

20th ANNIVERSARY EDITION



The Pragmatic Programmer

your journey to mastery

DAVID THOMAS
ANDREW HUNT



el pragmático Programador

TU VIAJE A LA MAESTRÍ A

POR DAVE THOMAS, ANDY HUNT

Versión: P1.0 (13 de septiembre de 2019)

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 el editor estaba al tanto de un reclamo de marca registrada, las designaciones se han impreso con letras mayúsculas iniciales o en mayúsculas.

"The Pragmatic Programmer" y el dispositivo g de enlace son marcas comerciales de The Pragmatic Programmers, LLC.

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.

Para obtener información sobre la compra de este título en grandes cantidades o para oportunidades de ventas especiales (que pueden incluir versiones electrónicas, diseños de portada personalizados y contenido específico para su negocio, objetivos de capacitación, enfoque de marketing o intereses de marca), comuníquese con nuestro departamento de ventas corporativas. en corpsales@pearsoned.com o (800) 382-3419.

Para consultas de ventas gubernamentales, comuníquese con Governmentsales@pearsoned.com. Si tiene preguntas sobre las ventas fuera de los EE. UU., comuníquese con intlcs@pearson.com. Visítanos en la web:

informit.com/aw

Número de control de la Biblioteca del Congreso: 2019944178

Copyright © 2020 Pearson Educación, Inc.

Imágenes de portada: Mihalec/Shutterstock,
Stockish/Shutterstock

Reservados todos los derechos. Esta publicación está protegida por derechos de autor y se debe obtener el permiso del editor antes de cualquier reproducción, almacenamiento en un sistema de recuperación o transmisión prohibida en cualquier forma o por cualquier medio, ya sea electrónico, mecánico, fotocopiado, grabación o similar. Para obtener información sobre permisos, formularios de solicitud y los contactos apropiados dentro del Departamento de Permisos y Derechos Globales de Pearson Education, visite www.pearsoned.com/permissions.

ISBN-13: 978-0-13-595705-9

ISBN-10: 0-13-595705-2

Para Julieta y Ellie,
Zachary y Elizabeth,
Enrique y Estuardo

Tabla de contenido

1. Prefacio

2. Prefacio a la segunda edición

1. Cómo está organizado el libro

2. ¿Qué hay en un nombre?

3. Código fuente y otros recursos

4. Envíenos sus comentarios

5. Agradecimientos de la segunda edición

3. Del Prefacio a la Primera Edición

1. ¿Quién debería leer este libro?

2. ¿Qué hace a un programador pragmático?

3. Pragmáticos individuales, grandes equipos

4. Es un proceso continuo

4. 1. Una filosofía pragmática

1. Tema 1. Es tu vida

2. Tema 2. El gato se comió mi código fuente

3. Tema 3. Entropía del Software

4. Tema 4. Sopa de piedra y ranas hervidas

5. Tema 5. Software suficientemente bueno

6. Tema 6. Su carpeta de conocimientos

7. Tema 7. ¡Comunica!

5. 2. Un enfoque pragmático

1. Tema 8. La esencia del buen diseño

2. Tema 9. DRY—Los males de la duplicación

3. Tema 10. Ortogonalidad

4. Tema 11. Reversibilidad

5. Tema 12. Viñetas trazadoras

6. Tema 13. Prototipos y Post-it Notes

7. Tema 14. Idiomas de dominio

8. Tema 15. Estimación

6. 3. Las herramientas básicas

1. Tema 16. El poder del texto sin formato

2. Tema 17. Juegos de conchas

3. Tema 18. Edición de potencia

4. Tema 19. Control de versiones

5. Tema 20. Depuración

6. Tema 21. Manipulación de textos

7. Tema 22. Agendas de Ingeniería

7. 4. Paranoia pragmática

1. Tema 23. Diseño por Contrato

2. Tema 24. Los programas muertos no cuentan mentiras

3. Tema 25. Programación Asertiva

4. Tema 26. Cómo equilibrar los recursos

5. Tema 27. No superes tus faros

8. 5. Doblar o romper

1. Tema 28. Desacoplamiento

2. Tema 29. Malabares con el mundo real

3. Tema 30. Transformación de la programación

4. Tema 31. Impuesto sobre Sucesiones

5. Tema 32. Configuración

9. 6. Conurrencia

1. Tema 33. Rompiendo el Acoplamiento Temporal

2. Tema 34. El estado compartido es un estado incorrecto

3. Tema 35. Actores y Procesos

4. Tema 36. Pizarras

10. 7. Mientras codificas

1. Tema 37. Escucha tu cerebro de lagarto

2. Tema 38. Programación por Coincidencia

3. Tema 39. Algoritmo de Velocidad

4. Tema 40. Refactorización

5. Tema 41. Test to Code

6. Tema 42. Pruebas basadas en propiedades

7. Tema 43. Manténgase seguro afuera

8. Tema 44. Nombrar cosas

11. 8. Antes del Proyecto

1. Tema 45. El pozo de requisitos
2. Tema 46. Resolución de acertijos imposibles
3. Tema 47. Trabajando juntos
4. Tema 48. La esencia de la agilidad

12. 9. Proyectos Pragmáticos

1. Tema 49. Equipos Pragmáticos
2. Tema 50. Los cocos no lo cortan
3. Tema 51. Kit de inicio pragmático
4. Tema 52. Deleite a sus usuarios
5. Tema 53. Orgullo y prejuicio

13. 10. Posfacio

14. A1. Bibliografía

15. A2. Posibles respuestas a los ejercicios

Elogios por la segunda edición de El programador pragmático

Algunos dicen que con El programador pragmático, Andy y Dave capturaron un rayo en una botella; que es poco probable que alguien escriba pronto un libro que pueda mover a toda una industria como lo hizo. A veces, sin embargo, los rayos caen dos veces, y este libro es una prueba. El contenido actualizado garantiza que permanecerá en la parte superior de las listas de "mejores libros sobre desarrollo de software" durante otros 20 años, justo donde pertenece.

—
VM (Vicky) Brasseur

Director de estrategia de código abierto, Juniper Networks

Si desea que su software sea fácil de modernizar y mantener, mantenga cerca una copia de The Pragmatic Programmer . Está repleto de consejos prácticos, tanto técnicos como profesionales, que le serán de gran utilidad a usted y a sus proyectos en los años venideros.

—
Andrea Goulet

CEO, Corgibytes; Fundador, LegacyCode.Rocks

El programador pragmático es el único libro que puedo señalar que desplazó por completo la trayectoria existente de mi carrera en software y me indicó la dirección del éxito. Leerlo abrió mi mente a las posibilidades de ser un artesano, no solo un engranaje en una gran máquina. Uno de los libros más importantes de mi vida.

— obie fernandez

Autor, *The Rails Way*

Los lectores primerizos pueden esperar una inducción fascinante al mundo moderno de la práctica del software, un mundo en el que la primera edición desempeñó un papel importante en la configuración. Los lectores de la primera edición redescubrirán aquí los conocimientos y la sabiduría práctica que hicieron que el libro fuera tan importante en primer lugar, curado y actualizado por expertos, junto con muchas novedades.

— David A. Negro

Autor, *The Well Grounded Rubyist*

Tengo una copia antigua en papel del programador pragmático original en mi estantería. Ha sido leído y releído y hace mucho tiempo cambió todo acerca de cómo abordaba mi trabajo como programador. En la nueva edición, todo y nada ha cambiado: ahora lo leo en mi iPad y los ejemplos de código usan programación moderna

lenguajes, pero los conceptos, ideas y actitudes subyacentes son atemporales y de aplicación universal. Veinte años después, el libro sigue tan vigente como siempre. Me alegra saber que los desarrolladores actuales y futuros tendrán la misma oportunidad de aprender de las ideas profundas de Andy y Dave que tuve yo en su momento.

—
Sandy Mamoli

Entrenador ágil, autor de *How Self-Selection Lets People Excel*

Hace veinte años, la primera edición de *El programador pragmático* cambió por completo la trayectoria de mi carrera. Esta nueva edición podría hacer lo mismo por la tuya.

—
mike cohn

Autor de *Succeeding with Agile*,
Estimación y planificación ágiles, y
Historias de usuario aplicadas

Prefacio

Recuerdo cuando Dave y Andy tuitearon por primera vez sobre la nueva edición de este libro. Fue una gran noticia. Observé cómo la comunidad de codificación respondía con entusiasmo. Mi feed zumbaba con anticipación. Después de veinte años, El programador pragmático es tan relevante hoy como lo fue entonces.

Dice mucho que un libro con tanta historia haya tenido tal reacción. Tuve el privilegio de leer una copia inédita para escribir este prólogo y entendí por qué creó tanto revuelo. Si bien es un libro técnico, llamarlo así no le hace ningún favor. Los libros técnicos a menudo intimidan. Están llenos de grandes palabras, términos oscuros, ejemplos intrincados que, sin querer, te hacen sentir estúpido. Cuanto más experimentado es el autor, más fácil es olvidar lo que es aprender nuevos conceptos, ser un principiante.

A pesar de sus décadas de experiencia en programación, Dave y Andy han conquistado el difícil desafío de escribir con el mismo entusiasmo de las personas que acaban de aprender estas lecciones. Ellos no te hablan mal. No asumen que eres un experto. Ni siquiera asumen que has leído la primera edición. Te toman como eres: programadores que solo quieren ser mejores. Pasan las páginas de este libro ayudándolo a llegar allí, un paso procesable a la vez.

Para ser justos, ya habían hecho esto antes. El lanzamiento original estaba lleno de ejemplos tangibles, nuevas ideas y consejos prácticos para desarrollar sus músculos de codificación y desarrollar su cerebro de codificación que aún se aplican en la actualidad. Pero esta edición actualizada hace dos mejoras en el libro.

La primera es la obvia: elimina algunas de las referencias más antiguas, los ejemplos desactualizados y los reemplaza con contenido nuevo y moderno. No encontrará ejemplos de invariantes de bucle o máquinas de construcción. Dave y Andy han tomado su poderoso contenido y se han asegurado de que las lecciones aún lleguen, libres de las distracciones de los ejemplos antiguos. Quita el polvo de viejas ideas como DRY (no te repitas) y les da una nueva capa de pintura, haciéndolas realmente brillar.

Pero lo segundo es lo que hace que este lanzamiento sea realmente emocionante. Después de escribir la primera edición, tuvieron la oportunidad de reflexionar sobre lo que estaban tratando de decir, lo que querían que sus lectores se llevaran y cómo estaba siendo recibido. Recibieron comentarios sobre esas lecciones. Vieron lo que se atascó, lo que necesitaba refinarse, lo que se malinterpretó. En los veinte años que este libro se ha abierto camino a través de las manos y los corazones de los programadores de todo el mundo, Dave y Andy han estudiado esta respuesta y formulado nuevas ideas, nuevos conceptos.

Aprendieron la importancia de la agencia y reconocieron que los desarrolladores tienen posiblemente más agencia que la mayoría de los otros profesionales. Comienzan este libro con un mensaje simple pero profundo: "es tu vida". Nos recuerda nuestro propio poder en nuestra base de código, en nuestros trabajos, en nuestras carreras. Establece el tono para todo lo demás en el libro, que es más que otro libro técnico lleno de ejemplos de código.

Lo que realmente lo hace destacar entre los estantes de libros técnicos es que comprende lo que significa ser un programador.

La programación consiste en tratar de hacer que el futuro sea menos doloroso. Se trata de facilitar las cosas a nuestros compañeros. Se trata de hacer las cosas mal y ser capaz de recuperarse. Se trata de formar buenos hábitos. Se trata de comprender su conjunto de herramientas. La codificación es solo parte del mundo de ser un programador, y este libro explora ese mundo.

Paso mucho tiempo pensando en el viaje de codificación. No crecí codificando; No lo estudié en la universidad. No pasé mi adolescencia jugando con la tecnología. Entré en el mundo de la codificación cuando tenía veintitantes años y tuve que aprender lo que significaba ser programador. Esta comunidad es muy diferente de otras de las que he sido parte. Hay una dedicación única al aprendizaje y la practicidad que es a la vez refrescante e intimidante.

Para mí, realmente se siente como entrar en un mundo nuevo. Una nueva ciudad, por lo menos. Tuve que conocer a los vecinos, elegir mi supermercado, encontrar las mejores cafeterías. Tomó un tiempo obtener la disposición del terreno, encontrar las rutas más eficientes, evitar las calles con el tráfico más pesado, saber cuándo era probable que llegara el tráfico. El clima es diferente, necesitaba un nuevo guardarropa.

Las primeras semanas, incluso meses, en una nueva ciudad pueden ser aterradoras. ¿No sería maravilloso tener un vecino amigable y conocedor que haya estado viviendo allí por un tiempo? ¿Quién puede darte un recorrido, mostrarte esas cafeterías? ¿Alguien que haya estado allí el tiempo suficiente para conocer la cultura, comprender el pulso de la ciudad, para que no solo se sienta como en casa, sino que también se convierta en un miembro contribuyente? Dave y Andy son esos vecinos.

Como un recién llegado relativo, es fácil sentirse abrumado no por la

acto de programar sino el proceso de convertirse en programador. Hay un cambio de mentalidad completo que debe suceder: un cambio en los hábitos, comportamientos y expectativas. El proceso de convertirse en un mejor programador no solo sucede porque sabes codificar; debe ser enfrentado con intención y práctica deliberada. Este libro es una guía para convertirse en un mejor programador de manera eficiente.

Pero no se equivoque, no le dice cómo debe ser la programación. No es filosófico o crítico de esa manera. Le dice, simple y llanamente, qué es un programador pragmático: cómo opera y cómo aborda el código. Te dejan a ti decidir si quieras ser uno. Si sientes que no es para ti, no te lo reprocharán. Pero si decides que lo es, son tus amigables vecinos, ahí para mostrarte el camino.

saron itbarek

Fundador y CEO de CodeNewbie

Anfitrión de Command Line Heroes

Prefacio a la segunda edición

En la década de 1990, trabajábamos con empresas cuyos proyectos tenían problemas. Nos encontramos diciendo las mismas cosas a cada uno: tal vez debería probar eso antes de enviarlo; ¿Por qué el código solo se basa en la máquina de Mary? ¿Por qué nadie preguntó a los usuarios?

Para ahorrar tiempo con nuevos clientes, empezamos a tomar notas. Y esas notas se convirtieron en *El programador pragmático*. Para nuestra sorpresa, el libro pareció tocar una fibra sensible y ha seguido siendo popular durante los últimos 20 años.

Pero 20 años son muchas vidas en términos de software. Tome un desarrollador de 1999 y colóquelo en un equipo hoy, y lucharán en este extraño mundo nuevo. Pero el mundo de la década de 1990 es igualmente extraño para el desarrollador de hoy. Las referencias del libro a cosas como CORBA, herramientas CASE y bucles indexados eran, en el mejor de los casos, pintorescas y probablemente confusas.

Al mismo tiempo, 20 años no han tenido ningún impacto en el sentido común. La tecnología puede haber cambiado, pero las personas no. Prácticas y enfoques que fueron una buena idea entonces siguen siendo una buena idea ahora. Esos aspectos del libro envejecieron bien.

Entonces, cuando llegó el momento de crear ^{el} esta Edición 20 Aniversario,

teníamos que tomar una decisión. Podríamos revisar y actualizar las tecnologías a las que hacemos referencia y llamarlo un día. O podríamos reexaminar los supuestos detrás de las prácticas que recomendamos a la luz de dos décadas adicionales de experiencia.

Al final, hicimos los dos.

Como resultado, este libro es una especie de Barco de Teseo. [1] Aproximadamente un tercio de los temas del libro son nuevos. Del resto, la mayoría han sido reescritos, ya sea parcial o totalmente. Nuestra intención era hacer las cosas más claras, más relevantes y, con suerte, algo atemporales.

Tomamos algunas decisiones difíciles. Quitamos el apéndice de Recursos , porque sería imposible mantenerse actualizado y porque es más fácil buscar lo que desea. Reorganizamos y reescribimos los temas relacionados con la concurrencia, dada la abundancia actual de hardware paralelo y la escasez de buenas formas de manejarlo. Agregamos contenido para reflejar actitudes y entornos cambiantes, desde el movimiento ágil que ayudamos a lanzar, hasta la creciente aceptación de lenguajes de programación funcional y la creciente necesidad de considerar la privacidad y la seguridad.

Curiosamente, sin embargo, hubo mucho menos debate entre nosotros sobre el contenido de esta edición que cuando escribimos la primera. Ambos sentimos que las cosas que eran importantes eran más fáciles de identificar.

De todos modos, este libro es el resultado. Por favor disfrutalo. Tal vez adoptar algunas prácticas nuevas. Tal vez decida que algunas de las cosas que sugerimos están mal. Involúcrate en tu oficio. Danos su opinión.

Pero, lo más importante, recuerda hacerlo divertido.

Cómo está organizado el libro

Este libro está escrito como una colección de temas breves. Cada tema es independiente y aborda un tema en particular. Encontrará numerosas referencias cruzadas, que ayudan a poner cada tema en contexto.

Siéntase libre de leer los temas en cualquier orden; este no es un libro que necesita leer de principio a fin.

Ocasionalmente, encontrará un cuadro con la etiqueta Sugerencia nn (como Sugerencia 1, Cuide su oficio). 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.

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 un apéndice, pero muy pocos tienen una única solución correcta . Los desafíos pueden formar la base de discusiones grupales o trabajos de ensayo en cursos de programación avanzada.

También hay una breve bibliografía que enumera los libros y artículos a los que hacemos referencia explícita.

¿Lo que hay en un nombre?

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 se han pasado por alto, y otros, como la base de datos relacional y de objetos , son de uso tan común que agregar una definición sería aburrido. Si se encuentra con un término que no ha visto antes, no lo pase por alto. 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 a la próxima edición.

Dicho todo esto, decidimos vengarnos de los informáticos. A veces, hay palabras de jerga perfectamente buenas para conceptos, palabras que hemos decidido ignorar. ¿Por qué?

Porque la jerga existente normalmente se restringe a un dominio de problema particular, oa 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 la organización del equipo, 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 [2] sitio web.

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. Envíenos un correo electrónico a ppbook@pragprog.com.

Agradecimientos de la segunda edición

Hemos disfrutado literalmente de miles de conversaciones interesantes sobre programación durante los últimos 20 años, conociendo gente en conferencias, cursos y, a veces, incluso en el avión. Cada uno de ellos se ha sumado a nuestra comprensión del proceso de desarrollo y ha contribuido a las actualizaciones de esta edición. Gracias a todos (y seguid diciéndonos cuando nos equivocamos).

Gracias a los participantes en el proceso beta del libro. Sus preguntas y comentarios nos ayudaron a explicar mejor las cosas.

Antes de pasar a la versión beta, compartimos el libro con algunas personas para que hicieran comentarios. Gracias a VM (Vicky) Brasseur, Jeff Langr y Kim Shrier por sus comentarios detallados, ya José Valim y Nick Cuthbert por sus revisiones técnicas.

Gracias a Ron Jeffries por dejarnos usar el ejemplo de Sudoku.

Mucha gratitud a la gente de Pearson que accedió a dejarnos crear este libro a nuestra manera.

Un agradecimiento especial a la indispensable Janet Furlow, que domina todo lo que emprende y nos mantiene en línea.

Y, por último, un agradecimiento a todos los programadores pragmáticos que han mejorado la programación para todos durante los últimos veinte años. Brindemos por veinte más.

[1] Si, a lo largo de los años, todos los componentes de un barco se reemplazan cuando fallan, ¿el barco resultante es el mismo barco?

[2] <https://pragprog.com/titles/tpp20>

Copyright © 2020 Pearson Educación, Inc.

Del prefacio al primero

Edición

Este libro te ayudará a convertirte en un mejor programador.

Puede ser un desarrollador solitario, un miembro de un gran equipo de proyecto o un consultor que trabaja con muchos clientes a la vez. No importa; 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 proviene del latín *practicus*, “hábil en los negocios”, que a su vez se deriva del griego *πραγματικός*, que significa “apto para usar” .

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. Tú

hacer 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 la 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 como para permitirle 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 pionero/adaptador

Tiene instinto para las tecnologías y las técnicas, y le encanta probar cosas. Cuando se le da 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 esta computación cuántica de la que he oído hablar? ¿Cómo se implementan los enlaces simbólicos? Eres un fanático de los pequeños hechos, cada uno de los cuales puede afectar algunas decisiones dentro de unos años. ahora.

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 sensación de

cuán difíciles son las cosas y cuánto tiempo tomarán.

Comprender profundamente que un proceso debería ser difícil o que tomará un tiempo completarlo le brinda la resistencia para continuar.

Aprendiz de

todos los oficios. 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:

Consejo 1

Cuida tu oficio

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

Consejo 2

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

Si esto le parece un trabajo duro, entonces está exhibiendo la característica realista . Esto va a tomar parte de su

tiempo valioso—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. “El software es una disciplina de ingeniería”, dicen, “que falla si los miembros individuales del equipo toman decisiones por sí mismos”.

Estamos totalmente en desacuerdo.

Debería haber ingeniería en la construcción de software. 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 sustentó los proyectos: Nosotros, los que cortamos simples piedras, siempre debemos estar imaginando catedrales.

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 constructores de catedrales medievales a los ingenieros civiles de hoy, mientras que nuestra artesanía seguirá siendo respetada.

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 cepilla el rocío todas las mañanas, córtalas cada dos días y pásalas una vez por 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 dejar caer la palabra kaizen en las 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.

Capítulo 1

Una filosofía pragmática

Este libro es sobre ti.

No se equivoque, es su carrera y, lo que es más importante, el Tema 1, es su vida. Es tuyo. Está aquí porque sabe que puede convertirse en un mejor desarrollador y ayudar a otros a mejorar también. Puedes convertirte en un programador pragmático.

¿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, colocándolo en su contexto más amplio y buscando el 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 los programadores pragmáticos asumen la responsabilidad de todo lo que hacen, lo cual analizamos en el Tema 2, 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 el Tema 3, Software Entropy, le decimos cómo mantener sus proyectos impecables.

La mayoría de las personas encuentran difícil el cambio, a veces por buenas razones,

a veces debido a la simple inercia de siempre. En el Tema 4, Sopa de piedra y ranas hervidas, analizamos una estrategia para instigar el cambio y (en aras del equilibrio) presentamos la historia de advertencia 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 el Tema 5, 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 el Tema 6, 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. Tema 7, ¡Comuníquese! 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.



Tema 1

Es tu vida

no estoy en este mundo para
estar a la altura de
tus expectativas y no
estás en este mundo para estar
a la altura de las mías.

Bruce Lee

es tu vida Es tuyo. Tú lo ejecutas. Tú lo creas.

Muchos desarrolladores con los que
hablamos están frustrados. Sus
preocupaciones son variadas.
Algunos sienten que están estancados en su
trabajo, otros que la tecnología los ha superado.
La gente siente que no los
aprecian o los pagan mal, o que sus equipos
son tóxicos. Tal vez quieran mudarse a Asia

o Europa, o trabajar desde casa.

Y la respuesta que damos es siempre la misma.

"¿Por qué no puedes cambiarlo?"

El desarrollo de software debe aparecer cerca de la parte superior de cualquier lista
de carreras en las que tenga control. Nuestras habilidades están en demanda,
nuestro conocimiento cruza fronteras geográficas, podemos trabajar de
forma remota. Estamos bien pagados. Realmente podemos hacer casi cualquier
cosa que queramos.

Pero, por alguna razón, los desarrolladores parecen resistirse al cambio. Se
agachan y esperan que las cosas mejoren. Miran, pasivamente, cómo sus
habilidades se vuelven anticuadas y se quejan de que sus empresas no los
capacitan. Miran anuncios de lugares exóticos en el autobús, luego bajan
bajo la lluvia helada y

entrar en el trabajo.

Así que aquí está el consejo más importante del libro.

Consejo 3

tienes agencia

¿Tu ambiente de trabajo apesta? ¿Tu trabajo es aburrido? Intenta arreglarlo. Pero no lo intentes para siempre. Como dice Martin Fowler, “puedes cambiar tu organización o cambiar tu organización” .

[3]

Si la tecnología parece estar pasando de largo, tómese un tiempo (en su propio tiempo) para estudiar cosas nuevas que parezcan interesantes. Estás invirtiendo en ti mismo, por lo que hacerlo mientras estás fuera de horario es razonable.

¿Quieres trabajar de forma remota? ¿Has preguntado? Si dicen que no, entonces busca a alguien que diga que sí.

Esta industria le brinda un notable conjunto de oportunidades. Sea proactivo, y tómelos.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 4, Sopa de piedra y ranas hervidas
- Tema 6, Su carpeta de conocimientos



Tema 2

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

Una de las piedras angulares de la filosofía pragmática es la idea de asumir la responsabilidad de uno mismo y de sus acciones en términos de su avance profesional, su aprendizaje y educación, su proyecto y su trabajo diario. Los programadores pragmáticos se hacen cargo de su propia carrera y no tienen 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 reconocer nuestras deficiencias, nuestra ignorancia y nuestros errores.

CONFIANZA DEL EQUIPO

Sobre todo, su equipo debe poder confiar en usted y confiar en usted, y usted también debe sentirse cómodo confiando en cada uno de ellos.

La confianza en un equipo es absolutamente esencial para la creatividad y la colaboración según la literatura de investigación.⁴¹ En un entorno saludable basado en la confianza, puede decir lo que piensa, presentar sus ideas y confiar en los miembros de su equipo que pueden en

a su vez confiar en ti. Sin confianza, bueno...

Imagina un equipo de ninjas sigilosos de alta tecnología infiltrándose en la guarida malvada del villano. Después de meses de planificación y delicada ejecución, lo logró en el lugar. Ahora es su turno de configurar la cuadrícula de guía láser: "Lo siento, amigos, no tengo el láser. El gato estaba jugando con el punto rojo y lo dejé en casa".

Ese tipo de abuso de confianza podría ser difícil de reparar.

ASUME 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. Tiene derecho a no asumir la responsabilidad de una situación imposible, o en la que los riesgos son demasiado grandes, o las implicaciones éticas demasiado vagas. Tendrás que hacer la llamada en base a tus propios valores y juicio.

Cuando 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 usted proporcionar soluciones, no excusas

Si existiera el riesgo de que el proveedor no cumpliera con

usted, entonces debería haber tenido un plan de contingencia. Si su almacenamiento masivo se derrite, llevándose consigo 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.

Consejo 4

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 has considerado eso?" ¿Cómo responderás? Antes de ir y decirles las malas noticias, ¿hay algo más que pueda probar? A veces, solo sabes lo que van a decir, así que ahórrate el problema.

En lugar de excusas, ofrezca opciones. No digas que no se puede hacer; explicar qué se puede hacer para salvar la situación. ¿Hay que borrar el código? Dígaselo y explíquenle el valor de la refactorización (consulte el Tema 40, Refactorización).

¿Necesita dedicar tiempo a la creación de prototipos para determinar la mejor manera de proceder (consulte el Tema 13, Prototipos y notas Post-it)? ¿Necesita introducir mejores pruebas (consulte el Tema 41, Prueba de código y Pruebas continuas y despiadadas) o automatización para evitar que vuelva a suceder?

Quizás necesite recursos adicionales para completar esta tarea. O

¿Quizás necesites pasar más tiempo con los usuarios? O tal vez solo eres tú: ¿necesitas aprender alguna técnica o tecnología en mayor profundidad? ¿Ayudaría un libro o un curso? 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

- Tema 49, Equipos Pragmáticos

RETOS

- ¿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?
- Cuando te encuentres diciendo “No lo sé” , asegúrate de continuar con “... pero lo averiguaré” . Es una gran manera de admitir lo que no sabes, pero luego asumir la responsabilidad como un profesional.



Tema 3

Entropía del software

Si bien el desarrollo de software es inmune a casi todas las leyes físicas, el aumento inexorable de la entropía nos golpea con fuerza. La 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 el desorden aumenta en el software, lo llamamos "podredumbre del software".

Algunas personas podrían llamarlo con el término más optimista, "deuda técnica", con la noción implícita de que algún día la pagarán.

Probablemente no lo harán.

Sin embargo, cualquiera que sea el nombre, tanto la deuda como la podredumbre pueden propagarse sin control.

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 cascos podridos. ¿Por qué? Investigadores en el campo del crimen

y urban decay descubrió un fascinante mecanismo desencadenante, uno que muy rápidamente convierte un edificio limpio, intacto y habitado [5] en un abandonado y destrozado.

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 lapso de tiempo relativamente corto, el edificio se daña más allá del deseo del propietario de repararlo, y la sensación de abandono se vuelve realidad.

¿Por qué eso haría una diferencia? Los psicólogos han realizado estudios que muestran^[6] que la desesperanza puede ser contagiosa. Piense en el virus de la gripe de cerca. Ignorar una situación claramente rota refuerza las ideas de que tal vez nada se pueda arreglar, que a nadie le importa, que todo está condenado; todos los pensamientos negativos que pueden propagarse entre los miembros del equipo, creando una espiral viciosa.

Consejo 5

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 tápelo. Quizás pueda comentar el código ofensivo, o mostrar un mensaje "No implementado", o sustituir datos ficticios en su lugar. Tome alguna medida para evitar más daños y para 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 hablaremos de algunos de ellos en otra parte, pero el descuido acelera la 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 es así, entonces será mejor que planee conseguir un contenedor de basura o mudarse a otro vecindario. No dejes que la entropía gane.

PRIMERO, NO HACER DAÑO

Andy una vez tuvo un conocido que era obscenamente rico. Su casa estaba inmaculada, repleta de antigüedades de valor incalculable, objetos de arte, etc. Un día, un tapiz que colgaba demasiado cerca de una chimenea se incendió. El departamento de bomberos 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.

Ahora eso suena bastante extremo. Seguramente la primera prioridad del departamento de bomberos es apagar el fuego, al diablo con los daños colaterales. Pero claramente habían evaluado la situación, confiaban en su capacidad para manejar el fuego y tuvieron cuidado de no infligir daños innecesarios a la propiedad. Así debe ser con el software: no cause daños colaterales solo porque hay algún tipo de crisis. Una ventana rota es demasiado.

Una ventana rota: una pieza de código mal diseñada, una mala decisión de gestión con la que el equipo debe vivir por el resto.

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 se desmanteló y se volcó en cuestión de horas.

Del mismo modo, si se encuentra en un proyecto en el que el código es prístinamente hermoso (escrito de forma 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.), no querrás ser el primero en causar un lío e infligir daño adicional.

Solo dígase a sí mismo: "No hay ventanas rotas".

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 10, Ortogonalidad
- Tema 40, Refactorización
- Tema 44, Nombrar cosas

RETOS

- Ayude a fortalecer su equipo encuestando el vecindario de su proyecto. 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?



Tema 4

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.

Después 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 unas cuantas zanahorias...” Un aldeano salió corriendo y regresó al poco tiempo con una canasta 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” . De corrió 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 corría 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 tenían.

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. Calcule lo que razonablemente puede pedir. Desarróllalo bien. Una vez que lo tengas, muéstralos a la gente y deja que se maravillen. Luego diga "por supuesto, sería mejor si agregáramos..." Pretenda 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. Muéstrales un vistazo del futuro y los obtendrás.

para reunirse alrededor.^[7]

Consejo 6

Sea un catalizador para el cambio

EL LADO DE LOS PUEBLO

Por otro lado, la historia de la sopa de piedra también se trata de una comida suave y

engaño paulatino. 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.

Consejo 1 Recuerde el panorama general

Nunca hemos probado esto, honesto. Pero “ellos” dicen que si tomas una rana y la dejas caer en agua hirviendo, volverá a saltar. 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.

Tenga en cuenta que el problema de la rana es diferente del problema de las ventanas rotas discutido en el Tema 3, Entropía del software. 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 legendaria. 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

- Tema 1, es tu vida

- Tema 38, Programación por Coincidencia

RETOS

- Mientras revisaba un borrador de la primera edición, 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?
- Rápido, sin mirar, ¿cuántas luces hay en el techo encima de ti? ¿Cuántas salidas en la habitación? ¿Cuanta gente? ¿Hay algo fuera de contexto, algo que parezca que no pertenece? Este es un ejercicio de conciencia situacional, una técnica practicada por personas que van desde Boy y Girl Scouts hasta Navy SEAL. Adquiera el hábito de realmente mirar y notar su entorno. Luego haz lo mismo para tu proyecto.

desarrollar el hábito de observar conscientemente y notar los detalles más sutiles.



Tema 5

Software suficientemente bueno

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

Shakespeare, el rey Lear 1.4

Hay un viejo chiste sobre una empresa 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. Adjunta 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 IEEE Software: Cuando el software lo suficientemente bueno es lo mejor [You95], puede disciplinarse para escribir software que sea lo suficientemente bueno, lo suficientemente bueno para sus usuarios, para los mantenedores futuros, 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 descuidado o mal

código producido. Todos los sistemas deben cumplir con los requisitos de sus usuarios para tener éxito y cumplir con los estándares básicos de rendimiento, privacidad y seguridad. 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 para sus necesidades.

INVOLUCRE A SUS USUARIOS EN EL TRADE-OFF

Normalmente estás escribiendo software para otras personas. A menudo te acordarás de averiguar lo que quieren. Pero, ^[8] ¿alguna vez les preguntas qué tan bueno quieren que sea su software? A veces no habrá otra opción. Si está trabajando en marcapasos, un piloto automático o una biblioteca de bajo nivel que se difundirá ampliamente, los requisitos serán más estrictos y sus opciones más limitadas.

Sin embargo, si está trabajando en un producto 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 efectivo. 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.

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

Consejo 8

Haga de la calidad una cuestión de requisitos

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

Sorprendentemente, muchos usuarios preferirían usar software con algunas asperezas hoy que esperar un año para la versión brillante y lujosa (y de hecho, lo que necesitarán dentro de un año puede ser completamente diferente de todos modos). Muchos departamentos de TI con presupuestos ajustados estarían de acuerdo. El gran software de hoy suele ser preferible a la fantasía del software perfecto del mañana. Si brinda a sus usuarios algo con lo que jugar desde el principio, sus comentarios a menudo lo llevarán a una mejor solución eventual (consulte el Tema 12, Tracer Bullets).

SABER CUANDO 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 7, Mientras programas, analizaremos las filosofías para desarrollar código en un mundo imperfecto).

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 45, El pozo de requisitos

- Tema 46, Resolviendo acertijos imposibles

RETOS

- Mire las herramientas de software y los sistemas operativos que usa regularmente. ¿Puede encontrar alguna evidencia de que estas organizaciones y/o desarrolladores se sientan cómodos 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 de software monolítico estrechamente acoplado tenga la calidad requerida en comparación con un sistema diseñado como módulos o microservicios muy poco acoplados? ¿Cuáles son las ventajas o desventajas de cada enfoque?
- ¿Puedes pensar en un software popular que sufra de exceso de funciones? Es decir, un software que contiene muchas más funciones de las que jamás usaría, cada función presenta más oportunidades para errores y vulnerabilidades de seguridad, y hace que las funciones que usa sean más difíciles de encontrar y administrar. ¿Corres tú mismo el peligro de caer en esta trampa?



Tema 6

Su carpeta de conocimientos

Una inversión en
el conocimiento siempre
paga el mejor interés.

Benjamin Franklin

Ah, el bueno de Ben Franklin, nunca le falta una homilía concisa. Bueno, 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. Tus conocimientos y experiencia son tus activos profesionales más importantes en el día a día.

Desafortunadamente, son activos que exiran. 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. Dado el ritmo cada vez mayor de cambio en nuestra sociedad tecnológica, esto puede suceder con bastante rapidez.

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.

Su capacidad para aprender cosas nuevas es su activo estratégico más importante. Pero, ¿cómo aprendes a aprender y cómo sabes qué aprender?

TU PORTAFOLIO 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 sus carpetas 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 conservadores y inversiones de alto riesgo y alta rentabilidad.
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 invertir en su cartera de conocimientos utilizando estas mismas pautas.

La buena noticia es que administrar este tipo de inversión es una habilidad como cualquier otra: se puede aprender. El truco es obligarte a hacerlo inicialmente y formar un hábito. Desarrolle una rutina que siga hasta que su cerebro la interiorice. En ese momento, te encontrarás absorbiendo nuevos conocimientos automáticamente.

CONSTRUYENDO TU CARTERA

Invierta con regularidad Al igual que en las inversiones financieras, debe invertir en su cartera de conocimientos con regularidad, aunque sea una pequeña cantidad. El hábito es tan importante como las sumas, así que planee usar un tiempo y lugar consistentes, lejos de las interrupciones. Algunos objetivos de muestra se enumeran en la siguiente sección.

Diversificar

Cuantas más cosas distintas sepas, más valiosas

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. Y no olvide todas las demás habilidades que necesita, incluidas aquellas en áreas no técnicas.

Gestionar el

riesgo La tecnología existe a lo largo de un espectro de riesgo, desde estándares arriesgados y potencialmente de alta recompensa hasta estándares de bajo riesgo y baja recompensa. Una buena idea invertir todo su dinero en acciones de empresas que podrían colapsar repentinamente, ni debe invertir en una cartera conservadora y perder posibles oportunidades. No ponga todos sus huevos técnicos en una canasta.

En el ámbito de la programación y el desarrollo de software, esto puede traducirse en no depender únicamente de una sola tecnología, lenguaje de programación o enfoque. Es importante estar abierto a explorar diferentes tecnologías y herramientas, y evaluar los riesgos y beneficios asociados a cada una. También implica diversificar las habilidades y conocimientos para estar preparado ante cambios en el mercado y aprovechar oportunidades emergentes.

Compre barato,

venda caro Aprender una tecnología emergente antes de que se vuelva popular puede ser tan difícil como encontrar una acción infravalorada, pero la recompensa puede ser igual de gratificante. Aprender Java cuando se introdujo por primera vez y era desconocido puede haber sido arriesgado en ese momento, pero valió la pena para los primeros usuarios cuando se convirtió en un pilar de la industria más tarde.

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 quizás

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:

Consejo 9

Invierta regularmente en su cartera de conocimientos

OBJETIVOS

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, aprender muchos idiomas es fácil gracias a la gran cantidad de software disponible de forma gratuita.

Leer un libro técnico cada mes.

Si bien hay una gran cantidad de ensayos de formato breve y, en ocasiones, respuestas confiables en la web, para una comprensión profunda, necesita libros de formato largo. Busque en las librerías libros técnicos sobre temas interesantes relacionados con su proyecto actual.^[10] Una vez que tenga el hábito, lea un libro al mes. Una vez que haya dominado las tecnologías que está utilizando actualmente, ramifique y estudie algunas que no se relacionen con su proyecto.

Lee también libros no técnicos

Es importante recordar que las personas utilizan las computadoras , personas cuyas necesidades intenta satisfacer. Trabajas con personas, eres empleado por personas y eres pirateado por personas. No olvide el lado humano de la ecuación, ya que requiere un conjunto de habilidades completamente diferente (irónicamente las llamamos habilidades blandas , pero en realidad son bastante difíciles de dominar).

Tomar clases

Busque cursos interesantes en un colegio o universidad local o en línea, o tal vez en la próxima feria comercial o conferencia cercana.

Participe en reuniones y grupos de usuarios locales El aislamiento puede ser mortal para su carrera; averigüe en qué está trabajando la gente fuera de su empresa. No se limite a ir y escuchar: participe activamente.

Experimente con diferentes entornos Si ha trabajado solo en Windows, dedique algún tiempo a Linux. Si solo ha usado [makefiles](#) y un editor, pruebe con un IDE sofisticado con características de vanguardia, y viceversa. viceversa

Manténgase

actualizado Lea noticias y publicaciones en línea sobre tecnología diferente a la de su proyecto actual. Es una excelente manera de averiguar qué experiencias tienen otras personas con él, la jerga particular que usan, etc.

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 procedimentales de manera diferente. Comprenda el paradigma de programación funcional y escribirá código orientado a objetos de manera diferente, y así sucesivamente.

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. Pregunta por ahí. Busque en la web: las partes académicas también, no solo las partes del consumidor.

Si no puede encontrar la respuesta usted mismo, averigüe quién puede hacerlo. 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 propio correo electrónico.

lector con usted, o puede encontrarse hojeando un artículo de 1973 sobre Papúa Nueva Guinea.

PENSAMIENTO CRÍTICO

El último punto importante es pensar críticamente sobre lo que lee y escucha. 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. Tenga cuidado con los fanáticos que insisten en que su dogma proporciona la única respuesta: puede o no ser aplicable a usted y 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.

Consejo 10

Analice críticamente lo que lee y escucha

El pensamiento crítico es una disciplina completa en sí misma, y lo alentamos a que lea y estudie todo lo que pueda al respecto. Mientras tanto, aquí hay una ventaja inicial con algunas preguntas para hacer y pensar.

Pregúntate los "cinco porqués"

Un truco de consultoría favorito: preguntar "¿por qué?" al menos cinco veces.

Haga una pregunta y obtenga una respuesta. Profundice preguntando "¿por qué?" Repite como si fueras un niño petulante de cuatro años (pero educado). Es posible que pueda acercarse a una causa raíz de esta manera.

¿A quién beneficia esto?

Puede sonar cínico, pero seguir el dinero puede ser un camino muy útil para analizar. Los beneficios para otra persona u otra organización pueden estar alineados con los suyos o no.

¿Cuál es el contexto?

Todo ocurre en su propio contexto, razón por la cual las soluciones de "talla única" a menudo no lo hacen. Considere un artículo o libro que promueva una "mejor práctica". Buenas preguntas a considerar son "¿mejor para quién?" ¿Cuáles son los requisitos previos, cuáles son las consecuencias, a corto y largo plazo?

¿Cuándo o dónde funcionaría esto?

¿Bajo qué circunstancias? ¿Es demasiado tarde? ¿Demasiado temprano? No se detenga con el pensamiento de primer orden (qué sucederá después), sino que utilice el pensamiento de segundo orden: ¿qué sucederá después de eso?

¿Por qué es esto un problema?

¿Hay un modelo subyacente? ¿Cómo funciona el modelo subyacente?

Desafortunadamente, ya hay muy pocas respuestas simples. Pero con su extenso portafolio, y aplicando un poco de análisis crítico al torrente de artículos técnicos que leerá, puede comprender las respuestas complejas .

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 1, es tu vida
- Tema 22, Cuadernos diarios de ingeniería

RETOS

- Comienza a aprender un nuevo idioma esta semana. ¿Siempre programado en el mismo lenguaje antiguo? Pruebe Clojure, Elixir, Elm, F#, Go, Haskell, Python, R, ReasonML, Ruby, Rust, Scala, Swift, TypeScript o [11] cualquier otra cosa que le atraiga y/o parezca que podría gustarle .—.
- 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 otros entusiastas en una reunión local.



Tema 7

¡Comunicar!

yo creo que es
mejor ser revisado
de lo que es ser
pasado por alto.

Mae West, Bella de la
años 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.

Trata el inglés (o cualquiera que sea tu lengua materna) como un lenguaje de programación más. Escriba lenguaje natural como escribiría código: respete el principio SECO, ETC, automatización, etc. (Discutimos los principios de diseño DRY y ETC en el próximo capítulo).

Consejo 11

El inglés es solo otro lenguaje de programación

Hemos reunido una lista de ideas adicionales que encontramos útiles.

CONOCE A TU AUDIENCIA

Solo se está comunicando si está transmitiendo lo que quiere transmitir; solo hablar no es suficiente. 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.

[12]

Supongamos que desea cambiar su sistema de monitoreo remoto para usar un intermediario de mensajes de terceros para difundir notificaciones de estado.

Puede presentar esta actualización de muchas maneras diferentes, según su audiencia. Los usuarios finales apreciarán que sus sistemas ahora pueden interoperar con otros servicios que utilizan el bróker.

Su departamento de marketing podrá utilizar este hecho para impulsar las ventas. Los gerentes de desarrollo y operaciones estarán felices porque el cuidado y mantenimiento de esa parte del sistema ahora es problema de otra persona. Finalmente, los desarrolladores pueden disfrutar adquiriendo experiencia con las nuevas API e incluso pueden encontrar nuevos usos para el intermediario de mensajes. Al hacer el lanzamiento apropiado para cada grupo, logrará que todos se entusiasmen con su proyecto.

Al igual que con todas las formas de comunicación, el truco aquí es recopilar comentarios. No se limite a esperar las preguntas: pregunte por ellas. Mire el lenguaje corporal y las expresiones faciales. Uno de los presupuestos de la Programación Neurolingüística es “El significado de tu comunicación es la respuesta que obtienes”. Mejore continuamente su conocimiento de su audiencia a medida que se comunica.

SABE LO QUE QUIERES 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 quieres decir. Los escritores de ficción a menudo trazan sus libros en detalle antes de comenzar, pero las personas que escriben documentos técnicos a menudo se sienten felices de sentarse frente a un teclado, ingresar:

1. Introducción

y empezar a escribir lo que se les ocurra a continuación.

Planifica lo que quieras decir. Escribe un esquema. Luego pregúntese: "¿Esto comunica lo que quiero expresar a mi audiencia de una manera que funcione para ellos?" Refina hasta que lo haga.

Este enfoque funciona para algo más que documentos. Cuando se enfrente a una reunión importante o una conversación con un cliente importante, anote las ideas que desea comunicar y planifique un par de estrategias para transmitirlas.

Ahora que sabe lo que quiere su audiencia, entreguemoslo.

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 computadora portátil.

Como parte de la comprensión de lo que su audiencia necesita escuchar, debe determinar cuáles son sus prioridades. Atrapa a una gerente a la que su jefe acaba de hacerle pasar un mal rato porque algunos

el código fuente se perdió y 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...?"

ELIJA 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. ¿Cuál es su nivel de habilidad y experiencia en esta área? ¿Son expertos? ¿Principiantes? ¿Necesitan que los tomen de la mano o simplemente un tl; dr rápido? 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 (o observador de Food Network) le dirá que puede esclavizarse en la cocina durante horas solo para arruinar sus esfuerzos con una mala presentación.

Hoy en día no hay excusa para producir documentos impresos de mala apariencia. El software moderno puede producir resultados sorprendentes, independientemente de si está escribiendo con Markdown o con un

procesador de textos. Necesitas aprender solo algunos comandos básicos. Si utiliza un procesador de textos, utilice sus hojas de estilo para mantener la coherencia. (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. Revisa la ortografía, primero automáticamente y luego a mano. Después del punzón, hay filetes de señorita de ortografía que el corrector puede anudar ketch.

INVOLUCRE A SU 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: escúchalos. 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 repitan la discusión con sus propias palabras. 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 logra volver a la gente

cuando te envían un correo electrónico o una nota 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 informadas a las personas hace que perdonen mucho más los deslices ocasionales y les hace sentir que no los ha olvidado.

Consejo 12

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.

DOCUMENTACIÓN

Finalmente, está la cuestión de la comunicación a través de la documentación. 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 propio código.

De hecho, queremos aplicar todos nuestros principios pragmáticos tanto a la documentación como al código.

Consejo 13

Incorpore la documentación, no la atornille

Es fácil producir documentación atractiva a partir de los comentarios en el código fuente, y recomendamos agregar

comentarios a los módulos y funciones exportadas para dar una ventaja a otros desarrolladores cuando lleguen a usarlo.

Sin embargo, esto no significa que estemos de acuerdo con la gente que dice que cada función, estructura de datos, declaración de tipo, etc., necesita su propio comentario. Este tipo de escritura de comentarios mecánicos en realidad hace que sea más difícil mantener el código: ahora hay dos cosas para actualizar cuando realiza un cambio. Por lo tanto, restrinja sus comentarios que no sean API para discutir por qué se hace algo, su propósito y su objetivo. El código ya muestra cómo se hace, por lo que comentar esto es redundante y es una violación del principio DRY.

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.

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.
- Mantenga el código y la documentación juntos.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 15, Estimación
- Tema 18, Edición avanzada
- Tema 45, El pozo de requisitos
- Tema 49, Equipos Pragmáticos

Comunicación en línea

Todo lo que hemos dicho sobre la comunicación por escrito se aplica igualmente al correo electrónico, publicaciones en redes sociales, blogs, etc. El correo electrónico en particular ha evolucionado hasta el punto en que es un pilar de las comunicaciones corporativas; 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 incoherentes y de aspecto desagradable alrededor del mundo.

Nuestros consejos son simples:

- Revisa antes de presionar ENVIAR .
- Verifique su ortografía y busque cualquier percance accidental de autocorrección.
- Mantenga el formato simple y claro.
- Mantenga las citas al mínimo. A nadie le gusta recibir su propio correo electrónico de 100 líneas con el mensaje "Acepto".
- Si está citando el correo electrónico de otras personas, asegúrese de atribuirlo y citarlo en línea (en lugar de como un archivo adjunto). Lo mismo al citar en plataformas de redes sociales.
- No llames ni actúes como un troll a menos que quieras que vuelva y te persiga más tarde. Si no se lo dirías a alguien en la cara, no lo digas en línea.
- Revisa tu lista de destinatarios antes de enviar. Se ha convertido en un cliché criticar al jefe por correo electrónico departamental sin darse cuenta de que el jefe está en la lista de cc. Mejor aún, no critiques al jefe por correo electrónico.

Como han descubierto innumerables grandes corporaciones y políticos, los correos electrónicos y las publicaciones en las redes sociales son para siempre. Trate de prestar la misma atención y cuidado al correo electrónico como lo haría con cualquier nota o informe escrito.

RETOS

- Hay varios buenos libros que contienen secciones sobre comunicaciones dentro de los equipos, incluidos The Mythical Man Month: Essays on Software Engineering [Bro96] y Peopleware: Productive Projects and Teams [DL13]. Asegúrese de intentar leerlos durante los próximos 18 meses. Además, Dinosaur Brains: Dealing with All That Impossible People at Work [BR89] 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 seguir los consejos de esta sección antes de comenzar. Identifique explícitamente a la audiencia y lo que necesita comunicar. Si corresponde, hable con su audiencia después y vea qué tan precisa fue su evaluación de sus necesidades.

notas al pie

[3] <http://wiki.c2.com/?ChangeYourOrganization>

[4] Consulte, por ejemplo, un buen metanálisis en Confianza y rendimiento del equipo: un metanálisis de los efectos principales, moderadores y covariables, <http://dx.doi.org/10.1037/apl0000110>

[5] Ver La policía y la seguridad del vecindario [WH82]

[6] Ver Depresión contagiosa: existencia, especificidad de los síntomas depresivos y el papel de la búsqueda de tranquilidad [Joi94]

[7] Mientras hace esto, puede sentirse recomfortado por la frase atribuida a la contraalmirante Dra. Grace Hopper: "Es más fácil pedir perdón que obtener permiso".

[8] ¡Se suponía que era una broma!

[9] 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.

[10] Puede que seamos parciales, pero hay una excelente selección disponible en <https://pragprog.com>.

[11] ¿Nunca has oido hablar de ninguno de estos idiomas? Recuerde, el conocimiento es un activo que caduca, al igual que la tecnología popular. La lista de lenguajes nuevos y experimentales calientes era muy diferente para la primera edición, y probablemente sea diferente nuevamente cuando lea esto. Razón de más para seguir aprendiendo.

[12] La palabra molestar proviene del francés antiguo enui, que también significa "aburrir".

Copyright © 2020 Pearson Educación, Inc.

Capítulo 2

Un enfoque pragmático

Hay ciertos consejos y trucos que se aplican en todos los niveles de desarrollo de software, procesos que son prácticamente universales e ideas que son casi axiomáticas. 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. Pero para su conveniencia, reuniremos estas ideas y procesos aquí.

El primer tema, y quizás el más importante, llega al corazón del desarrollo de software: el Tema 8, La esencia del buen diseño.

Todo se sigue de esto.

Las siguientes dos secciones, el Tema 9, SECO—Los males de la duplicación y el Tema 10, 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 el Tema 11, Reversibilidad, veremos algunas técnicas que ayudan a aislar sus proyectos de su entorno cambiante.

Las siguientes dos secciones también están relacionadas. En el Tema 12, Tracer Bullets, hablamos sobre un estilo de desarrollo que le permite recopilar requisitos, probar diseños e implementar código al mismo tiempo. Es la única manera de seguir el ritmo de la vida moderna.

