problema*, no la culpa.

TIP24

Solucioné 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

La persona más fácil de engañar es uno mismo.

► **Edward Bulwer Lytton, los repudiados**

Antes de comenzar a depurar, es importante adoptar la mentalidad correcta. Debe apagar muchas de las defensas que usa todos los días para proteger su ego, desconectarse de cualquier presión del proyecto que pueda estar bajo y ponerse cómodo. Sobre todo, recuerda la primera regla de depuración:

TIP25

No entrar en pánico

Es fácil entrar en pánico, especialmente si se enfrenta a una fecha límite, o si tiene un jefe nervioso o un cliente que está detrás de usted mientras intenta encontrar la causa del error. Pero es muy importante dar un paso atrás, y en realidad *pensar* sobre lo que podría estar causando los síntomas que usted cree que indican un error.

Si su primera reacción al presenciar un error o ver un informe de error es "eso es imposible", está claramente equivocado. No desperdices ni una sola neurona en el tren de pensamiento que comienza "pero eso no puede suceder" porque claramente *pueden*, y tiene.

Cuidado con la miopía al depurar. Resista la tentación de corregir solo los síntomas que ve: es más probable que la falla real esté varios pasos alejada 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 su apariencia particular.

Donde empezar

Antes de *ticomenzopara* ver el error, asegúrese de que está trabajando en un código que se compiló limpiamente, sin advertencias. Nos fijamos rutinariamente

niveles de advertencia del compilador tan altos como sea posible. ¡No tiene sentido perder el tiempo tratando de encontrar un problema que el compilador podría encontrar por ti! 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, la notificación de errores no es una ciencia exacta. Es fácil dejarse engañar por las coincidencias y no puede permitirse el lujo de perder el tiempo depurando las coincidencias. Primero debe ser preciso en sus observaciones.

La precisión en los informes de errores se reduce aún más cuando provienen de un tercero; en realidad, es posible que deba *releojel* usuario que informó el error en acción para obtener un nivel de detalle suficiente.

Andy una vez trabajó en una gran aplicación de gráficos. Cerca del lanzamiento, los evaluadores informaron que la aplicación fallaba cada vez que pintaban un trazo con un pincel en particular. El programador responsable argumentó que no había nada de malo en ello; había intentado pintar con él, y funcionó muy bien. Este diálogo fue de ida y vuelta durante varios días, y los ánimos subieron rápidamente.

Finalmente, los juntamos 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, quien luego admitió tímidamente que había hecho trazos de prueba solo desde la parte inferior izquierda hasta la parte superior derecha, lo que no expuso el error.

Hay dos puntos en esta historia:

- Es posible que deba entrevistar al usuario que informó el error para recopilar más datos de los que se le proporcionaron inicialmente.
- Las pruebas artificiales (como la pinzellada única del programador de abajo hacia arriba) no ejercitan lo suficiente una aplicación. Debe probar brutalmente tanto las condiciones límite como los patrones de uso realistas del usuario final. Tiene que hacer esto sistemáticamente (ver *Pruebas despiadadas*, página 237).

Estrategias de depuración

Una vez tú crees que sabe lo que está pasando, es hora de averiguar qué *programa* piensa que está pasando.

Reproducción de errores

No, nuestros bichos en realidad no se están multiplicando (aunque algunos de ellos probablemente tengan la edad suficiente para hacerlo legalmente). Estamos hablando de un tipo diferente de reproducción.

La mejor manera de comenzar a corregir un error es hacerlo reproducible. Después de todo, si no puede reproducirlo, ¿cómo sabrá si alguna vez se solucionó?

Pero queremos más que un error que pueda reproducirse siguiendo una larga serie de pasos; queremos un error que se pueda reproducir con un *comando único*. Es mucho más difícil corregir un error si tiene que pasar por 15 pasos para llegar al punto donde aparece el error. A veces, al obligarte a aislar las circunstancias que muestran el error, incluso obtendrás una idea de cómo solucionarlo.

Ver *Automatización ubicua*, página 230, para otras ideas en este sentido.

Visualice sus datos

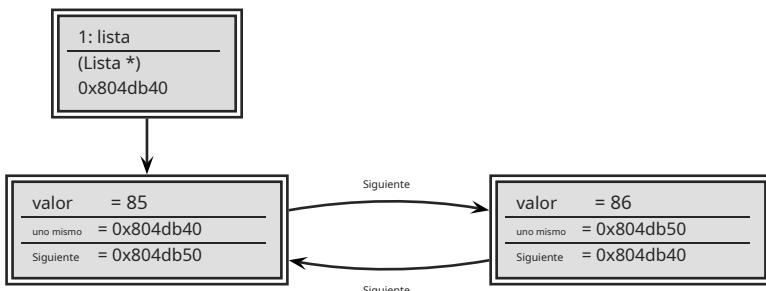
A menudo, la forma más fácil de discernir lo que está haciendo un programa, o lo que va a hacer, es observar bien los datos con los que está operando. El ejemplo más simple de esto es un simple "nombre de variable = valor de datos" enfoque, que puede implementarse como texto impreso o como campos en 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 permite *visualizartus* datos y todas las interrelaciones que existen. Hay depuradores que pueden representar sus datos como un vuelo 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 avanza en su programa, imágenes como estas pueden valer mucho más que mil palabras, ya que el error que ha estado buscando de repente salta a la vista.

Incluso si su depurador tiene soporte limitado para visualizar datos, aún 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 gratuitamente (ver [URL 19]). Es interesante notar que DDD trabaja con

Figura 3.2. Ejemplo de diagrama de depuración de una lista enlazada circular. Las flechas representan punteros a nodos.



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

Rastreo

Los depuradores generalmente se enfocan en el estado del programa *ahora*. A veces, necesita más: debe observar el estado de un programa o una estructura de datos a lo largo del tiempo. Ver un seguimiento de la pila solo puede decirle cómo llegó aquí directamente. No puede decirle qué estaba haciendo antes de esta cadena de llamadas, especialmente en sistemas basados en eventos.

Trazado de declaraciones son esos pequeños mensajes de diagnóstico que imprime 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 especialmente eficaz para diagnosticar varias clases de errores que los depuradores no pueden. El seguimiento tiene un valor incalculable en cualquier sistema en el que el tiempo mismo sea un factor: procesos concurrentes, sistemas en tiempo real y aplicaciones basadas en eventos.

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

Los mensajes de rastreo deben estar en un formato regular y consistente; es posible que desee analizarlos automáticamente. Por ejemplo, si necesita rastrear una fuga de recursos (como aperturas/cierres de archivos desequilibrados), podría rastrear cada abierto y cada cerca en un archivo de registro. Al procesar el 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 alrededor de esta variable corrupta. A menudo te dará una pista. En nuestro caso, examinar la memoria circundante como personajes nos muestra

```
20333231 6e69614d 2c745320 746f4e0a  
1 2 3      Principal      S t , n No \  
2c6e776f 2058580a 31323433 00000a33  
propio, n XX \  
3 4 2 1 3 norte 0\o \ \
```

Parece que alguien escribió la dirección de una calle sobre nuestro mostrador. Ahora sabemos dónde buscar.

archivo con Perl, podría identificar fácilmente dónde está el infractor abierto o estaba ocurriendo.

pato 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 su hombro a la pantalla y asentir con la cabeza constantemente (como un patito de goma que se balancea arriba y abajo en una bañera). No necesitan decir una palabra; el simple acto 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.⁷

Suena simple, pero al explicar el problema a otra persona, debe indicar explícitamente las cosas que puede dar por sentadas al revisar el código usted mismo. Al tener que verbalizar algunas de estas suposiciones, de repente puede obtener una nueva perspectiva del problema.

7. ¿Por qué "pato de goma"? 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 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 combinación 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 especializadas o algoritmos, etc.) y la plataforma entorno (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 debería ser su primera idea. 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 lo *hace* con un tercero, aún tendrá que eliminar su código antes de enviar el informe de error.

Trabajamos en un proyecto en el que un ingeniero superior estaba convencido de que la Selección era la causa del problema. Ninguna cantidad de persuasión lógica pudo hacerlo cambiar de opinión (el hecho de que todas las demás aplicaciones de red en la caja funcionaran bien era irrelevante). Pasó semanas escribiendo soluciones que, por alguna extraña razón, no parecían solucionar el problema. Cuando finalmente se vio obligado a sentarse y leer la documentación sobre

Selección, descubrió el problema y lo corrigió en cuestión de minutos. Ahora usamos la frase "la selección está rota" como un suave recordatorio cada vez que uno de nosotros comienza a culpar al sistema por una falla que probablemente sea nuestra.

TIP26

"selección" no está roto

Recuerde, si ve huellas de pezuñas, piense en caballos, no en cebras. El sistema operativo probablemente no esté roto. Y la base de datos probablemente esté bien.

Si "cambió solo una cosa" y el sistema dejó de funcionar, es probable que esa única cosa sea la 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 en 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. Cambio de API, cambios de funcionalidad; en resumen, es un juego de pelota completamente nuevo, y debe volver a probar el sistema bajo estos

nuevas condiciones Así que vigile de cerca el cronograma cuando considere una actualización; es posible que desee esperar hasta *después* el próximo lanzamiento.

Sin embargo, si no tiene un lugar obvio para comenzar a buscar, siempre puede confiar en una buena búsqueda binaria a la antigua. Vea si los síntomas están presentes en cualquiera de los dos puntos lejanos en el código. Entonces mira en el medio. Si el problema está presente, entonces el error se encuentra entre el inicio 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 (quizás incluso murmurando "eso es imposible" en voz baja donde no podemos escucharte), debes reevaluar las verdades que aprecias. En esa rutina de lista enlazada, la que sabía que era a prueba de balas y no podía ser la causa de este error, ¿probó *todas* las condiciones de contorno? Esa otra pieza de código que ha estado usando durante años, no es posible que todavía tenga un error. ¿Podría?

Por supuesto que puede. La cantidad de sorpresa que siente cuando algo sale mal es directamente proporcional a la cantidad de confianza y fe que tiene en el código que se está ejecutando. Por eso, cuando te enfrentas a un fracaso "sorprendente", debes darte cuenta de que una o más de tus suposiciones son incorrectas. No pase por alto una rutina o pieza de código involucrada en el error porque "sabe" que funciona. Pruébalo. Demuéstralos en este contexto, con estos datos, con estas condiciones de borde.

TIP27

No lo asuma, demuéstrelo

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

Además, si el error es el resultado de datos incorrectos 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 el

discusiones sobre fallas tempranas y afirmaciones en las páginas 120 y 122, respectivamente).

Mientras lo hace, ¿hay otros lugares en el código que puedan ser susceptibles a este mismo error? Ahora es el momento de encontrarlos y corregirlos. Asegúrate de *eso/lo que sea*sucedió, lo sabrás si vuelve a suceder.

Si tomó mucho tiempo 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 crear 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 no entiende, entonces es posible que muchas personas lo hagan.

Haga todo esto y, con suerte, no se sorprenderá la próxima vez.

Lista de comprobación de depuración

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

Las secciones relacionadas incluyen:

- *Programación Asertiva*, página 122
- *Programación por Coincidencia*, página 172
- *Automatización ubicua*, página 230 *Pruebas despiadadas*, página 237

Desafíos

- La depuración es suficiente desafío.

19

Manipulación de texto

Los programadores pragmáticos manipulan el texto de la misma manera que los carpinteros dan forma a la madera. En secciones anteriores discutimos algunos shells de herramientas, editores, depuradores específicos que usamos. Estos 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 el conjunto de herramientas básico no maneja fácilmente. Necesitamos una herramienta de manipulación de texto de propósito general.

Los lenguajes de manipulación de texto son para programar qué enrutadores⁸son para carpintería. Son ruidosos, desordenados y algo de fuerza bruta. Si comete 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 enrutadores como los lenguajes de manipulación de texto pueden ser increíblemente poderosos y versátiles. Puede recortar rápidamente algo para darle forma, hacer uniones y tallar. Usadas correctamente, estas herramientas tienen una delicadeza y sutileza sorprendentes. Pero toman tiempo para dominar.

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 comando, aumentado 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. Preferimos Ruby [TFH04] y Perl [URL 8] para hackear scripts cortos.

Estos lenguajes son importantes tecnologías habilitadoras. Utilizándolos, puede piratear rápidamente utilidades e ideas de prototipos, trabajos que pueden llevar cinco o diez veces más tiempo usando lenguajes convencionales. Y ese factor multiplicador es de vital 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 *La práctica de la programación* [KP99], Kernighan y Pike construyeron el mismo programa en cinco idiomas diferentes. La versión de Perl fue la más corta (17 líneas, en comparación con las 150 de C). Con Perl puedes

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

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

TIP28**Aprenda 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 secuencias de comandos de Perl tomó un archivo de texto sin formato 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 la propiedad de Java.Es un buen estilo de programación orientado a objetos restringir el acceso a las propiedades de un objeto, obligando a las clases externas a obtenerlas y configurarlas mediante métodos. Sin embargo, en el caso común donde una propiedad está representada dentro de la clase por una variable miembro simple, creando un método para cada variable es tedioso y mecánico.
Tenemos una secuencia de comandos Perl que modifica los archivos fuente e inserta las definiciones de método correctas para todas las variables marcadas apropiadamente.
- Prueba de generación de datos.Teníamos decenas de miles de registros de datos de prueba, repartidos en varios archivos y formatos diferentes, que debían unirse y convertirse en un formato 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 se haya probado primero. La mayor parte del código en este

libro ha sido. Sin embargo, usando el *SECOprincipio* (ver *Los males de la duplicación*, 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, garantizando virtualmente que nos olvidaríamos de 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. Se invoca un script relativamente simple cuando formateamos el libro: extrae un segmento con nombre de un archivo fuente, resalta la sintaxis y convierte el resultado al lenguaje de composición tipográfica que usamos.

- Interfaz C a Object Pascal.Un cliente tenía un equipo de desarrolladores que escribían Object Pascal en PC. Su código necesitaba interactuar con un cuerpo de código escrito en C. Desarrollamos un breve script de Perl que analizaba los archivos de encabezado de C, extrayendo las definiciones de todas las funciones exportadas y las estructuras de datos que usaban. Luego generamos unidades 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 siempre que cambiara el encabezado C, se construiría automáticamente una nueva unidad Object Pascal.
- Generación de documentación Web.Muchos equipos de proyectos están publicando su documentación en sitios web internos. Hemos escrito muchos 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 usamos Perl para envolver los documentos con encabezados y pies de página estándar y para transferirlos al sitio web.

Usamos lenguajes de manipulación de texto casi todos los días. Muchas de las ideas de este libro se pueden implementar de forma más sencilla 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 la duplicación*, página 26

Ejercicios

Responder
en P. 285

- 11** Su 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) para fines de depuración. Escriba un script que lea desde la entrada estándar un archivo que contenga

```
nombre
estado_a
estado_b
:
:
```

Producir el archivonombre.h, que contiene

```
carácter const externo* NOMBRE_nombres[];
enumeración typedef{
    estado_a,
    estado_b,
    :
} NOMBRE;
```

y el archivonombre.c, que contiene

```
carácter constante* NOMBRE_nombres[] = {
    "estado_a",
    "estado_b",
    :
};
```

- 12** A mitad de escribir este libro, nos dimos cuenta de que no habíamos puesto el uso estricto directiva en muchos de nuestros ejemplos de Perl. Escriba un script que pase por el .por favor archivos en un directorio y agrega un uso estricto al final del bloque de comentarios inicial a todos los archivos que aún no tienen uno. Recuerde mantener una copia de seguridad de todos los archivos que cambie.

► 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 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 solo necesitamos protegernos del síndrome del túnel carpiano reduciendo la escritura repetitiva.

De la misma manera que un carpintero invierte su tiempo en una plantilla, un programador puede construir un generador de código. Una vez construido, se puede utilizar a lo largo de la vida del proyecto prácticamente sin costo alguno.

TIP29

Escribir código que escribe código

Hay dos tipos principales de generadores de código:

1. *Generadores de código pasivo* 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 *Magos malvados*, página 198, junto con algunas herramientas CASE, son ejemplos de generadores de código pasivo.
2. *Generadores de código activo* utilizan cada vez que se requieren sus resultados. El resultado es desecharable: siempre puede ser reproducido por el generador de código. A menudo, los generadores de código activo leen algún tipo de script o archivo de control para producir sus resultados.

Generadores de código pasivo

Los generadores de código pasivo ahorran tipeo. 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 fuente completo en el proyecto; será editado, compilado y colocado bajo control de código fuente como cualquier otro archivo. Sus orígenes serán olvidados.

Los generadores de código pasivo tienen muchos usos:

- *Creación de nuevos archivos fuente.* 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 archivo nuevo 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 clase de esquema, ya completada.
- *Realización de conversiones únicas entre los lenguajes de programación.* Comenzamos a escribir este libro usando el sistema troff, pero cambiamos a LATEX después de completar 15 tramos. Escribimos un generador de código que leyó la fuente de 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. Puede elegir cuánto esfuerzo pone en el generador, en comparación con la energía que gasta arreglando su salida.

- *Producción de tablas de búsqueda y otros recursos.* que son caros de calcular en tiempo de ejecución. En lugar de calcular funciones trigonométricas, muchos de los primeros sistemas gráficos usaban tablas precalculadas de valores de seno y coseno. Por lo general, estas tablas fueron producidas por un generador de código pasivo y luego copiadas en la fuente.

Generadores de código activo

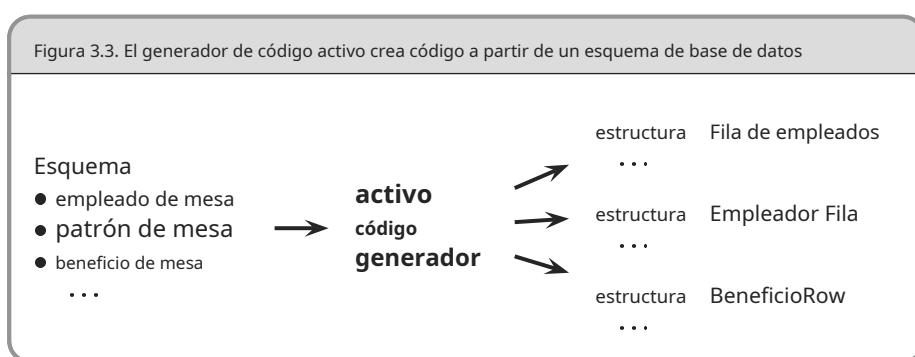
Mientras que los generadores de códigos pasivos son simplemente una conveniencia, sus primos activos son una necesidad si desea seguir el *SECO* principio. Con un generador de código activo, puede tomar una sola representación de algún conocimiento y convertirlo en todas las formas que necesita su aplicación. Esto *no es duplicación*, porque las formas derivadas son desecharables y el generador de código las genera según las necesidades (de ahí la palabra *activo*).

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

Tal vez esté desarrollando una aplicación de base de datos. Aquí, está tratando con dos entornos: la base de datos y el lenguaje de programación que está utilizando para acceder a ella. Tiene un esquema y necesita definir estructuras de bajo nivel que reflejen el diseño de ciertas tablas de la base de datos. Podría simplemente codificarlos directamente, pero esto viola el *SECO* principio: el conocimiento del esquema se expresaría entonces en dos lugares. Cuando cambia el esquema, debe recordar cambiar el código correspondiente. Si se elimina una columna de una tabla, pero la base del código no cambia, es posible que ni siquiera obtenga un error de compilación. Lo primero que sabrá es cuando sus pruebas comienzan a fallar (o cuando el usuario llame).

Una alternativa es utilizar 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 elimina una columna, su campo correspondiente en la estructura desaparecerá y cualquier código de nivel superior que use esa columna no podrá compilarse. Has detectado el error en tiempo de compilación,

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



no en producción. Por supuesto, este esquema solo funciona si haces que la generación de código forme parte del proceso de construcción en sí.⁹

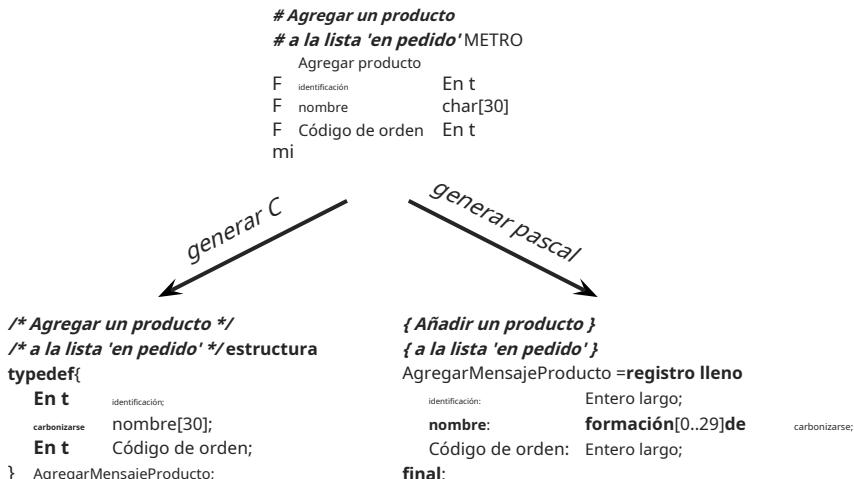
Otro ejemplo de entornos combinados que usan generadores de código ocurre cuando se usan diferentes lenguajes de programación en la misma aplicación. Para comunicarse, cada base de código necesitará cierta información en común: estructuras de datos, formatos de mensajes y nombres de campos, por ejemplo. En lugar de duplicar esta información, utilice un generador de código. A veces, puede analizar la información de los archivos fuente de un idioma y usarla para generar código en un segundo idioma. Sin embargo, a menudo es más sencillo expresarlo en una representación más simple e independiente del lenguaje y generar el código para ambos lenguajes, como se muestra en la Figura 3.4 en 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 del archivo sin formato de la generación de código.

Los generadores de código no necesitan ser complejos

Toda esta charla de *activo esto y pasivo eso* puede dejarte con la impresión de que los generadores de código son bestias complejas. No necesitan 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 vuelve

9. *solo cómo?* Se dedica a crear código a partir de un esquema de base de datos? Hay varias formas. Si el esquema se mantiene en un archivo plano (por ejemplo, como crear mesas declaraciones), entonces un script relativamente simple puede analizarlo y generar la fuente. Alternativamente, si usa 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 independiente del 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.



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

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 es 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 la duplicación*, página 26 *EI*
- *poder del texto sin formato*, página 73
- *Magos malvados*, página 198
- *Automatización ubicua*, página 230

Ejercicios

- 13** Escriba un generador de código que tome el archivo de entrada de la figura 3.4 y genere una salida en dos idiomas de su elección. Intenta que sea fácil agregar nuevos idiomas.

Capítulo 4

Paranoia pragmática

TIP30

No puedes escribir software perfecto

¿Dolio? no debería Acéptalo como un axioma de vida. Abrázalo. Celebrarlo. Porque el software perfecto no existe. Nadie en la breve historia de la computación ha escrito jamás una pieza de software perfecta. Es poco probable que seas el primero. Y a menos que acepte esto como un hecho, terminará perdiendo tiempo y energía persiguiendo un sueño imposible.

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

Todos saben que personalmente son los únicos buenos conductores en la Tierra. El resto del mundo está ahí afuera para atraparlos, pasando por alto las señales de alto, zigzagueando entre carriles, sin indicar giros, hablando por teléfono, leyendo el periódico y, en general, no cumpliendo con 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. Constantemente interactuamos con el código de otras personas (código que podría no estar a la altura de nuestros altos estándares) y lidiamos con entradas que pueden o no ser válidas. Así que se nos enseña a programar a la defensiva. Si hay alguna duda, validamos toda la información que nos dan. Usamos aserciones para detectar datos incorrectos. Verificamos la consistencia, ponemos restricciones en las columnas de la base de datos y, en general, nos sentimos bastante bien con nosotros mismos.

Pero los programadores pragmáticos llevan esto un paso más allá. *Ellos tampoco confían en sí mismos.* Sabiendo que nadie escribe un código perfecto, incluidos ellos mismos, los programadores pragmáticos codifican para defenderse de sus propios errores. Describimos la primera medida defensiva en *Diseño por contrato*: clientes y proveedores deben acordar derechos y responsabilidades.

En *Los programas muertos no cuentan mentiras*, queremos asegurarnos de no causar daños mientras solucionamos los errores. Por lo tanto, tratamos de verificar las cosas con frecuencia y terminamos el programa si las cosas salen mal.

Programación Asertiva describe un método fácil de verificar en el camino: escriba 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 *Cuándo usar excepciones*.

A medida que sus programas se vuelven más dinámicos, se encontrará haciendo malabarismos con los recursos del sistema: memoria, archivos, dispositivos y similares. En *Cómo equilibrar los recursos*, le 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.

Cuando todo el mundo en realidad es para atraparte, la paranoia es solo un buen pensamiento.

► **Woody Allen**

21

Diseño por contrato

Nada asombra tanto a los hombres como el sentido común y el trato sencillo.

► **Ralph Waldo Emerson, *Ensayos***

Tratar con los sistemas informáticos es difícil. Tratar con la gente 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 ideado durante los últimos milenios también se pueden aplicar a la escritura de software. Una de las mejores soluciones para garantizar un trato claro es la *contrato*.

Un contrato define sus derechos y responsabilidades, así como los de la otra parte. Además, existe un acuerdo sobre las repercusiones si cualquiera de las partes no cumple con el contrato.

Tal vez tenga un contrato de trabajo que especifique las horas que trabajará y las reglas de conducta que debe 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 usar el mismo concepto para ayudar a los módulos de software a interactuar? La respuesta es sí."

DBC

Bertrand Meyer [Mey97b] desarrolló el concepto de *Diseño por contrato* para la lengua Eiffel.¹ Es una técnica simple pero poderosa que se enfoca en documentar (y aceptar) los derechos y responsabilidades de los módulos de software para garantizar 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 *Diseño por contrato* (DBC, para abbreviar).

Cada función y método en un sistema de software *hace algo*. Antes de que empiece *eso* *alguna cosa*, la rutina puede tener alguna expectativa del estado del mundo y puede hacer una declaración sobre el estado del mundo cuando concluye. 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 obtener más información sobre Eiffel, consulte [URL 10] y [URL 11].

- **Condiciones previas.** Qué debe ser cierto para que se llame a la rutina; 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 (consulte el cuadro en la página 115).
- **Postcondiciones.** Lo que se garantiza que hará la rutina; el estado del mundo cuando se hace la rutina. El hecho de que la rutina tenga una poscondición implica que *voluntad* de conclusión: no se permiten bucles infinitos.
- **Invariantes de clase.** Una clase asegura que esta condición sea siempre verdadera desde la perspectiva de una persona que llama. Durante el procesamiento interno de una rutina, es posible que el invariante no se cumpla, pero para cuando la rutina sale y el control regresa al llamador, el invariante debe ser verdadero. (Tenga en cuenta que una clase no puede otorgar acceso de escritura sin restricciones a ningún miembro de datos que participe en el invariante).

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

```
/**
 * @invariant para todos los Nodos n en elementos() /
 *      n.prev() != nulo
 *      implica
 *          n.value().compareTo(n.prev().value()) > 0
 */

clase pública lista_dbc {
    /**
     * @pre contiene(unNodo) == falso
     * @post contiene(unNodo) == verdadero
     */
    vacío público insertarNodo(final Nodo aNodo) {
        //...
    }
}
```

Aquí estamos diciendo que los nodos en 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.

Escriba estas condiciones previas, condiciones posteriores e invariantes en el lenguaje de programación de destino, quizás con algunas extensiones. Por ejemplo, iContract proporciona operadores lógicos predicados: para todo, existe, y implica—además de las construcciones normales de Java. Sus afirmaciones pueden consultar el estado de cualquier objeto al que pueda acceder el método, pero asegúrese de que la consulta no tenga efectos secundarios (consulte la página 124).

DBC y parámetros constantes

A menudo, una condición posterior utilizará 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 `Javafinal` para indicar nuestras intenciones de que el parámetro no se cambie dentro del método. Esto no es infalible, las subclases son libres de volver a declarar el parámetro como no final. Alternativamente, puede usar la sintaxis de `iContractvariable@prepara` obtener el valor original de la variable tal como existía al ingresar al método.

El contrato entre una rutina y cualquier llamador potencial se puede leer como

Si la persona que llama cumple todas las condiciones previas de la rutina, la rutina garantizará que todas las condiciones posteriores e invariantes serán verdaderas cuando se complete.

Si cualquiera 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 finaliza, por ejemplo. Pase lo que pase, no se equivoque, 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.

TIP31

Diseño con Contratos

En *ortogonalidad*, 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. Recuerde, si su contrato indica que aceptará cualquier cosa y prometerá el mundo a cambio, ¡entonces tiene mucho código para escribir!

La herencia y el polimorfismo son las piedras angulares de los lenguajes orientados a objetos y un área donde los contratos realmente pueden brillar. Suponga 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 a la *Principio de sustitución de Liskov*[Lis88]:

Las subclases deben poder utilizarse a través de la interfaz de clase base sin necesidad de que el usuario sepa la diferencia.

En otras palabras, desea asegurarse de que el nuevo subtipo que ha creado realmente "es una especie de" el tipo base, que admita los mismos métodos y que los métodos tengan 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 todas las subclases futuras. Una subclase puede, opcionalmente, aceptar una gama más amplia de entrada o hacer garantías más fuertes. Pero debe aceptar al menos tanto y garantizar tanto como su matriz.

Por ejemplo, considere la clase base de Java.awt.Component. Puede tratar cualquier componente visual en AWT o Swing como unComponente, sin saber que la subclase real es un botón, un lienzo, un menú o lo que sea. Cada componente individual puede proporcionar una funcionalidad específica adicional, pero tiene que proporcionar al menos las capacidades básicas definidas porComponente. Pero no hay nada que le impida crear un subtipo de Componente que proporciona métodos correctamente nombrados que hacen lo incorrecto. Puedes crear fácilmente unpaintmétodo que no pinta, o un establecer fuente método que no establece la fuente. AWT no tiene contratos para detectar el hecho de que no cumplió con el acuerdo.

Sin un contrato, todo lo que el compilador puede hacer es asegurarse de que una subclase se ajuste a la firma de un método en particular. Pero si implementamos un contrato de clase base, ahora podemos asegurarnos de que cualquier subclase futura no pueda alterar el significado de nuestros métodos. Por ejemplo, es posible que desee establecer un contrato para establecer fuente como la siguiente, lo que garantiza que la fuente que establezca sea la fuente que obtenga:

```
/**  
 * @pre f != nulo  
 * @post getFont() == f  
 */  
vacío público establecer fuente (finalFuente f) {  
    //...
```

Implementando DBC

El mayor beneficio de usar DBC puede ser que pone en primer plano el tema de los requisitos y las garantías. Simplemente enumerando en el momento del diseño cuál es el rango del dominio de entrada, cuáles son las condiciones de contorno y qué promete entregar la rutina, o, más importante,

significativamente, lo que *nopromesa* de entregar— es un gran paso adelante en la escritura de un mejor software. Al no decir estas cosas, vuelves a *programación por coincidencia*(consulte la página 172), que es donde muchos proyectos comienzan, terminan y fracasan.

En 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, un *diseño* técnica. Incluso sin verificación automática, puede poner el contrato en el código como comentarios y aún así obtener un beneficio muy real. Al menos, los contratos comentados le brindan un lugar para comenzar a buscar cuando surjan 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. Puede emular esto parcialmente en algunos idiomas usando *afirmaciones*(ver *Programación Asertiva*, página 122). ¿Por qué solo parcialmente? ¿No puedes usar aserciones para hacer todo lo que DBC puede hacer?

Desafortunadamente, la respuesta es no. Para empezar, no hay soporte para propagar aserciones a lo largo de una jerarquía de herencia. Esto significa que si reemplaza un método de clase base que tiene un contrato, las aserciones que implementan ese contrato no se llamarán correctamente (a menos que las duplique manualmente en el nuevo código). Debe recordar llamar la invariante de clase (y todas las invariantes de clase base) manualmente antes de salir de cada método. El problema básico es que el contrato no se ejecuta automáticamente.

Además, no existe un concepto integrado 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 solo puede hacer referencia a "*variable@pre*," o con Eiffel, que apoya "antiguo *expresión*."

Finalmente, el sistema de tiempo de ejecución y las bibliotecas no están diseñados para admitir contratos, por lo que estas llamadas no se verifican. 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 *Los programas muertos no cuentan mentiras*, página 120 para una discusión más detallada).

Ayuda de idioma

Los lenguajes que cuentan con soporte integrado de DBC (como Eiffel y Sather [URL 12]) verifican las condiciones previas y posteriores automáticamente en el compilador y el sistema de tiempo de ejecución. Obtiene el mayor beneficio en este caso porque *todos* del código base (también las bibliotecas) deben cumplir sus contratos.

Pero, ¿qué pasa con los lenguajes más populares como C, C++ y Java? Para estos lenguajes, existen preprocesadores que procesan contratos incrustados en el código fuente original como comentarios especiales. El preprocesador expande estos comentarios al código que verifica las afirmaciones.

Para C y C++, es posible que desee investigarnana [URL 18].nanano maneja la herencia, pero usa el depurador en tiempo de ejecución para monitorear 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 asercción incluida.

Los preprocesadores no son tan buenos como una instalación integrada. Puede ser complicado integrarlos en su proyecto, y otras bibliotecas que use no tendrán contratos. Pero aún pueden ser muy útiles; cuando se descubre un problema de esta manera, especialmente uno que le gustaría *nunca* haber encontrado, es casi como magia.

DBC y Crashing Early

DBC encaja muy bien con nuestro concepto de bloqueo temprano (ver *Los programas muertos no cuentan mentiras*, página 120). Suponga que tiene un método que calcula raíces cuadradas (como en la clase EiffelDOBLE). Necesita una condición previa que restrinja el dominio a números positivos. Se declara una condición previa de Eiffel con la palabra claverequerir,y se declara una poscondición conasegurar,para que puedas escribir

```

sqrt:    DOBLEes
        -- Rutina de raíz cuadrada
requerir
        sqrt_arg_must_be_positive: actual >= 0;
.....
        -- calcule la raíz cuadrada aquí
.....
asegurar
        ((Resultado*Resultado) - Actual).abs <= epsilon*Actual.abs;
        -- El resultado debe estar dentro de la tolerancia de error
final;
```

¿Quién es responsable?

¿Quién es responsable de verificar la condición previa, la persona que llama o la rutina que se llama? Cuando se implementa como parte del lenguaje, la respuesta es ninguna: la condición previa se prueba entre bastidores después de que la persona que llama invoque la rutina, pero antes de que se ingrese la rutina misma. Por lo tanto, si hay alguna verificación explícita de los parámetros a realizar, debe ser realizada por el *llamador*, porque la rutina en sí nunca verá parámetros que violen su condición previa. (Para los idiomas sin compatibilidad integrada, deberá poner entre corchetes el *llamó*rutina con un preámbulo y/o un epílogo que verifica estas afirmaciones).

Considere un programa que lee un número de la consola, calcula su raíz cuadrada (*llamando*sqr $\sqrt{}$) e imprime el resultado. *lossqrt*
La función 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 asqr $\sqrt{}$. Este código de llamada tiene muchas opciones: podría terminar, podría emitir una advertencia y leer otro número, o podría hacer que el número sea positivo y agregar un “ ” al resultado devuelto porsqr $\sqrt{}$. Cualquiera que sea su elección, esto definitivamente no es sqr $\sqrt{}$ ’ problema

Expresando el dominio de la función raíz cuadrada en la condición previa de la sqrtrutina, transfiere la carga de la corrección a la persona que llama, donde pertenece. A continuación, puede diseñar el sqrtrutina segura en el conocimiento de que su entrada *voluntad* esté dentro del alcance.

Si su algoritmo para calcular la raíz cuadrada falla (o no está dentro de la tolerancia de error especificada), recibe un mensaje de error y un seguimiento de la pila para mostrarle la cadena de llamadas.

si pasa sqr $\sqrt{}$ un parámetro negativo, el tiempo de ejecución de Eiffel imprime el error “sqr $\sqrt{}$ _arg_must_be_positive,” junto con un seguimiento de la pila. Esto es mejor que la alternativa en lenguajes como Java, C y C++, donde pasar un número negativo a sqr $\sqrt{}$ devuelve el valor especial NaN (No un número). Puede ser algún tiempo más tarde en el programa que intente hacer algunos cálculos en Yaya, con resultados sorprendentes.

Es mucho más fácil encontrar y diagnosticar el problema bloqueando temprano, en el sitio del problema.

Otros usos de las invariantes

Hasta ahora hemos discutido las condiciones previas y posteriores que se aplican a métodos individuales e invariantes que se aplican a todos los métodos dentro de una clase, pero hay otras formas útiles de usar invariantes.

Invariantes de bucle

Obtener las condiciones de contorno correctas en un ciclo no trivial puede ser problemático. Los bucles están sujetos al problema de la banana (sé cómo se escribe “banana”, pero no sé cuándo parar), los errores de los postes de la cerca (no saber si contar los postes de la cerca o los espacios entre ellos) y el omnipresente “off”. por uno” error [URL 52].

Los invariantes pueden ayudar en estas situaciones: *a bucle invariante* es una declaración del objetivo final de un ciclo, pero se generaliza para que también sea válida antes de que se ejecute el ciclo y en cada iteración a través del ciclo. 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.

```
En tm = matriz[0];           // el ejemplo asume arr.length > 0
En tyo = 1;
// Bucle invariable: m = max(arr[0:i-1]) tiempo(i <
arr.longitud) {
    m = Math.max(m, arr[i]); tyo = tyo + 1;
}
```

(*arr[m:n]* es una conveniencia notacional que significa una porción de la matriz del índice *metroanorte*.) El invariante debe ser verdadero antes de que se ejecute el bucle, y el cuerpo del bucle debe garantizar que siga siendo verdadero mientras se ejecuta el bucle. De esta forma sabemos que el invariante también se cumple cuando termina el ciclo y, por lo tanto, nuestro resultado es válido. Los bucles invariantes se pueden codificar explícitamente como aserciones, pero también son útiles como herramientas de diseño y documentación.

Invariantes semánticas

Puedes usar *invariantes semánticas* 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 al usuario de una tarjeta de débito nunca se le aplicara la misma transacción a su cuenta dos veces. En otras palabras, no importa qué

tipo de modo de falla puede ocurrir, el error debe estar en el lado de *no* procesar 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 meras políticas que podrían cambiar con un nuevo régimen de gestión. Por eso usamos el término *semántico* invariantes—debe ser fundamental para el mismo *sentido* de una cosa, y no sujeto a los caprichos de la política (que es para lo que están las reglas comerciales 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 nota grande en la pizarra común que todos ven. Trate de expresarlo de manera clara y sin ambigüedades. Por ejemplo, en el ejemplo de la tarjeta de débito, podríamos escribir

MÍRRA 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 contratos como especificaciones fijas e inmutables. Pero en el panorama de los agentes autónomos, este no tiene por qué ser el caso. Según la definición de “autónomo”, los agentes son libres de *rechazar* peticiones que no quieren honrar. Son libres de renegociar el 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 tiene una crítica dependencia de arreglos 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 diseñe una pieza de software, diseñe también su contrato.

Las secciones relacionadas incluyen:

- *ortogonalidad*, página 34
- *Los programas muertos no cuentan mentiras*, página 120
- *Programación Asertiva*, página 122 *Cómo equilibrar los recursos*, página 129 *El desacoplamiento y la Ley de Deméter*, página 138 *Acoplamiento temporal*, página 150
- *Programación por Coincidencia*, página 172 *Código que es fácil de probar*, página 189 *Equipos pragmáticos*, página 224
-

Desafíos

- Puntos para reflexionar: si DBC es tan poderoso, ¿por qué no se usa más ampliamente? ¿Es difícil llegar al contrato? ¿Te hace pensar en temas que preferirías ignorar por ahora? ¿Te obliga a pensar!? Claramente, esta es una herramienta peligrosa!

Ejercicios

14 ¿Qué hace un buen contrato? Cualquiera puede agregar condiciones previas y posteriores, pero ¿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 CUERDA a una lista circular doblemente enlazada (recuerde que las condiciones previas están etiquetadas con requerir, y poscondiciones con asegurar).

```
-- Añadir un artículo único a una lista doblemente enlazada,
-- y devolver el NODO recién creado.

add_item(elemento: CADENA) : NODOes
    requerir
        artículo /= Vacío
        find_item(elemento) = Vacío
    asegurar
        resultado.siguiiente.anterior = resultado--Revisa el recién
        resultado.anterior.siguiiente = resultado--Enlaces de nodos añadidos.
        find_item(elemento) = resultado
    final
```

- 15.**A continuación, probemos un ejemplo en Java, algo similar al ejemplo del Ejercicio 14.insertarNúmeroinserta un entero en una lista ordenada. Las condiciones previas y posteriores se etiquetan como en iContract (ver [URL 17]).

Responder
en P. 288

```
interno privado datos[];
/**
 * @post datos[índice-1] < datos[índice] &&
 *       datos[índice] == unValor
 */
público Nodo insertNumber (int final un valor) {
```

En tindex = findPlaceToInsert(aValue); ...

- diecisésis.**Aquí hay un fragmento de una clase de pila en Java. ¿Es este un buen contrato?

Responder
en P. 289

```
/** 
 * @pre anItem != null           // Requerir datos reales
 * @post pop() == anItem // Verifica que sea
 *       // en la pila
 */
vacío público o empajar(final Cadena de un elemento)
```

- 17.**Los ejemplos clásicos de DBC (como en los Ejercicios 14–16) muestran una implementación de un ADT (Tipo de datos abstracto), generalmente una pila o una cola. Pero no mucha gente realmente escribe este tipo de clases de bajo nivel.

Responder
en P. 289

Entonces, para este ejercicio, diseñe una interfaz para una licuadora de cocina. Eventualmente será una licuadora basada en la Web, habilitada para Internet y compatible con CORBA, pero por ahora solo necesitamos la interfaz para controlarla. Tiene diez configuraciones de velocidad (0 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).

Aquí están los métodos. Agregue condiciones previas y posteriores apropiadas y un invariante.

```
En to obtenerVelocidad()
vacío velocidad fijada(En t X)
booleano está lleno()
vacío llenar()
vacío vacío()
```

- 18.**cuantos numeros hay en la serie

0, 5, 10, 15, ..., 100?

Responder
en P. 290

22

Los programas muertos no cuentan mentiras

¿Has notado que a veces otras personas pueden detectar que las cosas no van bien contigo antes de que tú mismo te des cuenta del problema? Es lo mismo con el código de otras personas. Si algo empieza 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 extraviado hizo que sobreescrivíramos un identificador de archivo con algo sin sentido. La próxima llamada aleerlo atrapará. Tal vez una saturación del búfer ha destrozado un contador que estamos a punto de usar para determinar cuánta memoria asignar. Tal vez obtengamos un fracaso demalloc. Un error lógico hace un par de millones de instrucciones significa que el selector para una declaración de caso ya no es el 1, 2 o 3 esperado. Presionaremos el defectocase (que es una de las razones por las que todas y cada una de las sentencias case/switch deben tener una cláusula predeterminada: queremos saber cuándo ha ocurrido lo "imposible").

Es fácil caer en la mentalidad de "eso no puede suceder". La mayoría de nosotros hemos escrito código que no verificó que un archivo se cerrara con éxito o que una declaración de seguimiento se escribiera como esperábamos. Y en igualdad de condiciones, es probable que no lo necesitáramos: el código en cuestión no fallaría en ninguna condición normal. Pero estamos codificando a la defensiva. Estamos buscando punteros deshonestos en otras partes de nuestro programa destrozando la pila. Estamos comprobando que se cargaron las versiones correctas de las bibliotecas compartidas.

Todos los errores te dan información. Podría convencerse 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.

TIP32

Accidente temprano

Accidente, no basura

Uno de los beneficios de detectar problemas lo antes posible es que puede fallar antes. Y muchas veces, bloquear su programa es lo mejor que puede hacer. La alternativa puede ser continuar, escribiendo corrupto

datos a alguna base de datos vital o ordenar la lavadora en su vigésimo ciclo de centrifugado consecutivo.

El lenguaje y las bibliotecas de Java han adoptado esta filosofía. Cuando sucede algo inesperado dentro del sistema de tiempo de ejecución, lanza unExcepción en tiempo de ejecución. 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 excepción, o si sus bibliotecas no arrojan excepciones, asegúrese de manejar los errores usted mismo. En C, las macros pueden ser muy útiles para esto:

```
# definirCOMPROBAR (LÍNEA, ESPERADO) {En
    trc = LINEA;
    si(rc != ESPERADO)
        ut_abort(_FILE_, _LINE_,           \
                \                                \
                \                                \
                \                                \
                # LÍNEA, rc, ESPERADO);      }

vacíout_abort(carbonizarse*expediente,En ten,carbonizarse*línea,En trc,En tExp) {
    fprintf(stderr, "%s línea %d n%s: esperaba %d, obtuvo %d n",
            archivo, ln, línea, exp, rc);
    salir(1);
}
```

Luego puede envolver llamadas que nunca deberían fallar usando

```
COMPROBAR(estadística("/tmp", &stat_buff), 0);
```

Si falla, recibirá un mensaje escrito paraestándar:

```
fuente.c línea 19
'stat("/tmp", &stat_buff)': se esperaba 0, se obtuvo -1
```

Claramente, a veces es inapropiado simplemente salir de un programa en ejecución. Es posible que haya reclamado recursos que quizás no se liberan, o que necesite escribir mensajes de registro, ordenar transacciones abiertas o interactuar con otros procesos. Las técnicas que discutimos en *Cuándo usar excepciones*, página 125, ayudará 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 termínelo lo antes posible. Un programa muerto normalmente hace mucho menos daño que uno lisiado.