El Tema 13, Prototipos y Post-it Notes, le muestra cómo utilizar la creación de prototipos para probar arquitecturas, algoritmos, interfaces e ideas. En el mundo moderno, es fundamental probar las ideas y recibir comentarios antes de comprometerse con ellas de todo corazón.

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 "hacerlo así", en el Tema 14, Idiomas de dominio, presentamos algunas sugerencias más modestas que puede implementar usted mismo.

Finalmente, todos trabajamos en un mundo de tiempo y recursos limitados. Puede sobrevivir mejor a estas escaseces (y mantener a sus jefes o clientes más felices) si se le da bien calcular cuánto tiempo tomarán las cosas, que tratamos en el Tema 15, Estimación .

Tenga en cuenta estos principios fundamentales durante el desarrollo y escribirá un código mejor, más rápido y más sólido. Incluso puedes hacer que parezca fácil.



Tema 8

La esencia del buen diseño

El mundo está lleno de gurús y expertos, todos ansiosos por transmitir su sabiduría ganada con tanto esfuerzo en lo que respecta a cómo diseñar software. Hay acrónimos, listas (que parecen preferir cinco entradas), patrones, diagramas, videos, charlas y (Internet es Internet) probablemente una serie genial sobre la Ley de Deméter explicada usando danza interpretativa.

Y nosotros, sus gentiles autores, también somos culpables de esto. Pero nos gustaría hacer las paces explicando algo que se nos hizo evidente recientemente. En primer lugar, la declaración general:

Consejo 14 Un buen diseño es más fácil de cambiar que un mal diseño

Una cosa está bien diseñada si se adapta a las personas que la usan. Para el código, eso significa que debe adaptarse cambiando. Por eso creemos en el principio de ETC: más fácil de cambiar. ETC. Eso es todo.

Por lo que sabemos, cada principio de diseño que existe es un caso especial de ETC.

¿Por qué es bueno el desacoplamiento? Porque al aislar las preocupaciones hacemos que cada una sea más fácil de cambiar. ETC.

¿Por qué es útil el principio de responsabilidad única? Porque un cambio en los requisitos se refleja en un cambio en un solo módulo. ETC.

¿Por qué es importante nombrar? Porque los buenos nombres hacen que el código sea más fácil de leer y tienes que leerlo para cambiarlo. ¡ETC!

ETC ES UN VALOR, NO UNA REGLA

Los valores son cosas que te ayudan a tomar decisiones: ¿debo hacer esto o aquello? Cuando se trata de pensar en software, ETC es una guía que lo ayuda a elegir entre caminos. Al igual que todos sus otros valores, debe estar flotando justo detrás de su pensamiento consciente, empujándolo sutilmente en la dirección correcta.

Pero, ¿cómo haces que eso suceda? Nuestra experiencia es que requiere un refuerzo consciente inicial. Es posible que deba pasar una semana más o menos preguntándose deliberadamente "¿Lo que acabo de hacer hizo que el sistema general sea más fácil o más difícil de cambiar?" Hazlo cuando guardes un archivo. Hazlo cuando escribas una prueba. Hazlo cuando corrijas un error.

Hay una premisa implícita en ETC. Asume que una persona puede decir cuál de muchos caminos será más fácil de cambiar en el futuro.

La mayor parte del tiempo, el sentido común será correcto y podrá hacer una conjetura informada.

A veces, sin embargo, no tendrás ni idea. Está bien. En esos casos, creemos que puedes hacer dos cosas.

En primer lugar, dado que no está seguro de qué forma tomará el cambio, siempre puede recurrir al último camino "fácil de cambiar": intente hacer que lo que escribe sea reemplazable. De esa manera, pase lo que pase en el futuro, este trozo de código no será un obstáculo. Parece extremo, pero en realidad es lo que deberías estar haciendo todo el tiempo de todos modos. Realmente es solo pensar en mantener el código desacoplado y cohesivo.

En segundo lugar, trata esto como una forma de desarrollar los instintos. Anote la situación en su diario de ingeniería: las opciones que tiene y algunas conjeturas sobre el cambio. Deja una etiqueta en la fuente. Luego, más tarde, cuando este código tenga que cambiar, podrá mirar hacia atrás y darse su opinión. Podría ayudar la próxima vez que llegue a una bifurcación similar en el camino.

El resto de las secciones de este capítulo tienen ideas específicas sobre el diseño, pero todas están motivadas por este principio.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 9, DRY—Los males de la duplicación
- Tema 10, Ortogonalidad
- Tema 11, Reversibilidad
- Tema 14, Idiomas de dominio
- Tema 28, Desacoplamiento
- Tema 30, Transformación de la programación
- Tema 31, Impuesto sobre Sucesiones

RETOS

- Piense en un principio de diseño que utilice con regularidad. ¿Tiene la intención de hacer que las cosas sean fáciles de cambiar?
- Piense también en lenguajes y paradigmas de programación (OO, FP, Reactivo, etc.). ¿Alguno tiene grandes aspectos positivos o negativos cuando se trata de ayudarlo a escribir código ETC? ¿Alguno tiene ambos?

Al codificar, ¿qué puede hacer para eliminar los aspectos negativos y [13] acentuar los aspectos positivos?

- Muchos editores tienen soporte (ya sea integrado o mediante extensiones) para ejecutar comandos cuando guarda un archivo. ¿Hacer que su editor muestre un ETC? [14] mensaje cada vez que guarde y úselo como una señal para pensar en el código que acaba de escribir. ¿Es fácil de cambiar?



Tema 9

DRY—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 su código.

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

Desafortunadamente, el conocimiento no es estable. Cambia, a menudo rápidamente. Su comprensión de un requisito puede cambiar después de una reunión con el cliente. El gobierno cambia una regulación y 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 y los requisitos existentes evolucionan a medida que avanzamos en el proyecto. 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 principio DRY:

Cada pieza de conocimiento debe tener una representación única, inequívoca y autorizada dentro de un sistema.

¿Por qué lo llamamos SECO?

Consejo 15

SECO: no se repita

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.

Encontrará que el principio DRY 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 duplicación y

sugerir estrategias generales para enfrentarlo.

SECO ES MÁS QUE CÓDIGO

Vamos a sacar algo del camino por adelantado. En la primera edición de este libro hicimos un mal trabajo al explicar lo que entendíamos por No te repitas. Mucha gente lo tomó para referirse solo al código: pensaron que DRY significa "no copiar y pegar líneas de fuente".

Eso es parte de DRY, pero es una parte pequeña y bastante trivial.

DRY se trata de la duplicación del conocimiento, de la intención. Se trata de expresar lo mismo en dos lugares diferentes, posiblemente de dos maneras totalmente diferentes.

Aquí está la prueba de fuego: cuando una sola faceta del código tiene que cambiar, ¿se encuentra haciendo ese cambio en múltiples lugares y en múltiples formatos diferentes? ¿Tiene que cambiar el código y la documentación, o el esquema de una base de datos y la estructura que lo contiene, o..? Si es así, su código no está SECO.

Entonces, veamos algunos ejemplos típicos de duplicación.

DUPLICACIÓN DE CÓDIGO

Puede ser trivial, pero la duplicación de código es muy común. Aquí hay un ejemplo:

```
def print_balance(cuenta)
    printf "Débitos: %10.2f\n", cuenta.débitos
    printf "Créditos: %10.2f\n", cuenta.créditos
    if cuenta.tasas < 0
        printf "Tarifas: %10.2f-\n", -cuenta.tarifas
    else
        printf "Tarifas: %10.2f\n", cuenta.tarifas
```

```

end
printf " —\n" si
cuenta.saldo < 0
printf "Saldo: %10.2f\n", -cuenta.saldo else printf
"Saldo: %10.2f\n", cuenta.saldo end end

```

Por ahora, ignore la implicación de que estamos cometiendo el error de novato de almacenar monedas en flotadores. En su lugar, vea si puede detectar duplicaciones en este código. (Podemos ver al menos tres cosas, pero es posible que veas más).

¿Qué encontraste? Aquí está nuestra lista.

Primero, claramente hay una duplicación de copiar y pegar en el manejo de los números negativos. Podemos arreglar eso agregando otra función:

```

def formato_cantidad(valor)
    resultado = sprintf("%10.2f", valor.abs)
    si valor < 0
        resultado + "_"
    más
        resultado ++
    +
final final

def print_balance(cuenta)
    printf "Débitos: %10.2f\n", cuenta.débitos
    printf "Créditos: %10.2f\n", cuenta.créditos
    printf "Comisiones: %s\n", format_amount(cuenta.cargos)
    printf " —\n" printf
    "Saldo: %s\n", format_amount(cuenta.saldo) end

```

Otra duplicación es la repetición del ancho del campo en todas las llamadas a `printf`. Podríamos arreglar esto introduciendo una constante y

pasándolo a cada llamada, pero ¿por qué no usar simplemente la función existente?

```
def formato_cantidad(valor)
    resultado = sprintf("%10.2f", valor.abs)
    si valor < 0
        resultado + "-"
    más
        resultado ++
    +
final final

def print_balance(cuenta)
    printf "Débitos: %s\n", format_amount(cuenta.débitos) printf
    "Créditos: %s\n", format_amount(cuenta.créditos) printf
    "Comisiones: %s\n", format_amount(cuenta .fees) printf
    " —\n" printf "Saldo:
    %s\n", format_amount(cuenta.saldo) end
```

¿Algo más? Bueno, y si el cliente pide un espacio extra entre las etiquetas y los números? Tendríamos que cambiar cinco líneas. Eliminemos esa duplicación:

```
def formato_cantidad(valor)
    resultado = sprintf("%10.2f", valor.abs)
    si valor < 0
        resultado + "-"
    más
        resultado ++
    +
final final

def print_line(etiqueta, valor)
    printf "%-9s%s\n", etiqueta, valor
fin

def report_line(etiqueta, cantidad)
    print_line(etiqueta + ":", format_amount(cantidad))
end
```

```

def imprimir_saldo(cuenta)
    línea_informe("Débitos", cuenta.débitos)
    línea_informe("Créditos", cuenta.créditos)
    línea_informe("Comisiones",
    cuenta.cuotas) línea_impresión("",  

    "—") línea_informe(" Saldo", cuenta.saldo)
end

```

Si tenemos que cambiar el formato de las cantidades, cambiamos `format_amount`.

Si queremos cambiar el formato de la etiqueta, cambiamos `report_line`.

Todavía hay una infracción DRY implícita: el número de guiones en la línea de separación está relacionado con el ancho del campo de cantidad. Pero no es una coincidencia exacta: actualmente es un carácter más corto, por lo que los signos menos finales se extienden más allá de la columna. Esta es la intención del cliente y es una intención diferente al formato real de las cantidades.

No toda la duplicación de código es duplicación de conocimiento

Como parte de su aplicación de pedido de vino en línea, está capturando y validando la edad de su usuario, junto con la cantidad que está pidiendo. Según el propietario del sitio, ambos deberían ser números y ambos mayores que cero. Entonces codificas las validaciones:

```

def validar_edad(valor):
    validar_tipo(valor, :entero)
    validar_min_entero(valor, 0)

def validar_cantidad(valor):
    validar_tipo(valor, :entero)
    validar_min_entero(valor, 0)

```

Durante la revisión del código, el residente sabelotodo rebota este código, alegando que es una infracción SECA: ambos cuerpos de función son iguales.

Están equivocados. El código es el mismo, pero el conocimiento que representan es diferente. Las dos funciones validan dos cosas separadas que simplemente tienen las mismas reglas. Eso es una coincidencia, no una duplicación.

DUPLICACIÓN EN DOCUMENTACIÓN

De alguna manera nació el mito de que debes comentar todas tus funciones. Aquellos que creen en esta locura luego producen algo como esto:

```
# Calcule las tarifas para esta cuenta.

## * Cada cheque devuelto cuesta
$20 # * Si la cuenta está en sobregiro por más de 3
días, # cobra $10 por cada
día # * Si el saldo promedio de la cuenta es mayor a
$2,000 # reduce las tarifas en un 50%

def fee(a)
  f = 0
  if a.returned_check_count > 0
    f += 20 * a.returned_check_count
  end
  if a.overdraft_days > 3
    f += 10*a.overdraft_days
  end
  if a.average_balance > 2_000
    f /= 2
  end
  f
end
```

La intención de esta función se da dos veces: una en el comentario y otra en el código. El cliente cambia una tarifa y tenemos que actualizar ambas. Con el tiempo, podemos garantizar que el comentario y el código saldrán de sintonía.

Pregúntese qué agrega el comentario al código. De nuestro

punto de vista, simplemente compensa algunos nombres y diseños incorrectos. Que tal solo esto:

```
def compute_account_fees(cuenta) tarifas = 20
    * cuenta.returned_check_count tarifas += 10 *

    cuenta.overdraft_days if cuenta.overdraft_days > 3 tarifas /= 2 si cuenta.promedio_saldo
    > 2_000 tarifas finalizan
```

El nombre dice lo que hace, y si alguien necesita detalles, se encuentran en la fuente. ¡Eso es SECO!

Violaciones DRY en datos

Nuestras estructuras de datos representan conocimiento y pueden infringir el principio DRY. Veamos una clase que representa una línea:

```
línea de clase {
    Punto de inicio;
    punto final;
    longitud doble ;};
```

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 tenemos 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:

```
línea de clase
{ Punto de
inicio; punto
final; double length() { return inicio.distanciaHasta(fin); } };
```

Más adelante en el proceso de desarrollo, puede optar por violar el principio DRY por motivos 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:

```

línea de clase
{ longitud doble privada ;
Punto de inicio privado ;
Extremo de punto privado ;

public Line(Punto de inicio, Punto final)
    { this.start = start;
      este.fin = fin;
      calcularLongitud(); }

// public
void setStart(Punto p) { this.start = p; calcularLongitud(); } void
setEnd(Punto p) { this.end = p; calcularLongitud(); }

Punto getStart()      { volver a empezar; }
Punto getEnd()        { volver al final; }

double getLength() { devolver longitud; }

private void calcularLongitud() { this.length
    = start.distanceTo(end); } };

```

Este ejemplo también ilustra un tema importante: cada vez que un módulo expone una estructura de datos, está acoplando todo el código que usa esa estructura a la implementación de ese módulo. Siempre que sea posible, utilice siempre funciones de acceso para leer y escribir los atributos de los objetos. Hará que sea más fácil agregar funcionalidad en el futuro.

Este uso de funciones de acceso se vincula con el Uniforme de Meyer. Principio de acceso, descrito en [Software Orientado a Objetos Construcción \[Mey97\]](#), 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 REPRESENTATIVA

Su código interactúa con el mundo exterior: otras bibliotecas a través de API, otros servicios a través de llamadas remotas, datos en fuentes externas, etc. Y casi cada vez que lo hace, introduce algún tipo de violación DRY: su código debe tener conocimiento que también está presente en el elemento externo. Necesita conocer la API, o el esquema, o el significado de los códigos de error, o lo que sea.

La duplicación aquí es que dos cosas (su código y la entidad externa) deben tener conocimiento de la representación de su interfaz. Cámbialo en un extremo, y el otro extremo se rompe.

Esta duplicación es inevitable, pero puede mitigarse. Aquí hay algunas estrategias.

Duplicación entre API internas Para

las API internas, busque herramientas que le permitan especificar la API en algún tipo de formato neutral. Estas herramientas generalmente generarán documentación, API simuladas, pruebas funcionales y clientes de API, estos últimos en varios idiomas diferentes. Idealmente, la herramienta almacenará todas sus API en un repositorio central, lo que les permitirá compartirlas entre equipos.

Duplicación entre API externas

Cada vez más, encontrará que las API públicas se documentan formalmente usando algo como OpenAPI. Esto ^[15] le permite importar la especificación de API en sus herramientas de API locales e integrarse de manera más confiable con el servicio.

Si no puede encontrar dicha especificación, considere crear una y

publicándolo. No solo otros lo encontrarán útil; incluso puede obtener ayuda para mantenerlo.

Duplicación con fuentes de datos

Muchas fuentes de datos le permiten realizar una introspección en su esquema de datos. Esto se puede usar para eliminar gran parte de la duplicación entre ellos y su código. En lugar de crear manualmente el código para contener estos datos almacenados, puede generar los contenedores directamente desde el esquema. Muchos marcos de persistencia harán este trabajo pesado por usted.

Hay otra opción, y una que a menudo preferimos. En lugar de escribir código que represente datos externos en una estructura fija (una instancia de una estructura o clase, por ejemplo), simplemente péguelo en una estructura de datos clave/valor (su idioma podría llamarlo mapa, hash, diccionario o incluso objeto).

Por sí solo, esto es arriesgado: pierde gran parte de la seguridad de saber con qué datos está trabajando. Por lo tanto, recomendamos agregar una segunda capa a esta solución: un conjunto simple de validación basado en tablas que verifica que el mapa que ha creado contiene al menos los datos que necesita, en el formato que necesita. Su herramienta de documentación API podría generar esto.

DUPLICACIÓN ENTRE DESARROLLADORES

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 que contenían cada uno una versión diferente del Seguro Social.

Código de validación de número.

A un alto nivel, aborde el problema creando un equipo fuerte y unido con buenas comunicaciones.

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 encima.

Creemos que la mejor manera de lidiar con esto es fomentar una comunicación activa y frecuente entre los desarrolladores.

Tal vez organice una reunión diaria de scrum standup. Configure foros (como canales de Slack) para discutir problemas comunes. Esto proporciona una forma no intrusiva de comunicarse, 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 recuerde, el acceso es recíproco: no se deje engañar por otras personas que examinan detenidamente (*¡manoseando?*) su código tampoco.

Consejo Haz 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 reutilizas, corres el riesgo

duplicidad de conocimientos.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 8, La esencia del buen diseño
- Tema 28, Desacoplamiento
- Tema 32, Configuración
- Tema 38, Programación por Coincidencia
- Tema 40, Refactorización



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 independientes. A medida que el número 1 en el diagrama se mueve hacia el norte, no cambia la distancia al este o al oeste. El número 2 se mueve hacia el este, pero no hacia el norte o el sur.

imágenes/ortogonalidad.png

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 almorzar pescado, gime repentinamente y se desmaya. Afortunadamente, te dejó flotando a 100 pies sobre el suelo.

Por suerte, había leído una página de Wikipedia sobre helicópteros la noche anterior. Sabes que los helicópteros tienen cuatro controles básicos. El cílico es el palo que sostienes en tu mano derecha. Muévelo, y el helicóptero se mueve en la dirección correspondiente. Su mano izquierda sostiene la 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. Finalmente, tiene dos pedales, que varían la cantidad de empuje del rotor de cola y ayudan a girar el helicóptero.

“¡ Fácil! ” , piensas. "Baje suavemente la palanca de paso colectivo y descenderá con gracia al suelo, un héroe". 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, descubre 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.

Consejo 17

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 en Diseño estructurado: fundamentos de una disciplina de diseño de sistemas y programas informáticos [YC79]). 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 se pueden diseñar, codificar, probar y luego olvidar; 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 su original

implementadores. 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 hace 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.

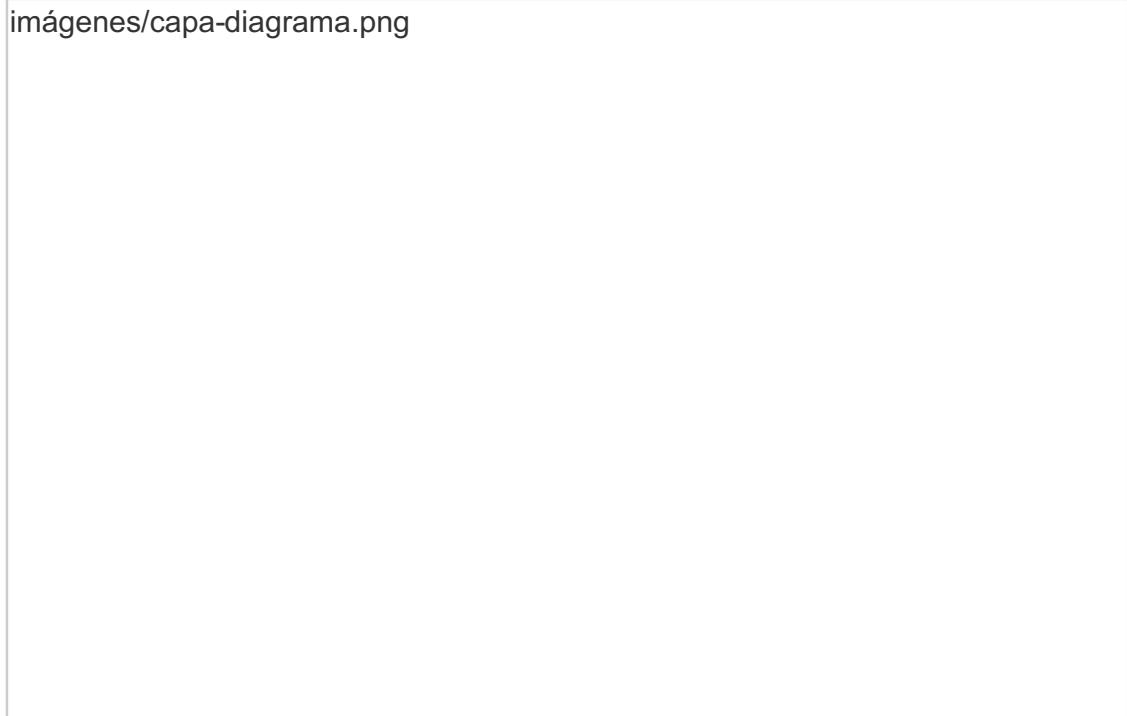
Veamos algunas de las formas en que puede aplicar el principio de ortogonalidad a su trabajo.

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:

imágenes/capa-diagrama.png



Hay una prueba fácil para el diseño ortogonal. Una vez que haya mapeado sus componentes, pregúntese: si cambio drásticamente los requisitos detrás de una función en particular, ¿cuántos módulos se ven afectados? En un sistema ortogonal, la respuesta debería ser "uno". Mover un botón en un panel GUI no debería ^{desdeñar} 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 una interfaz móvil que permita a los ingenieros monitorear los valores clave. 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.

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? Los códigos postales, los números de seguridad social o las identificaciones gubernamentales, las direcciones de correo electrónico y los dominios son identificadores externos sobre los que no tiene control y pueden cambiar en cualquier momento y por cualquier motivo. No confíes en las propiedades de las cosas que no puedes controlar.

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.

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 anotaciones, fuera de los métodos que hacen el trabajo. El mismo código de aplicación puede ejecutarse en diferentes entornos de transacciones EJB sin cambios.

En cierto modo, EJB es un ejemplo del patrón Decorator: agregar funcionalidad a las cosas sin cambiarlas. Este estilo de programación se puede usar en casi todos los lenguajes de programación y no requiere necesariamente un marco o biblioteca. Solo se necesita un poco de disciplina al programar.

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, que analizamos en el Tema 28, [Desacoplamiento](#). 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

que permanecerás 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 los constructores de objetos. En otro código, puede crear estructuras que contengan el contexto y pasarles referencias.

El patrón Singleton en Design Patterns: Elements of Reusable Object-Oriented Software [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, ya que también pueden generar vínculos innecesarios.

Evite funciones similares

A menudo se encontrará con un conjunto de funciones que se ven similares, tal vez compartan 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. Mirar

para cualquier oportunidad de reorganizarlo para mejorar su estructura y ortogonalidad. Este proceso se denomina refactorización y es tan importante que le hemos dedicado una sección (consulte el Tema 40, Refactorización).

PRUEBAS

Un sistema diseñado e implementado orthogonalmente 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 estas pruebas se realicen automáticamente como parte del proceso de compilación normal (consulte el Tema 41, Prueba de código).

Escribir pruebas unitarias es en sí mismo una interesante prueba de ortogonalidad. ¿Qué se necesita para que una prueba unitaria se construya y ejecute? ¿Tienes que importar un gran porcentaje del resto del código del sistema? 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 localizada está la solución. ¿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 usa un sistema de control de versiones (y lo hará después de leer el Tema 19, Control de versiones), 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 proporcionan hojas de estilo y macros que ayudan. Personalmente, preferimos usar un sistema de marcado como Markdown: cuando escribimos, nos enfocamos solo en el contenido y dejamos la presentación a la herramienta que usamos para renderizarla. [\[17\]](#)

VIVIR CON ORTOGONALIDAD

La ortogonalidad está estrechamente relacionada con el principio DRY. Con DRY, busca minimizar la duplicación dentro de un sistema, mientras que con ortogonalidad reduce la interdependencia entre los componentes del sistema. Puede que sea una palabra torpe, pero si utiliza el principio de ortogonalidad, combinado estrechamente con el principio DRY, descubrirá 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

- Tema 3, [Entropía del software](#)
- Tema 8, [La esencia del buen diseño](#)

- Tema 11, Reversibilidad
- Tema 28, Desacoplamiento
- Tema 31, Impuesto sobre Sucesiones
- Tema 33, Rompiendo el Acoplamiento Temporal
- Tema 34, El estado compartido es un estado incorrecto
- Tema 36, Pizarrones

Desafíos

- Considere la diferencia entre las herramientas que tienen una interfaz gráfica de usuario y las utilidades de línea de comandos pequeñas pero combinables que se usan en las indicaciones del 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? ¿Qué conjunto es más fácil de aprender?
- C++ admite herencia múltiple y Java permite que una clase implemente múltiples interfaces. Ruby tiene mixins. ¿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

Ejercicio 1 (respuesta posible)

Se le pide que lea un archivo una línea a la vez. Para cada línea, debe dividirla en campos. ¿Cuál de los siguientes conjuntos de definiciones de pseudoclase es probable que sea más ortogonal?

```
class Split1
{ constructor(fileName) # abre el archivo para leer def
  readNextLine() # pasa a la siguiente línea def getField(n)
                  # devuelve el campo enésimo en la línea actual
```

}

O

```
class Split2
{ constructor(línea)      # divide una línea
  def getField(n)          # devuelve el campo enésimo en la línea actual
```

Ejercicio 2 (respuesta posible)

¿Cuáles son las diferencias en la ortogonalidad entre los lenguajes orientados a objetos y funcionales? ¿Estas diferencias son inherentes a los idiomas en sí, o simplemente a la forma en que la gente los usa?



Tema 11

Reversibilidad

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

Emil-Auguste Chartier
(Alain), *Propos sur la religion*,
1938

Los ingenieros prefieren soluciones simples
y singulares a los problemas. Pruebas de
matemáticas que te permiten proclamar
con gran confianza que son más
cómodas que ensayos confusos y
cálidos sobre las innumerables causas
de la Revolución Francesa.

La gerencia tiende a estar de acuerdo con los
ingenieros: las respuestas singulares y
sencillas encajan muy bien en las hojas de
cálculo y los planes del proyecto.

¡ Si tan solo el mundo real cooperara! Desafortunadamente, si bien es hoy, es
 posible que deba ser mañana y la próxima semana.

Nada es para siempre, y si confía mucho en algún hecho, casi puede garantizar que
cambiará .

Siempre hay más de una forma de implementar algo y, por lo general, hay más
de un proveedor disponible para proporcionar un producto de terceros. Si inicia 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 en 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 estrecha de la realidad que tiene menos opciones.

Cuando se han tomado muchas decisiones críticas, el objetivo se vuelve tan pequeño que si se mueve, o si el viento cambia de dirección, o una mariposa en Tokio bate sus alas, te pierdes. Y puede fallar por una gran cantidad.^[18]

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

Una vez que decida utilizar la base de datos de este proveedor, o ese patrón arquitectónico, o un determinado modelo de implementación, 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 apegarnos a sus recomendaciones, especialmente el principio DRY, el desacoplamiento y el uso de la configuración externa, 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 documentos 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 realmente abstrajo 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 una aplicación basada en navegador, pero luego, al final del juego, marketing decide que lo que realmente quiere es una aplicación móvil. ¿Qué tan difícil sería para ti? En un mundo ideal, no debería afectarte demasiado, al menos en el lado del servidor. Estaría eliminando parte de la representación de HTML y reemplazándola con una API.

El error está en asumir que cualquier decisión es inamovible y en no prepararse para las contingencias que puedan surgir.

En lugar de tallar las 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.

Consejo 18

No hay decisiones finales

ARQUITECTURA FLEXIBLE

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

Estamos escribiendo esto en 2019. Desde el cambio de siglo, hemos visto las siguientes arquitecturas del lado del servidor de "mejores prácticas":

- Gran trozo de hierro
- Federaciones de hierro grande
- Clústeres de hardware básico con equilibrio de carga
- Máquinas virtuales basadas en la nube que ejecutan aplicaciones
- Máquinas virtuales basadas en la nube que ejecutan servicios
- Versiones en contenedores de lo anterior
- Aplicaciones sin servidor compatibles con la nube
- E, inevitablemente, un aparente regreso a los grandes trozos de hierro para algunas tareas.

Continúe y agregue las últimas y mejores modas a esta lista, y luego mírelo con asombro: es un milagro que algo haya funcionado.

¿Cómo se puede planificar para este tipo de volatilidad arquitectónica? no puedes

Lo que puedes hacer es facilitar el cambio. Oculte las API de terceros detrás de sus propias capas de abstracción. Divide tu código en componentes: incluso si terminas implementándolos en un solo servidor masivo, este enfoque es mucho más fácil que tomar una aplicación monolítica y dividirla. (Tenemos las cicatrices para probarlo.)

Y, aunque esto no es particularmente un problema de reversibilidad, un último consejo.

Consejo 19

Renunciar a las modas pasajeras

¡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

- Tema 8, La esencia del buen diseño
- Tema 10, Ortogonalidad
- Tema 19, Control de versiones
- Tema 28, Desacoplamiento
- Tema 45, El pozo de requisitos
- Tema 51, Kit de inicio pragmático

RETOS

- 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 (al menos mientras la caja permanezca cerrada). 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 en qué universo estás .

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?



Tema 12

Balas trazadoras

Preparaos, disparad, apuntad...

Luego

A menudo hablamos de alcanzar objetivos cuando desarrollamos software. En realidad, no estamos disparando nada en el campo de tiro, pero sigue siendo una metáfora útil y muy visual.

En particular, es interesante considerar cómo dar en el blanco en un mundo complejo y cambiante.

La respuesta, por supuesto, depende de la naturaleza del dispositivo con el que estés apuntando. Con muchos, solo tienes una oportunidad de apuntar, y luego puedes ver si aciertas o no. Pero hay una mejor manera.

¿Conoces todas esas películas, programas de televisión y videojuegos en los que la gente dispara ametralladoras? En estas escenas, a menudo verás la trayectoria de las balas como rayos brillantes en el aire. Estas rayas provienen de balas trazadoras.

Las balas trazadoras se cargan a intervalos 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.

Los soldados utilizan estos proyectiles trazadores para perfeccionar su puntería: es una retroalimentación pragmática en tiempo real en condiciones reales.

Ese mismo principio se aplica a los proyectos, particularmente cuando estás construyendo algo que no se ha construido antes. usamos el

desarrollo de viñetas trazadoras de términos para ilustrar visualmente la necesidad de una retroalimentación inmediata en condiciones reales con un objetivo en movimiento.

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. Produzca resmas de papel detallando cada requisito, atando todo lo desconocido y 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 el software equivalente a las balas 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 económica.

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.

Busque los requisitos importantes, los que definen el sistema. Busque las áreas donde tiene dudas y donde ve los mayores riesgos. Luego prioriza tu desarrollo para que estas sean las primeras áreas que codificas.

Consejo 20

Usa balas trazadoras para encontrar el objetivo

De hecho, dada la complejidad de la configuración de proyectos actual, con enjambres de dependencias y herramientas externas, las viñetas de seguimiento se vuelven aún más importantes. Para nosotros, la primera viñeta de rastreo es simplemente crear el proyecto, agregar un "¡Hola mundo!" y asegurarse de que se compile y se ejecute. Luego buscamos áreas de incertidumbre en la aplicación general y agregamos el esqueleto necesario para que funcione.

Echa un vistazo al siguiente diagrama. Este sistema tiene cinco capas arquitectónicas. Tenemos algunas dudas sobre cómo se integrarían, por lo que buscamos una característica simple que nos permita ejercitarnos juntos. La línea diagonal muestra la ruta que toma la característica a través del código. Para que funcione, solo tenemos que implementar las áreas sólidamente sombreadas en cada capa: las cosas con los garabatos se harán más tarde.

imágenes/capa-diagrama-trazador.png



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 interfaz de usuario del cliente, escrita en un idioma aleatorio A, usaba un conjunto de bibliotecas escritas en un idioma diferente 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 interfaz de usuario.

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 enumeraba todas las filas de una tabla, pero demostró que la interfaz de usuario podía comunicarse con las bibliotecas, las bibliotecas podían serializar y deserializar una consulta, y el servidor podía generar SQL a partir del resultado. 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 (consulte el
Tema 52, Deleite a sus usuarios), sus usuarios

saben 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 La hoja de papel más desalentadora es la 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.

Tiene una plataforma de integración Como el sistema está conectado de extremo a extremo, tiene un entorno al que puede agregar nuevas piezas 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.

Tiene algo que demostrar Los patrocinadores del proyecto y los altos mandos tienden a querer ver demostraciones en los momentos más inconvenientes. Con el código rastreador, siempre tendrás algo que mostrarles.

Tiene una mejor idea del progreso En el desarrollo de un código trazador, 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 Aciertan OBJETIVO

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. Así que cambie lo que tenga para acercarlo al objetivo y agradezca haber utilizado una metodología de desarrollo ajustada; 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 de forma rápida y económica. 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 RASTREADOR VERSUS 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, usted

Deseche todo lo que haya juntado cuando probó el concepto y vuelva a codificarlo correctamente usando las lecciones que ha 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 empaquetado óptimo son muy complejos.

Podría crear un prototipo de una interfaz de usuario para sus usuarios finales en una herramienta de interfaz de usuario. 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 Python, 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, se agrega a este marco con nuevas funciones,

completando rutinas cortadas. 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

- Tema 13, Prototipos y Post-it Notes
- Tema 27, No superes tus faros
- Tema 40, Refactorización
- Tema 49, Equipos Pragmáticos
- Tema 50, Los cocos no lo cortan
- Tema 51, Kit de inicio pragmático
- Tema 52, Deleite a sus usuarios



Tema 13

Prototipos y Post-it Notes

Muchas industrias 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. La gente de la vieja escuela podría usar 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. Los menos románticos harán su modelado en una pantalla de computadora o en realidad virtual, reduciendo 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 interfaz de usuario, 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 interfaz de usuario bastante pobre, o tal vez incluso sin interfaz de usuario.

Pero si se encuentra en un entorno en el que no puede renunciar a los detalles, debe preguntarse si realmente está construyendo un prototipo. Quizás un estilo de desarrollo con viñetas sería más apropiado en este caso (consulte el Tema 12, Viñetas).

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 los datos externos
- Herramientas o componentes de terceros
- Problemas de desempeño
- 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.

Consejo 21

Prototipo para aprender

CÓMO UTILIZAR PROTOTIPOS

Al construir un prototipo, ¿qué detalles puedes ignorar?

Exactitud

Es posible que pueda utilizar datos ficticios cuando corresponda.

Compleitud El

prototipo puede funcionar solo en un sentido muy limitado, tal vez con solo una pieza preseleccionada de datos de entrada y un elemento de menú.

Robustez

Es probable que la 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

El código del prototipo no debería tener muchos comentarios o documentación (aunque puede producir montones de documentación como resultado de su experiencia con el prototipo).

Los prototipos pasan por alto los detalles y se centran en aspectos específicos del sistema que se está considerando, por lo que es posible que desee implementarlos utilizando un lenguaje de secuencias de comandos de alto nivel, más alto que el resto.

del proyecto (tal vez un lenguaje como Python o Ruby), ya que estos lenguajes pueden salirse de su camino. Puede optar por continuar desarrollando en el lenguaje utilizado para el prototipo, o puede cambiar; después de todo, vas a tirar el prototipo lejos de todos modos.

Para crear prototipos de interfaces de usuario, utilice una herramienta que le permita concentrarse en la apariencia o las interacciones sin preocuparse por el código o el marcado.

Los lenguajes de secuencias de comandos también funcionan bien como "pegamento" para combinar piezas de bajo nivel en nuevas combinaciones. 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:

- ¿Las responsabilidades de las áreas principales están bien definidas y son apropiadas?
- ¿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?
- ¿Cada módulo tiene una ruta de acceso a los datos que necesita durante la ejecución? ¿Tiene ese acceso cuando 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 NO UTILIZAR PROTOTIPOS

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

Es fácil dejarse engañar por la aparente integridad de un prototipo demostrado, y los patrocinadores o la 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 la viñeta trazadora. Terminará con un marco sólido en el que basar el desarrollo futuro.

Los prototipos utilizados correctamente pueden ahorrarle una gran cantidad de tiempo, dinero y molestias 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

- Tema 12, Viñetas trazadoras
- Tema 14, Idiomas de dominio
- Tema 17, Juegos de conchas
- Tema 27, No superes tus faros
- Tema 37, Escucha tu cerebro de lagarto
- Tema 45, El pozo de requisitos
- Tema 52, Deleite a sus usuarios

EJERCICIOS

Ejercicio 3 (respuesta posible)

A Marketing le gustaría sentarse y hacer una lluvia de ideas sobre algunos diseños de páginas web con usted. 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?



Tema 14

Idiomas de dominio

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

Luis Wittgenstein

Los lenguajes informáticos influyen en cómo piensa sobre un problema y cómo piensa sobre la comunicación. Cada idioma viene con una lista de características: palabras de moda como tipo estático versus dinámico, enlace temprano versus tardío, funcional versus OO, modelos de herencia, mixins,

macros, todo lo cual puede sugerir u ocultar ciertas soluciones.

Diseñar una solución con C++ en mente producirá resultados diferentes a los de una solución basada en el estilo de pensamiento de Haskell, 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 utilizando el vocabulario del dominio de la aplicación ([consulte Mantener un glosario](#)). En algunos casos, los programadores pragmáticos pueden pasar al siguiente nivel y realmente programar usando el vocabulario, la sintaxis y la semántica (el lenguaje) del dominio.

Consejo 22

Programa cerca del dominio del problema

ALGUNOS IDIOMAS DE DOMINIO DEL MUNDO REAL

Veamos algunos ejemplos en los que la gente ha hecho precisamente eso.

Respec

RSpec ^[19] es una biblioteca de prueba para Ruby. Inspiró versiones para la mayoría de los otros idiomas modernos. Una prueba en RSpec pretende reflejar el comportamiento que espera de su código.

```
describe BowlingScore
  hazlo "total 12 si anotas 3 cuatro veces"
    do score = BowlingScore.new
      4.times { score.add_pins(3) }
      expect(score.total).to eq(12)
    end
  end
```

Pepino

Cucumber ^[20] es una forma neutral de lenguaje de programación de especificar pruebas. Las pruebas se ejecutan con una versión de Cucumber adecuada para el idioma que está utilizando. Para admitir la sintaxis similar al lenguaje natural, también debe escribir comparadores específicos que reconozcan frases y extraigan parámetros para las pruebas.

Característica: Puntuación

Fondo:

Dada una tarjeta de puntuación vacía

Escenario: lanzar muchos 3s

Dado tiro un 3

y tiro un 3

y tiro un 3

y tiro un 3

Entonces el puntaje debe ser 12

Las pruebas de Cucumber estaban destinadas a ser leídas por los clientes del software (aunque eso sucede con bastante poca frecuencia en la práctica; el siguiente aparte considera por qué podría ser así).

¿Por qué muchos usuarios comerciales no leen las características de Cucumber?

Una de las razones por las que el enfoque clásico de recopilación de requisitos, diseño, código y envío no funciona es que está anclado en el concepto de que sabemos cuáles son los requisitos. Pero rara vez lo hacemos. Los usuarios de su negocio tendrán una vaga idea de lo que quieren lograr, pero no saben ni se preocupan por los detalles. Eso es parte de nuestro valor: nosotros intuir la intención y convertirla en código.

Entonces, cuando obliga a una persona de negocios a firmar un documento de requisitos, o hace que acepte un conjunto de características de Cucumber, está haciendo el equivalente a hacer que revisen la ortografía en un ensayo escrito en sumerio. Harán algunos cambios aleatorios para salvar la cara y lo firmarán para sacarte de su oficina.

Sin embargo, dales un código que se ejecute y podrán jugar con él. Ahí es donde surgirán sus verdaderas necesidades.

Rutas del Fénix

Muchos marcos web tienen una función de enrutamiento, asignando solicitudes HTTP entrantes a funciones de controlador en el código. Aquí hay un ejemplo de Phoenix. [21]

```
scope "/", HelloPhoenix do
  pipe_through :browser # Usar la pila de navegador predeterminada

  obtener "/", PageController, :indexar
  recursos "/usuarios", final de
  UserController
```

Esto dice que las solicitudes que comienzan con "/" se ejecutarán a través de una serie de filtros apropiados para los navegadores. La función `de índice` en el módulo `PageController` manejará una solicitud a "/" . `UsersController` implementa las funciones necesarias para administrar un recurso accesible a través de la url / `usuarios`.

Ansible

Ansible es^[22]una herramienta que configura el software, generalmente en un grupo de servidores remotos. Para ello, lee una especificación que usted proporciona y luego hace lo que sea necesario en los servidores para que reflejen esa especificación. La especificación se puede escribir en YAML^[23],un lenguaje que construye estructuras de datos a partir de texto.

descripciones:

- nombre: instalar nginx apt: nombre=estado nginx=último
- nombre: asegúrese de que nginx se esté ejecutando (y habilítelo en el arranque) servicio: nombre = estado de nginx = iniciado habilitado = sí
- nombre: escribir la plantilla del archivo de configuración de nginx: src=templates/nginx.conf.j2 dest=/etc/nginx/nginx.conf notificar: - reiniciar nginx

Este ejemplo garantiza que la última versión de nginx esté instalada en mis servidores, que se inicie de forma predeterminada y que use un archivo de configuración que usted proporcionó.

CARACTERÍSTICAS DE LOS IDIOMAS DE DOMINIO

Veamos estos ejemplos más de cerca.

RSpec y el enrutador Phoenix están escritos en sus idiomas principales (Ruby y Elixir). Emplean un código bastante tortuoso, incluida la metaprogramación y las macros, pero en última instancia, se compilan y ejecutan como código normal.

Las pruebas de Cucumber y las configuraciones de Ansible están escritas en sus propios idiomas. Una prueba de Cucumber se convierte en código para ejecutar o en una estructura de datos, mientras que las especificaciones de Ansible siempre se convierten en una estructura de datos que Ansible mismo ejecuta.

Como resultado, RSpec y el código del enrutador están integrados en el código que ejecuta: son verdaderas extensiones del vocabulario de su código. Cucumber y Ansible se leen por código y se convierten en alguna forma que el código pueda usar.

Llamamos a RSpec y al enrutador ejemplos de lenguajes de dominio interno , mientras que Cucumber y Ansible usan lenguajes externos .

COMPENSACIONES ENTRE INTERNO Y EXTERNO IDIOMAS

En general, un idioma de dominio interno puede aprovechar las características de su idioma anfitrión: el idioma de dominio que crea es más poderoso y ese poder es gratuito. Por ejemplo, podría usar algún código Ruby para crear un montón de pruebas RSpec automáticamente. En este caso podemos probar puntuaciones donde no hay repuestos ni strikes:

```
describe BowlingScore do
  (0..4).each do |pins|
    (1..20).cada uno hace |
      lanza| objetivo = bolos * lanzamientos

    "totaliza # {objetivo} si anota #{pins} #{lanzamientos } "
    veces " . final final final
```

Son 100 pruebas que acabas de escribir. Tómate el resto del día libre.

La desventaja de los lenguajes de dominio interno es que estás sujeto a la sintaxis y la semántica de ese lenguaje. Aunque algunos lenguajes son notablemente flexibles en este sentido, aún se ve obligado a comprometerse entre el lenguaje que desea y el lenguaje que puede implementar.

En última instancia, lo que se le ocurra debe seguir siendo una sintaxis válida en su idioma de destino. Idiomas con macros (como

Elixir, Clojure y Crystal) le brinda un poco más de flexibilidad, pero en última instancia, la sintaxis es la sintaxis.

Los idiomas externos no tienen tales restricciones. Siempre que pueda escribir un analizador para el idioma, está listo para comenzar. A veces, puede usar el analizador de otra persona (como lo hizo Ansible al usar YAML), pero luego vuelve a hacer un compromiso.

Escribir un analizador probablemente signifique agregar nuevas bibliotecas y posiblemente herramientas a su aplicación. Y escribir un buen analizador no es un trabajo trivial. Pero, si se siente valiente, puede buscar generadores de analizadores como bison o ANTLR, y marcos de análisis como los muchos analizadores PEG que existen.

Nuestra sugerencia es bastante simple: no se esfuerce más de lo que ahorra. Escribir un lenguaje de dominio agrega algún costo a su proyecto, y deberá estar convencido de que hay ahorros compensatorios (potencialmente a largo plazo).

En general, utilice lenguajes externos estándar (como YAML, JSON o CSV) si puede. Si no, mira los idiomas internos.

Recomendamos usar lenguajes externos solo en los casos en que los usuarios de su aplicación escriban su idioma.

UN LENGUAJE DE DOMINIO INTERNO BARATO

Finalmente, hay un truco para crear idiomas de dominio interno si no le importa que se filtre la sintaxis del idioma anfitrión. No hagas un montón de metaprogramación. En su lugar, simplemente escriba funciones para hacer el trabajo. De hecho, esto es más o menos lo que hace RSpec:

```
describe BowlingScore
  hazlo "total 12 si anotas 3 cuatro veces"
    do score = BowlingScore.new
      4.times { score.add_pins(3) }
```

```
esperar(puntaje.total).to  
eq(12) fin fin
```

En este código, [describe](#), [it](#), [expect](#), [to](#) y [eq](#) son solo métodos de Ruby.

Hay un poco de plomería detrás de escena en términos de cómo se pasan los objetos, pero todo es solo código. Exploraremos eso un poco en los ejercicios.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 8, [La esencia del buen diseño](#)
- Tema 13, [Prototipos y Post-it Notes](#)
- Tema 32, [Configuración](#)

RETOS

- ¿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

Ejercicio 4 ([respuesta posible](#))

Queremos implementar un minilenguaje para controlar un sistema simple de gráficos de tortugas. El lenguaje consta de comandos de una sola letra, algunos seguidos de un solo número. Por ejemplo, la siguiente entrada dibujaría un rectángulo:

P 2 # seleccionar pluma 2
D # pluma hacia abajo
W 2 # dibuja al oeste 2cm
N 1 # luego norte 1
E 2 # luego este 2
S 1 # luego de regreso al sur
U # bolígrafo arriba

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

Ejercicio 5 (respuesta posible)

En el ejercicio anterior, implementamos un analizador para el lenguaje de dibujo: era un lenguaje de dominio externo. Ahora impleméntalo de nuevo como un lenguaje interno. No haga nada inteligente: simplemente escriba una función para cada uno de los comandos. Es posible que deba cambiar los nombres de los comandos a minúsculas y tal vez envolverlos dentro de algo para proporcionar algo de contexto.

Ejercicio 6 (respuesta posible)

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

Ejercicio 7 (respuesta posible)

Implemente un analizador para la gramática BNF en el ejercicio anterior utilizando un generador de analizador PEG en el idioma de su elección. La salida debe ser un número entero que contenga el número de minutos después de la medianoche.

Ejercicio 8 (respuesta posible)

Implemente el analizador de tiempo utilizando un lenguaje de secuencias de comandos y expresiones regulares.



Tema 15

Estimación

La Biblioteca del Congreso en Washington, DC, actualmente tiene alrededor de 75 terabytes de información digital en línea. ¡Rápido! ¿Cuánto tiempo llevará enviar toda esa información a través de una red de 1 Gbps? ¿Cuánto espacio de almacenamiento necesitará para un millón de nombres y direcciones? ¿Cuánto se tarda en comprimir 100Mb de texto? ¿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 conexión de red a S3", podrá saber intuitivamente si esto es práctico. Cuando esté codificando, podrá saber qué subsistemas necesitan optimizarse y cuáles se pueden dejar solos.

Consejo 23

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.

¿CUÁN EXACTO ES LO SUFFICIENTEMENTE EXACTO?

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?

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
3–6 semanas	Semanas
8–20 semanas	Meses
20+ semanas	Piensa 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

para entregar una estimación de “unos seis meses” .

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

DE DÓNDE VIENEN LAS ESTIMACIONES?

Todas las estimaciones se basan en modelos del problema. Pero antes de profundizar demasiado en las técnicas de construcción de modelos, 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. 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.

Comprenda lo que se le pregunta La primera parte de cualquier ejercicio de estimación es desarrollar una comprensión de lo que se le 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 básico. 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 utiliza su organización

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 podría hacer en aproximadamente la mitad del tiempo y solo pierde una característica".

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.

Divida 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 valor único que se agrega 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. Esperas 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 SSD y 32 GB de memoria, y un segundo con 16 GB de memoria" . (Observe cómo "tres cuartos de segundo" transmite una sensación de precisión diferente a 750 ms).

Durante la fase de cálculo, obtienes respuestas que parecen extrañas. No se apresure a descartarlos. Si tu aritmética

es correcto, su comprensión del problema o su modelo probablemente sea incorrecto. Esta es información valiosa.

Lleve un registro de su destreza en la

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. 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.

CALENDARIOS DE PROYECTOS ESTIMADOS

Normalmente se le pedirá que calcule cuánto tiempo llevará algo. Si ese “algo” es complejo, la estimación puede ser muy difícil de producir. En esta sección, veremos dos técnicas para reducir esa incertidumbre.

pintando el misil

“¿Cuánto tiempo tomará pintar la casa?”

“Bueno, si todo va bien, y esta pintura tiene la cobertura que dicen, podría tardar tan solo 10 horas. Pero eso es poco probable: supongo que una cifra más realista está más cerca de las 18 horas. Y, por supuesto, si el clima empeora, eso podría aumentarlo a 30 o más” .

Así es como la gente estima en el mundo real. No con un solo

número (a menos que los obligues a darte uno) pero con una variedad de escenarios.

Cuando la Marina de los EE. UU. necesitó planificar el proyecto del submarino Polaris, adoptaron este estilo de estimación con una metodología que llamaron Técnica de revisión de evaluación del programa, o PERT.

Cada tarea PERT tiene una estimación optimista, más probable y pesimista. Las tareas se organizan en una red de dependencia y, luego, utiliza algunas estadísticas simples para identificar los mejores y peores momentos probables para el proyecto en general.

Usar un rango de valores como este es una excelente manera de evitar una de las causas más comunes de error de estimación: llenar un número porque no está seguro. En cambio, las estadísticas detrás de PERT distribuyen la incertidumbre por usted y le brindan mejores estimaciones de todo el proyecto.

Sin embargo, no somos grandes fans de esto. Las personas tienden a producir gráficos del tamaño de una pared de todas las tareas de un proyecto e implícitamente creen que, solo porque usaron una fórmula, tienen una estimación precisa. Lo más probable es que no lo hagan, porque nunca lo han hecho antes.

Comiéndose el

elefante 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 con porciones muy finas de funcionalidad:

- Consultar requisitos

- Analice el riesgo (y priorice los elementos más riesgosos antes)
- 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 lo concrete como parte del plan inicial; sin embargo, 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 de la primera iteración.

Según esa experiencia, puede refinar su suposición inicial sobre la cantidad de iteraciones y lo que se puede incluir en cada una. El refinamiento mejora cada vez más y la confianza en el cronograma crece junto con él. Este tipo de estimación a menudo se realiza durante la revisión del equipo al final de cada ciclo iterativo.

Así también dice el viejo chiste que hay que comerse un elefante: un bocado a la vez.

Consejo 24

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 SOLICITA UN PRESUPUESTO

Usted dice "Me pondré en contacto con usted".

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

- Tema 7, ¡Comuníquese!
- Tema 39, Algoritmo de velocidad

RETOS

- 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

Ejercicio 9 (respuesta posible)

Se le pregunta "¿Qué tiene un mayor ancho de banda: una conexión de red de 1 Gbps o una persona que camina entre dos computadoras con un dispositivo de almacenamiento de 1 TB en su bolsillo?" ¿Qué restricciones pondrá en su respuesta para garantizar que el alcance de su respuesta es correcto? (Por ejemplo, podría decir que se ignora el tiempo necesario para acceder al dispositivo de almacenamiento).

Ejercicio 10 (respuesta posible)

Entonces, ¿cuál tiene el mayor ancho de banda?

notas al pie

[13] Parafraseando la vieja canción de Arlen/Mercer...

[14] O, tal vez, para mantener la cordura, cada 10 veces...

[15] <https://github.com/OAI/OpenAPI-Especificación>

[decisión] 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.

[17] De hecho, este libro está escrito en Markdown y escrito directamente desde Markdown fuente.

[18] 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?

[19] <https://rspec.info>

[20] <https://pepino.io/>

[21] <https://phoenixframework.org/>

[22] <https://www.ansible.com/>

[23] <https://yaml.org/>

Capítulo 3

Las herramientas básicas

Cada fabricante 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 sujetla la herramienta. En este punto, las herramientas se convierten en conductos desde el cerebro del fabricante 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, plantillas de cola de milano, todas maravillosas piezas de tecnología. Pero puedes apostar a que estarán más felices 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

sabe cómo usarlos, más productivo puede 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 fabricante, 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 lo hubiera ayudado. Deje que la necesidad impulse sus adquisiciones.

Muchos programadores nuevos cometen 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. Debe sentirse cómodo 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. Al igual que con cualquier buena discusión sobre herramientas, comenzaremos (en el Tema 16, El poder del texto sin formato) observando 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 el Tema 17, Shell Games. Ahora que tenemos material y un banco para trabajar, pasaremos a la herramienta que probablemente usará más que cualquier otra, su editor. En el Tema 18, Power Editing, sugeriremos formas de hacerlo más eficiente.

Para asegurarnos de que nunca perdemos nada de nuestro preciado trabajo, siempre debemos usar un sistema de Control de versiones del Tema 19, incluso para cosas personales como recetas o notas. Y, dado que Murphy era realmente un optimista después de todo, no puede ser un gran programador hasta que se vuelve muy hábil en el Tema 20, Depuración.

Necesitarás un poco de pegamento para unir gran parte de la magia.

Discutimos algunas posibilidades en el Tema 21, Manipulación de texto.

Finalmente, la tinta más pálida sigue siendo mejor que el mejor recuerdo.

Lleve un registro de sus pensamientos y su historial, como lo describimos en el Tema 22, Diarios de Ingeniería.

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.



Tema 16

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 el 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.

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.

¿QUÉ ES EL TEXTO SIMPLE?

El texto sin formato se compone de caracteres imprimibles en un formato que transmite información. Puede ser tan simple como una lista de compras:

*leche
*lechuga
*café

o tan complejo como la fuente de este libro (sí, está en texto sin formato, para disgusto del editor, que quería que usáramos un procesador de textos).

La parte de la información es importante. El siguiente no es texto sin formato útil:

hlj;uijn bfjxrctvh jkni'pio6p7gu;vh bjxrdi5rgvhj

Tampoco es esto:

Campo19=467abe

El lector no tiene idea de cuál puede ser el significado de [467abe](#).

Nos gusta que nuestro texto sin formato sea comprensible para los humanos.

Consejo 25

Mantener el conocimiento en texto sin formato

EL PODER DEL TEXTO

Texto sin formato no significa que el texto no esté estructurado; HTML, JSON, YAML, etc. son todos texto sin formato. También lo son la mayoría de los protocolos fundamentales de la red, como HTTP, SMTP, IMAP, etc. Y eso es por algunas buenas razones:

- Seguro contra la obsolescencia
- Aproveche las herramientas existentes
- Pruebas más fáciles

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

Puede analizar un archivo de este tipo con sólo 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 se le proporcione.
[24] 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 comprensible por humanos.

Mientras estamos en eso, **FIELD10** tampoco ayuda mucho. Algo como

```
<SEGURIDAD-SOCIAL-NO>123-45-6789</SEGURIDAD-SOCIAL-NO>
```

hace que el ejercicio sea una obviedad y asegura que los datos

sobrevivir a cualquier proyecto que lo creó.

Aproveche

Prácticamente todas las herramientas del universo informático, desde los sistemas de control de versiones hasta los editores y las herramientas de línea de comandos, pueden operar en texto sin formato.

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 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.

El texto sin formato también es más fácil de buscar. Si no puede recordar qué archivo de configuración administra las copias de seguridad de su sistema, un rápido `grep -r backup /etc` debería decírselo.

Por ejemplo, suponga que tiene una implementación de producción de una aplicación grande con un archivo de configuración complejo específico del sitio.

Si este archivo está en texto sin formato, puede colocarlo bajo un sistema de control de versiones (consulte el Tema 19, Control de versiones), para que mantenga automáticamente un historial de todos los cambios. Las herramientas de comparación de archivos como `diff` y `fc` le permiten ver de un vistazo qué cambios se han realizado, mientras que `sum` le permite generar una suma de verificación para monitorear el archivo en busca de modificaciones accidentales (o maliciosas).

Pruebas más

sencillas Si utiliza texto sin formato para crear datos sintéticos para impulsar las pruebas del sistema, entonces es muy sencillo 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 manera trivial con comandos de shell o un script simple.

MÍ NIMO COMÚ N DENOMINADOR

Incluso en el futuro de los agentes inteligentes basados en cadenas de bloques 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

- Tema 17, Juegos de conchas
- Tema 21, Manipulación de texto
- Tema 32, Configuración

RETOS

- 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.
 - Traduzca ese formato a un formato de texto sin formato usando XML o JSON.
 - 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? Cual

formulario fue más fácil de modificar? ¿Qué pasa con la conversión de datos existentes?



Todo carpintero necesita un banco de trabajo bueno, sólido y confiable, en algún lugar para sostener las piezas de trabajo a una altura conveniente mientras se les da forma. El banco de trabajo se convierte en el centro del taller de carpintería, el fabricante 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 y escribir correos electrónicos y construir e implementar su proyecto 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 podrás automatizar

tareas comunes, o use todo el poder de las herramientas disponibles para usted. Y no podrá combinar sus herramientas para crear macroherramientas personalizadas. Una ventaja de las GUI es WYSIWYG: lo que ve es lo que obtiene. La desventaja es WYSIAYG: lo que ve 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, necesita ir más allá del modelo. Los programadores pragmáticos no solo cortan código, desarrollan modelos de objetos, escriben documentación o automatizan el proceso de compilación: hacemos todas 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.

Consejo 26

Usa el poder de los proyectos 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":

sh/paquetes.sh

```
grep '^importar' *.java |  
sed -e's/.importar */' -e's/;.*$/ /'  
ordenar -u > lista
```

Si no ha pasado mucho tiempo explorando las capacidades del shell de comandos en los sistemas que usa, esto podría aparecer

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.

UNA CONCHA PROPIA

De la misma manera que un carpintero personalizará su espacio de trabajo, un desarrollador debe personalizar su estructura. Por lo general, esto también implica cambiar la configuración del programa de terminal que utiliza.

Los cambios comunes incluyen:

- Configuración de temas de color. Se pueden pasar muchas, muchas horas probando cada tema que está disponible en línea para su caparazón en particular.
- Configuración de un aviso. El aviso que le dice que el shell está listo para que escriba un comando se puede configurar para mostrar casi cualquier información que pueda desear (y un montón de cosas que nunca querría). Las preferencias personales lo son todo aquí: nos gustan las indicaciones simples, con un nombre de directorio actual abreviado y el estado de control de versión junto con la hora.
- Alias y funciones de shell. Simplifique su flujo de trabajo convirtiendo los comandos que usa mucho en simples alias. Tal vez actualice regularmente su caja de Linux, pero nunca puede recordar si actualiza y actualiza, o actualiza y actualiza. Crear un alias:

```
alias apt-up='sudo apt-get update && sudo apt-get upgrade'
```

Tal vez haya eliminado archivos accidentalmente con el comando `rm` solo una vez con demasiada frecuencia. Escriba un alias para que siempre aparezca en el futuro:

```
alias rm ='rm -iv'
```

- Finalización de comandos. La mayoría de los shells completarán los nombres de los comandos y archivos: escriba los primeros caracteres, presione el tabulador y completará lo que pueda. Pero puede llevar esto mucho más lejos, configurando el

shell para reconocer el comando que está ingresando y ofrecer finalizaciones específicas del contexto. Algunos incluso personalizan la finalización según el directorio actual.

Pasarás mucho tiempo viviendo en una de estas conchas. Sé como un cangrejo ermitaño y conviértelo en tu propio hogar.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 13, [Prototipos y Post-it Notes](#)
- Tema 16, [El poder del texto sin formato](#)
- Tema 21, [Manipulación de texto](#)
- Tema 30, [Transformación de la programación](#)
- Tema 51, [Kit de inicio pragmático](#)

RETOS

- ¿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.



Tema 18

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.

En la primera edición de este libro, recomendamos usar un solo editor para todo: código, documentación, memos, administración del sistema, etc. Hemos suavizado un poco esa posición.

Estamos encantados de que utilice tantos editores como desee. Nos gustaría que trabaje para lograr la fluidez en cada uno.

Consejo 27

Lograr la fluidez del editor

¿Por qué es esto un gran problema? ¿Estamos diciendo que ahorrará mucho tiempo? De hecho, sí: en el transcurso de un año, podría ganar una semana adicional si hace que su edición sea solo un 4% más eficiente y edita durante 20 horas a la semana.

Pero ese no es el beneficio real. No, la mayor ventaja es que, al volverse fluido, ya no tiene que pensar en la mecánica de la edición. La distancia entre pensar algo y que aparezca en un menú desplegable del búfer del editor. Tus pensamientos fluirán y tu programación se beneficiará. (Si alguna vez le ha enseñado a alguien a conducir, comprenderá la diferencia entre alguien que tiene que pensar en cada acción que realiza y un conductor más experimentado que controla el

coche instintivamente.)

¿QUÉ SIGNIFICA “FLUIDO” ?

¿Qué cuenta como hablar con fluidez? Aquí está la lista de desafíos:

- Al editar texto, mueva y realice selecciones por carácter, palabra, línea y párrafo.
- Al editar código, muévase por varias unidades sintácticas (coincidencia de delimitadores, funciones, módulos,...).
- Reindentar el código después de los cambios.
- Comenta y descomenta bloques de código con un solo comando.
- Deshacer y rehacer cambios.
- Divida la ventana del editor en varios paneles y navegue entre ellos.
- Navegue a un número de línea en particular.
- Ordenar las líneas seleccionadas.
- Busque cadenas y expresiones regulares y repita las búsquedas anteriores.
- Cree temporalmente múltiples cursores basados en una selección o en una coincidencia de patrón y edite el texto en cada uno en paralelo.
- Mostrar errores de compilación en el proyecto actual.
- Ejecute las pruebas del proyecto actual.

¿Puedes hacer todo esto sin usar un mouse/trackpad?

Podría decir que su editor actual no puede hacer algunas de estas cosas.

¿Quizás es hora de cambiar?

AVANZANDO HACIA LA FLUIDEZ

Dudamos que haya más de un puñado de personas que conozcan todos los comandos en un editor poderoso en particular. Tampoco esperamos que lo hagas. En su lugar, sugerimos un enfoque más pragmático: aprende los comandos que te hacen la vida más fácil.

La receta para esto es bastante simple.

Primero, mírate a ti mismo mientras estás editando. Cada vez que te encuentres haciendo algo repetitivo, adquiere el hábito de pensar “debe haber una mejor manera”. Entonces encuéntralo.

Una vez que haya descubierto una característica nueva y útil, ahora necesita instalarla en su memoria muscular, para que pueda usarla sin pensar. La única forma que conocemos de hacerlo es a través de la repetición. Busque conscientemente oportunidades para usar su nuevo superpoder, idealmente muchas veces al día. Después de una semana más o menos, descubrirá que lo usa sin pensar.

Cómo hacer crecer su

editor La mayoría de los potentes editores de código se construyen alrededor de un núcleo básico que luego se aumenta a través de extensiones. Muchos se suministran con el editor y otros se pueden agregar más tarde.

Cuando se encuentre con alguna limitación aparente del editor que está utilizando, busque una extensión que haga el trabajo.

Lo más probable es que no sea el único que necesite esa capacidad y, si tiene suerte, alguien más habrá publicado su solución.

Lleva esto un paso más allá. Profundice en el lenguaje de extensión de su editor. Averigua cómo usarlo para automatizar algunas de las cosas repetitivas que haces. A menudo, solo necesitará una línea o dos de código.

A veces puede ir más allá y se encontrará escribiendo una extensión completa. Si es así, publícalo: si tú lo necesitas, otras personas también lo harán.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 7, ¡Comuníquese!

RETOS

- No más autorrepetición.

Todo el mundo lo hace: tienes que borrar la última palabra que escribiste, así que presionas la **tecla de retroceso** y esperas a que se active la repetición automática.

De hecho, apostamos a que tu cerebro ha hecho esto tantas veces que puedes juzgar exactamente cuándo hacerlo. suelte la llave.

Por lo tanto, desactive la repetición automática y, en su lugar, aprenda las secuencias de teclas para mover, seleccionar y eliminar por caracteres, palabras, líneas y bloques.

- Este va a doler.

Pierde el mouse / trackpad. Durante una semana entera, edite usando solo el teclado. Descubrirá un montón de cosas que no puede hacer sin señalar y hacer clic, así que ahora es el momento de aprender. Tome notas (recomendamos ir a la vieja escuela y usar lápiz y papel) de las secuencias clave que aprende.

Recibirá un golpe de productividad durante unos días. Pero, a medida que aprenda a hacer cosas sin mover las manos de la posición de inicio, descubrirá que su edición se vuelve más rápida y fluida que nunca.

- Busca integraciones. Mientras escribía este capítulo, Dave se preguntó si podría obtener una vista previa del diseño final (un archivo PDF) en un búfer del editor. Una descarga más tarde, el diseño se encuentra junto al texto original, todo en el editor. Mantenga una lista de las cosas que le gustaría traer a su editor, luego búsquelas.
- Algo más ambicioso, si no puede encontrar un complemento o extensión que haga lo que quiere, escriba uno. Andy es aficionado a crear complementos Wiki personalizados basados en archivos locales para sus editores favoritos. si no puedes

¡encuéntralo, constrúyelo!



Tema 19

Control de versiones

El progreso, lejos de consistir en un cambio, depende de la retención. Aquellos quien no puede recordar el pasado están condenados a repetirlo.

Jorge Santayana, *Vida de Razón*

Una de las cosas importantes que buscamos en una interfaz de usuario es la tecla de deshacer , un 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 una versión

sistema de control (VCS): es una tecla de deshacer gigante : una máquina del tiempo para todo el proyecto que puede devolverlo a esos días felices de la semana pasada, cuando el código realmente se compiló y ejecutó.

Para muchas personas, ese es el límite de su uso de VCS. Esas personas se están perdiendo un mundo mucho más grande de colaboración, canales de implementación, seguimiento de problemas e interacción general del equipo.

Así que echemos un vistazo a VCS, primero como un repositorio de cambios y luego como un lugar de encuentro central para su equipo y su código.

Los directorios compartidos NO son control de versiones

Todavía nos encontramos con equipos ocasionales que comparten los archivos fuente de sus proyectos a través de una red: ya sea internamente o utilizando algún tipo de almacenamiento en la nube.

Esto no es viable.

Los equipos que hacen esto constantemente arruinan el trabajo de los demás, pierden cambios, rompen construcciones y se pelean a puñetazos en el estacionamiento. Es como escribir código concurrente con datos compartidos y sin mecanismo de sincronización. Utilice el control de versiones.

¡Pero hay más! Algunas personas usan el control de versiones y mantienen su repositorio principal en una red o en una unidad en la nube. Razonan que esto es lo mejor de ambos mundos: se puede acceder a sus archivos desde cualquier lugar y (en el caso del almacenamiento en la nube) se realiza una copia de seguridad fuera del sitio.

Resulta que esto es aún peor y te arriesgas a perderlo todo. El software de control de versiones utiliza un conjunto de archivos y directorios que interactúan. Si dos instancias realizan cambios simultáneamente, el estado general puede corromperse y no se sabe cuánto daño se hará. Y a nadie le gusta ver llorar a los desarrolladores.

EMPIEZA EN LA FUENTE

Los sistemas de control de versiones realizan un seguimiento de cada cambio que realiza en su código fuente y documentación. 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 versiones hace mucho más que deshacer errores. Un buen VCS 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 VCS 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.

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

Finalmente, los sistemas de control de versiones permiten 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.

Consejo 28

Utilice siempre el control de versiones

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. Asegúrese de que todo esté bajo control de versiones: documentación, listas de números de teléfono, memorandos a los proveedores, archivos MAKE, procedimientos de compilación y lanzamiento, ese pequeño script de shell que ordena los archivos de registro, todo. Habitualmente utilizamos el control de versiones 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.

RAMIFICACIÓN

Los sistemas de control de versiones no solo mantienen un único historial de su proyecto. Una de sus características más poderosas y útiles es la forma en que le permiten aislar islas de desarrollo en cosas llamadas ramas. Puede crear una rama en cualquier punto del historial de su proyecto, y cualquier trabajo que realice en esa rama se aislará de todas las demás ramas. En algún momento en el futuro, puede fusionar la rama en la que está trabajando con otra rama, de modo que la rama de destino ahora contenga los cambios que realizó en su rama. Varias personas pueden incluso estar trabajando en un

rama: en cierto modo, las ramas son como pequeños proyectos clonados.

Uno de los beneficios de las sucursales es el aislamiento que te brindan. Si desarrolla la función A en una rama y un compañero de equipo trabaja en la función B en otra, no interferirán entre sí.

Un segundo beneficio, que puede resultar sorprendente, es que las sucursales suelen estar en el corazón del flujo de trabajo del proyecto de un equipo.

Y aquí es donde las cosas se ponen un poco confusas. Las ramas de control de versiones y la organización de pruebas tienen algo en común: ambas tienen miles de personas que te dicen cómo debes hacerlo. Y ese consejo en gran parte no tiene sentido, porque lo que realmente están diciendo es "esto es lo que funcionó para mí".

Por lo tanto, use el control de versiones en su proyecto y, si se encuentra con problemas de flujo de trabajo, busque posibles soluciones. Y recuerda revisar y ajustar lo que estás haciendo a medida que adquieres experiencia.

Un experimento mental

Vierta una taza entera de té (desayuno inglés, con un poco de leche) en el teclado de su computadora portátil. Lleve la máquina a la barra de personas inteligentes y pídale que frunzan el ceño. Compre una computadora nueva. Llévalo a casa.

¿Cuánto tiempo llevaría que la máquina volviera al mismo estado en el que se encontraba (con todas las claves SSH, la configuración del editor, la configuración del shell, las aplicaciones instaladas, etc.) en el punto en el que levantó por primera vez esa fatídica copa? Este fue un problema que uno de nosotros enfrentó recientemente.

Prácticamente todo lo que definía la configuración y el uso de la máquina original se almacenaba en el control de versiones, incluido:

- Todas las preferencias de usuario y dotfiles
- La configuración del editor
- La lista de software instalado usando Homebrew

- El script de Ansible utilizado para configurar aplicaciones
- Todos los proyectos actuales

La máquina fue restaurada al final de la tarde.

CONTROL DE VERSIÓN COMO CENTRO DE PROYECTO

Aunque el control de versiones es increíblemente útil en proyectos personales, realmente se destaca cuando se trabaja con un equipo. Y gran parte de este valor proviene de cómo aloja su repositorio.

Ahora, muchos sistemas de control de versiones no necesitan alojamiento. Están completamente descentralizados, y cada desarrollador coopera de igual a igual. Pero incluso con estos sistemas, vale la pena considerar tener un repositorio central, porque una vez que lo haga, puede aprovechar un montón de integraciones para facilitar el flujo del proyecto.

Muchos de los sistemas de repositorio son de código abierto, por lo que puede instalarlos y ejecutarlos en su empresa. Pero esa no es realmente su línea de negocio, por lo que recomendamos que la mayoría de las personas alojen con un tercero. Busque características como:

- Buena seguridad y control de acceso.
- Interfaz de usuario intuitiva
- La capacidad de hacer todo desde la línea de comandos también (porque es posible que deba automatizarlo)
- Construcciones y pruebas automatizadas
- Buen soporte para la fusión de sucursales (a veces llamadas solicitudes de extracción)
- Gestión de incidencias (idealmente integrada en compromisos y fusiones, por lo que

puede conservar las métricas)

- Buenos informes (una visualización similar a un tablero Kanban de problemas y tareas pendientes puede ser muy útil)
- Buenas comunicaciones de equipo: correos electrónicos u otras notificaciones en cambios, un wiki, etc.

Muchos equipos tienen su VCS configurado para que un impulso a una rama en particular construya automáticamente el sistema, ejecute las pruebas y, si tiene éxito, implemente el nuevo código en producción.

¿Suena aterrador? No cuando te das cuenta de que estás usando el control de versiones. Siempre puedes revertirlo.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 11, Reversibilidad
- Tema 49, Equipos Pragmáticos
- Tema 51, Kit de inicio pragmático

RETOS

- Saber que puede retroceder a cualquier estado anterior usando el VCS es una cosa, pero ¿realmente puede hacerlo? ¿Conoces los comandos para hacerlo correctamente? Aprenda ahora, no cuando ocurra un desastre y esté bajo presión.
- Dedique algún tiempo a pensar en recuperar el entorno de su propio portátil en caso de desastre. ¿Qué necesitarías para recuperar? Muchas de las cosas que necesita son solo archivos de texto. Si no están en un VCS (alojado fuera de su computadora portátil), encuentre una manera de agregarlos. Luego piense en las otras cosas: aplicaciones instaladas, configuración del sistema, etc. ¿Cómo puedes expresar todo eso en archivos de texto para que también se pueda guardar?

Un experimento interesante, una vez que haya hecho algún progreso, es encontrar una computadora vieja que ya no use y ver si su nuevo sistema se puede usar para configurarla.

- Explore conscientemente las características de su VCS actual y su proveedor de alojamiento que no está utilizando. Si su equipo no usa ramas de características, experimente presentándolas. Lo mismo con las solicitudes de extracción/fusión. Integración continua. Construir tuberías. Incluso despliegue continuo. Mire también las herramientas de comunicación del equipo: wikis, tableros Kanban y similares.
No tienes que usar nada de eso. Pero necesita saber lo que hace para que pueda tomar esa decisión.
- Use el control de versiones también para cosas que no sean del proyecto.



Tema 20

depuración

es algo doloroso
Para mirar tu propio
problema y saber
Que tu mismo y nadie
mas lo ha hecho

Sófocles, Ájax

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 del primer 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 lo anotó (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 —tal vez sea incluso 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 les dices que hagan, no necesariamente lo que quieras que hagan.

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 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.

Acepte el hecho de que la depuración es solo la resolución de problemas y actúe 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 solucionar el problema, no en la culpa.

Consejo 29

Solucionar 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

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:

Consejo 30

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 pensar realmente en

qué podría estar causando los síntomas que 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 desperdicie ni una sola neurona en el tren de pensamiento que comienza "pero eso no puede suceder" porque claramente puede suceder y sucedió.

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 comenzar a ver el error, asegúrese de que está trabajando en un código que se generó de manera limpia, sin advertencias. Rutinariamente establecemos niveles de advertencia del compilador lo más altos posible.

¡No tiene sentido perder el tiempo tratando de encontrar un problema que la computadora podría encontrar por usted! Tenemos que concentrarnos en los problemas más difíciles que tenemos entre manos.

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

La precisión en los informes de errores disminuye aún más cuando provienen de un tercero; es posible que deba observar al 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 pincelada 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. Debe hacer esto de manera sistemática (consulte Pruebas continuas y despiadadas).

ESTRATEGIAS DE DEPURACIÓN

Una vez que cree que sabe lo que está pasando, es hora de averiguar qué cree el programa que está pasando.

Reproducción de

errores No, nuestros errores no se están multiplicando realmente (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 solo comando. Es mucho más difícil corregir un error si tiene que pasar por 15 pasos para llegar al punto donde aparece el error.

Así que aquí está la regla más importante de depuración:

Consejo 31

Prueba fallida antes de corregir el código

A veces, al obligarte a aislar las circunstancias que muestran el error, incluso obtendrás una idea de cómo solucionarlo. El acto de escribir la prueba informa la solución.

CODIFICADOR EN TIERRA EXTRAÑA

Toda esta charla sobre aislar el error está bien, cuando se enfrenta a 50,000 líneas de código y un reloj en marcha, ¿qué puede hacer un codificador pobre?

Primero, mira el problema. ¿Es un accidente? Siempre es sorprendente cuando enseñamos cursos que involucran programación cuántos desarrolladores ven una excepción emergente en rojo e inmediatamente saltan al código.

Consejo 32

Leer el maldito mensaje de error

'nuf dijo.

Malos resultados

¿Y si no es un accidente? ¿Qué pasa si es sólo un mal resultado?

Ingresé allí con un depurador y use su prueba fallida para desencadenar el problema.

Antes que nada, asegúrese de que también está viendo el valor incorrecto en el depurador. Ambos desperdiciamos horas tratando de rastrear un error solo para descubrir que esta ejecución particular del código funcionó bien.

A veces, el problema es obvio: la tasa de interés es 4,5 y debería ser 0,045. Más a menudo, debe buscar más profundamente para descubrir por qué el valor es incorrecto en primer lugar. Asegúrese de saber cómo moverse hacia arriba y hacia abajo en la pila de llamadas y examine el entorno de la pila local.

Descubrimos que a menudo ayuda tener lápiz y papel cerca para que podamos tomar notas. En particular, a menudo nos encontramos con una pista y la perseguimos, solo para descubrir que no funcionó. Si no apuntábamos dónde estábamos cuando comenzamos la persecución, podríamos perder mucho tiempo para volver allí.

A veces, está viendo un seguimiento de pila que parece desplazarse para siempre. En este caso, a menudo hay una forma más rápida de encontrar el problema que examinar todos y cada uno de los marcos de la pila: usar un corte binario. Pero antes de discutir eso, veamos otros dos escenarios de errores comunes.

Sensibilidad a los valores de entrada

Ya ha estado allí. Su programa funciona bien con todos los datos de prueba y sobrevive con honor a su primera semana en producción. Luego, de repente se bloquea cuando se alimenta con un conjunto de datos en particular.

Puedes intentar mirar el lugar donde se bloquea y trabajar hacia atrás. Pero a veces es más fácil comenzar con los datos. Obtenga una copia del conjunto de datos y aliméntelo a través de una copia de la aplicación que se ejecute localmente, asegurándose de que siga fallando. Luego corte en binario los datos hasta que aísle exactamente qué valores de entrada están provocando el bloqueo.

Regresiones entre lanzamientos

Está en un buen equipo y lanza su software a producción. En algún momento, aparece un error en el código que funcionó bien hace una semana. ¿No sería bueno si pudiera identificar el cambio específico que lo introdujo? ¿Adivina qué? Tiempo de corte binario.

EL CHOP BINARIO

Todos los estudiantes de informática se han visto obligados a codificar un corte binario (a veces llamado búsqueda binaria). La idea es sencilla. Está buscando un valor particular en una matriz ordenada. Podría mirar cada valor uno por uno, pero terminaría mirando aproximadamente la mitad de las entradas en promedio hasta que encontrara el valor que deseaba o encontrara un valor mayor, lo que significaría que el valor no está en el formación.

Pero es más rápido usar un enfoque de divide y vencerás . Elija un valor en el medio de la matriz. Si es el que estás buscando, detente. De lo contrario, puede cortar la matriz en dos. Si el valor que encuentra es mayor que el objetivo, entonces sabe que debe estar en la primera mitad de la matriz; de lo contrario, está en la segunda mitad. Repita el procedimiento en el subarreglo apropiado y en poco tiempo tendrá un resultado. (Como veremos cuando hablaremos de la notación Big-O, una búsqueda lineal es  y un corte binario es ).

Entonces, el corte binario es mucho, mucho más rápido en cualquier problema de tamaño decente. Veamos cómo aplicarlo a la depuración.

Cuando se enfrenta a un seguimiento de pila masivo y está tratando de averiguar exactamente qué función destruyó el valor por error, hace un corte eligiendo un marco de pila en algún lugar en el medio y viendo si el error se manifiesta allí. Si es así, entonces debes concentrarte en los fotogramas anteriores, de lo contrario, el problema está en los fotogramas posteriores. Picar de nuevo. Incluso si tiene 64 fotogramas en el seguimiento de la pila, este enfoque le dará una respuesta después de seis intentos como máximo.

Si encuentra errores que aparecen en ciertos conjuntos de datos, es posible que pueda hacer lo mismo. Divida el conjunto de datos en dos y vea si el problema ocurre si alimenta uno u otro a través de la aplicación. Siga dividiendo los datos hasta que obtenga un conjunto mínimo de valores que muestren el problema.

Si su equipo ha introducido un error durante un conjunto de lanzamientos, puede usar el mismo tipo de técnica. Cree una prueba que haga que la versión actual falle. A continuación, elija una versión intermedia entre ahora y la última versión funcional conocida. Ejecute la prueba nuevamente y decida cómo limitar su búsqueda. Poder hacer esto es solo uno de los muchos beneficios de tener un buen control de versiones en sus proyectos. De hecho, muchos sistemas de control de versiones llevarán esto más lejos y automatizarán el proceso, eligiendo las versiones según el resultado de la prueba.

Los depuradores de

registro y/o rastreo 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. Por lo general, no puede decirle qué estaba haciendo antes de esta cadena de llamadas, especialmente en caso de que sistemas basados [25]

Las declaraciones de seguimiento 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 instrucciones 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 seguimiento deben tener un formato regular y coherente, ya que es posible que desee analizarlos automáticamente. Por ejemplo, si necesita rastrear una fuga de recursos (como aperturas/cierres de archivos desequilibrados), puede rastrear cada [apertura](#) y cada [cierre](#) en un archivo de registro. Al procesar el archivo de registro con herramientas de procesamiento de texto o comandos de shell, puede identificar fácilmente dónde estaba ocurriendo la [apertura](#) infractora.

Rubber Ducking 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 [26] se anuncie.

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

verbalice algunas de estas suposiciones, de repente puede obtener una nueva perspectiva del problema. Y si no tienes una persona, un patito de goma, un osito de peluche o una planta en una maceta servirán.^[27]

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, marco web, 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 es de un tercero, deberá eliminar el código antes de enviar el informe de errores.

Trabajamos en un proyecto en el que un ingeniero senior estaba convencido de que la llamada al sistema de **selección** estaba interrumpida en un sistema Unix. Ninguna cantidad de persuasión o 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 **select**, 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.

Consejo 33

"seleccionar" no está roto

Recuerde, si ve huellas de pezuñas, piense en caballos, no en cebras. El sistema operativo probablemente no esté roto. Y [seleccionar](#) 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 estas nuevas condiciones. Así que vigile de cerca el cronograma cuando considere una actualización; es posible que desee esperar hasta después del próximo lanzamiento.

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 ese algoritmo de cálculo de descuento, el que sabía que era a prueba de balas y posiblemente 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 proporcional a la cantidad de confianza y fe que tiene en el código que se está ejecutando. Por eso, ante un fracaso "sorprendente" , debes aceptar que uno o

más de sus suposiciones es incorrecta. No pase por alto una rutina o pieza de código involucrada en el error porque "sabe" que funciona. Pruébalo. Demuéstrelo en este contexto, con estos datos, con estas condiciones de contorno.

Consejo 34

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 hubiera aislado antes (consulte las discusiones sobre fallas tempranas y afirmaciones aquí y aquí, 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 que pase lo que pase, sabrás si vuelve a pasar.

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 VERIFICACIÓN DE DEPURACIÓN

- ¿El problema que se informa es un resultado directo del error subyacente o simplemente un síntoma?
- ¿El error está realmente en el marco que está utilizando? ¿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 las pruebas con estos datos?
- ¿Existen las condiciones que causaron este error en algún otro lugar del sistema? ¿Hay otros insectos todavía en la etapa larval, esperando para salir del cascarón?

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 24, Los programas muertos no dicen mentiras

RETOS

- La depuración es suficiente desafío.



Tema 21

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 algunas herramientas específicas (shells, editores, depuradores) 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 la programación lo que los enrute^[28]dores son para la 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 enrute^[28]dores 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.

Afortunadamente, hay varios lenguajes de manipulación de texto excelentes. A los desarrolladores de Unix (y aquí incluimos a los usuarios de macOS) a menudo les gusta usar el poder de sus shells de comando, aumentado con herramientas como [awk](#) y [sed](#). Las personas que prefieren una herramienta más estructurada pueden preferir lenguajes como Python o Ruby.

Estos lenguajes son importantes tecnologías habilitadoras. Usándolos, puede piratear rápidamente utilidades e ideas de prototipos:

trabajos que pueden tardar entre cinco y diez veces más utilizando 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 *The Practice of Programming* [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 puede manipular texto, interactuar con programas, hablar a través de redes, manejar páginas web, realizar operaciones aritméticas de precisión arbitraria y escribir programas que parecen palabrotas de Snoopy.

Consejo 35

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 cosas que hemos hecho con Ruby y Python relacionadas con la creación de este libro:

Construcción del

libro El sistema de construcción de la estantería pragmática está escrito en Ruby. Autores, editores, gente de maquetación y gente de soporte utilizan las tareas de Rake para coordinar la creación de PDF y libros electrónicos.

Inclusión y resaltado de código

Creemos que es importante que cualquier código presentado en un libro se haya probado primero. La mayor parte del código de este libro ha sido. Sin embargo, usando el principio DRY (ver Tema 9, *DRY—Los males de la duplicación*) no queríamos copiar y pegar líneas de código de los programas probados en el libro. Eso significaría que estaríamos

duplicando el código, 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 Ruby. 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.

Actualización del

sitio web Tenemos un script simple que crea un libro parcial, extrae el índice y luego lo carga en la página del libro en nuestro sitio web.

También tenemos un script que extrae secciones de un libro y las sube como muestras.

Incluir ecuaciones Hay

un script de Python que convierte el marcado matemático de LaTeX en texto con un formato agradable.

Generación de

índices La mayoría de los índices se crean como documentos separados (lo que dificulta su mantenimiento si cambia un documento).

Los nuestros están marcados en el texto mismo, y un script de Ruby coteja y formatea las entradas.

Etcétera. De una manera muy real, Pragmatic Bookshelf se basa en la manipulación de texto. Y si sigue nuestros consejos para mantener las cosas en texto sin formato, entonces usar estos lenguajes para manipular ese texto traerá una gran cantidad de beneficios.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 16, El poder del texto sin formato
- Tema 17, Juegos de conchas

EJERCICIOS

Ejercicio 11

Está reescribiendo una aplicación que solía usar YAML como lenguaje de configuración. Su empresa ahora se ha estandarizado en JSON, por lo que tiene un montón de archivos .yaml que deben convertirse en .json. Escriba una secuencia de comandos que tome un directorio y convierta cada archivo .yaml en un archivo .json correspondiente (de modo que [base de datos.yaml](#) se convierta [en base de datos.json](#) y el contenido sea JSON válido).

Ejercicio 12

Inicialmente, su equipo eligió usar nombres [camelCase](#) para las variables, pero luego cambió de opinión colectiva y cambió a [snake_case](#). Escriba un script que escanee todos los archivos de origen en busca de nombres de camelCase e informe sobre ellos.

Ejercicio 13

Siguiendo con el ejercicio anterior, agregue la capacidad de cambiar esos nombres de variables automáticamente en uno o más archivos. Recuerde mantener una copia de seguridad de los originales en caso de que algo salga terriblemente mal.



Tema 22

Diarios de Ingeniería

Dave una vez trabajó para un pequeño fabricante de computadoras, lo que significaba trabajar junto a ingenieros electrónicos y, a veces, mecánicos.

Muchos de ellos caminaban con un cuaderno de papel, normalmente con un bolígrafo metido en el lomo. De vez en cuando, cuando hablábamos, abrían el cuaderno y escribían algo.

Eventualmente, Dave hizo la pregunta obvia. Resultó que los habían entrenado para llevar un diario de ingeniería, una especie de diario en el que registraban lo que hacían, las cosas que aprendían, bocetos de ideas, lecturas de medidores: básicamente cualquier cosa que tuviera que ver con su trabajo. Cuando el cuaderno se llenaba, escribían el intervalo de fechas en el lomo y luego lo pegaban en el estante junto a los diarios anteriores. Es posible que haya habido una competencia suave por el juego de libros que ocupó la mayor parte del estante.

espacio.

Usamos diarios para tomar notas en las reuniones, para anotar en qué estamos trabajando, para anotar los valores de las variables al depurar, para dejar recordatorios donde colocamos las cosas, para registrar ideas locas y, a veces, solo para garabatear.^[29]

El diario tiene tres ventajas principales:

- Es más fiable que la memoria. La gente podría preguntar "¿Cuál era el nombre de la compañía a la que llamó la semana pasada sobre el suministro de energía?"

"¿problema?" y puede retroceder una página más o menos y darles el nombre y el número.

- Le brinda un lugar para almacenar ideas que no son inmediatamente relevantes para la tarea en cuestión. De esa manera, puede continuar concentrándose en lo que está haciendo, sabiendo que la gran idea no se olvidará.
- Actúa como una especie de patito de goma (descrito [aquí](#)). Cuando te detienes a escribir algo, tu cerebro puede cambiar de marcha, casi como si estuviera hablando con alguien, una gran oportunidad para reflexionar. Puede comenzar a escribir una nota y luego, de repente, darse cuenta de que lo que acaba de hacer, el tema de la nota, es simplemente incorrecto.

También hay un beneficio adicional. De vez en cuando puedes recordar lo que estabas haciendo hace tantos años y pensar en las personas, los proyectos y la ropa y los peinados horribles.

Por lo tanto, intente llevar un diario de ingeniería. Use papel, no un archivo o un wiki: hay algo especial en el acto de escribir en comparación con escribir a máquina. Espere un mes y vea si obtiene algún beneficio.

Al menos, hará que escribir tus memorias sea más fácil cuando seas rico y famoso.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 6, Su carpeta de conocimientos
- Tema 37, Escucha tu cerebro de lagarto

notas al pie

[24] Todo el software se convierte en software heredado tan pronto como se escribe.

[25] Aunque el lenguaje Elm tiene un depurador de viajes en el tiempo.

JPor qué "pato de goma"? Mientras estudiaba en el Imperial College de Londres, Dave [26] 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...

[27] Las versiones anteriores del libro hablaban de hablar con tu planta de maceta. Fue un error tipográfico. Honesto.

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

[29] Existe cierta evidencia de que garabatear ayuda a concentrarse y mejora las habilidades cognitivas, por ejemplo, consulte [¿Qué hace garabatear? \[Y10\]](#).

Capítulo 4

Paranoia pragmática

Consejo 36

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 informática 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, enviando mensajes de texto por teléfono 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. Estamos interactuando constantemente con el código de otras personas (código que podría no cumplir con nuestros altos estándares) y lidiando 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 y desconfiar de los datos de posibles atacantes o trolls. 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 construyen defensas contra sus propios errores. Describimos la primera medida defensiva en el Tema 23, Diseño por contrato: los clientes y proveedores deben acordar derechos y responsabilidades.

En el Tema 24, 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.

El Tema 25, Programación asertiva, describe un método fácil de verificar en el camino: escriba código que verifique activamente sus suposiciones.

A medida que sus programas se vuelven más dinámicos, se encontrará haciendo malabarismos con los recursos del sistema: memoria, archivos, dispositivos y similares. En el Tema 26, Cómo equilibrar los recursos, sugeriremos formas de garantizar que no se le caiga ninguna pelota.

Y lo que es más importante, siempre nos ceñimos a los pequeños pasos, como se describe en el Tema 27, No supere sus luces delanteras, por lo que

no te caigas por el borde del precipicio.

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

Como dijo Woody Allen: “Cuando todo el mundo te persigue , la paranoia es solo un buen pensamiento” .



Tema 23

Diseño por contrato

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

Ralph Waldo Emerson,
Ensayos

contrato.

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

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 (Construcción de Software Orientada a Objetos [Mey97]) desarrolló el concepto de Diseño por Contrato para el lenguaje Eiffel. Es una técnica simple pero poderosa^[30] 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 Design by Contract (DBC, para abreviar).

Cada función y método en un sistema de software hace algo. Antes de que comience ese algo, la función 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:

Condiciones

previas Lo que 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 pasar buenos datos (vea el recuadro aquí).

poscondiciones

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 condición posterior implica que concluirá : 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 un

rutina, el invariante puede no ser válido, pero cuando la rutina sale y el control vuelve a la persona que llama, 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).

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), tal vez se presente una excepción o el programa finalice. 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.

Algunos lenguajes admiten mejor estos conceptos que otros. Clojure, por ejemplo, admite condiciones previas y posteriores, así como la instrumentación más completa proporcionada por las especificaciones. Aquí hay un ejemplo de una función bancaria para hacer un depósito utilizando condiciones previas y posteriores simples:

```
(defn accept-deposit [cantidad de id de
cuenta] { :pre [ (> cantidad
0.0) (cuenta abierta? id de cuenta) ]
:post [ (contiene? (cuenta-transacciones cuenta-id) %) ] }
"Acepte un depósito y devuelva la nueva identificación de transacción"
;; Otro procesamiento va aquí...
;; Devuelve la transacción recién creada:
```

```
(create-transaction account-id :cantidad del deposito))
```

Hay dos condiciones previas para la función [de aceptar-depositar](#). La primera es que el monto sea mayor que cero, y la segunda es que la cuenta esté abierta y sea válida, según lo determinado por alguna función denominada [cuenta-abierta?](#). También hay una condición posterior: la función garantiza que la nueva transacción (el valor de retorno de esta función, representado aquí por '%') se puede encontrar entre las transacciones de esta cuenta.

Si llama a [accept-deposit](#) con un monto positivo para el depósito y una cuenta válida, procederá a crear una transacción del tipo apropiado y realizará cualquier otro procesamiento que realice.

Sin embargo, si hay un error en el programa y de alguna manera pasó una cantidad negativa para el depósito, obtendrá una excepción de tiempo de ejecución:

Excepción en el hilo "principal"...

Causado por: java.lang.AssertionError: Afirmación fallida: (> cantidad 0.0)

De manera similar, esta función requiere que la cuenta especificada esté abierta y sea válida. Si no es así, verá esa excepción en su lugar:

Excepción en el hilo "principal"...

Causado por: java.lang.AssertionError: error de afirmación: (¿cuenta abierta? ID de cuenta)

Otros lenguajes tienen características que, aunque no son específicas de DBC, aún se pueden usar con buenos resultados. Por ejemplo, Elixir usa cláusulas de protección para despachar llamadas de función contra varios cuerpos disponibles:

```
defmodule Depósitos do
  def accept_deposit(account_id, cantidad) cuando (cantidad > 100000) do #
    ; Llama al administrador!
  end
  def accept_deposit(account_id, cantidad) cuando (cantidad > 10000) hacer
```

```
# Requisitos federales adicionales para la presentación
de informes # Algun
```

```
procesamiento... end def accept_deposit(id_de_cuenta, monto) cuando (cantidad
> 0) hacer # Algun
```

```
procesamiento... end end
```

En este caso, llamar a `accept_deposit` con una cantidad lo suficientemente grande puede desencadenar pasos y procesos adicionales. Sin embargo, intente llamarlo con una cantidad menor o igual a cero y obtendrá una excepción que le informará que no puede:

```
** (FunctionClauseError) ninguna cláusula de función coincide en
Deposits.accept_deposit/2
```

Este es un mejor enfoque que simplemente verificar sus entradas; en este caso, simplemente no puede llamar a esta función si sus argumentos están fuera de rango.

Consejo 37

Diseño con Contratos

En el Tema 10, Ortogonalidad, recomendamos escribir código "tímid". 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!

En cualquier lenguaje de programación, ya sea funcional, orientado a objetos o procedural, DBC te obliga a pensar.

DBC y desarrollo basado en pruebas

¿Se necesita el diseño por contrato en un mundo donde los desarrolladores practican pruebas unitarias, desarrollo dirigido por pruebas (TDD), pruebas basadas en propiedades o programación defensiva?

La respuesta corta es sí."

DBC y las pruebas son enfoques diferentes para el tema más amplio de la corrección del programa.

Ambos tienen valor y ambos tienen usos en diferentes situaciones. DBC ofrece varias ventajas sobre enfoques de prueba específicos:

- DBC no requiere ninguna configuración o simulación
- DBC define los parámetros para el éxito o el fracaso en todos los casos, mientras que las pruebas solo pueden enfocarse en un caso específico a la vez.
- TDD y otras pruebas ocurren solo en el "momento de la prueba" dentro del ciclo de construcción. Pero DBC y las aserciones son para siempre: durante el diseño, desarrollo, implementación y mantenimiento.
- TDD no se enfoca en verificar las invariantes internas dentro del código bajo prueba, es más un estilo de caja negra para verificar la interfaz pública
- DBC es más eficiente (y DRY-er) que la programación defensiva, donde todos tienen que validar los datos en caso de que nadie más lo haga.

TDD es una gran técnica, pero como con muchas técnicas, podría invitarlo a concentrarse en el "camino feliz" y no en el mundo real lleno de datos incorrectos, actores incorrectos, versiones incorrectas y especificaciones incorrectas.

Invariantes de clase y lenguajes funcionales Es

una cuestión de nombres. Eiffel es un lenguaje orientado a objetos, por lo que Meyer llamó a esta idea "invariante de clase". Pero, en realidad, es más general que eso. A lo que realmente se refiere esta idea es al estado. En un lenguaje orientado a objetos, el estado está asociado con instancias de clases. Pero otros idiomas también tienen estado.

En un lenguaje funcional, normalmente pasa el estado a las funciones y recibe un estado actualizado como resultado. Los conceptos de invariantes son igual de útiles en estas circunstancias.

IMPLEMENTACIÓN DE DCC

Simplemente enumerar cuál es el rango del dominio de entrada, cuáles son las condiciones límite y qué promete entregar la rutina (o, lo que es más importante, qué no promete entregar) antes de escribir el código es un gran paso adelante en la escritura.

mejor software. Al no decir estas cosas, regresa a la programación por coincidencia (vea la discusión aquí), que es donde muchos proyectos comienzan, terminan y fallan.

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, una técnica de diseño . Incluso sin verificación automática, puede poner el contrato en el código como comentarios o en las pruebas unitarias y aun así obtener un beneficio muy real.

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 lenguajes usando aserciones: comprobaciones en tiempo de ejecución de condiciones lógicas (consulte el Tema 25, Programación asertiva). ¿Por qué solo parcialmente? ¿No puedes usar aserciones para hacer todo lo que DBC puede hacer?

Desafortunadamente, la respuesta es no. Para empezar, en los lenguajes orientados a objetos probablemente no haya 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.

En otros entornos, las excepciones generadas a partir de aserciones de estilo DBC pueden desactivarse globalmente o ignorarse en el código.

Además, no existe un concepto integrado de valores "antiguos"; es decir, valores

tal como existían a la entrada de un método. Si está utilizando aserciones para hacer cumplir los contratos, debe agregar código a la condición previa para guardar cualquier información que desee usar en la condición posterior, si el lenguaje lo permite. En el lenguaje Eiffel, donde nació DBC, puedes usar una expresión [antigua](#).

Finalmente, los sistemas y bibliotecas de tiempo de ejecución convencionales 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 (vea el Tema 24, [Los programas muertos no dicen mentiras](#) para una discusión más detallada).

[¿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 es probado detrás de escena después de que la persona que llama invoca la rutina pero antes de la rutina misma es ingresado. Por lo tanto, si se debe realizar una verificación explícita de los parámetros, debe realizarla el llamador, porque la rutina en sí nunca verá parámetros que violen su condición previa. (Para lenguajes sin soporte incorporado, necesitaría poner entre paréntesis la rutina llamada con un preámbulo y/o un postámbulo que verifique estas afirmaciones).

Considere un programa que lee un número de la consola, calcula su raíz cuadrada (llamando a `sqrt`) e imprime el resultado. La función `sqrt` tiene una condición previa: su argumento no debe ser negativo. Si el usuario ingresa un número negativo en la consola, depende del código de llamada asegurarse de que nunca pase a `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 una `i` al resultado devuelto por `sqrt`.

Cualquiera que sea su elección, este definitivamente no es un problema [de `sqrt`](#).

Al expresar el dominio de la función de raíz cuadrada en la condición previa de la rutina `sqrt`, transfiere la carga de la corrección a la persona que llama, donde pertenece. Luego puede diseñar la rutina `sqrt` con la seguridad de que su entrada estará dentro del rango.

DBC Y BLOQUEO TEMPRANO

DBC encaja muy bien con nuestro concepto de bloqueo temprano (consulte el Tema 24, [Los programas muertos no cuentan mentiras](#)). Mediante el uso de una afirmación o DBC

mecanismo para validar las condiciones previas, las condiciones posteriores y las invariantes, puede colapsar antes y brindar información más precisa sobre el problema.

Por ejemplo, suponga que tiene un método que calcula raíces cuadradas. Necesita una condición previa de DBC que restrinja el dominio a números positivos. En los idiomas que admiten DBC, si pasa un parámetro negativo a `sqrt`, obtendrá un error informativo como `sqrt_arg_must_be_positive`, junto con un seguimiento de la pila.

Esto es mejor que la alternativa en otros lenguajes como Java, C y C++ donde pasar un número negativo a `sqrt` devuelve el valor especial `NaN` (No es un número). Puede pasar algún tiempo más tarde en el programa que intente hacer algo de matemáticas en `NaN`, con resultados sorprendentes.

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

INVARIANTES SEMÁNTICOS

Puede 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 pueda ocurrir, el error debe estar del 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 solucionar problemas complejos de recuperación de errores.

escenarios, 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. Es por eso que usamos el término invariantes semánticos : debe ser fundamental para el significado mismo de una cosa y no estar 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 ve Trate de expresarlo de manera clara y sin ambigüedades. Por ejemplo, en el ejemplo de la tarjeta de débito, podríamos escribir

Errar a favor del consumidor.

Esta es una declaración clara, concisa e inequívoca que es aplicable en muchas áreas diferentes del sistema. Es nuestro contrato con todos los usuarios del sistema, nuestra garantía de comportamiento.

CONTRATOS DINÁMICOS Y AGENTES

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 las solicitudes que no quieren cumplir. 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

dependencia crítica de los 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

- Tema 24, Los programas muertos no dicen mentiras
- Tema 25, Programación Asertiva
- Tema 38, Programación por Coincidencia
- Tema 42, Pruebas basadas en propiedades
- Tema 43, Manténgase seguro ahí afuera
- Tema 45, El pozo de requisitos

RETOS

- 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

Ejercicio 14 (respuesta posible)

Diseñe una interfaz para una licuadora de cocina. Eventualmente será una licuadora basada en la web y habilitada para IoT, 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.

```
int getSpeed()  
void setSpeed(int x)  
boolean isFull()  
void fill()  
void empty()
```

Ejercicio 15 (respuesta posible)

¿Cuántos números hay en la serie 0, 5, 10, 15, ..., 100?



Tema 24

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 o del marco de trabajo la que lo detecta primero. Tal vez hemos pasado un valor **nulo** o una lista vacía. Tal vez falta una clave en ese hash, o el valor que pensamos que contenía un hash realmente contiene una lista. Tal vez hubo un error de red o un error en el sistema de archivos que no detectamos y tenemos datos vacíos o corruptos. 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. Accederemos al caso **predeterminado** de forma inesperada. Esa es también una de las razones por las que todas y cada una de las declaraciones de caso/cambio deben tener una cláusula predeterminada: queremos saber cuándo sucedió 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. Nos aseguramos de que los datos sean lo que creemos que son, que el código en producción sea el código que creemos que es. Estamos comprobando que realmente se cargaron las versiones correctas de las dependencias.