Las secciones relacionadas incluyen:

- *Diseño por contrato*, página 109
- *Cuándo usar excepciones*, página 125

23

Programación Asertiva

Hay un lujo en el autorreproche. Cuando nos culpamos a nosotros mismos, sentimos que nadie más tiene derecho a culparnos.

► 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 informática, una creencia central que aprendemos a aplicar a los requisitos, diseños, código, comentarios, casi todo lo que hacemos. Va

TLO SUYO NUNCA PUEDE SUCEDER ...

"Este código no se utilizará dentro de 30 años, por lo que las fechas de dos dígitos están bien". "Esta aplicación nunca se utilizará en el extranjero, entonces, ¿por qué internacionalizarla?" "contar no puede ser negativo." "Este imprimirlo puede fallar."

No practiquemos este tipo de autoengaño, particularmente al codificar.

TIP33

Si no puede suceder, use afirmaciones para asegurarse de que no sucederá

Cada vez que se encuentre pensando "pero, por supuesto, eso nunca podría suceder", agregue código para verificarlo. La forma más fácil de hacer esto es con aserciones. En la mayoría de las implementaciones de C y C++, encontrará alguna forma de afirmarlo _afirmmacro que comprueba una condición booleana. Estas macros pueden ser invaluables. Si un puntero pasado a su procedimiento nunca debe ser NULO,luego compruébalo:

```
vacioescribirCadena(carbonizarse*cuerda) {
    afirmar (cadena!=NULL); ...
}
```

Las afirmaciones también son comprobaciones útiles del funcionamiento de un algoritmo. Tal vez hayas escrito un algoritmo de clasificación inteligente. Comprueba que funciona:

```
por(En tyo = 0; i < numero_entradas-1; i++) {
    afirmar(ordenado[i] <= ordenado[i+1]);
}
```

Por supuesto, la condición que se pasa a una aserción no debería tener un efecto secundario (ver el recuadro en la página 124). Recuerde también que las aserciones pueden desactivarse en el momento de la compilación; nunca ponga código que deba ser ejecutado en unafirmar.

No use aserciones en lugar del manejo real de errores. Las aserciones verifican cosas que nunca deberían suceder: no desea escribir código como

```
imprimirf("Ingrese 'Y' o 'N': "); ch = obtener  
char();  
afirmar ((ch == 'Y') || (ch == 'N'));  
/* ¡mala idea! */
```

Y solo porque el suministrado afirmar llamada de macro salida cuando una aserción falla, no hay razón por la que las versiones que escribas deban hacerlo. Si necesita liberar recursos, haga que un error de aserción genere una excepción, largojmpa un punto de salida o llame a un controlador de errores. Solo asegúrese de que el código que ejecuta en esos últimos milisegundos no se base en la información que desencadenó la falla de afirmación en primer lugar.

Deje las aserciones activadas

Existe un malentendido común acerca de las aserciones, promulgado por las personas que escriben compiladores y entornos de lenguaje. Es algo parecido a esto:

Las aserciones agregan algo de sobrecarga al código. Debido a que verifican cosas que nunca deberían suceder, solo se activarán por un error en el código. Una vez que el código ha sido probado y enviado, ya no son necesarios y deben desactivarse para que el código se ejecute más rápido. Las afirmaciones son una función de depuración.

Aquí hay dos supuestos evidentemente erróneos. Primero, 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 por las que pasará su código (ver *Pruebas despiadadas*, página 245). En segundo lugar, los optimistas se olvidan de que su programa se ejecuta en un mundo peligroso. Durante las pruebas, las ratas probablemente no roerán un cable de comunicaciones, alguien que esté jugando no agotará la memoria y los archivos de registro no llenarán el disco duro. Estas cosas pueden suceder cuando su programa se ejecuta en un entorno de producción. Su primera línea de defensa es verificar cualquier posible error, y la segunda es usar aserciones para tratar de detectar aquellos que se le pasaron por alto.

Desactivar afirmaciones cuando entrega un programa a producción es como cruzar un cable alto sin una red porque una vez lo logró en la práctica. Hay un valor dramático, pero es difícil obtener un seguro de vida.

Incluso si tú *hacerte* tiene problemas de rendimiento, apague solo aquellas afirmaciones que realmente lo golpean. El ejemplo de clasificación anterior puede ser una parte crítica de

Afirmaciones y efectos secundarios

Es vergonzoso cuando el código que agregamos para detectar errores en realidad termina creando nuevos errores. Esto puede suceder con las afirmaciones si la evaluación de la condición tiene efectos secundarios. Por ejemplo, en Java sería una mala idea codificar algo como

```
tiempo(iter.hasMoreElements()) {  
    Test ASSERT(iter.nextElement() Object obj =      !=  nulo);  
    iter.nextElement();  
    //....  
}
```

Los `.siguienteElemento()` llamar en el AFIRMAR 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

```
tiempo(iter.hasMoreElements()) {  
    Objeto obj = iter.nextElement();  
    Prueba ASSERT(obj !=nulo);  
    //....  
}
```

Este problema es una especie de "Heisenbug", una depuración que cambia el comportamiento del sistema que se está depurando (ver [URL 52]).

su solicitud, y es posible que deba ser rápido. Agregar la verificación significa otra pasada 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:

- depuración, página 90 *Diseño por contrato*,
- página 109 *Cómo equilibrar los recursos*,
- página 129 *Programación por Coincidencia*,
- página 172

2. En lenguajes basados en C, puede usar el preprocesador o usarside declaraciones para hacer aserciones opcionales. Muchas implementaciones desactivan la generación de código para la afirmar macro si se establece (o no se establece) un indicador de tiempo de compilación. De lo contrario, puede colocar el código dentro de un side declaración con una condición constante, que muchos compiladores (incluidos los sistemas Java más comunes) optimizarán.

Ejercicios

19 Una revisión rápida de la realidad. ¿Cuál de estas cosas "imposibles" puede suceder?

Responder
en P. 290

1. Un mes con menos de 28 días
2. estadística(".",&sb) == -1 (es decir, no 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. ≠ 180°
5. Un minuto que no tiene 60 segundos
6. En Java: (un + 1) <= un

20 Desarrolle una clase de verificación de aserciones simple para Java.

Responder
en P. 291

24

Cuándo usar excepciones

En *Los programas muertos no cuentan mentiras*, página 120, sugerimos que es una buena práctica verificar todos los errores posibles, en particular 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 que se parece a lo siguiente:

```
retcode = OK;
si(socket.read(nombre) != OK) {
    retcode = BAD_READ;
}
más{
    procesoNombre(nombre);
    si(socket.read(dirección) != OK) {
        retcode = BAD_READ;
    }
    más{
        direcciónDeProceso(dirección); si(
            socket.read(telNo) != OK) {
                retcode = MALA_LECTURA;
            }
        más {
            // etc, etc...
        }
    }
}
devolver código de retorno;
```

Afortunadamente, si el lenguaje de programación admite excepciones, puede reescribir este código de una forma mucho más ordenada:

```

retcode = OK;
probar{
    socket.read(nombre);
    nombre del proceso;
    socket.read(dirección);
    direcciónDeProceso(dirección);
    socket.read(número de teléfono);
    // etcétera etcétera...
}
captura(IOExcepción e) {
    retcode = BAD_READ;
    Registrador.log("Error al leer individuo: "+e.getMessage());
}
devolvercódigo de retorno;

```

El flujo de control normal ahora es claro, con todo el manejo de errores trasladado 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. Suponga 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 su código intenta abrir un archivo para lectura y ese archivo no existe, ¿debería generarse una excepción?

Nuestra respuesta es: "Depende". Si el archivo *debería* han estado allí, entonces se justifica una excepción. Ocurrió algo inesperado: un archivo que esperaba que existiera parece haber desaparecido. Por otro lado, si no tiene idea de si el archivo debe existir o no, entonces no parece excepcional si no puede encontrarlo, y un retorno de error es apropiado.

Veamos un ejemplo del primer caso. El siguiente código abre el archivo /etc/contraseña, que debería existir en todos los sistemas Unix. Si falla, pasa el Excepción de archivo no encontrado a su llamador.

```

vacío público abrir_contraseña()lanzaExcepción de archivo no encontrado {
    // Esto puede generar una excepción FileNotFoundException...
    ipstream =nuevoArchivoEntradaFlujo("/etc/contraseña");
    //...
}

```

Sin embargo, el segundo caso puede implicar abrir un archivo especificado por el usuario en la línea de comando. Aquí no se justifica una excepción, y el código se ve diferente:

```
booleano público open_user_file (nombre de cadena)
    lanzaExcepción de archivo no encontrado {
        Archivo f =nuevoNombre del archivo);
        si(!f.existe()) {
            falso retorno;
        }
        ipstream =nuevoFileInputStream(f); volver verdadero;
    }
```

Tenga en cuenta que el `FileInputStream` la llamada aún puede generar una excepción, que la rutina transmite. No obstante, la excepción se generará únicamente en circunstancias verdaderamente excepcionales; simplemente intentar abrir un archivo que no existe generará un retorno de error convencional.

TIP34

Usar excepciones para problemas excepcionales

¿Por qué sugerimos este enfoque para las excepciones? Bueno, una excepción representa una transferencia de control inmediata, no local, es una especie de cascada. Los programas que usan excepciones como parte de su procesamiento normal sufren todos los problemas de legibilidad y mantenibilidad del código spaghetti clásico. Estos programas rompen la encapsulación: las rutinas 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 que se llama cuando se detecta un error. Puede registrar una rutina para manejar una categoría específica de errores. Cuando se produce uno de estos errores, se llamará al controlador.

Hay momentos en los que es posible que desee utilizar controladores de errores, ya sea en lugar de las excepciones o junto con ellas. Claramente, si está utilizando un lenguaje como C, que no admite excepciones, esta es una de sus pocas opciones (vea 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 manejo de excepciones integrado.

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 manejar unExcepción Remota. Agregar código para manejar estas excepciones puede volverse tedioso y significa que es difícil escribir código que funcione con rutinas locales y remotas. Una posible solución es envolver sus objetos remotos en una clase que no sea remota. Luego, esta clase implementa una interfaz de manejo de errores, lo que permite que el código del cliente registre una rutina para llamarla cuando se detecte una excepción remota.

Las secciones relacionadas incluyen:

- *Los programas muertos no cuentan 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 se queden huérfanos? ¿Tiene sentido usar este tipo de solución cada vez que codifica en C?

Ejercicios

*Responder
en P. 292*

21 Al diseñar una nueva clase de contenedor, identifica las siguientes condiciones de error posibles:

1. No hay memoria disponible para un nuevo elemento en `elagregarrutina`
2. Entrada solicitada no encontrada en `elbuscarrutina`
3. `nulopuntero` pasado a `elagregarrutina`

¿Cómo se debe manejar cada uno? ¿Debe generarse un error, debe generarse una excepción o debe ignorarse la condición?

25

Cómo equilibrar los recursos

“Te traje a este mundo”, decía mi padre, “y puedo sacarte. No hace ninguna diferencia para mí. Voy a hacer otro 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: usted asigna el recurso, lo usa y luego lo desasigna.

Sin embargo, muchos desarrolladores no tienen un plan consistente para lidiar con la asignación y desasignación de recursos. Así que permítanos sugerir un consejo simple:

TIP35

Termina lo que empiezas

Este consejo es fácil de aplicar en la mayoría de las circunstancias. Simplemente significa que la rutina o el objeto que asigna un recurso debe ser responsable de desasignarlo. Veamos cómo se aplica mirando 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 manejo de errores para que el ejemplo sea más claro.

```

vacío leerCliente(carácter constante*fNombre, Cliente *cRec) {
    cArchivo = fopen(fNombre, "r+"); fread(cRec, tamaño de
    (*cRec), 1, cArchivo);
}

vacío escribirCliente(Cliente *cRec) {
    rebobinar(cArchivo);
    fwrite(cRec, tamaño de(*cRec), 1, cArchivo); fclose(cArchivo);

}

vacío actualizarCliente(carácter constante*fNombre, doble nuevo equilibrio) {
    Cliente cRec;
    leerCliente(fNombre, &cRec); cRec.saldo =
    nuevoSaldo; escribirCliente(&cRec);

}

```

A primera vista, la rutina `actualizarCliente` ve bastante bien. Parece implementar la lógica que requerimos: leer un registro, actualizar el saldo y volver a escribir el registro. Sin embargo, esta pulcritud esconde una

problema importante. las rutinas leercliente y escribircliente están estrechamente acoplados³—comparten la variable globalcArchivo. leerCliente abre el archivo y almacena el puntero del archivo encarchivo, y escribircliente usa ese puntero almacenado para cerrar el archivo cuando finaliza. Esta variable global ni siquiera aparece en elactualizarcliente rutina.

¿Por qué es esto 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. Ella entra en la fuente y cambiaactualizarCliente:

```
vacíoactualizarCliente(carácter constante*fNombre,doblenuevo equilibrio) {
    Cliente cRec;
    leerCliente(fNombre, &cRec);
    si(nuevoSaldo >= 0.0) {
        cRec.saldo = nuevoSaldo;
        escribirCliente(&cRec);
    }
}
```

Todo parece estar bien durante las pruebas. Sin embargo, cuando el código entra en producción, colapsa después de varias horas, quejándose de *demasiados archivos abiertos*. Porque escribircliente no se llama en algunas circunstancias, el archivo no se cierra.

Una muy mala solución a este problema sería tratar con el especial caso enactualizarCliente:

```
vacíoactualizarCliente(carácter constante*fNombre,doblenuevo equilibrio) {
    Cliente cRec;
    leerCliente(fNombre, &cRec);
    si(nuevoSaldo >= 0.0) {
        cRec.saldo = nuevoSaldo;
        escribirCliente(&cRec);
    }
    más
    fclose(cArchivo);
}
```

Esto solucionará el problema: el archivo ahora se cerrará independientemente del nuevo saldo, pero la solución ahora significa que *Tres* las rutinas se acoplan a través del globalcArchivo. Estamos cayendo en una trampa, y las cosas van a empezar a ir cuesta abajo rápidamente si seguimos en este curso.

3. Para una discusión sobre los peligros del código acoplado, consulte *El desacoplamiento y la Ley de Deméter*, página 138.

los *ftermina lo que empieza* tip nos dice que, idealmente, la rutina que asigna un recurso también debería liberarlo. Podemos aplicarlo aquí refactorizando ligeramente el código:

```
vacíoleerCliente(ARCHIVO *cArchivo, Cliente *cRec) {
    fread(cRec,tamaño de(*cRec), 1, cArchivo);
}

vacíoescribirCliente(ARCHIVO *cArchivo, Cliente *cRec) {
    rebobinar(cArchivo);
    fwrite(cRec,tamaño de(*cRec), 1, cArchivo);
}

vacíoactualizarCliente(carácter constante*fNombre,doblenuevo equilibrio) {
    ARCHIVO *cArchivo;
    Cliente cRec;

    cArchivo = fopen(fNombre, "r+");
    leerCliente(cArchivo, &cRec); //>---
    si(nuevoSaldo // /
        >= 0.0) { // /
        cRec.saldo = nuevoSaldo; // /
        escribirCliente(cArchivo, &cRec); // /
    } // /
    fclose(cArchivo); // <--
}
```

Ahora toda la responsabilidad del archivo está en el `actualizarCliente` rutina. Abre el archivo y (terminando lo que empieza) lo cierra antes de salir. La rutina equilibra el uso del archivo: abrir y cerrar 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 nido

El patrón básico para la asignación de recursos se puede ampliar para las rutinas que necesitan más de un recurso a la vez. Sólo hay dos sugerencias más:

1. Desasignar los recursos en el orden inverso al que los asignó. De esa forma, no dejará huérfanos los recursos si un recurso contiene referencias a otro.
2. Al asignar el mismo conjunto de recursos en diferentes lugares de su código, asígnelos siempre en el mismo orden. Esto reducirá la posibilidad de interbloqueo. (Si el proceso A reclama `recurso1` y está a punto de reclamar `recurso2`, mientras que el proceso B ha reclamado `recurso2` y está tratando de conseguir `recurso1`, los dos procesos esperarán por siempre.)

No importa qué tipo de recursos estemos usando (transacciones, memoria, archivos, subprocessos, ventanas), se aplica el patrón básico: quien

asigna un recurso debe ser responsable de desasignarlo. Sin embargo, en algunos idiomas podemos desarrollar más el concepto.

Objetos y excepciones

El equilibrio entre asignaciones y desasignaciones recuerda al constructor y destructor de una clase. La clase representa un recurso, el constructor le brinda un objeto particular de ese tipo de recurso y el destructor lo elimina de su alcance.

Si está programando en un lenguaje orientado a objetos, puede resultarle útil encapsular recursos en clases. Cada vez que necesita un tipo de recurso en particular, crea una instancia de un objeto de esa clase. Cuando el objeto queda fuera del alcance o el recolector de elementos no utilizados lo reclama, el destructor del objeto desasigna el recurso envuelto.

Este enfoque tiene beneficios particulares cuando se trabaja con lenguajes como C++, donde las excepciones pueden interferir con la desasignación de recursos.

Equilibrio y excepciones

Los idiomas que admiten excepciones pueden dificultar la desasignación de recursos. Si se lanza una excepción, ¿cómo garantiza que todo lo asignado antes de la excepción esté ordenado? La respuesta depende en cierta medida del idioma.

Equilibrio de recursos con excepciones de C++

C++ admite un aprobar ... captura mecanismo de excepción. Desafortunadamente, esto significa que siempre hay al menos dos caminos posibles al salir de una rutina que detecta y luego vuelve a generar una excepción:

```
vacío hacer algo(vacío) {
    Nodo *n = nuevo Nodo;
    probar {
        // hacer algo
    }
    captura (...) {
        Eliminar norte;
        lanzar;
    }
    Eliminar norte;
}
```

Observe que el nodo que creamos se libera en dos lugares: una vez en la ruta de salida normal de la rutina y otra en el controlador de excepciones. Esta es una violación evidente de la *SECOPrinципio* y un problema de mantenimiento esperando a suceder.

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 "norte" de un puntero a un *realNodoobjeto* en la pila:

```
vacíohacerAlgo1(vacío) {  
    Nodo norte;  
    probar {  
        // hacer algo  
    }  
    captura (...) {  
        lanzar;  
    }  
}
```

Aquí confiamos en C++ para manejar la destrucción del *Nodoobjeto* automáticamente, ya sea que se produzca una excepción o no.

Si el cambio desde un puntero no es posible, se puede lograr el mismo efecto envolviendo el recurso (en este caso, *unNodopuntero*) dentro de otra clase.

```
// Clase contenedora para recursos de Nodo clase  
    Recurso de nodo {  
        Nodo * n;  
    público:  
        NodoRecurso() { n =nuevoNodo; } ~RecursoNodo() {  
            EliminarNorte; }  
        Nodo *operator->() { devolverNorte; }  
    };  
vacíohacerAlgo2(vacío) {  
    NodeResource norte;  
    probar {  
        // hacer algo  
    }  
    captura (...) {  
        lanzar;  
    }  
}
```

Ahora la clase contenedora, *recurso de nodo*, asegura que cuando se destruyen sus objetos, también se destruyen los nodos correspondientes. Para mayor comodidad, el contenedor proporciona un operador de desreferenciación *->*, para que sus usuarios puedan acceder a los campos en el contenido *Nodoobjeto* directamente.

Debido a que esta técnica es tan útil, la biblioteca estándar de C++ proporciona la clase de plantilla `auto_ptr`, lo que le brinda envoltorios automáticos para objetos asignados dinámicamente.

```
vacío hacerAlgo3(vacío) {
    auto_ptr<Nodo> p (nuevoNodo);
    // Acceder al Nodo como p->...
    // Nodo borrado automáticamente al final
}
```

Equilibrio de 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 basura, y su finalización llamará al método en caso de que la recolección de basura los reclame. Si bien es conveniente para los desarrolladores, que ya no tienen la culpa de la mayoría de las fugas de memoria, dificulta la implementación de la limpieza de recursos mediante el esquema de C++. Afortunadamente, los diseñadores del lenguaje Java agregaron cuidadosamente una función de lenguaje para compensar, el `finally`. Cuando un `try`-bloque contiene un `finally`-cláusula, se garantiza que el código en esa cláusula se ejecutará si cualquier declaración en el `try`-bloque ejecuta un

`devolver`-el código en el `finally`-se ejecutará la cláusula. Esto significa que podemos equilibrar nuestro uso de recursos con código como

```
vacío público hacer algo() lanza IOException {
    Archivo tmpFile = nuevoArchivo(tmpNombreArchivo); FileWriter
    tmp = nuevoFileWriter(tmpFile);
    probar{
        // Haz algo de trabajo
    }
    finalmente{
        tmpFile.delete();
    }
}
```

La rutina utiliza un archivo temporal, que queremos eliminar, independientemente de cómo finalice la rutina. El `finally`-bloque nos permite expresar esto de manera concisa.

Cuando no puede equilibrar los recursos

Hay momentos en que el patrón básico de asignación de recursos simplemente no es apropiado. Comúnmente esto se encuentra en programas que usan dinámica

estructuras de datos. Una rutina asignará un área de memoria y la vinculará a una estructura más grande, donde puede permanecer por 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 agregados. ¿Qué sucede cuando desasignas la estructura de nivel superior? Tienes tres opciones principales:

1. La estructura de nivel superior también es responsable de liberar las subestructuras que contiene. Estas estructuras luego eliminan recursivamente los datos que contienen, y así sucesivamente.
2. La estructura de nivel superior simplemente se desasigna. Todas las estructuras a las que apuntó (a las que no se hace referencia en ningún otro lugar) quedan 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, debe hacerlo explícito para cada uno e implementar su decisión de manera consistente. Implementar cualquiera de estas opciones en un lenguaje procedural 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 ganchos transversales).

Finalmente, si hacer un seguimiento de los recursos se vuelve complicado, puede escribir su propia forma de recolección de basura automática limitada implementando un esquema de conteo de referencia en sus objetos asignados dinámicamente. El libro *C++ más efectivo*[Mey96] dedica un apartado a este tema.

Comprobación del saldo

Debido a que los programadores pragmáticos no confían en nadie, incluidos nosotros mismos, creemos que siempre es una buena idea crear un código que realmente verifique que los recursos se liberen de manera adecuada. Para la mayoría de las aplicaciones, esto normalmente significa producir envoltorios para cada tipo de recurso y utilizar 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 contenedores para verificar esto.

Por ejemplo, un programa de ejecución prolongada que atiende solicitudes probablemente tendrá un único punto en la parte superior de su ciclo de procesamiento principal donde espera que llegue la siguiente solicitud. Este es un buen lugar para asegurarse de que el uso de recursos no haya aumentado desde la última ejecución del bucle.

En un nivel más bajo, pero no menos útil, puede invertir en herramientas que (entre otras cosas) verifiquen si sus programas en ejecución tienen fugas de memoria. Purificar (www.racional.com) y Seguro++ (www.parasoft.com) son populares opciones

Las secciones relacionadas incluyen:

- *Diseño por contrato*, página 109 *Programación*
- *Asertiva*, página 122 *El desacoplamiento y la Ley de*
- *Deméter*, página 138

Desafíos

- Aunque no hay formas garantizadas de garantizar que siempre libere recursos, ciertas técnicas de diseño, cuando se aplican de manera consistente, ayudarán. En el texto discutimos cómo establecer una invariante semántica para las principales estructuras de datos podría dirigir las decisiones de desasignación de memoria. Considera cómo *Diseño por contrato*, página 109, podría ayudar a refinar esta idea.

Ejercicios

22Algunos desarrolladores de C y C++ insisten en establecer un puntero **paranulo** después de desasignar la memoria a la que hace referencia. ¿Por qué es una buena idea?

23Algunos desarrolladores de Java insisten en establecer una variable de objeto **paranulo** después de que hayan terminado de usar el objeto. ¿Por qué es una buena idea?

Responder
en P. 292

Responder
en P. 292

Capítulo 5

Doblar o romper

La vida no se detiene.

Tampoco el código que escribimos. Para mantenernos al día con el ritmo de cambio casi frenético actual, debemos esforzarnos al máximo para escribir un código que sea lo más flexible posible. De lo contrario, es posible que nuestro código se vuelva obsoleto rápidamente, o que sea demasiado frágil para arreglarlo, y que, en última instancia, se quede atrás en la loca carrera hacia el futuro.

En *Reversibilidad*, en la página 44, hablamos sobre los peligros de las decisiones irreversibles. En este capítulo, le diremos cómo hacer *reversible* decisiones, para que su código pueda permanecer flexible y adaptable frente a un mundo incierto.

Primero tenemos que mirar *acoplamiento*—las dependencias entre módulos de código. En *El desacoplamiento y la Ley de Deméter* mostraremos cómo mantener separados los conceptos separados y disminuir el acoplamiento.

Una buena manera de mantenerse flexible es escribir *menos código*. Cambiar el código lo deja abierto a la posibilidad de introducir nuevos errores. *Metaprogramación* explicará cómo sacar los detalles del código por completo, donde se pueden cambiar de forma más segura y sencilla.

En *Acoplamiento temporal*, veremos dos aspectos del tiempo en relación con el acoplamiento. ¿Dependes de que el “tick” venga antes del “tock”? No si quieres ser flexible.

Un concepto clave en la creación de código flexible es la separación de datos *modelo* a partir de una *vista*, o presentación, de ese modelo. Separaremos los modelos de las vistas en *es solo una vista*.

Finalmente, existe una técnica para desacoplar módulos aún más al proporcionar un lugar de encuentro donde los módulos pueden intercambiar datos de forma anónima y asincrónica. Este es el tema *depizarras*.

Armado con estas técnicas, puede escribir código que "seguirá los golpes".

26

El desacoplamiento y la Ley de Deméter

Buenas vallas hacen buenos vecinos.

► Robert Frost, "Reparando el muro"

En *ortogonalidad*, página 34, y *Diseño por contrato*, página 109, sugerimos que escribir código "tímid" es beneficioso. Pero "tímid" funciona de dos maneras: no te reveles a los demás y no interactúes con demasiada gente.

Espías, disidentes, revolucionarios y otros a menudo se organizan en pequeños grupos de personas llamados *células*. Aunque los individuos de cada celda pueden conocerse entre sí, no tienen conocimiento de los de otras celdas. Si se descubre una célula, ninguna cantidad de suero de la verdad revelará los nombres de otras personas fuera de la célula. Eliminar 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 ellos. Si un módulo se ve comprometido y tiene que ser reemplazado, los otros módulos deberían poder continuar.

Minimizar el acoplamiento

¿Qué tiene de malo tener módulos que se conozcan entre sí? Nada en principio, no necesitamos ser tan paranoicos como espías o disidentes. Sin embargo, debe tener cuidado *con cuantos* otros módulos con los que interactúa y, lo que es más importante, *cómo* viniste a interactuar con ellos.

Suponga que está remodelando su casa o construyendo una casa desde cero. Un arreglo típico involucra a un "contratista general". Usted contrata al contratista para que haga 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 software. Cuando le pedimos a un objeto un servicio en particular, nos gustaría que el servicio se realice en nuestro nombre. Nosotros *no hagaqueremos* que el objeto nos proporcione un objeto de terceros con el que tenemos que tratar para obtener el servicio requerido.

Por ejemplo, suponga que está escribiendo una clase que genera un gráfico de datos de registradores científicos. Tienes registradores de datos repartidos por todo el mundo; cada objeto registrador contiene un objeto de ubicación que proporciona su posición y zona horaria. Desea permitir que sus usuarios seleccionen una grabadora y tracen sus datos, etiquetados con la zona horaria correcta. podrías escribir

```
vacío públicoplotDate(Date aDate, Selection aSelection) {  
    Zona horaria tz =  
        aSelection.getRecorder().getLocation().getTimeZone();  
    ...  
}
```

Pero ahora la rutina de trazado está innecesariamente acoplada a *Tresclases—Selección, grabadora,yUbicación*.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 otra parte del sistema afectesucódigo. Por ejemplo, si Fred hace un cambio enUbicacíontal que ya no contiene directamente un Zona horaria,Tienes que cambiar tu código también.

En lugar de cavar a través de una jerarquía usted mismo, simplemente solicite lo que necesita directamente:

```
vacío públicoplotDate(Date aDate, TimeZone aTz) {  
    ...  
    plotDate(someDate, someSelection.getTimeZone());
```

Agregamos un método paraSelecciónpara obtener la zona horaria en nuestro nombre: a la rutina de trazado no le importa si la zona horaria proviene del Grabadoradirectamente, desde algún objeto contenido dentroGrabadora, o si Selecciónconstituye una zona horaria completamente diferente. La rutina de selección, a su vez, probablemente solo debería preguntarle a la grabadora su zona horaria, dejando que la grabadora la obtenga de su contenido. Ubicaciónobjeto.

Atravesar relaciones entre objetos directamente puede conducir rápidamente a una explosión combinatoria de las relaciones de dependencia. Puede ver los síntomas de este fenómeno de varias maneras:

1. Proyectos grandes de C o C++ donde el comando para vincular una prueba unitaria es más largo que el propio programa de prueba
2. Cambios "simples" a 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 la Ley de Deméter para diseñar nuestros métodos y funciones.

La ley de Deméter para las funciones

La Ley de Demeter 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 en la página siguiente.

Al escribir un código "tímido" que honre la Ley de Deméter tanto como sea posible, podemos lograr nuestro objetivo:

TIP36

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 mayor *conjuntos de respuesta* son más propensas a errores que las clases con conjuntos de respuestas más pequeños (un

1. Si todos los objetos se conocen entre sí, entonces un cambio en un solo objeto puede resultar en el otro. $n - 1$ objetos que necesitan cambios.

Figura 5.1. Ley de Deméter para funciones

```

clase Deméter {
    privado:
        Un *a;
        función int();
    público:
        //...
        ejemplo nulo (B& b);
}
void Demeter::ejemplo(B& b) {
    Cc;
    int f      = función(); ←→ sí mismo
    b.invert(); ←→ cualquier parámetro que se haya pasado al método
    a = nueva A();
    a->establecerActivo(); ←→ cualquier objeto que haya creado
    c.imprimir(); ←→ cualquier objeto componente directamente sostenido
}

```

La Ley de Deméter para funciones establece que cualquier método de un objeto debe llamar solo métodos pertenecientes a:

conjunto de respuestas se define como el número de funciones invocadas directamente por los métodos de la clase).

Debido a que seguir la Ley de Demeter reduce el tamaño del conjunto de respuestas en la clase que llama, 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 Demeter hará que su código sea más adaptable y robusto, pero a un costo: como "contratista general", su módulo debe delegar 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 una gran cantidad de métodos de contenedor que simplemente reenvían la solicitud a un delegado. Estos métodos de envoltorio impondrán tanto un costo de tiempo de ejecución como una sobrecarga de espacio, lo que puede ser significativo, incluso prohibitivo, en algunas aplicaciones.

Como con cualquier técnica, debe sopesar los pros y los contras para su aplicación particular. En el diseño de esquemas de bases de datos, es una práctica común "desnormalizar" el esquema para mejorar el rendimiento: para

Desacoplamiento físico

En esta sección nos ocupamos principalmente del diseño para mantener las cosas lógicamente desacopladas dentro de los sistemas. Sin embargo, existe otro tipo de interdependencia que se vuelve muy importante a medida que los sistemas crecen. En su libro *Diseño de software C++ a gran escala* [Lak96], John Lakos aborda los problemas relacionados con las relaciones entre los archivos, directorios y bibliotecas que componen un sistema. Grandes proyectos que ignoran estos *diseños físicos* 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. El Sr. Lakos argumenta de manera convincente que el diseño lógico y físico debe proceder en conjunto, que deshacer el daño causado a una gran cantidad 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 estrechamente acoplando varios módulos, puede obtener una importante ganancia de rendimiento. Siempre que sea bien conocido y aceptable que esos módulos se acoplen, su diseño está bien.

De lo contrario, puede encontrarse en el camino hacia un futuro frágil e inflexible. O ningún futuro en absoluto.

Las secciones relacionadas incluyen:

- *ortogonalidad*, página 34 *Reversibilidad*,
- página 44 *Diseño por contrato*, página 109
- *Cómo equilibrar los recursos*, página 129 *es*
- *solo una vista*, página 157 *Equipos*
- *pragmáticos*, página 224 *Pruebas*
- *despiadadas*, 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, reduce el acoplamiento. Sin embargo, 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

24Discutimos el concepto de desacoplamiento físico en el recuadro de la página anterior. ¿Cuál de los siguientes archivos de encabezado de C++ está más estrechamente relacionado con el resto del sistema?

Responder
en P. 293

persona1.h:

```
# incluir      "fecha.h"
clase        Persona1 {
    privado:
        fecha mi fecha de nacimiento;
    público:
        Persona1(Fecha &fecha de nacimiento);
        //...
```

persona2.h:

```
clase        Fecha;
clase        Persona2 {
    privado:
        Fecha *miFechaDeNacimiento;
    público:
        Persona2(Fecha &fecha de nacimiento);
        //...
```

25Para 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.

Responder
en P. 293

```
vacío público mostrar Saldo (Cuenta Bancaria) {
    cantidad de dinero = cuenta.obtenersaldo();
    printToScreen(amt.printFormat());
}
```

26Este ejemplo también está en Java.

Responder
en P. 294

```
clase pública Colada {
    privado    licuadora mi licuadora;
    privado    Vector mis cosas;
    público Colada() {
        miBlender =nuevoLicuadora(); mis cosas
        =nuevoVector();
    }
    vacío privado hacer algo() {
        myBlender.addIngredients(myStuff.elements());
    }
}
```

27Este ejemplo está en C++.

Responder
en P. 294

```
vacío procesarTransacción(Cuenta bancaria, En t) {
    Persona que;
    cantidad de dinero;
    cantidad.setValor(123.45);
    cuenta.establecerSaldo(cantidad);
    quien = cuenta.getOwner();
    markWorkflow(quién->nombre(), SET_BALANCE);
}
```

Metaprogramación

Ninguna cantidad de genio puede superar una preocupación por los detalles.

► Octava Ley de Levy

Los detalles estropean nuestro código pristino, especialmente si cambian con frecuencia. Cada vez que tenemos que entrar y cambiar el código para acomodar algún cambio en la lógica comercial, o en la ley, o en los gustos personales de la administración del día, corremos el riesgo de romper el sistema, de introducir un nuevo error.

Entonces decimos "¡fuera los detalles!" Sácalos del código. Mientras estamos en eso, podemos hacer que nuestro código sea altamente configurable y "suave", es decir, que se adapte fácilmente a los cambios.

Configuración dinámica

Primero, queremos que nuestros sistemas sean altamente configurables. No solo cosas como los colores de la pantalla y el texto de aviso, sino elementos profundamente arraigados como la elección de algoritmos, productos de base de datos, tecnología de middleware y estilo de interfaz de usuario. Estos elementos deben implementarse como opciones de configuración, no mediante integración o ingeniería.

TIP37

Configurar, no integrar

Usar *metadatos* 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 tal como lo haría con cualquier otro dato en la base de datos.

Usamos el término en su sentido más amplio. Los metadatos son cualquier dato que describe la aplicación: cómo debe ejecutarse, qué recursos debe usar, etc. Por lo general, se accede a los metadatos y se utilizan en tiempo de ejecución, no en tiempo de compilación. Utiliza metadatos todo el tiempo, al menos sus programas lo hacen. Suponga que hace clic en una opción para ocultar la barra de herramientas en su

Navegador web. 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 usar un mecanismo estándar. En Windows, ya sea un archivo de inicialización (usando el sufijo .ini) o las entradas en el Registro del sistema son típicas. 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, especifica una clave para recuperar un valor. Alternativamente, las implementaciones de metadatos más potentes y flexibles utilizan un lenguaje de secuencias de comandos incorporado (ver *Idiomas de dominio*, página 57, para más detalles).

De hecho, el navegador Netscape ha implementado preferencias utilizando ambas técnicas. En la versión 3, las preferencias se guardaban como simples pares clave/valor:

SHOW_TOOLBAR: Falso

Más tarde, las preferencias de la Versión 4 se parecían más a JavaScript:

```
user_pref("barra de herramientas personalizada.Browser.Navigation_Toolbar.open", false);
```

Aplicaciones impulsadas por metadatos

Pero queremos ir más allá del uso de metadatos para preferencias simples. Queremos configurar e impulsar la aplicación a través de metadatos tanto como sea posible. Nuestro objetivo es pensar declarativamente (especificando *qué* hay que hacer, no *cómo*) y crear programas altamente dinámicos y adaptables. Hacemos esto adoptando una regla general: programe para el caso general y coloque los detalles en otro lugar, fuera de la base del código compilado.

TIP38

Ponga abstracciones en código, detalles en metadatos

Hay varios beneficios de este enfoque:

- Te obliga a desacoplar tu diseño, lo que da como resultado un programa más flexible y adaptable.
- Lo obliga a crear un diseño más sólido y abstracto postergando los detalles, postergándolos por completo fuera del programa.

- Puede personalizar la aplicación sin volver a compilarla. También puede usar este nivel de personalización para proporcionar soluciones sencillas para errores críticos en los sistemas de producción en vivo.
- Los metadatos se pueden expresar de una manera mucho más cercana al dominio del problema de lo que podría ser un lenguaje de programación de propósito general (ver *Idiomas de dominio*, página 57).
- Incluso puede implementar varios proyectos diferentes utilizando el mismo motor de aplicación, pero con diferentes metadatos.

Queremos diferir 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 diseñar una solución que nos permita realizar cambios rápidamente, tenemos una mejor oportunidad de hacer frente a la avalancha de cambios direccionales que inundan muchos proyectos (ver *Reversibilidad*, página 44).

Lógica de negocios

Así que eligió el motor de la base de datos como una opción de configuración y proporcionó metadatos para determinar el estilo de la interfaz de usuario. ¿Podemos hacer más? Definitivamente.

Debido a que es más probable que cambien las políticas y reglas comerciales que cualquier otro aspecto del proyecto, tiene sentido mantenerlas en un formato muy flexible.

Por ejemplo, su aplicación de compras puede incluir varias políticas corporativas. Tal vez le pague a los proveedores pequeños en 45 días y a los grandes en 90 días. Haga que las definiciones de los tipos de proveedores, así como los períodos de tiempo en sí, sean configurables. Aproveche la oportunidad para generalizar.

Tal vez esté escribiendo un sistema con requisitos de flujo de trabajo horrendos. Las acciones se inician y se detienen de acuerdo con reglas comerciales complejas (y cambiantes). Considere codificarlos en algún tipo de sistema basado en reglas (o experto), integrado dentro de 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. Eche un vistazo a la página 58 para ver un ejemplo.

Cuándo configurar

Como se menciona en *El poder del texto sin formato*, página 73, recomendamos representar los metadatos de configuración en texto sin formato; hace la vida mucho más fácil.

Pero, ¿cuándo debería un programa leer esta configuración? Muchos programas escanean tales cosas solo al inicio, lo cual es desafortunado. Si necesita cambiar la configuración, esto lo obliga a reiniciar la aplicación. Un enfoque más flexible es escribir programas que puedan recargar su configuración mientras se ejecutan. Esta flexibilidad tiene un costo: es más compleja de implementar.

Así que considere cómo se usará su aplicación: si se trata de un proceso de servidor de ejecución prolongada, querrá proporcionar alguna forma de volver a leer y aplicar metadatos mientras se ejecuta el programa. Para una pequeña aplicación de GUI de cliente que se reinicia rápidamente, es posible que no necesite hacerlo.

Este fenómeno no se limita al código de la aplicación. A todos nos ha molestado que los sistemas operativos nos obliguen a reiniciar cuando instalamos alguna aplicación sencilla o cambiamos un parámetro inocuo.

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 usar los metadatos tanto para configurar aplicaciones y para reducir la complejidad de escribir código.

Suponga que desea crear un software Java que participará 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. Escribe un *frijol* —un objeto autónomo que sigue ciertas convenciones— y colóquelo en un *contenedor de frijol* que gestiona gran parte de los detalles de bajo nivel en su nombre. Puede escribir el código para un bean sin incluir ninguna operación de transacción o gestión de subprocessos; EJB usa metadatos para especificar cómo deben manejarse las transacciones.

La asignación de subprocessos y el equilibrio de carga se especifican como metadatos para el servicio de transacciones subyacente que utiliza 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 administrar transacciones en nombre del bean en uno de varios estilos diferentes (incluida una opción en la que controla sus propias confirmaciones y reversiones). Todos los parámetros que afectan el comportamiento del bean se especifican en el bean.*descriptor de despliegue*—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 usuarios y desarrolladores que configuran aplicaciones dinámicas. Pero, ¿qué sucede si dejas que las aplicaciones se configuren entre sí con un software que se adapte a su entorno? La configuración no planificada e improvisada del software existente es un concepto poderoso.

Los sistemas operativos ya se configuran a sí mismos en el hardware cuando arrancan, y los navegadores web se actualizan automáticamente con nuevos componentes.

Sus aplicaciones más grandes probablemente 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 ayude.

No escriba el código Dodo

Sin metadatos, su código no es tan adaptable o flexible como podría ser. ¿Esto es 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 rápidamente se extinguío.² Fue la primera extinción documentada de una especie a manos del hombre.

No dejes que tu proyecto (o tu carrera) vaya por el camino del dodo.

2. No ayudó que los colonos vencieran a los plácidos (léase *tonto*) pájaros a muerte con garrotes por deporte.

Las secciones relacionadas incluyen:

- *ortogonalidad*, página 34
- *Reversibilidad*, página 44 *Idiomas de dominio*
- *dominio*, página 57 *El poder del texto*
- *sin formato*, página 73

Desafíos

- Para su proyecto actual, considere qué parte de la aplicación podría trasladarse fuera del programa mismo a los metadatos. ¿Cómo sería el “motor” resultante? ¿Sería capaz de 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 cuál externamente como metadatos?

Responder
en P. 295

1. Asignaciones de puertos de comunicación
2. El 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 uso en pruebas unitarias

Acoplamiento temporal

Que es *acoplamiento temporal* todo sobre, usted puede pedir. Ya es 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 cronograma, 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 (las cosas suceden al mismo tiempo) y el orden (las posiciones relativas de las cosas en el tiempo).

Por lo general, no abordamos la programación con ninguno de estos aspectos en mente. Cuando las personas se sientan 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—*hacer estoy* luego siempre*Haz eso*. Pero pensar de esta manera conduce a *acoplamiento temporal*: acoplamiento en el tiempo. MétodoAsiempre debe llamarse antes del métodoB; solo se puede ejecutar un informe a la vez; debe esperar a que la pantalla se vuelva a dibujar antes de recibir el clic del botón. El tic debe suceder antes que el tac.

Este enfoque no es muy flexible ni muy realista.

Necesitamos permitir la concurrencia 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 *pueden* suceder al mismo tiempo, y lo que debe suceder en un orden estricto. Una forma de hacer esto es capturar su descripción del flujo de trabajo utilizando una notación como la *diagrama de actividad UML*.⁴

3. No entraremos en los detalles de la programación concurrente o paralela aquí; un buen libro de texto de ciencias de la computación debe cubrir los conceptos básicos, incluida la programación, interbloqueo, inanición, exclusión mutua/semáforos, etc.

4. Para obtener más información sobre todos los tipos de diagramas UML, consulte [FS97].

Un diagrama de actividad consta de 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 *barra de sincronización*. Una vez *que todas las* acciones que conducen a una barra de sincronización están completas, luego 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 actividad para maximizar el paralelismo identificando actividades que *podría serse* realizan en paralelo, pero no lo son.

TIP39

Analice el flujo de trabajo para mejorar la concurrencia

Por ejemplo, en nuestro proyecto blender (Ejercicio 17, página 119), los usuarios inicialmente pueden describir su flujo de trabajo actual de la siguiente manera.