Todos los errores te dan información. Podrías convencerte a ti mismo

que el error no puede ocurrir y elija ignorarlo. En cambio, los programadores pragmáticos se dicen a sí mismos que si hay un error, algo muy, muy malo ha sucedido. No te olvides de leer el maldito mensaje de error (ver [Coder in a Strange Land](#)).

CAPTURA Y LIBERACIÓN ES PARA PECES

Algunos desarrolladores sienten que es un buen estilo capturar o rescatar todas las excepciones, volviendo a generarlas después de escribir algún tipo de mensaje. Su código está lleno de cosas como esta (donde una declaración simple de [aumento](#) vuelve a generar la excepción actual):

```
intente
    hacer add_score_to_board (puntuación);
    rescate InvalidScore
        Logger.error("No se puede agregar una puntuación no válida.

Saliendo"); subir rescate BoardServerDown
        Logger.error("No se puede agregar puntaje: el tablero está inactivo. Saliendo");
        elevar
        rescate StaleTransaction
            Logger.error("No se puede agregar puntaje: transacción obsoleta. Saliendo");
            levantar
        el extremo
```

Así es como los programadores pragmáticos escribirían esto:

```
add_score_to_board(puntuación);
```

Lo preferimos por dos razones. Primero, el código de la aplicación no se ve eclipsado por el manejo de errores. Segundo, y quizás más importante, el código está menos acoplado. En el ejemplo detallado, tenemos que enumerar todas las excepciones que podría generar el método [add_score_to_board](#). Si el escritor de ese método agrega otra excepción, nuestro código está sutilmente desactualizado. En la segunda versión más pragmática, la nueva excepción se propaga automáticamente.

CRASH, NO BASURA

Uno de los beneficios de detectar problemas tan pronto como sea posible es que puede bloquearse antes, y bloquearse suele ser lo mejor que puede hacer. La alternativa puede ser continuar, escribiendo datos corruptos en alguna base de datos vital o ordenando a la lavadora que realice su vigésimo ciclo de centrifugado consecutivo.

Los lenguajes Erlang y Elixir adoptan esta filosofía. Joe Armstrong, inventor de Erlang y autor de Programación de Erlang: software para un mundo concurrente [Arm07], suele decir: “La programación defensiva es una pérdida de tiempo.

¡Que se estrelle!” En estos entornos, los programas están diseñados para fallar, pero esa falla se maneja con supervisores. Un supervisor es responsable de ejecutar el código y sabe qué hacer en caso de que el código falle, lo que podría incluir limpiarlo, reiniciarlo, etc. ¿Qué sucede cuando el propio supervisor falla? Su propio supervisor gestiona ese evento, lo que lleva a un diseño compuesto por árboles supervisores. La técnica es muy efectiva y ayuda a explicar el uso de estos lenguajes en sistemas tolerantes a fallas y de alta disponibilidad.

En otros entornos, puede ser inapropiado simplemente salir de un programa en ejecución. Es posible que haya reclamado recursos que quizás no se liberen, o que necesite escribir mensajes de registro, ordenar transacciones abiertas o interactuar con otros procesos.

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 terminelo.

lo antes posible.

Un programa muerto normalmente hace mucho menos daño que uno lisiado.
uno.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 20, Depuración
- Tema 23, Diseño por contrato
- Tema 25, Programación Asertiva
- Tema 26, Cómo equilibrar los recursos
- Tema 43, Manténgase seguro ahí afuera



Tema 25

Programación Asertiva

Hay un lujo en el autorreproche. Cuando nos culpamos a nosotros mismos sentir 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

Esto nunca puede pasar...

"Esta aplicación nunca será utilizado en el extranjero, entonces, ¿por qué internacionalizarlo? "contar no puede ser negativo". "El registro no puede fallar".

No practiquemos este tipo de autoengaño, particularmente al programar.

Consejo 39

Use afirmaciones para prevenir lo imposible

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 muchas implementaciones de lenguaje, encontrará alguna forma de **aserción** que verifica una condición booleana. Estos controles pueden ser invaluables. Si un parámetro o un resultado nunca debe ser **nulo**, verifíquelo explícitamente:

```
afirmar (resultado! = nulo);
```

En la implementación de Java, puede (y debe) agregar una cadena descriptiva:

```
afirmar resultado != nulo && resultado.tamaño() > 0 : "Resultado vacío de XYZ";
```

Las afirmaciones también son comprobaciones útiles del funcionamiento de un algoritmo.

Tal vez haya escrito un algoritmo de clasificación inteligente, llamado `my_sort`.

Comprueba que funciona:

```
libros = my_sort(find("scifi"))
assert(¿está_ordenado?(libros))
```

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

```
puts("Ingrese 'Y' o 'N': ") ans
= gets[0] # Toma el primer carácter de la respuesta
assert((ch == 'Y') || (ch == 'N')) # Muy mal ¡ idea!
```

Y solo porque la mayoría de las implementaciones [de aserción](#) terminarán el proceso cuando una aserción falla, no hay razón por la que las versiones que escribas deban hacerlo. Si necesita liberar recursos, capture la excepción de la aserción o atrape la salida y ejecute su propio 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.

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, sería una mala idea codificar algo como

```

while (iter.hasMoreElements())
{ afirmar(iter.nextElement() != null);
  Objeto obj = iter.nextElement(); // }
  ...

```

La llamada `.nextElement()` en la aserción tiene el efecto secundario de mover el iterador más allá del elemento que se está recuperando, por lo que el bucle procesará solo la mitad de los elementos de la colección. sería mejor escribir

```

while (iter.hasMoreElements())
{ Object obj = iter.nextElement();
  afirmar (objeto != null);
  ...
  nulo); // }

```

Este problema es una especie de Heisenbug:^[32] depuración que cambia el comportamiento del sistema que se está depurando.

(También creemos que hoy en día, cuando la mayoría de los lenguajes tienen un soporte decente para iterar funciones sobre colecciones, este tipo de bucle explícito es innecesario y de mala forma).

DEJAR AFIRMACIONES ACTIVADAS

Hay un malentendido común acerca de las afirmaciones. 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 por el que es poco probable que pruebe ni siquiera un porcentaje minúsculo de las permutaciones por las que pasará su código. En segundo lugar, los optimistas se olvidan de que su programa se ejecuta en un mundo peligroso. Durante las pruebas, es probable que las ratas no muerdan un cable de comunicaciones, que alguien que esté jugando no agote la memoria y que los archivos de registro no llenen la partición de almacenamiento. 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 tiene problemas de rendimiento, apague solo aquellas afirmaciones que realmente lo golpean. El ejemplo de ordenación anterior puede ser una parte fundamental de su aplicación y es posible que deba ser rápido. Agregar la verificación significa otra pasada a través de los datos, lo que podría ser inaceptable. Haga que ese cheque en particular sea opcional, pero deje el resto.

Use aserciones en producción, gane mucho dinero

Un antiguo vecino de Andy dirigía una pequeña empresa nueva que fabricaba dispositivos de red. Uno de los secretos de su éxito fue la decisión de dejar las afirmaciones en su lugar en los comunicados de producción. Estas aserciones fueron bien diseñadas para informar todos los datos pertinentes que llevaron a la falla y se presentaron a través de una interfaz de usuario atractiva para el usuario final. Este nivel de comentarios, de usuarios reales en condiciones reales, permitió a los desarrolladores tapar los agujeros y corregir estos errores oscuros y difíciles de reproducir, lo que resultó en un software a prueba de balas notablemente estable.

Esta pequeña y desconocida empresa tenía un producto tan sólido que pronto fue adquirida por cientos de millones de dólares.

Sólo digo'.

Ejercicio 16 (respuesta posible)

Una revisión rápida de la realidad. ¿Cuál de estas cosas “imposibles” puede suceder?

- Un mes con menos de 28 días
- Código de error de una llamada al sistema: no se puede acceder al directorio actual
- En C++: $a = 2; b = 3;$ pero $(a + b)$ no es igual a 5
- Un triángulo con una suma de ángulos interiores $\neq 180^\circ$
- Un minuto que no tiene 60 segundos
- $(un + 1) \leq un$

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 23, Diseño por contrato
- Tema 24, Los programas muertos no dicen mentiras
- Tema 42, Pruebas basadas en propiedades
- Tema 43, Manténgase seguro ahí afuera



Tema 26

Cómo equilibrar los recursos

Encender una vela es proyectar una sombra...

Ursula K. Le Guin, un mago de Terramar

Todos administramos recursos cada vez que codificamos: memoria, transacciones, subprocessos, conexiones de red, 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

desasignarlo.

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:

Consejo 40

Termina lo que empiezas

Este consejo es fácil de aplicar en la mayoría de las circunstancias. Simplemente significa que la función u objeto que asigna un recurso debe ser responsable de desasignarlo. Veamos cómo se aplica mirando un ejemplo de un código incorrecto: parte de un programa de Ruby 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:

```
def read_customer
  @customer_file = File.open(@name + ".rec", "r+")
  @balance = BigDecimal(@customer_file.gets)
end
```

```

def write_customer
  @customer_file.rewind
  @customer_file.puts @balance.to_s
  @customer_file.close
end

def actualizar_cliente(cantidad_transacción)
  leer_cliente
  @saldo = @saldo.añadir(cantidad_transacción,2)
  escribir_cliente
end

```

A primera vista, la rutina [update_customer](#) parece razonable. Parece implementar la lógica que requerimos: leer un registro, actualizar el saldo y volver a escribir el registro. Sin embargo, esta pulcritud esconde un gran problema. Las rutinas [leer_cliente](#) y [escribir_cliente](#) están estrechamente acopladas: comparten la variable de instancia [archivo_cliente](#). [read_customer](#) abre el archivo y almacena la referencia del archivo en [customer_file](#), y luego [write_customer](#) usa esa referencia almacenada para cerrar el archivo cuando finaliza. Esta variable compartida ni siquiera aparece en la rutina [update_customer](#).

¿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.

Entran en la fuente y cambian [update_customer](#):

```

def actualizar_cliente(cantidad_transacción)
  leer_cliente if
    (cantidad_transacción >= 0.00)
    @saldo = @saldo.add(cantidad_transacción,2)
    escribir_cliente
  end
end

```

Todo parece estar bien durante las pruebas. Sin embargo, cuando el código entra en producción, colapsa después de varias horas, quejándose de demasiado

muchos archivos abiertos. Resulta que no se llama a `write_customer` en algunas circunstancias. Cuando eso sucede, el archivo no se cierra.

Una muy mala solución a este problema sería tratar el caso especial en `update_customer`:

```
def update_customer(transaction_amount)
    read_customer if
        (transaction_amount >= 0.00) @balance
        += BigDecimal(transaction_amount, 2) write_customer else
        @customer_file.close
    #
    ¡ Mala idea! final final
```

Esto solucionará el problema: el archivo ahora se cerrará independientemente del nuevo saldo, pero la solución ahora significa que se acoplan tres rutinas a través de la variable compartida `archivo_cliente`, y el seguimiento de cuándo el archivo está abierto o no comenzará a alborotarse. Estamos cayendo en una trampa, y las cosas van a empezar a ir cuesta abajo rápidamente si seguimos en este curso. ¡ Esto no está equilibrado!

El consejo Termina lo que empiezas nos dice que, idealmente, la rutina que asigna un recurso también debería liberarlo. Podemos aplicarlo aquí refactorizando ligeramente el código:

```
def read_customer(archivo)
    @balance=BigDecimal(archivo.obtiene)
end

def escribir_cliente(archivo)

    archivo.rebobinar archivo.puts
    @balance.to_s end
```

```
def update_customer(transaction_amount)
    file=File.open(@name + ".rec", "r+")
    # >-
    read_customer(file) # | @balance =
    @balance.add(transaction_amount,2) # | archivo.cerrar final
    # <--
```

En lugar de mantener la referencia del archivo, hemos cambiado el código para pasarlo como parámetro.^[34] Ahora toda la responsabilidad del archivo está en la rutina `update_customer`. Abre el archivo y (terminando lo que empieza) lo cierra antes de volver. 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 compartida fea.

Hay otra pequeña pero importante mejora que podemos hacer. En muchos lenguajes modernos, puede limitar el tiempo de vida de un recurso a un bloque cerrado de algún tipo. En Ruby, hay una variación del archivo `abierto` que pasa en la referencia del archivo abierto a un bloque, que se muestra aquí entre el `do` y el `final`:

```
def actualizar_cliente(cantidad_transacción)
    Archivo.open(@nombre + ".rec", "r+") do |archivo|
        # >-
        read_customer(archivo) # | @balance =
        @balance.add(transaction_amount,2) # |
        escribir_cliente(archivo) # | final final
        # <--
```

En este caso, al final del bloque, la variable `del archivo` queda fuera del alcance y el archivo externo se cierra. Período. No es necesario que recuerde cerrar el archivo y liberar la fuente, está garantizado que sucederá para usted.

En caso de duda, siempre vale la pena reducir el alcance.

Consejo 41

actuar localmente

Equilibrio a lo largo del tiempo

En este tema, estamos analizando principalmente los recursos efímeros utilizados por su proceso en ejecución.

Pero es posible que desee considerar qué otros líos podría estar dejando atrás.

Por ejemplo, ¿cómo se manejan sus archivos de registro? Está creando datos y utilizando espacio de almacenamiento.

¿Existe algo para rotar los troncos y limpiarlos? ¿Qué hay de los archivos de depuración no oficiales que estás soltando? Si está agregando registros de registro en una base de datos, ¿existe un proceso similar para caducarlos? Para cualquier cosa que cree que consume un recurso finito, considere cómo equilibrarlo.

¿Qué más estás dejando atrás?

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:

- Desasigne 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.
- 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 [el recurso 1](#) y está a punto de reclamar [el recurso 2](#), mientras que el proceso B ha reclamado [el recurso 2](#) y está tratando de obtener [el recurso 1](#), los dos procesos esperarán eternamente).

No importa qué tipo de recursos estemos usando (transacciones, conexiones de red, 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 orientada a objetos. 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 trabaja con lenguajes 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 soporte lingüístico. Por lo general, tiene dos opciones:

1. Utilice el ámbito de las variables (por ejemplo, apilar variables en C++ o Rust)
2. Usa una cláusula "finally" en un bloque `try...catch`

Con las reglas de alcance habituales en lenguajes como C++ o Rust, la memoria de la variable se recuperará cuando la variable quede fuera del alcance a través de un retorno, una salida de bloque o una excepción. Pero también puede conectarse al destructor de la variable para limpiar cualquier elemento externo.

recursos. En este ejemplo, la variable de Rust denominada `cuentas` cerrará automáticamente el archivo asociado cuando quede fuera de alcance:

```
{ let mut account = File::open("mydata.txt")?; // >-- // usa
  'cuentas' // | // | } // <-- // 'cuentas' ahora está
  ...
  fuera
  del alcance, y el archivo se // cierra
  automáticamente
```

La otra opción, si el lenguaje lo admite, es la [cláusula final](#).

Una cláusula [finalmente](#) garantizará que el código especificado se ejecutará independientemente de si se generó o no una excepción en el bloque `try...catch`:

```
intente // captura de
algunas
cosas dudosas // finalmente
se
generó una excepción // limpia en cualquier caso
```

Sin embargo, hay una trampa.

Un antipatrón de excepción

Comúnmente vemos gente escribiendo algo como esto:

```
comenzar cosa =
  allocate_resource()
proceso
  (cosa) finalmente
desasignar (cosa) fin
```

¿Puedes ver lo que está mal?

¿Qué sucede si la asignación de recursos falla y genera una excepción? La [cláusula final](#) lo atrapará e intentará desasignar un

cosa que nunca se asignó.

El patrón correcto para manejar la desasignación de recursos en un entorno con excepciones es

```
cosa = allocate_resource()  
  
comenzar proceso  
(cosa)  
finalmente desasignar  
(cosa) fin
```

CUANDO NO SE PUEDEN 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 estructuras de datos dinámicas. 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:

- La estructura de nivel superior también es responsable de liberar cualquier subestructura que contenga. Estas estructuras luego eliminan recursivamente los datos que contienen, y así sucesivamente.
- 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.
- La estructura de nivel superior se niega a desasignarse si contiene subestructuras.

La elección aquí depende de las circunstancias de cada estructura de datos individual. Sin embargo, es necesario que sea explícito.

para cada uno, e implemente su decisión consistentemente.

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).

CONSULTAR EL 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 liberan 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.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 24, Los programas muertos no dicen mentiras
- Tema 30, Transformación de la programación

- Tema 33, Rompiendo el Acoplamiento Temporal

RETOS

- 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. Considere cómo el Tema 23, Diseño por contrato, podría ayudar a refinar esta idea.

Ejercicio 17 (respuesta posible)

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

Ejercicio 18 (respuesta posible)

Algunos desarrolladores de Java insisten en establecer una variable de objeto en **NULL** después de haber terminado de usar el objeto. ¿Por qué es una buena idea?



Tema 27

No superes tus faros

Es difícil hacer predicciones, especialmente sobre el futuro.

Lawrence "Yogi" Berra, según un proverbio danés

Es tarde en la noche, oscuro, lluvia torrencial. El biplaza se desliza por las curvas cerradas de las pequeñas y sinuosas carreteras de montaña, defendiéndose a duras penas de las esquinas. Sube una horquilla y el auto no la pasa, choca contra la diminuta barandilla y se eleva hacia un feroz estruendo en el valle de

abajo. La policía estatal llega a la escena y el oficial superior niega con la cabeza con tristeza "Debe haber dejado atrás sus faros".

¿El veloz biplaza había ido más rápido que la velocidad de la luz? No, ese límite de velocidad está firmemente fijado. A lo que se refirió el oficial fue a la capacidad del conductor para detenerse o girar a tiempo en respuesta a la iluminación de las luces delanteras.

Los faros tienen un cierto alcance limitado, conocido como distancia de proyección. Más allá de ese punto, la propagación de la luz es demasiado difusa para ser efectiva. Además, los faros solo se proyectan en línea recta y no iluminan nada fuera del eje, como curvas, colinas o depresiones en la carretera. De acuerdo con la Administración Nacional de Seguridad del Tráfico en las Carreteras, la distancia promedio iluminada por los faros de luz baja es de aproximadamente 160 pies. Desafortunadamente, la distancia de frenado a 40 mph es de 189 pies, y a 70¹⁰⁵ mph la friolera de 464 pies. De hecho, es bastante fácil dejar atrás las luces delanteras.

En el desarrollo de software, nuestros "faros" son igualmente limitados. No podemos ver muy lejos en el futuro, y cuanto más lejos del eje miras, más oscuro se vuelve. Así que los programadores pragmáticos tienen una regla firme:

Consejo 42

Tome pequeños pasos, siempre

Siempre tome pasos pequeños y deliberados, verificando la retroalimentación y ajustando antes de continuar. Considere que la tasa de retroalimentación es su límite de velocidad. Nunca da un paso o una tarea que es "demasiado grande".

¿Qué entendemos exactamente por retroalimentación? Cualquier cosa que confirme o refute de forma independiente su acción. Por ejemplo:

- Los resultados en un REPL brindan comentarios sobre su comprensión de las API y los algoritmos
- Las pruebas unitarias brindan retroalimentación sobre su último cambio de código
- La demostración y la conversación del usuario brindan comentarios sobre las funciones y la usabilidad

¿Qué es una tarea que es demasiado grande? Cualquier tarea que requiera "adivinación". Así como los faros de los automóviles tienen un alcance limitado, solo podemos ver el futuro, quizás uno o dos pasos, quizás unas pocas horas o días como máximo. Más allá de eso, puede pasar rápidamente de una suposición informada a una especulación salvaje. Es posible que te encuentres cayendo en la adivinación cuando tengas que:

- Estimar las fechas de finalización meses en el futuro
- Planifique un diseño para el mantenimiento futuro o la capacidad de ampliación
- Adivina las necesidades futuras del usuario

- Adivina la disponibilidad tecnológica futura

Pero, lo escuchamos llorar, ¿no se supone que debemos diseñar para el mantenimiento futuro? Sí, pero solo hasta cierto punto: solo hasta donde puedas ver. Cuanto más tenga que predecir cómo será el futuro, mayor será el riesgo de equivocarse. En lugar de desperdiciar esfuerzos diseñando para un futuro incierto, siempre puede recurrir al diseño de su código para que sea reemplazable. Facilite la eliminación de su código y reemplácelo con algo más adecuado. Hacer que el código sea reemplazable también ayudará con la cohesión, el acoplamiento y el SECO, lo que conducirá a un mejor diseño en general.

Aunque se sienta confiado en el futuro, siempre existe la posibilidad de que un cisne negro esté a la vuelta de la esquina.

CISNES NEGROS

En su libro, The Black Swan: The Impact of the Highly Improbable [Tal10], Nassim Nicholas Taleb postula que todos los eventos significativos en la historia provienen de eventos de alto perfil, difíciles de predecir y raros que están más allá del ámbito de lo normal. Expectativas. Estos valores atípicos, aunque estadísticamente raros, tienen efectos desproporcionados. Además, nuestros propios sesgos cognitivos tienden a cegarnos ante los cambios que surgen en los bordes de nuestro trabajo (consulte el Tema 4, Sopa de piedra y ranas hervidas).

Alrededor de la época de la primera edición de The Pragmatic Programmer, el debate se desató en revistas de informática y foros en línea sobre la pregunta candente: "¿Quién ganaría la guerra de las GUI de escritorio, Motif u OpenLook?" Era la pregunta equivocada.^[36] Lo más probable es que nunca haya oído hablar de estas tecnologías, ya que ninguna "ganó" y la web centrada en el navegador dominó rápidamente el panorama.

Consejo 43

Evite la adivinación

La mayor parte del tiempo, el mañana se parece mucho al hoy. Pero no cuentes con eso.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 12, Viñetas trazadoras
- Tema 13, Prototipos y Post-it Notes
- Tema 40, Refactorización
- Tema 41, Prueba de código
- Tema 48, La esencia de la agilidad
- Tema 50, Los cocos no lo cortan

notas al pie

[30] Basado en parte en trabajos anteriores de Dijkstra, Floyd, Hoare, Wirth y otros.

[31] En C y C++, generalmente se implementan como macros. En Java, las aserciones están deshabilitadas por defecto. Invoque la VM de Java con el indicador `-enableassertions` para habilitarlas y déjelas habilitadas.

[32] <http://www.eps.mcgill.ca/jargon/jargon.html#heisenbug>

[33] Para una discusión sobre los peligros del código acoplado, consulte el Tema 28, Desacoplamiento.

[34] Vea el consejo aquí.

[35] Según la NHTSA, Distancia de frenado = Distancia de reacción + Distancia de frenado, suponiendo un tiempo de reacción promedio de 1,5 s y una desaceleración de 17,02 pies/s².

[36] Motif y OpenLook eran estándares GUI para estaciones de trabajo Unix basadas en X-Windows.

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.

Volviendo al Tema 11, Reversibilidad, hablamos sobre los peligros de las decisiones irreversibles. En este capítulo, le diremos cómo tomar decisiones reversibles, para que su código pueda permanecer flexible y adaptable frente a un mundo incierto.

Primero observamos el acoplamiento: las dependencias entre bits de código. El Tema 28, Desacoplamiento, muestra cómo mantener separados los conceptos separados, disminuyendo el acoplamiento.

A continuación, veremos las diferentes técnicas que puede usar en el Tema 29, Malabares con el mundo real. Examinaremos cuatro estrategias diferentes para ayudar a administrar y reaccionar ante eventos, un aspecto crítico de las aplicaciones de software modernas.

El código de procedimiento tradicional y el orientado a objetos pueden estar demasiado acoplados para sus propósitos. En el Tema 30, Transformación,

En la programación, aprovecharemos el estilo más flexible y claro que ofrecen las canalizaciones de funciones, incluso si su idioma no las admite directamente.

El estilo orientado a objetos común puede tentarlo con otra trampa.

No se deje engañar, o terminará pagando un fuerte Tema 31, Impuesto sobre sucesiones. Exploraremos mejores alternativas para mantener su código flexible y más fácil de cambiar.

Y, por supuesto, una buena manera de mantenerse flexible es escribir menos código. Cambiar el código lo deja abierto a la posibilidad de introducir nuevos errores. El Tema 32, Configuración , explicará cómo sacar los detalles del código por completo, donde se pueden cambiar de forma más segura y sencilla.

Todas estas técnicas lo ayudarán a escribir código que se doble y no se rompa.



Tema 28

desacoplamiento

Cuando tratamos de seleccionar algo por sí mismo, lo encontramos ligado a todo lo demás en el Universo.

John Muir, Mi primer verano en la sierra

En el Tema 8, La esencia del buen diseño, afirmamos que el uso de buenos principios de diseño hará que el código que escriba sea fácil de cambiar.

El acoplamiento es el enemigo del cambio, porque une cosas que deben cambiar en paralelo. Esto hace que el cambio sea más difícil: o bien dedica tiempo a rastrear todas las piezas que deben cambiarse, o bien dedica tiempo

preguntándose por qué las cosas se rompieron cuando cambiaste “solo una cosa” y no las otras cosas a las que estaba acoplado.

Cuando estás diseñando algo que quieras que sea rígido, tal vez un puente o una torre, unes los componentes:

imágenes/enlaces-01.png

Los enlaces trabajan juntos para hacer que la estructura sea rígida.

Compara eso con algo como esto:

imágenes/enlaces-02.png

Aquí no hay rigidez estructural: los enlaces individuales pueden cambiar y otros simplemente se acomodan.

Cuando estás diseñando puentes, quieres que mantengan su forma; necesitas que sean rígidos. Pero cuando estás diseñando un software que querrás cambiar, quieres exactamente lo contrario: quieres que sea flexible. Y para ser flexibles, los componentes individuales deben acoplarse a la menor cantidad posible de otros componentes.

Y, para empeorar las cosas, el acoplamiento es transitivo: si A está acoplado a B y C, y B está acoplado a M y N, y C a X e Y, entonces A está realmente acoplado a B, C, M, N, X y Y.

Esto significa que hay un principio simple que debes seguir:

Consejo 44

El código desacoplado es más fácil de cambiar

Dado que normalmente no codificamos usando vigas de acero y remaches, ¿qué significa desacoplar el código? En esta sección hablaremos de:

- Descarrilamientos de trenes: cadenas de llamadas a métodos
- Globalización: los peligros de las cosas estáticas
- Herencia: por qué la subclasiación es peligrosa

Hasta cierto punto, esta lista es artificial: el acoplamiento puede ocurrir en cualquier momento en que dos piezas de código comparten algo, así que mientras lee lo que sigue, esté atento a los patrones subyacentes para poder aplicarlos a su código . Y esté atento a algunos de los síntomas del acoplamiento:

- Dependencias extravagantes entre módulos o bibliotecas no relacionados.
- Cambios "simples" en un módulo que se propagan a través de módulos no relacionados en el sistema o rompen elementos en otras partes del sistema.
- Desarrolladores que tienen miedo de cambiar el código porque no están seguros de qué podría verse afectado.
- Reuniones donde todos tienen que asistir porque nadie está seguro de quién se verá afectado por un cambio.

RUINAS DEL TREN

Todos hemos visto (y probablemente escrito) código como este:

```
public void applyDiscount(cliente, order_id, descuento) { totales =
    cliente.orders.find(order_id).getTotals();
    totales.grandTotal =
        totales.grandTotal - descuento; totales.descuento = descuento; }
```

Obtenemos una referencia a algunos pedidos de un objeto de cliente, la usamos para encontrar un pedido en particular y luego obtenemos el conjunto de totales para el pedido. Usando esos totales, restamos el descuento del total general del pedido y también los actualizamos con ese descuento.

Este fragmento de código atraviesa cinco niveles de abstracción, desde el cliente hasta las cantidades totales. En última instancia, nuestro código de nivel superior tiene que

sepa que un objeto de cliente expone pedidos, que los pedidos tienen un método de búsqueda que toma una identificación de pedido y devuelve un pedido, y que el objeto de pedido tiene un objeto de totales que tiene getters y setters para totales generales y descuentos. Eso es mucho conocimiento implícito. Pero lo que es peor, son muchas cosas que no pueden cambiar en el futuro si se quiere que este código siga funcionando. Todos los vagones de un tren están acoplados, al igual que todos los métodos y atributos en un choque de trenes.

Imaginemos que el negocio decide que ningún pedido puede tener un descuento de más del 40%. ¿Dónde pondríamos el código que hace cumplir esa regla?

Se podría decir que pertenece a la función `applyDiscount` que acabamos de escribir. Eso es ciertamente parte de la respuesta. Pero con el código tal como está ahora, no puedes saber que esta es la respuesta completa. Cualquier pieza de código, en cualquier lugar, podría establecer campos en el objeto de totales, y si el mantenedor de ese código no recibió la nota, no estaría comprobando la nueva política.

Una forma de ver esto es pensar en las responsabilidades. Seguramente el objeto de totales debería ser el encargado de gestionar los totales.

Y, sin embargo, no lo es: en realidad es solo un contenedor para un montón de campos que cualquiera puede consultar y actualizar.

La solución para eso es aplicar algo que llamamos:

Consejo 45

Di, no preguntes

Este principio dice que no debe tomar decisiones basadas en el estado interno de un objeto y luego actualizar ese objeto.

Si lo hace, destruye totalmente los beneficios de la encapsulación y, en

al hacerlo, se propaga el conocimiento de la implementación a lo largo del código. Entonces, la primera solución para nuestro choque de trenes es delegar el descuento al objeto total:

```
public void applyDiscount(cliente, order_id, descuento) {  
    cliente.orders.find(order_id).getTotals().applyDiscount(descuento);  
}
```

Tenemos el mismo tipo de problema de decir-no-preguntar (TDA) con el objeto del cliente y sus pedidos: no deberíamos obtener su lista de pedidos y buscarlos. En cambio, deberíamos obtener el pedido que queremos directamente del cliente:

```
public void applyDiscount(cliente, order_id, descuento) {  
    cliente  
        .findOrder(order_id).getTotals().applyDiscount(descuento);  
}
```

Lo mismo se aplica a nuestro objeto de pedido y sus totales. ¿Por qué el mundo exterior debería saber que la implementación de una orden utiliza un objeto separado para almacenar sus totales?

```
public void applyDiscount(cliente, order_id, descuento) {  
    cliente  
        .findOrder(order_id).applyDiscount(descuento);  
}
```

Y aquí es donde probablemente nos detendríamos.

En este punto, podría estar pensando que TDA nos obligaría a agregar un método `applyDiscountToOrder(order_id)` a los clientes. Y si

seguido servilmente, lo haría.

Pero TDA no es una ley de la naturaleza; es solo un patrón para ayudarnos a reconocer los problemas. En este caso, nos sentimos cómodos exponiendo el hecho de que un cliente tiene pedidos, y que podemos encontrar uno de esos pedidos preguntándolo al objeto del cliente. Esta es una decisión pragmática.

En cada aplicación hay ciertos conceptos de alto nivel que son universales. En esta aplicación, esos conceptos incluyen clientes y pedidos. No tiene sentido ocultar los pedidos totalmente dentro de los objetos del cliente: tienen existencia propia. Por lo tanto, no tenemos problemas para crear API que exongan objetos de pedido.

[La ley de Deméter](#)

La gente suele hablar de algo llamado la Ley de Deméter, o LoD, en relación con el acoplamiento. La LoD es un conjunto de pautas^[37] escritas a finales de los 80 por Ian Holland. Los creó para ayudar a los desarrolladores del Proyecto Demeter a mantener sus funciones más limpias y desacopladas.

El LoD dice que un método definido en una clase C solo debe llamar:

- Otros métodos de instancia en C
- Sus parámetros
- Métodos en los objetos que crea, tanto en la pila como en el montón
- Variables globales

En la primera edición de este libro dedicamos un tiempo a describir la LoD. En los 20 años intermedios, la flor se ha desvanecido en esa rosa en particular. Ahora no nos gusta la "variable global"

cláusula (por las razones que veremos en la siguiente sección). También descubrimos que es difícil usar esto en la práctica: es un poco como tener que analizar un documento legal cada vez que llamas a un método.

Sin embargo, el principio sigue siendo válido. Solo recomendamos una forma algo más simple de expresar casi lo mismo:

Consejo 46

No encadene llamadas de método

Trate de no tener más de un "." cuando accedes a algo.

Y acceder a algo también cubre los casos en los que usa variables intermedias, como en el siguiente código:

```
# Este es un estilo bastante
pobre cantidad = cliente_pedidos.último().totales().cantidad;

# y esto también...
pedidos = clientes_pedidos;
último = pedidos.último();
totales = ultimo_totales();
cantidad = totales.cantidad;
```

Hay una gran excepción a la regla de un punto: la regla no se aplica si es muy, muy poco probable que cambien las cosas que estás encadenando. En la práctica, se debe considerar que es probable que cambie cualquier cosa en su aplicación. Cualquier cosa en una biblioteca de terceros debe considerarse volátil, particularmente si se sabe que los mantenedores de esa biblioteca cambian las API entre versiones.

Sin embargo, las bibliotecas que vienen con el lenguaje probablemente sean bastante estables, por lo que estaríamos contentos con un código como:

```
personas .sort_by { |persona|
  persona.edad } .primer(10) .mapa { | persona | persona.nombre }
```

Ese código de Ruby funcionó cuando escribimos la primera edición, hace 20 años, y probablemente seguirá funcionando cuando ingresemos a la casa de los antiguos programadores (en cualquier momento...).

Cadenas y canalizaciones

En el Tema 30, [Transformación de la programación](#), hablamos sobre la composición de funciones en canalizaciones. Estas canalizaciones transforman los datos, pasándolos de una función a la siguiente. Esto no es lo mismo que un choque de trenes de llamadas a métodos, ya que no confiamos en detalles de implementación ocultos.

Eso no quiere decir que las tuberías no introduzcan algún acoplamiento: lo hacen. El formato de los datos devueltos por una función en una canalización debe ser compatible con el formato aceptado por el próximo.

Nuestra experiencia es que esta forma de acoplamiento es mucho menos una barrera para cambiar el código que la forma introducida por los choques de trenes.

LOS MAL DE LA GLOBALIZACIÓN

Los datos accesibles globalmente son una fuente insidiosa de acoplamiento entre los componentes de la aplicación. Cada pieza de datos globales actúa como si todos los métodos en su aplicación de repente ganaran un parámetro adicional: después de todo, esos datos globales están disponibles dentro de cada método.

Código de pareja global por muchas razones. La más obvia es que un cambio en la implementación del global afecta potencialmente a todo el código del sistema. En la práctica, por supuesto, el impacto es bastante limitado; el problema realmente se reduce a saber que ha encontrado todos los lugares que necesita cambiar.

Los datos globales también crean acoplamiento cuando se trata de burlarse de su

código aparte.

Mucho se ha hablado de los beneficios de la reutilización de código.

Nuestra experiencia ha sido que la reutilización probablemente no debería ser una preocupación principal al crear código, pero el pensamiento que implica hacer que el código sea reutilizable debería ser parte de su rutina de codificación. Cuando hace que el código sea reutilizable, le proporciona interfaces limpias, desacoplando del resto de su código. Esto le permite extraer un método o módulo sin arrastrar todo lo demás junto con él. Y si su código usa datos globales, se vuelve difícil separarlo del resto.

Verá este problema cuando esté escribiendo pruebas unitarias para código que usa datos globales. Se encontrará escribiendo un montón de código de configuración para crear un entorno global solo para permitir que se ejecute su prueba.

Consejo 47

Evite los datos globales

Los datos globales incluyen Singletons

En la sección anterior tuvimos cuidado de hablar de datos globales y no de variables globales. Eso es porque la gente a menudo nos dice “¡Mira! Sin variables globales. Lo envolví todo como datos de instancia en un objeto único o módulo global” .

Inténtalo de nuevo, Skippy. Si todo lo que tiene es un singleton con un montón de variables de instancia exportadas, entonces todavía son solo datos globales. Solo tiene un nombre más largo.

Entonces, la gente toma este singleton y oculta todos los datos detrás de los métodos. En lugar de codificar `Config.log_level`, ahora dicen `Config.log_level()` o `Config.getLogLevel()`. Esto es mejor, porque

significa que sus datos globales tienen un poco de inteligencia detrás de ellos. Si decide cambiar la representación de los niveles de registro, puede mantener la compatibilidad mediante la asignación entre el nuevo y el antiguo en la API de configuración. Pero aún tiene solo un conjunto de datos de configuración.

Los datos globales incluyen recursos externos

Cualquier recurso externo mutable es un dato global. Si su aplicación utiliza una base de datos, un almacén de datos, un sistema de archivos, una API de servicio, etc., corre el riesgo de caer en la trampa de la globalización. Una vez más, la solución es asegurarse de que siempre envuelve estos recursos detrás del código que controla.

Consejo 48

Si es lo suficientemente importante como para ser global, envuélvalo en una API

LA HERENCIA AÑADE ACOPLAMIENTO

El mal uso de la subclasiﬁcación, donde una clase hereda el estado y el comportamiento de otra clase, es tan importante que lo analizamos en su propia sección, Tema 31, Impuesto sobre sucesiones.

DE NUEVO, TODO SE TRATA DE CAMBIO

El código acoplado es difícil de cambiar: las alteraciones en un lugar pueden tener efectos secundarios en otras partes del código y, a menudo, en lugares difíciles de encontrar que solo salen a la luz un mes más tarde en la producción.

Mantener su código tímido: hacer que solo se ocupe de las cosas que conoce directamente, ayudará a mantener sus aplicaciones desacopladas y eso las hará más susceptibles de cambiar.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 8, La esencia del buen diseño
- Tema 9, DRY—Los males de la duplicación
- Tema 10, Ortogonalidad
- Tema 11, Reversibilidad
- Tema 29, Malabares con el mundo real
- Tema 30, Transformación de la programación
- Tema 31, Impuesto sobre Sucesiones
- Tema 32, Configuración
- Tema 33, Rompiendo el Acoplamiento Temporal
- Tema 34, El estado compartido es un estado incorrecto
- Tema 35, Actores y Procesos
- Tema 36, Pizarrones
- Hablamos de Diga, no pregunte en nuestro artículo de 2003 Construcción de software [38] El arte de la creación de errores.



Tema 29

Malabares con el mundo real

Las cosas no suceden
simplemente; están hechos
para suceder.

john f kennedy

En los viejos tiempos, cuando sus autores todavía tenían su aspecto juvenil, las computadoras no eran particularmente flexibles. Por lo general, organizamos la forma en que interactuamos con ellos en función de sus limitaciones.

Hoy, esperamos más: las computadoras tienen que integrarse en nuestro mundo,

no al revés. Y nuestro mundo es desordenado: las cosas suceden constantemente, las cosas se mueven, cambiamos de opinión, ... Y las aplicaciones que escribimos de alguna manera tienen que averiguar qué hacer.

Esta sección se trata de escribir estas aplicaciones receptivas.

Comenzaremos con el concepto de un evento.

EVENTOS

Un evento representa la disponibilidad de información. Puede provenir del mundo exterior: un usuario que hace clic en un botón o una actualización de cotización de acciones. Puede ser interno: el resultado de un cálculo está listo, finaliza una búsqueda. Incluso puede ser algo tan trivial como buscar el siguiente elemento en una lista.

Sea cual sea la fuente, si escribimos aplicaciones que respondan a eventos y ajustamos lo que hacen en función de esos eventos, esos

las aplicaciones funcionarán mejor en el mundo real. Sus usuarios encontrarán que son más interactivos y las propias aplicaciones harán un mejor uso de los recursos.

Pero, ¿cómo podemos escribir este tipo de aplicaciones? Sin algún tipo de estrategia, rápidamente nos encontraremos confundidos y nuestras aplicaciones serán un lío de código estrechamente acoplado.

Veamos cuatro estrategias que ayudan.

1. Máquinas de estados finitos
2. El patrón del observador 3.
- Publicar/suscribir 4.
- Programación reactiva y secuencias

MÁQUINAS DE ESTADOS FINITOS

Dave descubre que escribe código utilizando una máquina de estados finitos (FSM) casi todas las semanas. Muy a menudo, la implementación de FSM será solo un par de líneas de código, pero esas pocas líneas ayudan a desenredar una gran cantidad de problemas potenciales.

Usar un FSM es trivialmente fácil y, sin embargo, muchos desarrolladores los evitan. Parece existir la creencia de que son difíciles, o que solo se aplican si está trabajando con hardware, o que necesita usar alguna biblioteca difícil de entender. Ninguno de estos es cierto.

La anatomía de un FSM pragmático

Una máquina de estado es básicamente una especificación de cómo manejar eventos. Consiste en un conjunto de estados, uno de los cuales es el estado actual. Para cada estado, enumeramos los eventos que son significativos para ese estado. Para cada uno de esos eventos, definimos el nuevo estado actual del sistema.

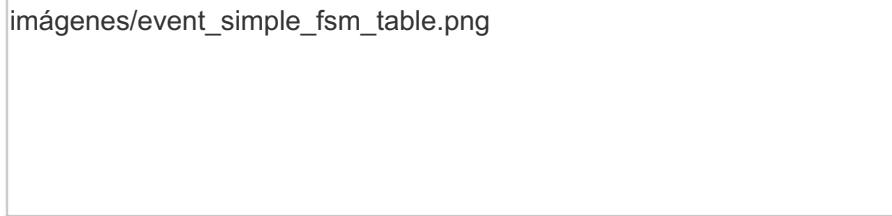
Por ejemplo, podemos estar recibiendo mensajes de varias partes de un websocket. El primer mensaje es un encabezado. A esto le sigue cualquier número de mensajes de datos, seguido de un mensaje final. Esto podría representarse como un FSM como este:

imágenes/eventos_simple_fsm.png

Comenzamos en el “Estado inicial” . Si recibimos un mensaje de encabezado, hacemos la transición al estado "Leyendo mensaje". Si recibimos algo más mientras estamos en el estado inicial (la línea etiquetada con un asterisco), hacemos la transición al estado "Error" y hemos terminado.

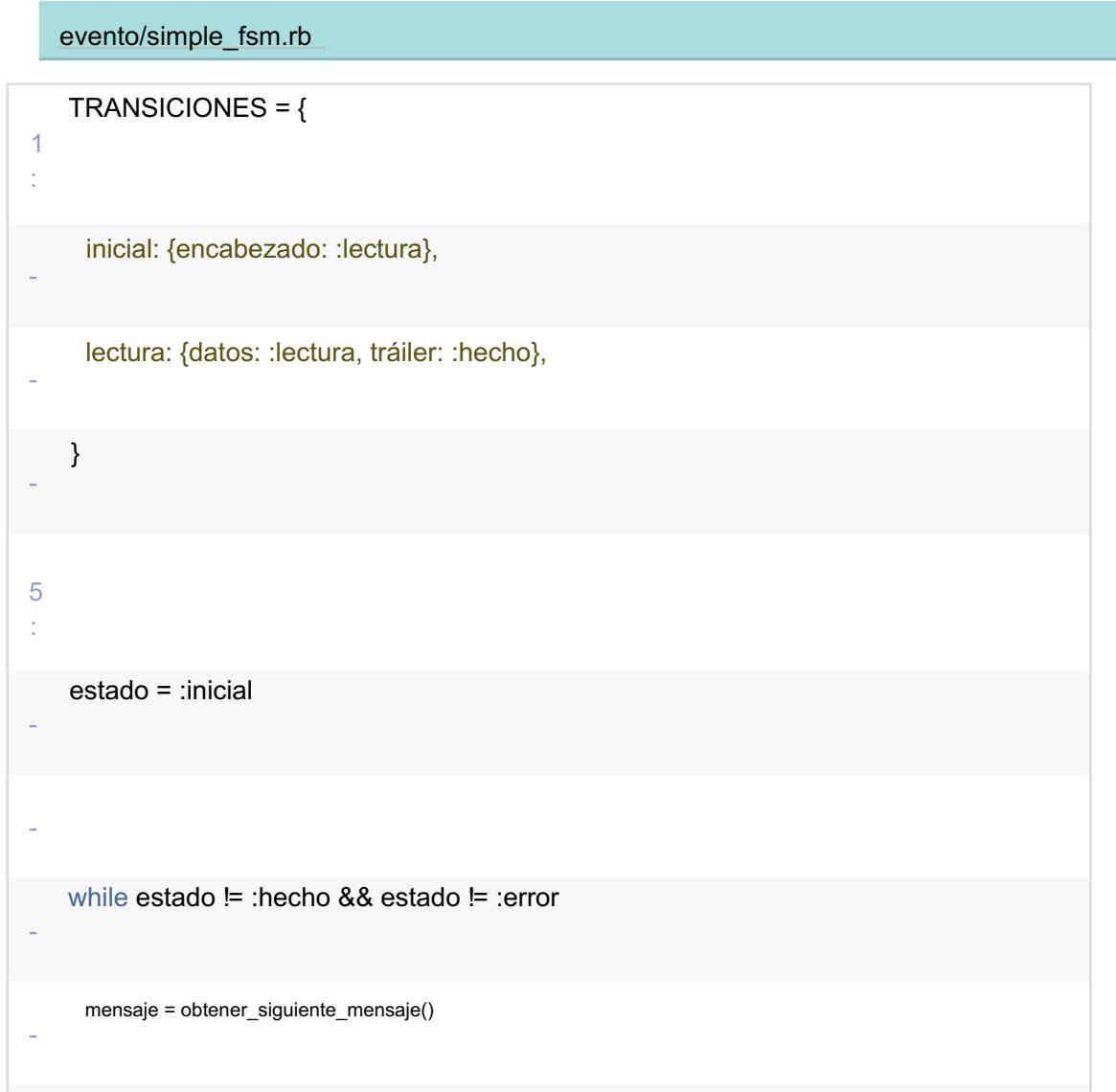
Mientras estamos en el estado "Leyendo mensaje", podemos aceptar mensajes de datos, en cuyo caso continuamos leyendo en el mismo estado, o podemos aceptar un mensaje final, que nos lleva al estado "Terminado". Cualquier otra cosa provoca una transición al estado de error.

Lo bueno de las FSM es que podemos expresarlas puramente como datos. Aquí hay una tabla que representa nuestro analizador de mensajes:

A placeholder box containing the path 'imágenes/event_simple_fsm_table.png'.

Las filas de la tabla representan los estados. Para saber qué hacer cuando ocurre un evento, busque la fila del estado actual, busque la columna que representa el evento, el contenido de esa celda es el nuevo estado.

El código que lo maneja es igualmente simple:

A screenshot of a code editor showing a Ruby script named 'evento/simple_fsm.rb'. The code defines a state transition table and a loop to handle events.

```
TRANSICIONES = {  
  1: {  
    :inicial: {encabezado: :lectura},  
    :lectura: {datos: :lectura, tráiler: :hecho},  
  },  
  5: {  
    :lectura: {tráiler: :hecho},  
    :hecho: {tráiler: :hecho},  
  },  
  :hecho: {tráiler: :error},  
  :error: {},  
}  
  
estado = :inicial  
  
while estado != :hecho && estado != :error  
  mensaje = obtener_siguiente_mensaje()  
  if TRANSICIONES[estado].key?(mensaje)  
    estado = TRANSICIONES[estado][mensaje]  
  end  
end
```

```
    estado = TRANSICIONES[estado][msg.msg_type] || :error
1
0
:
fin
-
```

El código que implementa las transiciones entre estados está en la línea 10. Indexa la tabla de transiciones usando el estado actual y luego indexa las transiciones para ese estado usando el tipo de mensaje. Si no hay un nuevo estado que coincida, establece el estado en : [error](#).

Adición de

acciones Un FSM puro, como el que acabamos de ver, es un analizador de flujo de eventos. Su única salida es el estado final. Podemos reforzarlo agregando acciones que se activan en ciertas transiciones.

Por ejemplo, es posible que necesitemos extraer todas las cadenas en un archivo fuente. Una cadena es texto entre comillas, pero una barra invertida en una cadena escapa al siguiente carácter, por lo que "[Ignorar \"comillas\"](#)" es una sola cadena. Aquí hay un FSM que hace esto:

imágenes/event_string_fsm.png

Esta vez, cada transición tiene dos etiquetas. El de arriba es el

evento que lo desencadena, y el de abajo es la acción a tomar mientras nos movemos entre estados.

Expresaremos esto en una tabla, como lo hicimos la última vez. Sin embargo, en este caso, cada entrada de la tabla es una lista de dos elementos que contiene el siguiente estado y el nombre de una acción:

evento/cadenas_fsm.rb

```
TRANSICIONES = {

    # nuevo estado actual           acción a tomar
    #-----


    buscar_cadena: { =>
        [ :en_cadena, :iniciar_nueva_cadena ], :predeterminado
        => [ :buscar_cadena, :ignorar ], },

    in_string: { =>
        [ :look_for_string, :finish_current_string ], '\\"' =>
        [ :copy_next_char, :add_current_to_string ], :default =>
        [ :in_string, :add_current_to_string ], },

    copy_next_char:
        { :predeterminado => :añadir_actual_a_cadena },
        [ :in_string, ], }


```

También agregamos la capacidad de especificar una transición predeterminada, tomada si el evento no coincide con ninguna de las otras transiciones para este estado.

Ahora veamos el código:

evento/cadenas_fsm.rb

```
estado = :buscar_cadena
```

```
resultado = []

while ch = estado STDIN.getc,
    acción = TRANSICIONES[estado][ch] || TRANSICIONES[estado][:predeterminado] acción de caso

    when :ignore
    when :start_new_string result =
        []
    when :add_current_to_string result <<
        ch
    when :finish_current_string puts
        result.join end end
```

Esto es similar al ejemplo anterior, en el que recorremos los eventos (los caracteres en la entrada), activando transiciones.

Pero hace más que el código anterior. El resultado de cada transición es tanto un nuevo estado como el nombre de una acción. Usamos el nombre de la acción para seleccionar el código que se ejecutará antes de regresar al ciclo.

Este código es muy básico, pero hace el trabajo. Hay muchas otras variantes: la tabla de transición podría usar funciones anónimas o punteros de función para las acciones, podría envolver el código que implementa la máquina de estado en una clase separada, con su propio estado, etc.

No hay nada que decir que tiene que procesar todas las transiciones de estado al mismo tiempo. Si está siguiendo los pasos para registrar a un usuario en su aplicación, es probable que haya una serie de transiciones a medida que ingresan sus detalles, validan su correo electrónico, aceptan las 107 advertencias legisladas diferentes que las aplicaciones en línea ahora deben dar y pronto. Mantener el estado en un almacenamiento externo y usarlo para controlar una máquina de estado es una excelente manera de manejar este tipo de requisitos de flujo de trabajo.

Las máquinas de estado son un comienzo

Las máquinas de estado están infrautilizadas por los desarrolladores y nos gustaría animarle a buscar oportunidades para aplicarlas. Pero no resuelven todos los problemas asociados con los eventos. Así que pasemos a otras formas de ver los problemas de los eventos de malabarismo.

EL PATRÓN DEL OBSERVADOR

En el patrón del observador tenemos una fuente de eventos, llamada observable y una lista de clientes, los observadores, que están interesados en esos eventos.

Un observador registra su interés en lo observable, normalmente pasando una referencia a una función a llamar. Posteriormente, cuando ocurre el evento, el observable itera hacia abajo en su lista de observadores y llama a la función que cada uno pasó. El evento se proporciona como un parámetro para esa llamada.

Aquí hay un ejemplo simple en Ruby. El módulo [Terminator](#) se utiliza para terminar la aplicación. Sin embargo, antes de hacerlo, notifica a todos sus observadores que la aplicación se va a cerrar. [\[39\]](#)
Podrían usar esta notificación para ordenar recursos temporales, confirmar datos, etc.:

evento/observador.rb

terminador de módulo

DEVOLUCIONES DE LLAMADA = []

```
def auto.registro(devolución de llamada)
```

DEVOLUCIONES DE LLAMADA << fin de devolución de

llamada

```
def self.exit(estado_salida)
```

DEVOLUCIONES DE LLAMADA.each { |devolución de llamada| devolución de llamada.(exit_status) }

```
salir!(exit_status) fin fin
```

```
Terminator.register(-> (estado) { pone "la devolución de llamada 1 ve #{estado}" })
```

```
Terminator.register(-> (estado) { pone "la devolución de llamada 2 ve #{estado}" })
```

```
Terminator.exit(99)
```

```
$ ruby event/observer.rb devolución
```

```
de llamada 1 ve 99
```

```
devolución de llamada 2 ve 99
```

No hay mucho código involucrado en la creación de un observable: inserta una referencia de función en una lista y luego llama a esas funciones cuando ocurre el evento. Este es un buen ejemplo de cuándo no usar una biblioteca.

El patrón observador/observable se ha utilizado durante décadas y nos ha resultado muy útil. Prevalece particularmente en los sistemas de interfaz de usuario, donde las devoluciones de llamada se utilizan para informar a la aplicación que se ha producido alguna interacción.

Pero el patrón del observador tiene un problema: debido a que cada uno de los observadores tiene que registrarse con el observable, introduce acoplamiento. Además, debido a que en la implementación típica las devoluciones de llamadas son manejadas en línea por el observable, sincrónicamente, puede introducir cuellos de botella en el rendimiento.

Esto se soluciona con la siguiente estrategia, Publicar/Suscribir.

PUBLICAR/SUSCRIBIRSE

Publish/Subscribe (pubsub) generaliza el patrón del observador, al mismo tiempo que resuelve los problemas de acoplamiento y rendimiento.

En el modelo pubsub, tenemos editores y suscriptores. Estos están conectados a través de canales. Los canales se implementan en un cuerpo de código separado: a veces una biblioteca, a veces un proceso y, a veces, una infraestructura distribuida. Todos estos detalles de implementación están ocultos en su código.

Cada canal tiene un nombre. Los suscriptores registran interés en uno o más de estos canales designados y los editores escriben eventos para ellos. A diferencia del patrón de observador, la comunicación entre el editor y el suscriptor se maneja fuera de su código y es potencialmente asíncrona.

Aunque usted mismo podría implementar un sistema pubsub muy básico, probablemente no quiera hacerlo. La mayoría de los proveedores de servicios en la nube tienen ofertas de pubsub, lo que le permite conectar aplicaciones en todo el mundo. Cada idioma popular tendrá al menos una biblioteca pubsub.

Pubsub es una buena tecnología para desacoplar el manejo de eventos asíncronos. Permite agregar y reemplazar código, posiblemente mientras se ejecuta la aplicación, sin alterar el código existente. La desventaja es que puede ser difícil ver lo que está pasando en un sistema que usa mucho pubsub: no se puede mirar a un editor e inmediatamente ver qué suscriptores están involucrados con un mensaje en particular.

En comparación con el patrón de observador, pubsub es un gran ejemplo de cómo reducir el acoplamiento mediante la abstracción a través de una interfaz compartida (el canal). Sin embargo, sigue siendo básicamente un sistema de paso de mensajes. La creación de sistemas que respondan a combinaciones de eventos necesitará más que esto, así que veamos formas en que podemos agregar una dimensión de tiempo al procesamiento de eventos.

PROGRAMACIÓN REACTIVA, FLUJOS Y EVENTOS

Si alguna vez ha usado una hoja de cálculo, estará familiarizado con la programación reactiva. Si una celda contiene una fórmula que hace referencia a una segunda celda, la actualización de esa segunda celda hace que la primera también se actualice. Los valores reaccionan a medida que cambian los valores que utilizan.

Hay muchos marcos que pueden ayudar con este tipo de reactividad a nivel de datos: en el ámbito del navegador, React y Vue.js son los favoritos actuales (pero, al ser JavaScript, esta información estará desactualizada antes de que se publique este libro). incluso impreso).

Está claro que los eventos también se pueden usar para desencadenar reacciones en el código, pero no es necesariamente fácil conectarlos. Ahí es donde entran las secuencias .

Los flujos nos permiten tratar los eventos como si fueran una colección de datos. Es como si tuviéramos una lista de eventos, que se alarga cuando llegan nuevos eventos. La belleza de eso es que podemos tratar los flujos como cualquier otra colección: podemos manipular, combinar, filtrar y hacer todas las demás cosas relacionadas con los datos que conocemos tan bien. Incluso podemos combinar flujos de eventos y colecciones regulares. Y las transmisiones pueden ser asíncronas, lo que significa que su código tiene la oportunidad de responder a los eventos a medida que llegan.

La línea de base actual de hecho para el manejo de eventos reactivos se define en el sitio <http://reactivex.io>, que define un conjunto de principios independientes del lenguaje y documenta algunas implementaciones comunes. Aquí usaremos la biblioteca RxJs para JavaScript.

Nuestro primer ejemplo toma dos flujos y los comprime juntos: el

el resultado es una nueva secuencia en la que cada elemento contiene un elemento de la primera secuencia de entrada y un elemento de la otra. En este caso, la primera secuencia es simplemente una lista de cinco nombres de animales. La segunda transmisión es más interesante: es un temporizador de intervalos que genera un evento cada 500 ms. Debido a que los flujos se comprimen juntos, solo se genera un resultado cuando hay datos disponibles en ambos, por lo que nuestro flujo de resultados solo emite un valor cada medio segundo:

evento/rx0/index.js

```
import * as Observable from 'rxjs'  
import {logValues} from "../rxcommon/logger.js"  
  
let animales = Observable.of("hormiga", "abeja", "gato", "perro", "alce")  
let ticker = Observable.interval(500)  
  
let combinado = Observable.zip(animales, ticker)  
  
combinado.subscribe(siguiente => logValues(JSON.stringify(siguiente)))
```

Este código utiliza una función de registro simple que [40] agrega elementos a una lista en la ventana del navegador. Cada elemento tiene una marca de tiempo con el tiempo en milisegundos desde que el programa comenzó a ejecutarse. Esto es lo que muestra para nuestro código:



imágenes/eventos_rxjs_0.png

Observe las marcas de tiempo: recibimos un evento de la transmisión cada 500 ms.

Cada evento contiene un número de serie (creado por el `intervalo` observable) y el nombre del siguiente animal de la lista. Al verlo en vivo en un navegador, las líneas de registro aparecen cada medio segundo.

Los flujos de eventos normalmente se completan a medida que ocurren los eventos, lo que implica que los observables que los completan pueden ejecutarse en paralelo. Aquí hay un ejemplo que obtiene información sobre los usuarios de un sitio remoto. Para esto usaremos <https://reqres.in>, un sitio público que proporciona una interfaz REST abierta. Como parte de su API, podemos obtener datos sobre un usuario (falso) en particular realizando una solicitud GET a `usuarios/«id»`. Nuestro código busca a los usuarios con los ID 3, 2 y 1:

evento/rx1/index.js

```
importar * como Observable de
'rxjs' importar { mergeMap } de 'rxjs/operators'
importar {ajax} de 'rxjs/ajax'
```

```
importar {logValues} desde "../rxcommon/logger.js"

let usuarios = Observable.of(3, 2, 1)

let result =
  users.pipe( mergeMap((usuario) => ajax.getJSON(`https://reqres.in/api/users/${user}`)) )

result.subscribe( resp
  => logValues(JSON.stringify(resp.data)), err =>
  console.error(JSON.stringify(err)) )
```

Los detalles internos del código no son demasiado importantes. Lo emocionante es el resultado, que se muestra en la siguiente captura de pantalla:

imágenes/eventos_tres_usuarios.png

Mire las marcas de tiempo: las tres solicitudes, o tres secuencias separadas, se procesaron en paralelo. La primera en regresar, para id 2, tomó 82 ms, y las dos siguientes regresaron 50 y 51 ms más tarde.

Los flujos de eventos son colecciones asíncronas En el ejemplo anterior, nuestra lista de ID de usuario (en los [usuarios observables](#)) era estática. Pero no tiene que ser así. Tal vez queramos recopilar esta información cuando las personas inicien sesión en nuestro sitio. Todo lo que tenemos que hacer es generar un evento observable que contenga su ID de usuario cuando se crea su sesión, y usar ese observable en lugar del estático. Entonces estaríamos obteniendo detalles sobre el

usuarios cuando recibimos estos ID, y presumiblemente almacenándolos en algún lugar.

Esta es una abstracción muy poderosa: ya no necesitamos pensar en el tiempo como algo que tenemos que administrar. Los flujos de eventos unifican el procesamiento síncrono y asíncrono detrás de una API común y conveniente.

LOS EVENTOS SON UBICUOS

Los eventos están en todas partes. Algunos son obvios: un clic en un botón, un temporizador que expira. Otros lo son menos: alguien que inicia sesión, una línea en un archivo que coincide con un patrón. Pero sea cual sea su origen, el código creado en torno a eventos puede responder mejor y estar mejor desacoplado que su contraparte más lineal.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 28, Desacoplamiento
- Tema 36, Pizarrones

EJERCICIOS

Ejercicio 19 (respuesta posible)

En la sección FSM, mencionamos que puede mover la implementación de la máquina de estado genérica a su propia clase. Esa clase probablemente se inicializaría pasando una tabla de transiciones y un estado inicial.

Intente implementar el extractor de cadenas de esa manera.

Ejercicio 20 (respuesta posible)

¿Cuál de estas tecnologías (quizás en combinación) sería una buena opción para las siguientes situaciones:

- Si recibe tres eventos de caída de la interfaz de red en cinco minutos, notifique al personal de operaciones.
- Si es después de la puesta del sol y se detecta movimiento en la parte inferior de las escaleras seguido de movimiento detectado en la parte superior de las escaleras, encienda las luces de arriba.
- Desea notificar a varios sistemas de informes que se completó un pedido.
- Para determinar si un cliente califica para un préstamo de automóvil, la aplicación debe enviar solicitudes a tres servicios de backend y esperar las respuestas.



Tema 30

Transformando la Programación

Si no puede describir lo que está haciendo como un proceso, no sabe lo que está haciendo.

W. Edwards Deming, (atribuido)

Todos los programas transforman datos, convirtiendo una entrada en una salida. Y, sin embargo, cuando pensamos en diseño, rara vez pensamos en crear transformaciones. En cambio, nos preocupamos por las clases y los módulos, las estructuras de datos y los algoritmos, los lenguajes y los marcos.

Creemos que este enfoque en el código a menudo pierde el punto: necesitamos volver a pensar en los programas como algo que transforma las entradas en salidas. Cuando lo hacemos, muchos de los detalles que antes nos preocupaban simplemente se evaporan. La estructura se vuelve más clara, el manejo de errores más consistente y el acoplamiento se reduce.

Para comenzar nuestra investigación, llevemos la máquina del tiempo a la década de 1970 y pidamos a un programador de Unix que nos escriba un programa que enumere los cinco archivos más largos en un árbol de directorios, donde más largo significa "tener la mayor cantidad de líneas".

Podrías esperar que buscaran un editor y comenzaran a escribir en C. Pero no lo harían, porque están pensando en esto en términos de lo que tenemos (un árbol de directorios) y lo que queremos (una lista de archivos). Luego irían a una terminal y escribirían algo como:

```
$ encontrar . -tipo f | xargs wc -l | ordenar -n | cola -5
```

Esta es una serie de transformaciones:

`encontrar .`

`-type f` Escribe una lista de todos los archivos (`-type f`) en o debajo del directorio actual (`.`) a la salida estándar.

`xargs wc -l`

Lee líneas de la entrada estándar y organiza para que todas se pasen como argumentos al comando `wc -l`. El programa `wc` con la opción `-l` cuenta el número de líneas en cada uno de sus argumentos y escribe cada resultado como "nombre de archivo de recuento" en la salida estándar.

`ordenar -n`

Ordene la entrada estándar asumiendo que cada línea comienza con un número (`-n`), escribiendo el resultado en la salida estándar.

`cola -5`

Lea la entrada estándar y escriba solo las últimas cinco líneas en la salida estándar.

Ejecute esto en el directorio de nuestro libro y obtendremos

```
470 ./test_to_build.pml
487 ./dbc.pml
719 ./domain_languages.pml 727 ./
dry.pml 9561
total
```

Esa última línea es el número total de líneas en todos los archivos (no solo los que se muestran), porque eso es lo que hace `wc` . Podemos eliminarlo solicitando una línea más de la `cola` y luego ignorando la última línea:

```
$ encontrar . -tipo f | xargs wc -l | ordenar -n | cola -6 |
head -5 470 ./
debug.pml 470 ./
test_to_build.pml
487 ./dbc.pml 719 ./
domain_languages.pml 727 ./dry.pml
```

imágenes/tubería-wc.png

Figura 1. La tubería de búsqueda como una serie de transformaciones

Miremos esto en términos de los datos que fluyen entre los pasos individuales.

Nuestro requisito original, "los 5 archivos principales en términos de líneas", se convierte en una serie de transformaciones (que también se muestran en la figura).

nombre del directorio
→ lista de
archivos → lista con números
de línea → lista
ordenada → cinco más altos
+ total → cinco más altos

Es casi como una línea de montaje industrial: alimenta datos sin procesar en uno

final y el producto terminado (información) sale por el otro.

Y nos gusta pensar en todo el código de esta manera.

Consejo 49

La programación se trata de código, pero los programas son
Acerca de los datos

ENCONTRAR TRANSFORMACIONES

A veces, la forma más fácil de encontrar las transformaciones es comenzar con el requisito y determinar sus entradas y salidas.

Ahora ha definido la función que representa el programa general. Luego puede encontrar pasos que lo lleven de la entrada a la salida. Este es un enfoque de arriba hacia abajo .

Por ejemplo, desea crear un sitio web para personas que juegan juegos de palabras que encuentre todas las palabras que se pueden formar a partir de un conjunto de letras. Su entrada aquí es un conjunto de letras, y su salida es una lista de palabras de tres letras, palabras de cuatro letras, etc.:

"Ivyin"

se transforma en →

3 => hiedra, lin, cero, yin

4 => solo, liny, viny

5 => vinilo

(Sí, todas son palabras, al menos según el diccionario de macOS).

El truco detrás de la aplicación general es simple: tenemos un diccionario que agrupa palabras por una firma, elegida para que todas

las palabras que contengan las mismas letras tendrán la misma firma. La función de firma más simple es solo la lista ordenada de letras en la palabra. Luego podemos buscar una cadena de entrada generando una firma para ella y luego ver qué palabras (si las hay) en el diccionario tienen esa misma firma.

Por lo tanto, el buscador de anagramas se divide en cuatro transformaciones separadas:

Ste	Transformación	Data de muestra
pag		
Paso 0:	entrada inicial	"ylvin"
Paso 1:	Todas las combinaciones de tres o más letras	vin, viy, vil, vny, vnl, vyl, iny, inl, iyl, nyl, viny, vinl, viyl, vnyl, inyl, vinilo
Paso 2:	Firmas de las combinaciones	inv, hiedra, ilv, nvy, Inv, ivy, iny, iln, ily, Iny, invy, ilnv, ilvy, Invy, ilny, ilnvy
Paso 3:	Lista de todas las palabras del diccionario que coinciden con cualquiera de las firmas	hiedra, yin, nulo, lin, viny, liny, solo, vinilo
Etapa 4:	Palabras agrupadas por longitud	3 => hiedra, lin, cero, yin 4 => solo, liny, viny 5 => vinilo

Transformaciones hasta el final

Comencemos mirando el paso 1, que toma una palabra y crea una lista de todas las combinaciones de tres o más letras. Este paso se puede expresar como una lista de transformaciones:

Paso	Transformación	Data de muestra
Paso 1.0:	entrada inicial	"vinilo"
Paso 1.1: Convertir a caracteres		v, yo, n, y, l
Paso 1.2: Obtener todos los subconjuntos		<ul style="list-style-type: none"> [] [v] [yo] ... [v, i] [v, n] [v, y] ... [v, i, n] [v, i, y] ... [v, n, y, l] [i, n, y, l] [v, i, n, y, l]
Paso 1.3:	Solo aquellos de más de tres caracteres	<ul style="list-style-type: none"> [v, i, n] [v, i, y] ... [i, n, y, l] [v, i, n, y, l]
Paso 1.4: Convertir de nuevo a cadenas		[vin, viy, ... inyl, vinyl]

Ahora hemos llegado al punto en el que podemos implementar fácilmente cada transformación en el código (usando Elixir en este caso):

```
canalizaciones de funciones/anagramas/lib/anagramas.ex
```

```
defp all_subsets_longer_than_tres_caracteres(word) do word |>
  String.codepoints()
  |> Comb.subconjuntos()
  |> Stream.filter(fn subconjunto -> longitud(subconjunto) >= 3 fin)
  |> Stream.map(&List.to_string(&1)) end
```

¿Qué pasa con el `|>` Operador?

Elixir, junto con muchos otros lenguajes funcionales, tiene un operador de tubería, a veces llamado tubería directa o simplemente tubería⁴⁴¹. Todo lo que hace es tomar el valor a su izquierda e insertarlo como el primer parámetro de la función a su derecha, entonces

```
"vinilo" |> String.codepoints |> Comb.subsets()
```

es lo mismo que escribir

```
Comb.subsets(String.codepoints("vinilo"))
```

(Otros idiomas pueden inyectar este valor canalizado como el último parámetro de la siguiente función; depende en gran medida del estilo de las bibliotecas integradas).

Podrías pensar que esto es solo azúcar sintáctico. Pero de una manera muy real, el operador de la tubería es una oportunidad revolucionaria para pensar de manera diferente. El uso de una canalización significa que está pensando automáticamente en términos de transformación de datos; cada vez que ve `|>` en realidad está viendo un lugar donde los datos fluyen entre una transformación y la siguiente.

Muchos lenguajes tienen algo similar: Elm, F# y Swift tienen `|>`, Clojure tiene `->` y `->>` (que funcionan un poco diferente), R tiene `%>%`. Haskell tiene operadores de tuberías y facilita la declaración de otros nuevos. Mientras escribimos esto, se habla de agregar `|>` a JavaScript.

Si su idioma actual admite algo similar, está de suerte. Si no es así, consulte Idioma X no tiene canalizaciones.

De todos modos, volvamos al código.

Sigue Transformando...

Ahora mire el Paso 2 del programa principal, donde convertimos los subconjuntos en firmas. Nuevamente, es una transformación simple: una lista de subconjuntos se convierte en una lista de firmas:

Paso	Transformación	Data de muestra
Paso 2.0:	entrada inicial	vin, viy, ...inyl, vinilo
Paso 2.1:	convertir a firmas	inv, ivy... ilny, inlv

El código de Elixir en la siguiente lista es igual de simple:

canalizaciones de funciones/anagramas/lib/anagramas.ex

```
defp as_unique_signatures(subconjuntos)
  do
    subconjuntos > Stream.map(&Dictionary.signature_of/
      1) end
```

Ahora transformamos esa lista de firmas: cada firma se asigna a la lista de palabras conocidas con la misma firma, o **cero** si no existen tales palabras. Luego tenemos que eliminar los **ceros** y aplanar las listas anidadas en un solo nivel:

canalizaciones de funciones/anagramas/lib/anagramas.ex

```
defp find_in_dictionary(firmas) hacer firmas |
  >
  Stream.map(&Dictionary.lookup_by_signature/1)
```

```

    ➤ Corriente.rechazar(&is_nil/1)
    ➤ Corriente.concat(&(&1)) end

```

El paso 4, agrupar las palabras por longitud, es otra transformación simple, convirtiendo nuestra lista en un mapa donde las claves son las longitudes y los valores son todas las palabras con esa longitud:

canalizaciones de funciones/anagramas/lib/anagramas.ex

```

defp group_by_length(palabras) hacer
  palabras
    ➤ Enum.sort()
    ➤ Enum.group_by(&String.longitud/1) end

```

El lenguaje X no tiene canalizaciones

Los oleoductos existen desde hace mucho tiempo, pero solo en lenguajes de nicho. Recientemente se han convertido en la corriente principal y muchos lenguajes populares aún no admiten el concepto.

La buena noticia es que pensar en transformaciones no requiere una sintaxis de lenguaje particular: es más una filosofía de diseño. Todavía construyes tu código como transformaciones, pero las escribes como una serie de asignaciones:

```

const contenido = Archivo.leer(nombre_archivo);

const líneas = find_matching_lines(contenido, patrón)

const resultado = truncar_líneas(líneas)

```

Es un poco más tedioso, pero hace el trabajo.

Poniendo todo junto Hemos

escrito cada una de las transformaciones individuales. Ahora es el momento de encadenarlos todos juntos en nuestra función principal:

canalizaciones de funciones/anagramas/lib/anagramas.ex

```
def anagramas_en(palabra) hacer
```

```

palabra |> todos_subconjuntos_más_más_de_tres_caracteres()
|> como_firmas_únicas() |>
buscar_en_diccionario() |>
agrupar_por_longitud()
end

```

¿Funciona? Vamos a intentarlo:

```

iex(1)> Anagrams.anagrams_in "lyvin" %{ 3 =>

["ivy", "lin", "nil", "yin"], 4 => ["inly",
"liny", "viny"] , 5 => ["vinilo"] }

```

¡POR QUÉ ES ESTO TAN GRANDE?

Veamos de nuevo el cuerpo de la función principal:

```

palabra |> todos_subconjuntos_más_más_de_tres_caracteres()
|> como_firmas_únicas() |>
buscar_en_diccionario() |>
agrupar_por_longitud()

```

Es simplemente una cadena de las transformaciones necesarias para cumplir con nuestro requisito, cada una de las cuales toma la entrada de la transformación anterior y pasa la salida a la siguiente. Eso es lo más cercano a un código alfabetizado que puede obtener.

Pero también hay algo más profundo. Si su experiencia es la programación orientada a objetos, entonces sus reflejos exigen que oculte datos, encapsulándolos dentro de objetos. Estos objetos luego parlotean de un lado a otro, cambiando el estado del otro. Esto introduce mucho acoplamiento, y es una gran razón por la que los sistemas OO pueden ser difíciles de cambiar.

Consejo 50

No atesore el estado; Pásalo

En el modelo transformacional, le damos la vuelta a eso. En lugar de pequeños grupos de datos repartidos por todo el sistema, piense en los datos como un río poderoso, un flujo. Los datos se vuelven iguales a la funcionalidad: una canalización es una secuencia de código → datos → código → datos.... Los datos ya no están vinculados a un grupo particular de funciones, como lo están en una definición de clase. En cambio, es libre de representar el progreso de desarrollo de nuestra aplicación a medida que transforma sus entradas en sus salidas. Esto significa que podemos reducir en gran medida el acoplamiento: una función se puede usar (y reutilizar) en cualquier lugar donde sus parámetros coincidan con la salida de alguna otra función.

Sí, todavía hay un grado de acoplamiento, pero en nuestra experiencia es más manejable que el estilo OO de comando y control.

Y, si está utilizando un lenguaje con verificación de tipos, recibirá advertencias en tiempo de compilación cuando intente conectar dos elementos incompatibles.

¿QUÉ PASA CON EL MANEJO DE ERRORES?

Hasta ahora, nuestras transformaciones han funcionado en un mundo donde nada sale mal. Sin embargo, ¿cómo podemos usarlos en el mundo real? Si solo podemos construir cadenas lineales, ¿cómo podemos agregar toda esa lógica condicional que necesitamos para la verificación de errores?

Hay muchas formas de hacer esto, pero todas se basan en una convención básica: nunca pasamos valores sin procesar entre transformaciones.

En cambio, los envolvemos en una estructura de datos (o tipo) que también nos dice si el valor contenido es válido. En Haskell, por ejemplo, este envoltorio se llama [Quizás](#). En F # y Scala es [Opción](#).

La forma en que usa este concepto es específica del idioma. En general, sin embargo, hay dos formas básicas de escribir el código: puede

manejar la comprobación de errores dentro de sus transformaciones o fuera de ellas.

Elixir, que hemos usado hasta ahora, no tiene este soporte incorporado. Para nuestros propósitos, esto es algo bueno, ya que podemos mostrar una implementación desde cero. Algo similar debería funcionar en la mayoría de los demás idiomas.

Primero, elija una representación

Necesitamos una representación para nuestro envoltorio (la estructura de datos que lleva un valor o una indicación de error). Puedes usar estructuras para esto, pero Elixir ya tiene una convención bastante fuerte: las funciones tienden a devolver una tupla que contiene `{:ok, valor}` o `{:error, razón}`. Por ejemplo, `File.open` devuelve `:ok` y un proceso de E/S o `:error` y un código de motivo:

```
iex(1)> Archivo.open("/etc/passwd") {:ok,
#PID<0.109.0>} iex(2)>
Archivo.open("/etc/wombat") {:error, :enoent}
```

Usaremos la tupla `:ok/:error` como nuestro envoltorio cuando pasemos cosas a través de una canalización.

Luego manéjelo dentro de cada transformación

Escribamos una función que devuelva todas las líneas de un archivo que contienen una cadena dada, truncada a los primeros 20 caracteres. Queremos escribirlo como una transformación, por lo que la entrada será un nombre de archivo y una cadena para que coincida, y la salida será una tupla `:ok` con una lista de líneas o una tupla `:error` con algún tipo de motivo. La función de nivel superior debería verse así:

canalizaciones de funciones/anagramas/lib/grep.ex

```
def find_all(nombre_de_archivo, patrón) hacer
```

```
File.read(file_name) |>
  find_matching_lines(patrón) |>
  truncate_lines() end
```

No hay una verificación de error explícita aquí, pero si algún paso en la canalización devuelve una tupla de error, la canalización devolverá ese error sin ejecutar las funciones que siguen. Hacemos esto^[42] usando la coincidencia de patrones de Elixir:

canalizaciones de funciones/anagramas/lib/grep.ex

```
defp find_matching_lines({:ok, contenido}, patrón) hacer
  contenido
    |> Cadena.split(~r\n/)
    |> Enum.filter(&String.coincidencia?(&1, patrón)) |> ok_a
      menos que_vacio() end

defp find_matching_lines(error, _), do: error

# -------

defp truncate_lines({:ok, lines }) do lines |>
  Enum.map(&String.slice(&1, 0, 20)) |> ok() end

defp truncar_líneas(error), hacer: error

# -------

defp ok_unless_empty([], do: error("no se encontró nada") defp
  ok_unless_empty(resultado), do: ok(resultado)

defp ok(resultado), do: { :ok, resultado } defp
  error(motivo), do: { :error, motivo }
```

Eche un vistazo a la función `find_matching_lines`. Si su primer parámetro es una tupla `:ok`, usa el contenido de esa tupla para encontrar

líneas que coincidan con el patrón. Sin embargo, si el primer parámetro no es una tupla `:ok`, se ejecuta la segunda versión de la función, que solo devuelve ese parámetro. De esta forma, la función simplemente reenvía un error por la canalización. Lo mismo se aplica a [truncate_lines](#).

Podemos jugar con esto en la consola:

```
iex> Grep.find_all "/etc/passwd", ~r/www/  
{:ok, ["_www:*:70:70:World W", "_wwwproxy*:252:252:"]}
```

```
iex> Grep.find_all "/etc/passwd", ~r/wombat/  
{:error, "no se encontró  
nada"} iex> Grep.find_all "/etc/koala", ~r/  
www/ {:error, :enoent}
```

Puede ver que un error en cualquier parte de la canalización se convierte inmediatamente en el valor de la canalización.

O manéjelo en la canalización

Es posible que esté mirando las funciones [find_matching_lines](#) y [truncate_lines](#) pensando que hemos trasladado la carga del manejo de errores a las transformaciones. Tendrías razón. En un lenguaje que utiliza la coincidencia de patrones en las llamadas a funciones, como Elixir, el efecto se reduce, pero sigue siendo feo.

Sería bueno si Elixir tuviera una versión del operador de tubería `>` que conociera las tuplas `:ok/:error` y que cortocircuitara la ejecución cuando ocurriera un error. Pero el hecho de que `not`^[42] nos permite agregar algo similar, y de una manera que es aplicable a una serie de otros idiomas.

El problema al que nos enfrentamos es que cuando ocurre un error, no queremos ejecutar el código más adelante en la canalización y no queremos que el código sepa que esto está sucediendo. Esto significa que nosotros

necesitamos aplazar la ejecución de funciones de canalización hasta que sepamos que los pasos anteriores en la canalización se realizaron correctamente. Para hacer esto, necesitaremos cambiarlos de llamadas de función a valores de función que se puedan llamar más tarde. Aquí hay una implementación:

canalizaciones de funciones/anagramas/lib/grep1.ex

```

defmodule Grep1 hacer

  def and_then({ :ok, value }, func), do: func.(value) def
  and_then(anything_else, _func), do: anything_else

  def buscar_todos(nombre_archivo, patrón) do
    Archivo.leer(nombre_archivo)
    > and_then(&buscar_líneas_coincidentes(&1, patrón)) >
    and_then(&truncar_líneas(&1)) end

  defp find_matching_lines(contenido, patrón) hacer
    contenido
    > Cadena.split(~r/\n/)
    > Enum.filter(&String.coincidencia?(&1, patrón)) > ok_a
      menos que_vacío() end

  defp truncar_líneas(líneas) hacer
    líneas
    > Enum.map(&String.slice(&1, 0, 20)) > ok() end

  defp ok_unless_empty([], do: error("no se encontró nada")) defp
  ok_unless_empty(resultado), do: ok(resultado)

  defp ok(resultado), do: { :ok, resultado } defp
  error(razón), do: { :error, razón } end

```

La función `and_then` es un ejemplo de una función de vinculación : toma un valor envuelto en algo, luego aplica una función a eso

valor, devolviendo un nuevo valor envuelto. El uso de la función `and_then` en la canalización requiere un poco de puntuación adicional porque se debe indicar a Elixir que convierta las llamadas a funciones en valores de función, pero ese esfuerzo adicional se compensa con el hecho de que las funciones de transformación se vuelven simples: cada una solo toma un valor (y cualquier parámetros adicionales) y devuelve `{:ok, new_value}` o `{:error, Reason}`.

TRANSFORMACIONES TRANSFORMAR PROGRAMACIÓN

Pensar en el código como una serie de transformaciones (anidadas) puede ser un enfoque liberador de la programación. Lleva un tiempo acostumbrarse, pero una vez que haya desarrollado el hábito, encontrará que su código se vuelve más limpio, sus funciones más cortas y sus diseños más planos.

Darle una oportunidad.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 8, La esencia del buen diseño
- Tema 17, Juegos de conchas
- Tema 26, Cómo equilibrar los recursos
- Tema 28, Desacoplamiento
- Tema 35, Actores y Procesos

EJERCICIOS

Ejercicio 21 (respuesta posible)

¿Puede expresar los siguientes requisitos como una transformación de nivel superior? Es decir, para cada uno, identifique la entrada y la salida.

1. Los impuestos de envío y ventas se agregan a un pedido
2. Su aplicación carga información de configuración de un archivo con nombre 3. Alguien inicia sesión en una aplicación web

Ejercicio 22 (respuesta posible)

Ha identificado la necesidad de validar y convertir un campo de entrada de una cadena en un número entero entre 18 y 150. La transformación general se describe mediante

```
contenido del campo como cadena  
→ [validar y convertir] →  
{:ok, valor} | {:error, motivo}
```

Escriba las transformaciones individuales que componen validar & convertir.

Ejercicio 23 (respuesta posible)

En Language X Doesn't Have Pipelines escribimos:

```
const contenido =  
Archivo.leer(nombre_archivo); const líneas =  
find_matching_lines(contenido, patrón) const resultado = truncate_lines(líneas)
```

Muchas personas escriben código orientado a objetos encadenando llamadas a métodos, y podrían tener la tentación de escribir esto de la siguiente manera:

```
const resultado =  
content_of(file_name).find_matching_lines(patrón).truncate_lines()
```

¿Cuál es la diferencia entre estas dos piezas de código? ¿Cuál crees que preferimos?



Tema 31

Impuestos de sucesión

Querías un plátano

pero lo que obtuviste fue un
gorila sosteniendo el
plátano y todo el
selva.

joe armstrong

¿Programas en un lenguaje orientado
a objetos? ¿Utiliza la herencia?

Si es así, ¡detente! Probablemente no es
lo que quieras hacer.

Veamos por qué.

ALGUNOS ANTECEDENTES

La herencia apareció por primera vez en Simula 67 en 1969. Era una solución elegante al problema de poner en cola varios tipos de eventos en la misma lista. El enfoque de Simula era usar algo llamado clases de prefijo. Podrías escribir algo como esto:

```
enlazar coche
CLASE; ... implementación de coche

enlace CLASE bicicleta; ...
implementación de bicicleta
```

El [enlace](#) es una clase de prefijo que agrega la funcionalidad de las listas enlazadas. Esto le permite agregar automóviles y bicicletas a la lista de cosas que esperan en (digamos) un semáforo. En la terminología actual, [enlace](#) sería una clase principal.

El modelo mental utilizado por los programadores de Simula era que los datos de la instancia y la implementación de la clase [link](#) se anteponían a la implementación de las clases [auto](#) y [bicicleta](#). La parte [del enlace](#) se consideraba casi como un contenedor que transportaba automóviles y bicicletas. Esto les dio una forma de polimorfismo: los automóviles y las bicicletas implementaron la interfaz [de enlace](#) porque ambos contenían el código [de enlace](#).

Después de Simula llegó Smalltalk. Alan Kay, uno de los creadores de Smalltalk, describe en una respuesta de Quora de 2019 ^[44] por qué Smalltalk tiene herencia:

Entonces, cuando diseñé Smalltalk-72, y fue divertido pensar en Smalltalk-71, pensé que sería divertido usar su dinámica similar a Lisp para hacer experimentos con "programación diferencial" (es decir: varias formas de lograr "esto es así excepto").

Esta es una subclasificación puramente por comportamiento.

Estos dos estilos de herencia (que en realidad tenían bastante en común) se desarrollaron durante las siguientes décadas. El enfoque de Simula, que sugería que la herencia era una forma de combinar tipos, continuó en lenguajes como C++ y Java.

La escuela Smalltalk, donde la herencia era una organización dinámica de comportamientos, se vio en lenguajes como Ruby y JavaScript.

Entonces, ahora nos enfrentamos a una generación de desarrolladores OO que usan la herencia por una de dos razones: no les gusta escribir o les gustan los tipos.

Aquellos a los que no les gusta escribir, guárdense los dedos usando

herencia para agregar funcionalidad común de una clase base a clases secundarias: la clase [Usuario](#) y la clase [Producto](#) son ambas subclases de [ActiveRecord::Base](#).

Aquellos a los que les gustan los tipos utilizan la herencia para expresar la relación entre clases: un [Coche](#) es-un-tipo-de [-Vehículo](#).

Desafortunadamente, ambos tipos de herencia tienen problemas.

PROBLEMAS AL UTILIZAR HERENCIA PARA COMPARTIR CÓDIGO

La herencia es el acoplamiento. La clase secundaria no solo está acoplada al padre, al padre del padre, etc., sino que el código que usa el hijo también está acoplado a todos los ancestros. Aquí hay un ejemplo:

```
clase Vehículo
  def inicializar
    @velocidad =
    0 fin
  def detener
    @velocidad =
    0 fin
  def move_at(velocidad)
    @velocidad =
      velocidad fin fin

clase Coche < Información de
  definición del vehículo
    "Conduzco un automóvil a #{@velocidad}"
  end
end

# código de nivel
superior my_ride =
Car.new my_ride.move_at(30)
```

Cuando el nivel superior llama a [my_car.move_at](#), el método que se invoca está en [Vehicle](#), el parente de [Car](#).

Ahora, el desarrollador a cargo de [Vehicle](#) cambia la API, por lo que [move_at](#) se convierte en [set_velocity](#) y la variable de instancia [@speed](#) se convierte en [@velocity](#).

Se espera que un cambio de API rompa los clientes de la clase [Vehicle](#). Pero el nivel superior no lo es: en lo que a él se refiere, está usando un [automóvil](#). Lo que hace la clase [Car](#) en términos de implementación no es la preocupación del código de nivel superior, pero aún falla.

De manera similar, el nombre de una variable de instancia es puramente un detalle de implementación interna, pero cuando [Vehicle](#) cambia, también (silenciosamente) rompe [Car](#).

Tanto acoplamiento.

Problemas al usar la herencia para construir tipos

Algunas personas ven la herencia como una forma de definir nuevos tipos. Su diagrama de diseño favorito muestra jerarquías de clases. Ven los problemas de la misma manera que los científicos victorianos veían la naturaleza, como algo que se puede dividir en categorías.



Desafortunadamente, estos diagramas pronto se convierten en monstruosidades que cubren las paredes, agregadas capa sobre capa para expresar el más mínimo matiz de diferenciación entre clases. Esta complejidad adicional puede hacer que la aplicación sea más frágil, ya que los cambios pueden extenderse hacia arriba y hacia abajo en muchas capas.

Aún peor, sin embargo, es el problema de la herencia múltiple. Un [automóvil](#) puede ser un tipo de [vehículo](#), pero también puede ser un tipo de [activo](#), [artículo](#) [asegurado](#), [garantía de préstamo](#), etc. Modelar esto correctamente necesitaría

herencia múltiple.

C++ le dio mala fama a la herencia múltiple en la década de 1990 debido a algunas semánticas de desambiguación cuestionables. Como resultado, muchos lenguajes OO actuales no lo ofrecen. Por lo tanto, incluso si está satisfecho con los árboles de tipos complejos, de todos modos no podrá modelar su dominio con precisión.

Consejo 51

No pagues impuesto de sucesiones

LAS ALTERNATIVAS SON MEJORES

Permítanos sugerir tres técnicas que significan que nunca más necesitará usar la herencia:

- Interfaces y protocolos
- Delegación
- Mezclas y rasgos

[Interfaces y Protocolos](#)

La mayoría de los lenguajes OO le permiten especificar que una clase implementa uno o más conjuntos de comportamientos. Se podría decir, por ejemplo, que una clase [Coche](#) implementa el comportamiento [Conducible](#) y el comportamiento [Localizable](#). La sintaxis utilizada para hacer esto varía: en Java, podría verse así:

```
Implementos de automóviles de clase pública Manejables, Localizables {  
  
    // Código para la clase Coche. Este código debe  
    incluir // la funcionalidad de Driveble //  
    y Locatable  
  
}
```

[Driveble](#) y [Locatable](#) son lo que Java llama interfaces; otros lenguajes los llaman protocolos, y algunos los llaman rasgos (aunque esto no es lo que llamaremos un rasgo más adelante).

Las interfaces se definen así:

```
interfaz pública manejable  
{ doble getSpeed();  
parada vacía  
(); }  
  
interfaz pública Localizable()  
{  
    Coordinar getLocation();  
    ubicación booleana es válida  
(); }
```

Estas declaraciones no crean código: simplemente dicen que cualquier clase que implemente [Driveble](#) debe implementar los dos métodos [getSpeed](#) y [stop](#), y una clase que sea [Locatable](#) debe implementar [getLocation](#) y [locationIsValid](#). Esto significa que nuestra definición de clase anterior de [Car](#) solo será válida si incluye estos cuatro métodos.

Lo que hace que las interfaces y los protocolos sean tan poderosos es que podemos usarlos como tipos, y cualquier clase que implemente la interfaz adecuada será compatible con ese tipo. Si [Car](#) y [Phone](#) implementan [Locatable](#), podríamos almacenar ambos en una lista de elementos localizables:

```
List<Locatable> items = new ArrayList<>();

items.add(nuevo Coche(...));
items.add(nuevo Teléfono(...));
items.add(nuevo Coche(...)); //
...
```

Luego podemos procesar esa lista, con la certeza de que cada elemento tiene [getLocation](#) y [locationIsValid](#):

```
void printLocation(Elemento localizable)
{ if (item.locationIsValid()
    { print(item.getLocation().asString()); }

// ...

items.forEach(printLocation);
```

Consejo 52

Preferir interfaces para expresar polimorfismo

Las interfaces y protocolos nos dan polimorfismo sin herencia.

La herencia

de delegación alienta a los desarrolladores a crear clases cuyos objetos tengan una gran cantidad de métodos. Si una clase principal tiene 20 métodos, y la subclase quiere hacer uso de solo dos de ellos, sus objetos seguirán teniendo los otros 18 simplemente tirados y llamables. La clase ha perdido el control de su interfaz. Esto es un

Problema común: muchos marcos de trabajo de persistencia e interfaz de usuario insisten en que los componentes de la aplicación subclasifiquen alguna clase base proporcionada:

```
clase Cuenta < PersistenciaBaseClase final
```

La clase `Cuenta` ahora lleva consigo toda la API de la clase de persistencia. En su lugar, imagina una alternativa usando delegación, como en el siguiente ejemplo:

```
class Account
  def initialize( . . . )
    @repo = Persister.for(self) end

  def
    guardar
    @repo.save() final final
```

Ahora no exponemos ninguna de las API del marco a los clientes de nuestra clase `Cuenta`: ese desacoplamiento ahora está roto. Pero hay más

Ahora que ya no estamos limitados por la API del marco que estamos usando, somos libres de crear la API que necesitamos.

Sí, podíamos hacer eso antes, pero siempre corríamos el riesgo de que la interfaz que escribimos pudiera pasarse por alto y usar la API de persistencia en su lugar. Ahora lo controlamos todo.

Consejo 53

Delegado a Servicios: Has-A Trumps Is-A

De hecho, podemos llevar esto un paso más allá. ¿Por qué una `cuenta` debe saber cómo persistir? ¿No es su trabajo conocer y hacer cumplir las reglas comerciales de la cuenta?

```
clase Cuenta #
  nada más que cosas de la cuenta
```

fin

```
clase Registro de cuenta
# envuelve una cuenta con la capacidad
# de ser recuperada y
almacenada
```

Ahora estamos realmente desvinculados, pero ha tenido un costo. Tenemos que escribir más código y, por lo general, parte de él será repetitivo: es probable que todas nuestras clases de registro necesiten un método **de búsqueda**, por ejemplo.

Afortunadamente, eso es lo que los mixins y los rasgos hacen por nosotros.

Mixins, rasgos, categorías, extensiones de protocolo, ...

Como industria, nos encanta poner nombres a las cosas. Muy a menudo le daremos muchos nombres a la misma cosa. Más es mejor, ¿verdad?

Eso es a lo que nos enfrentamos cuando miramos a los mixins. La idea básica es simple: queremos poder extender clases y objetos con nueva funcionalidad sin usar la herencia. Así que creamos un conjunto de estas funciones, le damos un nombre a ese conjunto y luego, de alguna manera, extendemos una clase u objeto con ellas. En ese momento, ha creado una nueva clase u objeto que combina las capacidades del original y todos sus complementos. En la mayoría de los casos, podrá realizar esta extensión incluso si no tiene acceso al código fuente de la clase que está extendiendo.

Ahora, la implementación y el nombre de esta función varían según el idioma.

Tenderemos a llamarlos mixins aquí, pero realmente queremos que pienses en esto como una característica agnóstica del idioma.

Lo importante es la capacidad que tienen todas estas implementaciones: fusionar funcionalidad entre cosas existentes y cosas nuevas.

Como ejemplo, volvamos a nuestro ejemplo de [AccountRecord](#). Como lo dejamos, un [AccountRecord](#) necesitaba saber sobre ambas cuentas y sobre nuestro marco de persistencia. También necesitaba delegar todos los métodos en la capa de persistencia que quería exponer al mundo exterior.

Los mixins nos dan una alternativa. Primero, podríamos escribir un mixin que implemente (por ejemplo) dos de los tres métodos de búsqueda estándar. Luego podríamos agregarlos a [AccountRecord](#) como una mezcla.

Y, a medida que escribimos nuevas clases para cosas persistentes, también podemos agregarles el mixin:

```
mezclar CommonFinders { def
    buscar(id) { ... } def
    buscarTodos() { ... } end

class AccountRecord extiende BasicRecord con CommonFinders class OrderRecord extiende
BasicRecord con CommonFinders
```

Podemos llevar esto mucho más lejos. Por ejemplo, todos sabemos que nuestros objetos comerciales necesitan un código de validación para evitar que datos incorrectos se infiltrén en nuestros cálculos. Pero, ¿qué entendemos exactamente por validación?

Si tomamos una cuenta, por ejemplo, probablemente haya muchas capas diferentes de validación que podrían aplicarse:

- Validar que una contraseña codificada coincida con una ingresada por el usuario
- Validación de los datos del formulario ingresados por el usuario cuando se crea una cuenta

- Validando los datos del formulario ingresados por una persona administradora actualizando los detalles del usuario
- Validación de datos agregados a la cuenta por otros componentes del sistema

- Validación de la consistencia de los datos antes de que se conserven

Un enfoque común (y creemos que menos que ideal) es agrupar todas las validaciones en una sola clase (el objeto de negocio/objeto de persistencia) y luego agregar banderas para controlar qué se dispara en qué circunstancias.

Creemos que una mejor manera es usar mixins para crear clases especializadas para situaciones apropiadas:

```
clase AccountForCustomer extiende Cuenta con  
    AccountValidations,AccountCustomerValidations  
  
la clase AccountForAdmin extiende la cuenta con  
    AccountValidations,AccountAdminValidations
```

Aquí, ambas clases derivadas incluyen validaciones comunes a todos los objetos de cuenta. La variante de cliente también incluye validaciones apropiadas para las API orientadas al cliente, mientras que la variante de administrador contenía (presumiblemente menos restrictiva) validaciones de administrador.

Ahora, al pasar instancias de `AccountForCustomer` o `AccountForAdmin` de un lado a otro, nuestro código garantiza automáticamente que se aplique la validación correcta.

Consejo 54

Use Mixins para compartir funcionalidad

LA HERENCIA RARAMENTE ES LA RESPUESTA

Hemos echado un vistazo rápido a tres alternativas a la herencia de clases tradicional:

- Interfaces y protocolos

- Delegación
- Mezclas y rasgos

Cada uno de estos métodos puede ser mejor para usted en diferentes circunstancias, dependiendo de si su objetivo es compartir información de tipo, agregar funcionalidad o compartir métodos. Como con cualquier cosa en la programación, intente utilizar la técnica que mejor exprese su intención.

Y trata de no arrastrar a toda la jungla en el viaje.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 8, La esencia del buen diseño
- Tema 10, Ortogonalidad
- Tema 28, Desacoplamiento

RETOS

- La próxima vez que se encuentre subclasificando, tómese un minuto para examinar las opciones. ¿Puedes lograr lo que quieras con interfaces, delegación y/o mixins? ¿Puedes reducir el acoplamiento al hacerlo?



Tema 32

Configuración

Deja que todas tus cosas tengan su lugar; deja que cada parte de tu negocio tenga su tiempo.

Benjamín Franklin, trece Virtudes, autobiografía

Cuando el código se basa en valores que pueden cambiar después de que la aplicación se haya puesto en marcha, mantenga esos valores externos a la aplicación. Cuando su aplicación se ejecute en diferentes entornos, y potencialmente para diferentes clientes, mantenga los valores específicos del entorno y del cliente fuera de la aplicación. De esta manera, está

parametrizando su aplicación; el código se adapta a los lugares donde se ejecuta.

Consejo 55

Parametriza tu aplicación usando externo Configuración

Las cosas comunes que probablemente querrá poner en los datos de configuración incluyen:

- Credenciales para servicios externos (base de datos, API de terceros, etc.)
- Niveles de registro y destinos
- Nombres de puerto, dirección IP, máquina y clúster que usa la aplicación
- Parámetros de validación específicos del entorno
- Parámetros establecidos externamente, como tasas impositivas
- Detalles de formato específicos del sitio

- Claves de licencia

Básicamente, busque cualquier cosa que sepa que tendrá que cambiar y que pueda expresar fuera de su cuerpo principal de código, y colóquelo en algún cubo de configuración.

CONFIGURACIÓN ESTÁTICA

Muchos marcos y bastantes aplicaciones personalizadas mantienen la configuración en archivos planos o tablas de bases de datos. Si la información está en archivos planos, la tendencia es utilizar algún formato de texto sin formato comercial. Actualmente, YAML y JSON son populares para esto. A veces, las aplicaciones escritas en lenguajes de secuencias de comandos utilizan archivos de código fuente de propósito especial, dedicados a contener solo la configuración. Si la información está estructurada y es probable que el cliente la cambie (tasas de impuestos sobre las ventas, por ejemplo), sería mejor almacenarla en una tabla de base de datos. Y, por supuesto, puede usar ambos, dividiendo la información de configuración según el uso.

Independientemente de la forma que utilice, la configuración se lee en su aplicación como una estructura de datos, normalmente cuando se inicia la aplicación. Comúnmente, esta estructura de datos se hace global, pensando que esto facilita que cualquier parte del código llegue a los valores que contiene.

Preferimos que no hagas eso. En su lugar, envuelva la información de configuración detrás de una API (delgada). Esto desvincula tu código de los detalles de la representación de la configuración.

CONFIGURACIÓN-COMO-UN-SERVICIO

Si bien la configuración estática es común, actualmente preferimos una

enfoque diferente. Todavía queremos que los datos de configuración se mantengan fuera de la aplicación, pero en lugar de en un archivo plano o una base de datos, nos gustaría verlos almacenados detrás de una API de servicio. Esto tiene una serie de beneficios:

- Múltiples aplicaciones pueden compartir información de configuración, con autenticación y control de acceso limitando lo que cada una puede ver
- Los cambios de configuración se pueden hacer globalmente
- Los datos de configuración se pueden mantener a través de una interfaz de usuario especializada
- Los datos de configuración se vuelven dinámicos

Ese último punto, que la configuración debe ser dinámica, es fundamental a medida que avanzamos hacia aplicaciones de alta disponibilidad. La idea de que deberíamos detener y reiniciar una aplicación para cambiar un solo parámetro está irremediablemente fuera de contacto con las realidades modernas. Usando un servicio de configuración, los componentes de la aplicación podrían registrarse para recibir notificaciones de actualizaciones de los parámetros que usan, y el servicio podría enviarles mensajes que contengan nuevos valores si se modifican.

Cualquiera que sea la forma que tome, los datos de configuración impulsan el comportamiento de tiempo de ejecución de una aplicación. Cuando los valores de configuración cambian, no es necesario reconstruir el código.

NO ESCRIBIR CÓDIGO DODO

Sin una configuración externa, 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 extinguieron.

[45] 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.

imágenes/dodo.png



LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 9, DRY—Los males de la duplicación
- Tema 14, Idiomas de dominio
- Tema 16, El poder del texto sin formato
- Tema 28, Desacoplamiento

no te excedas

En la primera edición de este libro, sugerimos usar configuración en lugar de código de manera similar, pero aparentemente deberíamos haber sido un poco más específicos en nuestras instrucciones. Cualquier consejo puede llevarse al extremo o usarse de manera inapropiada, así que aquí hay algunas precauciones:

No te excedas. Uno de nuestros primeros clientes decidió que todos los campos de su aplicación deberían ser configurables. Como resultado, llevó semanas realizar incluso el cambio más pequeño, ya que tuvo que implementar tanto el campo como todo el código de administración para guardarlo y editarlo. Tenían unas 40.000 variables de configuración y una pesadilla de codificación en sus manos.

No empuje las decisiones a la configuración por pereza. Si hay un debate genuino sobre si una función debería funcionar de esta manera o de otra, o si debería ser la elección de los usuarios, pruébela de una manera y obtenga comentarios sobre si la decisión fue buena.

notas al pie

[37] Así que no es realmente una ley. Es más como La Buena Idea de Deméter.

[38] https://media.pragprog.com/articles/jan_03_enbug.pdf

[39] Sí, sabemos que Ruby ya tiene esta capacidad con su función `at_exit`.

[40] <https://media.pragprog.com/titles/tpp20/code/event/rxcommon/logger.js>

[41] Parece que el primer uso de los caracteres `>` como tubería data de 1994, en una discusión sobre el lenguaje Isobelle/ML, archivada en <https://blogs.msdn.microsoft.com/dsyme/2011/05/17/semitótica-arqueológica-el-nacimiento-del-símbolo-del-oleoducto-1994/>

[42] Aquí nos hemos tomado una libertad. Técnicamente ejecutamos las siguientes funciones.
Simplemente no ejecutamos el código en ellos.

[43] De hecho, podría agregar dicho operador a Elixir usando su función de macro; un ejemplo de esto es la biblioteca Monad en hexadecimal. También puede usar Elixir's `con` construcción, pero luego pierde gran parte del sentido de escribir transformaciones que obtiene con canalizaciones.