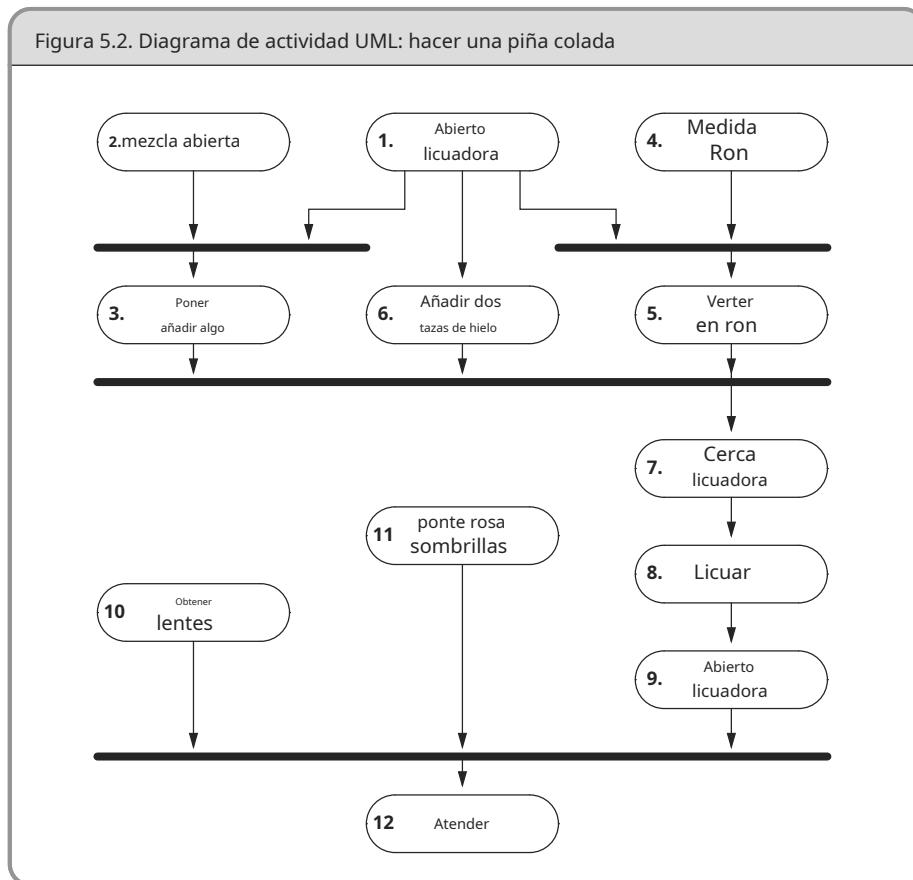
1. Licuadora abierta
2. Mezcla abierta de piña colada
3. Ponga la mezcla en la licuadora
4. Medir 1/2 taza de ron blanco
5. Vierta el ron
6. Agrega 2 tazas de hielo
7. Cierra la licuadora
8. Licuar por 2 minutos
9. Licuadora abierta
10. Consigue gafas
11. Consigue sombrillas rosas
12. Servir

Aunque describen estas acciones en serie, e incluso pueden realizarlas en serie, notamos que muchas de ellas podrían realizarse en paralelo, como mostramos en el diagrama de actividades de la Figura 5.2 en 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 al mismo tiempo, por adelantado. Las tareas 3, 5 y 6 pueden realizarse en paralelo más adelante.

Si participó en un concurso de elaboración de piñas coladas, estas optimizaciones pueden marcar la diferencia.

Figura 5.2. Diagrama de actividad UML: hacer una piña colada



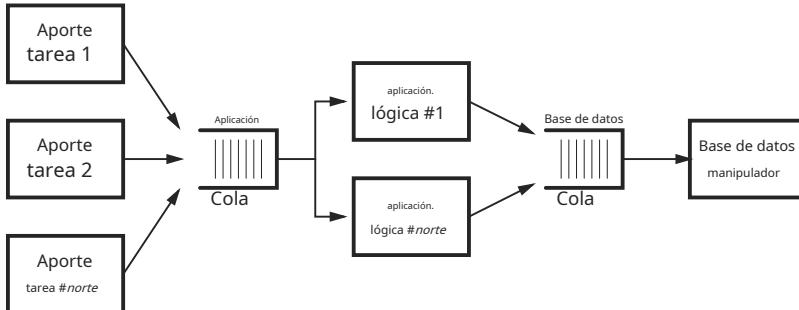
Arquitectura

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 contra la base de datos. Pero escribimos una aplicación distribuida multiproceso 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 lo hizo más fácil escribir. Echemos un vistazo más de cerca a este proyecto.

El sistema recibe solicitudes de una gran cantidad de líneas de comunicación de datos y procesa transacciones contra una base de datos de back-end.

El diseño aborda las siguientes limitaciones:

Figura 5.3. Descripción general de la arquitectura OLTP



- Las operaciones de la 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 la base de datos.
- El rendimiento de la base de datos sufre con demasiadas sesiones simultáneas.
- Múltiples transacciones están en progreso simultáneamente en cada línea de datos.

La solución que nos dio el mejor rendimiento y la arquitectura más limpia se parecía a la Figura 5.3.

Cada cuadro representa un proceso separado; los procesos se comunican a través de colas de trabajo. Cada proceso de entrada supervisa 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 en busca de más tráfico. De manera similar, el servidor de aplicaciones realiza solicitudes al proceso de la base de datos,⁵ 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 *consumidor hambriento* modelo.

5. Aunque mostramos la base de datos como una sola entidad monolítica, no lo es. El software de la base de datos se divide en varios procesos y subprocesos del cliente, pero el software de la base de datos lo maneja internamente y no forma parte de nuestro ejemplo.

En un modelo de consumidor hambriento, reemplaza el programador central con una serie de tareas de consumidor independientes y una cola de trabajo centralizada. Cada tarea del consumidor toma una pieza de la cola de trabajo y continúa con el negocio de procesarla. A medida que cada tarea termina su trabajo, vuelve a la cola por más. De esta manera, si alguna tarea en particular se atasca, las demás pueden tomar el relevo y cada componente individual puede avanzar a su propio ritmo. Cada componente está temporalmente desacoplado de los demás.

TIP40**Diseño usando servicios**

En lugar de componentes, realmente hemos creado *servicios*—objetos independientes y concurrentes detrás de interfaces bien definidas y consistentes.

Diseño para concurrencia

La creciente aceptación de Java como plataforma ha expuesto a más desarrolladores a la programación multiproceso. Pero la programación con subprocessos impone algunas restricciones de diseño, y eso es algo bueno. Esas restricciones son realmente tan útiles que queremos respetarlas cada vez que programamos. Nos ayudará a desacoplar nuestro código y luchar *programación por coincidencia* (ver página 172).

Con el código lineal, es fácil hacer suposiciones que conducen a una programación descuidada. Pero la concurrencia 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 concurrente. Ahora puede ser un buen momento para preguntarse *por qué* necesitas 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 su objeto se encuentra en un estado no válido entre ciertas llamadas, es posible que confíe en una coincidencia de que nadie puede llamar a su objeto en ese momento.

Suponga que tiene un subsistema de ventanas donde los widgets se crean primero 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 los llama, y se pueden llamar en los momentos más incómodos. Debe asegurarse de que un objeto esté en un estado válido *ningún* tiempo que posiblemente podría ser llamado. 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). Usando invariantes de clase, discutidos en *Diseño por contrato*, página 109, le ayudará a evitar esta trampa.

Interfaces más limpias

Pensar en la concurrencia y las dependencias ordenadas por tiempo también puede llevártelo a diseñar interfaces más limpias. Considere la rutina de la biblioteca C `strtok`, que divide una cadena en tokens.

el diseño de `strtok` es seguro para subprocessos,⁶ pero eso no es lo peor: mira la dependencia del tiempo. Debe hacer la primera llamada a `strtok` con la variable que desea analizar, y todas las llamadas sucesivas con un NULO en cambio. Si pasa en un no-NULO valor, reinicia el análisis en ese búfer en su lugar. Sin siquiera considerar los hilos, suponga que desea utilizar `strtok` para analizar dos cadenas separadas al mismo tiempo:

```
carbonizarse buf1[BUFSIZ];
carbonizarse buf2[BUFSIZ];
carbonizarse *p, *q;
strcpy(buf1, "esto es una prueba"); strcpy(buf2, "esto no va a
funcionar");
p = strtok(buf1, " "); q = strtok(buf2,
" "); tiempo(p && q) {
    imprimirf("%s %s\n", p, q);
    p = strtok(NULO, " "); q =
    strtok(NULO, " ");
}
```

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 subprocessos. Además, golpea el primer argumento que pasa, lo que puede llevar a algunas sorpresas desagradables.

El código como se muestra no funcionará: hay un estado implícito retenido en strtokentre llamadas. tienes que usar strtok un solo búfer a la vez.

Ahora, en Java, el diseño de un analizador de cadenas tiene que ser diferente. Debe ser seguro para subprocessos y presentar un estado coherente.

```
Tokenizador de cadena st1 =nuevoTokenizador de cadena("esto es una prueba"); Tokenizador de
cadena st2 =nuevoTokenizador de cadena("esta prueba funcionará");
tiempo(st1.hasMoreTokens() && st2.hasMoreTokens()) {
    System.out.println(st1.nextToken());
    System.out.println(st2.nextToken());
}
```

Tokenizador de cadenas es una interfaz mucho más limpia y fácil de mantener. No contiene sorpresas y no causará errores misteriosos en el futuro, como strtok puede que.

TIP41

Diseñe siempre para la concurrencia

Despliegue

Una vez que haya diseñado una arquitectura con un elemento de concurrencia, será más fácil pensar en manejar *muchos* servicios concurrentes: el modelo se vuelve omnipresente.

Ahora puede ser flexible en cuanto a cómo se implementa la aplicación: independiente, cliente-servidor o *norte-nivel*. Al diseñar su sistema como servicios independientes, también puede hacer que la configuración sea dinámica. Al planificar la simultaneidad y desacoplar las operaciones a tiempo, tiene todas estas opciones, incluida la opción independiente, donde puede elegir *no* ser concurrente.

Yendo hacia el otro lado (intentar agregar concurrencia a una aplicación no concurrente) es *mucho* más difícil. Si diseñamos para permitir la concurrencia, podemos cumplir con mayor facilidad los requisitos de escalabilidad o rendimiento cuando llegue el momento, y si el momento nunca llega, todavía tenemos el beneficio de un diseño más limpio.

¿No es hora?

Las secciones relacionadas incluyen:

- *Diseño por contrato*, página 109 *Programación*
- *por Coincidencia*, página 172

Desafíos

- ¿Cuántas tareas realizas en paralelo cuando te preparas para el trabajo por la mañana? ¿Podría expresar esto en un diagrama de actividad UML? ¿Puedes encontrar alguna manera de prepararte más rápidamente aumentando la concurrencia?

29

es solo una vista

*Sin embargo, un hombre oye
lo que quiere oír y no tiene
en cuenta el resto La la la...*

► Simon y Garfunkel, "El boxeador"

Al principio se nos enseña a no escribir un programa como una sola pieza grande, sino que debemos "dividir y conquistar" y separar un programa en módulos. Cada módulo tiene sus propias responsabilidades; de hecho, una buena definición de un módulo (o clase) es que tiene una única responsabilidad bien definida.

Pero una vez que separa un programa en diferentes módulos según la responsabilidad, tiene un nuevo problema. En tiempo de ejecución, ¿cómo se comunican los objetos entre sí? ¿Cómo gestionas las dependencias lógicas entre ellos? Es decir, ¿cómo sincroniza los cambios de estado (o las actualizaciones de los valores de los datos) en estos diferentes objetos? Debe hacerse de una manera limpia y flexible: no queremos que sepan demasiado unos de otros. Queremos que cada módulo sea como el hombre de la canción y solo escuche lo que quiere escuchar.

Comenzaremos con el concepto de un evento. 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 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 enfocado en su propia agenda (de la cual el remitente está felizmente inconsciente).

Sin embargo, debemos tener cuidado al usar eventos. En una versión anterior de Java, por ejemplo, una rutina recibió *todos* los eventos destinados a una determinada aplicación. No es exactamente el camino hacia un fácil mantenimiento o evolución.

Publicar/Suscribirse

¿Por qué es malo empujar todos los eventos a través de una sola rutina? Viola la encapsulación de objetos: 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* acoplamiento. Debido a que los objetos mismos también deben tener conocimiento de estos eventos, probablemente violará el *SECO* principio, ortogonalidad, y quizás incluso secciones de la Convención de Ginebra. Es posible que haya visto este tipo de código; generalmente está dominado por una enorme declaración o multivía si entonces. Podemos hacerlo mejor.

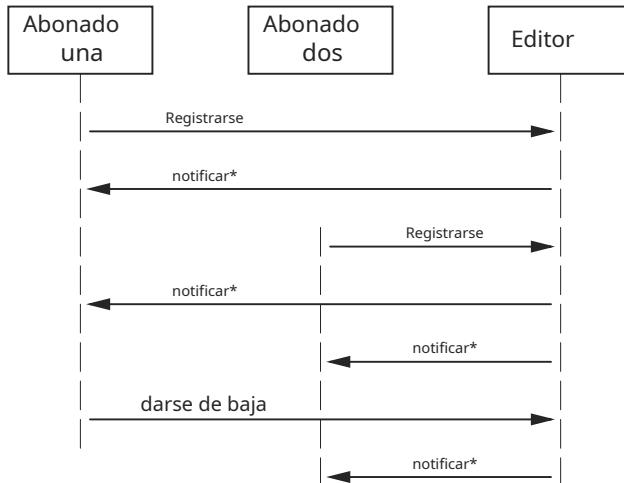
Los objetos deben poder registrarse para recibir solo los eventos que necesitan y nunca deben recibir eventos que no necesitan. ¡No queremos enviar spam a nuestros objetos! En su lugar, podemos usar un *publicar/suscribir* protocolo, ilustrado usando el *diagrama de secuencia UML* en la Figura 5.4 en 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 *Editor*, todo lo que tenemos que hacer es registrarnos. El *Editor* hace un seguimiento de todos los interesados abonados a los objetos; cuando el *Editor* genera un evento de interés, llamará a cada abonado a su vez y notificarles que el evento ha ocurrido.

7. Consulte también el patrón Observer en [GHJV95] para obtener más información.

Figura 5.4. Protocolo de publicación/suscripción



Hay varias variaciones sobre este tema, que reflejan otros estilos de comunicación. Los objetos pueden usar la publicación/suscripción de igual a igual (como vimos anteriormente); pueden usar un "bus de software" donde un objeto centralizado mantiene la base de datos de oyentes y envía mensajes de manera adecuada. Incluso podría tener un esquema en el que los eventos críticos se transmiten 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 recuadro 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

Suponga 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 Servicio de Eventos CORBA

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 manejo de eventos y también desvincula a los productores de eventos de los consumidores de eventos. Funciona de dos formas básicas: *empujar* y *jalar*.

En el modo push, los proveedores de eventos informan al canal de eventos que se ha producido un evento. Luego, el canal distribuye automáticamente ese evento a todos los objetos de cliente que han registrado interés.

En el modo pull, 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 de CORBA se puede usar para implementar todos los modelos de eventos discutidos 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. Sentado sobre CORBA, el servicio de eventos le brinda una forma desacoplada de interactuar con aplicaciones de todo el mundo, escritas por personas que nunca ha conocido, 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. Entonces creamos un *modelo*—los datos en sí, con operaciones comunes para manipularlos. Entonces podemos crear por separado *puntos de vista* que muestran 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 sus propias *controlador*. La vista de gráfico puede tener un controlador que le permita acercar o alejar, o desplazarse por los datos, por ejemplo. Nada de esto afecta los datos en sí, solo esa vista.⁸

Este es el concepto clave detrás del modismo Model-View-Controller (MVC): separar el modelo de la GUI que lo representa y los controles que gestionan 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 usar visores comunes en muchos modelos de datos diferentes. Incluso puede admitir varios controladores para proporcionar mecanismos de entrada no tradicionales.

TIP42

Vistas separadas de los modelos

Al aflojar el acoplamiento entre el modelo y la vista/controlador, obtiene 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 transitable y en el que se puede hacer clic) 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 ÁrbolModeloInterfaz. Su código ahora se convierte en el modelo para el árbol.

La vista es creada por el TreeCellRendereryTreeCellEditor clases, que se pueden heredar y personalizar para proporcionar diferentes colores, fuentes e íconos en el widget. JÁrbolactúa como controlador para el widget de árbol y proporciona algunas funciones de visualización general.

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, solo proporciona una fuente de datos.

Suponga que el vicepresidente se le acerca y quiere una aplicación 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 ÁrbolModelo, y voilà: tiene un widget de árbol totalmente navegable.

Ahora puede ponerse 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 dictan los nuevos estándares corporativos

el uso de un ícono de calavera y tibias cruzadas para ciertos empleados, puede realizar los cambios en TreeCellRenderers sin tocar ningún otro código.

Más allá de las GUI

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 (quizás un subconjunto); no es necesario que sea 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 abstracto 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 cambios en el modelo y eventos lógicos del controlador.
- **Controlador.** Una forma de controlar la vista y proporcionar al modelo nuevos datos. Publica eventos tanto en el modelo como en la vista.

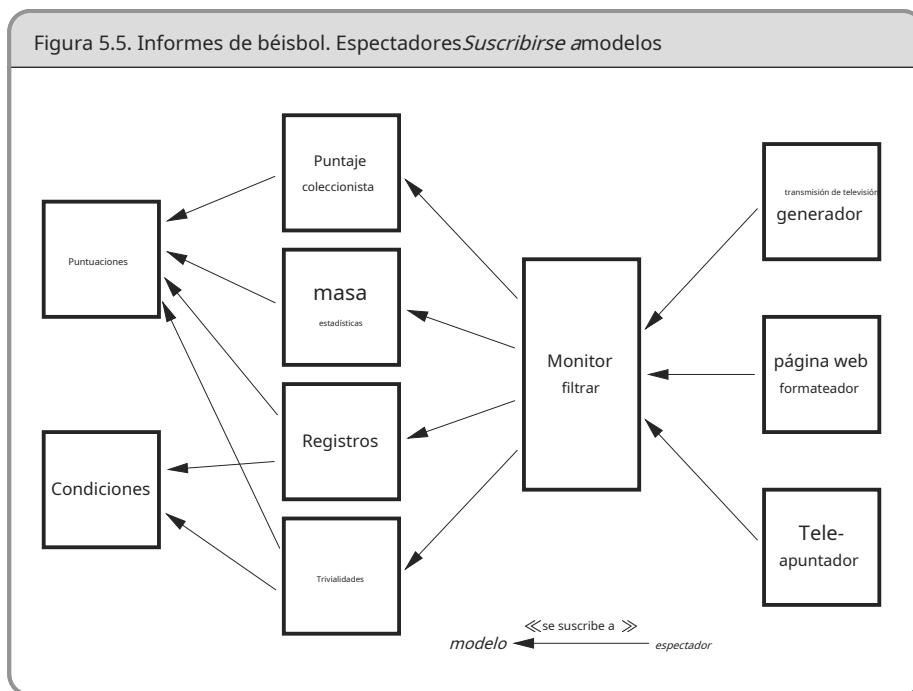
Veamos un ejemplo no gráfico.

El béisbol es una institución única. ¿Dónde más puedes aprender gemas de trivia como "este se ha convertido en el juego con mayor puntaje jugado un martes, bajo la lluvia, bajo luces artificiales, entre equipos cuyos nombres comienzan con una vocal?" Supongamos que se nos encargara desarrollar un software para apoyar a los locutores intrépidos que deben informar diligentemente sobre los puntajes, las estadísticas y las trivialidades.

Claramente, necesitamos información sobre el juego en curso: los equipos que juegan, las condiciones, el jugador al bate, el puntaje, etc. Estos hechos forman nuestros modelos; se actualizarán a medida que llegue nueva información (se cambia un lanzador, se poncha un jugador, comienza a llover). ...

Entonces tendremos una cantidad de objetos de vista que usan estos modelos. Una vista puede buscar carreras para poder 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 trivia, responsable de encontrar esos hechos extraños e inútiles que emocionan al público que los ve.

Figura 5.5. Informes de béisbol. Espectadores *Suscribirse a* modelos



Pero no queremos inundar al pobre locutor con todas estas opiniones 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 repentinamente en modelos para el objeto de nivel superior, que a su vez podría ser un modelo para diferentes visores de formato. Un visor de formato podría crear el guión del teleprompter para el locutor, otro podría generar subtítulos de video directamente en el enlace ascendente del satélite, otro podría actualizar las páginas web de la red o del equipo (consulte la Figura 5.5).

Este tipo de red modelo-visor es una técnica de diseño común (y valiosa). Cada enlace desacopla los datos sin procesar de los eventos que los 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 vuela por encima probablemente no sea interesante a menos que sea el avión número 100 que sobrevuela esa noche.

modelo puede tener *muchos visores*, y un visor puede trabajar con varios modelos.

En sistemas avanzados como este, puede ser útil tener *vistas de depuración*—vistas especializadas que le muestran detalles detallados del modelo. Agregar una función para rastrear eventos individuales también puede ser un gran ahorro de tiempo.

Todavía acoplados (después de todos estos años)

A pesar de la disminución de acoplamiento que hemos logrado, los oyentes y los generadores de eventos (suscriptores y editores) todavía tienen *algún conocimiento* el uno del otro. En Java, por ejemplo, deben acordar definiciones de interfaz comunes y convenciones de llamadas.

En la siguiente sección, veremos formas de reducir aún más el acoplamiento mediante el uso de una forma de publicación y suscripción donde *ninguno* de los participantes necesitan conocerse o llamarse directamente.

Las secciones relacionadas incluyen:

- *ortogonalidad*, página 34
- *Reversibilidad*, página 44
- *El desacoplamiento y la Ley de Deméter*, página 138
- *pizarras*, página 165 *todo es escritura*, página 248
-

Ejercicios

29Suponga que tiene un sistema de reservas de una aerolínea que incluye el concepto de un vuelo:

```
interfaz pública Vuelo {
    // Devuelve falso si el vuelo está lleno. booleano público
    addPassenger(Pasajero p); vacío público addToWaitList(Pasajero
    p); público
        En t obtenerCapacidadVuelo();
    público En t obtenerNumPasajeros();
}
```

Si agrega un pasajero a la lista de espera, se lo colocará en el vuelo automáticamente cuando haya una vacante disponible.

Hay un trabajo de informes masivo que consiste en buscar vuelos llenos o con exceso de reservas para sugerir cuándo se pueden 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: lleva demasiado tiempo ejecutarlo. Utilice las ideas de esta sección para rediseñar esta interfaz.

30

pizarras

La escritura está en la pared...

Es posible que no se asocie habitualmente *elegancia* con detectives de la policía, imaginando en cambio una especie de cliché de café y donas. Pero considere cómo los detectives podrían usar un *pizarrapara* coordinar y resolver una investigación de asesinato.

Supongamos que el inspector jefe comienza instalando una gran pizarra en la sala de conferencias. En él, escribe una sola pregunta:

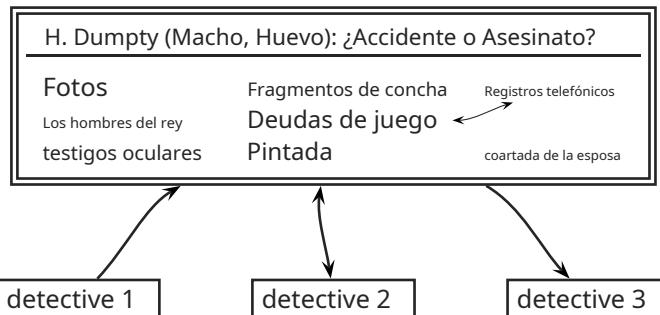
HDvacío(METROcerveza Inglesa, migg): AACCIDENTE O ASESINATO?

¿Humpty realmente se cayó o fue empujado? Cada detective puede hacer contribuciones a este posible misterio de asesinato agregando 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 pizarrón de muestra.

Algunas características clave del enfoque de pizarra son:

- Ninguno de los detectives necesita saber de la existencia de ningún otro detective: observan la pizarra en busca de nueva información y agregan sus hallazgos.
- Los detectives pueden estar capacitados en diferentes disciplinas, pueden tener diferentes niveles de educación y experiencia, y es posible que ni siquiera trabajen en el mismo recinto. 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 los registros telefónicos. Tal vez estaba recibiendo llamadas telefónicas amenazantes.



- 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, oraciones, evidencia física, etc.

Hemos trabajado en una serie de proyectos que involucraron un flujo de trabajo o un proceso de recopilación de datos distribuidos. Con cada uno, el diseño de una solución en torno a un modelo de pizarra simple nos dio una metáfora sólida con la que trabajar: todas las funciones enumeradas anteriormente que usan detectives son igualmente aplicables a objetos y módulos de código.

Un sistema de pizarra nos permite desacoplar nuestros objetos entre sí por completo, proporcionando un foro donde los consumidores y productores de conocimiento pueden intercambiar datos de forma anónima y asíncrona. Como puede suponer, también reduce la cantidad de código que tenemos que escribir.

Implementaciones de pizarra

Los sistemas de pizarra basados en computadora 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 tipo pizarra distribuidos, como JavaSpaces y T Spaces [URL 50, URL 25], se basan primero en un modelo de pares clave/valor.

popularizado en Linda [CG90], donde el concepto era conocido como *espacio de tupla*.

Con estos sistemas, puede almacenar objetos Java activos, no solo datos en la pizarra, y recuperarlos mediante coincidencias parciales de campos (mediante plantillas y comodines) o por subtipos. Por ejemplo, suponga que tiene un tipo `Autor`, que es un subtipo de `Persona`. Podrías buscar en una pizarra que contenga `Persona` objetos usando un `Autor` plantilla con un apellido valor de "Shakespeare". Obtendrías 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. Coloque
escribe	un elemento 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.

Como podemos almacenar objetos, podemos usar una pizarra para diseñar algoritmos basados en un *Floridajo de objetos*, no solo datos. Es como si nuestros detectives pudieran fijar a las personas en la pizarra, a los propios testigos, no solo a sus declaraciones. Cualquiera puede hacer preguntas a un testigo en el desarrollo 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 sistemas como estos es que tiene una única interfaz coherente con la pizarra. Al crear una aplicación distribuida convencional, puede dedicar una gran cantidad de tiempo a crear llamadas de API únicas para cada transacción e interacción distribuidas en el sistema. Con la explosión combinatoria de interfaces e interacciones, el proyecto puede convertirse rápidamente en una pesadilla.

Organizar tu pizarra

Cuando los detectives trabajan en casos grandes, la pizarra puede abarrotarse y puede resultar difícil ubicar los datos en la pizarra. La solución es dividirla pizarra y comenzar a organizar los datos en la pizarra de alguna manera.

Diferentes sistemas de software manejan esta partición de diferentes maneras; algunos usan bastante plano zonas o grupos de interés, mientras que otros adoptan una estructura de árbol más jerárquica.

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 préstamos o hipotecas. Las leyes que rigen esta área son odiosamente complejas, con gobiernos federales, estatales y locales que tienen su opinión. El prestamista debe probar que ha revelado ciertas cosas y debe solicitar cierta información, pero deben hacer ciertas otras preguntas, y así sucesivamente, y así sucesivamente.

Más allá del miasma de la ley aplicable, también tenemos los siguientes problemas con los que lidiar.

- No hay garantía sobre el orden en que llegan los datos. Por ejemplo, las consultas para una verificación de crédito o una búsqueda de títulos pueden llevar mucho 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.
- Algunos datos recopilados pueden ser realizados automáticamente por otros sistemas. Estos datos también pueden llegar de forma asíncrona.
- No obstante, ciertos datos aún pueden depender 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 tratar de manejar todas las combinaciones y circunstancias posibles utilizando un sistema de flujo de trabajo. Existen muchos de estos sistemas, pero pueden ser complejos y requerir mucho programador. A medida que cambian las regulaciones, se debe reorganizar el flujo de trabajo: es posible que las personas deban cambiar sus procedimientos y que se deba reescribir el código integrado.

Una pizarra, en combinación con un motor de reglas que resume los requisitos legales, es una solución elegante para las dificultades encontradas aquí. El orden de llegada de los datos es irrelevante: cuando se publica un hecho, puede desencadenar las reglas apropiadas. Los comentarios también se manejan fácilmente: la salida de cualquier conjunto de reglas puede publicarse en la pizarra y provocar la activación de aún más reglas aplicables.

TIP43

Use pizarras para coordinar el flujo de trabajo

Podemos usar la pizarra para coordinar hechos y agentes dispares, mientras mantenemos la independencia e incluso el aislamiento entre los participantes.

Puede lograr los mismos resultados con más métodos de fuerza bruta, por supuesto, pero tendrá 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 funcionar su programa.

Las secciones relacionadas incluyen:

- *El poder del texto sin formato*, página 73
- *es solo una vista*, página 157

Desafíos

- ¿Utiliza sistemas de pizarra en el mundo real: el tablero de mensajes junto al refrigerador o la gran pizarra en el trabajo? ¿Qué los hace efectivos? ¿Los mensajes se publican alguna vez con un formato coherente? ¿Importa?

Ejercicios

Responder
en P. 297

30 Para cada una de las siguientes aplicaciones, ¿sería adecuado o no un sistema de pizarra? ¿Por qué?

1. Procesamiento de imágenes. Le gustaría tener una serie de procesos paralelos que tomen fragmentos de una imagen, los procesen y vuelvan a colocar el fragmento completo.
2. Calendario de grupos. Tiene personas repartidas 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 codificas

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 sentencias ejecutables. Creemos que esta actitud es la principal razón por la que muchos programas son feos, inefficientes, mal estructurados, imposibles de mantener y simplemente erróneos.

La codificación no es mecánica. Si lo 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 deben tomarse 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 podría funcionar, pero no hay una razón particular para ello. En *Programación por Coincidencia*, 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 brindamos algunos consejos sobre cómo detectar posibles problemas antes de que sucedan.

Los programadores pragmáticos piensan críticamente sobre todo el código, incluido el nuestro. Constantemente vemos espacio para mejorar en nuestros programas y nuestros diseños. En *refactorización*, buscamos técnicas que nos ayuden a corregir el código existente incluso mientras estamos en medio de un proyecto.

Algo que debe tener en mente siempre que esté produciendo código es que algún día tendrá que probarlo. Facilitar el código

para probar, y aumentará la probabilidad de que realmente se pruebe, un pensamiento que desarrollamos en *Código que es fácil de probar*.

Finalmente, en *Magos malvados*, le sugerimos que tenga cuidado con las herramientas que escriben montones de código en su nombre a menos que comprenda lo que están haciendo.

La mayoría de nosotros podemos conducir un automóvil en gran parte con el piloto automático: no ordenamos explícitamente a nuestro pie que presione un pedal o nuestro brazo para girar el volante, solo pensamos "disminuya la velocidad y gire a la derecha". Sin embargo, los conductores buenos y seguros revisan constantemente la situación, verifican posibles problemas y se colocan en buenas posiciones en caso de que ocurra algo inesperado. Lo mismo ocurre con la codificación: puede ser en gran medida una rutina, pero mantener su ingenio bien podría evitar un desastre.

31

Programación por Coincidencia

¿Alguna vez has visto viejas películas de guerra en blanco y negro? El soldado cansado avanza con cautela fuera de la maleza. Hay un claro más adelante: ¿hay minas terrestres o es seguro cruzar? No hay indicios de que sea un campo minado, ni señales, alambre de púas ni cráteres. El soldado golpea el suelo delante de él con su bayoneta y se estremece, esperando una explosión. No hay uno. Así que avanza minuciosamente por el campo durante un tiempo, empujando y empujando a medida que avanza. Finalmente, convencido de que el campo es seguro, se endereza y marcha con orgullo hacia adelante, solo para volar en pedazos.

Las pruebas iniciales del soldado en busca de minas no revelaron nada, pero esto fue simplemente suerte. Fue llevado a una conclusión falsa, con resultados desastrosos.

Como desarrolladores, también trabajamos en campos minados. Hay cientos de trampas esperando para atraparnos cada día. Recordando la historia del soldado, debemos tener cuidado de sacar conclusiones falsas. Debemos evitar la programación por coincidencia, confiando en la suerte y los éxitos accidentales a favor de *deprogramando deliberadamente*.

Cómo programar por coincidencia

Supongamos que a Fred se le asigna una tarea de programación. Fred escribe algo de código, lo prueba y parece funcionar. Fred escribe algo más de código, lo prueba 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. Fred bien puede pasar una cantidad significativa de tiempo persiguiendo este fragmento de código sin poder arreglarlo. No importa lo que haga, parece que nunca funciona bien.

Fred no sabe por qué falla el código porque *él no sabía por qué funcionó en primer lugar*. Pareció funcionar, dadas las "pruebas" limitadas que hizo Fred, pero eso fue solo una coincidencia. Animado por una falsa confianza, Fred cargó hacia el olvido. Ahora, la mayoría de las personas inteligentes pueden conocer a alguien como Fred, pero *nosotros* conocemos mejor. No confiamos en las coincidencias, ¿verdad?

A veces podemos. 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 está escrito actualmente el código. Termina confiando en un error no documentado o en condiciones de contorno.

Suponga que llama a una rutina con datos incorrectos. La rutina responde de una manera particular y usted 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", su código puede romperse. En el caso más extremo, es posible que la rutina que llamó ni siquiera esté diseñada para hacer lo que usted quiere, pero *parece* para trabajar bien. Llamar a las cosas en el orden incorrecto, o en el contexto incorrecto, es un problema relacionado.

```
pintura(g);  
invalidar();  
validar();  
revalidar();  
repintar();  
pintar inmediatamente (r);
```

Aquí parece que Fred está tratando desesperadamente de sacar algo en la pantalla. Pero estas rutinas nunca fueron diseñadas para llamarse así; aunque parecen funcionar, en realidad es solo una coincidencia.

Para colmo de males, cuando el componente finalmente se dibuja, Fred no intentará volver atrás y eliminar las llamadas falsas. "Funciona ahora, mejor déjalo en paz. . . ."

Es fácil dejarse engañar por esta línea de pensamiento. ¿Por qué debería correr el riesgo de jugar con algo que está funcionando? Bueno, podemos pensar en varias razones:

- Puede que realmente no esté funcionando, puede parecer que sí 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 su código sea más lento.
- Las llamadas adicionales también aumentan el riesgo de introducir nuevos errores propios.

Para el código que escribe 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 (ver *Diseño por contrato*, página 109) puede ayudar a eliminar malentendidos.

Para las rutinas que llame, confíe solo en el comportamiento documentado. Si no puede, por cualquier motivo, documente bien su suposición.

Accidentes de contexto

También puede tener "accidentes de contexto". Suponga que está escribiendo un módulo de utilidad. Solo porque actualmente está codificando para un entorno GUI, ¿el módulo tiene que depender de la presencia de una GUI? ¿Está confiando en los usuarios de habla inglesa? ¿Usuarios alfabetizados? ¿En qué más estás confiando que no está garantizado?

Suposiciones implícitas

Las coincidencias pueden inducir a error en todos los niveles, desde la generación de requisitos hasta las pruebas. Las pruebas están particularmente cargadas de causalidades falsas y resultados coincidentes. Es fácil asumir que X causas Y , pero como dijimos en *depuración*, página 90: no lo asuma, demuéstrelo.

En todos los niveles, las personas operan con muchos supuestos en mente, pero estos supuestos rara vez se documentan y, a menudo, están en conflicto entre diferentes desarrolladores. Las suposiciones que no se basan en hechos bien establecidos son la ruina de todos los proyectos.

TIP44

No programes por coincidencia

Cómo programar deliberadamente

Queremos dedicar menos tiempo a generar código, detectar y corregir errores lo antes posible en el ciclo de desarrollo y, para empezar, crear menos errores.

Ayuda si podemos programar deliberadamente:

- Sea siempre consciente de lo que está haciendo. Fred dejó que las cosas se salieran de control lentamente, hasta que terminó hirviendo, como la rana en *Sopa de piedra y ranas hervidas*, página 7.
- No codifiques con los ojos vendados. Intentar crear una aplicación que no comprende del todo 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 esté en su cabeza, en el reverso de una servilleta de cóctel o en una copia impresa del tamaño de una pared de una herramienta CASE.
- Confía solo en cosas confiables. No dependa de accidentes o suposiciones. Si no puede notar la diferencia en circunstancias particulares, asuma lo peor.
- Documente sus suposiciones. *Diseño por contrato*, página 109, puede ayudar a aclarar sus suposiciones en su propia mente, así como ayudar a comunicarlas a los demás.
- No solo pruebe su código, sino también sus suposiciones. No adivine; en realidad intentarlo. Escribe una afirmación para probar tus suposiciones (ver *Programación Asertiva*, página 122). Si su afirmación es correcta, ha mejorado la documentación en su código. Si descubre que su suposición es incorrecta, entonces considérese afortunado.
- Prioriza tu esfuerzo. Dedique tiempo a los aspectos importantes; más que probable, estas son las partes difíciles. Si no tienes fundamen-

tals o infraestructura correcta, campanas y silbatos brillantes serán irrelevantes.

- No seas esclavo de la historia. No permita que el código existente dicte el código futuro. Todo el código se puede reemplazar si ya no es apropiado. Incluso dentro de un programa, no permita que lo que ya ha hecho restrinja lo que hará a continuación: esté listo para refactorizar (ver *refactorización*, página 184). Esta decisión puede afectar el cronograma del proyecto. La suposición es que el impacto será menor que el costo de no haciendo 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:

- *Sopa de piedra y ranas hervidas*, página 7
- *depuración*, página 90 *Diseño por contrato*,
- página 109 *Programación Asertiva*, página
- 122 *Acoplamiento temporal*, página 150
- *refactorización*, página 184 *todo es escritura*
- , página 248
-

Ejercicios

31 Puedes identificar algunas coincidencias en el siguiente fragmento de código C? Suponga que este código está enterrado profundamente en una rutina de biblioteca.

```
fprintf(stderr, "Error, continuar?"); obtiene(buf);
```

32 Esta pieza de código C podría funcionar algunas veces, en algunas máquinas. Entonces de nuevo, tal vez no. ¿Qué ocurre?

```
/* Trunca la cadena a sus últimos caracteres maxlen */
vacíocadena_cola(carbonizarse*cuerda,En tmaxlen) {
    En tlen = strlen(cadena); Si(len > maxlen)
    {
        strcpy(cadena, cadena + (len - maxlen));
    }
}
```

1. También puedes 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 seguimiento de Java de uso general. La función escribe una cadena en un archivo de registro. Pasa su prueba unitaria, pero falla cuando uno de los desarrolladores web lo usa. ¿En qué coincidencia se basa?

```
vacio estático público depurar (Cadena s) lanza IOException {
    FileWriter fw = nuevoFileWriter("registro de depuración", verdadero);
    fw.write(s);
    fw.flush();
    fw.cerrar();
}
```

32

Velocidad del algoritmo

En *Estimación*, página 64, hablamos sobre estimar cosas como cuánto tiempo lleva caminar por la ciudad o cuánto tiempo llevará terminar un proyecto. Sin embargo, hay otro tipo de estimación que los programadores pragmáticos usan casi a diario: estimar los recursos que usan los algoritmos: tiempo, procesador, memoria, etc.

Este tipo de estimación suele ser crucial. Ante la posibilidad de elegir entre dos formas de hacer algo, ¿cuál eliges? Sabe cuánto tiempo se ejecuta su programa con 1000 registros, pero ¿cómo escalará a 1 000 000? ¿Qué partes del código necesitan optimizarse?

Resulta que estas preguntas a menudo se pueden responder usando el sentido común, un poco de análisis y una forma de escribir aproximaciones llamada notación "gran O".

¿Qué queremos decir con algoritmos de estimación?

La mayoría de los algoritmos no triviales manejan algún tipo de entrada variable: clasificación n cuerdas, invirtiendo un $m \times n$ matriz, o descifrar un mensaje con un n -tecla 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 utilizada.

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 al 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 . 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 recursivas, verificamos inconscientemente los requisitos de memoria y tiempo de ejecución. Esto rara vez es un proceso formal, sino más bien una confirmación rápida de que lo que estamos haciendo es sensato en las circunstancias. Sin embargo, a veces nosotros *hacer* y nosotros mismos realizando un análisis más detallado. Ahí es cuando la notación se vuelve útil.

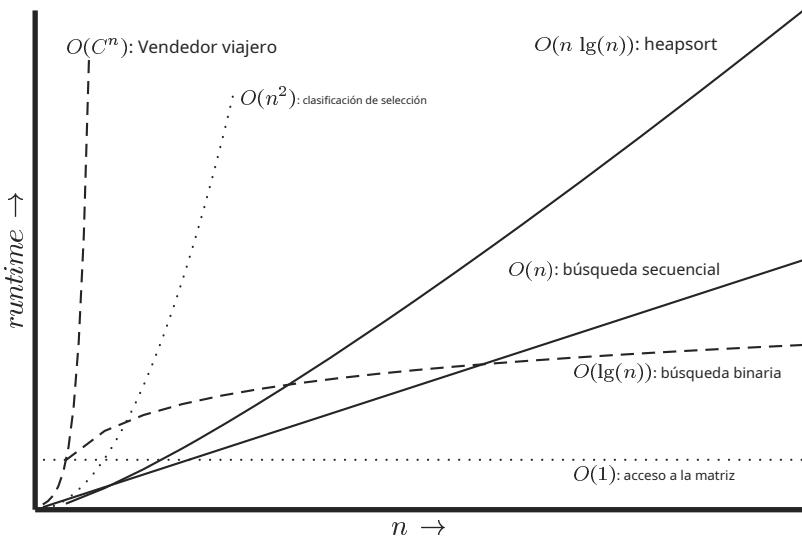
los $O()$ Notación

los $O()$ 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 tiempo en el peor de los casos variará como el cuadrado de . Duplica el número de registros y el tiempo se multiplicará aproximadamente por cuatro. Piense en el como significado de *orden*. La notación pone un límite superior al valor de lo que estamos midiendo (tiempo, memoria, etc.). Si decimos que una función toma tiempo $O(n^2)$ entonces sabemos que el límite superior del tiempo que toma no crecerá más rápido que . A veces encontramos funciones bastante complejas, pero debido a que el término de mayor orden dominará el valor a medida que aumenta, la convención es eliminar todos los términos de menor orden y no molestarse en mostrar ningún factor de multiplicación constante. $O(\frac{n^2}{2} + 3n)$ es igual que $O(\frac{n^2}{2})$, que es equivalente a $O(n^2)$. Esto es en realidad una debilidad. del $O()$ notación—uno $O(n^2)$ algoritmo puede ser 1.000 veces más rápido que otro $O(n^2)$ algoritmo, pero no lo sabrá por la notación.

La Figura 6.1 muestra varios $O()$ anotaciones con las que te encontrarás, 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, suponga que tiene una rutina que tarda 1 segundo en procesar 100 registros. ¿Cuánto tiempo tomará procesar 1,000? Si su código $O(1)$ aún tardará 1 s. Si es $O(\lg(n))$, entonces probablemente estarás esperando 3 s. $O(n)$ mostrará un aumento lineal a 10 s, mientras que $O(n \lg(n))$ tomará a unos 33 s. Si tienes la mala suerte de tener un $O(n^2)$ rutina, luego siéntate volver durante 100 s mientras hace sus cosas. Y si estás usando un exponencial

Figura 6.1. Tiempos de ejecución de varios algoritmos.



Algo común $\mathcal{O}()$ notaciones

- | | |
|---------------|---|
| $O(1)$ | Constante (elemento de acceso en matriz, declaraciones simples) |
| $O(\lg(n))$ | Logarítmica (búsqueda binaria)
<i>[La notación es $\lg(n)$ una abreviatura de $\log_2(n)$]</i> |
| $O(n)$ | 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 cuadrática (clases de selección e inserción) |
| $O(n^3)$ | Cúbico (multiplicación de 2 $n \times n$ matrices) |
| $O(C^n)$ | Exponencial (problema del viajante de comercio, partición de conjuntos) |
-

algoritmo $O(2^n)$, es posible que desee preparar una taza de café: su rutina debería terminar en aproximadamente 10^{263} años. Háganos saber cómo termina el universo.

los $O()$ la notación no se aplica solo al tiempo; puedes usarlo para representar cualquier otro recurso utilizado por un algoritmo. Por ejemplo, a menudo es útil poder modelar el consumo de memoria (vea 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 se ejecuta desde el 1 a n , entonces el algoritmo, es probable que sea $O(n)$ —el tiempo aumenta linealmente con n . Examen Los ejemplos incluyen búsquedas exhaustivas, encontrar el valor máximo en una matriz y generar sumas de verificación.
- Bucles anidados. Si anida un bucle dentro de otro, entonces su algoritmo el ritmo se convierte $O(m \times n)$, donde m y n son los límites de los dos bucles. Esto suele ocurrir en algoritmos de clasificación simples, como la clasificación de burbuja, en la que el bucle externo escanea cada elemento de la matriz por turnos y el bucle interno determina dónde colocar ese elemento en el resultado ordenado. Tales algoritmos de clasificación tienden a ser $O(n^2)$.
- Corte binario. Si su algoritmo reduce a la mitad el conjunto de cosas que considera cada vez alrededor del ciclo, entonces es probable que sea logarítmico, $O(\lg(n))$ (ver Ejercicio 37, página 183). Una búsqueda binaria de una lista ordenada, atravesar un árbol binario y encontrar el primer bit establecido en una palabra de máquina puede ser todo $O(\lg(n))$.
- Divide y conquistarás. Los algoritmos que dividen 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 recursivamente cada una. Aunque técnicamente $O(n^2)$, porque su comportamiento se degrada cuando se alimenta de 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 factorial es $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$ permutaciones de los dígitos del 1 al 5. El tiempo una combinatoria

algoritmo para cinco elementos: se tardará seis veces más en ejecutarlo para seis y 42 veces más para siete. Los ejemplos incluyen algoritmos para muchos de los reconocidos *difícil* problemas: el problema del vendedor ambulante, empaquetar cosas de manera óptima en un contenedor, dividir un conjunto de números para que cada conjunto tenga el mismo total, etc. A menudo, las heurísticas se utilizan 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 pase mucho tiempo durante su 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 encuentras escribiendo un bucle simple, sabes que tienes un

$O(n)$ algoritmo. Si ese ciclo contiene un ciclo interno, entonces estas mirando $O(m \times n)$. Deberías preguntarte qué tan grande estos valores pueden obtener. Si los números están acotados, sabrá cuánto tardará en ejecutarse el código. Si los números dependen de factores externos (como el número de registros en una ejecución por lotes durante la noche o el número de nombres en una lista de personas), es posible que desee detenerse y considerar el efecto que pueden tener los valores grandes en su ejecución.

tiempo o consumo de memoria.

TIP45

Calcule el orden de sus algoritmos

Hay algunos enfoques que puede tomar para abordar los problemas potenciales.

Si tienes un algoritmo que es $O(n^2)$, trata de encontrar una división y conquista enfoque que te llevará a $O(n \lg(n))$.

Si no está seguro de cuánto tiempo tomará su código o cuánta memoria usará, intente ejecutarlo, variando el recuento de registros de entrada o lo que sea que pueda afectar el tiempo de ejecución. Luego grafica los resultados. Pronto debería tener una buena idea de la forma de la curva. ¿Se curva hacia arriba, es una línea recta o se aplana a medida que aumenta el tamaño de entrada? Tres o cuatro puntos deberían darle una idea.

También considere lo que está haciendo en el código mismo. Un bucle simple puede funcionar mejor que uno complejo, $O(n \lg(n))$ uno para los más pequeños

valores de , particularmente si el $O(n \lg(n))$ El algoritmo tiene un costo bucle interno.

En medio de toda esta teoría, no olvides que también hay consideraciones prácticas. El tiempo de ejecución puede parecer que 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 desmoronarse. Si prueba una rutina de clasificación con claves de entrada aleatorias, es posible que se sorprenda la primera vez que encuentre una entrada ordenada. Los programadores pragmáticos intentan cubrir tanto las bases teóricas como las prácticas. Después de todas estas estimaciones, el único tiempo que cuenta es la velocidad de su código, ejecutándose en el entorno de producción, con datos reales.² Esto lleva a nuestro siguiente consejo.

TIP46

Pruebe sus estimaciones

Si es complicado obtener tiempos precisos, use *perfmonitors de código* para contar la cantidad de veces que se ejecutan los diferentes pasos en su algoritmo, y graficar estas cifras contra el tamaño de la entrada.

Lo mejor no siempre es lo mejor

También debe ser pragmático al elegir los algoritmos apropiados; el más rápido no siempre es el mejor para el trabajo. Con un conjunto de entrada pequeño, una ordenación por inserción directa funcionará tan bien como una ordenación rápida y le llevará menos tiempo escribir y depurar. También debe tener cuidado si el algoritmo que elige 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 ten cuidado con *optimización prematura*. Siempre es una buena idea asegurarse de que un algoritmo sea realmente un cuello de botella antes de invertir su valioso tiempo tratando de mejorarlo.

2. De hecho, mientras probaban los algoritmos de clasificación utilizados como ejercicio para esta sección en un Pentium de 64 MB, los autores se quedaron sin memoria real mientras ejecutaban la clasificación radix con más de siete millones de números. El tipo comenzó a usar espacio de intercambio y los tiempos se degradaron drásticamente.

Las secciones relacionadas incluyen:

- *Estimación*, página 64

Desafíos

- Cada desarrollador debe 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). Recomendamos agregar uno de sus libros a su colección y asegurarse de leerlo.
- Para aquellos a quienes les gustan más detalles de los que proporciona Sedgewick, lea el libro definitivo de Donald Knuth.*Arte de la programación informática* libros, que analizan una amplia gama de algoritmos [Knu97a, Knu97b, Knu98].
- En el Ejercicio 34, analizamos la clasificación 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 clave afecta 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 de nuestro sitio web (www.pragmaticprogrammer.com).Ejecútelo en varias máquinas disponibles para usted. ¿Sus cifras siguen las curvas esperadas? ¿Qué puede deducir acerca de las velocidades relativas de sus máquinas? ¿Cuáles son los efectos de varias configuraciones de optimización del compilador? ¿Es realmente lineal el tipo radix?

Responder
en P. 299

35.La siguiente rutina imprime el contenido de un árbol binario. Suponiendo que el árbol esté equilibrado, ¿cuánto espacio de pila usará 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).

Responder
en P. 300

```
vacíoimprimirÁrbol(constanteNodo *nodo) {
    carbonizarsebúfer[1000];
    si(nodo) {
        imprimirÁrbol(nodo->izquierda);
        getNodeAsString(nodo, búfer); pone (búfer);

        imprimirÁrbol(nodo->derecha);
    }
}
```

36.¿Puede ver alguna forma de reducir los requisitos de pila de la rutina del ejercicio 35 (aparte de reducir el tamaño del búfer)?

Responder
en P. 300

37.En la página 180, afirmamos que esto es una tajada $O(\lg(n))$. puedes probar binaria.

Responder
en P. 301

refactorización

Cambio y decadencia en todo lo que veo...

► HF Lyte, “Permanece conmigo”

A medida que evoluciona un programa, será necesario repensar decisiones anteriores y reelaborar partes del código. Este proceso es perfectamente natural. El código necesita evolucionar; no es algo estático.

Desafortunadamente, la metáfora más común para el desarrollo de software es la construcción de edificios (Bertrand Meyer [Mey97b] usa el término “Construcción de software”). Pero usar la construcción como metáfora guía implica estos pasos:

1. Un arquitecto elabora planos.
2. Los contratistas cavan los cimientos, construyen la superestructura, cablean y aploman, 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. En lugar de construcción, el software es más como jardinería—es más orgánico que concreto. Plantas muchas cosas en un jardín de acuerdo con un plan y condiciones iniciales. Algunos prosperan, otros están destinados a terminar como abono. Puede mover las plantaciones entre sí para aprovechar la interacción de la luz y la sombra, el viento y la lluvia. Las plantas demasiado grandes se parten o se podan, y los colores que chocan pueden trasladarse a lugares estéticamente más agradables. Quitas las malas hierbas y fertilizas las plantaciones que necesitan ayuda adicional. Supervisa 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 científico que la jardinería, es repetible, hay una jerarquía de informes 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 cierta rutina ha crecido demasiado, o está intentando

para lograr demasiado, debe dividirse en dos. Las cosas que no funcionan según lo planeado deben eliminarse o podarse.

Reescribir, reelaborar y rediseñar el código se conoce colectivamente como *refactorización*.

¿Cuándo debe refactorizar?

Cuando te encuentras con un obstáculo porque el código ya no encaja, o notas dos cosas que realmente deberían fusionarse, o cualquier otra cosa te parece "incorrecta", *no dudes en cambiarlo*. No hay tiempo como el presente. Cualquier número de cosas puede hacer que el código califique para la refactorización:

- Duplicación. Has descubierto una violación de la SECO principio (*Los males de la duplicación*, página 26).
- Diseño no ortogonal. Ha descubierto algún código o diseño que podría hacerse más ortogonal (*ortogonalidad*, página 34).
- Conocimiento obsoleto. Las cosas cambian, los requisitos se desplazan y su conocimiento del problema aumenta. El código necesita mantenerse al día.
- Actuación. Debe mover la funcionalidad de un área del sistema a otra para mejorar el rendimiento.

Refactorizar su código (mover la funcionalidad y actualizar las decisiones anteriores) es realmente un ejercicio en *el manejo del dolor*. Seamos realistas, cambiar el código fuente puede ser bastante doloroso: casi funcionaba y ahora está *De Verdad roto*. Muchos desarrolladores son reacios a comenzar a descifrar el código solo porque no es del todo correcto.

Complicaciones del mundo real

Así que vas con 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 usa a menudo como una excusa para no refactorizar. Pero esta excusa simplemente no se sostiene: si no refactoriza ahora, habrá una inversión de tiempo mucho mayor para solucionar el problema en el futuro, cuando haya más dependencias a tener en cuenta. ¿Habrá más tiempo disponible entonces? No en nuestra experiencia.

Es posible que desee explicar este principio al jefe utilizando una analogía médica: piense en el código que necesita ser refactorizado como un "crecimiento". Quitarlo requiere cirugía invasiva. Puedes entrar ahora y sacarlo mientras aún es pequeño. O bien, puede esperar mientras crece y se propaga, pero eliminarlo entonces será más costoso y más peligroso. Espere aún más y puede perder al paciente por completo.

TIP47

Refactorice temprano, refactorice a menudo

Mantenga un registro de las cosas que necesitan ser refactorizadas. Si no puede refactorizar algo de inmediato, asegúrese de incluirlo en el cronograma. Asegúrese de que los usuarios del código afectado *saben* que está programado para ser refactorizado y cómo esto podría afectarlos.

¿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 audiencia más amplia. 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, refactorizar es rediseñar. Cualquier cosa que usted u otros miembros de su equipo hayan diseñado se puede rediseñar a la luz de nuevos hechos, conocimientos más profundos, requisitos cambiantes, etc. Pero si procede a romper grandes cantidades de código con total abandono, es posible que se encuentre en una posición peor que cuando comenzó.

Claramente, la refactorización es una actividad que debe llevarse a cabo de forma lenta, deliberada y cuidadosa. Martin Fowler ofrece los siguientes consejos simples sobre cómo refactorizar sin hacer más daño que bien (consulte el recuadro en la página 30 en [FS97]):

1. No intente refactorizar y agregar funcionalidad al mismo tiempo.
2. Asegúrese de tener buenas pruebas antes de comenzar a refactorizar. Ejecute las pruebas con la mayor frecuencia posible. De esa manera, sabrá rápidamente si sus 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 navegadores web, los navegadores de clase permiten a los usuarios navegar y examinar las jerarquías y métodos de clase.

Por lo general, los navegadores de clases le permiten editar código, crear nuevos métodos y clases, etc. La siguiente variación de esta idea es la *navegación de refactorización*.

Un navegador de refactorización puede realizar operaciones de refactorización comunes de forma semiautomática: dividir una rutina larga en otras más pequeñas, propagar automáticamente los cambios en los nombres de métodos y variables, arrastrar y soltar para ayudarlo a mover el código, etc.

Mientras escribimos este libro, esta tecnología aún no ha aparecido fuera del mundo de Smalltalk, pero es probable que esto cambie a la misma velocidad que cambia Java, rápidamente. Mientras tanto, el navegador pionero de refactorización de Smalltalk se puede encontrar en línea en [URL 20].

3. Tome medidas breves y deliberadas: mueva un campo de una clase a otra, fusione 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 sus pasos pequeños y prueba después de cada paso, evitará una depuración prolongada.

Hablaremos más sobre las pruebas a este nivel en *Código que es fácil de probar*, página 189, y pruebas a mayor escala en *Pruebas despiadadas*, 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 vea un fragmento de código que no es como debería ser, arréglelo y todo lo que dependa de él. Controle el dolor: si le duele ahora, pero le va a doler aún más tarde, es mejor que lo supere.

con. Recuerda las lecciones de *Entropía del software*, página 4: no vivas con las ventanas rotas.

Las secciones relacionadas incluyen:

- *El gato se comió mi código fuente*, página 2
- *Entropía del software*, página 4
- *Sopa de piedra y ranas hervidas*, página 7 *Los males de la duplicación*, página 26
- *ortogonalidad*, página 34 *Programación por coincidencia*, página 172 *Código que es fácil de probar*, página 189 *Pruebas despiadadas*,
- página 237

Ejercicios

38. Obviamente, el siguiente código se ha actualizado varias veces a lo largo de los años, pero los cambios no han mejorado su estructura. Refactorizarlo.

```
Responder  
en P. 302
si(estado == TEXAS) {
    Velocidad = TASA_TX;
    cantidad = base * TASA_TX;
    calc = 2*base(importe) + adicional(importe)*1.05;
}
más si(estado == OHIO) || (estado == MAINE) {
    tasa = (estado == OHIO) ? TASA_OH : TASA_ME; amt = base * tasa;

    calc = 2*base(importe) + adicional(importe)*1.05; si(estado
== OHIO)
        puntos = 2;
}
más{
    Velocidad = 1;
    cantidad = base;
    calc = 2*base(importe) + adicional(importe)*1.05;
}
```

39. La siguiente clase de Java necesita admitir algunas formas más. Refactorice la clase para prepararla para las adiciones.

```
Responder  
en P. 303
clase públicaForma {
    int final estático públicoCUADRADO int final      = 1;
    estático públicoCIRCULO          = 2;
    int final estático públicoTRIÁNGULO_DERECHO = 3;
    privado   En t      tipo de forma;
    privado   doble    Talla;
    públicoForma(En tipo de forma,dobleTalla) {
        este.shapeType      = tipo de forma;
        este.Talla          = Talla;
    }
    //... otros métodos ...
}
```

```

publico dobleárea() {
    cambiar (tipo de forma) {
        caso CUADRADO: devolver tamaño*tamaño;
        caso CÍRCULO: devolver Matemáticas.PI*tamaño*tamaño/4.0;
        caso TRIÁNGULO RECTÁNGULO:devolvertamaño*tamaño/2.0; }

        devolver0;
    }
}

```

40Este código Java es parte de un marco que se utilizará a lo largo de su proyecto. Refactorícelo para que sea más general y más fácil de extender en el futuro.

Responder
en P. 303

```

clase públicaVentana {
    públicoVentana(En tancho,En tallura) { ... } vacío público establecerTamaño(En
    tancho,En tallura) { ... }
    booleano público se superpone (Ventana w) { ... } public int
    obtenerÁrea() { ... }
}

```

34

Código que es fácil de probar

Los *Circuito integrado de software*es una metáfora que a la gente le gusta usar cuando se habla de reutilización y desarrollo basado en componentes.³ La idea es que los componentes de software se combinan de la misma manera que se combinan los chips de circuitos integrados. Esto funciona solo si se sabe que los componentes que está utilizando son confiables.

Los chips están diseñados para probarse, 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 de autoprueba integrada (BIST) completa que ejecuta algunos diagnósticos de nivel básico internamente, o un mecanismo de acceso a 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, debemos incorporar la capacidad de prueba en el software desde el principio y probar cada pieza a fondo antes de intentar conectarlas.

3. El término "Software IC" (Circuito Integrado) parece haber sido inventado en 1986 por Cox y Novobilski en su libro Objective-C. *Programación orientada a objetos*[CN91].

Examen de la unidad

La prueba a nivel de chip para hardware es más o menos equivalente a *examen de la unidad* 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. Luego verifica 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 "IC 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 de unidad para probar el sistema como un todo. Hablamos de esta comprobación a gran escala del sistema en *Pruebas despiadadas*, página 237.

Sin embargo, antes de llegar tan lejos, debemos decidir qué probar a nivel de unidad. Por lo general, los programadores arrojan algunos bits aleatorios de datos al código y lo llaman probado. Podemos hacerlo mucho mejor, usando las ideas detrás *diseño por contrato*.

Prueba contra contrato

Nos gusta pensar en las pruebas unitarias como *prueba contra contrato* (ver *Diseño por contrato*, página 109). Queremos escribir casos de prueba que aseguren que una unidad determinada cumpla 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? Veamos la rutina de raíz cuadrada que encontramos por primera vez en la página 114. Su contrato es simple:

```
requerir
    argumento >= 0;
asegurar
    ((Resultado * Resultado) - argumento).abs <= epsilon*argumento;
```

Esto nos dice qué probar:

- Pase un argumento negativo y asegúrese de que sea rechazado.
- Pase un argumento de cero para asegurarse de que se acepta (este es el valor límite).
- Pase valores entre cero y el máximo argumento 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 realiza su propia verificación de condiciones previas y posteriores, podemos escribir un script de prueba básico para ejercitarse la función de raíz cuadrada.

```
vacio público valor de prueba (doble número, doble esperado) {
    dobles resultado = 0,0;
    probar{
        resultado = miSqrt(num);           // Podemos lanzar un
    }                                       // excepción de condición previa
    captura(arrojable e) {
        si(número < 0.0)
            devolver;
        más
        afirmar(falso);                // Si la entrada es < 0, entonces //
                                         // estamos esperando la // excepción,
                                         // de lo contrario // fuerza una prueba
                                         // fallida
    }
    afirmar(Math.abs(resultado-esperado) < épsilon*esperado);
}
```

Entonces podemos llamar a esta rutina para probar nuestra función de raíz cuadrada:

valor de prueba (-4.0,	0,0);
valor de prueba (0.0,	0,0);
valor de prueba (2.0,	1.4142135624);
valor de prueba (64.0,	8.0);
valor de prueba (1.0e7,	3162.2776602);

Esta es una prueba bastante simple; en el mundo real, es probable que cualquier módulo no trivial dependa de una serie de otros módulos, entonces, ¿cómo hacemos para probar la combinación?

Supongamos que tenemos un módulo A que usa una lista enlazada y una clasificación. En orden, probaremos:

1. Una lista enlazada's contrato, en su totalidad

2. Clasificar's contrato, en su totalidad

3. A's contrato, 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 Lista enlazada y Clasificar las pruebas pasaron, pero A's falló, podemos estar bastante seguros de que el problema está en A, o en A's *susar 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 vamos a todo este problema? Sobre todo, queremos evitar crear una "bomba de relojería", algo que pase desapercibido y explote en un momento incómodo más adelante en el proyecto. Al enfatizar las pruebas contra contrato, podemos tratar de evitar tantos de esos desastres posteriores como sea posible.

TIP48

Diseño para probar

Cuando diseña un módulo, o incluso una sola rutina, debe diseñar tanto su contrato como el código para probar ese contrato. Al diseñar el código para pasar una prueba y cumplir su contrato, es posible que tenga en cuenta las condiciones de contorno 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 construir las pruebas *antes de implementar* el código, puedes probar la interfaz antes de comprometerte con ella.

Escribir pruebas unitarias

Las pruebas unitarias para un módulo no deben colocarse en un rincón lejano del árbol de código fuente. Deben estar convenientemente ubicados. Para proyectos pequeños, puede incrustar la prueba de unidad para un módulo en el propio módulo. Para proyectos más grandes, sugerimos mover cada prueba a un subdirectorio. De cualquier manera, recuerda que si no es fácil de encontrar, no se usará.

Al hacer que el código de prueba sea fácilmente accesible, proporciona a los desarrolladores que pueden usar su código dos recursos invalables:

1. Ejemplos de cómo usar toda la funcionalidad de tu 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 excepto el archivo de clase principal de la aplicación, el principal rutina se puede utilizar para ejecutar pruebas unitarias; se ignorará cuando se ejecute la propia aplicación. 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) usando `#ifdef` para compilar el código de prueba de unidad 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 un valor de prueba rutina similar a la de Java definida anteriormente:

```
#ifdef __PRUEBA__
En tprincipal(En targc,carbonizarse**argv)
{
    argc--; argv++;
    // salta el nombre del programa
    si(argc < 2) {
        // hacer pruebas estándar si no hay
        valor de prueba (-4.0,      argumentos 0,0);
        valor de prueba (  0.0,      0,0);
        valor de prueba (  2.0,      1.4142135624);
        valor de prueba (64.0,      8.0);
        valor de prueba (1.0e7,     3162.2776602);
    }
    más{
        doblenum, esperado;
        tiempo(argumento >= 2) {
            num = atof(argv[0]);
            esperado = atof(argv[1]);
            testValue(num,esperado); argc
                - = 2;
            argv + = 2;
        }
    }
    devolver0;
}
# terminara si
```

Esta prueba de unidad ejecutará un conjunto mínimo de pruebas o, si se le dan argumentos, le permitirá pasar datos desde el mundo exterior. Un script de shell podría usar esta capacidad para ejecutar un conjunto de pruebas mucho más completo.

¿Qué hace si la respuesta correcta para una prueba unitaria es salir o abortar el programa? En ese caso, debe poder seleccionar la prueba para ejecutar, tal vez especificando un argumento en la línea de comando. lo harás

también necesita 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* pruebas de vez en cuando.

Uso de arneses de prueba

porque solemos escribir *mucho* código de prueba y hacemos muchas pruebas, nos facilitaremos la vida y desarrollaremos un arnés de prueba estándar para el proyecto. Lo principal que se muestra en la sección anterior es un arnés de prueba muy simple, pero por lo general necesitaremos más funciones que eso.

Un arnés de prueba puede manejar operaciones comunes como el estado de registro, el análisis de salida para los resultados esperados y la selección y ejecución de las pruebas. Los arneses pueden estar controlados por GUI, pueden estar escritos en el mismo idioma de destino que el resto del proyecto o pueden implementarse como una combinación de hacer archivos y secuencias de comandos Perl. Un arnés de prueba simple se muestra en la respuesta al Ejercicio 41 en la página 305.

En lenguajes y entornos orientados a objetos, puede crear una clase base que proporcione estas operaciones comunes. Las pruebas individuales pueden subclasicarse a partir de eso y agregar un código de prueba específico. Puede usar una convención de nomenclatura estándar y una reflexión en Java para crear una lista de pruebas dinámicamente. Esta técnica es una buena manera de honrar el *SECO* principio: no tiene que mantener una lista de pruebas disponibles. Pero antes de salir y comenzar a escribir su propio arnés, es posible que desee investigar xUnit de Kent Beck y Erich Gamma en [URL 22]. También puede consultar nuestro libro *Pruebas unitarias pragmáticas* [HT03] para una introducción a JUnit.

Independientemente de la tecnología que decida utilizar, 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 en busca de resultados esperados (o inesperados). Una
- forma estandarizada de notificación de fallas.

Las pruebas deben ser componibles; es decir, una prueba puede estar compuesta de 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, podemos terminar creando algunas pruebas particulares sobre la marcha. Estos pueden ser tan simples como una impresión en instrucción 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 nuevamente. No se deshaga de la prueba que creó; agréguelo a la prueba unitaria existente.

Por ejemplo, usando JUnit (el miembro de Java de la familia xUnit), podríamos escribir nuestra prueba de raíz cuadrada de la siguiente manera:

```
clase públicaJUnitEjemploextiendeCaso de prueba {
    públicoJUnitEjemplo(finalNombre de cadena) {
        súper(nombre);
    }
    vacío protegidoconfiguración() {
        // Cargar datos de prueba...
        testData.addElement(nuevo ParDbl(-4.0,0.0));
        testData.addElement(nuevo ParDbl(0.0,0.0));
        testData.addElement(nuevo ParDbl(64.0,8.0));
        testData.addElement(nuevo ParDbl(Doble.MAX_VALUE,
                                         1.3407807929942597E154));
    }
    vacío públicopruebaMiSqrt() {
        doblenum, esperado, resultado = 0.0;
        Enumeración enum = testData.elements(); tiempo(
        enum.hasMoreElements()) {
            DblPair p = (DblPair)enum.nextElement(); número
            = p.getNum();
            esperado = p.getExpected();
            valor de prueba (num, esperado);
        }
    }
    público estáticoBanco de pruebas() {
        Paquete TestSuite=nuevo Banco de pruebas(); suite.addTest(nuevo
        JUnitEjemplo("pruebaMiSqrt")); devolversuite;
    }
}
```

JUnit está diseñado para ser componible: podríamos agregar tantas pruebas como quisieramos a esta suite, y cada una de esas pruebas podría a su vez ser 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 carpintería.

Esto significa que a menudo necesitará 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 *spines de prueba* en el software, pero nosotros *pueden* proporcionar varias vistas del estado interno de un módulo, sin usar el depurador (que puede ser inconveniente o imposible en una aplicación de producción).

Los archivos de registro que contienen mensajes de seguimiento son uno de esos mecanismos. Los mensajes de registro deben estar en un formato regular y consistente; es posible que desee analizarlos automáticamente para deducir el tiempo de procesamiento o las rutas lógicas que tomó el programa. Los diagnósticos con formato deficiente o incoherente son simplemente "vomitar": son difíciles de leer y poco prácticos de analizar.

Otro mecanismo para acceder al 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 le revelaría a los usuarios finales, pero puede ser muy útil para la mesa de ayuda.

Para un 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 se encuentra 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 sonar difícil de implementar, pero no lo es. Los servidores web HTTP integrados y disponibles gratuitamente están disponibles en una variedad de idiomas modernos. Un buen lugar para empezar a buscar es [URL 58].

Una cultura de prueba

Todo el software que escribes *voluntad* ser probado, si no por usted y su equipo, entonces por los usuarios eventuales, por lo que también podría planear probarlo a fondo. Un poco de previsión puede contribuir en gran medida a minimizar los costos de mantenimiento y las llamadas a la mesa de ayuda.