[44] <https://www.quora.com/What-does-Alan-Kay-think-about-inheritance-in-object-oriented-programming>

[45] No ayudó que los colonos golpearan a los plácidos (léase: estúpidos) pájaros hasta matarlos con garrotes por deporte.

Capítulo 6

conurrencia

Solo para que todos estemos en la misma página, comenzemos con algunas definiciones:

La concurrencia es cuando la ejecución de dos o más piezas de código actúan como si se ejecutaran al mismo tiempo. El paralelismo es cuando se ejecutan al mismo tiempo.

Para tener simultaneidad, debe ejecutar el código en un entorno que pueda cambiar la ejecución entre diferentes partes de su código cuando se está ejecutando. Esto a menudo se implementa usando cosas como fibras, hilos y procesos.

Para tener paralelismo, necesita hardware que pueda hacer dos cosas a la vez. Esto puede ser varios núcleos en una CPU, varias CPU en una computadora o varias computadoras conectadas entre sí.

todo es concurrente

Es casi imposible escribir código en un sistema de tamaño decente que no tiene aspectos concurrentes. Pueden ser explícitos o pueden estar enterrados dentro de una biblioteca. La simultaneidad es un requisito si desea que su aplicación pueda manejar el mundo real, donde las cosas son asincrónicas: los usuarios interactúan, se obtienen datos, se llama a servicios externos, todo al mismo tiempo. Si obliga a que este proceso sea en serie, con una cosa que sucede, luego la siguiente, y así sucesivamente, su sistema se siente lento y probablemente no esté aprovechando al máximo la potencia del hardware en el que se ejecuta.

En este capítulo veremos la concurrencia y el paralelismo.

Los desarrolladores a menudo hablan sobre el acoplamiento entre fragmentos de código. Se refieren a las dependencias y cómo esas dependencias hacen que las cosas sean difíciles de cambiar. Pero hay otra forma de acoplamiento. El acoplamiento temporal ocurre cuando su código impone una secuencia en las cosas que no se requieren para resolver el problema en cuestión. ¿Dependes de que el “tick” venga antes del “tock” ?

No si quieres ser flexible. ¿Su código accede a múltiples servicios de back-end secuencialmente, uno tras otro? No si quieres mantener a tus clientes. En el Tema 33, Cómo romper el acoplamiento temporal, veremos formas de identificar este tipo de acoplamiento temporal.

¿Por qué es tan difícil escribir código concurrente y paralelo? Una de las razones es que aprendimos a programar usando sistemas secuenciales, y nuestros lenguajes tienen características que son relativamente seguras cuando se usan secuencialmente, pero que se vuelven una desventaja una vez que dos cosas pueden

suceder al mismo tiempo. Uno de los mayores culpables aquí es el estado compartido. Esto no solo significa variables globales: cada vez que dos o más fragmentos de código contienen referencias a la misma pieza de datos mutables, tiene un estado compartido. Y el Tema 34, El estado compartido es un estado incorrecto. La sección describe una serie de soluciones para esto, pero en última instancia, todas son propensas a errores.

Si eso te pone triste, ¡nil desperandum! Hay mejores formas de construir aplicaciones concurrentes. Uno de ellos es usar el modelo de actor, donde los procesos independientes, que no comparten datos, se comunican a través de canales usando una semántica simple y definida.

Hablamos tanto de la teoría como de la práctica de este enfoque en el Tema 35, Actores y Procesos.

Finalmente, veremos el Tema 36, Pizarras. Estos son sistemas que actúan como una combinación de un almacén de objetos y un intermediario inteligente de publicación/suscripción. En su forma original, en realidad nunca despegaron. Pero hoy estamos viendo más y más implementaciones de capas de middleware con semántica similar a una pizarra. Usados correctamente, estos tipos de sistemas ofrecen una gran cantidad de desacoplamiento.

El código concurrente y paralelo solía ser exótico. Ahora se requiere.



Tema 33

Rompiendo el Acoplamiento Temporal

“¿De qué se trata el acoplamiento temporal ?” , te preguntarás. 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. Así es como piensa la mayoría de la gente: haz esto y luego siempre haz aquello. Pero pensar de esta manera conduce al acoplamiento temporal: acoplamiento en el tiempo.

El método A siempre debe llamarse antes que el método B; solo se puede ejecutar un informe a la vez; debe esperar a que la pantalla se vuelva a dibujar antes de 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 tiempo u orden de dependencias. 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,

y despliegue. El resultado serán sistemas más fáciles de razonar, que potencialmente responderán más rápido y de manera más confiable.

BUSCANDO CONCURRENCIA

En muchos proyectos, necesitamos modelar y analizar los flujos de trabajo de la aplicación como parte del diseño. Nos gustaría saber qué puede suceder al mismo tiempo y qué debe suceder en un orden estricto. Una forma de hacerlo es capturar el flujo de trabajo utilizando una notación como el diagrama de actividad.

[46]

Consejo 56

Analice el flujo de trabajo para mejorar la concurrencia

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 se completan todas las acciones que conducen a una barra de sincronización, puede continuar con las flechas que salen de la barra. Una acción sin flechas que conduzcan a ella se puede iniciar en cualquier momento.

Puede usar diagramas de actividades para maximizar el paralelismo mediante la identificación de actividades que podrían realizarse en paralelo, pero no lo hacen.

Por ejemplo, podemos estar escribiendo el software para un fabricante robótico de piña colada. Nos dicen que los pasos son:

1. Licuadora abierta

1. Cierra la licuadora

2. Mezcla abierta de piña colada

2. Licuar por 1 minuto

- | | |
|------------------------------------|------------------------------|
| 3. Ponga la mezcla en la licuadora | 3. Licuadora abierta |
| 4. Medir 1/2 taza de ron blanco | 4. Consigue gafas |
| 5. Vierta el ron | 5. Consigue sombrillas rosas |
| 6. Agrega 2 tazas de hielo | 6. Servir |

Sin embargo, un cantinero perdería su trabajo si siguiera estos pasos, uno por uno, en orden. Aunque describen estas acciones en serie, muchas de ellas podrían realizarse en paralelo.

Usaremos el siguiente diagrama de actividad para capturar y razonar acerca de la simultaneidad potencial.

images/pina-colada.png

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.

Formateo más rápido

Este libro está escrito en texto plano. Para construir la versión que se va a imprimir, o un libro electrónico, o lo que sea, ese texto se alimenta a través de una tubería de procesadores. Algunos buscan construcciones particulares (citas de bibliografía, entradas de índice, marcas especiales para sugerencias, etc.). Otros procesadores operan sobre el documento como un todo.

Muchos de los procesadores en la tubería tienen que acceder a información externa (leer archivos, escribir archivos, canalizar a través de programas externos). Todo este trabajo de velocidad relativamente lenta nos da la oportunidad de explotar la concurrencia: de hecho, cada paso en la canalización se ejecuta simultáneamente, leyendo desde el paso anterior y escribiendo en el siguiente.

Además, algunas partes del proceso hacen un uso relativamente intensivo del procesador. Uno de ellos es la conversión de fórmulas matemáticas. Por varias razones históricas cada ecuación puede tardar hasta 500 ms en convertirse. Para acelerar las cosas, aprovechamos el paralelismo. Debido a que cada fórmula es independiente de las demás, convertimos cada una en su propio proceso paralelo y recopilamos los resultados nuevamente en el libro a medida que están disponibles.

Como resultado, el libro se construye mucho, mucho más rápido en máquinas multinúcleo.

(Y, sí, de hecho descubrimos una serie de errores de concurrencia en nuestra canalización en el camino...)

OPORTUNIDADES DE CONCURRENCIA

Los diagramas de actividad muestran las áreas potenciales de concurrencia, pero no tienen nada que decir sobre si vale la pena explotar estas áreas. Por ejemplo, en el ejemplo de la piña colada, un cantinero necesitaría cinco manos para poder ejecutar todas las posibles tareas iniciales a la vez.

Y ahí es donde entra la parte del diseño. Cuando miramos las actividades, nos damos cuenta de que el número 8, licuar, tomará un minuto. Durante ese tiempo, nuestro cantinero puede obtener los vasos y sombrillas (actividades 10 y 11) y probablemente todavía tenga tiempo para atender a otro cliente.

Y eso es lo que buscamos cuando diseñamos para la concurrencia. Esperamos encontrar actividades que requieran tiempo, pero no tiempo en nuestro código. Consultar una base de datos, acceder a un externo

servicio, esperando la entrada del usuario: todas estas cosas normalmente detendrían nuestro programa hasta que se completen. Y todas estas son oportunidades para hacer algo más productivo que el equivalente de CPU de jugar con los pulgares.

OPORTUNIDADES PARA EL PARALELISMO

Recuerde la distinción: la concurrencia es un mecanismo de software y el paralelismo es una preocupación de hardware. Si tenemos varios procesadores, ya sea de forma local o remota, entonces, si podemos dividir el trabajo entre ellos, podemos reducir el tiempo total que tardan las cosas.

Lo ideal para dividir de esta manera son piezas de trabajo que son relativamente independientes, donde cada uno puede proceder sin esperar nada de los demás. Un patrón común es tomar un gran trabajo, dividirlo en partes independientes, procesar cada una en paralelo y luego combinar los resultados.

Un ejemplo interesante de esto en la práctica es la forma en que funciona el compilador del lenguaje Elixir. Cuando comienza, divide el proyecto que está construyendo en módulos y compila cada uno en paralelo. A veces, un módulo depende de otro, en cuyo caso su compilación se detiene hasta que los resultados de la compilación del otro módulo estén disponibles. Cuando se completa el módulo de nivel superior, significa que se han compilado todas las dependencias. El resultado es una compilación rápida que aprovecha todos los núcleos disponibles.

IDENTIFICAR OPORTUNIDADES ES LA PARTE FÁCIL

Vuelva a sus aplicaciones. Hemos identificado lugares donde se beneficiará de la concurrencia y el paralelismo. Ahora viene la parte difícil: cómo podemos implementarlo de forma segura. ese es el tema del resto

del capítulo

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 10, Ortogonalidad
- Tema 26, Cómo equilibrar los recursos
- Tema 28, Desacoplamiento
- Tema 36, Pizarrones

RETOS

- ¿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?



Tema 34

El estado compartido es un estado incorrecto

Estás en tu restaurante favorito. Terminas tu plato principal y le preguntas a tu servidor si queda algo de tarta de manzana. Mira por encima del hombro, ve una pieza en la vitrina y dice que sí. Lo ordenas y suspiras satisfecho.

Mientras tanto, al otro lado del restaurante, otro cliente le hace la misma pregunta a su mesero. Ella también mira, confirma que hay una pieza y que el cliente ordena.

Uno de los clientes va a estar decepcionado.

Cambie la vitrina por una cuenta bancaria conjunta y convierta a los camareros en dispositivos de punto de venta. Usted y su pareja deciden comprar un nuevo teléfono al mismo tiempo, pero solo hay suficiente en la cuenta para uno. Alguien, el banco, la tienda o usted, se va a sentir muy infeliz.

Consejo 57

El estado compartido es un estado incorrecto

El problema es el estado compartido. Cada mesero en el restaurante miró la vitrina sin tener en cuenta al otro. Cada dispositivo de punto de venta miraba el saldo de una cuenta sin importar el otro.

ACTUALIZACIONES NO ATÓMICAS

Veamos nuestro ejemplo de restaurante como si fuera código:



images/pie_case.png

Los dos camareros operan simultáneamente (y, en la vida real, en paralelo). Veamos su código:

```
if display_case.pie_count > 0
    promete_pastele_al_cliente()
    display_case.tomar_pastele()
    dar_pastele_al_cliente()
end
```

El mesero 1 obtiene el conteo de pasteles actual y descubre que es uno. Le promete el pastel al cliente. Pero en ese momento, el camarero 2 se ejecuta. También ve que el conteo de pasteles es uno y le hace la misma promesa a su cliente. Uno de los dos agarra el último trozo de pastel y el otro mesero entra en algún tipo de estado de error (lo que probablemente implique mucha humillación).

El problema aquí no es que dos procesos puedan escribir en la misma memoria. El problema es que ningún proceso puede garantizar que su visión de esa memoria sea consistente. Efectivamente, cuando un camarero ejecuta `display_case.pie_count()`, copia el valor de la vitrina en su propia memoria. Si el valor en la vitrina cambia, su memoria (que están usando para tomar decisiones) ahora está desactualizada.

Todo esto se debe a que obtener y luego actualizar el recuento circular no es una operación atómica: el valor subyacente puede cambiar en el medio.

Entonces, ¿cómo podemos hacerlo atómico?

Semáforos y otras formas de exclusión mutua

Un semáforo es simplemente una cosa que solo una persona puede poseer a la vez. Puede crear un semáforo y luego usarlo para controlar el acceso a algún otro recurso. En nuestro ejemplo, podríamos crear un semáforo para controlar el acceso al caso circular y adoptar la convención de que cualquier persona que desee actualizar el contenido del caso circular solo puede hacerlo si tiene ese semáforo.

Digamos que el comensal decide solucionar el problema del pastel con un semáforo físico. Colocan un duende de plástico en la caja de pastel. Antes de que cualquier mesero pueda vender un pastel, debe tener al Leprechaun en la mano. Una vez que se ha completado su pedido (lo que significa entregar el pastel a la mesa), pueden devolver al Leprechaun a su lugar custodiando el tesoro de los pasteles, listo para mediar en el próximo pedido.

Veamos esto en código. Clásicamente, la operación para tomar el semáforo se llamaba P y la operación para liberarlo se llamaba V. Hoy usamos términos como bloquear/desbloquear, reclamar/liberar, etc.^[47]

```
case_semaphore.lock()

if display_case.pie_count > 0
    promete_pastele_al_cliente()
    display_case.tomar_pastele()
    dar_pastele_al_cliente()
end
```

```
case_semaphore.desbloquear()
```

Este código asume que ya se ha creado y almacenado un semáforo en la variable `case_semaphore`.

Supongamos que ambos camareros ejecutan el código al mismo tiempo. Ambos intentan bloquear el semáforo, pero solo uno lo consigue. El que recibe el semáforo sigue funcionando con normalidad. El que no recibe el semáforo se suspende hasta que el semáforo esté disponible (el camarero espera...). Cuando el primer camarero completa el pedido, desbloquea el semáforo y el segundo camarero continúa corriendo. Ahora ven que no hay pastel en el caso y se disculpán con el cliente.

Hay algunos problemas con este enfoque. Probablemente lo más significativo es que solo funciona porque todos los que acceden al caso circular están de acuerdo con la convención de usar el semáforo. Si alguien se olvida (es decir, algún desarrollador escribe código que no sigue la convención), entonces volvemos al caos.

Hacer que el recurso sea transaccional

El diseño actual es deficiente porque delega la responsabilidad de proteger el acceso a la caja circular a las personas que la usan. Cambiémoslo para centralizar ese control. Para hacer esto, tenemos que cambiar la API para que los camareros puedan verificar el conteo y también tomar una porción del pastel en una sola llamada:

```
rebanada = mostrar_caso.obtener_pastel_si_disponible()
if
    rebanada dar_pastele_al_cliente()
end
```

Para que esto funcione, necesitamos escribir un método que se ejecute como parte de la propia vitrina:

```

def obtener_pie_si_disponible() ##### if
    @slices.size > 0           #
        update_sales_data(:pie) # devolver
        @slices.shift else #      #
            ¡código incorrecto! falso # fin fin

        #
#####

```

Este código ilustra un concepto erróneo común. Hemos trasladado el acceso a los recursos a un lugar central, pero aún se puede llamar a nuestro método desde múltiples subprocessos simultáneos, por lo que aún debemos protegerlo con un semáforo:

```

def get_pie_if_disponible()
    @case_semaphore.lock()

    if @slices.size > 0
        update_sales_data(:pie)
        devuelve @slices.shift
    else
        fin
        falso

    @case_semaphore.desbloquear()
fin

```

Incluso este código podría no ser correcto. Si `update_sales_data` genera una excepción, el semáforo nunca se desbloqueará y todos los accesos futuros al caso circular se bloquearán indefinidamente. Necesitamos manejar esto:

```

def get_pie_if_disponible()
    @case_semaphore.lock()

pruebe { si @slices.size
    > 0 update_sales_data(:pie)
    devuelva @slices.shift
else
    false

```

```

end }

asegurar
{ @case_semaphore.unlock() } end

```

Debido a que este es un error tan común, muchos lenguajes proporcionan bibliotecas que manejan esto por usted:

```

def get_pie_if_disponible()
  @case_semaphore.protect() { if
    @slices.size > 0
      update_sales_data(:pie) return
    @slices.shift else false
  end }
end

```

TRANSACCIONES DE MÚLTIPLES RECURSOS

Nuestro restaurante acaba de instalar un congelador de helados. Si un cliente pide pastel a la moda, el mesero deberá verificar que tanto el pastel como el helado estén disponibles.

Podríamos cambiar el código del camarero a algo como:

```

rebanada = vitrina.obtener_pastel_si_disponible()
cucharada = congelador.obtener_helado_crema_si_disponible()

if slice && scoop
  give_order_to_customer() end

```

Sin embargo, esto no funcionará. ¿Qué sucede si reclamamos una porción de pastel, pero cuando tratamos de obtener una bola de helado, descubrimos que no hay? Ahora nos queda un pastel con el que no podemos hacer nada (porque nuestro cliente debe tener helado). Y

el hecho de que tengamos el pastel significa que no está en el estuche, por lo que no está disponible para otro cliente que (siendo un purista) no quiere un helado con él.

Podríamos arreglar esto agregando un método al caso que nos permita devolver una porción del pastel. Tendremos que agregar el manejo de excepciones para asegurarnos de no conservar los recursos si algo falla:

```
rebanada = display_case.get_pie_if_disponible()

si la

rebanada intenta { cucharada =

}

congelador.obtener_helado_crema_si_disponible() si la cucharada intenta { dar_pedido_al_cliente() } rescatar { congelador.devolver_devolu
}

rescate

{ display_case.give_back(slice) } end
```

Una vez más, esto es menos que ideal. El código ahora es realmente feo: averiguar lo que realmente hace es difícil: la lógica comercial está enterrada en toda la limpieza.

Anteriormente solucionamos esto moviendo el código de manejo de recursos al recurso mismo. Aquí, sin embargo, tenemos dos recursos.
¿Deberíamos poner el código en la vitrina o en el congelador?

Creemos que la respuesta es “no” a ambas opciones. El enfoque pragmático sería decir que el “pastel de manzana a la moda” es su propio recurso. Moveríamos este código a un nuevo módulo, y luego el

el cliente podría simplemente decir "tráeme pastel de manzana con helado" y tendrá éxito o fallará.

Por supuesto, en el mundo real es probable que haya muchos platos compuestos como este, y no querrás escribir nuevos módulos para cada uno. En su lugar, probablemente desee algún tipo de elemento de menú que contenga referencias a sus componentes, y luego tener un método genérico `get_menu_item` que haga que el recurso baile con cada uno.

ACTUALIZACIONES NO TRANSACCIONALES

Se presta mucha atención a la memoria compartida como fuente de problemas de concurrencia, pero de hecho los problemas pueden surgir en cualquier lugar donde el código de su aplicación comparta recursos mutables: archivos, bases de datos, servicios externos, etc. Cada vez que dos o más instancias de su código pueden acceder a algún recurso al mismo tiempo, está viendo un problema potencial.

A veces, el recurso no es tan obvio. Mientras escribíamos esta edición del libro, actualizamos la cadena de herramientas para hacer más trabajo en paralelo usando subprocessos. Esto hizo que la compilación fallara, pero de formas extrañas y en lugares aleatorios. Un hilo común a través de todos los errores era que no se podían encontrar archivos o directorios, aunque en realidad estaban exactamente en el lugar correcto.

Rastreamos esto hasta un par de lugares en el código que cambiaron temporalmente el directorio actual. En la versión no paralela, el hecho de que este código restaurara el directorio fue suficiente. Pero en la versión paralela, un subprocesso cambiaría el directorio y luego, mientras estaba en ese directorio, otro subprocesso comenzaría a ejecutarse. Ese subprocesso esperaría estar en el directorio original, pero debido a que el directorio actual es

compartido entre hilos, ese no era el caso.

La naturaleza de este problema genera otro consejo:

Sugieren

Las fallas aleatorias suelen ser problemas de concurrencia

OTROS TIPOS DE ACCESO EXCLUSIVO

La mayoría de los idiomas tienen soporte de biblioteca para algún tipo de acceso exclusivo a recursos compartidos. Pueden llamarlo mutexes (para exclusión mutua), monitores o semáforos. Todos estos se implementan como bibliotecas.

Sin embargo, algunos idiomas tienen soporte de simultaneidad integrado en el propio idioma. Rust, por ejemplo, hace cumplir el concepto de propiedad de los datos; solo una variable o parámetro puede contener una referencia a cualquier dato mutable en particular a la vez.

También podría argumentar que los lenguajes funcionales, con su tendencia a hacer que todos los datos sean inmutables, simplifican la concurrencia. Sin embargo, aún enfrentan los mismos desafíos, porque en algún momento se ven obligados a ingresar al mundo real y mutable.

DOCTORA, DUELE...

Si no quita nada más de esta sección, tome esto: la simultaneidad en un entorno de recursos compartidos es difícil, y administrarlo usted mismo está lleno de desafíos.

Es por eso que recomendamos el remate del viejo chiste:

Doctor, me duele cuando hago esto.

Entonces no hagas eso.

Las próximas dos secciones sugieren formas alternativas de obtener los beneficios de la simultaneidad sin el dolor.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 10, Ortogonalidad
- Tema 28, Desacoplamiento
- Tema 38, Programación por Coincidencia



Tema 35

Actores y Procesos

Sin escritores, las historias no serían escrito,

Sin actores, las historias no podrían cobrar vida.

Angie-Marie Delsante

Los actores y procesos ofrecen formas interesantes de implementar la concurrencia sin la carga de sincronizar el acceso a los recursos compartidos. memoria.

Sin embargo, antes de adentrarnos en ellos, debemos definir lo que queremos decir.

Y esto va a sonar académico.

No temas, estaremos trabajando en todo en poco tiempo.

- Un actor es un procesador virtual independiente con su propio estado local (y privado). Cada actor tiene un buzón. Cuando aparece un mensaje en el buzón y el actor está inactivo, cobra vida y procesa el mensaje. Cuando termina de procesar, procesa otro mensaje en el buzón o, si el buzón está vacío, vuelve a dormir.

Al procesar un mensaje, un actor puede crear otros actores, enviar mensajes a otros actores que conoce y crear un nuevo estado que se convertirá en el estado actual cuando se procese el siguiente mensaje.

- Un proceso suele ser un procesador virtual de propósito más general, a menudo implementado por el sistema operativo para facilitar la concurrencia. Los procesos pueden estar restringidos (por convención) para que se comporten como actores, y ese es el tipo de proceso al que nos referimos aquí.

LOS ACTORES SOLO PUEDEN SER CONCURRENTES

Hay algunas cosas que no encontrarás en la definición de actores:

- No hay una sola cosa que esté en control. Nada programa lo que sucede a continuación ni organiza la transferencia de información desde los datos sin procesar hasta el resultado final.
- El único estado en el sistema se mantiene en los mensajes y en el estado local de cada actor. Los mensajes no pueden ser examinados a menos que sean leídos por su destinatario, y el estado local es inaccesible fuera del actor.
- Todos los mensajes son unidireccionales: no existe el concepto de respuesta. Si desea que un actor le devuelva una respuesta, incluya su propia dirección de buzón en el mensaje que le envíe y (eventualmente) enviará la respuesta como un mensaje más a ese buzón.
- Un actor procesa cada mensaje hasta su finalización y solo procesa un mensaje a la vez.

Como resultado, los actores ejecutan simultáneamente, de forma asíncrona y no comparten nada. Si tuviera suficientes procesadores físicos, podría ejecutar un actor en cada uno. Si tiene un solo procesador, algún tiempo de ejecución puede manejar el cambio de contexto entre ellos. De cualquier manera, el código que se ejecuta en los actores es el mismo.

Consejo 59

Usar actores para concurrencia sin estado compartido

UN SIMPLE ACTOR

Implementemos nuestro comedor usando actores. En este caso, tendremos tres (el cliente, el mesero y el pastelero).

El flujo de mensajes general se verá así:

- Nosotros (como una especie de ser externo, parecido a Dios) le decimos al cliente que

tienen hambre

- En respuesta, le pedirán pastel al mesero.
- El mesero le pedirá a la caja de pasteles que lleve un poco de pastel al cliente.
- Si la caja de tarta tiene una porción disponible, se la enviará al cliente y también le notificará al mesero para que la agregue a la cuenta.
- Si no hay pastel, el caso le dice al mesero, y el mesero se disculpa con el cliente.

Hemos optado por implementar el código en JavaScript usando el biblioteca nact. ^[48] Hemos agregado un pequeño contenedor a esto que nos permite escribir actores como objetos simples, donde las claves son los tipos de mensajes que recibe y los valores son funciones para ejecutar cuando se recibe ese mensaje en particular. (La mayoría de los sistemas de actores tienen un tipo de estructura similar, pero los detalles dependen del idioma anfitrión).

Comencemos con el cliente. El cliente puede recibir tres mensajes:

- Tienes hambre (enviado por el contexto externo)
- Hay pastel en la mesa (enviado por la caja de pasteles)
- Lo siento, no hay pastel (enviado por el mesero)

Aquí está el código:

concurrencia/actores/index.js

```
const clienteActor =
  { 'hambriento de pastel': (msj, ctx, estado)
    => { return
      dispatch(estado.camarero, { tipo: "pedido", cliente: ctx.self, quiere: 'pastel' })
    },
  }
```

```
'poner en la mesa': (msg, ctx, _state)
=> console.log(`${ctx.self.name} ve aparecer "${msg.food}" en la mesa`),

'no queda pastel': (_msg, ctx, _state)
=> console.log(`${ctx.self.name} se enfada...`)
}
```

El caso interesante es cuando recibimos un mensaje de "hambre de pastel", donde luego enviamos un mensaje al mesero. (Veremos cómo el cliente sabe sobre el actor camarero en breve).

Aquí está el código del camarero:

concurrencia/actores/index.js

```
const waiterActor =
{ "order": (msg, ctx, state) => { if
(msg.wants == "pie")
{ dispatch(state.pieCase,
{ type: "get slice", customer: msg.customer, camarero: ctx.self })

} else
{ console.dir('No sé cómo pedir ${msg.wants}'); } },

"añadir al pedido": (msg, ctx) =>
console.log(`El camarero añade ${msg.food} al pedido de ${msg.customer.name}` ),

"error": (msg, ctx) =>
{ dispatch(msg.customer, { type: 'no pie left', msg: msg.msg });
console.log(`\nEl mesero se disculpa con ${msg.customer.name}: ${
msg.msg}` ) }

};
```

Cuando recibe el mensaje **de 'pedido'** del cliente, verifica si la solicitud es para un pastel. Si es así, envía una solicitud al pastel.

caso, pasando referencias tanto a sí mismo como al cliente.

El caso del pastel tiene un estado: una matriz de todas las porciones del pastel que contiene. (Nuevamente, vemos cómo se configura eso en breve). Cuando recibe un mensaje [de "obtener rebanada"](#) del mesero, ve si le quedan rebanadas.

Si lo hace, pasa la porción al cliente, le dice al mesero que actualice el pedido y finalmente devuelve un estado actualizado, que contiene una porción menos. Aquí está el código:

concurrencia/actores/index.js

```
const pieCaseActor = { 'get
  slice': (msg, context, state) => { if
    (state.slices.length == 0)
    { dispatch(msg.waiter,
      { type: 'error', msg: "no queda pastel ", cliente: mensaje.cliente })
    estado de retorno

  } else
  { var rebanada = estado.rebanadas.shift("rebanada
  circular";
    despacho(mensaje.cliente, { tipo: 'poner en
  la mesa', comida:
      rebanada }); despacho(msg.waiter, { type: 'add to order', food: slice,
    customer:

  msg.customer}); estado de retorno ; } } }
```

Aunque a menudo encontrará que los actores son iniciados dinámicamente por otros actores, en nuestro caso lo simplificaremos e iniciaremos nuestros actores manualmente. También pasaremos a cada uno algún estado inicial:

- El caso circular obtiene la lista inicial de sectores circulares que contiene
- Le daremos al mesero una referencia a la caja de pasteles.
- Le daremos a los clientes una referencia al mesero.

concurrencia/actores/index.js

```

const actorSystem = inicio();

let pieCase =
    start_actor( actorSystem, 'pie-case', pieCaseActor, { rebanadas: ["manzana", "melocotón", "cereza"] });

let camarero =
    start_actor( actorSystem, 'waiter', waiterActor, { pieCase: pieCase });

let c1 = actor_inicial(sistemaactor, 'cliente1',
    actorCliente, { camarero: camarero }); let
c2 = start_actor(actorSistema, 'cliente2', clienteActor,
    { camarero: camarero });

```

Y finalmente damos el pistoletazo de salida. Nuestros clientes son codiciosos. El cliente 1 pide tres rebanadas de pastel y el cliente 2 pide dos:

concurrencia/actores/index.js

```

despacho(c1, { tipo: 'hambriento de pastel', mesero: mesero });
despacho(c2, { tipo: 'hambriento de pastel', mesero: mesero });
despacho(c1, { tipo: 'hambriento de pastel', mesero: mesero });
despacho(c2, { tipo: 'hambriento de pastel', mesero: mesero });
despacho(c1, { tipo: 'hambriento de pastel', mesero: mesero });

dormir(500) .then(() =>
    { detener(actorSistema); })

```

Cuando lo ejecutamos, podemos ver a los actores comunicándose.^[49] El orden que ves bien puede ser diferente:

```

$ nodo index.js
cliente1 ve "rebanada de pastel de manzana" en la mesa
cliente2 ve "rebanada de pastel de durazno" en la mesa

```

El mesero agrega una rebanada de pastel de manzana al pedido del cliente 1

El camarero agrega una rebanada de pastel de durazno al pedido del cliente

2. El cliente 1 ve que aparece "rebanada de pastel de cereza" en la mesa.

El mesero agrega una rebanada de pastel de cerezas al pedido del cliente 1

El mesero se disculpa con el cliente1: no quedó pastel el cliente1 se enfurruña...

El mesero se disculpa con el cliente2: no queda pastel el cliente2 se enfurruña...

SIN CONCURRENCIA EXPLÍCITA

En el modelo de actor, no es necesario escribir ningún código para manejar la simultaneidad, ya que no hay un estado compartido. Tampoco hay necesidad de codificar en lógica explícita de extremo a extremo "haz esto, haz aquello", ya que los actores lo resuelven por sí mismos en función de los mensajes que reciben.

Tampoco se menciona la arquitectura subyacente. Este conjunto de componentes funciona igual de bien en un solo procesador, en múltiples núcleos o en múltiples máquinas en red.

ERLANG PREPARA EL ESCENARIO

El lenguaje y el tiempo de ejecución de Erlang son excelentes ejemplos de la implementación de un actor (aunque los inventores de Erlang no habían leído el artículo original de Actor). Erlang llama procesos a los actores, pero no son procesos regulares del sistema operativo. En cambio, al igual que los actores que hemos estado discutiendo, los procesos de Erlang son livianos (puede ejecutar millones de ellos en una sola máquina) y se comunican mediante el envío de mensajes. Cada uno está aislado de los demás, por lo que no se comparte el estado.

Además, el tiempo de ejecución de Erlang implementa un sistema de supervisión que administra la vida útil de los procesos, lo que podría reiniciar un proceso o un conjunto de procesos en caso de falla. Y Erlang también ofrece carga de código activo: puede reemplazar el código en un

ejecutar el sistema sin detener ese sistema. Y el sistema Erlang ejecuta algunos de los códigos más confiables del mundo, a menudo citando la disponibilidad de nueve nueves.

Pero Erlang (y su progenie Elixir) no son únicos: hay implementaciones de actores para la mayoría de los idiomas. Considere usarlos para sus implementaciones simultáneas.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 28, Desacoplamiento
- Tema 30, Transformación de la programación
- Tema 36, Pizarrones

RETOS

- ¿Tiene actualmente un código que utiliza la exclusión mutua para proteger los datos compartidos? ¿Por qué no probar un prototipo del mismo código escrito con actores?
- El código de actor para el comensal solo admite pedidos de rebanadas de pastel. Extiéndalo para permitir que los clientes pidan pastel a la moda, con agentes independientes que manejen las rebanadas de pastel y las bolas de helado. Arregle las cosas para que maneje la situación en la que se agote uno u otro.



La escritura está en la pared...

Daniel 5 (referencia)

Considere cómo los detectives podrían usar una pizarra para coordinar y resolver una investigación de asesinato. El inspector jefe comienza instalando una gran pizarra en la sala de conferencias. En él, ella escribe una sola pregunta:

H. Dumpty (Hombre, Huevo): ¿Accidente? ¿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 se muestra una pizarra de muestra .

imágenes/pizarra.png



Figura 2. Alguien encontró una conexión entre las deudas de juego de Humpty y los registros telefónicos. Tal vez estaba recibiendo llamadas telefónicas amenazantes.

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.

- 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.

Esta es una forma de concurrencia de laissez faire . Los detectives son procesos independientes, agentes, actores, etc. Algunos almacenan datos en la pizarra. Otros toman datos del pizarrón, tal vez combinándolos o procesándolos, y agregan más información al pizarrón. Gradualmente, la pizarra les ayuda a llegar a una conclusión.

Los sistemas de pizarra basados en computadora se usaron originalmente 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.

Uno de los primeros sistemas de pizarra fue Linda de David Gelernter. Almacenaba hechos como tuplas mecanografiadas. Las aplicaciones podrían escribir nuevas tuplas en Linda y consultar las tuplas existentes utilizando una forma de coincidencia de patrones.

Más tarde llegaron sistemas distribuidos similares a pizarras como JavaSpaces y T Spaces. 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**. Puede buscar en una pizarra que contenga objetos **Persona** utilizando una plantilla **Autor** con un valor **lastName** de "Shakespeare". Obtendría a Bill Shakespeare como autor, pero no a Fred Shakespeare como jardinero.

Creemos que estos sistemas nunca despegaron realmente, en parte, porque aún no se había desarrollado la necesidad del tipo de procesamiento cooperativo concurrente.

UNA PIZARRA EN ACCIÓN

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 no debe hacer otras preguntas, y así sucesivamente.

Más allá del miasma de la ley aplicable, también tenemos los siguientes problemas que enfrentar:

- Las respuestas pueden llegar en cualquier orden. 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. Muchos de estos sistemas

existen, pero pueden ser complejos y requieren 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.

Consejo 60

Use pizarras para coordinar el flujo de trabajo

LOS SISTEMAS DE MENSAJERÍA PUEDEN SER COMO PIZARRAS

Mientras escribimos esta segunda edición, muchas aplicaciones se construyen usando pequeños servicios desacoplados, todos comunicándose a través de alguna forma de sistema de mensajería. Estos sistemas de mensajería (como Kafka y NATS) hacen mucho más que simplemente enviar datos de A a B. En particular, ofrecen persistencia (en forma de registro de eventos) y la capacidad de recuperar mensajes a través de una forma de coincidencia de patrones. Esto significa que puede usarlos como un sistema de pizarra y/o como una plataforma en la que puede ejecutar un grupo de actores.

PERO NO ES TAN SENCILLO...

El enfoque de actor y/o pizarra y/o microservicio de la arquitectura elimina toda una clase de posibles problemas de concurrencia de sus aplicaciones. Pero ese beneficio tiene un costo. Estos enfoques son más difíciles de razonar, porque muchos

de la acción es indirecta. Le resultará útil mantener un repositorio central de formatos de mensajes y/o API, especialmente si el repositorio puede generar el código y la documentación por usted.

También necesitará buenas herramientas para poder rastrear mensajes y hechos a medida que avanzan en el sistema. (Una técnica útil es agregar una identificación de seguimiento única cuando se inicia una función comercial en particular y luego propagarla a todos los actores involucrados. Luego podrá reconstruir lo que sucede a partir de los archivos de registro).

Finalmente, este tipo de sistema puede ser más problemático de implementar y administrar, ya que hay más partes móviles. Hasta cierto punto, esto se ve compensado por el hecho de que el sistema es más granular y puede actualizarse reemplazando actores individuales, y no todo el sistema.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 28, Desacoplamiento
- Tema 29, Malabares con el mundo real
- Tema 33, Rompiendo el Acoplamiento Temporal
- Tema 35, Actores y Procesos

EJERCICIOS

Ejercicio 24 (respuesta posible)

¿Sería apropiado un sistema estilo pizarra para las siguientes aplicaciones?

¿Por qué o por qué no?

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.

Calendario de grupos. Tiene personas repartidas por todo el mundo, en diferentes zonas horarias y hablando diferentes idiomas, tratando de programar una reunión.

Herramienta de monitoreo de red. El sistema recopila estadísticas de rendimiento y recopila informes de problemas, que los agentes utilizan para buscar problemas en el sistema.

RETOS

- ¿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?

notas al pie

[46] Aunque UML se ha desvanecido gradualmente, muchos de sus diagramas individuales aún existen de una forma u otra, incluido el muy útil diagrama de actividades. Para obtener más información sobre todos los tipos de diagramas UML, consulte [UML destilado: una breve guía para el lenguaje de modelado de objetos estándar \[Fow04\]](#).

[47] Los nombres P y V provienen de las letras iniciales de palabras holandesas. Sin embargo, hay algunos debate sobre exactamente qué palabras. El inventor de la técnica, Edsger Dijkstra, sugirió *passering* y *prolaag* para P, y *vrijgave* y posiblemente *verhogen* para V.

[48] <https://github.com/ncthbrt/nact>

[49] Para ejecutar este código, también necesitará nuestras funciones de contenedor, que no se muestran aquí. Puedes descargarlos desde <https://media.pragprog.com/titles/tpp20/code/concurrency/actors/index.js>

Capítulo 7

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 los proyectos de software fallan y muchos sistemas terminan siendo feos, ineficientes, mal estructurados, inmantenibles o simplemente incorrectos.

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.

No todas las decisiones son ni siquiera conscientes. Puede aprovechar mejor sus instintos y pensamientos no conscientes cuando utiliza el Tema 37, Escuche su cerebro de lagarto. Veremos cómo escuchar con más atención y buscar formas de responder activamente a estos pensamientos que a veces son molestos.

Pero escuchar sus instintos no significa que pueda volar con el piloto automático. Los desarrolladores que no piensan activamente en su código están programando por coincidencia: el código puede funcionar, pero

no hay ninguna razón en particular por qué. En el Tema 38, 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 el Tema 39, 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 el Tema 40, Refactorización, analizamos técnicas que nos ayudan a corregir el código existente de forma continua a medida que avanzamos.

La prueba no se trata de encontrar errores, se trata de obtener comentarios sobre su código: aspectos del diseño, la API, el acoplamiento, etc. Eso significa que los principales beneficios de las pruebas ocurren cuando piensas y escribes las pruebas, no solo cuando las ejecutas. Exploraremos esta idea en el Tema 41, Test to Code.

Pero, por supuesto, cuando prueba su propio código, puede aportar sus propios sesgos a la tarea. En el Tema 42, Pruebas basadas en propiedades, veremos cómo hacer que la computadora realice pruebas de gran alcance por usted y cómo manejar los errores inevitables que surgen.

Es fundamental que escriba un código que sea legible y fácil de razonar. Es un mundo duro, lleno de malos actores que están tratando activamente de entrar en su sistema y causar daño. Discutiremos algunas técnicas y enfoques muy básicos para ayudarlo Tema 43, Manténgase seguro afuera.

Finalmente, una de las cosas más difíciles en el desarrollo de software es el Tema 44, Nombrar cosas. Tenemos que nombrar muchas cosas y, en muchos sentidos, los nombres que elegimos definen la realidad que creamos. Debe estar al tanto de cualquier desviación semántica potencial mientras codifica.

La mayoría de nosotros podemos conducir un automóvil en gran medida con el piloto automático; no le ordenamos explícitamente a nuestro pie que pise un pedal, ni a nuestro brazo que gire 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.



Tema 37

Escucha tu cerebro de lagarto

Solo los seres humanos pueden mirar directamente algo, tener toda la información que necesitan para hacer una lectura precisa, predicción, tal vez incluso momentáneamente hacer la preciosa predicción, y luego decir que no es así.

Gavin de Becker, El regalo de Miedo

emerge de la puerta oscura...

Los instintos son simplemente una respuesta a los patrones empaquetados en nuestro cerebro no consciente. Algunos son innatos, otros se aprenden a través de la repetición. A medida que adquiere experiencia como programador, su cerebro va estableciendo capas de conocimiento tácito: cosas que funcionan, cosas que no funcionan, las causas probables de un tipo de error, todas las cosas que nota a lo largo de sus días. Esta es la parte de tu cerebro que presiona la tecla Guardar archivo cuando te detienes a conversar con alguien, incluso cuando no te das cuenta de que est

El trabajo de toda la vida de Gavin de Becker es ayudar a las personas a protegerse. Su libro, The Gift of Fear: And Other Survival Signals That Protect Us from Violence [de 98], resume su mensaje.

Uno de los temas clave que recorren el libro es que, como humanos sofisticados, hemos aprendido a ignorar nuestro lado más animal; nuestros instintos, nuestro cerebro de lagarto. Afirma que la mayoría de las personas que son agredidas en la calle son conscientes de sentirse incómodas o nerviosas antes del ataque. Estas personas simplemente se dicen a sí mismas que están siendo tontas. Entonces la figura

haciéndolo.

Cualquiera que sea su fuente, los instintos comparten una cosa: no tienen palabras. Los instintos te hacen sentir, no pensar. Y así, cuando se activa un instinto, no ves una bombilla parpadeante con una pancarta envuelta alrededor. En cambio, te pones nervioso, te mareas o sientes que esto es demasiado trabajo.

El truco es primero notar lo que está sucediendo y luego averiguar por qué. Veamos primero un par de situaciones comunes en las que tu lagarto interior intenta decirte algo. Luego discutiremos cómo puedes dejar que ese cerebro instintivo salga de su envoltorio protector.

MIEDO A LA PAGINA EN BLANCO

Todo el mundo teme a la pantalla vacía, al solitario cursor parpadeante rodeado de un montón de nada. 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 empezar.

Pensamos que hay dos problemas que provocan esto, y que ambos tienen la misma solución.

Un problema es que tu cerebro de lagarto está tratando de decirte algo; hay algún tipo de duda al acecho justo debajo de la superficie de la percepción. Y eso es importante.

Como desarrollador, ha 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 cuando se enfrenta a una tarea, puede ser que esa experiencia intente hablarle. Hazle caso. Es posible que no puedas

para señalar exactamente lo que está mal, pero dale tiempo y tus dudas probablemente se cristalicen en algo más sólido, algo que puedas abordar. Deje que sus instintos contribuyan a su rendimiento.

El otro problema es un poco más prosaico: es posible que simplemente tengas miedo de cometer un error.

Y ese es un temor razonable. Los desarrolladores ponemos mucho de nosotros mismos en nuestro código; podemos tomar errores en ese código como reflejos de nuestra competencia. Quizás también haya un elemento del síndrome del impostor ; podemos pensar que este proyecto está más allá de nosotros. No podemos ver nuestro camino hasta el final; llegaremos tan lejos y luego nos veremos obligados a admitir que estamos perdidos.

LUCHANDO USTED MISMO

A veces, el código simplemente vuela de su cerebro al editor: las ideas se convierten en bits aparentemente sin esfuerzo.

Otros días, la codificación se siente como caminar cuesta arriba en el barro. Dar cada paso requiere un esfuerzo tremendo, y cada tres pasos retrocedes dos.

Pero, siendo un profesional, sigues adelante, dando paso tras paso embarrado: tienes un trabajo que hacer. Desafortunadamente, eso es probablemente exactamente lo contrario de lo que deberías hacer.

Tu código está tratando de decirte algo. Está diciendo que esto es más difícil de lo que debería ser. Tal vez la estructura o el diseño sean incorrectos, tal vez estés resolviendo el problema equivocado, o tal vez solo estés creando una granja de hormigas llena de errores. Cualquiera que sea la razón, tu cerebro de lagarto está detectando la retroalimentación del código y está tratando desesperadamente de que escuches.

CÓMO HABLAR LAGARTO

Hablamos mucho de escuchar tus instintos, tu cerebro inconsciente de lagarto. Las técnicas son siempre las mismas.

Consejo 61

Escucha a tu lagarto interior

Primero, deja de hacer lo que estás haciendo. Date un poco de tiempo y espacio para que tu cerebro se organice solo. Deja de pensar en el código y haz algo que sea bastante tonto por un tiempo, lejos de un teclado. Dar un paseo, almorzar, charlar con alguien.

Tal vez dormir en ello. Deje que las ideas se filtren a través de las capas de su cerebro por sí solas: no puede forzarlas. Eventualmente, pueden burbujejar hasta el nivel consciente, y tienes uno de esos, ¡ja! momentos

Si eso no funciona, intente externalizar el problema. Haz garabatos sobre el código que estás escribiendo o explícaselo a un compañero de trabajo (preferiblemente a uno que no sea programador) o a tu patito de goma.

Exponga diferentes partes de su cerebro al problema y vea si alguna de ellas maneja mejor lo que le preocupa. Hemos perdido la cuenta de la cantidad de conversaciones que hemos tenido en las que uno de nosotros estaba explicando un problema al otro y de repente dijo: "¡Oh! ¡Por supuesto!" y se separó para arreglarlo.

Pero tal vez has probado estas cosas y todavía estás atascado. Es hora de actuar. Necesitamos decirle a tu cerebro que lo que estás a punto de hacer no importa. Y lo hacemos mediante la creación de prototipos.

¡ES HORA DE JUGAR!

Andy y Dave han pasado horas mirando los búferes vacíos del editor. Escribiremos un código, luego miraremos el techo, luego tomaremos otro trago, luego escribiremos un poco más de código, luego iremos

lea una historia divertida sobre un gato con dos colas, luego escriba un poco más de código, luego seleccione todo/borrar y comience de nuevo. Y otra vez. Y otra vez.

Y a lo largo de los años hemos encontrado un truco mental que parece funcionar. Dígase a sí mismo que necesita hacer un prototipo de algo. Si se encuentra frente a una pantalla en blanco, busque algún aspecto del proyecto que desee explorar. Tal vez esté usando un nuevo marco y quiera ver cómo hace el enlace de datos. O tal vez es un nuevo algoritmo y desea explorar cómo funciona en casos extremos.

O tal vez quiera probar un par de estilos diferentes de interacción con el usuario.

Si está trabajando en un código existente y está retrocediendo, guárdelo en algún lugar y haga un prototipo de algo similar en su lugar.

Haz lo siguiente.

1. Escriba "Estoy creando prototipos" en una nota adhesiva y péguela en el costado de tu pantalla
2. Recuerde que los prototipos están hechos para fallar. Y recuerda que los prototipos se tiran, incluso si no fallan.
No hay inconveniente en hacer esto.
3. En su búfer de editor vacío, cree un comentario que describa en uno
ora lo que quieres aprender o hacer.
4. Comience a codificar.

Si empiezas a tener dudas, mira la nota adhesiva.

Si, en medio de la codificación, esa duda persistente cristaliza repentinamente en una preocupación sólida, entonces abórdela.

Si llega al final del experimento y todavía se siente incómodo, comience de nuevo con la caminata, la charla y el tiempo libre.

Pero, según nuestra experiencia, en algún momento durante el primer prototipo, se sorprenderá al encontrarse tarareando junto con su música, disfrutando la sensación de crear código. El nerviosismo se habrá evaporado, reemplazado por un sentimiento de urgencia: ¡hagámoslo!

En esta etapa, ya sabes qué hacer. Elimine todo el código prototipo, deseche la nota adhesiva y llene ese búfer vacío del editor con un código nuevo y brillante.

NO SOLO TU CÓDIGO

Una gran parte de nuestro trabajo se trata de código existente, a menudo escrito por otras personas. Esas personas tendrán instintos diferentes a los tuyos, por lo que las decisiones que tomen serán diferentes. No necesariamente peor; Sólo diferente.

Puede leer su código mecánicamente, repasándolo y tomando notas sobre cosas que parecen importantes. Es una tarea, pero funciona.

O puedes intentar un experimento. Cuando vea cosas hechas de una manera que parezca extraña, anótelas. Continúe haciendo esto y busque patrones. Si puede ver qué los llevó a escribir código de esa manera, es posible que el trabajo de comprenderlo se vuelva mucho más fácil. Podrás aplicar conscientemente los patrones que aplicaron tácitamente.

Y es posible que aprendas algo nuevo en el camino.

NO SOLO CÓDIGO

Aprender a escuchar tu instinto cuando codificas es una habilidad importante para fomentar. Pero se aplica a la imagen más grande están bien. A veces, un diseño simplemente se siente mal, o algún requisito te hace sentir incómodo. Deténgase y analice estos sentimientos. Si se encuentra en un entorno de apoyo, expréselo en voz alta. Explóralos. Lo más probable es que haya algo al acecho en esa puerta oscura. Escuche sus instintos y evite el problema antes de que salte a la vista.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 13, Prototipos y Post-it Notes
- Tema 22, Cuadernos diarios de ingeniería
- Tema 46, Resolviendo acertijos imposibles

RETOS

- ¿Hay algo que sabes que debes hacer, pero lo has pospuesto porque da un poco de miedo o es difícil? Aplicar las técnicas de esta sección. Reduzca el tiempo a una hora, tal vez dos, y prométase que cuando suene la campana borrará lo que hizo. ¿Qué aprendiste?



Tema 38

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. Deberíamos evitar la programación por coincidencia, confiando en la suerte y los éxitos accidentales, en favor de la programación deliberada.

COMO 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, en primer lugar, no sabía por qué funcionaba. 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 sabemos más. 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 el código está escrito actualmente. 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 desea, pero parece funcionar bien. Llamar a las cosas en el orden incorrecto, o en el contexto incorrecto, es un problema relacionado.

Aquí parece que Fred está tratando desesperadamente de mostrar algo en la pantalla usando algún marco de renderizado GUI particular:

```
pintar();  
invalidar();  
validar();  
revalidar();  
repintar();  
pintarInmediatamente();
```

Pero estas rutinas nunca fueron diseñadas para llamarse así; aunque parecen funcionar, en realidad es solo una coincidencia.

Para colmo de males, cuando finalmente se dibuja la escena, Fred no intentará volver atrás y eliminar las llamadas falsas. "Funciona ahora, mejor déjalo lo suficientemente bien solo..."

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, más núcleos de CPU), 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 aumentan el riesgo de introducir nuevos errores de sus 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 ([consulte el Tema 23, Diseño por contrato](#)) 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.

¿No es lo

suficientemente cerca? Una vez trabajamos en un gran proyecto que informaba sobre datos alimentados desde una gran cantidad de unidades de recopilación de datos de hardware en el campo. Estas unidades abarcaban estados y zonas horarias y, por diversas razones logísticas e históricas, cada unidad ^[50] se configuró en la hora local. Como resultado de las interpretaciones conflictivas de la zona horaria y las inconsistencias en las políticas del horario de verano, los resultados casi siempre fueron incorrectos, pero solo se desviaron por uno. Los desarrolladores del proyecto se habían acostumbrado a simplemente sumar uno o restar uno para obtener la respuesta correcta, razonando que solo fallaba por uno en esta situación. Y luego la siguiente función vería el valor como desactivado por el otro lado, y lo cambiaría de nuevo.

Pero el hecho de que "solo" estuviera fuera de lugar por una parte del tiempo fue una coincidencia, enmascarando una falla más profunda y fundamental. Sin un modelo adecuado de gestión del tiempo, todo el gran código base se había convertido con el tiempo en una masa insostenible de declaraciones [+1](#) y [-1](#). Al final, nada de eso fue correcto y el proyecto fue desecharido.