A pesar de su reputación de hacker, la comunidad de Perl tiene un compromiso muy fuerte con las pruebas unitarias y de regresión. El procedimiento de instalación del módulo estándar de Perl admite una prueba de regresión invocando

```
% hacer 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 a un lugar específico y tienen un resultado esperado determinado. *Las pruebas son más culturales que técnicas;* podemos inculcar esta cultura de prueba en un proyecto independientemente del idioma que se utilice.

TIP49

Pruebe su software, o sus usuarios lo harán

Las secciones relacionadas incluyen:

- *El gato se comió mi código fuente*, página 2
- *ortogonalidad*, página 34 *Diseño por contrato*,
- página 109 *refactorización*, página 184
- *Pruebas despiadadas*, página 237
-

Ejercicios

41. Diseñe una plantilla de prueba para la interfaz de blender descrita en la respuesta al Ejercicio 17 en la página 289. Escriba un script de shell que realice una prueba de regresión para blender. Debe probar la funcionalidad básica, las condiciones de límite y de error y cualquier obligación contractual. ¿Qué restricciones se imponen al cambiar la velocidad? ¿Están siendo honrados?

Responder
en P. 305

Magos malvados

No se puede negar: las aplicaciones son cada vez más difíciles de escribir. Las interfaces de usuario en particular se están volviendo cada vez más sofisticadas. Hace veinte años, la aplicación promedio tendría una interfaz de teletipo de vidrio (si es que tuviera una interfaz). Los terminales asincrónicos suelen proporcionar una pantalla interactiva de caracteres, mientras que los dispositivos encuestables (como el omnipresente IBM 3270) le permiten completar una pantalla completa antes de presionar . 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 la integración del navegador web y la compatibilidad con clientes ligeros.

Todo el tiempo, las propias aplicaciones se vuelven más complejas. La mayoría de los desarrollos ahora usan 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 luchan por mantenerse al día. Si estuviéramos usando el mismo tipo de herramientas que produjeron las aplicaciones básicas de terminal tonta hace 20 años, nunca haríamos nada.

Así que los fabricantes de herramientas y los vendedores de infraestructura han ideado una fórmula mágica, *la mago*. Los magos son geniales. ¿Necesita una aplicación MDI compatible con contenedores OLE? Simplemente haga clic en un solo botón, responda un par de preguntas simples y el asistente generará automáticamente un código de esqueleto para usted. El entorno de Microsoft Visual C++ crea automáticamente más de 1200 líneas de código para este escenario. Los magos también trabajan duro en otros contextos. Puede usar asistentes para crear componentes de servidor, implementar beans Java y manejar interfaces de red en todas las áreas complejas en las que es bueno contar con la ayuda de expertos.

Pero usar un asistente diseñado por un gurú no convierte automáticamente a Joe en un desarrollador igualmente experto. Joe puede sentirse bastante bien: acaba de producir una gran cantidad de código y un programa que se ve bastante elegante. Simplemente agrega la funcionalidad de la aplicación específica y está listo para enviarse. Pero a menos que Joe entienda realmente el código que se ha producido en su nombre, se está engañando a sí mismo. Está programando por coincidencia. Los magos son una calle de sentido único: cortan el código por usted y luego siguen adelante. Si el

el código que producen no es del todo correcto, o si las circunstancias cambian y necesita adaptar el código, está solo.

No estamos en contra de los magos. Al contrario, le dedicamos un apartado entero (*Generadores de código*, página 102) a escribir el suyo propio. pero si tu *hacer* usa un asistente y no entiende 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.

TIP50

No use el código del asistente que no entiende

Algunas personas sienten que esta es una posición extrema. Dicen que los desarrolladores suelen confiar en cosas que no entienden del todo: la mecánica cuántica de los circuitos integrados, la estructura de interrupción del procesador, los algoritmos utilizados para programar procesos, el código de las bibliotecas suministradas, etc. Estamos de acuerdo. Y sentiríamos lo mismo acerca de los asistentes si fueran simplemente un conjunto de llamadas de biblioteca o servicios de sistema operativo estándar en los que los desarrolladores pudieran confiar. Pero no lo son. Los asistentes generan código que se convierte en parte integral de la aplicación de Joe. El código del asistente no se tiene en cuenta detrás de una interfaz ordenada: está entretejido línea por línea con la funcionalidad que Joe escribe.⁴ Eventualmente, deja de ser el código del mago y comienza a ser el de Joe. Y nadie debería estar produciendo código que no entienda por completo.

Las secciones relacionadas incluyen:

- *ortogonalidad*, página 34
- *Generadores de código*, página 102

Desafíos

- Si tiene un asistente de creación de GUI disponible, utilícelo para generar una aplicación de esqueleto. Revise 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, frijoles y AOP, en *ortogonalidad*, página 34.

Esta página se dejó en blanco intencionalmente

Capítulo 7

Antes del Proyecto

¿Alguna vez tiene la sensación de que su proyecto está condenado, incluso antes de que comience? A veces puede ser así, a menos que primero establezca algunas reglas básicas. De lo contrario, también podría sugerir que se cierre ahora y ahorrar algo de dinero al patrocinador.

Al comienzo de un proyecto, deberá determinar los requisitos. No basta con escuchar a los usuarios: leer *El pozo de requisitos* para descubrir más.

La sabiduría convencional y la gestión de restricciones son los temas de *Resolviendo rompecabezas imposibles*. Ya sea que esté realizando requisitos, análisis, codificación o pruebas, surgirán problemas difíciles. La mayoría de las veces, no serán tan difíciles como parecen al principio.

Cuando crea que ha resuelto los problemas, es posible que aún no se sienta cómodo con saltar y comenzar. ¿Es simple postergación o es algo más? *No hasta que estés listo* ofrece consejos sobre cuándo puede ser prudente escuchar esa voz de advertencia dentro de su cabeza.

Comenzar demasiado pronto es un problema, pero esperar demasiado puede ser aún peor. En *La trampa de las especificaciones*, discutiremos las ventajas de la especificación con un ejemplo.

Finalmente, veremos algunas de las trampas de los procesos y metodologías formales de desarrollo *encírculos y flechas*. No importa qué tan bien pensado esté, e independientemente de las "mejores prácticas" que incluya, ningún método puede reemplazarlo.*pensando*.

Con estos problemas críticos resueltos *antes de* el proyecto se pone en marcha, puede estar mejor posicionado para evitar la "parálisis de análisis" y realmente comenzar su proyecto exitoso.

El pozo de requisitos

La perfección se logra, no cuando no queda nada que agregar, sino cuando no queda nada que quitar. . . .

► Antoine de St. Exupéry, *Viento, arena y estrellas*, 1939

Muchos libros y tutoriales se refieren a *recopilación de requisitos* como una fase inicial 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 de fondo. "Recolectar" implica que los requisitos ya están allí; simplemente necesita encontrarlos, colocarlos en su canasta y seguir felizmente su camino.

No funciona de esa manera. Los requisitos rara vez se encuentran en la superficie. Normalmente, están enterrados bajo capas de suposiciones, conceptos erróneos y política.

TIP51

No reúna los requisitos: excavé por ellos

Buscando requisitos

¿Cómo puede reconocer un verdadero requisito mientras excava en toda la tierra que lo rodea? La respuesta es a la vez simple y compleja.

La respuesta simple es que un requisito es una declaración de algo que debe cumplirse. Los buenos requisitos pueden incluir lo siguiente:

- Un registro de empleado solo puede ser visto por un grupo designado de personas.
- La temperatura de la culata no debe exceder el valor crítico, que varía según el motor.
- El editor resaltará las palabras clave, que se seleccionarán según el 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 en 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, pero incorpora la política comercial 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 vincular las dos. Haga que el requisito sea la declaración general y brinde a los desarrolladores la información de la política como un ejemplo del tipo de cosas que necesitarán respaldar en la implementación. Eventualmente, la política puede terminar como metadatos en la aplicación.

Esta es una distinción relativamente sutil, pero tendrá profundas implicaciones para los desarrolladores. Si el requisito se establece 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 declaración es "Solo los usuarios autorizados pueden acceder a un registro de empleado", el desarrollador probablemente diseñará e implementará 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, la recopilación de requisitos de esta manera lo lleva naturalmente a un sistema que está bien factorizado para admitir 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 habilidad de elegir, pero están usando *cuadro de lista* como ejemplo, entonces puede que no lo sea. El recuadro de la página 205 analiza un proyecto que salió terriblemente mal porque se ignoraron las necesidades de interfaz de los usuarios.

Es importante descubrir la razón subyacente por qué los usuarios hacen una cosa en particular, en lugar de simplemente la manera actualmente lo hacen. Al final del día, su desarrollo tiene que resolver su problema de negocios, no solo cumplir con los requisitos establecidos. Documentar las razones detrás de los requisitos brindará a su equipo información invaluable al momento de tomar decisiones de implementación diarias.

Existe una técnica simple para conocer los requisitos de los usuarios que no se utiliza con la suficiente frecuencia: convertirse en usuario. ¿Estás escribiendo un sistema?

para la mesa de ayuda? Pase un par de días monitoreando los teléfonos con una persona de soporte experimentada. ¿Está automatizando un sistema de control de stock manual? Trabajar en el almacén durante una semana.¹ Además de darle una idea de cómo funcionará el sistema *De Verdad* ser utilizado, se sorprendería de cómo la solicitud "¿Puedo sentarme durante una semana mientras haces tu trabajo?" ayuda a generar confianza y establece una base para la comunicación con sus usuarios. ¡Solo recuerda no estorbar!

TIP52

Trabajar con un usuario para pensar como un usuario

El proceso de extracción de requisitos también es el momento de comenzar a construir una relación con su base de usuarios, aprendiendo sus expectativas y esperanzas para el sistema que está construyendo. Ver *Grandes expectativas*, página 255, para obtener más información.

Requisitos de documentación

Así que se está sentando con los usuarios y extrayéndoles requisitos genuinos. Se encuentra con algunos escenarios probables que describen lo que debe hacer la aplicación. Siempre que sea un profesional, querrá anotarlos y publicar un documento que todos puedan usar como base para las discusiones: los desarrolladores, los usuarios finales y los patrocinadores del proyecto.

Esa es una audiencia bastante amplia.

Ivar Jacobson [Jac94] propuso el concepto de *casos de uso* para capturar requerimientos. Te permiten describir un particular *usuario* del sistema no en términos de interfaz de usuario, sino de una manera más abstracta. Desafortunadamente, el libro de Jacobson fue 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 parece mucho tiempo? Realmente no lo es, en particular cuando observa procesos en los que la gerencia y los trabajadores ocupan mundos diferentes. La gerencia le dará una visión de cómo funcionan las cosas, pero cuando se siente en el piso, encontrará una realidad muy diferente, una que le llevará tiempo asimilar.

A veces la interfaz *Es el sistema*

En un artículo en *cableadorevista* (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 al sonido que se puede hacer. Y, sin embargo, en lugar de dejar que los músicos hagan mejor música o produzcan una grabación más rápido 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 circuito de retroalimentación innato entre sus oídos y las yemas de sus dedos: faders deslizantes, perillas giratorias, etc. Sin embargo, la interfaz del nuevo mezclador no aprovechó esas capacidades. En cambio, obligó a sus usuarios a escribir en un teclado o hacer clic con el mouse. Las funciones que proporcionaba eran completas, pero estaban empaquetadas en formas desconocidas y exóticas. Las funciones que necesitaban los ingenieros a veces se escondían 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 la duplicación servil de lo que ya existe no permite el progreso, debemos ser capaces de proporcionar una transición al futuro.

Por ejemplo, los ingenieros de grabación pueden haber estado mejor atendidos por algún tipo de interfaz de pantalla táctil, todavía táctil, todavía montada como podría estar una mesa de mezclas tradicional, pero que permite que el software vaya 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 impulsada por objetivos. Alistair Cockburn tiene un documento 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 en la página siguiente muestra un ejemplo abreviado de su plantilla, mientras que la Figura 7.2 muestra su caso de uso de muestra.

Mediante el uso de una plantilla formal como una *ayuda memoria*, 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 CARACTERÍSTICA

- Gol en contexto
- Alcance
- Nivel
- Condiciones previas
- Condición final de éxito
- Condición final fallida
- Actor principal
- Generar

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
- Canal al actor principal
- Actores secundarios
- Canalizar 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 comentarios de usuarios como "oh, excepto si obtenemos unxxx condición, entonces tenemos que haceryyy en cambio." La plantilla también sirve como una agenda preparada para 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, *cargo posterior y publicar crédito* o ambos elaboran *transacción posterior*.

Diagramas de casos de uso

El flujo de trabajo se puede capturar con diagramas de actividad UML, y los diagramas de clase 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

UNA. INFORMACIÓN CARACTERÍSTICA

- **Meta en contexto:** El comprador emite la solicitud directamente a nuestra empresa, espera que los productos se envíen y se facturen.
- **Alcance:** Compañía
- **Nivel:** Resumen
- **Condiciones previas:** Conocemos al comprador, su dirección, etc.
- **Condición final de éxito:** El comprador tiene bienes, tenemos dinero para los bienes.
- **Condición final fallida:** No hemos enviado la mercancía, el comprador no ha enviado el dinero.
- **Actor principal:** Comprador, cualquier agente (o computadora) que actúe en nombre del cliente
- **Generar:** Entra solicitud de compra.

B. PRINCIPAL ESCENARIO DE ÉXITO

1. El comprador llama con una solicitud de compra.
2. La empresa captura el nombre del comprador, la dirección, los bienes solicitados, etc.
3. La empresa brinda información al comprador sobre bienes, 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 no tiene uno de los artículos pedidos: Renegociar pedido.
- 4a. El comprador paga directamente con tarjeta de crédito: Tome el pago con tarjeta de crédito (utilice caso 44).
- 7a. Comprador devuelve bienes: Manejar bienes devueltos (caso de uso 105).

D. VARIACIONES

1. El comprador puede usar el 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:** Parte superior
- **Objetivo de rendimiento:** 5 minutos para el pedido, 45 días hasta que se pague
- **Frecuencia:** 200/día
- **Caso de uso superior:** Gestionar la relación con el cliente (caso de uso 2). **Casos de uso subordinados:** Crear pedido (15). Acepta el pago con tarjeta de crédito (44). Gestionar mercancías devueltas (105).
- **Canal al actor principal:** Puede ser teléfono, archivo o interactivo **Actores secundarios:** Compañía de tarjeta de crédito, banco, servicio de envío

F. CALENDARIO

- **Fecha de vencimiento:** Versión 1.0

GRAMO. PROBLEMAS ABIERTOS

- ¿Qué pasa si tenemos parte del pedido? ¿Qué pasa si me roban la tarjeta de crédito?

Figura 7.3. Casos de uso de UML: ¡tan simple que un niño podría hacerlo!



a mano. Pero los verdaderos casos de uso son 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 considere seriamente documentar información tan densa utilizando solo personas de palo simplistas como la Figura 7.3. No seas esclavo de ninguna notación; use el método que mejor comunique los requisitos a su audiencia.

sobreespecificar

Un gran peligro al producir un documento de requisitos es ser demasiado específico. Los buenos documentos de requisitos siguen siendo abstractos. En lo que respecta a los requisitos, la declaración más simple que refleje con precisión la necesidad comercial es la mejor. 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 diseño, ni son la interfaz de usuario. Los requisitos son *necesitar*.

Ver más lejos

El problema del año 2000 a menudo se atribuye a los programadores miope, desesperados por ahorrar unos pocos bytes en los días en que los mainframes tenían menos memoria que un control remoto de TV 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 Y2K surgió por dos causas principales: la falta de visión más allá de las prácticas comerciales actuales y la violación de la *SECO* principio.

Las empresas usaban el atajo de dos dígitos mucho antes de que las computadoras aparecieran en escena. Era una práctica común. Las primeras aplicaciones de procesamiento de datos simplemente automatizaban los procesos comerciales existentes y simplemente repetían el error. Incluso si la arquitectura requería años de dos dígitos para la entrada, el informe y el almacenamiento de datos, debería haber una abstracción de unFECHA que "sabía" que los dos dígitos eran una forma abreviada de la fecha real.

TIP53

Las abstracciones viven más que los detalles

¿"Ver más allá" requiere que usted prediga el futuro? No. Significa generar declaraciones como

El sistema hace un uso activo de una abstracción de FECHAS. El sistema implementará servicios DATE, como formateo, almacenamiento y operaciones matemáticas, de manera uniforme y universal.

Los requisitos especificarán únicamente que se utilizan fechas. Puede insinuar que se pueden hacer algunas matemáticas en las fechas. Puede indicarle que las fechas se almacenarán en varias formas de almacenamiento secundario. Estos son requisitos genuinos para unFECHAmódulo o clase.

Solo una menta más delgada como una oblea. . .

Muchas fallas de los proyectos se atribuyen a un aumento en el alcance, también conocido como aumento de características, aumento de características o aumento de requisitos. Este es un aspecto del síndrome de la rana hervida de *Sopa de piedra y ranas hervidas*, página 7. ¿Qué podemos hacer para evitar que los requisitos se nos acerquen sigilosamente?

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 administrar el crecimiento de los requisitos es señalar el impacto de cada característica nueva en el cronograma a los patrocinadores del proyecto. Cuando el proyecto tiene un año de retraso desde 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 función más", pero al rastrear los requisitos puede obtener una imagen más clara de que "solo una función más" es realmente la decimoquinta función nueva agregada este mes.

mantener un glosario

Tan pronto como comience a hablar sobre los requisitos, los usuarios y los expertos en el dominio usarán ciertos términos que tienen un significado específico para ellos. Pueden diferenciar entre un "cliente" y un "cliente", por ejemplo. Entonces sería inapropiado usar cualquiera de las palabras casualmente en el sistema.

Crear y mantener un *glosario de proyectos*—un lugar que define todos los términos y vocabulario específicos utilizados en un proyecto. Todos los participantes en el proyecto, desde los usuarios finales hasta el personal de apoyo, deben usar el glosario para garantizar la coherencia. Esto implica que el glosario debe ser ampliamente accesible, un buen argumento para la documentación basada en la Web (más sobre esto en un momento).

TIP54

Usar un glosario de proyectos

Es muy difícil tener éxito en un proyecto en el que los usuarios y los desarrolladores se refieren a lo mismo con nombres diferentes o, peor aún, se refieren a cosas diferentes con el mismo nombre.

Correr la voz

En *todo es escritura*, página 248, analizamos la publicación de los documentos del proyecto en sitios web internos para facilitar el acceso de todos los participantes. Este método de distribución es particularmente útil para los documentos de requisitos.

Al presentar los requisitos como un documento de hipertexto, podemos abordar mejor las necesidades de una audiencia diversa: podemos brindarle a cada lector lo que necesita.

ellos quieren. Los patrocinadores del proyecto pueden navegar a un alto nivel de abstracción para garantizar que se cumplan los objetivos comerciales. Los programadores pueden usar hipervínculos para "profundizar" a niveles crecientes de detalle (incluso haciendo referencia a definiciones apropiadas o especificaciones de ingeniería).

La distribución basada en la Web también evita el típico archivador de dos pulgadas de grosor titulado *Análisis de requerimientos* que nadie nunca lee y que se vuelve obsoleto en el instante en que la tinta llega al papel.

Si está en la Web, los programadores pueden incluso leerlo.

Las secciones relacionadas incluyen:

- *Sopa de piedra y ranas hervidas*, página 7
- *Software suficientemente bueno*, página 9
- *círculos y flechas*, página 220 *todo es escritura*,
- página 248 *Grandes expectativas*, página 255
-

Desafíos

- ¿Puedes usar el software que estás escribiendo? ¿Es posible tener una buena idea de los requisitos? *sin querer* capaz de utilizar el software usted mismo?
- Elija un problema no relacionado con la computadora que actualmente necesita resolver. Generar requerimientos para una solución no informática.

Ejercicios

42. ¿Cuáles de los siguientes son probablemente requisitos genuinos? Repita los que no lo son para hacerlos más útiles (si es posible).

Responder
en 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 ingresa 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 256kB.

Resolviendo rompecabezas imposibles

Gordius, el rey de Frigia, hizo una vez un nudo que nadie podía desatar. Se decía que quien resolviera el enigma del Nudo Gordiano gobernaría toda Asia. Entonces llega Alejandro Magno, que corta el nudo en pedazos con su espada. Solo una interpretación un poco diferente de los requisitos, eso es todo y terminó gobernando la mayor parte de Asia.

De vez en cuando, se verá envuelto en medio de un proyecto cuando surja un rompecabezas realmente difícil: alguna pieza de ingeniería que simplemente no puede manejar, o tal vez algún fragmento de código que resulta ser mucho más difícil de escribir de lo que pensabas. Tal vez parezca imposible. Pero, ¿realmente es tan difícil como parece?

Considera los rompecabezas del mundo real: esos pequeños y tortuosos pedazos 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 las personas intenten lo mismo, una y otra vez, pensando que debe haber una manera.

Por supuesto, no lo hay. La solución está en otra parte. El secreto para resolver el rompecabezas es identificar las restricciones reales (no imaginarias) y encontrar una solución en ellas. Algunas restricciones son *absoluto*; otros son simplemente *nociónes preconcebidas*. Restricciones absolutas *deberían* ser honrados, por desagradables o estúpidos que puedan parecer. Por otro lado, algunas restricciones aparentes pueden no ser restricciones reales en absoluto. Por ejemplo, existe ese viejo truco de bar en el que tomas 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 engañosos.

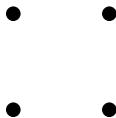
Grados de libertad

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 precisa. Si la "caja" es el límite de las restricciones y condiciones, entonces el truco consiste en *enfiDakota del Nort* la caja, que puede ser considerablemente más grande de lo que piensas.

La clave para resolver acertijos es tanto reconocer las restricciones que se le imponen como reconocer los grados de libertad que necesita. *hacertienes*, porque en ellos encontrarás tu solución. Por eso algunos acertijos son tan efectivos; puede descartar soluciones potenciales con demasiada facilidad.

Por ejemplo, ¿puede conectar todos los puntos en el siguiente rompecabezas y regresar al punto de partida con solo tres líneas rectas, sin levantar la pluma del papel ni volver sobre sus pasos [Hol78]?



Debe desafiar las nociones preconcebidas y evaluar si son o no restricciones reales y estrictas.

No se trata de si piensas dentro o fuera de la caja. El problema radica en *encontrarla caja*—identificando las restricciones reales.

TIP55

No piense fuera de la caja—*Encontrarla caja*

Cuando se enfrente a un problema intratable, enumere *todas* las posibles vías que tienes ante ti. No descarte nada, no importa cuán inútil o estúpido suene. Ahora repase la lista y explique por qué no se puede tomar cierto camino. ¿Está seguro? Puede *demonstrar* eso?

Considere el caballo de Troya: una solución novedosa para un problema intratable. ¿Cómo metes tropas en una ciudad amurallada sin que te descubran? Puede apostar que "a través de la puerta principal" fue descartado inicialmente como suicidio.

Categorice y priorice sus limitaciones. 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, queremos identificar primero 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 se sienta como si estuviera yendo por el camino equivocado, ¡que debe haber una manera más fácil que esta! Tal vez ahora se está retrasando en el cronograma, o incluso se está desesperando por lograr que el sistema funcione porque este problema en particular es "imposible".

Ahí es cuando das un paso atrás y te haces estas preguntas:

- *¿Hay una manera más fácil?*
- *¿Está tratando de resolver el problema correcto o se ha distraído por un tecnicismo periférico?*
- *Por qué*? Esto es un problema?
- *¿Qué* es lo que hace que sea tan difícil de resolver?
- *¿Tiene que hacerse de esta manera?*
- *¿Tiene que hacerse en absoluto?*

Muchas veces te llegará una revelación sorprendente mientras tratas de responder a una de estas preguntas. Muchas veces, una reinterpretación de los requisitos puede hacer desaparecer toda una serie de problemas, como el nudo gordiano.

Todo lo que necesita son las restricciones reales, las restricciones engañosas y la sabiduría para reconocer la diferencia.

Desafíos

- Eche un vistazo detenidamente a cualquier problema difícil en el que se encuentre envuelto hoy. ¿Puedes cortar el nudo gordiano? Hágase las preguntas clave que describimos anteriormente, especialmente “*¿Tiene que hacerse de esta manera?*”
- ¿Se le entregó un conjunto de restricciones cuando se inscribió en su proyecto actual? ¿Siguen siendo aplicables todos, y sigue siendo válida su interpretación?

38

No hasta que estés listo

El que duda a veces se salva.

► James Thurber, *El vaso en el campo*

Los grandes artistas comparten un rasgo: saben cuándo empezar y cuándo esperar. El clavadista se para en el tablero alto, esperando el momento perfecto para saltar. El director se para frente a la orquesta, con los brazos en alto, hasta que ella intuye que es el momento adecuado para comenzar la pieza.

Eres un gran intérprete. Tú también necesitas escuchar la voz que susurra "espera". Si se sienta para comenzar a escribir y hay alguna duda persistente en su mente, preste atención.

TIP56

Escuche Dudas persistentes: comience cuando esté listo

Solía haber un estilo de entrenamiento de tenis llamado "tenis interno". Pasarías horas golpeando pelotas sobre la red, sin tratar particularmente de precisión, sino verbalizando dónde golpeó 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 mejorases 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 sienta una duda persistente, o experimente cierta reticencia al enfrentarse a una tarea, preste atención. Es posible que no pueda identificar exactamente lo que está mal, pero déle tiempo y sus dudas probablemente se cristalicen en algo más sólido, algo que pueda abordar. El desarrollo de software aún no es una ciencia. Deje que sus instintos contribuyan a su desempeño.

¿Buen juicio o procrastinación?

Todo el mundo le 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 aplazar el compromiso inicial de

comenzando. Entonces, ¿cómo puedes saber cuándo simplemente estás procrastinando, en lugar de esperar responsablemente a que todas las piezas encajen en su lugar?

Una técnica que nos ha funcionado en estas circunstancias es empezar a hacer prototipos. Elija un área que crea que será difícil y comience a producir algún tipo de prueba de concepto. Por lo general, sucederá una de dos cosas. Poco después de comenzar, puede sentir que está perdiendo el tiempo. Este aburrimiento es probablemente una buena indicación de que su desgana inicial era solo un deseo de posponer el compromiso de comenzar. Abandone el prototipo y acceda al desarrollo real.

Por otro lado, a medida que avanza el prototipo, es posible que tengas uno de esos momentos de revelación cuando de repente te das cuenta de que alguna premisa básica estaba equivocada. No solo eso, sino que verá claramente cómo puede corregirlo. Te sentirás cómodo abandonando el prototipo y adentrándote en el proyecto propiamente dicho. Tus instintos eran correctos, y acabas de ahorrarte a ti y a tu equipo una cantidad considerable de esfuerzo desperdiciado.

Cuando tome la decisión de crear prototipos como una forma de investigar su malestar, asegúrese de recordar por qué lo está haciendo. Lo último que desea es encontrarse varias semanas en un desarrollo serio antes de recordar que comenzó a escribir un prototipo.

Con cierto cinismo, comenzar a trabajar en un prototipo también podría ser más aceptable políticamente que simplemente anunciar que "no me siento bien al comenzar" y encenderlo.solitario.

Desafíos

- Discuta el síndrome de miedo a empezar con sus colegas. ¿Otros experimentan lo mismo? ¿Le hacen caso? ¿Qué trucos utilizan para superarlo? ¿Puede un grupo ayudar a superar la renuencia de un individuo, o es solo presión de grupo?

La trampa de las especificaciones

El Piloto de Aterrizaje es el Piloto No Handling hasta la llamada de 'altitud de decisión', cuando el Piloto Handling No Aterrizaje entrega el manejo al Piloto Non Handling Landing, a menos que este último diga 'motor y al aire', en cuyo caso el Handling El piloto sin aterrizaje continúa con el manejo y el piloto con aterrizaje sin manejo continúa sin manejo hasta la próxima llamada de 'aterrizar' o 'motor y al aire', según corresponda. En vista de las confusiones recientes sobre estas reglas, se consideró necesario reformularlas claramente.

► **Memorándum de British Airways, citado en revista piloto, diciembre de 1996**

La especificación del programa es el proceso de tomar un requisito y reducirlo hasta el punto en que la habilidad de un programador pueda hacerse cargo. Es un acto de comunicación, explicando y clarificando el mundo de tal manera que se eliminen 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 se fije en detalles insoportables, no han ganado su dólar diario.

Esto es un error por varias razones. En primer lugar, es ingenuo suponer que una especificación capturará cada detalle y matiz de un sistema o sus requisitos. En 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 estropea 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 entienden el requisito y que aprueben el documento de 200 páginas que usted produce, pero puede garantizar que una vez que vean el sistema en ejecución, se verá inundado con solicitudes de cambio.

En segundo lugar, existe 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 a realizar.² Y el lenguaje natural realmente no está a la altura del trabajo. Mire la redacción de cualquier contrato: en un intento de ser precisos, los abogados tienen que modificar 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. ¡Adelante, pruébalo!

Si eres como nosotros, probablemente te hayas dado por vencido en algún lugar alrededor de "ahora gira el pulgar y el índice para que el extremo libre pase por debajo y dentro del cordón izquierdo".... Es algo fenomenalmente difícil de hacer. Y, sin embargo, la mayoría de nosotros podemos atarnos los zapatos sin un pensamiento consciente.

TIP57

Algunas cosas están mejor hechas que descritas

Finalmente, está el efecto de camisa de fuerza. Un diseño que no deja espacio para la interpretación del codificador le roba al programador cualquier habilidad y arte. Algunos dirían que esto es lo mejor, pero están equivocados. A menudo, solo durante la codificación se hacen evidentes ciertas opciones. Mientras codificas, puedes pensar *"Mira eso. Debido a la forma particular en que codifiqué esta rutina, pude agregar esta funcionalidad adicional casi sin esfuerzo"* o *"La especificación dice que haga esto, pero podría lograr un resultado casi idéntico si lo hiciera de una manera diferente, y podría hacerlo en la mitad del tiempo"*. Claramente, no debería simplemente piratear y hacer los cambios, sino que ni siquiera habría detectado la oportunidad si estuviera limitado por un diseño demasiado prescriptivo.

Como programador pragmático, debe tender a ver la recopilación, el diseño y la implementación de requisitos como diferentes facetas del mismo proceso: la entrega de un sistema de calidad. Desconfíe de los entornos donde se recopilan los requisitos, se escriben las especificaciones y luego comienza la codificación, todo de forma aislada. En su lugar, intente adoptar un enfoque uniforme: la especificación y la implementación son simplemente aspectos diferentes del mismo proceso: un intento de capturar y codificar un requisito. cada uno debe

2. Hay algunas técnicas formales que intentan expresar operaciones algebraicamente, pero estas técnicas rara vez se usan en la práctica. Todavía requieren que los analistas expliquen el significado a los usuarios finales.

fluir directamente al siguiente, sin límites artificiales. Descubrirá que un proceso de desarrollo saludable fomenta la retroalimentación de la implementación y las pruebas en el proceso de especificación.

Para que quede claro, no estamos en contra de generar especificaciones. De hecho, reconocemos que hay momentos en los que se exigen especificaciones increíblemente detalladas, por razones contractuales, por el entorno en el que trabaja o por la naturaleza del producto que está desarrollando.³ Solo tenga en cuenta que llega a un punto de rendimiento decreciente, o incluso negativo, 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 de soporte o creación de prototipos; es demasiado fácil especificar algo que no se puede construir.

Cuanto más tiempo permita que las especificaciones sean mantas de seguridad, protegiendo a los desarrolladores del aterrador mundo de escribir código, más difícil será pasar a piratear el código. No caiga en esta espiral de especificaciones: ¡en algún momento, debe comenzar a codificar! Si encuentra a su equipo envuelto en especificaciones cálidas y cómodas, divídálos. Mire la creación de prototipos o considere el desarrollo de una bala trazadora.

Las secciones relacionadas incluyen:

- *Balas trazadoras*, página 48

Desafíos

- El ejemplo del cordón de los zapatos mencionado en el texto es una ilustración interesante de los problemas de las descripciones escritas. ¿Consideró describir el proceso usando diagramas en lugar de palabras? ¿Fotografías? ¿Alguna notación formal de topología? ¿Modelos con cordones de alambre? ¿Cómo enseñarías a un niño pequeño?

A veces una imagen vale más que cualquier número de palabras. A veces no vale nada. Si te das cuenta de que específicas demasiado, ¿te 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 su 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

[fotografías] con círculos y flechas y un párrafo en el reverso de cada una explicando qué era cada una, para ser usadas como prueba en nuestra contra. . .

► Arlo Guthrie, “El restaurante de Alicia”

Desde la programación estructurada, pasando por los equipos de programadores principales, 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, la computación nunca ha carecido de métodos destinados a hacer programación más como 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, quizás sólo el primero, la programación estructurada, ha disfrutado de 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 un naufragio se afellan a la madera flotante que pasa. A medida que cada nueva pieza pasa flotando, nadan dolorosamente, con la esperanza de que sea mejor. Sin embargo, al final del día, no importa cuán buenos sean los restos flotantes, los desarrolladores todavía están a la deriva sin rumbo fijo.

No nos malinterpretes. 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.

TIP58

No sea esclavo de los métodos formales

Los métodos formales tienen algunas deficiencias graves.

- La mayoría de los métodos formales capturan los requisitos mediante 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 existe 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. Vemos algún beneficio en capturar los requisitos de esta manera, pero preferimos, cuando sea posible, mostrarle 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 comunicación deficiente y esfuerzo desperdiciado. También hay una tendencia a volver a caer en *losotros contra ellos* Mentalidad de diseñadores contra codificadores. Preferimos entender la totalidad del sistema en el que estamos trabajando. Puede 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 registro 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 lo desviarán y lo alentarán a establecer relaciones estáticas entre objetos que realmente deberían entrelazarse dinámicamente.

¿Pagan los métodos?

En un artículo del CACM de 1999 [Gla99b], Robert Glass revisa la investigación sobre las mejoras de productividad y calidad obtenidas utilizando siete tecnologías de desarrollo de software diferentes (4GL, técnicas estructuradas, herramientas CASE, métodos formales, metodología de sala limpia, modelos de procesos y orientación a objetos) . Él informa que la exageración inicial que rodeó 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 subestime el costo de adoptar nuevas herramientas y métodos. Esté preparado para tratar los primeros proyectos utilizando estas técnicas como una experiencia de aprendizaje.

¿Deberíamos usar métodos formales?

Absolutamente. Pero recuerde siempre que los métodos formales de desarrollo son solo una herramienta más en la caja de herramientas. Si, después de un análisis cuidadoso, siente que necesita usar un método formal, acéptelo, pero recuerde quién está a cargo. Nunca te vuelvas esclavo de una metodología: círculos y

las flechas hacen amos pobres. 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 mejora cada mes. Esto es crucial. Debe trabajar constantemente para refinar y mejorar sus procesos. Nunca acepte los confines rígidos de una metodología como los límites de su mundo.

No cedas ante la falsa autoridad de un método. Las personas pueden ingresar a las reuniones con un acre de diagramas de clase y 150 casos de uso, pero todo ese papel sigue siendo solo su interpretación falible de los requisitos y el diseño. Trate de no pensar en cuánto cuesta una herramienta cuando observa su rendimiento.

TIP59

Las herramientas caras no producen mejores diseños

Los métodos formales ciertamente tienen su lugar en el desarrollo. Sin embargo, si te encuentras con un proyecto donde la filosofía es “el diagrama de clases es la aplicación, el resto es codificación mecánica”, sabes que estás frente a un equipo de proyecto inundado y una larga remada a casa.

Las secciones relacionadas incluyen:

- *El pozo de requisitos*, página 202

Desafíos

- Los diagramas de casos de uso son parte del proceso UML para recopilar requisitos (ver *El pozo de requisitos*, página 202). ¿Son una forma efectiva de comunicarse con sus usuarios? Si no, ¿por qué los estás usando?
- ¿Cómo puede saber si un método formal está aportando beneficios a su equipo? ¿Qué puedes medir? ¿Qué constituye una mejora? ¿Puede distinguir entre los beneficios de la herramienta y una mayor experiencia por parte de los miembros del equipo?
- ¿Dónde está el punto de equilibrio para introducir nuevos métodos a su equipo? ¿Cómo evalúa la compensación 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 proyectos pequeños? ¿Qué tal si fuera de la otra manera?

Capítulo 8

Proyectos pragmáticos

A medida que su proyecto se pone en marcha, debemos alejarnos de los problemas de filosofía y codificación individuales para hablar sobre problemas más grandes del tamaño del proyecto. No vamos a entrar en detalles específicos de la gestión de proyectos, pero hablaremos sobre 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 *Equipos pragmáticos*, le mostraremos cómo hacer esto respetando la filosofía pragmática.

El factor más importante para hacer 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 *Automatización ubicua*.

Anteriormente, hablamos sobre las pruebas a medida que codifica. En *Pruebas despiadadas*, 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. Ya sea que tenga escritores técnicos que lo ayuden o lo esté haciendo por su cuenta, le mostraremos cómo hacer que la tarea sea menos dolorosa y más productiva en *todo es escritura*.

El éxito está en el ojo del espectador: el patrocinador del proyecto. La percepción del éxito es lo que cuenta, y en *Grandes expectativas* te mostraremos algunos trucos para deleitar al patrocinador de cada proyecto.

El último consejo del libro es una consecuencia directa de todo lo demás. En *Orgullo y prejuicio*, lo alentamos a que firme su trabajo y se enorgullezca de lo que hace.

41

Equipos pragmáticos

En el Grupo L, Stoffel supervisa a seis programadores de primer nivel, un desafío gerencial casi comparable con pastorear gatos.

► ***la revista del poste de washington,9 de junio de 1985***

Hasta ahora en este libro hemos visto técnicas pragmáticas que ayudan a un individuo a ser un mejor programador. ¿Estos métodos también pueden funcionar para equipos?

¡La respuesta es un sí rotundo!" Hay ventajas en ser un individuo pragmático, pero estas ventajas se multiplican si el individuo trabaja en un equipo pragmático.

En esta sección veremos brevemente cómo se pueden aplicar las técnicas pragmáticas a los equipos como un todo. Estas notas son sólo un comienzo. Una vez que tenga un grupo de desarrolladores pragmáticos trabajando en un entorno propicio, desarrollarán y refinará rápidamente su propia dinámica de equipo que funcione para ellos.

Reformulemos algunas de las secciones anteriores en términos de equipos.

Sin 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 desalienta activamente al desarrollador para que dedique tiempo a estas correcciones.

Los equipos en su conjunto no deben tolerar las ventanas rotas, esas pequeñas imperfecciones que nadie arregla. El equipo *deber* responsabilizarse de la calidad del producto, apoyando a los desarrolladores que entienden el *sin ventanas rotas* filosofía que describimos en *Entropía del software*, página 4, y animando a los que aún no lo han descubierto.

Algunas metodologías de equipo tienen un *oficial de calidad*—alguien en 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

Recuerda la pobre rana en la olla de agua, de vuelta en *Sopa de piedra y ranas hervidas*, página 7? No nota el cambio paulatino de su entorno, y acaba cocinado. Lo mismo les puede pasar a las personas que no están atentas. Puede ser difícil mantener un ojo en su entorno general en el fragor del desarrollo del proyecto.

Es incluso más fácil que los equipos en su conjunto se hiervan. Las personas asumen que alguien más está manejando un problema, o que el líder del equipo debe haber aprobado un cambio que solicita su usuario. Incluso los equipos con las mejores intenciones pueden ignorar los cambios significativos en sus proyectos.

Lucha contra esto. Asegúrese de que todos controlen activamente el entorno en busca de cambios. Tal vez designe un *jefe de pruebas de agua*. Haga que esta persona verifique 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 métricas sobre nuevos requisitos (consulte la página 209). No es necesario que el equipo rechace los cambios sin más, simplemente debe ser consciente de que están ocurriendo. De lo contrario, serás tú el agua caliente.

Comunicar

Es obvio que los desarrolladores de un equipo deben hablar entre ellos. Dimos algunas sugerencias para facilitar esto en *Comunicar* en la página 18. Sin embargo, es fácil olvidar que el equipo mismo tiene una presencia dentro de la organización. El equipo como entidad necesita comunicarse claramente con el resto del mundo.

Para los extraños, los peores equipos de proyecto son aquellos que parecen hoscos y reticentes. Hacen reuniones sin estructura, donde nadie quiere hablar. Sus documentos son un desastre: no hay dos que se vean iguales y cada uno usa una terminología diferente.

Los grandes equipos de proyecto tienen una personalidad distinta. La gente espera reunirse con ellos, porque saben que verán un equipo bien preparado

desempeño que hace que todos se sientan bien. La documentación que producen es nítida, precisa y consistente. El equipo habla con una sola voz.¹ Incluso pueden tener sentido del humor.

Hay un truco de marketing simple que ayuda a los equipos a comunicarse como uno solo: generar una marca. Cuando comience un proyecto, piense en un nombre para él, idealmente algo extravagante. (En el pasado, nombramos proyectos con 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 notas e informes. Use el nombre de su equipo generosamente cuando hable con la gente. Suena tonto, pero le da a su equipo una identidad sobre la cual construir, y al mundo algo memorable para asociar con su trabajo.

No te repitas

En *Los males de la duplicación*, página 26, hablamos sobre las dificultades de eliminar el trabajo duplicado entre los miembros de un equipo. Esta duplicación conduce a un esfuerzo desperdiciado y puede resultar en una pesadilla de mantenimiento. Claramente, una buena comunicación puede ayudar aquí, pero a veces se necesita algo extra.

Algunos equipos designan a un miembro como bibliotecario del proyecto, responsable de coordinar los repositorios de documentación y código. Otros miembros del equipo pueden usar a esta persona como el primer puerto de escala cuando buscan algo. Un buen bibliotecario también podrá detectar una 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 personas como puntos focales para varios aspectos funcionales del trabajo. Si las personas quieren hablar sobre el manejo de fechas, deben saber hablar con Mary. Si hay un problema con el esquema de la base de datos, consulte a Fred.

Y no olvide el valor de los sistemas de trabajo en grupo y los grupos de noticias locales de Usenet para comunicarse y archivar preguntas y respuestas.

1. El equipo habla con una sola voz, externamente. Internamente, alentamos encarecidamente un debate vivo y sólido. Los buenos desarrolladores tienden a ser apasionados por su trabajo.

ortogonalidad

La organización tradicional de equipos se basa en el antiguo método de construcción de software en cascada. A los individuos se les asignan roles en función de su función laboral. Encontrará analistas de negocios, arquitectos, diseñadores, programadores, evaluadores, 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 estrictas divisiones de responsabilidad; los codificadores no pueden hablar con los evaluadores, quienes a su vez no pueden hablar con el arquitecto jefe, y así sucesivamente. Algunas organizaciones luego complican 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 muchos 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.

TIP60

Organícese en función de la funcionalidad, no de las funciones laborales

Estamos a favor de dividir 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 en equipos.

La funcionalidad aquí no significa necesariamente casos de uso del usuario final. La capa de acceso a la base de datos cuenta, al igual que el subsistema de ayuda. Estamos buscando equipos de personas cohesivos y en gran medida autónomos, exactamente el

2. En *El proceso unificado racional: una introducción*, el autor identifica 27 roles separados dentro de un equipo de proyecto. [Kru98]

mismos criterios que deberíamos usar cuando modularizamos el código. Hay señales de advertencia de que la organización del equipo es incorrecta; un ejemplo clásico es tener dos subequipos trabajando en el mismo módulo o clase del programa.

¿Cómo ayuda este estilo funcional de organización? Organizar nuestros recursos usando las mismas técnicas que usamos para organizar código, usando técnicas como contratos (*Diseño por contrato*, página 109), desacoplamiento (*El desacoplamiento y la Ley de Deméter*, página 138), y ortogonalidad (*ortogonalidad*, 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 marketing de repente decida utilizar una herramienta lista para usar para la función de calendario, el grupo de calendario recibe un golpe. Ejecutado correctamente, este tipo de enfoque grupal puede reducir drásticamente la cantidad de interacciones entre el trabajo de los individuos, reducir las escalas de tiempo, aumentar la calidad y reducir la cantidad de defectos. Este enfoque también puede conducir a un conjunto más comprometido de desarrolladores. Cada equipo sabe que solo 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 "jefes", uno técnico y otro administrativo. 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 el panorama general, tratando de encontrar puntos en común innecesarios entre los equipos que podrían reducir la ortogonalidad del esfuerzo general. El jefe administrativo, o jefe de proyecto, programa los recursos que necesitan los equipos, supervisa e informa sobre el progreso y ayuda a decidir las prioridades en términos de necesidades comerciales.

Los equipos en proyectos más grandes necesitan recursos adicionales: un bibliotecario que indexe y almacene código y documentación, un creador de herramientas que proporcione herramientas y entornos comunes, soporte operativo, etc.

Este tipo de organización de equipo es similar en espíritu al antiguo concepto de equipo de programadores jefe, documentado por primera vez en 1972 [Bak72].

Automatización

Una excelente manera de garantizar tanto la consistencia como la precisión es automatizar todo lo que hace el equipo. ¿Por qué diseñar el código manualmente cuando su editor puede hacerlo automáticamente mientras escribe? ¿Por qué completar formularios de prueba cuando la compilación nocturna puede ejecutar pruebas automáticamente?

La automatización es un componente esencial de cada equipo de proyecto, lo suficientemente importante como para que le dediquemos una sección completa, comenzando en la página siguiente. Para asegurarse de que las cosas se automaticen, designe a uno o más miembros del equipo como *constructores de herramientas* para construir e implementar las herramientas que automatizan el trabajo pesado del proyecto. Pídale que produzcan makefiles, scripts de shell, plantillas de edición, 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. Ofrézcales la estructura suficiente para apoyarlos y garantizar que el proyecto cumpla con sus requisitos. Entonces, como el pintor en *Software suficientemente bueno*, página 11, resista la tentación de agregar más pintura.

Las secciones relacionadas incluyen:

- *Entropía del software*, página 4
- *Sopa de piedra y ranas hervidas*, página 7
- *Software suficientemente bueno*, página 9
- *¡Comunicar!*, página 18
- *Los males de la duplicación*, página 26
- *ortogonalidad*, página 34
- *Diseño por contrato*, página 109
- *El desacoplamiento y la Ley de Deméter*, página 138
- *Automatización ubicua*, página 230

Desafíos

- Busque equipos exitosos fuera del área de desarrollo de software. ¿Qué los hace exitosos? ¿Usan alguno de los procesos discutidos en esta sección?

- La próxima vez que inicie un proyecto, intente convencer a la gente para que lo marque. Dele tiempo a su organización para que se acostumbre a la idea y luego realice una auditoría rápida para ver qué diferencia hizo, tanto dentro del equipo como externamente.
- Equipo de Álgebra: En la escuela, nos dan problemas como "Si 4 trabajadores tardan 6 horas en cavar una zanja, ¿cuánto tiempo tardarán 8 trabajadores?" Sin embargo, en la vida real, ¿qué factores afectan la respuesta a: "Si a 4 programadores les toma 6 meses desarrollar una aplicación, ¿cuánto tiempo les tomaría a 8 programadores?" ¿En cuántos escenarios se reduce realmente el tiempo?

42

Automatización ubicua

La civilización avanza ampliando el número de operaciones importantes que podemos realizar sin pensar.

► **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 automóviles modernos, simplemente gira la llave: el procedimiento de arranque es automático e infalible. Una persona que sigue una lista de instrucciones puede inundar el motor, pero el motor de arranque automático no.

Aunque la informática es todavía una industria en la etapa del Modelo-T, no podemos darnos el lujo de leer dos páginas de instrucciones una y otra vez para alguna operación común. Ya sea que se trate del procedimiento de compilación y lanzamiento, el papeleo de revisión de código o cualquier otra tarea recurrente en el proyecto, tiene que ser automático. Es posible que tengamos que construir el motor de arranque y el inyector de combustible desde cero, pero una vez que esté hecho, podemos simplemente 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 los aspectos del procedimiento están abiertos a la interpretación de diferentes personas.

Todo en Automático

Una vez estábamos en el sitio de un cliente donde todos los desarrolladores usaban el mismo IDE. Su administrador del sistema le 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 eso y vuelva a hacerlo.

No es sorprendente que la máquina de cada desarrollador se cargara de forma ligeramente diferente. Sutiles diferencias en el comportamiento de la aplicación ocurrieron cuando diferentes desarrolladores ejecutaron el mismo código. Los errores aparecerían en una máquina pero no en otras. Rastrear las diferencias de versión de cualquier componente generalmente revelaba una sorpresa.

TIP61

No use procedimientos manuales

Las personas simplemente no son tan repetibles como 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 fuente, por lo que también puede examinar los cambios en el procedimiento a lo largo del tiempo ("pero *usó* trabajar...").

Otra herramienta favorita de automatización es cron (o "a" en Windows NT). Nos permite programar tareas desatendidas para que se ejecuten periódicamente, generalmente en medio de la noche. Por ejemplo, el siguiente crontab file especifica que un proyecto nocturno ejecutar el comando a las 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 reporte de gastos ejecutarse a la medianoche del primer día del mes.

#	MIN	HORA	DÍA	MES	DÍA DE LA SEMANA	DOMINIO
#-----	0	*	*	*	*	-----5
15	3	*	*	1-5		/proyectos/Manhattan/bin/nightly /usr/local/bin/backup
0	0	1	*	*		/inicio/contabilidad/informes_de_gastos

Usando cron, podemos programar copias de seguridad, la compilación nocturna, el mantenimiento del sitio web y cualquier otra cosa que deba hacerse, sin supervisión, automáticamente.

Compilando el Proyecto

Compilar el proyecto es una tarea que debe ser confiable y repetible. Generalmente compilamos proyectos con makefiles, incluso cuando usamos un entorno IDE. Hay varias ventajas en el uso de archivos MAKE. Es un procedimiento automático con guión. Podemos agregar ganchos para generar código para nosotros y ejecutar pruebas de regresión automáticamente. Los IDE tienen sus ventajas, pero solo con los IDE puede ser difícil lograr el nivel de automatización que estamos buscando. Queremos verificar, construir, probar y enviar con un solo comando.

Código de generación

En *Los males de la duplicación*, página 26, abogábamos por generar código para derivar conocimiento de fuentes comunes. podemos explotar hacer's mecanismo de análisis de dependencia para facilitar este proceso. Es un asunto bastante simple agregar reglas a un archivo MAKE para generar un archivo de alguna otra fuente automáticamente. 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 convertir.pl $<          > $@
.java.clase:
    $(JAVAC) $(JAVAC_BANDERAS)      $<
```

Escribir hacer prueba.clase, y hacer buscará automáticamente un archivo llamado prueba.xml, construir un .java-fíarchivo mediante la ejecución de un script de Perl, y luego compilar ese archivo para producir prueba.clase.

También podemos usar el mismo tipo de reglas para generar código fuente, archivos de encabezado o documentación automáticamente desde algún otro formulario (ver *Generadores de código*, página 102).

Pruebas de regresión

También puede usar el archivo MAKE para ejecutar pruebas de regresión por usted, ya sea para un módulo individual o para un subsistema completo. Puede probar fácilmente el *completoproyecto* con solo un comando en la parte superior del árbol de fuentes, o puede probar un módulo individual usando el mismo comando en un solo directorio. Ver *Pruebas despiadadas*, página 237, para obtener más información sobre las pruebas de regresión.

marca recursiva

Muchos proyectos configuran makefiles recursivos y jerárquicos para compilaciones y pruebas de proyectos. Pero tenga en cuenta algunos problemas potenciales.

hacer calcula las dependencias entre los distintos objetivos que tiene que construir. Pero solo puede analizar las dependencias que existen dentro de un solo hacerinvocación. En particular, una recursivahacer no tiene conocimiento de las dependencias que otras invocaciones de hacerpuede tener. Si es cuidadoso y preciso, puede obtener los resultados adecuados, pero es fácil generar trabajo adicional innecesariamente, o pasar por alto una dependencia y norecompilar cuando sea necesario.

Además, las dependencias de compilación pueden no ser las mismas que las dependencias de prueba y es posible que necesite jerarquías separadas.

Automatización de compilación

A construir es un procedimiento que toma un directorio vacío (y un entorno de compilación conocido) y crea el proyecto desde cero, produciendo lo que espera producir como entrega final: una imagen maestra de CD-ROM o un archivo autoextraíble, por ejemplo. Por lo general, la construcción de un proyecto abarcará los siguientes pasos.

1. Consulte el código fuente del repositorio.
2. Genere el proyecto desde cero, generalmente desde un archivo MAKE de nivel superior. Cada compilación está marcada con alguna forma de lanzamiento o número de versión, o tal vez un sello de fecha.
3. Crea una imagen distribuible. Este procedimiento puede implicar la corrección de la propiedad y los permisos de los archivos, y la producción de todos los ejemplos, la documentación, los archivos LÉAME y cualquier otra cosa que se envíe con el producto, en el formato exacto que se requerirá cuando lo envíe.³
4. Ejecute las pruebas especificadas (hacer prueba).

3. Si está produciendo un CD-ROM en formato ISO9660, por ejemplo, debería ejecutar el programa que produce una imagen bit por bit del sistema de archivos 9660. ¿Por qué esperar hasta la noche anterior al 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 pruebas más completas que las que podría ejecutar un individuo mientras construye una parte específica del proyecto. El punto importante es ejecutar la compilación completa *atodospruebas* disponibles. Quiere saber si una prueba de regresión falló debido a uno de los cambios de código de hoy. Al identificar el problema cerca de la fuente, tiene más posibilidades de encontrarlo y solucionarlo.

Cuando no ejecuta pruebas regularmente, puede descubrir que la aplicación se rompió debido a un cambio de código realizado hace tres meses. Buena suerte encontrando ese.

Construcciones finales

Construcciones finales, que pretende 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 los indicadores de optimización y depuración se establezcan de manera diferente, etc. Nos gusta usar un separado hacer objetivo (como `hacer final`) que establece todos estos parámetros a la vez.

Recuerde que si el producto se compila de manera diferente a las versiones anteriores, debe probarlo con *esta* versión de nuevo.

Administración automática

¿No sería bueno si los programadores pudieran dedicar todo su tiempo a la programación? Por desgracia, esto no suele ser el caso. Hay correos electrónicos que responder, papeleo que completar, documentos que publicar 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.

Porque la memoria es lo segundo que pierdes con la edad, no queremos confiar demasiado en él. Podemos ejecutar secuencias de comandos para realizar procedimientos para nosotros de forma automática, en función de la *contenido* de código fuente y documentos. Nuestro objetivo es mantener un flujo de trabajo automático, desatendido y basado en contenido.

4. ¿Cuál es el primero? Yo olvido.

Generación de sitios web

Muchos equipos de desarrollo utilizan un sitio web interno para la comunicación del proyecto y creemos que es una gran idea. Pero no queremos perder demasiado tiempo manteniendo el sitio web, y no queremos que se vuelva obsoleto o desactualizado. La información engañosa es peor que ninguna información en absoluto.

La documentación que se extrae del código, los análisis de requisitos, los documentos de diseño y cualquier dibujo, diagrama o gráfico deben publicarse en la Web periódicamente. Nos gusta publicar estos documentos automáticamente como parte de la compilación nocturna o como un gancho en el procedimiento de verificación del código fuente.

Independientemente de cómo se haga, el contenido web debe generarse automáticamente a partir de la información del repositorio y publicarse *sin que* intervención humana. Esta es realmente otra aplicación de la *SECO* principio: la información existe en una forma como código y documentos registrados. La vista desde el 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 estar 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 incluye advertencias del compilador, errores y estado actual), regresión pruebas, 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, las revisiones de código o diseño deben programarse y seguirse, es posible que sea necesario otorgar aprobaciones, etc. Podemos utilizar la automatización, y especialmente el sitio web, para ayudar a aliviar la carga del papeleo.

Suponga que desea automatizar la programación y aprobación de la revisión del código. Puede colocar un marcador especial en cada archivo de código fuente:

```
/* Estado: necesita_revisión */
```

Un simple script podría revisar todo el código fuente y buscar todos los archivos que tenían un estado de `necesita_revisión`, indicando que estaban listos para ser revisados. A continuación, puede publicar una lista de esos archivos como un

página web, envíe correos electrónicos automáticamente a las personas apropiadas o incluso programe una reunión automáticamente usando algún software de calendario.

Puede configurar un formulario en una página web para que los revisores registren su aprobación o desaprobación. Después de la revisión, el estado se puede cambiar automáticamente a revisados. Depende de usted si tiene un recorrido de código con todos los participantes; usted todavía puede hacer el papeleo de forma automática. (En un artículo del CACM de abril de 1999, Robert Glass resume una investigación que parece indicar que, si bien la inspección del código es eficaz, 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 hacer, Ant y CruiseControl para automatización (ver [Cla04]). Y tenemos Ruby, Perl y otros lenguajes de secuencias de comandos de alto nivel para desarrollar rápidamente herramientas personalizadas, generadores de páginas web, generadores de código, arneses de prueba, etc.

Deje que la computadora haga lo repetitivo, lo mundano; hará un mejor trabajo 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 sin formato*, 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*, página 248
-
-

Desafíos

- Mira tus hábitos a lo largo de la jornada laboral. ¿Ves alguna tarea repetitiva? ¿Escribes la misma secuencia de comandos una y otra vez?

Intenta escribir algunos scripts de shell para automatizar el proceso. ¿Siempre hace clic en la misma secuencia de iconos repetidamente? ¿Puedes crear una macro para hacer todo eso por ti?

- ¿Cuánto del papeleo de su proyecto se puede automatizar? Dado el alto gasto del personal de programación,⁵determinar cuánto del presupuesto del proyecto se está desperdiciando en procedimientos administrativos. ¿Puede justificar la cantidad de tiempo que llevaría diseñar una solución automatizada en función de los ahorros de costos generales que lograría?

43

Pruebas despiadadas

La mayoría de los desarrolladores odian las pruebas. Tienden a probar suavemente, inconscientemente sabiendo dónde se romperá el código y evitando los puntos débiles. Los programadores pragmáticos son diferentes. Estamos *impulsados* para encontrar nuestros errores *ahora*, para que no tengamos que soportar la vergüenza de que otros encuentren nuestros errores más tarde.

Encontrar insectos es algo así como pescar con una red. Usamos 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 reparamos los agujeros que encontramos, con la esperanza de atrapar más y más defectos resbaladizos que nadan en la piscina de nuestro proyecto.

TIP62

Prueba temprano. Prueba a menudo. Prueba automáticamente.

Queremos comenzar 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. Para propósitos de estimación, puede calcular un promedio de la industria de alrededor de US\$100.000 per cápita—eso es salario más beneficios, capacitación, espacio de oficina y 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 automatizadas tienen muchas más posibilidades de éxito. Las pruebas que se ejecutan con cada compilación son mucho más efectivas que los planes de prueba que se encuentran en un estante.

Cuanto antes se encuentre un error, más barato será remediarlo. "Programa un poco, prueba un poco" es un dicho popular en el mundo de Smalltalk, y podemos adoptar ese mantra como propio escribiendo código de prueba al mismo tiempo (o incluso antes) que escribimos el código de producción.

De hecho, un buen proyecto bien puede tener *más* código de prueba que el código de producción. El tiempo que lleva producir este código de prueba vale la pena. Termina siendo mucho más barato a largo plazo y, de hecho, tiene la posibilidad de producir un producto con casi cero defectos.

Además, saber que pasó la prueba le brinda un alto grado de confianza de que una parte del código está "terminada".

TIP63

La codificación no se hace hasta que se ejecutan todas las pruebas

El hecho de que haya terminado de piratear un fragmento de código no significa que pueda ir a decirle a su jefe o a su cliente que *es hecho*. No es. En primer lugar, el código nunca se termina realmente. Más importante aún, no puede afirmar que cualquiera puede utilizarlo hasta que pase todas las pruebas disponibles.

Necesitamos observar 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:

- Examen de la unidad
- 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 y algunos proyectos especializados también requerirán otros tipos de pruebas. Pero nos da un buen punto de partida.

Examen de la unidad

Apueba de unidades código que ejercita un módulo. Cubrimos este tema por sí mismo en *Código que es fácil de probar*, página 189. La prueba unitaria es la base de todas las demás formas de prueba que analizaremos en esta sección. Si las partes no funcionan por sí solas, probablemente no funcionarán bien juntas. Todos los módulos que está utilizando deben pasar sus propias pruebas unitarias antes de que pueda 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

Pruebas de integración 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 lo contrario, la integración se convierte en un caldo de cultivo fértil para los errores. De hecho, a menudo es la mayor fuente de errores en el sistema.

Las pruebas de integración son realmente solo una extensión de las pruebas unitarias que hemos descrito, solo que ahora está probando cómo los subsistemas completos cumplen sus contratos.

Validación y Verificación

Tan pronto como tenga una interfaz de usuario ejecutable o prototipo, debe responder una pregunta muy importante: los usuarios le dijeron lo que querían, pero ¿es eso lo que necesitan?

¿Cumple con los requisitos funcionales del sistema? Esto también necesita ser probado. Un sistema libre de errores que responde a la pregunta incorrecta no es muy útil. Sea consciente de los patrones de acceso de los usuarios finales y

en qué se diferencian de los datos de prueba del desarrollador (para ver un ejemplo, consulte la historia sobre las pinceladas en la página 92).

Agotamiento de recursos, errores y recuperación

Ahora que tiene una idea bastante buena de que el sistema se comportará correctamente en condiciones ideales, necesita descubrir cómo se comportará en condiciones ideales. *mundo real* condiciones. En el mundo real, sus programas no tienen recursos ilimitados; se quedan sin cosas. Algunos límites que su código puede encontrar incluyen:

- Memoria
- Espacio del disco
- Ancho de banda de la CPU
- Hora del reloj de pared
- Ancho de banda del disco
- Ancho de banda de la red
- Paleta de color
- Resolución de video

En realidad, puede verificar si hay fallas en la asignación de memoria o espacio en disco, pero ¿con qué frecuencia prueba los demás? ¿Cabe su aplicación en un 640×480 pantalla con 256 colores? ¿Funcionará en un 1600×1280 pantalla con 24- poco de color sin parecer un sello de correos? ¿Será el lote
¿Finalizar el trabajo antes de que comience el archivo?

Puede detectar limitaciones ambientales, como las especificaciones de video, y adaptarlas según corresponda. Sin embargo, no todas las fallas 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 nada 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 corrector de estilo quería que cambiáramos esta oración a "Si el sistema falla. "Nos... 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 reales, con la cantidad esperada 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.

Mire la usabilidad en términos de factores humanos. ¿Hubo algún malentendido durante el análisis de 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, debe realizar pruebas de usabilidad lo antes posible, mientras todavía hay tiempo para hacer las correcciones. Para proyectos más grandes, es posible que desee traer especialistas en factores humanos. (Al 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 probar

hemos mirado qué Probar. Ahora centraremos nuestra atención en cómo para probar, incluyendo:

- Pruebas de regresión
- Datos de prueba
- Ejercicio de sistemas GUI
- Probando las pruebas
- probando a fondo

Pruebas de diseño/metodología

¿Puedes probar el diseño del código en sí y la metodología que usaste para construir el software? Después de una moda, sí se puede. Lo haces analizando métrica—mediciones de varios aspectos del código. La métrica más simple (y a menudo la menos interesante) es *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, que incluyen:

- Métrica de complejidad ciclomática de McCabe (mide la complejidad de las estructuras de decisión)
- Fan-in de herencia (número de clases base) y fan-out (número de módulos derivados que usan este como parente)
- Conjunto de respuesta (ver *El desacoplamiento y la Ley de Deméter*, página 138)

- Relaciones de acoplamiento de clase (ver [URL 48])

Algunas métricas están diseñadas para darle una "calificación aprobatoria", mientras que otras son útiles solo por comparación. Es decir, calcula estas métricas para cada módulo en el sistema y ve cómo un módulo en particular se relaciona con sus hermanos. Las técnicas estadísticas estándar (como la media y la desviación estándar) se utilizan generalmente aquí.

Si encuentra un módulo cuyas métricas son marcadamente diferentes del resto, debe preguntarse si eso es apropiado. Para algunos módulos, puede estar bien "explotar 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 valores anteriores (o conocidos). Podemos asegurarnos de que los errores que solucionamos hoy no rompieron las cosas que funcionaban ayer. Esta es una importante red de seguridad 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 código nuevo. 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. En realidad, necesitamos usar ambos, porque las diferentes naturalezas de este tipo 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 algún tipo de prototipo. Representa datos de usuario típicos. Las grandes sorpresas vienen cuando descubres qué *tipico* medio. Es muy probable que esto revele defectos y malentendidos en el análisis de requisitos.

Los datos sintéticos se generan artificialmente, quizás bajo ciertas restricciones estadísticas. Es posible que deba utilizar datos sintéticos por cualquiera de los siguientes motivos.

- Necesita muchos datos, posiblemente más de los que podría proporcionar cualquier muestra del mundo real. Es posible que pueda usar los datos del mundo real como una semilla para generar un conjunto de muestras más grande y modificar ciertos campos que deben ser únicos.
- Necesita datos para enfatizar las condiciones de contorno. Estos datos pueden ser completamente sintéticos: campos de fecha que contengan el 29 de febrero de 1999, tamaños de registro enormes o direcciones con códigos postales extranjeros.
- Necesita datos que exhiban ciertas propiedades estadísticas. ¿Quiere ver qué sucede si falla una de cada tres transacciones? ¿Recuerda el algoritmo de clasificación que se ralentiza cuando se entregan datos preordenados? Puede presentar datos en orden aleatorio o ordenado para exponer este tipo de debilidad.

Ejercicio de sistemas GUI

La prueba de sistemas intensivos en GUI a menudo requiere herramientas de prueba especializadas. Estas herramientas pueden basarse en un modelo simple de captura/reproducción de eventos, o pueden requerir scripts escritos especialmente 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 reduce el tamaño de un botón, es posible que la prueba no pueda

encontrarlo, y puede fallar. La mayoría de las herramientas modernas de prueba de GUI utilizan una serie de técnicas diferentes para solucionar este problema e intentan ajustarse a las diferencias menores 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. Desafortunadamente, durante la prueba, no podía simplemente tomar un mapa de bits y comparar el resultado con una ejecución anterior, porque fue 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 (ver *El desacoplamiento y la Ley de Deméter*, página 138) es una prueba más modular. 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 la aplicación *.sin querer tener una GUI presente*. Esta idea es similar a probar primero sus 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 instalada (es probable que los errores hayan sido creados por el código de la interfaz de usuario).

Probando las pruebas

Debido a que no podemos escribir software perfecto, tampoco podemos escribir software de prueba perfecto. Tenemos que probar las pruebas.

Piense en nuestro conjunto de suites de prueba 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, *causael* error deliberadamente y asegúrese de que la prueba se queje. Esto asegura que la prueba detectará el error si sucede de verdad.

TIP64

Use saboteadores para probar sus pruebas

Si usted es *De Verdadserio* acerca de las pruebas, es posible que desee designar a un *saboteador de proyectos*. El papel del saboteador es tomar una copia separada del

árbol fuente, introduzca errores a propósito y verifique que las pruebas los atrapen.

Al redactar pruebas, asegúrese de que las alarmas suenen cuando deben hacerlo.

Probar a fondo

Una vez que esté seguro de que sus pruebas son correctas y encuentre los errores que crea, ¿cómo sabe si ha probado la base del código lo suficientemente a fondo?

La respuesta corta es "no lo harás", y nunca lo harás. Pero hay productos en el mercado que pueden ayudar. Estas *análisis de cobertura* Las herramientas observan su código durante las pruebas y realizan un seguimiento de qué líneas de código se han ejecutado y cuáles no. Estas herramientas lo ayudan a tener una idea general de cuán completas son sus pruebas, pero no espere ver una cobertura del 100%.

Incluso si llegas a cada línea de código, esa no es la imagen completa. Qué es importante es el número de estados que su programa puede tener. Los estados no son equivalentes a líneas de código. Por ejemplo, suponga que tiene una función que toma dos números enteros, cada uno de los cuales puede ser un número del 0 al 999.

```
En tprueba(En ta,En tb){  
    devolvera / (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 $a + b$ sea igual a cero). El simple hecho de saber que ejecutó esta línea de código no le dice eso: necesitaría identificar todos los estados posibles del programa. Desafortunadamente, en general, esta es un *realmente difícil* problema. Duro como en, "El sol será un bullo frío y duro antes de que puedas resolverlo".

Típico y cinco

Probar la cobertura del estado, no la cobertura del código

Incluso con una buena cobertura de código, los datos que utiliza para las pruebas siguen teniendo un gran impacto y, lo que es más importante, la orden en la que atraviesa el código puede tener el mayor impacto de todos.

Cuándo probar

Muchos proyectos tienden a dejar las pruebas para el último minuto, justo donde se cortarán contra el límite de una fecha límite.⁸Tenemos que empezar mucho antes que eso. Tan pronto como exista un código de producción, debe probarse.

La mayoría de las pruebas deben realizarse automáticamente. Es importante tener en cuenta que por "automáticamente" queremos decir que la pruebas *resultados* también se interpretan automáticamente. Ver *Automatización ubicua*, página 230, para más información sobre este tema.

Nos gusta probar con la mayor frecuencia posible y siempre antes de verificar el código en el repositorio fuente. Algunos sistemas de control de código fuente, como Aegis, pueden hacer esto automáticamente. De lo contrario, simplemente escribimos

```
% hacer prueba
```

Por lo general, no es un problema ejecutar regresiones en todas las pruebas unitarias individuales y pruebas de integración con la frecuencia necesaria.

Pero algunas pruebas pueden no ejecutarse fácilmente con tanta frecuencia. Las pruebas de estrés, por ejemplo, pueden requerir una configuración o equipo especial, y algo de sujeción manual. Estas pruebas pueden realizarse con menos frecuencia, quizás semanal o mensualmente. Pero es importante que se ejecuten de forma regular y programada. Si no se puede hacer automáticamente, asegúrese de que aparezca en el cronograma, con todos los recursos necesarios asignados a la tarea.

apretando la red

Finalmente, nos gustaría revelar el concepto más importante en las pruebas. Es obvio, y prácticamente todos los libros de texto dicen que se haga de esta manera. Pero por alguna razón, la mayoría de los proyectos aún 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. **plazode-líñoporte**(1864) una línea trazada dentro o alrededor de una prisión por la que pasa un preso con el riesgo de ser fusilado—*Diccionario colegiado de Webster*.

TIP66

Encuentra errores una vez

Una vez que un evaluador humano encuentra un error, debe ser *el último* que un probador humano encuentra ese error. Las pruebas automatizadas deben modificarse para verificar ese error en particular a partir de ese momento, cada vez, sin excepciones, sin importar cuán trivial sea, y sin importar cuánto se queje el desarrollador y diga: "Oh, eso nunca volverá a suceder".

Porque volverá a pasar. Y simplemente no tenemos tiempo para perseguir errores que las pruebas automatizadas podrían haber encontrado para nosotros. Tenemos que pasar nuestro tiempo escribiendo código nuevo y errores nuevos.

Las secciones relacionadas incluyen:

- *El gato se comió mi código fuente*, página 2
- *depuración*, página 90
- *El desacoplamiento y la Ley de Deméter*, página 138
- *refactorización*, página 184
- *Código que es fácil de probar*, página 189
- *Automatización ubicua*, página 230

Desafíos

- ¿Puedes probar automáticamente tu proyecto? Muchos equipos se ven obligados a responder "no". ¿Por qué? ¿Es demasiado difícil definir los resultados aceptables? ¿No dificultará esto 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 acerca de la GUI? Sobre el acoplamiento?

todo es escritura

La tinta más pálida es mejor que el mejor recuerdo.

► **Proverbio chino**

Por lo general, los desarrolladores no le dan mucha importancia a 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 de desarrollo general. La escritura de la documentación se puede hacer más fácil si no se duplica el esfuerzo ni se pierde el tiempo, y si se mantiene la documentación a mano, en el código mismo, si es posible.

Estos no son pensamientos exactamente originales o novedosos; la idea del código de matrimonio 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 (*veres solo una vista*, página 157). De hecho, queremos ir un poco más allá y aplicar *todos* nuestros principios pragmáticos tanto a la documentación como al código.

TIP67

Tratar el inglés como un lenguaje de programación más

Hay básicamente dos tipos de documentación producida para un proyecto: interna y externa. La documentación interna incluye comentarios del código fuente, documentos de diseño y prueba, etc. La documentación externa es todo lo que se envía o publica al mundo exterior, como los manuales de usuario. Pero independientemente de la audiencia prevista o del rol del escritor (desarrollador o escritor técnico), toda la documentación es un espejo del código. Si hay una discrepancia, el código es lo que importa, para bien o para mal.

TIP68

Incorpore la documentación, no la atornille

Comenzaremos con 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 *tener* 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 *por qué* algo se hace, su propósito y su meta. El código ya aparece *cómo* ya está hecho, por lo que comentar sobre esto es redundante y es una violación de la *SECO* principio.

Comentar el código fuente le brinda la oportunidad perfecta para documentar esas partes escurridizas de un proyecto que no se pueden documentar en ningún otro lugar: compensaciones de ingeniería, por qué se tomaron 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 describe cómo se usa la función y todo lo que hace que no es obvio.

Los nombres de las variables, por supuesto, deben ser bien elegidos y significativos. Foo, por ejemplo, no tiene sentido, como lo es hazloOgerenteocosas. La notación húngara (en la que se codifica la información del tipo de variable en el propio nombre) es completamente inapropiada en los sistemas orientados a objetos. Recuerde que usted (y otros después de usted) serán *lectura* el código cientos de veces, pero sólo *escritura* unas cuantas veces. Tómese el tiempo para deletrear afueraconexionPiscinaen vez de cp.

Incluso peor que los nombres sin sentido son *engañosos* nombres. ¿Alguna vez alguien le ha explicado las inconsistencias en el código heredado como, "La rutina llamada obtener datos realmente escribe datos en el disco"? El cerebro humano estropeará esto repetidamente: se llama el *Efecto Stroop*[Str35]. Puede intentar el siguiente experimento usted mismo para ver los efectos de dicha interferencia. Consigue algunos bolígrafos de colores y utilízalos para escribir los nombres de los colores. Sin embargo, nunca escriba el nombre de un color con ese rotulador de color. Podrías escribir la palabra "azul" en verde, la palabra "café" en rojo, y así sucesivamente. (Alternativamente, tenemos un conjunto de muestra de colores ya dibujado en nuestro sitio web en www.pragmaticprogrammer.com.) Una vez que tengas los nombres de los colores dibujados, trate de decir en voz alta el color con el que se dibuja cada palabra, lo más rápido que pueda. En algún momento te tropezarás y comenzarás 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.

Puedes documentar parámetros, pero pregúntate 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 específica  
 * gama de muestras  
 *  
 * @param aRange Rango de fechas para buscar datos. aUmbral Valor  
 * @param mínimo a considerar.  
 * @return the value, o <code>null</code> si no se encuentra ningún valor  
 * mayor o igual que el umbral.  
 */  
públicoMuestra findPeak(DateRange aRange,doble aUmbral);
```

Aquí hay una lista de cosas que deberían *no* 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 ti. Úselos, y se garantiza que la lista estará actualizada.
- Revisión histórica. Para esto están los sistemas de control de código fuente (ver *Control de código fuente*, 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 mueve o cambia el nombre del archivo, no querrá tener que acordarse de editar el encabezado.

Uno de los datos más importantes que *debería* aparecer en el archivo de origen es el nombre del autor, no necesariamente quién editó el archivo por última vez, sino el propietario. Adjuntar responsabilidad y rendición de cuentas al código fuente hace maravillas para mantener a las personas honestas (ver *Orgullo y prejuicio*, página 258).

9. Este tipo de información, así como el nombre del archivo, es proporcionada por el RCS \$identificación\$ etiqueta.

El proyecto también puede requerir que aparezcan ciertos avisos de derechos de autor u otro texto estándar legal en cada archivo de origen. Haga que su editor los inserte automáticamente.

Con comentarios significativos en su lugar, 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 usamos: *documentos ejecutables*.

Documentos ejecutables

Supongamos que tenemos una especificación que enumera las columnas en una tabla de base de datos. Entonces 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. Esta es una clara violación de la *SECO* principio.

Para corregir este problema, debemos elegir la fuente autorizada de información. 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 especificación como fuente. ahora es nuestro *model* para este proceso. Entonces necesitamos encontrar una manera de exportar la información que contiene como diferentes *puntos de vista*—un esquema de base de datos y un registro de lenguaje de alto nivel, por ejemplo.¹⁰

Si su documento está almacenado como texto sin formato con comandos de marcado (usando HTMLATEX, o troff, por ejemplo), luego puede usar herramientas como Perl para extraer el esquema y reformatearlo automáticamente. Si su documento está en formato binario de un procesador de texto, consulte 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 concuerdan. Minimiza la cantidad de trabajo que tiene que hacer para cada cambio y puede actualizar las vistas del cambio automáticamente.

10. Veres solo una vista, página 157, para obtener más información sobre modelos y vistas.

¿Qué sucede si mi documento no es texto sin formato?

Desafortunadamente, cada vez más documentos de proyectos se escriben utilizando procesadores de texto que almacenan el archivo en el disco en algún formato propietario. Decimos "lamentablemente" porque esto restringe severamente sus opciones para procesar el documento automáticamente. Sin embargo, todavía tienes un par de opciones:

- Escribir macros. Los procesadores de texto más 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 dolorosa, siempre puede exportar la sección adecuada a un archivo de texto sin formato de formato estándar y luego usar una herramienta como Perl para convertir esto en los formularios 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 autorizada). Luego, escriba una herramienta que exporte esta información a un formato que el documento pueda importar. Tenga cuidado, sin embargo. Debe asegurarse de que esta información se importe cada vez que se imprima 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 usando 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 siempre es trabajar en el modelo, ya sea que el modelo sea el código en sí mismo o algún otro documento, y que todas las vistas se actualicen automáticamente (ver *Automatización ubicua*, 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 sucede cuando tienes escritores técnicos profesionales involucrados en el proyecto? Con demasiada frecuencia, los programadores simplemente lanzan material "por encima de la pared" a los escritores técnicos y

déjelos valerse por sí mismos para producir manuales de usuario, piezas promocionales, etc.

Esto es un error. El hecho de que los programadores no estén escribiendo estos documentos no significa que podamos abandonar los principios pragmáticos. Queremos que los escritores adopten los mismos principios básicos que hace un programador pragmático, especialmente honrando el *SECO* principio, la ortogonalidad, el concepto de vista de modelo y el uso de automatización y secuencias de comandos.

Imprímelo o tejelo

Un problema inherente a la documentación en papel publicada es que puede quedar obsoleta tan pronto como se imprime. La documentación de cualquier forma es solo una instantánea.

Así que tratamos de producir toda la documentación en un formato que pueda publicarse en línea, en la Web, completo con hipervínculos. Es más fácil mantener actualizada esta vista de la documentación que rastrear cada copia en papel existente, grabarla y reimprimir y redistribuir nuevas copias. También es una mejor manera de abordar las necesidades de una amplia audiencia. Recuerde, sin embargo, colocar 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 necesita presentar la misma documentación en diferentes formatos: un documento impreso, páginas web, ayuda en línea o quizás una presentación de diapositivas. La solución típica se basa en gran medida en cortar y pegar, creando una serie de nuevos documentos independientes 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>

generar un nuevo capítulo en la versión de informe del documento y titular una nueva diapositiva en la presentación de diapositivas. Tecnologías como XSLyCSS11se puede usar 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 publicación en la Web.

Lenguajes de marcas

Finalmente, para proyectos de documentación a gran escala, recomendamos mirar algunos de los esquemas más modernos para marcar la documentación.

Muchos autores técnicos ahora usan 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 convertirlo en cualquier cantidad de formatos diferentes. El proyecto de documentación de Linux utiliza DocBook para publicar información en formatos RTF, TEX, info, PostScript y HTML.

Siempre que su marcado original sea lo suficientemente rico como para expresar todos los conceptos que necesita (incluidos los hipervínculos), la traducción a cualquier otro formato publicable puede ser fácil y automática. Puede producir ayuda en línea, manuales publicados, productos destacados para el sitio web e incluso un calendario de sugerencias diarias, todo desde la misma fuente, que por supuesto está bajo control de fuente y se crea junto con la compilación nocturna (consulte *Automatización ubicua*, página 230).

La documentación y el código son vistas diferentes del mismo modelo subyacente, pero la vista *estadoso* 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 la duplicación*, página 26
- *ortogonalidad*, página 34
- *El poder del texto sin formato*, página 73
- *Control de código fuente*, página 86 *es solo una vista*, página 157 *Programación por Coincidencia*, página 172 *El pozo de requisitos*,
- página 202 *Automatización ubicua*, página 230
-

Desafíos

- ¿Escribió un comentario explicativo para el código fuente que acaba de escribir? ¿Por qué no? ¿Presionado por el tiempo? No estoy seguro de si el código realmente funcionará, ¿solo está probando una idea como prototipo? Tirarás el código después, ¿verdad? No llegará al proyecto sin comentarios y experimental, ¿verdad?
- A veces es incómodo documentar el diseño del código fuente porque el diseño no está claro en tu mente; todavía está evolucionando. No sientes que debes desperdiciar esfuerzos describiendo lo que hace algo hasta que realmente lo hace. ¿Suena esto como programación por coincidencia (página 172)?

45

Grandes expectativas

Asombraos, oh cielos, de esto, y tened un miedo terrible...

► **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 empresa no cumplió con las expectativas de los analistas. Un niño abre un costoso regalo de Navidad y se echa a llorar; no era el muñeco barato que 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 paga solo facturas abstractas.

En realidad, el éxito de un proyecto se mide por qué tan bien cumple con los *Expectativas* de sus usuarios. Un proyecto que cae por debajo de sus expectativas se considera un fracaso, sin importar cuán bueno sea el entregable en términos absolutos. Sin embargo, como el padre del niño que espera la muñeca barata, vaya demasiado lejos y también será un fracaso.

TIP69

Supere suavemente las expectativas de sus usuarios

Sin embargo, la ejecución de este consejo requiere algo de trabajo.

Comunicar expectativas

Los usuarios inicialmente vienen a usted con alguna visión de lo que quieren. Puede ser incompleto, inconsistente o técnicamente imposible, pero *essuyo*, y, como el niño en Navidad, tienen algo de emoción invertida en ello. No puedes simplemente ignorarlo.

A medida que desarrolle su comprensión de sus necesidades, encontrará áreas en las que sus expectativas no se pueden cumplir, o en las que sus expectativas son quizás demasiado conservadoras. Parte de su función es comunicar esto. Trabaje con sus usuarios para que su comprensión de lo que entregará sea precisa. Y hacer esto durante todo el proceso de desarrollo. Nunca pierda de vista los problemas comerciales que su aplicación pretende resolver.

Algunos consultores denominan a este proceso "gestión de expectativas": controlar activamente lo que los usuarios deberían 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 su lugar, 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; todos deben entender lo que se espera y cómo se construirá.

Hay algunas técnicas importantes que se pueden utilizar para facilitar este proceso. De estos, *Balas trazadoras*, página 48, y *Prototipos y 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 ellos.

la milla extra

Si trabaja en estrecha colaboración con sus usuarios, compartiendo sus expectativas y comunicando lo que está haciendo, habrá pocas sorpresas cuando se entregue el proyecto.

Esto es uncosa MALA. Intenta sorprender a tus usuarios. No asustarlos, eso sí, pero *deleitar*los.

Dales ese poco más de lo que esperaban. El poco esfuerzo adicional que requiere agregar alguna función orientada al usuario al sistema se pagará por sí mismo una y otra vez con buena voluntad.

Escuche a sus usuarios a medida que avanza el proyecto para obtener pistas sobre qué características realmente les encantarán. Algunas cosas que puede agregar con relativa facilidad que se ven bien para el usuario promedio incluyen:

- Atajos de teclado de ayuda con globo o
- información sobre herramientas
- Una guía de referencia rápida como complemento del manual del usuario
- Colorización
- Analizadores de archivos de registro
- Instalación automatizada
- Herramientas para comprobar la integridad del sistema.
- La capacidad de ejecutar múltiples versiones del sistema para capacitación Una
- pantalla de inicio personalizada para su organización

Todas estas cosas son relativamente superficiales y en realidad no sobrecargan el sistema con una gran cantidad de funciones. Sin embargo, cada uno les dice a sus usuarios que el equipo de desarrollo se preocupó por producir un gran sistema, uno diseñado para un uso real. Solo recuerda no romper el sistema agregando estas nuevas características.

Las secciones relacionadas incluyen:

- *Software suficientemente bueno*, página 9
- *Balas trazadoras*, página 48 *Prototípos y*
- *Post-it Notes*, página 53 *El pozo de requisitos*
- , página 202

Desafíos

- A veces, los críticos más duros de un proyecto son las personas que trabajaron en él. ¿Alguna vez te has sentido decepcionado porque tus propias expectativas no se cumplieron con algo que produjiste? ¿Cómo es posible? Tal vez haya algo más que lógica en el trabajo aquí.
- ¿Qué comentan sus usuarios cuando entrega software? ¿Su atención a las diversas áreas de la aplicación es proporcional al esfuerzo que invirtió en cada una? ¿Qué les encanta?

46

Orgullo y prejuicio

Nos has deleitado lo suficiente.

► Jane Austen, *Orgullo y prejuicio*

Los programadores pragmáticos no eluden la responsabilidad. En cambio, nos regocijamos en aceptar desafíos y en dar a conocer nuestra experiencia. Si somos responsables de un diseño o una pieza de código, hacemos un trabajo del que podemos estar orgullosos.

TIP70

Firme su trabajo

Los artesanos de una época anterior estaban orgullosos de firmar su trabajo. Tú también deberías estarlo.

Sin embargo, los equipos de proyecto todavía están formados por personas y esta regla puede causar problemas. En algunos proyectos, la idea de *propiedad del código* puede causar problemas de cooperación. Las personas pueden volverse territoriales o no estar dispuestas a trabajar sobre 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 deberías defender celosamente tu código contra los intrusos; del mismo modo, debe 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 código incorrecto. Se vuelve demasiado fácil verse a sí mismo como un engranaje en la rueda, produciendo excusas tontas en informes de estado interminables en lugar de un buen código.

Si bien el código debe ser propiedad, no es necesario que sea propiedad de un individuo. De hecho, el exitoso método de programación eXtreme de Kent Beck [URL 45] recomienda la propiedad comunitaria del código (pero esto también requiere prácticas adicionales, como la programación en pareja, para protegerse contra los peligros del anonimato).

Queremos ver orgullo de propiedad. "Escribí esto y estoy detrás de 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, esté bien escrito, probado y documentado. Un trabajo realmente profesional. Escrito por un verdadero profesional.

Un programador pragmático.

Esta página se dejó en blanco intencionalmente

Apéndice A

Recursos

La única razón por la que pudimos cubrir tanto terreno en este libro es que vimos muchos de nuestros sujetos desde una gran altura. 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 ayudarlo con este proceso.

En la sección *Sociedades profesionales*, damos detalles del IEEE y el ACM. Recomendamos que los programadores pragmáticos se unan a una (o ambas) de estas sociedades. Entonces, en *construyendo una biblioteca*, destacamos publicaciones periódicas, libros y sitios web que consideramos que contienen información pertinente y de alta calidad (o que simplemente son divertidos).

A lo largo del libro, mencionamos muchos recursos de software accesibles a través de Internet. En el *Recursos de Internet* sección, 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 cuando 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 clase mundial para programadores: la Association for Computing Machinery (ACM)¹ y la Sociedad de Computación IEEE.² Recomendamos que todos los programadores pertenezcan a una (o ambas) de estas sociedades. Además, los desarrolladores fuera de los Estados Unidos pueden querer unirse a sus sociedades nacionales, como la BCS en el Reino Unido.

La membresía en una sociedad profesional tiene muchos beneficios. Las conferencias y reuniones locales le brindan grandes oportunidades para conocer personas con intereses similares, y los grupos de interés especial y los comités técnicos le brindan la oportunidad de participar en el establecimiento de estándares y pautas que se utilizan en todo el mundo. También sacará mucho provecho de sus publicaciones, desde debates de alto nivel sobre la práctica de la industria hasta teoría informática de bajo nivel.

construyendo una biblioteca

Somos grandes en la lectura. Como señalamos en *Su carpeta de conocimientos*, página 12, un buen programador siempre está aprendiendo. Mantenerse al día con libros y publicaciones periódicas puede ayudar. Aquí hay algunos que nos gustan.

Periódicos

Si es como nosotros, guardará revistas y periódicos viejos hasta que se apilen lo suficiente como para convertir los de abajo en láminas planas de diamante. Esto significa que vale la pena ser bastante selectivo. Aquí hay algunos periódicos que leemos.

- Computadora IEEE. Enviado a los miembros de la IEEE Computer Society, *Computadora* tiene un enfoque práctico pero no le teme a la teoría. Algunos números están orientados en torno a un tema, mientras que otros son simplemente

1. Servicios para miembros de ACM, PO Box 11414, Nueva York, NY 10286, EE. UU.

⇒ www.acm.org

2. 1730 Massachusetts Avenue NW, Washington, DC 20036-1992, EE. UU.

⇒ www.computadora.org

colección de artículos interesantes. Esta revista 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, *MCC4* 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 en 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 aquellos a los que les gusta profundizar en la programación.
- Diario del Dr. Dobbs. Una revista mensual, disponible por suscripción y en quioscos, *Dra. Dobbs* es peculiar, pero tiene artículos que van desde la práctica de nivel de bits hasta la teoría pesada.
- El Diario Perl. Si te gusta Perl, probablemente deberías suscribirte a *E/diario de Perl* (www.tpj.com).
- Revista de desarrollo de software. Revista mensual centrada en temas generales de gestión de proyectos y desarrollo de software.

Documentos comerciales semanales

Hay varios periódicos semanales publicados para desarrolladores y sus gerentes. Estos documentos son en gran parte una colección de comunicados de prensa de la empresa, presentados como artículos. Sin embargo, el contenido sigue siendo valioso: le permite realizar un seguimiento de lo que sucede, 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 espere mucha cobertura técnica detallada.

Libros

Los libros de computación pueden ser costosos, pero elija con cuidado y son una inversión que vale la pena. Es posible que desee consultar nuestros títulos de Pragmatic Bookshelf en <http://pragmaticprogrammer.com>. Además, aquí hay algunos de los muchos otros libros que nos gustan.

Análisis y Diseño

- *Construcción de software orientada a objetos, 2^a edición.* El libro épico de Bertrand Meyer sobre los fundamentos del desarrollo orientado a objetos, todo en unas 1300 páginas [Mey97b].
- *Patrones de diseño.* Un patrón de diseño describe una forma de resolver una clase particular de problemas en un nivel más alto que un lenguaje de programación. Este libro ahora clásico [GHJV95] del *Pandilla de cuatro* describe 23 patrones de diseño básicos, incluidos Proxy, Visitor y Singleton.
- *Patrones de análisis.* Un tesoro oculto 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 información de muchos años de experiencia en modelado [Fow96].

Equipos y Proyectos

- *El mes del hombre mítico.* El clásico de Fred Brooks sobre los peligros de organizar equipos de proyectos, actualizado recientemente [Bro95].
- *Dinámica del Desarrollo de Software.* Una serie de ensayos breves sobre la creación de software en grandes equipos, centrándose en la dinámica entre los miembros del equipo y entre el equipo y el resto del mundo [McC95].
- *Proyectos orientados a objetos sobrevivientes: una guía para gerentes.* Los “informes desde las trincheras” de Alistair Cockburn ilustran muchos de los peligros y escollos de administrar un proyecto OO, especialmente el primero. El Sr. Cockburn brinda consejos y técnicas para ayudarlo a superar los problemas más comunes [Coc97b].

Entornos específicos

- Unix.W. Richard Stevens tiene varios libros excelentes que incluyen *Programación Avanzada en el Entorno Unix* y *el Programación de red Unix* libros [Ste92, Ste98, Ste99].
- Ventanas.Cerebro de marshall *Servicios del sistema Win32*[Bra95] es una referencia concisa a las API de bajo nivel. de charles petzold *programación de ventanas*[Pet98] es la biblia del desarrollo de GUI de Windows.
- C++. Tan pronto como se encuentre en un proyecto de C++, corra, no camine, vaya a la librería y compre el libro de Scott Meyer. *C++ efectivo*, y posiblemente *C++ más efectivo*[Mey97a, Mey96]. Para construir sistemas de cualquier tamaño apreciable, necesita John Lakos' *Diseño de software C++ a gran escala*[Lak96]. Para técnicas avanzadas, recurra a Jim Coplien. *Modismos y estilos de programación avanzados en C++*[Cop92].

Además, el O'Reilly *Cáscara de nuezserie* (www.ora.com) brinda tratamientos rápidos y completos de diversos temas e idiomas como perl, yacc, sendmail, Elementos internos de Windows 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.

- barra oblicua.Anunciado como "Noticias para nerds. Cosas que importan", Slashdot es uno de los hogares de red de la comunidad Linux. Además de actualizaciones periódicas sobre noticias de Linux, el sitio ofrece información sobre tecnologías interesantes y problemas que afectan a los desarrolladores.
⇒ www.slashdot.org
- Enlaces 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 sólo un gran recurso, el sitio WikiWikiWeb es un experimento interesante en la edición colectiva de ideas.
⇒ www.c2.com

Recursos de Internet

Los enlaces a continuación son para recursos disponibles en Internet. Eran válidos en el momento de escribir este artículo, pero (siendo la Red lo que es) es posible que estén desactualizados para cuando lea esto. Si es así, puede intentar una búsqueda general de los nombres de archivo, o ir a Pragmatic Programmer Web sitio (www.pragmaticprogrammer.com)y sigue nuestros enlaces.

Editores

Emacs y vi no son los únicos editores multiplataforma, pero están disponibles gratuitamente y se usan ampliamente. Un vistazo rápido a través de una revista como *Dra. Dobbs* aparecerán varias alternativas comerciales.

Emacs

Tanto Emacs como XEmacs están disponibles en plataformas Unix y Windows.

[URL 1]El editor de Emacs

⇒ www.gnu.org

Emacs, lo último en grandes editores, que contiene todas las características que cualquier editor haya tenido, tiene una curva de aprendizaje casi vertical, pero paga generosamente una vez que lo dominas. 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

Creado a partir del Emacs original hace algunos años, se dice que XEmacs tiene un interior más limpio y una interfaz más atractiva.

vi

Hay al menos 15 clones vi diferentes disponibles. De estos,empujeprobablemente esté adaptado a la mayoría de las plataformas, por lo que sería una buena elección de editor si se encuentra trabajando en muchos entornos diferentes.

[URL 3]El Editor Vim

⇒ <ftp://ftp.fu-berlin.de/misc/editors/vim>

De la documentación: "Hay muchas mejoras por encima de vi: deshacer de múltiples niveles, múltiples ventanas y búferes, resaltado de sintaxis, edición de línea de comando, finalización de nombre de archivo, ayuda en línea, selección visual, etc. " . . .

[URL 4]el editor de elvis

⇒ elvis.la-pequena-pelirroja.org

Un clon vi mejorado con soporte para X.

[URL 5]Modo Viper de Emacs

⇒ <http://www.cs.sunysb.edu/~kifer/emacs.html>

Viper es un conjunto de macros que hacen que Emacs parezca vi. Algunos pueden dudar de la sabiduría de tomar el editor más grande del mundo y extenderlo para emular un editor cuyo punto fuerte es su tamaño compacto. 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 separó de gcc, está en proceso de fusionarse nuevamente con el redil).

[URL 7]El lenguaje Java de Sun

⇒ java.sun.com

Hogar de Java, que incluye SDK descargables, documentación, tutoriales, noticias y más.

[URL 8]Página de inicio del idioma Perl

⇒ www.perl.com

O'Reilly alberga 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 un poco peculiar y un seguimiento amplio y leal.

[URL 10]PequeñoEiffel

⇒ PequeñoEiffel.loria.fr

El compilador GNU Eiffel se ejecuta en cualquier máquina que tenga un compilador ANSI C y un entorno de tiempo 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 admitir funciones de orden superior y 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

Hogar del entorno VisualWorks Smalltalk. Las versiones no comerciales para Windows y Linux están disponibles de forma gratuita.

[URL 14]El entorno del lenguaje Squeak

⇒ squeak.cs.uiuc.edu

Squeak es una implementación portátil y disponible gratuitamente 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.jerbo.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 clústeres en red de cajas Linux económicas.

[URL 17]iContract: herramienta de diseño por contrato para Java

⇒ www.reliable-systems.com

Diseño por formalismo de contrato de precondiciones, postcondiciones e invariantes, implementado como un preprocesador para Java. Respeta la herencia, implementa cuantificadores existenciales y más.

[URL 18]Nana: registro y aserciones para C y C++

⇒ www.gnu.org/software/nana/nana.html

Compatibilidad mejorada para la comprobación de aserciones y el inicio de sesión en C y C++. También proporciona algo de soporte para el diseño por contrato.

[URL 19]DDD: depurador de visualización de datos

⇒ <http://www.gnu.org/software/ddd/>

Una interfaz gráfica gratuita 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/doc++/index.html

DOC++ es un sistema de documentación para C/C++ y Java que genera tanto La Salida TEX y HTML para una exploración en línea sofisticada de su documentación directamente desde el encabezado C++ o archivos de clase Java.

[URL 22]xUnit—Marco de pruebas unitarias

⇒ www.XProgramming.com

Un concepto simple pero poderoso, el marco de prueba de unidad xUnit proporciona una plataforma consistente para probar software escrito en una variedad de lenguajes.

[URL 23]El lenguaje Tcl

⇒ www.scriptics.com

Tcl ("Tool Command Language") es un lenguaje de secuencias de comandos diseñado para ser fácil de integrar en una aplicación.

[URL 24]Expect—Automatiza la interacción con los programas

⇒ esperar.nist.gov

Una extensión basada en Tcl [URL 23],suponerle permite programar la interacción con los programas. Además de ayudarlo a escribir archivos de comando que (por ejemplo) obtienen archivos de servidores remotos o amplían el poder de su shell, suponerpuede ser útil al realizar pruebas de regresión. Una versión gráfica,esperando,le permite empaquetar aplicaciones que no son GUI con un front-end de ventana.

[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 computadoras y sistemas operativos heterogéneos. T Spaces proporciona servicios de comunicación grupal, servicios de base 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 analizador de bisontes

⇒ www.gnu.org/software/bison/bison.html

bisontetoma una especificación de gramática de entrada y genera a partir de ella el código fuente C de un analizador adecuado.

[URL 28]SWIG: contenedor simplificado y generador de interfaz

⇒ 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 Common Object Request Broker Architecture (CORBA) y el Internet Inter-ORB Protocol (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 de Unix bajo DOS

[URL 30]Las herramientas de desarrollo de UWIN

⇒ www.gtinc.com/uwin.html

Global Technologies, Inc., Old Bridge, Nueva Jersey

El paquete UWIN proporciona bibliotecas de enlace dinámico (DLL) de Windows que emulan una gran parte de la interfaz de biblioteca de nivel C de Unix. Usando esta interfaz, GTL ha portado una gran cantidad de herramientas de línea de comandos de Unix a Windows.

Véase también [URL 31].

[URL 31]Las herramientas de Cygnus Cygwin

⇒ [sourceware.cygnus.com/cygwin/](http://sourcware.cygnus.com/cygwin/)

Soluciones Cygnus, Sunnyvale, CA

El paquete Cygnus también emula la interfaz de la biblioteca Unix C y proporciona una gran variedad de herramientas de línea de comandos de Unix en el sistema operativo Windows.

[URL 32]Herramientas eléctricas Perl

⇒ www.perl.com/pub/language/ppt/

Un proyecto para volver a implementar el clásico conjunto de comandos de Unix en Perl, haciendo que los comandos estén disponibles en todas las plataformas que admiten Perl (y eso es un montón de plataformas).

Herramientas de control de código fuente

[URL 33]RCS—Sistema de Control de Revisión

⇒ 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 disponible gratuitamente para Unix y Windows NT. Extiende 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 revisión orientada a procesos que impone estándares de proyecto (como verificar que el código registrado pasa las pruebas).

[URL 36]ClearCase

⇒ www.racional.com

Control de versiones, espacio de trabajo y gestión de compilación, control de procesos.

[URL 37]Integridad de la fuente MKS

⇒ www.mks.com

Control de versiones y gestión de la configuración. Algunas versiones incorporan funciones que permiten a los desarrolladores remotos trabajar en los mismos archivos simultáneamente (muy parecido a CVS).

[URL 38]Gestión de configuración de PVCS

⇒ www.merant.com

Un sistema de control de código fuente, muy popular para los sistemas Windows.

[URL 39]Visual SourceSafe

⇒ www.microsoft.com

Un sistema de control de versiones que se integra con las herramientas de desarrollo visual de Microsoft.

[URL 40]Forzosamente

⇒ 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 Computing, Inc., Mansfield, CT

Una utilidad de archivado de archivos basada en Windows. Soporta ambosCódigo Postal y alquitránformatos.

[URL 42]La concha Z

⇒ sunsite.auc.dk/zsh

Un shell diseñado para uso interactivo, aunque también es un potente lenguaje de scripting. Muchas de las funciones 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. Samba incluye:

- Un servidor SMB, para proporcionar servicios de archivos e impresión al estilo de 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, brinda soporte para la navegación. Samba puede ser el navegador maestro en su LAN si lo desea.
- Un cliente SMB tipo ftp que le permite acceder a recursos de PC (discos e impresoras) desde Unix, Netware y otros sistemas operativos.

Papeles y Publicaciones

[URL 44]Preguntas frecuentes sobre comp.object

⇒ www.cyberdyne-object-sys.com/oofaq2

Preguntas frecuentes sustanciales y bien organizadas para elcomp.objectgrupo de noticias.

[URL 45]Programación extrema

⇒ www.XProgramming.com

Del sitio web: "En XP, usamos una combinación muy liviana de prácticas para crear un equipo que pueda producir rápidamente software extremadamente confiable, eficiente y bien diseñado. Muchas de las prácticas de XP se crearon y probaron como parte del proyecto Chrysler C3, que es un sistema de nómina muy exitoso implementado en Smalltalk".

[URL 46]Página de inicio de Alistair Cockburn

⇒ miembros.aol.com/acockburn

Busque "Estructuración de casos de uso con objetivos" y plantillas de casos de uso.

[URL 47]Página de inicio de Martin Fowler

⇒ ourworld.compuserve.com/homepages/martin_fowler

Autor de *Patrones de análisis* y coautor de *UML destilado* y *Refactorización: mejora del diseño del código existente*. La página de inicio de Martin Fowler analiza sus libros y su trabajo con UML.

[URL 48]Página de inicio de Robert C. Martin

⇒ www.objectmentor.com

Buenos artículos introductorios sobre técnicas orientadas a objetos, incluidos análisis de dependencia y 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 soporta 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 la jerga

⇒ www.jargon.org

eric s raymond

Definiciones de muchos términos comunes (y no tan comunes) de la industria informática, junto con una buena dosis de folclore.

[URL 53]Documentos de Eric S. Raymond

⇒ www.tuxedo.org/~esr

Los papeles de Eric en *La catedral y el bazary* *Ocupando la Noosfera* describiendo la base psicosocial y las implicaciones del movimiento Open Source.

[URL 54]El entorno de escritorio K

⇒ www.kde.org

De su página web: "KDE es un poderoso 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.

Misceláneas

[URL 57] El Proyecto GNU

⇒ www.gnu.org

Fundación de 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 similar a Unix, libre y completo.

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 están disponibles gratuitamente.

Bibliografía

- [Bak72] FT Baker. Jefe de equipo de programadores dirección de programación de producción. *Diario de sistemas de IBM*, 11(1):56–73, 1972.
- [BBM96] V. Basili, L. Briand y WL Melo. Una validación de métricas de diseño orientado a objetos como indicadores de calidad. *Transacciones IEEE sobre ingeniería de software*, 22(10):751–761, octubre de 1996.
- [Ber96] Albert J. Bernstein. *Cerebros de dinosaurio: lidar con todas esas personas imposibles en el trabajo*. Ballantine Books, Nueva York, NY, 1996.

- [Sujetador95] Cerebro marshall. *Servicios del sistema Win32*. Prentice Hall, Englewood Cliffs, Nueva Jersey, 1995.
- [Hermano95] 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.
- [Cla04] mike clark *Automatización pragmática de proyectos*. The Pragmatic Programmers, LLC, Raleigh, NC y Dallas, TX, 2004.
- [CN91] Brad J. Cox y Andrex J. Novobilski. *Orientado a objetos Programación, un enfoque evolutivo*. Lectura, Addison Wesley, MA, 1991.
- [Coc97a] Alistair Cockburn. Objetivos y casos de uso. *Diario de Programación Orientada a Objetos*, 9(7):35–40, septiembre de 1997.
- [Coc97b] Alistair Cockburn. *Sobrevivir a proyectos orientados a objetos: una guía para gerentes*. Addison Wesley Longman, Lectura, MA, 1997.
- [Cop92] James O. Coplien. *Modismos y estilos de programación avanzados en 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, Lectura, MA, 1999.
- [Seguir96] Martín Fowler. *Patrones de análisis: modelos de objetos reutilizables*. Addison Wesley Longman, Lectura, MA, 1996.
- [FS99] Martin Fowler y Kendall Scott. *UML destilado: 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 del software orientado a objetos reutilizable*. Addison-Wesley, Reading, MA, 1995.

- [Gla99a] Robert L. Vidrio. Inspecciones—Algunos hallazgos sorprendentes. *Comunicaciones de la ACM*, 42(4):17–19, abril de 1999.
- [Gla99b] Robert L. Vidrio. Las realidades de los beneficios de la tecnología de software. *Comunicaciones de la ACM*, 42(2):74–79, febrero de 1999.
- [Hol78] Michael Holt. *Rompecabezas y juegos matemáticos*. Dorset Press, Nueva York, NY, 1978.
- [HT03] Andy Hunt y Dave Thomas. *Pruebas unitarias pragmáticas en Java con JUnit*. The Pragmatic Programmers, LLC, Raleigh, NC 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 *Conferencia Europea sobre Programación Orientada a Objetos (ECOOP)*, volumen LNCS 1241. Springer-Verlag, junio de 1997.
- [Knu97a] Donald Ervin Knuth. *El arte de la programación informática: algoritmos fundamentales*, volumen 1. Addison Wesley Longman, Reading, MA, tercera edición, 1997.
- [Knu97b] Donald Ervin Knuth. *El arte de la programación informática: algoritmos semínuméricos*, volumen 2. Addison Wesley Longman, Reading, MA, tercera edición, 1997.
- [Knu98] Donald Ervin Knuth. *El arte de la programación informática: 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, Lectura, MA, 1999.
- [Kru98] Felipe Kruchten. *El proceso unificado racional: una introducción*. Addison Wesley Longman, Lectura, MA, 1998.
- [Lak96] Juan Lacos. *Diseño de software C++ a gran escala*. Addison Wesley Longman, Lectura, MA, 1996.

- [LH89] Karl J. Lieberherr e Ian Holland. Garantizar un buen estilo para los programas orientados a objetos. *Software IEEE*, 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, Washington, 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++ eficaz: 50 formas específicas de mejorar sus programas y diseños*. Addison Wesley Longman, Reading, MA, segunda edición, 1997.
- [Mey97b] Bertrand Meyer. *Construcción de software orientada a objetos*. Prentice Hall, Englewood Cliffs, NJ, segunda edición, 1997.
- [Mascota98] Carlos Petzold. *Programando Windows, El Definitivo Guía de la API de Win32*. Microsoft Press, Redmond, WA, quinta edición, 1998.
- [Sch95] Bruce Schneier. *Criptografía aplicada: protocolos, algoritmos ritmos y código fuente en C*. John Wiley & Sons, Nueva York, NY, segunda edición, 1995.
- [Sed83] Roberto Sedgewick. *Algoritmos*. Addison-Wesley, Reading, MA, 1983.
- [Sed92] Roberto Sedgewick. *Algoritmos en C++*. Addison-Wesley, Reading, MA, 1992.
- [SF96] Robert Sedgewick y Philippe Flajolet. *Una 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.
- [St99] W.Richard Stevens. *Programación de redes Unix, Volumen 2: Comunicaciones entre procesos*. Prentice Hall, Englewood Cliffs, NJ, segunda edición, 1999.
- [Str35] James Ridley Stroop. Los estudios de interferencia en las reacciones verbales de serie. *Revista de Psicología Experimental*, 18:643–662, 1935.
- [TFH04] Dave Thomas, Chad Fowler y Andy Hunt. *Programación de Ruby, la guía de los programadores pragmáticos*. The Pragmatic Programmers, LLC, Raleigh, NC y Dallas, TX, 2004.
- [TH03] Dave Thomas y Andy Hunt. *Control de versiones pragmático usando CVS*. The Pragmatic Programmers, LLC, Raleigh, NC y Dallas, TX, 2003.
- [WK82] James Q. Wilson y George Kelling. La policía y la seguridad vecinal. *El Atlántico Mensual*, 249(3):29–38, marzo de 1982.
- [YC86] Edward Yourdon y Larry L.Constantine. *Diseño estructurado: fundamentos de una disciplina de diseño de sistemas y programas informáticos*. Prentice Hall, Englewood Cliffs, NJ, segunda edición, 1986.
- [Tú95] Eduardo Yourdon. Cuando un software lo suficientemente bueno es lo mejor. *Software IEEE*, mayo de 1995.

apéndice B

Respuestas a ejercicios

Ejercicio 1: de Ortogonalidad en la página 43

Estás escribiendo una clase llamada `Separar`, que divide las líneas de entrada en campos. ¿Cuál de las siguientes dos firmas de clase Java es el diseño más ortogonal?

```
claseDividir1 {  
    públicoSplit1(InputStreamReader rdr) { ... vacío público leerLíneaSiguiente() lanza  
    IOExcepción { ... public int numFields() { ...  
  
    públicoCadena getField(En tcampoNo) { ...  
}  
  
claseDividir2 {  
    públicoDividir2 (línea de cadena) { ... public int  
    numFields() { ...  
    públicoCadena getField(En tcampoNo) { ...  
}
```

Respuesta 1: A nuestra forma de pensar, `claseDividir1` es más ortogonal. se centra 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. `Dividir2` 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 conducirá a un diseño más ortogonal: cuadros de diálogo modales o no modales?