Patrones fantasma

Los seres humanos están diseñados para ver patrones y causas, incluso cuando es solo una coincidencia. Por ejemplo, los líderes rusos siempre alternan entre ser calvo y peludo: un líder estatal calvo (o obviamente calvo) de Rusia ha sucedido a uno no calvo ("peludo"), y viceversa, durante casi 200 años. ^[51]

Pero aunque no escribirías código que dependiera del siguiente

El líder ruso es calvo o peludo, en algunos dominios pensamos de esa manera todo el tiempo. Los jugadores imaginan patrones en números de lotería, juegos de dados o ruleta, cuando en realidad se trata de eventos estadísticamente independientes. En finanzas, el comercio de acciones y bonos está igualmente plagado de coincidencias en lugar de patrones reales y discernibles.

Un archivo de registro que muestra un error intermitente cada 1000 solicitudes puede ser una condición de carrera difícil de diagnosticar o puede ser un error simple. Las pruebas que parecen pasar en su máquina pero no en el servidor pueden indicar una diferencia entre los dos entornos, o tal vez sea solo una coincidencia.

No lo asumas, demuéstralos.

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?

¿Confía en que se pueda escribir en el directorio actual? ¿Están presentes ciertas variables de entorno o archivos de configuración? Si la hora en el servidor es precisa, ¿dentro de qué tolerancia? ¿Confía en la disponibilidad y la velocidad de la red?

Cuando copió el código de la primera respuesta que encontró en la red, ¿está seguro de que su contexto es el mismo? ¿O está construyendo un código de "culto a la carga", simplemente imitando la forma sin contenido?^[52]

Encontrar una respuesta que se ajuste no es lo mismo que la

respuesta correcta.

Consejo 62

No programes por coincidencia

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 causa Y, pero como dijimos en el Tema 20, Depuración: no lo asuma, pruébelo.

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.

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 aquí.
- ¿Puede explicar el código, en detalle, a un programador más joven? Si no es así, quizás estés confiando en las coincidencias.
- No codifiques en la oscuridad. Cree una aplicación que no entienda por completo, o use una tecnología que no entienda, y es probable que las coincidencias lo muerdan. Si no está seguro de por qué funciona, no sabrá por qué falla.
- Proceda a partir de un plan, ya sea que ese plan esté en su cabeza, en el reverso de una servilleta de cóctel o en una pizarra.

- Confía solo en cosas confiables. No dependas de suposiciones. Si no puede saber si algo es confiable, asuma lo peor.
- Documente sus suposiciones. El Tema 23, Diseño por contrato, puede ayudar a aclarar sus suposiciones en su propia mente, así como ayudar a comunicarlas a otros.
- No solo pruebe su código, sino también sus suposiciones. No adivine; en realidad intentarlo. Escriba una afirmación para probar sus suposiciones (consulte el Tema 25, Programación asertiva). 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 tiene los fundamentos o la infraestructura correctos, las 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 deje que lo que ya ha hecho restrinja lo que hará a continuación: esté listo para refactorizar (consulte el Tema 40, Refactorización).
Esta decisión puede afectar el cronograma del proyecto. La suposición es que el impacto será menor que el costo de no hacer el cambio. [53]

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

- Tema 4, Sopa de piedra y ranas hervidas
- Tema 9, DRY—Los males de la duplicación
- Tema 23, Diseño por contrato
- Tema 34, El estado compartido es un estado incorrecto
- Tema 43, Manténgase seguro ahí afuera

EJERCICIOS

Ejercicio 25 (respuesta posible)

Una fuente de datos de un proveedor le brinda una matriz de tuplas que representan pares clave-valor. La clave de `DepositAccount` contendrá una cadena del número de cuenta en el valor correspondiente:

```
[  
...  
{:DepositAccount, "564-904-143-00"}  
...  
]
```

Funcionó perfectamente en la prueba en las computadoras portátiles de desarrollo de 4 núcleos y en la máquina de compilación de 12 núcleos, pero en los servidores de producción que se ejecutan en contenedores, sigue obteniendo números de cuenta incorrectos. ¿Qué está sucediendo?

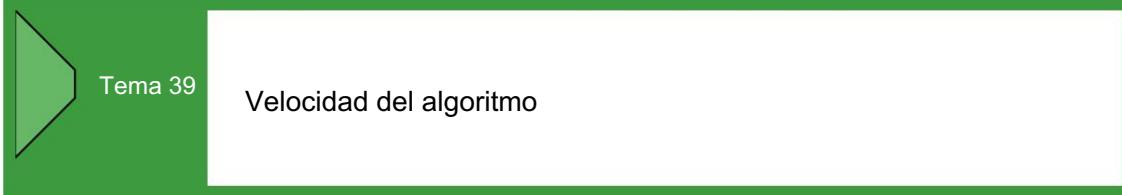
Ejercicio 26 (respuesta posible)

Está codificando un marcador automático para alertas de voz y tiene que administrar una base de datos de información de contacto. La ITU especifica que los números de teléfono no deben tener más de 15 dígitos, por lo que almacena el número de teléfono del contacto en un campo numérico garantizado para contener al menos 15 dígitos. Ha realizado pruebas exhaustivas en toda América del Norte y todo parece estar bien, pero de repente recibe una serie de quejas de otras partes del mundo. ¿Por qué?

Ejercicio 27 (respuesta posible)

Ha escrito una aplicación que amplía las recetas comunes para el comedor de un crucero con capacidad para 5000 personas. Pero recibe quejas de que las conversiones no son precisas. Verifica y el código usa la fórmula de conversión de 16 tazas a un galón.

Así es, ¿no?



En el Tema 15, Estimación, hablamos sobre la estimación de cosas como cuánto tiempo se tarda en caminar por la ciudad o cuánto se tardará en 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. en.

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 Big-O .

¿QUÉ ENTENDEMOS POR ALGORITMOS DE ESTIMACIÓN?

La mayoría de los algoritmos no triviales manejan algún tipo de entrada variable: ordenar cadenas, invertir una matriz o descifrar un mensaje con una clave de bits. Normalmente, el tamaño de esta entrada afectará al algoritmo: cuanto mayor sea la entrada, mayor será el tiempo de ejecución o más memoria utilizada.

Si la relación fuera siempre lineal (de modo que el tiempo aumentara en proporción directa al valor de), 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, para

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 nos encontramos realizando un análisis más detallado. Ahí es cuando la notación Big-O es útil.

NOTACIÓN O GRANDE

La notación Big-O, escrita sobre , es una forma matemática de aproximaciones. Cuando decimos que una rutina de clasificación en particular clasifica los registros en el tiempo, simplemente estamos diciendo que el peor de los casos, el tiempo tomado variará como el cuadrado de . Duplica el número de registros y el tiempo se multiplicará aproximadamente por cuatro. Piense en el como significado en el orden de.

La notación pone un límite superior al valor de lo que estamos midiendo (tiempo, memoria, etc.). Si decimos que una función toma tiempo, 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:

En realidad, esta es una característica de la notación: un algoritmo puede ser 1000 veces más rápido que otro algoritmo, pero no lo sabrá por la notación. Big-O nunca le dará números reales para el tiempo o la memoria o lo que sea: simplemente le dice cómo cambiarán estos valores a medida que cambie la entrada.

La Figura 3, Tiempos de ejecución de varios algoritmos, muestra varias notaciones comunes con las que se encontrará, 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 □.

Por ejemplo, suponga que tiene una rutina que tarda un segundo en procesar 100 registros. ¿Cuánto tiempo tomará procesar 1,000? Si su código es □ entonces, aún tomará un segundo. Si es así, □, probablemente estarás esperando unos tres segundos. □ mostrará un aumento lineal de diez segundos, mientras que □ tardará unos 33 segundos. Si tiene la mala suerte de tener una □ rutina, siéntese durante 100 segundos mientras hace su trabajo. Y si está utilizando un algoritmo exponencial, es □ posible que desee preparar una taza de café: su rutina debería terminar en unos □ años. Háganos saber cómo termina el universo.

La □ notación no se aplica solo al tiempo; puede usarlo para representar cualquier otro recurso utilizado por un algoritmo. Por ejemplo, a menudo es útil poder modelar el consumo de memoria (consulte los ejercicios para ver un ejemplo).

Logarítmica (búsqueda binaria). La base del logaritmo no
materia, por lo que esto es equivalente.

Lineal (búsqueda secuencial)

Peor que lineal, pero no mucho peor. (Tiempo de ejecución promedio de quicksort,
heapsort)

Ley cuadrática (clases de selección e inserción)

Cúbico (multiplicación de dos matrices)

Exponencial (problema del viajante de comercio, partición de conjuntos)

imágenes/grande-o.png



Figura 3. Tiempos de ejecución de varios algoritmos

ESTIMACIÓN DE SENTIDO COMÚN

Puede estimar el orden de muchos algoritmos básicos usando sentido común.

Bucles

simples Si un bucle simple va de a , entonces el algoritmo es probablemente el tiempo aumenta linealmente con .

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 se convierte en dónde están 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 .

Corte binario

Si su algoritmo reduce a la mitad el conjunto de cosas que considera cada vez que da la vuelta al ciclo, entonces es probable que sea logarítmico.

· Una búsqueda binaria de una lista ordenada, atravesar un árbol binario y encontrar el primer bit establecido en una palabra de máquina puede todo sea .

Divide y vencerás

Los algoritmos que dividen su trabajo de entrada en las dos mitades de forma independiente y luego combinan el resultado. El

ejemplo clásico es el ordenamiento rápido, que funciona dividiendo los datos en dos mitades y ordenando cada una recursivamente. Aunque técnicamente, debido a que su comportamiento se degrada cuando se alimenta de entrada ordenada, el promedio

el tiempo de ejecución de quicksort es .

combinatoria

Cada vez que los algoritmos comienzan a observar las permutaciones de las cosas, sus tiempos de ejecución pueden salirse de control.

Esto se debe a que las permutaciones involucran factoriales (hay permutaciones de los dígitos del 1 al 5).

Cronometre un algoritmo combinatorio para cinco elementos: llevará seis veces más tiempo ejecutarlo para seis y 42 veces más para siete. Los ejemplos incluyen algoritmos para muchos de los problemas difíciles reconocidos: el problema del viajante de comercio, 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.

LA 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 algoritmo. Si ese ciclo contiene un ciclo interno, entonces estás viendo . Debería preguntarse qué tan grandes pueden llegar a ser estos valores. 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.

Consejo 63

Calcule el orden de sus algoritmos

Hay algunos enfoques que puede tomar para abordar los problemas potenciales. Si tiene un algoritmo que es , trate de encontrar un enfoque de divide y vencerás que te lleve a .



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 ciclo simple puede funcionar mejor que uno complejo para valores más pequeños de , particularmente si el algoritmo tiene un ciclo interno costoso.

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. Intenta 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. Esto lleva a nuestro próximo consejo.

Consejo 64

Pruebe sus estimaciones

Si es complicado obtener tiempos precisos, use generadores de perfiles de código para contar la cantidad de veces que se ejecutan los diferentes pasos en su algoritmo, y represente 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 tenga cuidado con la 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.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 15, Estimación

RETOS

- 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 ([Algorithms \[SW11\]](#) [An Introduction to the Analysis of Algorithms \[SF13\]](#) 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 los libros definitivos de Donald Knuth [El arte de la programación informática](#), que analizan una amplia gama de algoritmos.
- [El arte de la programación informática, Volumen 1:](#)

Algoritmos fundamentales [Knu98]

- El arte de la programación informática, Volumen 2:
Algoritmos semiméricos [Knu98a]
 - El arte de la programación informática, Volumen 3: Clasificación y búsqueda [Knu98b]
 - El arte de la programación informática, Volumen 4A:
Algoritmos combinatorios, Parte 1 [Knu11].
-
- En el primer ejercicio que sigue, 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

Ejercicio 28 (respuesta posible)

Codificamos un conjunto de rutinas de clasificación simples en Rust.
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?

Ejercicio 29 (respuesta posible)

En Common Sense Estimation, afirmamos que una tajada binaria es . ¿Puedes probar esto?

Ejercicio 30 (respuesta posible)

En la Figura 3, Tiempos de ejecución de varios algoritmos, afirmamos eso es lo mismo que (o, de hecho, logaritmos en cualquier base). Poder

[] explicas porque? []



Tema 40

refactorización

Cambio y decadencia en
todo lo que veo...

HF Lyte, quédate 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. El trabajo clásico de Bertrand Meyer, Construcción de software orientada a objetos [Mey97], utiliza el término "Construcción de software", e incluso sus humildes autores editaron la columna de Construcción de software para IEEE Software a principios de la década de 2000.^[55]

Pero usar la construcción como metáfora guía implica los siguientes pasos:

1. Un arquitecto elabora planos.
2. Los contratistas cavan los cimientos, construyen la superestructura, cablean y aploman y aplicar los toques finales.
3. Los inquilinos se mudan y viven felices para siempre, llamando al edificio mantenimiento para solucionar cualquier problema.

Bueno, el software no funciona de esa manera. Más que una construcción, el software se parece más a la 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. Quizás cierta rutina se ha vuelto demasiado grande o está tratando de lograr demasiado; debe dividirse en dos. Las cosas que no funcionan según lo planeado deben ser eliminadas o podadas.

Reescribir, reelaborar y rediseñar el código se conoce colectivamente como reestructuración. Pero hay un subconjunto de esa actividad que se ha practicado como refactorización.

Martin Fowler define la refactorización [Fow19] como:

técnica disciplinada para reestructurar un cuerpo de código existente, alterando su estructura interna sin cambiar su comportamiento externo.

Las partes críticas de esta definición son que:

1. La actividad es disciplinada, no libre para todos.

2. El comportamiento externo no cambia; este no es el momento de agregar características

La refactorización no pretende ser una actividad especial, de alta ceremonia, de vez en cuando, como arar todo el jardín para volver a plantar. En cambio, la refactorización es una actividad del día a día, tomando pequeños pasos de bajo riesgo, más como desmalezar y rastrillar. En lugar de una reescritura total y gratuita del código base, se trata de un enfoque específico y de precisión para ayudar a mantener el código fácil de cambiar.

Para garantizar que el comportamiento externo no haya cambiado, necesita buenas pruebas unitarias automatizadas que validen el comportamiento del código.

¿CUÁNDO DEBE REFACTAR?

Refactorizas cuando has aprendido algo; cuando entiendes algo mejor que el año pasado, ayer o incluso hace diez minutos.

Tal vez te hayas topado 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 cambiarla . . No hay tiempo como el presente. Cualquier número de cosas puede hacer que el código califique para la refactorización:

Duplicación

Has descubierto una violación del principio DRY.

Diseño no ortogonal Has

descubierto algo que podría hacerse más ortogonal.

Conocimiento obsoleto

Las cosas cambian, los requisitos se desvían y su conocimiento del problema aumenta. El código necesita mantenerse al día.

Uso A

medida que personas reales utilizan el sistema en circunstancias reales, se da cuenta de que algunas funciones ahora son más importantes de lo que se pensaba anteriormente, y las funciones "imprescindibles" quizás no lo eran.

Rendimiento

Debe mover la funcionalidad de un área del sistema a otra para mejorar el rendimiento.

Las pruebas pasan

Sí. En serio. Dijimos que la refactorización debería ser una actividad a pequeña escala, respaldada por buenas pruebas. Entonces, cuando haya agregado una pequeña cantidad de código y pase esa prueba adicional, ahora tiene una gran oportunidad para sumergirse y ordenar lo que acaba de escribir.

Refactorizar su código (mover la funcionalidad y actualizar las decisiones anteriores) es realmente un ejercicio de manejo del dolor. Seamos realistas, cambiar el código fuente puede ser bastante doloroso: estaba funcionando, tal vez sea mejor dejarlo en paz. Muchos desarrolladores son reacios a entrar y volver a abrir un fragmento de código solo porque no está del todo bien.

Complicaciones del mundo

real Así que acude a tus compañeros de equipo o cliente y les dices: "Este código funciona, pero necesito otra semana para refactorizarlo por completo".

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.

Es posible que desee explicar este principio a los demás mediante el uso de una analogía médica: piense en el código que necesita refactorización 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.

Consejo 65

Refactorice temprano, refactorice a menudo

Los daños colaterales en el código pueden ser igual de mortales con el tiempo (consulte el Tema 3, [Entropía del software](#)). La refactorización, como con la mayoría de las cosas, es más fácil de hacer mientras los problemas son pequeños, como una actividad continua mientras se codifica. No debería necesitar "una semana para refactorizar" un fragmento de código, eso es una reescritura completa. Si ese nivel de interrupción es necesario, es posible que no pueda hacerlo de inmediato. En cambio, asegúrese de que se coloque en el cronograma. Asegúrese de que los usuarios del código afectado sepan que está programado para ser reescrito y cómo esto podría afectarlos.

¿CÓMO REFACCIONAR?

La refactorización comenzó en la comunidad de Smalltalk y acababa de empezar a ganar una audiencia más amplia cuando escribimos la primera edición de este libro, probablemente gracias al primer libro importante sobre refactorización ([Refactoring: Improving the Design of Existing Code \[Fow19\]](#), ahora en su [segunda edición](#)).

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:^[56]

1. No intente refactorizar y agregar funcionalidad al mismo tiempo.
2. Asegúrese de tener buenas pruebas antes de comenzar a refactorizar. Correr las pruebas con la mayor frecuencia posible. De esa manera, sabrá rápidamente si sus cambios han roto algo.
3. Tome medidas breves y deliberadas: mueva un campo de una clase a otra, divida un método, cambie el nombre de una variable. 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 [57] una depuración prolongada.

Refactorización automática

En la primera edición notamos que “esta tecnología aún no ha aparecido fuera del mundo de Smalltalk, pero es probable que esto cambie...”. Y, de hecho, lo hizo, ya que la refactorización automática está disponible en muchos IDE y para la mayoría de los lenguajes principales.

Estos IDE pueden cambiar el nombre de variables y métodos, dividir una rutina larga en otras más pequeñas, propagar automáticamente los cambios necesarios, arrastrar y soltar para ayudarlo a mover el código, etc.

Hablaremos más sobre las pruebas a este nivel en el Tema 41, Test to Code, y las pruebas a mayor escala en Ruthless and Continuous Testing, pero el punto del Sr. Fowler de mantener una buena regresión

tests es la clave para refactorizar de forma segura.

Si tiene que ir más allá de la refactorización y termina cambiando el comportamiento externo o las interfaces, entonces puede ser útil interrumpir deliberadamente la compilación: los clientes antiguos de este código deberían fallar al compilar.

De esa manera, sabrá qué necesita actualizarse. La próxima vez que vea un fragmento de código que no es como debería ser, arréglelo. Controle el dolor: si le duele ahora, pero le va a doler aún más más tarde, es mejor que termine de una vez. Recuerde las lecciones del Tema 3, Entropía del software: no viva con las ventanas rotas.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 3, [Entropía del software](#)
- Tema 9, [DRY—Los males de la duplicación](#)
- Tema 12, [Viñetas trazadoras](#)
- Tema 27, [No superes tus faros](#)
- Tema 44, [Nombrar cosas](#)
- Tema 48, [La esencia de la agilidad](#)



Tema 41

Prueba de código

La primera edición de este libro se escribió en tiempos más primitivos, cuando la mayoría de los desarrolladores no escribían pruebas. Para qué molestarse, pensaron, el mundo se acabaría en el año 2000 de todos modos.

En ese libro, teníamos una sección sobre cómo crear código que fuera fácil de probar. Era una forma astuta de convencer a los desarrolladores para que realmente escribieran pruebas.

Estos son tiempos más ilustrados. Si hay algún desarrollador que aún no escribe pruebas, al menos saben que deberían hacerlo.

Pero todavía hay un problema. Cuando les preguntamos a los desarrolladores por qué escriben pruebas, nos miran como si solo les preguntáramos si todavía codifican con tarjetas perforadas y dirían "para asegurarse de que el código funcione", con un tácito "tonto" al final. Y creemos que eso está mal.

Entonces, ¿qué creemos que es importante acerca de las pruebas? ¿Y cómo creemos que debería hacerlo?

Comencemos con la afirmación en negrita:

Consejo 66

La prueba no se trata de encontrar errores

Creemos que los principales beneficios de las pruebas ocurren cuando piensa y escribe las pruebas, no cuando las ejecuta.

PENSANDO EN PRUEBAS

Es un lunes por la mañana y te acomodas para empezar a trabajar en un código nuevo. Tiene que escribir algo que consulte la base de datos para obtener una lista de personas que miran más de 10 videos a la semana en su sitio de "videos de lavado de platos más divertidos del mundo".

Enciende su editor y comienza escribiendo la función que realiza la consulta:

```
def return_avid_viewers hacer  
... hmmmm ...  
# final
```

¡Detener! ¿Cómo sabes que lo que estás a punto de hacer es algo bueno?

La respuesta es que no puedes saber eso. Nadie puede. Pero pensar en las pruebas puede hacerlo más probable. Así es como funciona.

Comience imaginando que ha terminado de escribir la función y ahora tiene que probarla. ¿Cómo lo harías tú? Bueno, le gustaría usar algunos datos de prueba, lo que probablemente significa que quiere trabajar en una base de datos que usted controla. Ahora, algunos marcos pueden manejar eso por usted, ejecutando pruebas en una base de datos de prueba, pero en nuestro caso eso significa que deberíamos pasar la instancia de la base de datos a nuestra función en lugar de usar una global, ya que eso nos permite cambiarla mientras probamos:

```
def return_avid_users(db) hacer
```

Entonces tenemos que pensar en cómo poblaríamos esos datos de prueba. El requisito solicita una "lista de personas que ven más de 10 videos a la semana". Así que miramos el esquema de la base de datos para los campos.

eso podría ayudar. Encontramos dos campos probables en una tabla de quién vio qué: [video_abierto](#) y [video_completado](#). Para escribir nuestros datos de prueba, necesitamos saber qué campo usar. Pero no sabemos qué significa el requisito, y nuestro contacto comercial no está disponible.

Hagamos trampa y pasemos el nombre del campo (lo que nos permitirá probar lo que tenemos y posiblemente cambiarlo más adelante):

```
def return_avid_users(db, nombre_del_campo_de_calificación) do
```

Comenzamos pensando en nuestras pruebas, y sin escribir una línea de código, ya hicimos dos descubrimientos y los usamos para cambiar la API de nuestro método.

PRUEBAS CONDUCCIÓN CODIFICACIÓN

En el ejemplo anterior, pensar en las pruebas nos hizo reducir el acoplamiento en nuestro código (al pasar una conexión de base de datos en lugar de usar una global) y aumentar la flexibilidad (al convertir el nombre del campo que probamos en un parámetro). Pensar en escribir una prueba para nuestro método nos hizo mirarlo desde afuera, como si fuéramos un cliente del código y no su autor.

Consejo Una prueba es el primer usuario de su código

Creemos que este es probablemente el mayor beneficio que ofrecen las pruebas: las pruebas son comentarios vitales que guían su codificación.

Una función o método que está estrechamente acoplado a otro código es difícil de probar, porque debe configurar todo ese entorno antes de poder ejecutar su método. Entonces, hacer que tus cosas sean comprobables también reduce su acoplamiento.

Y antes de que puedas probar algo, tienes que entenderlo.

Eso suena tonto, pero en realidad todos nos hemos lanzado a un pedazo de

código basado en una comprensión nebulosa de lo que teníamos que hacer. Nos aseguramos de que lo resolveremos sobre la marcha. Ah, y también agregaremos todo el código para admitir las condiciones de contorno más adelante. Ah, y el manejo de errores. Y el código termina siendo cinco veces más largo de lo que debería porque está lleno de lógica condicional y casos especiales. Pero brille la luz de una prueba en ese código, y las cosas se aclararán. Si piensa en probar las condiciones de contorno y cómo funcionará eso antes de comenzar a codificar, es posible que encuentre los patrones en la lógica que simplificarán la función. Si piensa en las condiciones de error que necesitará probar, estructurará su función en consecuencia.

Desarrollo basado en

pruebas Hay una escuela de programación que dice que, dados todos los beneficios de pensar en las pruebas desde el principio, ¿por qué no seguir adelante y escribirlas también desde el principio? Practican algo llamado desarrollo impulsado por pruebas o TDD. También verá esto llamado test-first desarrollo. [58]

El ciclo básico de TDD es:

1. Decida qué pequeña función desea agregar.
2. Escriba una prueba que pasará una vez que se implemente esa funcionalidad.
3. Ejecute todas las pruebas. Verifica que la única falla sea la que acabas de escribir.
4. Escriba la menor cantidad de código necesaria para que la prueba pase y verifique que las pruebas ahora se ejecutan sin problemas.
5. Refactorice su código: vea si hay una forma de mejorar lo que acaba de escribir (la prueba o la función). Asegúrate de que las pruebas aún pasen cuando hayas terminado.

La idea es que este ciclo debe ser muy corto: cuestión de minutos, de modo que esté constantemente escribiendo pruebas y luego haciendo que funcionen.

Vemos un gran beneficio en TDD para las personas que recién comienzan con las pruebas. Si sigue el flujo de trabajo de TDD, garantizará que siempre tendrá pruebas para su código. Y eso significa que siempre estarás pensando en tus exámenes.

Sin embargo, también hemos visto a personas convertirse en esclavas de TDD. Esto se manifiesta de varias maneras:

- Dedican una cantidad excesiva de tiempo a asegurarse de que siempre tengan una cobertura de prueba del 100 %.
- Tienen muchas pruebas redundantes. Por ejemplo, antes de escribir una clase por primera vez, muchos seguidores de TDD primero escribirán una prueba fallida que simplemente hace referencia al nombre de la clase. Falla, luego escriben una definición de clase vacía y pasa. Pero ahora tienes una prueba que no hace absolutamente nada; la próxima prueba que escriba también hará referencia a la clase, por lo que hace que la primera sea innecesaria. Hay más cosas para cambiar si el nombre de la clase cambia más adelante. Y esto es sólo un ejemplo trivial.
- Sus diseños tienden a comenzar desde abajo y avanzar hacia arriba.
(Ver De abajo hacia arriba vs. De arriba hacia abajo vs. La forma en que debe hacerlo).

De abajo hacia arriba versus de arriba hacia abajo versus la forma en que debe hacerlo

Cuando la informática era joven y despreocupada, había dos escuelas de diseño: de arriba hacia abajo y de abajo hacia arriba. La gente de arriba hacia abajo dijo que debe comenzar con el problema general que está tratando de resolver y dividirlo en una pequeña cantidad de piezas. Luego divide cada uno de estos en partes más pequeñas, y así sucesivamente, hasta que termine con partes lo suficientemente pequeñas como para expresarlas en código.

La gente de abajo hacia arriba construye código como si construyeras una casa. Comienzan desde abajo, produciendo una capa de código que les brinda algunas abstracciones que están más cerca del problema que están tratando de resolver. Luego agregan otra capa, con abstracciones de mayor nivel. Continúan hasta que la capa final es una abstracción que resuelve el problema. "Hazlo así..."

Ninguna escuela funciona realmente, porque ambas ignoran uno de los aspectos más importantes del desarrollo de software: no sabemos lo que estamos haciendo cuando empezamos. Las personas de arriba hacia abajo asumen que pueden expresar todo el requisito por adelantado: no pueden. La gente de abajo hacia arriba asume que puede construir una lista de abstracciones que eventualmente los llevará a un

única solución de nivel superior, pero ¿cómo pueden decidir sobre la funcionalidad de las capas cuando no saben hacia dónde se dirigen?

Consejo 68

Cree de extremo a extremo, no de arriba hacia abajo o de abajo hacia arriba

Creemos firmemente que la única forma de construir software es incrementalmente. Cree pequeñas piezas de funcionalidad de extremo a extremo, aprendiendo sobre el problema a medida que avanza. Aplique este aprendizaje a medida que continúa desarrollando el código, involucre al cliente en cada paso y pídale que guíe el proceso.

Por supuesto, practica TDD. Pero, si lo hace, no olvide detenerse de vez en cuando y mirar el panorama general. Es fácil dejarse seducir por el mensaje verde de "[pruebas aprobadas](#)" , escribiendo mucho código que en realidad no lo acerca a una solución.

TDD: NECESITAS SABER DONDE VAS

El viejo chiste pregunta "¿Cómo te comes un elefante?" El remate: "Un bocado a la vez". Y esta idea a menudo se promociona como un beneficio de TDD. Cuando no pueda comprender todo el problema, dé pequeños pasos, una prueba a la vez. Sin embargo, este enfoque puede confundirlo, animándolo a concentrarse y pulir sin cesar los problemas fáciles mientras ignora la verdadera razón por la que está codificando. Un ejemplo interesante de esto sucedió en 2006, cuando Ron Jeffries, una figura destacada en el movimiento de la agilidad, comenzó una serie de publicaciones de blog que documentaban su codificación basada en pruebas de un ^[59]solucionador de Sudoku. Después de cinco publicaciones, refinó la representación del tablero subyacente, refactorizando varias veces hasta que estuvo satisfecho con el modelo de objetos. Pero luego abandonó el proyecto. Es interesante leer las publicaciones del blog en orden y ver cómo una persona inteligente puede distraerse por las minucias, reforzadas por el entusiasmo de aprobar las pruebas.

Como contraste, Peter Norvig describe un enfoque alternativo

[60]

que se siente de carácter muy diferente: en lugar de estar impulsado por las pruebas, comienza con una comprensión básica de cómo se resuelven tradicionalmente este tipo de problemas (usando la propagación de restricciones) y luego se enfoca en refinar su algoritmo. Aborda la representación del tablero en una docena de líneas de código que fluyen directamente de su discusión sobre la notación.

Las pruebas definitivamente pueden ayudar a impulsar el desarrollo. Pero, como con cada viaje, a menos que tenga un destino en mente, puede terminar yendo en círculos.

VOLVER AL CÓDIGO

El desarrollo basado en componentes ha sido durante mucho tiempo un objetivo elevado del desarrollo^[61] de software. La idea es que los componentes de software genéricos estén disponibles y se combinen tan fácilmente como se combinan los circuitos integrados (CI) comunes. Pero esto funciona solo si se sabe que los componentes que está utilizando son confiables y si tiene voltajes comunes, estándares de interconexión, sincronización, etc.

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 autopregunta 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.

EXAMEN DE LA UNIDAD

Las pruebas a nivel de chip para hardware son aproximadamente equivalentes a las pruebas unitarias en software: pruebas realizadas en cada módulo, de forma aislada, para verificar su comportamiento. Podemos tener una mejor idea de cómo reaccionará un módulo en el gran mundo una vez que lo hayamos probado a fondo en condiciones controladas (incluso artificiales).

Una prueba unitaria de software es un código que ejercita un módulo. Por lo general, la prueba unitaria establecerá algún tipo de entorno artificial y luego invocará rutinas en el módulo que se está probando. 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 unitaria para probar el sistema en su totalidad. Hablamos de esta comprobación a gran escala del sistema en Ruthless and Continuous Testing.

Sin embargo, antes de llegar tan lejos, debemos decidir qué probar a nivel de unidad. Históricamente, los programadores lanzaban algunos bits aleatorios de datos al código, miraban las declaraciones de impresión y lo llamaban probado. Podemos hacer mucho mejor.

PRUEBA CONTRA CONTRATO

Nos gusta pensar en las pruebas unitarias como pruebas contra contrato (consulte el Tema 23, Diseño por contrato). 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 de contorno.

¿Qué significa esto en la práctica? Comencemos con un ejemplo numérico simple: una rutina de raíz cuadrada. Su contrato documentado es simple:

condiciones

previas: argumento ≥ 0 ;

condiciones

posteriores: $((resultado * resultado) - argumento).abs \leq epsilon * argumento$;

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 (ϵ psilon).

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.

Entonces podemos llamar a esta rutina para probar nuestra función de raíz cuadrada:

```
afirmarDentro de Epsilon(mi_raíz(0), 0)
afirmarDentro de Epsilon(mi_raíz(2.0), 1.4142135624)
afirmarDentro de Epsilon(mi_raíz(64.0), 8.0)
afirmarDentro de Epsilon(mi_raíz(1.0e7), 3162.2776602)
assertRaisesException fn => mi_raíz(-4) .0 ) fin
```

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 un [DataFeed](#) y un [LinearRegression](#). En orden, probaríamos:

1. El contrato de [DataFeed](#), en su totalidad
2. El contrato de [LinearRegression](#), en su totalidad
3. El contrato de A, que se basa en los otros contratos pero no exponerlos directamente

Este estilo de prueba requiere que primero pruebe los subcomponentes de un módulo. Una vez que se han verificado los subcomponentes, se puede probar el módulo en sí.

Si las pruebas de [DataFeed](#) y [LinearRegression](#) pasaron, pero la prueba de A falló, podemos estar bastante seguros de que el problema está en A, o en el uso que hace A de uno de esos subcomponentes. Esta técnica es una excelente manera de reducir el esfuerzo de depuración: podemos concentrarnos rápidamente en el origen probable del problema dentro del módulo A y no perder el tiempo en volver a examinar 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.

Consejo 69

Diseño para probar

PRUEBAS AD HOC

No debe confundirse con "trucos extraños", las pruebas ad-hoc son cuando

ejecute poke en nuestro código manualmente. Esto puede ser tan simple como un [archivo console.log\(\)](#), o un fragmento de código ingresado de manera interactiva en un depurador, entorno IDE o REPL.

Al final de la sesión de depuración, debe formalizar esta 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 al arsenal de pruebas unitarias existente.

CONSTRUIR 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 pines de prueba en el software, pero podemos proporcionar varias vistas del estado interno de un módulo, sin usar el depurador (lo que puede ser inconveniente o imposible en una aplicación de producción).

Los archivos de registro que contienen mensajes de seguimiento son uno de 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" o la URL mágica. Cuando se presiona esta combinación particular de teclas, o se accede a la URL, aparece una ventana de control de diagnóstico con mensajes de estado, etc. esto no es

algo que normalmente revelaría a los usuarios finales, pero que puede ser muy útil para la mesa de ayuda.

De manera más general, podría usar un cambio de función para habilitar diagnósticos adicionales para un usuario o clase de usuarios en particular.

Una confesión

Se sabe que yo (Dave) le digo a la gente que ya no escribo exámenes. En parte lo hago para sacudir la fe de aquellos que han convertido la prueba en una religión. Y en parte lo digo porque es (algo) cierto.

He estado programando durante 45 años y escribiendo pruebas automatizadas para más de 30 de ellos. Pensar en las pruebas está integrado en la forma en que abordo la codificación. Se sentía cómodo. Y mi personalidad insiste en que cuando algo comienza a sentirse cómodo, debo pasar a otra cosa.

En este caso, decidí dejar de escribir pruebas durante un par de meses y ver qué le pasaba a mi código. Para mi sorpresa, la respuesta fue "no mucho". Así que pasé un tiempo averiguando por qué.

Creo que la respuesta es que (para mí) la mayor parte del beneficio de las pruebas proviene de pensar en las pruebas y su impacto en el código. Y, después de hacerlo durante tanto tiempo, podía hacerlo pensando sin tener que escribir pruebas. Mi código aún era comprobable; simplemente no fue probado.

Pero eso ignora el hecho de que las pruebas también son una forma de comunicarse con otros desarrolladores, por lo que ahora escribo pruebas en código compartido con otros o que depende de las peculiaridades de las dependencias externas.

Andy dice que no debería incluir esta barra lateral. Le preocupa que los desarrolladores inexpertos se sientan tentados a no probar. Aquí está mi compromiso:

¿Deberías escribir pruebas? Sí. Pero después de haberlo estado haciendo durante 30 años, no dude en experimentar un poco para ver dónde se encuentra el beneficio para usted.

UNA CULTURA DE PRUEBA

Todo el software que escriba será 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.

Realmente solo tienes unas pocas opciones:

- prueba primero
- prueba durante
- Probar nunca

Test First, incluido Test-Driven Design, es probablemente su mejor opción en la mayoría de las circunstancias, ya que garantiza que se realicen las pruebas. Pero a veces eso no es tan conveniente o útil, por lo que la prueba durante la codificación puede ser una buena alternativa, donde escribe algo de código, juega con él, escribe las pruebas para él y luego pasa a la siguiente parte. La peor opción a menudo se llama "Prueba más tarde", pero ¿a quién engañas? "Prueba más tarde" realmente significa "Prueba nunca".

Una cultura de prueba significa que todas las pruebas pasan todo el tiempo. Ignorar una serie de pruebas que "siempre fallan" hace que sea más fácil ignorar todas las pruebas y comienza la espiral viciosa (consulte el Tema 3, Entropía del software).

Trate el código de prueba con el mismo cuidado que cualquier código de producción. Manténgalo desacoplado, limpio y robusto. No confíe en cosas poco confiables (consulte el Tema 38, Programación por coincidencia) como la posición absoluta de los widgets en un sistema GUI, las marcas de tiempo exactas en un registro del servidor o la redacción exacta de los mensajes de error. Las pruebas para este tipo de cosas darán como resultado pruebas frágiles.

Consejo 70

Pruebe su software, o sus usuarios lo harán

No se equivoque, las pruebas son parte de la programación. No es algo que se deje a otros departamentos o al personal.

Pruebas, diseño, codificación: todo es programación.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 27, No superes tus faros
- Tema 51, Kit de inicio pragmático



Tema 42

Pruebas basadas en propiedades

Доверяй, но проверяй
(Confiar pero verificar)

proverbio ruso

Recomendamos escribir pruebas unitarias para sus funciones. Lo hace pensando en cosas típicas que podrían ser un problema, según su conocimiento de lo que está probando.

Sin embargo, hay un pequeño pero potencialmente significativo problema al acecho en ese párrafo. Si escribe el código original y escribe las pruebas, ¿es posible que se exprese una suposición incorrecta en ambos? El código pasa las pruebas, porque hace lo que se supone que debe hacer según su comprensión.

Una forma de evitar esto es que diferentes personas escriban las pruebas y el código bajo prueba, pero esto no nos gusta: como dijimos en el Tema 41, Test to Code, uno de los mayores beneficios de pensar en las pruebas es la forma en que informa el código que escribes. Pierde eso cuando el trabajo de prueba se separa de la codificación.

En cambio, preferimos una alternativa, donde la computadora, que no comparte sus ideas preconcebidas, hace algunas pruebas por usted.

CONTRATOS, INVARIANTES Y PROPIEDADES

En el Tema 23, Diseño por contrato, hablamos sobre la idea de que el código tiene contratos que cumple: usted cumple las condiciones cuando lo alimenta, y hará ciertas garantías sobre el

salidas que produce.

También hay invariantes de código, cosas que siguen siendo verdaderas sobre algún estado cuando se pasa a través de una función. Por ejemplo, si ordena una lista, el resultado tendrá la misma cantidad de elementos que el original; la longitud es invariable.

Una vez que resolvamos nuestros contratos e invariantes (que vamos a agrupar y llamar propiedades), podemos usarlos para automatizar nuestras pruebas. Lo que terminamos haciendo se llama prueba basada en propiedades.

Consejo 71

Utilice pruebas basadas en propiedades para validar su suposiciones

Como ejemplo artificial, podemos construir algunas pruebas para nuestra lista ordenada. Ya hemos establecido una propiedad: la lista ordenada tiene el mismo tamaño que la original. También podemos afirmar que ningún elemento del resultado puede ser mayor que el que le sigue.

Ahora podemos expresar eso en código. La mayoría de los lenguajes tienen algún tipo de marco de prueba basado en propiedades. Este ejemplo está en Python y usa la herramienta Hypothesis y pytest, pero los principios son bastante universales.

Aquí está la fuente completa de las pruebas:

proptest/ordenar.py

```
de la importación de hipótesis dada la  
importación de hipótesis.estrategias como algunas
```

```
@given(some.lists(some.integers())) def  
test_list_size_is_invariant_across_sorting(a_list):
```

```

longitud_original = len(una_lista)

una_lista.ordenar() afirmar len(una_lista) == longitud_original

@given(some.lists(some.text())) def
test_sorted_result_is_ordered(a_list): a_list.sort()
for i in
range(len(a_list) - 1):
    afirmar una_lista[i] <= una_lista[i + 1]

```

Esto es lo que sucede cuando lo ejecutamos:

```

$ pytest sort.py
=====
      inicia sesión de prueba
=====

...
complementos: hipótesis-4.14.0

ordenar.py .. [100%]

=====
      2 pasaron en 0.95 segundos
=====


```

No hay mucho drama allí. Pero, detrás de escena, Hypothesis ejecutó nuestras dos pruebas cien veces, pasando en una lista diferente cada vez. Las listas tendrán diferentes longitudes y tendrán diferentes contenidos. Es como si hubiéramos cocinado 200 pruebas individuales con 200 listas aleatorias.

GENERACIÓN DE DATOS DE PRUEBA

Como la mayoría de las bibliotecas de pruebas basadas en propiedades, Hypothesis le brinda un minilenguaje para describir los datos que debe generar. El lenguaje se basa en llamadas a funciones en el módulo `de hipótesis.estrategias`, al que llamamos `algunos`, solo porque se lee mejor.

Si escribimos:

```
@dato(algunos.enteros())
```

Nuestra función de prueba se ejecutaría varias veces. Cada vez, se pasaría un entero diferente. Si en cambio escribimos lo siguiente:

```
* @dato(algunos.enteros(min_value=5, max_value=10).mapa(lambda x: x * 2))
```

entonces obtendríamos los números pares entre 10 y 20.

También puede componer tipos, de modo que

```
@dato(algunas.listas(algunos.enteros(min_value=1), max_size=100))
```

serán listas de números naturales que tienen como máximo 100 elementos.

No se supone que este sea un tutorial sobre ningún marco en particular, por lo que omitiremos un montón de detalles interesantes y, en cambio, veremos un ejemplo del mundo real.

ENCONTRAR MALAS SUPOSICIONES

Estamos escribiendo un sistema simple de procesamiento de pedidos y control de existencias (porque siempre hay espacio para uno más). Modela los niveles de stock con un objeto [Almacén](#). Podemos consultar un almacén para ver si hay algo en stock, eliminar cosas del stock y obtener los niveles de stock actuales.

Aquí está el código:

```
proptest/stock.py
```

```
clase Almacén: def
    __init__(self, stock): self.stock
        = stock

    def en_existencias(self, item_name):
```

```

    return (item_name en self.stock) y (self.stock[item_name] > 0)

def tomar_de_stock(self, nombre_artículo, cantidad): if
    cantidad <= self.stock[nombre_artículo]:
        self.stock[nombre_artículo] -= cantidad
    else:
        aumentar Excepción("Sobrevendido {}".format(nombre_artículo))

def stock_count(self, item_name): return
    self.stock[item_name]

```

Escribimos una prueba de unidad básica, que pasa:

proptest/stock.py

```

def test_warehouse(): wh
    = Almacén({"zapatos": 10, "sombreros": 2, "paraguas": 0}) afirmar
    wh.in_stock("zapatos") afirmar
    wh.in_stock("sombreros")
    afirmar no qué .in_stock("paraguas")

    wh.take_from_stock("zapatos", 2)
    afirmar wh.in_stock("zapatos")

    wh.take_from_stock("sombreros", 2)
    afirmar no wh.in_stock("sombreros")

```

Luego escribimos una función que procesa una solicitud para ordenar artículos del almacén. Devuelve una tupla donde el primer elemento es "ok" o "no disponible", seguido del artículo y la cantidad solicitada. También escribimos algunas pruebas, y pasan:

proptest/stock.py

```

def pedido(almacén, artículo, cantidad): if
    almacén.en_existencias(article):
        almacén.tomar_de_existencias(article, cantidad)
        devolución ("ok", artículo, cantidad)
    else:
        devolución ("no disponible", artículo, cantidad)

```

proptest/stock.py

```

def test_order_in_stock():
    wh = Almacén({"zapatos": 10, "sombreros": 2, "paraguas": 0})
    estado, artículo, cantidad = pedido(wh,
                                         "sombreros", 1) afirmar
    estado == "ok" afirmar
    artículo == "sombreros"
    afirmar cantidad == 1 afirmar wh.stock_count("sombreros") == 1

def test_order_not_in_stock(): wh
    = Almacén({"zapatos": 10, "sombreros": 2, "paraguas": 0}) estado,
    artículo, cantidad = pedido(wh, "paraguas", 1) afirmar
    estado == "no disponible" afirmar
    elemento == "paraguas" afirmar
    cantidad == 1 afirmar
    wh.stock_count("paraguas") == 0

def test_order_unknown_item(): wh =
    Almacén({"zapatos": 10, "sombreros": 2, "paraguas": 0}) estado,
    artículo, cantidad = pedido(wh, "bagel", 1) afirmar
    estado == "no disponible" afirmar
    artículo == "bagel" afirmar
    cantidad == 1

```

En la superficie, todo se ve bien. Pero antes de enviar el código, agreguemos algunas pruebas de propiedades.

Una cosa que sabemos es que las acciones no pueden aparecer y desaparecer en nuestra transacción. Esto significa que si tomamos algunos artículos del almacén, el número que tomamos más el número que se encuentra actualmente en el almacén debe ser el mismo que el número originalmente en el almacén. En la siguiente prueba, ejecutamos nuestra prueba con el parámetro del artículo elegido al azar de `"sombrero"` o `"zapato"` y la cantidad elegida de 1 a 4:

proptest/stock.py

```

@given(item = some.sampled_from(["zapatos", "sombreros"]),
       cantidad = algunos.integers(min_value=1, max_value=4))

```

```
def test_stock_level_plus_quantity_equals_original_stock_level(artículo, cantidad):
    wh =
        Almacén({"zapatos": 10, "sombreros": 2, "paraguas": 0})
        initial_stock_level = wh.stock_count(artículo)
        (estado, artículo, cantidad) = pedido( wh, artículo, cantidad)
        if status == "ok":
            afirmar wh.stock_count(artículo) + cantidad == initial_stock_level
```

Vamos a ejecutarlo:

```
$ pytest stock.py
...
stock.py:72:
-----
stock.py:76: en test_stock_level_plus_quantity_equals_original_stock_level
    (estado, artículo, cantidad) = pedido (wh, artículo, cantidad)
stock.py:40: en orden
    warehouse.take_from_stock(artículo, cantidad)

-----
self = <stock.Objeto de almacén en 0x10cf97cf8>, item_name = cantidad de
'sombreros' = 3

    def tomar_de_stock(self, nombre_artículo, cantidad): if
        cantidad <= self.stock[nombre_artículo]:
            self.stock[nombre_artículo] -= cantidad
        else:
            > aumentar la excepción ("Sobrevendido {}". formato (nombre_artículo))
E      Excepción: Sombreros sobrevendidos

stock.py:16: Excepción
----- Hipótesis ----- Ejemplo de
falsificación:
    test_stock_level_plus_quantity_equals_original_stock_level( item='hats',
        cantidad=3)
```

Explotó en `warehouse.take_from_stock`: intentamos sacar tres sombreros del almacén, pero solo hay dos en stock.

Nuestras pruebas de propiedad encontraron una suposición errónea: nuestra función `en_stock` solo verifica que haya al menos uno del artículo dado en

existencias. En cambio, debemos asegurarnos de tener suficiente para completar el pedido:

proptest/stock1.py

```
def en_stock(self, item_name, cantidad):
» return (nombre_artículo en self.stock) y (self.stock[item_name] >=
cantidad)
```

Y también cambiamos la función [de orden](#) :

proptest/stock1.py

```
orden def (almacén, artículo, cantidad):
» if warehouse.in_stock(artículo, cantidad):
    almacén.take_from_stock(artículo, cantidad)
    devolución ( "ok", artículo,
cantidad) más: devolución ( "no disponible", artículo, cantidad)
```

Y ahora pasa nuestra prueba de propiedad.

LAS PRUEBAS BASADAS EN PROPIEDADES A MENUDO LO SORPRENDEN

En el ejemplo anterior, utilizamos una prueba basada en propiedades para comprobar que los niveles de existencias se ajustaron correctamente. La prueba encontró un error, pero no tenía nada que ver con el ajuste del nivel de existencias. En cambio, encontró un error en nuestra función [in_stock](#) .

Este es tanto el poder como la frustración de las pruebas basadas en propiedades. Es poderoso porque configura algunas reglas para generar entradas, configura algunas afirmaciones para validar la salida y luego simplemente deja que se rompa. Nunca se sabe muy bien lo que sucederá. La prueba puede pasar. Una afirmación puede fallar. O el código puede fallar totalmente porque no pudo manejar las entradas que se le dieron.

La frustración es que puede ser complicado determinar qué falló.

Nuestra sugerencia es que cuando falla una prueba basada en propiedades, averigüe qué parámetros estaba pasando a la función de prueba y luego use esos valores para crear una prueba unitaria independiente y regular. Esta prueba unitaria hace dos cosas por ti. En primer lugar, le permite concentrarse en el problema sin que el marco de prueba basado en propiedades realice todas las llamadas adicionales en su código. En segundo lugar, esa prueba unitaria actúa como una prueba de regresión. Debido a que las pruebas basadas en propiedades generan valores aleatorios que pasan a su prueba, no hay garantía de que se usarán los mismos valores la próxima vez que ejecute las pruebas. Tener una prueba unitaria que obligue a usar esos valores asegura que este error no se escape.

LAS PRUEBAS BASADAS EN PROPIEDADES TAMBIÉN AYUDAN A SU DISEÑO

Cuando hablamos de pruebas unitarias, dijimos que uno de los principales beneficios era la forma en que te hacía pensar en tu código: una prueba unitaria es el primer cliente de tu API.

Lo mismo ocurre con las pruebas basadas en propiedades, pero de una forma ligeramente diferente. Te hacen pensar en tu código en términos de invariantes y contratos; piensas en lo que no debe cambiar, y lo que debe ser verdad. Esta información adicional tiene un efecto mágico en su código, ya que elimina los casos extremos y resalta las funciones que dejan los datos en un estado inconsistente.

Creemos que las pruebas basadas en propiedades son complementarias a las pruebas unitarias: abordan diferentes preocupaciones y cada una aporta sus propios beneficios. Si no los estás usando actualmente, pruébalos.

LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 23, [Diseño por contrato](#)
- Tema 25, [Programación Asertiva](#)

- Tema 45, El pozo de requisitos

EJERCICIOS

Ejercicio 31 (respuesta posible)

Vuelva a mirar el ejemplo del almacén. ¿Hay otras propiedades que puedas probar?

Ejercicio 32 (respuesta posible)

Su empresa envía maquinaria. Cada máquina viene en una caja, y cada caja es rectangular. Las cajas varían en tamaño. Su trabajo es escribir un código para empacar tantas cajas como sea posible en una sola capa que quepa en el camión de reparto. La salida de su código es una lista de todas las cajas. Para cada caja, la lista proporciona la ubicación en el camión, junto con el ancho y la altura. ¿Qué propiedades de la salida podrían probarse?

RETOS

Piensa en el código en el que estás trabajando actualmente. ¿Qué son las propiedades: los contratos y las invariantes? ¿Puedes usar un marco de prueba basado en propiedades para verificar esto automáticamente?



Tema 43

Mantente a salvo ahí fuera

Buenas vallas hacen
buenos vecinos.

Robert Frost, reparando el muro

En la discusión de la primera edición sobre el acoplamiento de códigos, hicimos una declaración audaz e ingenua: "no necesitamos ser tan paranoicos como los espías o los disidentes". Estuvimos equivocados. De hecho, necesitas ser así de paranoico, todos los días.

Mientras escribimos esto, las noticias diarias están llenas de historias de filtraciones de datos devastadoras, sistemas secuestrados y fraude cibernético. Cientos de millones de registros robados a la vez, miles y miles de millones de dólares en pérdidas y reparación, y estas cifras aumentan rápidamente cada año. En la gran mayoría de los casos, no es porque los atacantes fueran terriblemente inteligentes, o incluso vagamente competentes.

Es porque los desarrolladores fueron descuidados.

EL OTRO 90%

Al codificar, puede pasar por varios ciclos de "¡funciona!" y "¿por qué no funciona?" con el ocasional "no hay forma de que eso haya sucedido..." Después de varias colinas y baches en esta subida cuesta arriba,^[62] es fácil decirse a sí mismo, "¡uf, todo funciona!" y proclamar el código hecho. Por supuesto, aún no está hecho. Has terminado el 90 %, pero ahora tienes que considerar el otro 90 %.