Respuesta 2: Si se hace correctamente, probablemente sin modelo. Un sistema que utiliza mode- menos cuadros de diálogo estarán menos preocupados por lo que sucede en un momento determinado. Es probable que tenga una mejor infraestructura de comunicaciones entre módulos que un sistema modal, que puede tener suposiciones integradas sobre el estado del sistema, suposiciones que conducen a un mayor acoplamiento y una menor ortogonalidad.

Ejercicio 3:de *Ortogonalidad en la página 43*

¿Qué hay de los lenguajes procedimentales frente a la tecnología de objetos? ¿Cuál da como resultado un sistema más ortogonal?

Respuesta 3:Esto es un poco complicado. Tecnología de objetos *pueden* proporcionar un sistema más ortogonal, pero debido a que tiene más funciones para abusar, en realidad es más fácil crear uno *ortogonal* sistema usando objetos que usando un lenguaje procedural. Características como la herencia múltiple, las excepciones, la sobrecarga de operadores y la anulación del método principal (a través de subclases) brindan una amplia oportunidad para aumentar el acoplamiento de formas no obvias.

Con tecnología de objetos y un poco de esfuerzo adicional, puede lograr un sistema mucho más ortogonal. Pero si bien siempre puede escribir "código de espagueti" en un lenguaje de procedimiento, los lenguajes orientados a objetos mal utilizados pueden agregar albóndigas a su espagueti.

Ejercicio 4:de *Prototipos y Post-it Notes en la página 56*

A 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 decidir un modelo para la imagen, tal vez sea un automóvil, un teléfono o una casa. Tiene una lista de páginas de destino y contenido; les gustaría ver algunos prototipos. Oh, por cierto, tienes 15 minutos. ¿Qué herramientas podría usar?

Respuesta 4: ¡Baja tecnología al rescate! Dibuja algunas caricaturas con marcadores en un pizarrón: un automóvil, un teléfono y una casa. No tiene que ser un gran arte; los contornos de stickfigure están bien. Coloque notas Post-it 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 perfeccionar los dibujos y las ubicaciones de las notas Post-it.

Ejercicio 5:de *Idiomas de dominio en la página 63*

Queremos implementar un mini-lenguaje para controlar un paquete de dibujo simple (quizás un sistema de gráficos de tortugas). El lenguaje consta de comandos de una sola letra. Algunos comandos van seguidos de un solo número. Por ejemplo, la siguiente entrada dibujaría un rectángulo.

```

PAQS # Seleccione bolígrafo 2
D   # lápiz abajo
W 2 # dibujar oeste 2cm
norte 1 # después norte 1
mi 2 # después este 2
S 1 # después espalda sur
tu   # lápiz arriba

```

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

Respuesta 5: Como queremos que el lenguaje sea extensible, haremos que la tabla del analizador sea controlada. Cada entrada en la tabla contiene la letra del comando, una bandera para indicar si se requiere un argumento y el nombre de la rutina a llamar para manejar ese comando en particular.

```
definición de tipo estructura{
    carbonizarse cmd;                                /* la letra de comando */ /* necesita un
    En ttieneArg;                                    argumento */
    vacío(*función)(En t,En t);/*rutina para llamar */Dominio;
}

estáticoComando cmds[] = {
    { 'PAGS', ARG, doSelectPen      },
    { 'tú',   NO_ARG, doPenUp       },
    { 'D',    NO_ARG, doPenDown     },
    { 'NORTE', ARG, doPenDir      },
    { 'MI',   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 del controlador.

```
tiempo(fgets(aficionado,tamaño de(mejora), entrada estándar)) {
    Comando *cmd = findCommand(*buff);
    si(cmd) {
        En t     argumento = 0;
        si(cmd->hasArg && !getArg(buff+1, &arg)) {
            fprintf(stderr, "%c' necesita un argumento n", *fuerte); Seguir;
        }
        cmd->func(*buff, argumento);
    }
}
```

La función que busca un comando realiza una búsqueda lineal en la tabla y devuelve la entrada coincidente ONULO.

```
Comando *buscarComando(En tcmd) {
    En ti;
    por(yo = 0; i < TAMAÑO_ARRAY(cmds); i++) {
        si(cmds[i].cmd == cmd)
            devolvercmds + i;
    }
    fprintf(stderr, "Comando desconocido %c n", cmd); devolver0;
}
```

Finalmente, leer el argumento numérico es bastante simple usandoescanear

```
En tobtenerArg(carácter constante*fuerte,En t*resultado) {
    devolverescanear (mejorar, "%d", resultado) == 1;
}
```

Ejercicio 6: de Idiomas de dominio en la página 63

Diseñe una gramática BNF para analizar una especificación de tiempo. Todos los siguientes ejemplos deben ser aceptados.

4pm, 7:38pm, 23:42, 3:16, 3:16am

Respuesta 6: Usando BNF, una especificación de tiempo podría ser

```

<tiempo> ::= <hora> <am PM> |
              <hora> : <minuto> <am PM> |
              <hora> : <minuto>

<am PM> ::= am | PM

<hora> ::= <dígito> |
              <dígito> <dígito>

<minuto> ::= <dígito> <dígito>

<dígito> ::= 0 | 2|3 | 5|6 | 8|9 | | |

```

Ejercicio 7: de Idiomas de dominio en la página 63

Implemente un analizador para la gramática BNF en el Ejercicio 6 usando 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. Mire la fuente en nuestro sitio web para ver la implementación completa.

```

tiempo:      especificaciónEND_TOKEN
{ $ips1 >= 24*60 } yyerror("El tiempo es demasiado grande");
              imprimirf("%d minutos pasada la medianoche n", $1); salir(0);

}

;

Especificaciones:
hora ':> minuto
{ $$ = $1 + $3;
}

| hora ':> minuto amperio
{ $ips1 > 11*60 } yyerror("Hora fuera de rango");
      $$ = $1 + $3 + $4;
}

| hora amperios
{ $ips1 > 11*60 } yyerror("Hora fuera de rango");
      $$ = $1 + $2;
}

;

hora:      numero_hora
{ $ips1 > 23 } yyerror("Hora fuera de rango");
      $$ = $1 * 60;
}

```

```

minuto: DÍGITO DÍGITO
        { $$ = $1*10 + $2;
          si($$>59) yyerror( "minuto fuera de rango" );
        };

am PM:   SOY
        { $$ = AM_MINS; } { $$ =
        | PM
        { $$ = PM_MINS; }

núm hora: DÍGITO
        { $$ = $1; }
        | DÍGITO DÍGITO
        { $$ = $1*10 + $2; }

```

Ejercicio 8: de Idiomas de dominio en la página 63

Implemente el analizador de tiempo usando Perl. [*Insinuación:Las expresiones regulares son buenos analizadores.*]

Respuesta 8:

Ejercicio 9: de Estimación en la página 69

Se le pregunta "¿Qué tiene un mayor ancho de banda: 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 en su respuesta para asegurarse de que el alcance de su respuesta sea correcto? (Por ejemplo, podría decir que se ignora el tiempo necesario para acceder a la cinta).

Respuesta 9: Nuestra respuesta debe basarse 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 lleva 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 bits, por lo que 32×10^9 una línea de 1 Mbps tendría que bombar datos durante aproximadamente 32,000 segundos, o aproximadamente horas, para transferir la cantidad equivalente de información. Si la persona camina a una mph constante, entonces nuestras dos máquinas deberían estar al menos a millas de distancia para que la línea de comunicaciones supere a nuestro servicio de mensajería. De lo contrario, la persona gana.

Ejercicio 11:de *Manipulación de texto en la página 102*

Su 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) para fines de depuración. Escriba un script que lea desde la entrada estándar un archivo que contenga

```
nombre
estado_a
estado_b
:
:
```

Producir el archivo *nombre.h*, que contiene

```
carácter const externo* NOMBRE_nombres[];
enumeración typedef{
    estado_a,
    estado_b,
    :
    :
} NOMBRE;
```

y el archivo *nombre.c*, que contiene

```
carácter constante* NOMBRE_nombres[] = {
    "estado_a",
    "estado_b",
    :
    :
};
```

Respuesta 11: Implementamos nuestra respuesta usando Perl.

```

mi@constantes;
mipsnombre = <>;
morir "Formato inválido - nombre faltante" a menos que se defina($nombre);
morderpsnombre;
# Leer el resto del archivo tiempo(<>){

    morder;
    s/^$^V\;/s/spss      \
    morir "Línea no válida: $_" a no ser que/^($+)$/;      \
    empujar@constantes, $_;
}

# Ahora genera el archivo
abierto(HDR,>"$nombre.h") o morir ";No puedo abrir $nombre.h: $!";
abierto(SRC,
">$nombre.c") o morir ";No puedo abrir $nombre.c: $!";
mipsnombre_uc = uc($nombre);
mipsnombre_matriz = $nombre_uc . "_nombres";
impresiónHDR /*Archivo generado automáticamente - no editar */n;
impresiónHDR "externo
const char *$ {array_name}[];"; impresiónHDR "typedef enumeración { n ";
impresiónHDRunirse
",n ",@consts; impresiónHDR"n} $uc_nombre; nn";
                                \
                                \
impresiónSRC /*Archivo generado automáticamente - no editar */n;
impresiónSRC "const char
*$ {array_name}[] = { n impresiónSRCunirse " ", norte impresiónSRC" "norte";";
                                \
                                \
cerca(CRS);
cerca(HDR);

```

Utilizando el *SECO* principio, no cortaremos y pegaremos este nuevo archivo en nuestro código. En su lugar, vamos a #incluirlo—el archivo plano es la fuente maestra de estas constantes. Esto significa que necesitaremos un archivo MAKE para regenerar el encabezado cuando cambie el archivo. El siguiente extracto es del banco de pruebas de 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 escribir este libro, nos dimos cuenta de que no habíamos puesto el uso estricto directiva en muchos de nuestros ejemplos de Perl. Escriba un script que pase por el .por favorarchivos en un directorio y agrega un uso estricto al final del bloque de comentarios inicial a todos los archivos que aún no tienen uno. Recuerde mantener una copia de seguridad de todos los archivos que cambie.

Respuesta 12: Aquí está nuestra respuesta, escrita en Perl.

```
mipsdirección =cambiar o morir "Falta directorio"; para mips
expediente (globo( "$dir/*.pl" )) {
    abierto(PI,psexpediente)"o morir "Abriendo $archivo: $!"; indefinido
    ps                                # Desactivar el separador de registros de entrada --
    mipscontenido = <IP>;#leer todo el archivo como una cadena. cerca(PI);

    sipscontenido !~ /^usarestricto/metro) {
        rebautizarpsexpediente,psarchivo.bak"o morir "Renombrando $archivo: $!";
        abierto(OP,>"$expediente")o morir "Creando $archivo: $!";
        # Ponga 'uso estricto' en la primera línea que
        # no comienza '#'
        $contenido =~s/^#!/nuse estricto; nn/m;           \
        impresiónOP $contenido;
        cerca(OP);
        impresión "Actualizado $archivo n";
    }
    más{
        impresiónpsarchivo ya estricto 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 una 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 (según un parámetro de la línea de comandos), luego lee su entrada y llama a las rutinas de generación de código según el contenido de cada línea. No somos particularmente quisquillosos con el manejo de errores: nos enteraremos bastante rápido si las cosas salen mal.

```
mipsidioma =cambiar o morir "falta el idioma", $idioma .= "_cg.pm"
;
requerirpslang"o morir "No se pudo cargar $lang";
# Leer y analizar el archivo mips
nombre;
tiempo(<>)      {
    morder;
    si      (/^\sps            { CG::líneaenblanco();      }
    elseif   (/^\s\#(.*)/)     { CG::comentario($1);      }
    elsif(/^\METROS*(.+)/) { CG::startMsg($1); $nombre = $1; } elsif(/^\MI/)
                                { CG::endMsg($nombre); }
    elsif(/^\Fs*(w+)s*(w+)\$/) { \                                \
                                { CG::tipoSimple($1,$2); }}
```

```

    elsif(/^Fs*(w+)\((\d+)\)$/) \ \ \ \
        { CG::arrayType($1,$2,$3); } }

    más{
        morir "Línea no válida: $_[0]";
    }
}
}

```

Escribir un back-end de lenguaje es simple: proporcione un módulo que implemente los seis puntos de entrada requeridos. Aquí está el generador C:

```

#!/usr/bin/perl-w
paqueteGC;
usarestricto;

# Generador de código para 'C' (ver cg_base.pl)

sub linea en blanco() { impresión "norte" } sub
comentario() { impresión $ps0 } *n" } { impresión "estructura
sub iniciarMensaje() typedef {n" } { impresión $ps0 } nn" } \
sub finMensaje() \ \
subtipo de matriz () {
    mipsnombre, $tipo, $tamaño) = @_;
    impresión " $tipo $nombre[$tamaño]; norte";
}

sub tipoSimple() {
    mipsnombre, $tipo) = @_; impresión
    " $tipo $nombre; norte";
}
1;

```

Y aquí está el de Pascal:

```

#!/usr/bin/perl-w
paqueteGC;
usarestricto;

# Generador de código para 'Pascal' (ver cg_base.pl)

sub linea en blanco() { impresión "norte" } sub
comentario() { impresión $ps0 } n" } \
sub iniciarMensaje() { impresión $ps0 } = registro empaquetado n" } \
sub finMensaje() { impresión "final; nn" } \
subtipo de matriz () {
    mipsnombre, $tipo, $tamaño) = @_;
    $tamaño--;
    impresión " $nombre: arreglo[0..$tamaño] de $tipo; norte", \
}
sub tipoSimple() {
    mipsnombre, $tipo) = @_; impresión
    " $nombre: $tipo; norte";
}
1;

```

Ejercicio 14: de Diseño por contrato en la página 118

¿Qué hace un buen contrato? Cualquiera puede agregar condiciones previas y posteriores, pero ¿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 UNCUERDA a una lista circular doblemente enlazada (recuerde que las condiciones previas están etiquetadas con requerir, y poscondiciones con asegurar).

```

- - Añadir un artículo único a una lista doblemente enlazada,
- - y devolver el NODO recién creado.

add_item (elemento: CADENA) : NODOes
    requerir
        artículo /= Vacío
        find_item(elemento) = Vacío
    diferido
        asegurar
            resultado.siguiente.anterior = resultado--Revisa el recién
            resultado.anterior.siguiente = resultado--Enlaces de nodos añadidos.
            find_item(elemento) = resultado
            final
                -- '/=' es 'no igual'.
                -- Debe ser único
                -- Clase base abstracta.
                -- Debería encontrarlo.

```

Respuesta 14: Este ejemplo de Eiffel es *bueno*. Requerimos datos no nulos para ser y garantizamos que se respeta 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 optar por usar punteros, una matriz o lo que sea; mientras respete 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 del Ejercicio 14. insertaNúmero inserta un entero en una lista ordenada. Las condiciones previas y posteriores se etiquetan como en iContract (ver [URL 17]).

```

interno privado datos[];
/**
 * @post datos[índice-1] < datos[índice] &&
 *           datos[índice] == unValor
 */
público Nodo insertNumber (int finalun valor) {
    En tindex = findPlaceToInsert(aValue); ...

```

Respuesta 15: Esto es *malo*. Las matemáticas en la cláusula de índice (índice-1) no funcionará en condiciones de contorno como la primera entrada. La condición posterior asume una implementación particular: queremos que los contratos sean más abstractos que eso.

Ejercicio 16: de Diseño por contrato en la página 119

Aquí hay un fragmento de una clase de pila en Java. ¿Es este un buen contrato?

```
/**
 * @pre anItem != null           // Requerir datos reales
 * @post pop() == anItem // Verifica que sea
 *      // en la pila
 */
vacío público o empujar(final Cadena de un elemento)
```

Respuesta 16: Es un buen contrato, pero una mala implementación. Aquí, el infante mouse "Heisenbug" [URL 52] levanta su feocabeza. El programador probablemente cometió un simple error tipográfico: estallido en vez de parte superior. Si bien este es un ejemplo simple y artificial, los efectos secundarios en las afirmaciones (o en cualquier lugar inesperado en el 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 14–16) muestran una implementación de un ADT (Tipo de datos abstracto), generalmente una pila o una 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. Eventualmente será una licuadora basada en la Web, habilitada para Internet y compatible con CORBA, pero por ahora solo necesitamos la interfaz para controlarla. Tiene diez configuraciones de velocidad (0 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).

Aquí están los métodos. Agregue condiciones previas y posteriores apropiadas y un invariante.

```
En tobtenerVelocidad()
vacío velocidad fijada(En t X)
booleano está lleno()
vacío llenar()
vacío vacío()
```

Respuesta 17: Mostraremos las firmas de funciones en Java, con el pre- y condiciones posteriores etiquetadas como en iContract.

Primero, el invariante para la clase:

```
/**
 * @invariante obtenerVelocidad() > 0
 *             implica isFull()                                // No te quedes vacío
 * @invariante obtenerVelocidad() >= 0 &&
 *             obtenerVelocidad() < 10                         // Comprobación de rango
 */
```

A continuación, las condiciones previas y posteriores:

```

    /**
     * @pre Math.abs(getSpeed() - x) <= 1 // Solo cambia por uno
     * @pre x >= 0 && x < 10
     * @post obtenerVelocidad() == x
     */
    vacío público velocidad(fijada(int finalX)
    /**
     * @pre !isFull()
     * @post isFull()
     */
    vacío llenar()
    /**
     * @pre está lleno()
     * @post !isFull()
     */
    vacío vacío()

```

Ejercicio 18: de Diseño por contrato en la página 119

cuantos numeros hay en la serie 0, 5, 10, 15, ..., 100?

Respuesta 18: Hay 21 términos en la serie. Si dijiste 20, simplemente experimentaste cometió un error de poste de cerca.

Ejercicio 19: de Programación asertiva en la página 125

Una revisión rápida de la realidad. ¿Cuál de estas cosas "imposibles" puede suceder?

1. Un mes con menos de 28 días
- 2.estadística(".",&sb) == -1 (es decir, no 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: (un + 1) <= un

Respuesta 19:

- El 1 de septiembre de 1752 tuvo sólo 19 días. Esto se hizo para sincronizar el calendario. dars como parte de la Reforma Gregoriana.
- El directorio podría haber sido eliminado por otro proceso, es posible que no tenga permiso para leerlo, &sbpodría no ser válido, usted obtiene la imagen.
- Disimuladamente no especificamos los tipos deayb.La sobrecarga de operadores podría haber definido +, = o != para tener un comportamiento inesperado. También,aybpueden ser alias para la misma variable, por lo que la segunda asignación sobrescribirá el valor almacenado en la primera.
- En geometría no euclíadiana, la suma de los ángulos de un triángulo no suman .
Piense 180° un triángulo trazado 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 ocurrir en C y C++).

Ejercicio 20: de *Programación asertiva en la página 125* Desarrolle una clase de verificación de aserciones simple para Java.

Respuesta 20: Elegimos implementar una clase muy simple con una sola estática método, `PRUEBA`, que imprime un mensaje y un seguimiento de la pila si pasó condición el parámetro es falso.

```

paquete com.pragprog.util;
importar java.lang.System; // por salida()
importar java.lang.Subproceso; // por volcarPila()
clase pública afirmar {
    /** Escribir un mensaje, imprimir un seguimiento de la pila y salir si
     * nuestro parámetro es falso.
     */
    vacío estático público PRUEBA(booleano condición) {
        si(!condición) {
            Sistema.fuera.println("==== Aserción fallida ===="); Subproceso.dumpStack();

            Sistema.salir(1);
        }
    }
    // Banco de pruebas. Si nuestro argumento está 'bien', intente con una afirmación que //
    // tenga éxito, si 'falla' intente con una que falle
    vacío final estático público principal(Cadena de argumentos[]) {
        si(argumentos[0].comparar con("okey") == 0) {
            PRUEBA(1 == 1);
        }
        más si(argumentos[0].comparar con("fallar") == 0) {
            PRUEBA(1 == 2);
        }
        más{
            lanzarnuevoExcepción en tiempo de ejecución("Mal argumento");
        }
    }
}

```

Ejercicio 21: de *Cuándo usar excepciones en la página 128*

Al diseñar una nueva clase de contenedor, identifica las siguientes condiciones de error posibles:

1. No hay memoria disponible para un nuevo elemento en la `agregar` rutina
2. Entrada solicitada no encontrada en la `buscar` rutina
3. Nulo puntero pasado a la `agregar` rutina

¿Cómo se debe manejar cada uno? ¿Debe generarse un error, debe generarse una excepción o debe ignorarse la condición?

Respuesta 21: Quedarse sin memoria es una condición excepcional, por lo que nos sentimos ese caso (1) debería generar una excepción.

El hecho de no encontrar una entrada es probablemente una ocurrencia bastante normal. La aplicación que llama a nuestra clase de colección bien puede escribir código que verifique si hay una entrada 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 importante para la aplicación, entonces se puede agregar justificadamente al contenedor. Sin embargo, si no tiene sentido almacenar valores nulos, probablemente se debería lanzar una excepción.

Ejercicio 22: *de Cómo equilibrar los recursos en la página 136*

Algunos desarrolladores de C y C++ insisten en establecer un puntero paraNULOdespués 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 verificar que un puntero realmente apunte a una memoria válida. Un error común es desasignar un bloque de memoria y hacer referencia a esa memoria más adelante en el programa. Para entonces, la memoria señalada bien puede haber sido reasignada a algún otro propósito. Al poner el puntero enNULO, los programadores esperan evitar estas referencias maliciosas, en la mayoría de los casos, eliminando la referencia a unNULOpuntero generará un error de tiempo de ejecución.

Ejercicio 23: *de Cómo equilibrar los recursos en la página 136*

Algunos desarrolladores de Java insisten en establecer una variable de objeto paraNULOdespués de que hayan terminado de usar el objeto. ¿Por qué es una buena idea?

Respuesta 23: Al establecer la referencia anULO, reduce el número de punteros al objeto al que se hace referencia en uno. Una vez que este recuento llega a cero, el objeto es apto para la recolección de elementos no utilizados. Establecer las referencias anULOpuede ser importante para los programas de ejecución prolongada, donde los programadores deben asegurarse de que la utilización de la memoria no aumente con el tiempo.

Ejercicio 24: de 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:

```
# incluir      "fecha.h"
clase        Persona1 {
privado:
    fecha mi fecha de nacimiento;
pùblico:
    Persona1(Fecha &fecha de nacimiento);
//...
```

persona2.h:

```
clase        Fecha;
clase        Persona2 {
privado:
    Fecha *miFechaDeNacimiento;
pùblico:
    Persona2(Fecha &fecha de nacimiento);
//...
```

Respuesta 24: Se supone que un archivo de encabezado define la interfaz entre el implementación correspondiente y el resto del mundo. El archivo de encabezado en sí mismo no tiene necesidad de saber acerca de las partes internas delFechaclase: simplemente necesita decirle al compilador que el constructor toma unaFecha como parámetro. Entonces, a menos que el archivo de encabezado usefechas en funciones en línea, el segundo fragmento funcionará bien.

¿Qué tiene de malo el primer fragmento? En un proyecto pequeño, nada, excepto que estás haciendo innecesariamente todo lo que usa unPersona1clase también incluyen el archivo de encabezado paraFecha. Una vez que este tipo de uso se vuelve común en un proyecto, pronto se da cuenta de que incluir un archivo de encabezado termina incluyendo la mayor parte del resto del sistema, lo que supone un serio lastre para los tiempos de compilación.

Ejercicio 25: de 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.

```
vacío pùblicomostrar Saldo (Cuenta Bancaria) {
    cantidad de dinero = cuenta.obtenersaldo();
    printToscreen(amt.printFormat());
}
```

Respuesta 25: La variable cuenta pasa como un parámetro, por lo que el getBalance permite la llamada. Vocación amt.printFormat(), sin embargo, no lo es. No somos "dueños" de cantidad y no se nos pasó. Podríamos eliminar mostrarSaldo's de acoplamiento a Dinero con algo como esto:

```
vacío mostrarSaldo(CuentaBancaria b) {
    b.imprimirSaldo();
}
```

Ejercicio 26: de Desacoplamiento y la Ley de Deméter en la página 143 Este ejemplo también está en Java.

```

clase públicaColada {
    privado    licuadora mi licuadora;
    privado    Vector mis cosas;

    públicoColada()      {
        miBlender =nuevoLicuadora(); mis cosas
        =nuevoVector();
    }
    vacío privadohacer algo() {
        myBlender.addIngredients(myStuff.elements());
    }
}

```

Respuesta 26: Ya que Colada crea y posee ambos miBlender y mis cosas, las llamadas añadirIngredientes y elementos están permitidos.

Ejercicio 27: de Desacoplamiento y la Ley de Deméter en la página 143 Este ejemplo está en C++.

```

vacíoprocesarTransacción(Cuenta bancaria,En t) {
    Persona que;
    cantidad de dinero;
    cantidad.setValor(123.45);
    cuenta.establecerSaldo(cantidad);
    quien = cuenta.getOwner();
    markWorkflow(quién->nombre(), SET_BALANCE);
}

```

Respuesta 27: En este caso, procesoTransacción posee cantidad—se crea en la pila. cuenta se pasa, por lo que ambos valor ajustado y establecer el saldo están permitidos. Pero procesoTransacción no posee quién, entonces la llamada quien->nombre() es en violación. La Ley de Deméter sugiere reemplazar esta línea con

```
markWorkflow(cuenta.nombre(), SET_BALANCE);
```

El código en procesoTransacción no debería tener que saber qué subobjeto dentro de un Cuenta bancaria tiene el nombre, este conocimiento estructural no debería mostrarse a través Cuenta bancaria' contrato de s. En su lugar, le preguntamos a Cuenta bancaria para el nombre en la cuenta. Sabe dónde guarda el nombre (tal vez en un Persona, en un Negocio, o en un polimórfico Cliente objeto).

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. El 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 uso en pruebas unitarias

Respuesta 28: No hay respuestas definitivas aquí, las preguntas fueron destinado principalmente a darle algo en que pensar. Sin embargo, esto es lo que pensamos:

1. Asignaciones de puertos de comunicación. Claramente, esta información debe almacenarse como metadatos. Pero, ¿hasta qué nivel de detalle? Algunos programas de comunicaciones de Windows le permiten seleccionar solo la velocidad en baudios y el puerto (por ejemplo, COM1 a COM4). Pero tal vez necesite especificar el tamaño de palabra, la paridad, los bits de parada y también la configuración de dúplex. Trate de permitir el mejor nivel de detalle cuando sea práctico.
2. El soporte de un editor para resaltar la sintaxis de varios idiomas. Esto debe implementarse como metadatos. No le gustaría 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 querría sobrecargar su aplicación con varios controladores de dispositivo solo para seleccionar uno en tiempo de ejecución. Sin embargo, podría 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 disfuncional, es posible que no pueda usar el programa para restablecerlo.
4. Una 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 estándares 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 de forma externa.
5. Valores de muestra y resultados para su uso en pruebas unitarias. La mayoría de las aplicaciones defina estos valores en línea en el arnés de prueba, pero puede obtener una mayor flexibilidad al mover los datos de prueba, y la definición de los resultados aceptables, fuera del código mismo.

Ejercicio 29: de Es solo una vista en la página 164

Suponga que tiene un sistema de reservas de una aerolínea que incluye el concepto de un vuelo:

```
interfaz públicaVuelo {
    // Devuelve falso si el vuelo está lleno. booleano público
    addPassenger(Pasajero p); vacío públicoaddToWaitList(Pasajero
    p); público
        En t obtenerCapacidadVuelo();
    público En t obtenerNumPasajeros();
}
```

Si agrega un pasajero a la lista de espera, se lo colocará en el vuelo automáticamente cuando haya una vacante disponible.

Hay un trabajo de informes masivo que consiste en buscar vuelos llenos o con exceso de reservas para sugerir cuándo se pueden 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: lleva demasiado tiempo ejecutarlo. Utilice las ideas de esta sección para rediseñar esta interfaz.

Respuesta 29: TomaremosVueloy agregue algunos métodos adicionales para main- Contiene dos listas de oyentes: una para notificación de lista de espera y otra para notificación de vuelo completo.

```
interfaz públicaPasajero {
    vacío públicoListaEsperaDisponible();
}

interfaz públicaVuelo {
    ...
    vacío públicoaddWaitListListener(Pasajero p); vacío público
    removeWaitListListener(Pasajero p);
    vacío públicoaddFullListener(FullListener b);
    vacío públicoremoveFullListener(FullListener b); ...
}

interfaz públicaGran informeextiendeOyente completo {
    vacío públicoFlightFullAlert(Vuelo f);
}
```

Si tratamos de agregar unPasajeroy falla porque el vuelo está lleno, podemos, opcionalmente, poner elPasajeroen la lista de espera. Cuando se abre un lugar,lista de espera-disponiblesera llamado. Este método puede optar por agregar elPasajero automáticamente, o haga que un representante de servicio llame al cliente para preguntarle si todavía está interesado, o lo que sea. Ahora tenemos la flexibilidad de realizar diferentes comportamientos por cliente.

A continuación, queremos evitar tener laGran informerecorrer toneladas de registros en busca de vuelos completos. Por tenerGran informeregistrado como oyente envuelos,

cada individuo vuela podemos informar cuando está lleno, o casi lleno, si queremos. Ahora los usuarios pueden obtener informes en vivo y actualizados al minuto de gran informe al instante, sin esperar horas a que se ejecute como lo hacía anteriormente.

Ejercicio 30: de pizarras en la página 170

Para cada una de las siguientes aplicaciones, ¿sería adecuado o no un sistema de pizarra? ¿Por qué?

1. Procesamiento de imágenes. Le gustaría tener una serie de procesos paralelos que tomen fragmentos de una imagen, los procesen y vuelvan a colocar el fragmento completo.
2. Calendario de grupos. Tiene personas repartidas 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.

Respuesta 30:

1. Procesamiento de imágenes. Para la programación simple de una carga de trabajo entre los paralelos de los procesos, 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 de deformación de imágenes 3D complejas.

2. Calendario de grupos. Esto podría ser una buena opción. Puede publicar reuniones programadas y disponibilidad en la pizarra. Tiene entidades que funcionan de manera autónoma, la retroalimentación de las decisiones es importante y los participantes pueden ir y venir.

Es posible que desee considerar dividir este tipo de sistema de pizarra según 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 director ejecutivo 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 diferentes formatos solo 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 a donde es necesario, y no nos construye artificialmente.

3. Herramienta de monitoreo de red. Esto es muy similar al programa de solicitud de hipoteca/préstamo descrito en la página 168. Tiene informes de problemas enviados por los usuarios y estadísticas informadas automáticamente, todas publicadas en la pizarra. Un agente humano o de software puede analizar la pizarra para diagnosticar

fallas en la red: dos errores en una línea pueden ser simplemente rayos cósmicos, pero 20,000 errores y tienes un problema de hardware. Así como los detectives resuelven el misterio del asesinato, puedes tener múltiples entidades analizando y aportando ideas para resolver los problemas de la red.

Ejercicio 31:de Programación por Coincidencia en la página 176

¿Puedes identificar algunas coincidencias en el siguiente fragmento de código C? Suponga que este código está enterrado profundamente en una rutina de biblioteca.

```
fprintf(stderr, "Error, continuar?"); obtiene(buf);
```

Respuesta 31: Hay varios problemas potenciales con este código. Primero, asume untyambiente. Eso puede estar bien si la suposición es cierta, pero ¿qué pasa si este código se llama desde un entorno GUI donde niestándarni

Entrada estándar¿Esta abierto?

En segundo lugar, está la problemáticaobtiene,que escribirá tantos caracteres como reciba en el búfer pasado. Los usuarios malintencionados han utilizado esto sin poder crear saturación de búferagujeros de seguridad en muchos sistemas diferentes. Nunca usar obtiene().

Tercero, el código asume que el usuario entiende inglés.

Finalmente, nadie en su sano juicio enterraría una interacción de usuario como esta en una rutina de biblioteca.

Ejercicio 32:de Programación por Coincidencia en la página 176

Esta pieza de código C podría funcionar algunas veces, en algunas máquinas. Entonces de nuevo, tal vez no. ¿Qué ocurre?

```
/* Trunca la cadena a sus últimos caracteres maxlen */
vacíocadena_cola(carbonizarse*cuerda,En tmaxlen) {
    En tlen = strlen(cadena); si(len > maxlen)
    {
        strcpy(cadena, cadena + (len - maxlen));
    }
}
```

Respuesta 32: POSIXestresadono se garantiza que funcione para cadenas superpuestas. Puede 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 seguimiento de Java de uso general. La función escribe una cadena en un archivo de registro. Pasa su prueba unitaria, pero falla cuando uno de los desarrolladores web lo usa. ¿En qué coincidencia se basa?

```
vacío estático público depurar (Cadena s)lanza IOException {
    FileWriter fw = nuevoFileWriter("registro de depuración", verdadero);
    fw.write(s);
    fw.flush();
    fw.cerrar();
}
```

Respuesta 33: No funcionará en un contexto de applet con restricciones de seguridad. contra la escritura en el disco local. Nuevamente, cuando tenga la opción de ejecutar en contextos 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 otro lugar que no sea el disco local si no se puede acceder a él.

Ejercicio 34: de Algorithm Speed 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 acerca de las velocidades relativas de sus máquinas? ¿Cuáles son los efectos de varias configuraciones de optimización del compilador? ¿Es realmente lineal el tipo radix?

Respuesta 34: Claramente, no podemos dar respuestas absolutas a este ejercicio. Sin embargo, podemos darle un par de consejos.

Si encuentra que sus resultados no siguen una curva suave, es posible que desee verificar si alguna otra actividad está utilizando parte de la potencia de su procesador. Probablemente no obtendrá buenas cifras en un sistema multiusuario, e incluso si es el único usuario, es posible que los procesos en segundo plano quiten periódicamente ciclos de sus programas. También es posible que desee verificar la memoria: si la aplicación comienza a usar el espacio de intercambio, el rendimiento se desplomará.

Es interesante experimentar con diferentes compiladores y diferentes configuraciones de optimización. Descubrimos algunos que eran posibles aceleraciones bastante sorprendentes al permitir una optimización agresiva. También descubrimos que en las arquitecturas RISC más amplias, los compiladores del fabricante a menudo superaban al GCC más portátil. Presumiblemente, el fabricante conoce los secretos de la generación eficiente de código en estas máquinas.

Ejercicio 35: de *Algorithm Speed* en la página 183

La siguiente rutina imprime el contenido de un árbol binario. Suponiendo que el árbol esté equilibrado, ¿cuánto espacio de pila usará 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).

```
vacíoimprimirÁrbol(constanteNodo *nodo) {
    carbonizarsebúfer[1000];
    si(nodo) {
        imprimirÁrbol(nodo->izquierda);
        getNodeAsString(nodo, búfer); pone (búfer);

        imprimirÁrbol(nodo->derecha);
    }
}
```

Respuesta 35: Los `imprimirÁrbol` rutina usa alrededor de 1,000 bytes de espacio de pila para el búfer variable. Se llama a sí mismo recursivamente para descender a través del árbol, y cada llamada anidada agrega otros 1000 bytes a la pila. También se llama a sí mismo cuando llega a los nodos hoja, pero sale inmediatamente cuando descubre que el puntero pasado es `NULO`. Si la profundidad del árbol es D , el requisito de pila máxima es aproximadamente $1000 \times (D + 1)$.

Un árbol binario equilibrado contiene el doble de elementos en cada nivel. Un árbol de profundidad D sostiene $2 + 4 + 8 + \dots + 2^{(D-1)}$, o $2^D - 1$, elementos. Nuestro elemento del millón por lo tanto, el árbol necesitará $\lceil \lg(1,000,000) \rceil$, o 20 niveles.

Por lo tanto, esperaríamos que nuestra rutina use aproximadamente 21,000 bytes de pila.

Ejercicio 36: de *Algorithm Speed* en la página 183

¿Puede ver alguna forma de reducir los requisitos de pila de la rutina del ejercicio 35 (aparte de reducir el tamaño del búfer)?

Respuesta 36: Un par de optimizaciones vienen a la mente. Primero el `imprimirÁrbol` la rutina se llama a sí misma en los nodos hoja, solo para salir porque no hay hijos. Esta llamada aumenta la profundidad máxima de la pila en unos 1000 bytes. También podemos eliminar la recursividad de la cola (la segunda llamada recursiva), aunque esto no afectará el uso de la pila en el peor de los casos.

```
tiempo(nodo) {
    si(nodo->izquierda) printTree(nodo->izquierda);
    getNodeAsString(nodo, búfer); pone (búfer);

    nodo = nodo->derecho;
}
```

Sin embargo, la mayor ganancia proviene de la asignación de un solo búfer, compartido por todas las invocaciones de imprimirÁrbol. Pase este búfer como un parámetro para las llamadas recursivas y solo se asignarán 1000 bytes, independientemente de la profundidad de la recursividad.

```

vacíoimprimirÁrbolPrivado(constanteNodo *nodo,carbonizarse*buffer) {
    si(nodo) {
        printTreePrivate(nodo->izquierda, búfer);
        getNodeAsString(nodo, búfer); pone (búfer);

        printTreePrivate(nodo->derecho, búfer);
    }
}
vacónuevoÁrbolImpresión(constanteNodo *nodo) {
    carbonizarsebúfer[1000];
    printTreePrivate(nodo, búfer);
}

```

Ejercicio 37: de Algorithm Speed en la página 183

En la página 180, dijimos que una tajada binaria es $O(\lg(n))$. ¿Puedes probar esto?

Respuesta 37: Hay un par de maneras de llegar allí. Una es girar el problema en su cabeza. Si la matriz tiene solo un elemento, no iteramos alrededor del ciclo. Cada iteración adicional duplica el tamaño de la matriz que podemos buscar. Por lo tanto, la fórmula general para el tamaño de la matriz es $n = 2^m$, donde **esta** el número de iteraciones. Si llevas troncos a la base 2 de cada lado, obtienes $\lg(n) = \lg(2^m)$, que por la definición de registros 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. Refactorizarlo.

```

si(estado == TEXAS) {
    Velocidad = TASA_TX;
    cantidad = base * TASA_TX;
    calc = 2*base(importe) + adicional(importe)*1.05;
}
más si((estado == OHIO) || (estado == MAINE)) {
    tasa = (estado == OHIO) ? TASA_OH : TASA_ME;
    amt = base * tasa;

    calc = 2*base(importe) + adicional(importe)*1.05; si(estado
    == OHIO)
        puntos = 2;
}
más{
    Velocidad = 1;
    cantidad = base;
    calc = 2*base(importe) + adicional(importe)*1.05;
}

```

Respuesta 38: Podríamos sugerir una reestructuración bastante suave aquí: asegúrese de que cada prueba se realice una sola vez, y que todos los cálculos sean comunes. Si la expresión $2 * \text{base}(...)^2 * 1.05$ aparece en otros lugares del programa, probablemente deberíamos convertirlo en una función. No nos hemos molestado aquí.

Hemos agregado unrate_lookuparray, inicializado para que las entradas que no sean Texas, Ohio y Maine tengan un valor de 1. Este enfoque facilita agregar valores para otros estados en el futuro. Dependiendo del patrón de uso esperado, es posible que deseemos hacer que el puntos ficampo una búsqueda de matriz también.

```

Velocidad = rate_lookup[estado];
cantidad = base * tasa;
calc = 2*base(importe) + adicional(importe)*1.05;
si(estado == OHIO)
    puntos = 2;
```

Ejercicio 39: de Refactorización en la página 188

La siguiente clase de Java necesita admitir algunas formas más. Refactorice la clase para prepararla para las adiciones.

```

clase públicaForma {
    int final estático público CUADRADO int final      = 1;
    estático público CIRCULO          = 2;
    int final estático público TRIÁNGULO_DERECHO = 3;
    privado   En t      tipo de forma;
    privado   doble     Talla;
    públicoForma(En t tipo de forma, dobleTalla) {
        este.shapeType      = tipo de forma;
        este.Talla          = Talla;
    }
    //... otros métodos ...
    público dobleárea() {
        cambiar (tipo de forma) {
            caso CUADRADO: devolver tamaño*tamaño;
            caso CIRCULO:  devolver Matemáticas.PI*tamaño*tamaño/4.0;
            caso TRIÁNGULO RECTÁNGULO:devolvertamaño*tamaño/2.0; }

        devolver0;
    }
}
```

Respuesta 39: Cuando vea a alguien usando tipos enumerados (o sus equivalentes) alent en Java) para distinguir entre variantes de un tipo, a menudo puede mejorar el código creando subclases:

```

clase públicaForma {
    doble privadoTalla;
    público Forma(dobleTalla) {
        este.tamaño = tamaño;
    }
    publico doble obtenerTamaño() {devolverTalla; }
}
clase públicaCuadradoextiendeForma {
    público Cuadrado(dobleTalla) {
        súper(Talla);
    }
    publico doble área() {
        doble tamaño = obtenerTamaño();
        devolver tamaño*tamaño;
    }
}
clase públicaCírculoextiendeForma {
    público Círculo(dobleTalla) {
        súper(Talla);
    }
    publico doble área() {
        doble tamaño = obtenerTamaño(); devolver
        Matemáticas.PI*tamaño*tamaño/4.0;
    }
}
//etc...
```

Ejercicio 40: de Refactorización en la página 189

Este código Java es parte de un marco que se utilizará a lo largo de su proyecto. Refactorícelo para que sea más general y más fácil de extender en el futuro.

```

clase públicaVentana {
    público Ventana(En tancho,En taltura) { ... }
    vacío público establecerTamaño(En tancho,En taltura) { ... }
    booleano público se superpone (Ventana w) { ... }
    public int obtenerÁrea() { ... }
}
```

Respuesta 40: Este caso es interesante. A primera vista, parece razonable que una ventana debe tener un ancho y una altura. Sin embargo, considere el futuro. Imaginemos que queremos admitir ventanas con formas arbitrarias (lo que será difícil si la VentanaClase sabe todo acerca de los rectángulos y sus propiedades).

Sugerimos abstraer la forma de la ventana de la VentanaClase en sí.

```

clase pública abstractaForma {
    //...
    booleano abstracto públicosuperposiciones (Forma s); resumen
    público inttenerÁrea();
}

clase públicaVentana {
    privadoforma forma;
    públicoVentana (forma de forma) {
        este.forma = forma; ...
    }
    vacío públicosetShape(Forma forma) {
        este.forma = forma; ...
    }
    booleano públicose superpone (Ventana w) {
        devolverforma.superposiciones(w.forma);
    }
    public inttenerÁrea() {
        devolverforma.getArea();
    }
}

```

Tenga en cuenta que en este enfoque hemos utilizado la delegación en lugar de la subclasificación: una ventana no es una "especie de" forma, una ventana "tiene una" forma. Utiliza una forma para hacer su trabajo. A menudo encontrará útil la delegación al refactorizar.

También podríamos haber ampliado este ejemplo mediante la introducción de una interfaz Java que especificara los métodos que debe admitir una clase para admitir las funciones de forma. Esta es una buena idea. Significa que cuando extiende el concepto de una forma, el compilador le advertirá sobre las clases que ha afectado. Recomendamos usar interfaces de esta manera cuando delega 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 blender descrita en la respuesta al Ejercicio 17 en la página 289. Escriba un script de shell que realice una prueba de regresión para blender. Debe probar la funcionalidad básica, las condiciones de límite y de error y cualquier obligación contractual. ¿Qué restricciones se imponen al cambiar la velocidad? ¿Están siendo honrados?

Respuesta 41: Primero, agregaremos un principal para actuar como un controlador de prueba de la unidad. aceptará un lenguaje muy pequeño y simple como argumento: "E" para vaciar la licuadora, "F" para llenarla, dígitos 0-9 para configurar la velocidad, etc.

```

vacío estático público principal(Cadena de argumentos[])
{
    // Crea la licuadora para probar licuadora
    dbc_ex =nuevodbc_ex();
    // Y probarlo de acuerdo con la cadena en la entrada estándar probar{

        En t a;
        carbonizarse C;
        tiempo((a = Sistema.en.leer()) != -1) {
            c = (carbonizarse)a;
            si (Carácter.es un espacio en blanco (c)) {
                Seguir;
            }
            si(Carácter.isDigit(c))
                blender.setSpeed(Carácter.dígito(c, 10));
            }
            más{
                cambiar (C){
                    caso 'F': licuadora.fill();
                    descanso;
                    caso 'MI': licuadora.vacío();
                    descanso;
                    caso's': Sistema.out.println("VELOCIDAD: "+
                        licuadora.getSpeed());
                    descanso;
                    caso'f': Sistema.out.println("COMPLETO "+
                        blender.isFull());
                    descanso;
                    predeterminado: tirar nuevaExcepción en tiempo de ejecución
                        "Directiva de prueba desconocida";
                }
            }
        }
        captura(java.io.IOException e){
            Sistema.err.println("La plantilla de prueba falló:" +e.getMessage());
        }
        Sistema.err.println("Combinación completada n"); Sistema.salir(0);
    }
}

```

Luego viene el script de shell para conducir las pruebas.

```

#!/bin/sh
CMD="Java      dbcdbc_ex"
reuento de errores = 0
esperar_bien() {
    siecops| $CMD #>/desarrollo/nulo 2>&1
    después
    :
    más
    eco ";HA FALLADO! ps cuentafallas='expr
    $cuentafallas + 1'
    fi
}
esperar_fallar() {
    siecops| $CMD >/desarrollo/null 2>&1
    después
    eco ";HA FALLADO! (Debería haber fallado): $*"
    cuentafallas='expr $cuentafallas + 1'
    fi
}
reporte() {
    sipsreuento de errores -gt 0 ]
    después
    eco -e "nn*** PRUEBAS FALLIDAS $contador de fallas n" salida1#Por si
    somos parte de algo más grande
    más
    salida0#Por si somos parte de algo más grande
    fi
}
#
# Iniciar las pruebas
#
esperar_bien F123456789876543210E#debe ejecutar a través de esperar_fallar
F5          # Falla, velocidad demasiado alta
esperar_fallar 1      # Falla, vacío
expect_fail F10E1#falla, vacío expect_fail F1238#
falla, salta esperar_bien
    FE      # Nunca gire      en
esperar_fallar  F1E     # Vaciado      mientras corre
esperar_bien   F10E    # Debería      estar bien
reporte        # Reporte      resultados

```

Las pruebas verifican si se detectan cambios de velocidad ilegales, si intenta vaciar la licuadora mientras está funcionando, etc. Ponemos esto en el archivo MAKE para que podamos compilar y ejecutar la prueba de regresión simplemente escribiendo

```
% hacer
% hacer prueba
```

Tenga en cuenta que tenemos la salida de prueba con lo entones podemos usar esto como parte de una prueba más grande también.

No había nada en los requisitos que hablara de manejar este componente a través de un script, o incluso usando un lenguaje. Los usuarios finales nunca lo verán. Pero tenemos una poderosa herramienta que podemos usar para probar nuestro código, de forma rápida y exhaustiva.

Ejercicio 42: de *El Pozo de Requisitos en la página 211*

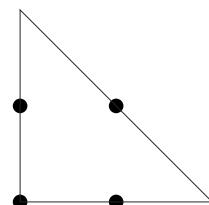
¿Cuáles de los siguientes son probablemente requisitos genuinos? Repita los que no lo son para hacerlos más útiles (si es posible).

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 ingresa 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 256kB.

Respuesta 42:

1. Esta declaración suena como un requisito real: puede haber restricciones colocadas en la aplicación por su entorno.
2. Aunque esto puede ser un estándar corporativo, no es un requisito. Sería mejor expresarlo como "El fondo del diálogo debe ser configurable por el usuario final. Tal como se envía, el color será gris". Aún mejor sería la declaración más amplia "Todos los elementos visuales de la aplicación (colores, fuentes e idiomas) deben ser configurables por el usuario final".
3. Esta declaración no es un requisito, es arquitectura. Cuando te enfrentas a algo como esto, tienes 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 le advertirá cuando se realicen estas entradas".
5. Esta declaración es probablemente un requisito difícil.

Una solución al rompecabezas de los Cuatro Postes planteado en la página 213.



Esta página se dejó en blanco intencionalmente

Índice

A

Función accesoria, 31
ACM, ver Asociación de Informática
 Maquinaria
 Generador de código activo, 104
Diagrama de actividad, 150
Estilos de programación avanzados en C++ y Modismos, 265
Programación Avanzada en Unix Ambiente, 264
Configuración basada en transacciones de Aegis
 gestión, 246, 271
Agente, 76, 117, 297
Algoritmo
 corte binario, 180
 elegir, 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 análisis, 264
Anonimato, 258
AOP, ver Arquitectura de programación orientada a aspectos
 despliegue, 156
 flexibilidad, 46
 creación de prototipos, 55
 desacoplamiento temporal, 152
Arte de la programación informática, 183
Inteligencia artificial, merodeadores, 26
Programación orientada a aspectos (AOP), 39, 273
Afirmación, 113, 122, 175
 efectos secundarios, 124

apagar, 123
Asociación para Maquinaria de Computación (ACM), 262
 Comunicaciones de la ACM, 263
 SIGPLAN, 263
Suposiciones, pruebas, 175
"a"mando, 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
intencocáscara, 80, 82n
Frijol, ver Enterprise Java Beans (EJB)
Beck, Kent, 194, 258
proyecto Beowulf, 268
Notación "grande", 177
"Imagen grande", 8
Corte binario, 97, 180
Formato binario, 73
 problemas de análisis, 75
bisonte, 59, 269
BIST, ver Sistema de pizarra de autodiagnóstico 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, ver *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
contrásopa de piedra,
 9 Brooks, Fred, 264
- navegador, clase, 187
- Navegador, refactorización, 187, 268
- Error, 90
 contrato fallido como, 111 ver
también Depuración; Error
- Construir
 automatización, 88, 233
 dependencias, 233
 definitivo, 234
 todas las noches, 231
 refactorización, 187
- Autoprueba integrada (BIST), 189
- Lógica empresarial, 146
- Política empresarial, 203

C

-
- lenguaje C**
 afirmaciones, 122
 DBC, 114
 duplicación, 29
 manejo de errores, 121
 mensajes de error, 115
 macros, 121
 Interfaz Object Pascal, 101
- Lenguaje C++, 46**
 afirmaciones, 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
- culpar, 3**
- pastoreo, 224**
- de Schrödinger, 47**
- Catalizando el cambio, 8**
- catedrales, xx**
- enlaces Cetus, 265**
- Cambio, catalizador, 8**
- Christiansen, Tom, 81 años**
- Clase**
 afirmaciones, 113
 base, 112
 acoplamiento, 139, 142
 relaciones de acoplamiento, 242
 recurso encapsulante, 132
 invariante, 110, 113
 número de estados, 245
 asignación de recursos, 132
 subclase, 112
 contendedor, 132, 133, 135, 141
- Navegador de clases, 187**
- ClearCase, 271**
- Cockburn, Alistair, xxiii, 205, 264, 272**
- Generador de código, 28, 102**
 activo, 104
 archivos MAKE, 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 del servidor, 196
- “tímido”, 40, 138
- especificaciones, 219
- balas trazadoras, 49–51
- pruebas unitarias, 190, 192
 - ver también* código acoplado; desacoplado
 - código; metadatos; Sistema de control de código fuente (SCCS)
- Cohesión, 35
- COM, *ver* Modelo de objetos
- componentes Explosión combinatoria, 140, 167 Algoritmo combinatorio, 180
- Shell de comando, 77
 - intento, 80
 - Cygwin, 80
 - contr*interfaz gráfica de usuario, 78
 - UWIN, 81
 - ventanas, 80
- Comentario, 29, 249
 - evitando duplicaciones, 29
 - DBC, 113
 - parámetros, 250
 - tipos de, 249
 - innecesario, 250
 - ver también* Documentación
- Agente de solicitud de objetos comunes (CORBA), 29, 39, 46
- Servicio de Eventos, 160
- Comunicar, 18
 - audiencia, 19, 21
 - duplicación, 32
 - correo electrónico, 22
 - y métodos formales, 221
 - presentación, 20
 - estilo, 20
 - equipos, 225
 - usuarios, 256
 - escribir, 18
- Comunicaciones de la ACM*, 263 Preguntas frecuentes sobre objetos comp., 272
- Compilación, 232
 - compiladores, 267
 - DBC, 113
 - advertencias y depuración, 92
 - Modelo de objetos componentes (COM), 55
 - Sistemas basados en componentes, *ver* Sistema modular
- concurrencia, 150
 - diseño, 154
 - interfaces, 155
 - y Programación por Coincidencia, 154
 - análisis de requisitos de, 150 flujo de trabajo, 150
- Sistema de versión concurrente (CVS), 271
- Configuración
 - 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
 - ver también* Diseño por contrato (DBC)
- Controlador (MVC), 162
- Coplien, Jim, 265
- CORBÁ, *ver* Solicitud de objeto común
 - Corredor
 - código acoplado, 130
 - relaciones de acoplamiento, 242
 - minimizando, 138, 158
 - actuación, 142
 - acoplamiento temporal, 150
 - ver también* Análisis de cobertura
 - de código desacoplado, 245
- Cox, Brad J., 189n
- Choque, 120
- Pensamiento crítico, 16
- cron, 231
- css, *ver* Hojas de estilo en cascada CVS, *ver* Sistema de versión concurrente Cygwin, 80, 270

D

- Datos
 - sistema de pizarra, 169
 - almacenamiento en caché, 31
- diccionario, 144
- estructuras de datos dinámicos, 135
- global, 40
- lenguaje, 60
- normalizando, 30

- legible *contracomprensible*, 75
- prueba, 100, 243
- vistas, 160
- visualizando, 93
 - ver también* Metadatos
- Depurador de visualización de datos (DDD), 93, 268
- Base de datos
 - generador de código activo, esquema 104, 105f, 141, mantenimiento de esquema 144, 100 DBC, *ver* Diseño por contrato DDD, *ver*
- Fecha límite del depurador de visualización de datos, 6, 246
- punto muerto, 131
- Depuración, 90
 - afirmaciones, 123
 - búsqueda binaria, 97
 - ubicación de errores, 96
 - reproducción de insectos, 93
 - lista de control, 98
 - advertencias del compilador y, 92
 - variables corruptas, 95
 - Heisenbug, 124
 - patito de goma, 95
 - y bifurcación del código fuente, 87
 - error sorpresa, 97
 - y prueba, 92, 195
 - bomba de tiempo, 192
 - rastreo, 94
 - ver, 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
 - minimizar el acoplamiento, 138
 - pruebas modulares, 244
 - desacoplamiento físico, 142
 - acoplamiento temporal, 150
 - flujo de trabajo, 150
 - ver también* Código acoplado
- Codificación defensiva, 107
- Delegación, 304
- Delfos, 55
 - ver también* Objeto
- projeto Pascal Demeter, 274
- Deméter, Ley de, 140
- dependencia, reducción, *ver* Modular sistema; ortogonalidad
- Despliegue, 156
- Descriptor de implementación, 148
- Diseño
 - funciones accesorias, 31
 - conurrencia, 154
 - contexto, 174
 - despliegue, 156
 - pruebas de diseño/metodología, 242
 - metadatos, 145
 - ortogonalidad, 34, 37
 - físico, 142
 - refactorización, 186
 - usar servicios, 154
- Diseño por contrato (DBC), 109, 155
 - y agentes, 117
 - afirmaciones, 113
 - clase invariante, 110
 - como comentarios, 113
 - contratos dinámicos, 117
 - iContrato, 268
 - soporte lingüístico, 114
 - ejemplo de inserción de lista, 110, 113, 114
 - predicados, 110
 - pruebas unitarias, 190
- Patrones de diseño, 264
 - observador, 158
 - soltero, 41
 - estrategia, 41
- Destructor, 132
- detectives, 165
- Árbol de desarrollo, 87
- Desarrollo, iterativo, 69 Algoritmo divide y vencerás, 180 Generador de documentación DOC++, 251, 269
- Libro de documentos, 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 marcas, 254
- ortogonalidad, 42
- esquema, 18
- requisitos, 204
- escritores técnicos, 252
- procesadores de texto, 252, 254
- especificaciones de escritura, 218
- ver también* Comentario; Web documentación
- Dodó, 148
- Dominio, problema, 58, 66 No te repitas, *ver* SECO principio
- Descargando código fuente, *ver* Ejemplo código
- Diario del Dr. Dobbs*, 263
- Principio SECO, 27, 29, 42
 - ver también* Duplicación
- pato, goma, *ver* Pato de goma
- Dumpty, Humpty, xxii, 165
- Duplicación, 26
 - los generadores de código evitan, 28 y las 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 *Dinámica del desarrollo de software*, 264
- 
- correo electrónico, 22
 - dirección para comentarios, xxiii
- Editor, 82
 - sangría automática, 85
 - movimiento del cursor, 84
 - características, 83
 - generar código, 103
 - cuantos aprender, 82
 - plantilla, 84
 - tipos de, 266
- Bloc de notas de Windows, 84
- C++ efectivo, 265 Eiffel, 109, 114, 267
- EJB, *ver* Beans de Java empresariales elvisredactor, 267
- Editor de Emacs, 84, 266
 - vibora viemulador, 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 de dominio, 59
 - accidente temprano, 120
 - mensajes de registro, 196
 - ortogonalidad, 41
 - pruebas, 240, 247
 - ver también* Excepción
- Controlador de errores, 127
- Estimación, 64
 - exactitud, 64
 - algoritmos, 177, 178
 - iterativo, 69
 - modelos, 66
 - dominio del problema, 66
 - cronogramas de proyectos, 68
 - registros, 68
 - pruebas, 182
- Universidad de Eton, XXI
- Evento, 157
- canal de eventos, 160
- Código de ejemplo
 - añadir registro, 40
 - reservas aéreas, 164, 296
 - afirmmacro, 122
 - auto_pregemplo, 134
 - mal equilibrio de recursos, 129, 130
 - descargando, xxiii
 - manejo de errores de excepción, 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 de cadenas con
 Tokenizador de cadenas, 156
análisis de cadenas constrtok, 155
clase no normalizada, 30
Código de ejemplo por nombre
 AOP, 40
 Varios.java, 156
afirmar, 122
mal_saldo.c, 129, 130
equilibrio.cc, 134
equilibrio.c, 131–133
línea de clase, 30, 31
excepción, 125
encontrar pico, 250
interfaz de vuelo, 164, 296
misc.c, 155
openpasswd.java, 126
abrirarchivodeusuario.java, 127
parcelafecha, 139
efecto secundario, 124
espaguetis, 125
cuadrado, 190
excepción, 121
 efectos de, 127
 y manejadores de errores, 127
archivos faltantes, 126
equilibrio de recursos, 132
cuándo usar, 125
excusas, 3
Documento ejecutable, 251
suponer, 269
Experto, *ver*Gurú
Activo que expira, 12
Lenguaje de estilo extensible (XSL), 253
Extinción, 148
Programación extrema, 238n, 258, 272
registro, 196
archivo make, 232
fuente, 103
Construcción final, 234
Peces, peligros de, 34
Flexibilidad, 46
Métodos formales, 220, 221 Four Posts
Puzzle, 213 Fowler, Martin, xxiii, 186,
273 Free Software Foundation, *ver*GNU
Proyecto
Rana, hervida, *ver*Función de
rana hervida
 accesorio, 31
 Ley de Deméter para *s*, 140
 similares, 41
-
- GRAMO**
- Gama, Erich, 194
Recolección de basura, 134
Metáfora de jardinería, 184
Gehrke, Peter, xxiv
Glass, Robert, 221, 236 Variables
globales, 40, 130, 154 Glosario,
proyecto, 210
Proyecto GNU, 274
 compilador C/C++, 267
 Licencia pública general (GPL), Programa de
 manipulación de imágenes 80 GNU
 (GIMP), 274
 PequeñoEiffel, 267
“Software lo suficientemente bueno”, *ver*Software,
 calidad
nudo gordiano, 212
Ir a, 127
- sistema de interfaz gráfica de usuario
 contrashell de comandos,
 78 interfaz, 78
 prueba, 244
Gurú, 17, 198

F

- fluencia característica, 10
Comentarios, dirección de correo electrónico, Archivo
xxiii
excepción, 126
cabecera, 29
implementación, 29

H

- Hachís, seguro, 74
archivo de cabecera, 29
“Heisenbug”, 124, 289
Helicóptero, 34n
Hopper, Grace, 8n, 90 Secuencia de
“tecla 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, *ver*
 Desarrollo Integrado Ambiente Sociedad informática IEEE, 262 *Computadora IEEE*, 262 *Software IEEE*, 263 Lenguaje imperativo, 60 Implementación accidentes, 173 codificación, 173 especificaciones, 219 Duplicación impuesta, 28 Duplicación inadvertida, 30 Sangría, automática, 85 Independencia, *ver* Infraestructura de ortogonalidad, 37 Herencia, 111 afirmaciones, 113 abanico de entrada/salida, 242 tenis interior, 215 inspección, código, *ver* Revisiones de código Insure++, 136 Circuito integrado, entorno de desarrollo integrado 189n (IDE), 72, 232 Plataforma de integración, 50 Pruebas de integración, 239 Interfaz sistema de pizarra, 168 C/Objeto Pascal, 101 concurrencia, 155 controlador de errores, 128 interfaz gráfica de usuario, 78 creación de prototipos, 55 usuario, 203 Invariante, 110, 113, 155 bucle, 116 semántico, 116, 135 Formato ISO9660, 233n Desarrollo iterativo, 69

j

Jacobson, Ivar, 204 Jerga, XXII, 210 archivo de jerga, 273 Java, 46, 267 generación de código, 232 DBC, 114 Enterprise Java Beans, 39, 147 mensajes de error, 115 excepciones, 121 iContract, 110, 114, 268 javacc, 59, 269 JavaDoc, 248, 251 JavaSpaces, 166, 273 JUnidad, 195 programación multiproceso, 154 acceso a la propiedad, 100 archivos de 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, *ver* Java

k

Entorno de escritorio K, 273 Kaizen, xxi, 14 *ver también* Portafolio de conocimientos Kernighan, Brian, 99 Combinación de teclas, 82 Kirk, James T., 26 Conocimiento productores y consumidores, 166 Portafolio de conocimiento, 12 edificio, 13 pensamiento crítico, 16 aprender y leer, 14 investigar, 15 Knuth, Donald, 183, 248 Korn, David, 81 Kramer, Reto, XXIV Kruchten, Philippe, 227n

l

Lakos, John, xxiv, 9, 142, 265 Lame excusas, 3

- lenguaje, programacion
 conversiones, 103, 105
 DBC, 114
 dominio, 57
 duplicación en, 29
 aprendizaje, 14
 prototipos, 55
 guión, 55, 145
 especificación, 58, 62
 manipulación de texto, 99
ver también Mini-idioma
- Diseño de software C++ a gran escala*, 142, 265
- LaSistema TEX, 103 Ley de Deméter, 140
- Céspedes, cuidado de, xxi Diseño en capas, 37
- sistema de capas,*ver* Sistema modular
- código "perezoso", 111
- Lex y Yacc*, 59 Bibliotecario,*ver* Bibliotecario de proyectos Código de biblioteca, 39
- Linda modelo, 167
- Algoritmos lineales, 177
- Linux, 15, 254, 265
- Principio de sustitución de Liskov, 111
- Escuchar, 21
- Programación alfabetizada, 248
- Registro, 39, 196
ver también Rastreo
- Tabla de consulta, 104
- Círculo
 anidado, 180
 sencillo, 180
- Bucle invariante, 116
-
- METRO
- Macro, 78, 86
 afirmaciones, 122
 documentación, 252
 manejo de errores, 121
 mantenimiento, 26
 lenguajes imperativos, 61
- Makefile, 232
 recursivo, 233
- Gestión de expectativas, 256
- Lenguaje de marcas, 254
- Martín, Roberto C., 273
- Métrica de complejidad ciclomática de McCabe, 242
- variables miembro,*ver* Accesorio funciones
- Asignación de memoria, 135
- Metadatos, 144, 203
 lógica de negocios, 146
 configuración, 147
 control de transacciones, 39
 código desacoplado, 145
 y métodos formales, 221 en texto plano, 74
- métrico, 242
- Meyer, Bertrand, 31n, 109, 184, 264
- Meyer, Scott, 265
- Microsoft Visual C++, 198
- Microsoft Windows, 46
- Minilenguaje, 59
 lenguaje de datos, 60
 incrustado, 62
 imperativo, 60
 análisis, 62
 independiente, 62
- Mesa de mezclas, 205
- Integridad de 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
 creación de prototipos, 55
 asignación de recursos, 135
 reversibilidad, 45
 pruebas, 41, 190, 244
- C++ más efectivo*, 265
- Mozilla, 273
- Programación multiproceso, 154
- MVC,*ver* Modelo-vista-controlador *El mes del hombre mítico*, 264
-
- norte
- 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
 singleton, 41
 estado válido/no válido, 154
 espectador, 163
 Grupo de gestión de objetos (OMG), 270
 Object Pascal, 29
 interfaz C, 101
Programación orientada a objetos, 189n
Construcción de software orientada a objetos, 264
 Obsolescencia, 74
 oltp, verTransacción en línea
 Sistema de procesamiento
 DIOS MÍO, verSistema de procesamiento de
 transacciones en línea del grupo de gestión de objetos
 (OLTP), 152
 Opciones, proporcionar, 3
 ordenar, verflujo de trabajo
 ortogonalidad, 34
 codificación, 34, 36, 40
 diseño, 37
 documentación, 42
 principio SECO, 42
 sistema no ortogonal, 34
 productividad, 35
 equipos de proyecto, 36, 227
 pruebas, 41
 juegos de herramientas y
 bibliotecas, 39 ver tambiénSistema
 modular Sobre embellecedor, 11

PAGS

Manejo del dolor, 185
 pintar()método, 173
 pintura, 11
 Papúa Nueva Guinea, 16

Programación en paralelo, 150
 Parrots, killer, verAnálisis de
 marca, 59
 generadores de código, 105
 mensajes de registro, 196
 minilenguaje, 62
 cuerdas, 155
 Partición, 168
 Pascual, 29
 Generador de código pasivo, 103
 Pruebas de rendimiento, 241
 Perl, 55, 62, 99
 Interfaz C/Object Pascal, 101 generación
 de esquema de base de datos, 100 página
 de inicio, 267
 Acceso a la propiedad de 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
Diario Perl, 263
 Persistencia, 39, 45
 Petzold, Carlos, 265
 Lucio, Rob, 99
 Piloto
 desembarque, manipulación, etc., 217
 que comieron pescado, 34
 Texto sin formato, 73
 contraformato binario, 73
 inconvenientes, 74
 documentos ejecutables, 251
 apalancamiento, 75
 obsolescencia, 74
 y pruebas más sencillas, 76
 Unix, 76
 Polimorfismo, 111
 Post-it, 53, 55
 Powerbuilder, 55
La práctica de la programación, 99
 Programador pragmático
 características, xviii
 dirección de correo electrónico, xiii
 sitio web, XXIII
 Condición previa y posterior, 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
expensas de, 237

programación de ventanas, 265

Proyecto

glosario, 210
“cabezas”, 228
saboteador, 244
horarios, 68
ver también Automatización;
equipo, proyecto

Bibliotecario de proyectos, 33,

226 Prototipos, 53, 216

arquitectura, 55
código desecharable, 56
tipo 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

controlar, 9
requisitos, 11
equipos, 225

Credo del trabajador de la cantera,

xx Algoritmo Quicksort, 180

R

Proceso racional unificado, 227n

Raymond, Eric S., 273

rcs, ver Sistema de control de revisión

Datos del mundo real, 243

Refactorización, 5, 185

automático, 187
y diseño, 186
prueba, 187
limitaciones de tiempo, 185

Navegador de refactorización,
187, 268 Refinamiento, excesivo,
11 Regresión, 76, 197, 232, 242

Relación

tiene-a, 304
tipo de, 111, 304

Liberaciones, y SCCS, 87

Invocación de método remoto (RMI), 128

manejo de excepciones, 39

Llamada a procedimiento remoto (RPC), 29, 39

Repositorio, 87

Requisito, 11, 202

problema empresarial, 203
cambiando, 26
arrastrarse, 209
DBC, 110
distribución, 211
documentando, 204
en 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

investigando, 15

Equilibrio de recursos, 129

Excepciones de C++, 132
comprobación, 135
código acoplado, 130
estructuras de datos dinámicas,
135 encapsulación en clase, 132
Java, 134
asignación de nodos, 131

Conjunto de respuestas, 141, 242

Responsabilidad, 2, 250, 258

Reutilización, 33, 36

Reversibilidad, 44

arquitectura flexible, 46 Sistema de
control de revisión (RCS), 250,
271

Gestión de riesgos, 13

ortogonalidad, 36

RMI, ver Invocación de método remoto

Rock-n-roll, 47

RPC, ver Llamada a procedimiento remoto

Patinaje elástico, 3, 95

Motor de reglas, 169

S

Saboteador, 244

samba, 272

programas de muestra, *ver*Código de ejemplo Sather, 114, 268

SCCS, *ver*Sistema de control de código fuente

Calendario, proyecto, 68

Schrödinger, Erwin (y su gato), 47

Alcance, requisito, 209

raspado de pantalla, 61

Lenguaje de secuencias de comandos, 55,

145 Hash seguro, 74

sed, 99

Sedgewick, Roberto, 183

Componentes autónomos, *ver* ortogonalidad; Cohesión

Semántica invariante, 116, 135

enviar correopograma, 60

Diagrama de secuencia, 158

Código del servidor, 196

Servicios, diseño usando, 154

Shell, comando, 77

*ver también*Consola de comandos

"Código tímido", 40

Efecto secundario, 124

SIGPLAN, 263

Bucle simple, 180

objeto único, 41

Punto oblicuo, 265

PequeñoEiffel, 267

Smalltalk, 46, 186, 187, 268, 272

Software

 tecnologías de desarrollo, 221

 calidad, 9

 requisitos, 11

bus software, 159

"Construcción de software", 184 *Revista de desarrollo de software*, 263 CI de

software, 189n

Descomposición del software", 4

Solaris, 76

Código fuente

 gato comiendo, 3

 documentación, *ver*Descarga de

 comentarios, *ver*Ejemplo de

 duplicación de código en, 29

 generar, 103

 reseñas, *ver*Revisiones de código

Sistema de control de código fuente (SCCS), 86

Égida, 246

construye usando, 88

CVS, 271

árbol de desarrollo, 87

texto plano y, 76

RCS, 250, 271

depósito, 87

 herramientas, 271

Especialización, 221

Especificación, 58

 implementación, 219

 lenguaje, 62

 como manta de seguridad, 219

 escritura, 218

Células espía, 138

Chirrido, 268

Minilenguaje autónomo, 62 "Fatiga de

puesta en marcha", 7

Comenzando un proyecto

 resolución de problemas, 212

 creación de prototipos, 216

 especificaciones, 217

*ver también*Requisito

Stevens, W. Richard, 264

Sopa de piedra, 7

contraventanas rotas, 9 Credo

del cortador de piedra, xx Analizador

de cadenas, 155

efecto Stroop, 249

strikrutina, 155

*Tutoriales estructurados, *ver*Código*

reseñas

Hoja de estilo, 20, 254 Estilo,

comunicación, 20 Subclase,

112

Algoritmo sublineal, 177

*Proveedor, *ver*Vendedor*

Proyectos orientados a objetos sobre vivientes: A

Guía del gerente, 264

SWIG, 55, 270

Barra de sincronización, 151

Resaltado de sintaxis, 84

Datos sintéticos, 243

T

Espacios T, 166, 269

TAM, *ver*Prueba de mecanismo de

acceso 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
 constructores de herramientas, 229
- Redactor técnico**, 252
- Plantilla**, caso de uso, 205
- Acoplamiento temporal**, 150
- Mecanismo de acceso de prueba (TAM), 189
- Arnés de prueba, 194
- Pruebas**
 automatizado, 238
 de 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
 función del texto sin formato, 76 refactorización, 187
 regresión, 76, 197, 232, 242
 agotamiento de recursos, 240
 saboteador, 244
 datos de prueba, 100, 243
 usabilidad, 241
 validación y verificación, 239; *ver también* Pruebas unitarias Lenguaje de manipulación de texto, 99 Lenguaje de programación TOM, 268 Toolkits, 39
- Herramientas, adaptables, 205
 Código rastreador, 49
 ventajas de, 50
 y prototipos, 51
 rastreo, 94
ver también Inicio sesión
- papel comercial, 263
 compensaciones, 249
Transacciones, EJB, 39
 Widget de árbol, 161
 sistema troff, 103
 espacio tupla, 167
-
- ## tu
- UML, *ver* Lenguaje de modelado unificado (UML)
 DESHACER llave, 86
- Lenguaje unificado de modelado UML**
 diagrama de actividad, 150
 diagrama de secuencia, 158
 diagrama de casos de uso, 208
 Principio de acceso uniforme, 31n
 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
Programación de red Unix, 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**
 corrupto, 95
 mundial, 130, 154
 nombre, 249
 Vendedor
 bibliotecas, 39
 reducir la dependencia de, 36, 39, 46

viredactor, 266

Vista

- depuración, 164
- documentos ejecutables, 251
- Vista de árbol de Java, 161
- modelo-vista-controlador, 160, 162
- red modelo-visor, 162

empujeredactor, 266

visual básico, 55

VisualC++, 198

Visual SourceSafe, 271

Obras visuales, 268

W

tutoriales, *ver* 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

Servicios del sistema Win32, 265

Ventanas, 46

- "a"mando, 231

- libros, 265

- Cygwin, 80

- metadatos, 145

- bloc de notas, 84

Utilidades de Unix, 80, 81

UWIN, 81

WinZip, 272

SABIDURÍAacróstico, 20

Mago, 198

Procesador de textos, 252, 254

Flujo de trabajo, 150

- sistema de pizarra, 169

- impulsado por el contenido, 234

Envoltura, 132, 133, 135, 141

Escriptura, 18

ver también Documentación

www.pragmaticprogrammer.com, XXIII

WYSIWYG, *ver* Lo que ves es lo que

Usted obtiene

X

editor XEmacs, 266

Parque Xerox, 39

XSL, *ver* Lenguaje de estilo extensible

xUnit, 194, 269

Y

yacc, 59

Yourdon, Edward, 10, 35

Problema Y2K, 32, 208

Z

caparazón Z, 272



InformIT es una marca de Pearson y la presencia en línea de los principales editores de tecnología del mundo. Es su fuente de conocimiento y contenido confiable y calificado, y brinda acceso a las mejores marcas, autores y colaboradores de la comunidad tecnológica.

▲ Addison-Wesley **Cisco Press** EXAM/**CRAM** **IBM**
Press. **QUE** **PRENTICE**
HALL **SAMS** | **Safari**
Books Online

a

¿Está buscando un libro, un libro electrónico o un video de capacitación sobre una nueva tecnología?

¿Está buscando información y tutoriales oportunos y relevantes? ¿Busca opiniones de expertos, consejos y sugerencias? **InformIT tiene la solución.**

- Obtenga información sobre nuevos lanzamientos y promociones especiales suscribiéndose a una amplia variedad de boletines.
Visitar informit.com/boletines.
- Acceda a podcasts GRATUITOS de expertos en informit.com/podcasts.
- Lea los últimos artículos de autor y capítulos de muestra en informit.com/articulos.
- Acceda a miles de libros y videos en la biblioteca digital Safari Books Online en safari.informit.com.
- Obtenga consejos de blogs de expertos en informit.com/blogs.

Visitar informit.com/aprender para descubrir todas las formas en que puede acceder al contenido tecnológico más actual.

¿Eres parte del

¿Multitud?

¡Conéctese con los autores y editores de Pearson a través de fuentes RSS, Facebook, Twitter, YouTube y más! Visitar informit.com/socialconnect.





Addison
Wesley

REGISTER



ESTE PRODUCTO

informit.com/registrar

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/registrar para iniciar sesión o crear una cuenta. Luego se le pedirá que ingrese el ISBN de 10 o 13 dígitos que aparece en la contraportada de su producto.

El registro de sus productos puede desbloquear los siguientes beneficios:

- Acceso a contenido complementario, incluidos capítulos de bonificación, 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 DE APRENDIZAJE TECNOLÓGICO DE CONFIANZA

INFORMIT ES EL HOGAR DE LAS IMPRESAS EDITORIALES LÍDERES EN 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 Safari Books Online, InformIT tiene una solución para usted.

Addison Wesley | Prensa de Cisco | Examen intensivo IBM Press | What | Prentice Hall | sams

El programador pragmático

Esta tarjeta resume los consejos y las listas de verificación que se encuentran en *El programador pragmático*.

Para más información sobre TÉLPAGSRAGMATICOPAGSROGRAMMERS LLC, código fuente para los ejemplos, punteros actualizados a recursos web y una bibliografía en línea, visítenos www.pragmaticprogrammer.com.

Guia de referencia rápida

TIPS1A22

1. Preocúpate por tu oficio.xix
¿Por qué pasarse la vida desarrollando software a menos que le importe hacerlo bien?
2. ¡Piensa! Acerca de su trabajo.xix Apague el piloto automático y tome el control. Criticar y evaluar constantemente su trabajo.
3. Proporcione opciones, no ponga excusas tontas.3 En lugar de excusas, ofrezca opciones. No digas que no se puede hacer; explica que *pueden* hacerse
4. No vivas con las ventanas rotas.5 Repare los malos diseños, las decisiones equivocadas y el código deficiente cuando los vea.
5. Sea un catalizador para el cambio.8 No se puede forzar el cambio en las personas. En su lugar, muéstrelas cómo podría ser el futuro y ayúdelos a participar en su creación.
6. Recuerde el panorama general.8 No te enfrasques tanto en los detalles que te olvides de comprobar lo que sucede a tu alrededor.
7. Haga de la calidad una cuestión de requisitos.11 Involucre a sus usuarios en la determinación de los requisitos reales de calidad del proyecto.
8. Invierta regularmente en su cartera de conocimientos.14 Haga del aprendizaje un hábito.
9. Analice críticamente lo que lee y escucha.16 No se deje influir por los vendedores, la exageración de los medios o el dogma. Analiza la información en términos tuyos y de tu proyecto.
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.
11. *SECO—Dno Repetir Y nosotros mismos*.27 Cada pieza de conocimiento debe tener una representación única, inequívoca y autorizada dentro de un sistema.
12. Facilite su reutilización.33 Si es fácil de reutilizar, la gente lo hará. Cree un entorno que admite la reutilización.
13. Eliminar efectos entre cosas no relacionadas.35 Diseñar componentes que sean autónomos, independientes y que tengan un propósito único y bien definido.
14. No hay decisiones finales.46 Ninguna decisión está grabada en piedra. En su lugar, considere cada uno como escrito en la arena de la playa y planifique el cambio.
15. Usa balas trazadoras para encontrar el objetivo.49 Las balas trazadoras le permiten acercarse a su objetivo probando cosas y viendo qué tan cerca aterrizan.
16. Prototipar para aprender.54 La creación de prototipos es una experiencia de aprendizaje. Su valor no radica en el código que produce, sino en las lecciones que aprende.
17. Programa cerca del dominio del problema.58 Diseña y programa en el idioma de tu usuario.
18. Estimar para evitar sorpresas.64 Estima antes de empezar. Detectará problemas potenciales desde el principio.
19. Iterar el Horario con el Código.69 Use la experiencia que gana a medida que implementa para refinar las escalas de tiempo del proyecto.
20. Mantenga el conocimiento en texto sin formato.74 El texto sin formato no quedará obsoleto. Ayuda a aprovechar su trabajo y simplifica la depuración y las pruebas.
21. Usa el poder de los proyectiles de comando.80 Utilice el shell cuando las interfaces gráficas de usuario no sean suficientes.
22. Use un pozo de un solo editor.82 El editor debe ser una extensión de tu mano; asegúrese de que su editor sea configurable, extensible y programable.

23. Utilice siempre el control de código fuente.....88 El control del código fuente es una máquina del tiempo para su trabajo: usted <i>pueden</i> regresa.	35. Termina lo que empiezas.....129 Siempre que sea posible, la rutina o el objeto que asigna un recurso debe ser responsable de desasignarlo.
24. Solucione el problema, no la culpa.....91 En realidad, no importa si el error es culpa tuya o de otra persona; sigue siendo tu problema y aún debe corregirse.	36. Minimice el acoplamiento entre módulos.....140 Evite el acoplamiento escribiendo código "tímidos" y aplicando la Ley de Deméter.
25. No entre en pánico al depurar.....91 Tome una respiración profunda y PENSAR! sobre lo que podría estar causando el error.	37. Configure, no integre.....144 Implementar opciones tecnológicas para una aplicación como opciones de configuración, no mediante integración o ingeniería.
26. "seleccionar" no está roto.....96 Es raro encontrar un error en el sistema operativo o el compilador, o incluso en un producto o biblioteca de terceros. Lo más probable es que el error esté en la aplicación.	38. Ponga abstracciones en código, detalles en metadatos.145 Programe para el caso general y coloque los detalles fuera del código base compilado.
27. No lo asuma, demuéstrelo97 Demuestre sus suposiciones en el entorno real, con datos reales y condiciones de contorno.	39. Analice el flujo de trabajo para mejorar la concurrencia...151 Explorar la concurrencia en el flujo de trabajo de su usuario.
28. Aprenda un lenguaje de manipulación de texto.....100 Pasas gran parte de cada día trabajando con texto. ¿Por qué no hacer que la computadora haga algo por tí?	40. Diseño utilizando servicios.....154 Diseño en términos <i>deservicios</i> —objetos independientes y concurrentes detrás de interfaces bien definidas y consistentes.
29. Escribir código que escribe código.....103 Los generadores de código aumentan su productividad y ayudan a evitar la duplicación.	41. Diseñe siempre para la concurrencia.....156 Permita la concurrencia y diseñará interfaces más limpias con menos suposiciones.
30. No se puede escribir el software perfecto.....107 El software no puede ser perfecto. Proteja su código y a los usuarios de los errores inevitables.	42. Vistas separadas de modelos.....161 Obtenga flexibilidad a bajo costo diseñando su aplicación en términos de modelos y vistas.
31. Diseño con Contratos.....111 Use contratos para documentar y verificar que el código no hace más ni menos de lo que dice hacer.	43. Use pizarras para coordinar el flujo de trabajo.....169 Utilice pizarras para coordinar hechos y agentes dispares, manteniendo la independencia y el aislamiento entre los participantes.
32. Accidente temprano.....120 Un programa muerto normalmente hace mucho menos daño que uno paralizado.	44. No programe por coincidencia.....175 Confíe sólo en cosas confiables. Tenga cuidado con la complejidad accidental y no confunda una feliz coincidencia con un plan con propósito.
33. Use afirmaciones para prevenir lo imposible.....122 Las afirmaciones validan sus suposiciones. Úselos para proteger su código de un mundo incierto.	45. Estime el orden de sus algoritmos.....181 Obtener una idea de cuánto tiempo es probable que tomen las cosas <i>antes de</i> escribes código.
34. Utilice excepciones para problemas excepcionales.....127 Las excepciones pueden sufrir todos los problemas de legibilidad y mantenibilidad del código spaghetti clásico. Reserve excepciones para cosas excepcionales.	46. Pruebe sus estimaciones.....182 El análisis matemático de los algoritmos no te dice todo. Intento cronometrar su código en su entorno de destino.

47. Refactorice temprano, refactorice a menudo.186 Del mismo modo que podría desyerbar y reorganizar un jardín, reescriba, vuelva a trabajar y rediseñe el código cuando lo necesite. Arreglar la raíz del problema.	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.
48. Diseño para probar.192 Empiece a pensar en probar antes de escribir una línea de código.	60. Organice equipos en torno a la funcionalidad.227 No separe a los diseñadores de los codificadores, a los evaluadores de los modeladores de datos. Cree equipos de la misma manera que crea código.
49. Pruebe su software, o lo harán sus usuarios.197 Prueba sin piedad. No haga que sus usuarios encuentren errores por usted.	61. No use procedimientos manuales.231 Un script de shell o un archivo por lotes ejecutará las mismas instrucciones, en el mismo orden, una y otra vez.
50. No use el código de asistente que no entiende. 199 Wizards puede generar montones de código. Asegúrate de entender <i>todos</i> antes de incorporarlo a su proyecto.	62. Prueba temprano. Prueba a menudo. Prueba automáticamente.237 Las pruebas que se ejecutan con cada compilación son mucho más efectivas que los planes de prueba que se encuentran en un estante.
51. No reúna requisitos: excave en busca de ellos.202 Los requisitos rara vez se encuentran en la superficie. Están enterrados profundamente bajo capas de suposiciones, conceptos erróneos y política.	63. La codificación no se realiza hasta que se ejecutan todas las pruebas.238 'Nuff dijo.
52. Trabaje con un usuario para pensar como un usuario.204 Es la mejor manera de obtener información sobre cómo funcionará el sistema <i>De Verdad</i> ser usado.	64. Usa saboteadores para probar tus pruebas.244 Introduzca errores a propósito en una copia separada de la fuente para verificar que las pruebas los detecten.
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.	65. Probar la cobertura del estado, no la cobertura del código.245 Identificar y probar estados significativos del programa. Solo probar líneas de código no es suficiente.
54. Utilice un glosario de proyectos.210 Cree y mantenga una fuente única de todos los términos y vocabulario específicos para un proyecto.	66. Encuentra errores una vez.247 Una vez que un probador humano encuentra un error, debe ser el <i>ultimo</i> vez que un probador humano encuentra ese error. Las pruebas automáticas deberían verificarlo a partir de ese momento.
55. No piense fuera de la caja— <i>Encontrarla</i> caja.213 Cuando te enfrentes a un problema imposible, identifica el <i>real</i> estricciones Pregúntese: "¿Tiene que hacerse de esta manera? ¿Tiene que hacerse en absoluto?	67. El inglés es solo un lenguaje de programación.248 Escriba documentos como escribiría código: respete las <i>SECO</i> principio, uso de metadatos, MVC, generación automática, etc.
56. Comience cuando esté listo.215 Has estado acumulando experiencia toda tu vida. No ignore las dudas inquietantes.	68. Incorpore la documentación, no la atornille.248 Es menos probable que la documentación creada por separado del código sea correcta y esté actualizada.
57. Algunas cosas se hacen mejor que se describen.218 No caiga en la espiral de las especificaciones: en algún momento tendrá que empezar a programar.	69. Supere suavemente las expectativas de sus usuarios.255 Comprenda las expectativas de sus usuarios y luego entregue un poco más.
58. No seas esclavo de los métodos formales.220 No adopte ciegamente ninguna técnica sin ponerla en el contexto de sus prácticas y capacidades de desarrollo.	70. Firme su trabajo.258 Los artesanos de una época anterior estaban orgullosos de firmar su trabajo. Tú también deberías estarlo.

- Idiomas para aprender.....página 17
 ¿Cansado de C, C++ y Java? Pruebe CLOS, Dylan, Eiffel, Objective C, Prolog, Smalltalk o TOM. Cada uno de estos lenguajes tiene diferentes capacidades y un "sabor" diferente. Pruebe un pequeño proyecto en casa usando uno o más de ellos.

- LOSSABIDURÍAAcróstico.....página 20

 W_oQué quieres que aprendan?
 ¿Cuál es su interés en lo que tienes que decir?
 Cóm_oSon sofisticados?
 Cuánto_dDetalles quieren? a quien
 quiereso_tTienes la información?
 Como puedesmetro_rLos motivas a que te escuchen?

- Cómo mantener la ortogonalidad.....página 34

- Diseñe componentes independientes y bien definidos.
- Mantenga su 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

- Cuestiones arquitectónicas.....página 55

- ¿Están bien definidas las responsabilidades?
- ¿Están bien definidas las colaboraciones? ¿Se minimiza el acoplamiento?
- ¿Puedes identificar posibles duplicaciones?
- ¿Son aceptables las definiciones de interfaz y las restricciones?
- ¿Pueden los módulos acceder a los datos necesarios? *cuando* necesario?

- Lista de comprobación de depuración.....página 98

- ¿El problema que se informa es un resultado directo del error subyacente o simplemente un síntoma?
- es el error *De Verdad* en el compilador? ¿Está en el sistema operativo? ¿O está en tu código?
- Si le explicaras en detalle este problema a un compañero de trabajo, ¿qué le dirías?
- Si el código sospechoso pasa sus pruebas unitarias, ¿son las pruebas lo suficientemente completas? ¿Qué sucede si ejecuta la prueba unitaria con este dato?
- ¿Existen las condiciones que causaron este error 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 pertenecen a:

- Si mismo
- Cualquier parámetro pasado
- en Objetos que crea
- Objetos componentes

- Cómo programar deliberadamente.....página 172

- Mantente al tanto de lo que estás haciendo. No codifiques con los ojos vendados.
- Partir de un plan. Confía solo en cosas confiables. Documenta sus suposiciones. Prueba las suposiciones y el código. Prioriza tu esfuerzo.
- No seas esclavo de la historia.

- Cuándo refactorizar.....página 185

- Usted descubre una violación del *SECO* principio.
- Encuentras cosas que podrían ser más ortogonales. Tu conocimiento mejora.
- Los requisitos evolucionan.
- Necesitas mejorar el rendimiento.

- Cortando el Nudo Gordiano.....página 212

Al resolver *imposible* problemas, pregúntese:

- hay una manera mas fácil?
- ¿Estoy resolviendo el problema correcto? *Por qué* es esto un problema?
- ¿Qué lo hace difícil? ¿Tengo que hacerlo de esta manera? ¿Tiene que hacerse en absoluto?

- Aspectos de las Pruebas.....página 237

- Pruebas unitarias
- Pruebas de integración
- Validación y verificación
- Agotamiento de recursos, errores y recuperación
- Pruebas de rendimiento
- Pruebas de usabilidad
- Probando las pruebas mismas