Lo siguiente que debe hacer es analizar el código en busca de formas en que puede salir mal y agregarlo a su conjunto de pruebas. Considerará cosas como pasar parámetros incorrectos, fugas o recursos no disponibles; esa clase de cosas.

En los buenos viejos tiempos, esta evaluación de errores internos puede haber sido suficiente. Pero hoy eso es solo el comienzo, porque además de los errores por causas internas, debe considerar cómo un actor externo podría arruinar deliberadamente el sistema.

Pero tal vez usted protesta, "Oh, a nadie le importará este código, no es importante, nadie sabe acerca de este servidor..." Hay un gran mundo ahí afuera, y la mayor parte está conectado. Ya sea que se trate de un niño aburrido en el otro lado del planeta, terrorismo patrocinado por el estado, bandas criminales, espionaje corporativo o incluso un ex vengativo, están ahí afuera y apuntan a ti. El tiempo de supervivencia de un sistema obsoleto y sin parches en la red abierta se mide en minutos, o incluso menos.

La seguridad a través de la oscuridad simplemente no funciona.

PRINCIPIOS BÁSICOS DE SEGURIDAD

Los programadores pragmáticos tienen una cantidad saludable de paranoia. Sabemos que tenemos fallas y limitaciones, y que los atacantes externos aprovecharán cualquier abertura que dejemos para comprometer nuestros sistemas.

Sus entornos particulares de desarrollo e implementación tendrán sus propias necesidades centradas en la seguridad, pero hay algunos principios básicos que siempre debe tener en cuenta:

1. Minimice el área de superficie de ataque
 2. Principio de privilegio mínimo 3.
- Valores predeterminados
- seguros 4. Cifrar datos
- confidenciales 5. Mantener actualizaciones de seguridad

Echemos un vistazo a cada uno de estos.

Minimizar el área de superficie de ataque

El área de superficie de ataque de un sistema es la suma de todos los puntos de acceso donde un atacante puede ingresar datos, extraer datos o invocar la ejecución de un servicio. Aquí están algunos ejemplos:

La complejidad del código conduce a vectores de ataque La complejidad del código hace que la superficie de ataque sea más grande, con más oportunidades de efectos secundarios imprevistos. Piense en el código complejo como si hiciera que el área de la superficie fuera más porosa y abierta a la infección. Una vez más, el código simple y más pequeño es mejor. Menos código significa menos errores, menos oportunidades para un agujero de seguridad paralizante. El código más simple, más ajustado y menos complejo es más fácil de razonar, más fácil de detectar posibles debilidades.

Los datos de entrada son un vector de ataque. Nunca confíe en los datos de una entidad externa, siempre límpielos antes de pasarlos a una base de datos, visualización u otro procesamiento. Algunos^[63] idiomas pueden ayudar con esto.

En Ruby, por ejemplo, las variables que contienen entradas externas están contaminadas, lo que limita las operaciones que se pueden realizar en ellas. Por ejemplo, este código aparentemente usa la utilidad `wc` para informar sobre la cantidad de caracteres en un archivo cuyo nombre se proporciona en tiempo de ejecución:

```
seguridad/taint.rb
""

pone "Ingrese un nombre de archivo para
contar: nombre
= obtiene sistema ("wc -c #{nombre}")"
```

Un usuario nefasto podría causar daños como este:

Ingrese un nombre de archivo para

contar: test.dat; rm -rf /

Sin embargo, establecer el nivel SAFE en 1 contaminará los datos externos, lo que significa que no se puede usar en contextos peligrosos:

seguridad/taint.rb

» \$SEGURO = 1

"
pone "Ingrese un nombre de archivo para
contar: nombre
= obtiene sistema ("wc -c #{nombre}")"

~~~ session \$ ruby taint.rb Ingrese un nombre de archivo para contar:  
test.dat; rm -rf /

code/safety/taint.rb:5:in system': Operación insegura - sistema

(SecurityError) from code/safety/taint.rb:5:in main' ~~~

Los servicios no autenticados son un vector de ataque

Por su propia naturaleza, cualquier usuario en cualquier parte del mundo puede llamar a servicios no autenticados, por lo que, salvo cualquier otro manejo o limitación, ha creado inmediatamente una oportunidad para un ataque de denegación de servicio como mínimo. Un buen número de filtraciones de datos altamente públicos recientemente fueron causadas por desarrolladores que colocaron accidentalmente datos en almacenes de datos no autenticados y de lectura pública en la nube.

Los servicios autenticados son un vector de ataque

Mantenga el número de usuarios autorizados en un mínimo absoluto. Elimine usuarios y servicios no utilizados, antiguos u obsoletos. Se ha descubierto que muchos dispositivos habilitados para la red contienen contraseñas predeterminadas simples o cuentas administrativas no utilizadas y desprotegidas. Si una cuenta con implementación

las credenciales están comprometidas, todo su producto está comprometido.

Los datos de salida son un vector de ataque Hay una historia (posiblemente apócrifa) sobre un sistema que informó debidamente el mensaje de error [Otro usuario usa la contraseña](#). No regales información. Asegúrate de que los datos que reportes sean apropiados para la autorización de ese usuario. Truncar u ofuscar información potencialmente riesgosa, como el Seguro Social u otros números de identificación del gobierno.

La información de depuración es un vector de ataque. No hay nada más conmovedor que ver un seguimiento completo de la pila con datos en su cajero automático local, un quiosco de aeropuerto o una página web bloqueada. La información diseñada para facilitar la depuración también puede facilitar la introducción. Asegúrese de que cualquier "ventana de prueba" (discutida [aquí](#)) y el informe de excepción de tiempo de ejecución estén protegidos contra el espionaje [64] ojos.<sup>..</sup>

Consejo 72

### Manténgalo simple y minimice las superficies de ataque

#### Principio de privilegios mínimos

Otro principio clave es utilizar la cantidad mínima de privilegios durante el menor tiempo posible. En otras palabras, no obtenga automáticamente el nivel de permiso más alto, como [raíz](#) o [administrador](#). Si se necesita ese alto nivel, tómelo, haga la cantidad mínima de trabajo y renuncie a su permiso rápidamente para reducir el riesgo. Este principio se remonta a principios de la década de 1970:

Cada programa y cada usuario privilegiado del sistema debe operar usando la menor cantidad de privilegios necesarios para completar el trabajo.  
—Jerome Saltzer, Communications of the ACM, 1974.

Tome el programa [de inicio de sesión](#) en sistemas derivados de Unix.

Inicialmente se ejecuta con privilegios de root. Sin embargo, tan pronto como termina de autenticar al usuario correcto, deja caer el privilegio de alto nivel al del usuario.

Esto no solo se aplica a los niveles de privilegio del sistema operativo. ¿Su aplicación implementa diferentes niveles de acceso? ¿Es una herramienta contundente, como "administrador" frente a "usuario"? Si es así, considere algo más detallado, donde sus recursos confidenciales se dividen en diferentes categorías y los usuarios individuales tienen permisos solo para algunas de esas categorías.

Esta técnica sigue el mismo tipo de idea que minimizar el área de superficie: reducir el alcance de los vectores de ataque, tanto por tiempo como por nivel de privilegio. En este caso, menos es más.

#### Valores predeterminados seguros

La configuración predeterminada en su aplicación, o para sus usuarios en su sitio, debe ser la más segura. Es posible que estos no sean los valores más fáciles de usar o convenientes, pero es mejor dejar que cada individuo decida por sí mismo las compensaciones entre seguridad y conveniencia.

Por ejemplo, el valor predeterminado para la entrada de contraseña podría ser ocultar la contraseña tal como se ingresó, reemplazando cada carácter con un asterisco. Si está ingresando una contraseña en un lugar público lleno de gente, o proyectada ante una gran audiencia, es un valor predeterminado sensato. Pero algunos usuarios pueden querer ver la contraseña escrita

fueras, tal vez por accesibilidad. Si hay poco riesgo de que alguien esté mirando por encima del hombro, esa es una opción razonable para ellos.

### Cifrar datos confidenciales

No deje información de identificación personal, datos financieros, contraseñas u otras credenciales en texto sin formato, ya sea en una base de datos o en algún otro archivo externo. Si los datos quedan expuestos, el cifrado ofrece un nivel adicional de seguridad.

En el Tema 19, [Control de versiones](#), recomendamos poner todo lo necesario para el proyecto bajo control de versiones. Bueno, casi todo. Aquí hay una gran excepción a esa regla:

No registre secretos, claves API, claves SSH, contraseñas de cifrado u otras credenciales junto con su código fuente en el control de versiones.

Las claves y los secretos deben administrarse por separado, generalmente a través de archivos de configuración o variables de entorno como parte de la compilación y la implementación.

#### Antipatrones de contraseña

Uno de los problemas fundamentales de la seguridad es que, a menudo, la buena seguridad va en contra del sentido común o la práctica común. Por ejemplo, podría pensar que los requisitos estrictos de contraseña aumentarían la seguridad de su aplicación o sitio. Serías equivocado.

Las políticas de contraseña estrictas en realidad reducirán su seguridad. Aquí hay una breve lista de muy malas [65] ideas, junto con algunas recomendaciones del NIST:

- No restrinja la longitud de la contraseña a menos de 64 caracteres. NIST recomienda 256 como una buena longitud máxima.
- No trunque la contraseña elegida por el usuario.
- No restrinja caracteres especiales como `!();&%$#` o `/`. Consulte la nota sobre Bobby Tables anteriormente en esta sección. Si los caracteres especiales en su contraseña comprometen su sistema, tiene problemas mayores. El NIST dice que acepte todos los caracteres ASCII de impresión, espacios y Unicode.

- No proporcione pistas de contraseña a usuarios no autenticados, ni solicite tipos específicos de información (p. ej., "¿cuál era el nombre de su primera mascota?").
- No deshabilite la función [de pegar](#) en el navegador. Paralizar la funcionalidad del navegador y los administradores de contraseñas no hace que su sistema sea más seguro, de hecho, impulsa a los usuarios a crear contraseñas más simples y cortas que son mucho más fáciles de comprometer. Tanto el NIST en los EE. UU. como el Centro Nacional de Seguridad Cibernética en el Reino Unido requieren verificadores específicamente para permitir la funcionalidad de pegado por este motivo.
- No imponer otras reglas de composición. Por ejemplo, no exija ninguna combinación particular de mayúsculas y minúsculas, números o caracteres especiales, ni prohíba la repetición de caracteres, etc.
- No solicite arbitrariamente a los usuarios que cambien sus contraseñas después de un período de tiempo. Solo haga esto por una razón válida (por ejemplo, si ha habido una infracción).

Desea fomentar contraseñas largas y aleatorias con un alto grado de entropía. Poner restricciones artificiales limita la entropía y fomenta malos hábitos de contraseña, dejando su las cuentas de los usuarios son vulnerables a la toma de control.

## Mantener actualizaciones

de seguridad Actualizar los sistemas informáticos puede ser un gran dolor. Necesita ese parche de seguridad, pero como efecto secundario, rompe una parte de su aplicación. Puede decidir esperar y posponer la actualización hasta más tarde. Esa es una idea terrible, porque ahora su sistema es vulnerable a un exploit conocido.

Consejo 73

### Aplicar parches de seguridad rápidamente

Este consejo afecta a todos los dispositivos conectados a la red, incluidos teléfonos, automóviles, electrodomésticos, computadoras portátiles personales, máquinas de desarrollo, máquinas de construcción, servidores de producción e imágenes en la nube. Todo. Y si cree que esto realmente no importa, solo recuerde que las filtraciones de datos más grandes en la historia (hasta ahora) fueron causadas por sistemas que estaban atrasados en sus actualizaciones.

No dejes que te pase a ti.

## SENTIDO COMÚN VS. CRIPTO

Es importante tener en cuenta que el sentido común puede fallar cuando se trata de cuestiones de criptografía. La primera y más importante regla cuando se trata de criptografía es nunca hacerlo usted mismo.

[66] Incluso para algo tan simple como contraseñas, común

Las prácticas están equivocadas (ver la barra lateral [Contraseña Antipatrones](#)). Una vez que ingrese al mundo de las criptomonedas, incluso el error más pequeño y aparentemente insignificante puede comprometerlo todo: su nuevo e inteligente algoritmo de encriptación hecho en casa probablemente pueda ser descifrado por un experto en minutos. No desea hacer el cifrado usted mismo.

Como hemos dicho en otra parte, confíe solo en cosas confiables: bibliotecas y marcos de trabajo de código abierto, bien examinados, bien examinados, bien mantenidos, actualizados con frecuencia.

Más allá de las simples tareas de encriptación, analice detenidamente otras características relacionadas con la seguridad de su sitio o aplicación. Tome la autenticación, por ejemplo.

Para implementar su propio inicio de sesión con contraseña o autenticación biométrica, debe comprender cómo funcionan los hashes y las sales, cómo los crackers usan cosas como las tablas Rainbow, por qué no debe usar MD5 o SHA1 y muchas otras inquietudes. E incluso si lo hace todo bien, al final del día sigue siendo responsable de conservar los datos y mantenerlos seguros, sujeto a cualquier nueva legislación y obligaciones legales que surjan.

O bien, podría adoptar el enfoque pragmático y dejar que otra persona se preocupe por ello y utilizar un proveedor de autenticación de terceros.

Este puede ser un servicio listo para usar que ejecuta internamente, o podría ser un tercero en la nube. Los servicios de autenticación son

a menudo disponibles a través de proveedores de correo electrónico, teléfono o redes sociales, que pueden o no ser apropiados para su aplicación. En cualquier caso, estas personas pasan todos sus días manteniendo sus sistemas seguros y son mejores que usted.

Mantente a salvo ahí fuera.

#### LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 23, Diseño por contrato
- Tema 24, Los programas muertos no dicen mentiras
- Tema 25, Programación Asertiva
- Tema 38, Programación por Coincidencia
- Tema 45, El pozo de requisitos



Tema 44

nombrar cosas

El principio de la sabiduría es llamar

las cosas por su propio nombre.

Confucio

¿Lo que hay en un nombre? Cuando estamos programando, la respuesta es "¡todo!"

Creamos nombres para aplicaciones, subsistemas, módulos, funciones, variables; estamos constantemente creando cosas nuevas y otorgándoles nombres. Y esos nombres son muy, muy importantes, porque revelan mucho sobre

tu intención y creencia.

Creemos que las cosas deben nombrarse de acuerdo con el papel que desempeñan en su código. Esto significa que, cada vez que creas algo, debes hacer una pausa y pensar "¿cuál es mi motivación para crear esto?"

Esta es una pregunta poderosa, porque lo saca de la mentalidad inmediata de resolución de problemas y lo hace ver el panorama general. Cuando considera el rol de una variable o función, está pensando en qué tiene de especial, qué puede hacer y con qué interactúa. A menudo, nos damos cuenta de que lo que estabamos a punto de hacer no tenía sentido, todo porque no pudimos encontrar una forma adecuada.

nombre.

Hay algo de ciencia detrás de la idea de que los nombres son profundamente

significativo. Resulta que el cerebro puede leer y comprender palabras muy rápido: más rápido que muchas otras actividades. Esto significa que las palabras tienen cierta prioridad cuando tratamos de dar sentido a algo. Esto se puede demostrar usando el efecto Stroop.

[67]

Mira el siguiente panel. Tiene una lista de nombres de colores o tonos, y cada uno se muestra en un color o tono. Pero los nombres y colores no necesariamente coinciden. Aquí está la primera parte de la desafío: diga en voz alta el nombre de cada color tal como está escrito:<sup>[68]</sup>.

imágenes/stroop\_color.png

Ahora repita esto, pero diga en voz alta el color usado para dibujar la palabra. Más difícil, ¿eh? Es fácil tener fluidez al leer, pero es mucho más difícil tratar de reconocer los colores.

Tu cerebro trata las palabras escritas como algo que debe ser respetado. Necesitamos asegurarnos de que los nombres que usamos estén a la altura de esto.

Veamos un par de ejemplos:

- Estamos autenticando a las personas que acceden a nuestro sitio que vende joyas hechas con tarjetas gráficas antiguas:

dejar usuario = autenticar (credenciales)

La variable es `usuario` porque siempre es `usuario`. ¿Pero por qué? Significa

nada. ¿Qué tal el cliente o el comprador? De esa manera, recibimos recordatorios constantes a medida que codificamos lo que esta persona está tratando de hacer y lo que eso significa para nosotros.

- Tenemos un método de instancia que descuenta un pedido:

```
porcentaje de deducción de anulación pública ( cantidad doble)
// ...
```

Dos cosas aquí. Primero, `deductPercent` es lo que hace y no por qué lo hace. Entonces, el nombre de la `cantidad` del parámetro es, en el mejor de los casos, engañoso: ¿es una cantidad absoluta, un porcentaje?

Tal vez esto sería mejor:

```
public void applyDiscount(Porcentaje de descuento) //
...
```

El nombre del método ahora aclara su intención. También hemos cambiado el parámetro de un `doble` a un `Porcentaje`, un tipo que hemos definido.

No sabemos ustedes, pero cuando se trata de porcentajes, nunca sabemos si se supone que el valor debe estar entre 0 y 100 o entre 0,0 y 1,0. El uso de un tipo documenta lo que espera la función.

- Tenemos un módulo que hace cosas interesantes con los números de Fibonacci.

Una de esas cosas es calcular el número en la secuencia. Detente y piensa cómo llamarías a esta función.

La mayoría de las personas a las que preguntamos lo llamarían `mentira`. Parece razonable, pero recuerda que normalmente se llamará en el contexto de su módulo, por lo que la llamada sería `Fib.fib(n)`. ¿Qué tal llamarlo `of` o `nth` en su lugar?

```
Fib.de(0) # => 0
Fib.nth(20) # => 4181
```

Al nombrar cosas, busca constantemente formas de aclarar lo que quiere decir, y ese acto de clarificación lo llevará a una mejor comprensión de su código a medida que lo escribe.

Sin embargo, no todos los nombres tienen por qué ser candidatos a un premio literario.

**La excepción que confirma la regla**

Si bien nos esforzamos por la claridad en el código, la marca es un asunto completamente diferente.

Existe una tradición bien establecida de que los proyectos y los equipos de proyectos deben tener nombres oscuros e "inteligentes". Nombres de Pokémon, superhéroes de Marvel, lindos mamíferos, personajes de El Señor de los Anillos , lo que sea.

Literalmente.

## HONRAR LA CULTURA

La mayoría de los textos de introducción a la informática le advertirán que nunca utilice variables de una sola letra como i, j o k.<sup>[69]</sup>

Creemos que están equivocados. Algo así como.

De hecho, depende de la cultura de ese lenguaje de programación o entorno en particular. En el lenguaje de programación C, i, j y k se usan tradicionalmente como variables de incremento de bucle, s se usa para una cadena de caracteres, etc. Si programa en ese entorno, eso es lo que está acostumbrado a ver y sería discordante (y por lo tanto incorrecto) violar esa norma.

Por otro lado, usar esa convención en un entorno diferente donde no se espera es igual de incorrecto. Nunca harías algo atroz como este ejemplo de Clojure que asigna una cadena a la variable i:

```
(let [i "Hola Mundo"]
  (println i))
```

Algunas comunidades lingüísticas prefieren camelCase, con letras mayúsculas incrustadas, mientras que otras prefieren snake\_case con guiones bajos incrustados para separar palabras. Los propios idiomas, por supuesto, aceptarán cualquiera de los dos, pero eso no lo hace correcto. Honrar la cultura local.

Algunos idiomas permiten un subconjunto de Unicode en los nombres. Obtén una idea de lo que espera la comunidad antes de ponerte lindo con nombres como [sn](#) o [εξέρχεται](#).

## CONSISTENCIA

Emerson es famoso por escribir "Una consistencia tonta es el duende de las mentes pequeñas...", pero Emerson no estaba en un equipo de programadores.

Cada proyecto tiene su propio vocabulario: palabras de jerga que tienen un significado especial para el equipo. "Orden" significa una cosa para un equipo que crea una tienda en línea y algo muy diferente para un equipo cuya aplicación registra el linaje de los grupos religiosos. Es importante que todos en el equipo sepan lo que significan estas palabras y que las usen de manera constante.

Una forma es fomentar mucha comunicación. Si todos emparejan programas y los pares cambian con frecuencia, entonces la jerga se extenderá osmóticamente.

Otra forma es tener un glosario del proyecto, enumerando los términos que tienen un significado especial para el equipo. Este es un documento informal, posiblemente mantenido en un wiki, posiblemente solo fichas en una pared en algún lugar.

Después de un tiempo, la jerga del proyecto cobrará vida propia. A medida que todos se sientan cómodos con el vocabulario, podrá usar la jerga como abreviatura, expresando muchos significados de manera precisa y concisa. (Esto es exactamente lo que es un lenguaje de patrones ).

## CAMBIAR EL NOMBRE ES AÚN MÁS DIFÍCIL

No importa cuánto esfuerzo pongas al principio, las cosas cambian. El código se refactoriza, el uso cambia, el significado se altera sutilmente. Si no está atento a actualizar los nombres sobre la marcha, puede caer rápidamente en una pesadilla mucho peor que los nombres sin sentido: nombres engañosos . ¿Alguna vez alguien le ha explicado las inconsistencias en el código, como "La rutina llamada `getData` realmente escribe datos en un archivo de almacenamiento"?

Como discutimos en el Tema 3, Software Entropy, cuando detecte un problema, arréglelo, aquí y ahora. Cuando vea un nombre que ya no expresa la intención, o es engañoso o confuso, arréglelo. Tiene pruebas de regresión completas, por lo que detectará cualquier instancia que se haya perdido.

Sugerencia      Nombra bien; Renombrar cuando sea necesario

Si por alguna razón no puede cambiar el nombre ahora incorrecto, entonces tiene un problema mayor: una violación de ETC (consulte el Tema 8, La esencia del buen diseño). Arregle eso primero, luego cambie el nombre ofensivo. Facilite el cambio de nombre y hágalo con frecuencia.

De lo contrario, tendrá que explicarle a la gente nueva del equipo que `getData` realmente escribe datos en un archivo, y tendrá que hacerlo con seriedad.

## LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 3, Entropía del software
- Tema 40, Refactorización
- Tema 45, El pozo de requisitos

## RETOS

- Cuando encuentre una función o método con un nombre demasiado genérico, intente cambiarle el nombre para expresar todo lo que realmente hace. Ahora es un objetivo más fácil para la refactorización.
- En nuestros ejemplos, sugerimos usar nombres más específicos como comprador en lugar del usuario más tradicional y genérico. ¿Qué otros nombres utilizas habitualmente que podrían ser mejores?
- ¿Los nombres en su sistema son congruentes con los términos de usuario del dominio? Si no, ¿por qué? ¿Causa esto una disonancia cognitiva al estilo del efecto Stroop para el equipo?
- ¿Son difíciles de cambiar los nombres en su sistema? ¿Qué puedes hacer para arreglar esa ventana rota en particular?

---

notas al pie

[50] Nota de los veteranos: UTC está ahí por una razón. úsalos

[51] [https://en.wikipedia.org/wiki/Correlation\\_does\\_not\\_imply\\_causation](https://en.wikipedia.org/wiki/Correlation_does_not_imply_causation)

[52] Consulte el Tema 50, Los cocos no son suficientes.

[53] 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.

[54] [https://media-origin.pragprog.com/titles/tpp20/code/algorithm\\_speed/sort/src/main.rs](https://media-origin.pragprog.com/titles/tpp20/code/algorithm_speed/sort/src/main.rs)

[55] Y sí, expresamos nuestras preocupaciones sobre el título.

[56] Visto originalmente en UML Destilado: una breve guía para el lenguaje de modelado de objetos estándar [Fow00].

[57] Este es un excelente consejo en general (vea el Tema 27, No supere sus faros).

[58] Algunas personas argumentan que las pruebas primero y el desarrollo basado en pruebas son dos cosas diferentes, diciendo que las intenciones de los dos son diferentes. Sin embargo, históricamente, la prueba primero (que proviene de eXtreme Programming) era idéntica a lo que la gente ahora llama TDD.

[59] <https://ronjeffries.com/categories/sudoku>. Un gran “gracias” a Ron por dejarnos usar esta historia.

[60] <http://norvig.com/sudoku.html>

Hemos estado intentándolo desde al menos 1986, cuando Cox y Novobilski acuñaron el término [61] "software IC" en su libro Objective-C Programación orientada a objetos Programación orientada a objetos: un enfoque evolutivo [CN91].

[62] Consulte el Tema 20, Depuración.

[63] ¿Recuerdas a nuestro buen amigo, el pequeño Bobby Tables (<https://xkcd.com/327>)? Mientras recuerda, eche un vistazo a <https://bobby-tables.com>, que enumera las formas de desinfectar los datos pasados a las consultas de la base de datos.

[64] Esta técnica ha demostrado ser exitosa a nivel de chip de CPU, donde los exploits bien conocidos apuntan a las instalaciones administrativas y de depuración. Una vez agrietada, toda la máquina queda expuesta.

Presenta y crea Publicación especial de NIST 800-63B: Directrices de identidad digital: Autenticación y gestión del ciclo de vida, disponible gratis en línea en <https://doi.org/10.6028/NIST.SP.800-63b>

[66] A menos que tenga un doctorado en criptografía, e incluso entonces solo con una importante revisión por pares, extensas pruebas de campo con una recompensa por errores y presupuesto para mantenimiento a largo plazo.

[67] Estudios de interferencia en reacciones verbales en serie [Str35]

[68] Tenemos dos versiones de este panel. Uno usa diferentes colores y el otro usa tonos de gris. Si ve esto en blanco y negro y desea la versión en color, o si tiene problemas para distinguir los colores y desea probar la versión en escala de grises, acceda a <https://pragprog.com/the-pragmatic-programmer/> efecto stroop.

[69] ¿Sabes por qué **i** se usa comúnmente como una variable de bucle? La respuesta proviene de hace más de 60 años, cuando las variables que comenzaban con **I** a **N** eran números enteros en el FORTRAN original. Y FORTRAN a su vez fue influenciado por el álgebra.

## Capítulo 8

# Antes del Proyecto

Al comienzo de un proyecto, usted y el equipo deben conocer los requisitos. Simplemente que se les diga qué hacer o escuchar a los usuarios no es suficiente: lea el Tema 45, El foso de los requisitos y aprenda cómo evitar las trampas y escollos comunes.

La sabiduría convencional y la gestión de restricciones son los temas del Tema 46, Resolviendo acertijos 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.

Y cuando surge ese proyecto imposible, nos gusta recurrir a nuestra arma secreta: Tema 47, Trabajando juntos. Y por "trabajar juntos" no nos referimos a compartir un documento de requisitos masivo, enviar correos electrónicos con muchas copias o soportar reuniones interminables. Nos referimos a resolver problemas juntos mientras codifican. Le mostraremos a quién necesita y cómo empezar.

Aunque el Manifiesto Ágil comienza con "Individuos e interacciones sobre procesos y herramientas", prácticamente todos los proyectos "ágiles" comienzan con una discusión irónica sobre qué proceso y qué herramientas usarán. Pero por muy bien pensado que esté,

e independientemente de las "mejores prácticas" que incluya, ningún método puede reemplazar el pensamiento. No necesita ningún proceso o herramienta en particular, lo que sí necesita es el Tema 48, La esencia de la agilidad.

Con estos problemas críticos resueltos antes de que el proyecto comience, puede estar mejor posicionado para evitar la "parálisis de análisis" y comenzar, y completar, su proyecto exitoso.



Tema 45

## El pozo de requisitos

La perfección se logra, no cuando hay  
no queda nada más que añadir, pero cuando hay  
no queda nada para tomar  
lejos...

Antoine de St. Exupery, Viento,  
arena y estrellas, 1939

alegriamente en tu camino.

Muchos libros y tutoriales se refieren a la 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 cesta y estar listo.

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. Peor aún, a menudo no existen en absoluto.

Consejo 75

Nadie sabe exactamente lo que quiere

## EL MITO DE LOS REQUISITOS

En los primeros días del software, las computadoras eran más valiosas (en términos de costo amortizado por hora) que las personas que trabajaban con ellas. Ahorramos dinero tratando de corregir las cosas

primera vez. Parte de ese proceso fue tratar de especificar exactamente qué íbamos a hacer que hiciera la máquina. Empezaríamos por obtener una especificación de los requisitos, convertirla en un documento de diseño, luego en diagramas de flujo y pseudocódigo, y finalmente en código. Sin embargo, antes de pasarlo a una computadora, dedicábamos tiempo a revisarlo en el escritorio.

Costó mucho dinero. Y ese costo significaba que las personas solo intentaban automatizar algo cuando sabían exactamente lo que querían. Como las primeras máquinas eran bastante limitadas, el alcance de los problemas que resolvían estaba restringido: en realidad era posible comprender todo el problema antes de comenzar.

Pero ese no es el mundo real. El mundo real es desordenado, conflictivo y desconocido. En ese mundo, las especificaciones exactas de cualquier cosa son raras, si no completamente imposibles.

Ahí es donde entramos los programadores. Nuestro trabajo es ayudar a las personas a entender lo que quieren. De hecho, ese es probablemente nuestro atributo más valioso. Y vale la pena repetir:

Consejo 76

Los programadores ayudan a las personas a entender lo que Desear

## LA PROGRAMACIÓN COMO TERAPIA

Llamemos a las personas que nos piden que escribamos software nuestros clientes.

El cliente típico viene a nosotros con una necesidad. La necesidad puede ser estratégica, pero es igual de probable que sea una cuestión táctica: una respuesta a un problema actual. La necesidad puede ser un cambio en un sistema existente o puede pedir algo nuevo. A veces, la necesidad se expresará en términos comerciales y, a veces, en

los técnicos.

El error que suelen cometer los nuevos desarrolladores es tomar esta declaración de necesidad e implementar una solución para ella.

En nuestra experiencia, esta declaración inicial de necesidad no es un requisito absoluto. El cliente puede no darse cuenta de esto, pero en realidad es una invitación a explorar.

Tomemos un ejemplo simple.

Trabajas para una editorial de libros en papel y electrónicos. Se le da un nuevo requisito:

El envío debe ser gratuito en todos los pedidos que cuesten \$50 o más.

Deténgase por un segundo e imagínese en esa posición. ¿Qué es lo primero que te viene a la mente?

Hay muchas posibilidades de que tenga preguntas:

- ¿Los \$50 incluyen impuestos?
- ¿Los \$50 incluyen los gastos de envío actuales?
- ¿Los \$50 tienen que ser para libros impresos o el pedido también puede incluir libros electrónicos?
- ¿Qué tipo de envío se ofrece? ¿Prioridad? ¿Suelo?
- ¿Qué pasa con los pedidos internacionales?
- ¿Con qué frecuencia cambiará el límite de \$50 en el futuro?

Éso es lo que hacemos. Cuando se nos da algo que parece simple, molestamos a la gente buscando casos extremos y preguntando sobre

a ellos.

Lo más probable es que el cliente ya haya pensado en algunos de estos, y simplemente asumió que la implementación funcionaría de esa manera.

Hacer la pregunta simplemente elimina esa información.

Pero otras preguntas probablemente serán cosas que el cliente no había considerado previamente. Ahí es donde las cosas se ponen interesantes y donde un buen desarrollador aprende a ser diplomático.

Tú:

Nos preguntábamos sobre el total de \$50. ¿Eso incluye lo que normalmente cobraríamos por el envío?

Cliente:

Por supuesto. Es el total que nos pagarían.

Tú:

Eso es bueno y simple para que nuestros clientes lo entiendan: puedo ver la atracción. Pero puedo ver a algunos clientes menos escrupulosos tratando de jugar con ese sistema.

Cliente:

¿Cómo es eso?

Tú:

Bueno, digamos que compran un libro por \$25 y luego seleccionan el envío al día siguiente, la opción más cara. Es probable que sea alrededor de \$ 30, lo que hace que el pedido total sea de \$ 55. Luego haríamos que el envío fuera gratis, y ellos obtendrían el envío al día siguiente de un libro de \$25 por solo \$25.

(En este punto, el desarrollador experimentado se detiene. Entregar

hechos y dejar que el cliente tome las decisiones)

Cliente:

Ay. Eso ciertamente no era lo que pretendía; perderíamos dinero en esos pedidos. ¿Cuales son las opciones?

Y esto inicia una exploración. Su papel en esto es interpretar lo que dice el cliente y retroalimentarlo con las implicaciones.

Este es un proceso tanto intelectual como creativo: usted está pensando sobre la marcha y está contribuyendo a una solución que probablemente sea mejor que la que usted o el cliente habrían producido solos.

#### LOS REQUISITOS SON UN PROCESO

En el ejemplo anterior, el desarrollador tomó los requisitos y retroalimentó una consecuencia al cliente. Esto inició la exploración. Durante esa exploración, es probable que obtenga más comentarios a medida que el cliente juegue con diferentes soluciones.

Esta es la realidad de la recopilación de todos los requisitos:

Consejo 77

Los requisitos se aprenden en un circuito de retroalimentación

Su trabajo es ayudar al cliente a comprender las consecuencias de sus requisitos establecidos. Lo hace generando retroalimentación y permitiéndoles usar esa retroalimentación para refinar su pensamiento.

En el ejemplo anterior, la retroalimentación fue fácil de expresar en palabras. A veces ese no es el caso. Y, a veces, honestamente, no sabrá lo suficiente sobre el dominio para ser tan específico.

En esos casos, los programadores pragmáticos confían en el "¿es esto lo que

"¿te referías?" escuela de retroalimentación. Producimos maquetas y prototipos, y dejamos que el cliente juegue con ellos. Idealmente, las cosas que producimos son lo suficientemente flexibles como para que podamos cambiarlas durante nuestras conversaciones con el cliente, permitiéndonos responder a "eso no es lo que quise decir" con "¿así que más como esto?"

A veces, estas maquetas se pueden armar en una hora más o menos. Obviamente, son solo trucos para transmitir una idea.

Pero la realidad es que todo el trabajo que hacemos es en realidad una especie de maqueta. Incluso al final de un proyecto todavía estamos interpretando lo que quiere nuestro cliente. De hecho, en ese momento es probable que tengamos más clientes: el personal de control de calidad, operaciones, marketing y quizás incluso grupos de prueba de clientes.

Entonces, el programador pragmático considera todo el proyecto como un ejercicio de recopilación de requisitos. Es por eso que preferimos iteraciones cortas; los que terminan con comentarios directos del cliente. Esto nos mantiene en el buen camino y asegura que, si vamos en la dirección equivocada, se minimice la cantidad de tiempo perdido.

## PONTE EN LOS ZAPATOS DE TU CLIENTE

Hay una técnica simple para meterse en la cabeza de sus clientes que no se usa con la suficiente frecuencia: conviértase en un cliente. ¿Está escribiendo un sistema para la mesa de ayuda? Pase un par de días monitoreando los teléfonos con una persona de apoyo experimentada. ¿Está automatizando un sistema de control de stock manual? Trabaja en el almacén [70] durante una semana.

Además de brindarle una idea de cómo se usará realmente el sistema, se sorprenderá de cómo la solicitud "¿Puedo sentarme una semana mientras hace su trabajo?" ayuda a generar confianza y establece una base para la comunicación con tus clientes solo recuerda no

para meterse en el camino!

### Consejo 7 Trabaje con un usuario para pensar como un usuario

La recopilación de comentarios también es el momento de comenzar a construir una relación con su base de clientes, conociendo sus expectativas y esperanzas para el sistema que está construyendo. Consulte el Tema 52, [Deleite a sus usuarios](#), para obtener más información.

## REQUISITOS VS. POLÍTICA

Imaginemos que mientras habla sobre un sistema de recursos humanos, un cliente dice: "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.

¿Política de la empresa? ¿Requisito? Es una distinción relativamente sutil, pero tendrá profundas implicaciones para los desarrolladores. Si el requisito se establece como "Solo los supervisores y el personal pueden ver un registro de empleado", el desarrollador puede terminar codificando una prueba explícita cada vez que la aplicación accede a estos datos. 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.

De hecho, hay una regla general aquí:

Consejo 79

La política es metadatos

Implemente el caso general, con la información de la política como un ejemplo del tipo de cosas que el sistema necesita soportar.

## REQUISITOS VS. REALIDAD

En un artículo de la revista *Wired* de enero de 1999<sup>[71]</sup>, 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.

Este ejemplo también ilustra nuestra creencia de que las herramientas exitosas se adaptan a las manos que las usan. La recopilación exitosa de requisitos tiene esto en cuenta. Y es por eso que la retroalimentación temprana, con prototipos o viñetas, permitirá que sus clientes digan “sí, hace lo que quiero, pero no como quiero”.

## REQUISITOS DE DOCUMENTACIÓN

Creemos que la mejor documentación de requisitos, quizás

la única documentación de requisitos es el código de trabajo.

Pero eso no significa que pueda salirse con la suya sin documentar su comprensión de lo que quiere el cliente. Simplemente significa que esos documentos no son entregables: no son algo que le das a un cliente para que los firme. En cambio, son simplemente hitos para ayudar a guiar la implementación. proceso.

### Los documentos de requisitos no son para

clientes En el pasado, tanto Andy como Dave han estado en proyectos que produjeron requisitos increíblemente detallados. Estos documentos sustanciales ampliaron la explicación inicial de dos minutos del cliente sobre lo que quería, produciendo obras maestras de una pulgada de grosor llenas de diagramas y tablas. Las cosas se especificaron hasta el punto en que casi no había lugar para la ambigüedad en la implementación. Dadas las herramientas suficientemente poderosas, el documento podría ser el programa final.

Crear estos documentos fue un error por dos razones. Primero, como hemos discutido, el cliente realmente no sabe lo que quiere por adelantado. Entonces, cuando tomamos lo que dicen y lo expandimos a lo que es casi un documento legal, estamos construyendo un castillo increíblemente complejo sobre arenas movedizas.

Podría decir “pero luego le llevamos el documento al cliente y lo firman. Estamos recibiendo comentarios”. Y eso nos lleva al segundo problema con estas especificaciones de requisitos: el cliente nunca las lee.

El cliente utiliza programadores porque, mientras que el cliente está motivado por resolver un problema de alto nivel y algo nebuloso, los programadores están interesados en todos los detalles y

máticas El documento de requisitos está escrito para desarrolladores y contiene información y sutilezas que a veces son incomprensibles y frecuentemente aburridas para el cliente.

Envíe un documento de requisitos de 200 páginas, y es probable que el cliente lo sopesa para decidir si pesa lo suficiente como para ser importante, puede leer los primeros párrafos (razón por la cual los dos primeros párrafos siempre se titulan Resumen de gestión ) y puede hojear el resto, a veces deteniéndose cuando hay un diagrama claro.

Esto no es menospreciar al cliente. Pero darles un gran documento técnico es como darle al desarrollador promedio una copia de la Ilíada en griego homérico y pedirle que codifique el videojuego a partir de él.

**Los documentos de requisitos son para la planificación** Por lo tanto, no creemos en el documento de requisitos monolítico, lo suficientemente pesado como para aturdir a un buey. Sin embargo, sabemos que los requisitos deben escribirse, simplemente porque los desarrolladores de un equipo necesitan saber lo que harán.

¿Qué forma toma esto? Preferimos algo que pueda caber en una tarjeta de índice real (o virtual). Estas breves descripciones a menudo se denominan historias de usuario. Describen lo que debería hacer una pequeña parte de la aplicación desde la perspectiva de un usuario de esa funcionalidad.

Cuando se escriben de esta manera, los requisitos se pueden colocar en un tablero y moverse para mostrar tanto el estado como la prioridad.

Podría pensar que una sola ficha no puede contener la información necesaria para implementar un componente del

solicitud. Tendrías razón. Y eso es parte del punto. Al mantener breve esta declaración de requisitos, está alejando a los desarrolladores a hacer preguntas aclaratorias. Está mejorando el proceso de retroalimentación entre clientes y programadores antes y durante la creación de cada pieza de código.

## SOBREESPECIFICACIÓN

Otro gran peligro al producir un documento de requisitos es ser demasiado específico. Los buenos requisitos son 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 necesarios.

## SÓLO UNA MENTA MÁS DELGADA...

Muchas fallas de 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 del Tema 4, Sopa de piedra y ranas hervidas. ¿Qué podemos hacer para evitar que los requisitos se nos acerquen sigilosamente?

La respuesta (nuevamente) es la retroalimentación. Si está trabajando con el cliente en iteraciones con comentarios constantes, entonces el cliente experimentará de primera mano el impacto de "solo una característica más". Verán otra tarjeta de historia en el tablero y podrán ayudar a elegir otra tarjeta para pasar a la siguiente iteración para hacer espacio. La retroalimentación funciona en ambos sentidos.

## 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.

Cree y mantenga un glosario del proyecto, un lugar que define todos los términos y el vocabulario específicos que se usan 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 en línea.

Consejo 80

### Usar un glosario de proyectos

Es difícil tener éxito en un proyecto si los usuarios y los desarrolladores llaman a lo mismo por nombres diferentes o, peor aún, se refieren a cosas diferentes por el mismo nombre.

## LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 5, Software suficientemente bueno
- Tema 7, ¡Comuníquese!
- Tema 11, Reversibilidad
- Tema 13, Prototipos y Post-it Notes
- Tema 23, Diseño por contrato
- Tema 43, Manténgase seguro ahí afuera
- Tema 44, Nombrar cosas
- Tema 46, Resolviendo acertijos imposibles

- Tema 52, Deleite a sus usuarios

## EJERCICIOS

### Ejercicio 33 (respuesta posible)

¿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 ~500ms.
2. Las ventanas modales tendrán un fondo gris.
3. La aplicación se organizará como una serie de front-end procesos y un servidor back-end.
4. Si un usuario ingresa caracteres no numéricos en un campo numérico, el sistema parpadeará el fondo del campo y no los aceptará.
5. El código y los datos de esta aplicación integrada deben caber dentro 32Mb.

## RETOS

- ¿Puedes usar el software que estás escribiendo? ¿Es posible tener una buena idea de los requisitos sin poder usar el software usted mismo?
- Elija un problema no relacionado con la computadora que actualmente necesita resolver. Generar requerimientos para una solución no informática.



## Tema 46

### Resolviendo rompecabezas imposibles

Gordio, el rey de Frigia, una vez ató un nudo que nadie pudo desatar. Se dijo que quien resolvió el El enigma del Nudo Gordiano goberaría toda Asia. Entonces llega Alejandro Magno, que corta el nudo en pedazos con su espada.

Solo una pequeña interpretación diferente de los requisitos, eso es todo... Y terminó dominando 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 ser mucho más difícil de escribir de lo que pensabas. Tal vez parezca imposible. Pero, ¿realmente es tan difícil como parece?

Considere 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 las ventas de garaje. Todo lo que tienes que hacer es quitar el anillo, o encajar las piezas en forma de T en la caja, o lo que sea.

Así que tiras del anillo, o intentas poner las T en la caja, y rápidamente descubres que las soluciones obvias simplemente no funcionan. El rompecabezas no se puede resolver de esa manera. Pero aunque es obvio, eso no impide que la gente intente lo mismo, una y otra vez, pensando que debe haber una manera.

Por supuesto, 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 absolutas; otros son meras nociónes preconcebidas. Deben respetarse las restricciones absolutas , por desagradables o estúpidas que parezcan.

Por otro lado, como demostró Alexander, algunas restricciones aparentes pueden no ser restricciones reales en absoluto. Muchos problemas de software pueden ser igual de engañosos.

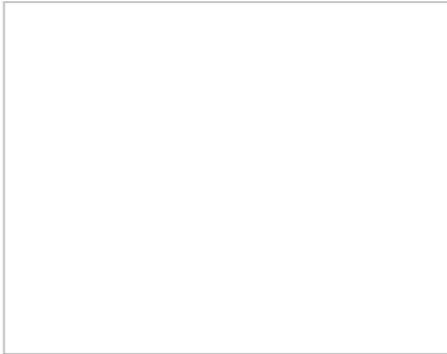
### GRADOS DE LIBERTAD

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 es encontrar la caja, que puede ser considerablemente más grande de lo que cree.

La clave para resolver acertijos es tanto reconocer las limitaciones que se te imponen como reconocer los grados de libertad que tienes , porque en ellos encontrarás la 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 acertijo y volver al punto de partida con solo tres líneas rectas, sin levantar el bolígrafo del papel ni volver sobre sus pasos (Acertijos y juegos matemáticos [Hol92] ) ?

---



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 encontrar la caja, identificar las restricciones reales.

Consejo 81

### No piense fuera de la caja: encuentre la caja

Cuando se enfrente a un problema intratable, enumere todas las vías posibles que tiene ante usted. 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? ¿Puedes probarlo?

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, una solución al rompecabezas de los Cuatro Postes se muestra en la

final del libro.

### ¡SALGA DE SU PROPIO CAMINO!

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".

Este es un momento ideal para hacer otra cosa por un tiempo. Trabaja en algo diferente. Ve a pasear al perro. Duerme en él.

Su cerebro consciente es consciente del problema, pero su cerebro consciente es realmente bastante tonto (sin ofender). Así que es hora de darle a tu cerebro real, esa increíble red neuronal asociativa que se esconde debajo de tu conciencia, algo de espacio. Te sorprenderá la frecuencia con la que la respuesta se te viene a la cabeza cuando te distraes deliberadamente.

Si eso te suena demasiado místico, no lo es. Psychology Today informa:

[72]

En pocas palabras, las personas que estaban distraídas obtuvieron mejores resultados en una tarea compleja de resolución de problemas que las personas que se esforzaron conscientemente.

Si aún no está dispuesto a dejar el problema por un tiempo, lo mejor que puede hacer es encontrar a alguien a quien explicárselo. A menudo, la distracción de simplemente hablar de ello te llevará a la iluminación.

Pídeles que te hagan preguntas como:

- ¿Por qué estás resolviendo este problema?
- ¿Cuál es el beneficio de resolverlo?
- ¿Los problemas que tiene están relacionados con casos extremos? ¿Puedes eliminarlos?
- ¿Hay algún problema relacionado más simple que puedas resolver?

Este es otro ejemplo de Rubber Ducking en la práctica.

### LA FORTUNA FAVORECE A LA MENTE PREPARADA

Se dice que Louis Pasteur dijo:

Dans les champs de l'observation le hasard ne favorise que les esprits préparés.

(Cuando se trata de observación, la fortuna favorece a la mente preparada).

Eso también es cierto para la resolución de problemas. Para tener esos eureka! momentos, su cerebro no consciente necesita tener suficiente materia prima; experiencias previas que pueden contribuir a una respuesta.

Una excelente manera de alimentar su cerebro es darle retroalimentación sobre lo que funciona y lo que no funciona mientras realiza su trabajo diario. Y describimos una excelente manera de hacerlo usando un Diario de Ingeniería (Tema 22, Diarios de Ingeniería).

Y recuerda siempre el consejo de la portada de La guía del autoestopista galáctico: NO TE ENTRES EN PÁNICO.

### LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 5, Software suficientemente bueno

- Tema 37, Escucha tu cerebro de lagarto
- Tema 45, El pozo de requisitos
- Andy escribió un libro completo sobre este tipo de cosas: Pensamiento y aprendizaje pragmáticos: Refactorice su Wetware [Hun08].

## RETOS

- Eche un vistazo detenidamente a cualquier problema difícil en el que se encuentre envuelto hoy. ¿Puedes cortar el nudo gordiano? ¿Tienes que hacerlo de esta manera? ¿Tienes que hacerlo en absoluto?
- ¿Se le entregó un conjunto de restricciones cuando se inscribió en su proyecto actual? ¿Siguen siendo aplicables todos, y sigue siendo válida su interpretación?



Tema 47

## Trabajando juntos

nunca he conocido a un  
ser humano que  
querría leer  
17.000 páginas de  
documentación, y si las  
hubiera, lo mataría para  
sacarlo del acervo  
genético.

José Costello, presidente de  
Cadencia

Fue uno de esos proyectos "imposibles", del tipo que escuchas que suena emocionante y aterrador al mismo tiempo. Un sistema antiguo se acercaba al final de su vida útil, el hardware desaparecía físicamente y se tenía que diseñar un sistema completamente nuevo que coincidiera exactamente con el comportamiento (a menudo no documentado) . Muchos cientos de millones de dólares del dinero de otras personas pasarían a través de este sistema, y el plazo desde el inicio hasta

el despliegue era del orden de meses.

Y ahí es donde Andy y Dave se conocieron por primera vez. Un proyecto imposible con un plazo ridículo. Solo hubo una cosa que hizo que el proyecto fuera un éxito rotundo. La experta que había manejado este sistema durante años estaba sentada en su oficina, al otro lado del pasillo de nuestra sala de desarrollo del tamaño de un armario de escobas. Siempre disponible para preguntas, aclaraciones, decisiones y demostraciones.

A lo largo de este libro, recomendamos trabajar en estrecha colaboración con los usuarios; son parte de tu equipo. En ese primer proyecto juntos,

practicamos lo que ahora podría llamarse programación en pareja o programación en masa: una persona escribiendo código mientras uno o más miembros del equipo comentan, reflexionan y resuelven problemas juntos. Es una forma poderosa de trabajar juntos que trasciende reuniones interminables, memorandos y documentación legalista sobrecargada apreciada por su peso sobre su utilidad.

Y eso es lo que realmente queremos decir con "trabajar con": no solo hacer preguntas, tener discusiones y tomar notas, sino hacer preguntas y tener discusiones mientras estás programando.

#### Ley de Conway

En 1967, Melvin Conway introdujo una idea en *¿Cómo inventan los comités?* [Con68] que se conocería como Ley de Conway:

Las organizaciones que diseñan sistemas están obligadas a producir diseños que sean copias de las estructuras de comunicación de estas organizaciones.

Es decir, las estructuras sociales y las vías de comunicación del equipo y la organización se reflejarán en la aplicación, el sitio web o el producto que se está desarrollando.

Varios estudios han mostrado un fuerte apoyo a esta idea. Lo hemos presenciado de primera mano innumerables veces, por ejemplo, en equipos en los que nadie se habla en absoluto, lo que da como resultado sistemas de "tubo de estufa" en silos. O equipos que se dividieron en dos, lo que resultó en una división cliente/servidor o frontend/backend.

Los estudios también ofrecen soporte para el principio inverso: puede estructurar deliberadamente su equipo de la forma en que desea que se vea su código. Por ejemplo, se muestra que los equipos distribuidos geográficamente tienden hacia un software distribuido más modular.

Pero lo más importante es que los equipos de desarrollo que incluyan usuarios producirán software que refleje claramente esa participación, y los equipos que no se molesten también lo reflejarán.

## PROGRAMACIÓN POR PAREJAS

La programación en pares es una de las prácticas de eXtreme Programming que se ha vuelto popular fuera del propio XP. En la programación en pareja, un desarrollador opera el teclado y

el otro no. Ambos trabajan juntos en el problema y pueden cambiar las tareas de mecanografía según sea necesario.

Son muchos los beneficios de la programación en pareja. Diferentes personas aportan diferentes antecedentes y experiencias, diferentes técnicas y enfoques de resolución de problemas, y diferentes niveles de enfoque y atención a cualquier problema dado. El desarrollador que actúa como mecanógrafo debe centrarse en los detalles de bajo nivel de la sintaxis y el estilo de codificación, mientras que el otro desarrollador es libre de considerar los problemas y el alcance de nivel superior. Si bien eso puede parecer una pequeña distinción, recuerde que los humanos solo tenemos un ancho de banda cerebral limitado. Jugar con escribir palabras y símbolos esotéricos que el compilador aceptará a regañadientes requiere bastante de nuestro propio poder de procesamiento. Tener el cerebro completo de un segundo desarrollador disponible durante la tarea aporta mucho más poder mental.

La presión de grupo inherente de una segunda persona ayuda contra los momentos de debilidad y los malos hábitos de nombrar variables `foo` y tal. Está menos inclinado a tomar un atajo potencialmente vergonzoso cuando alguien está mirando activamente, lo que también da como resultado un software de mayor calidad.

## PROGRAMACIÓN MÚLTIPLES

Y si dos cabezas piensan mejor que una, ¿qué tal tener una docena de personas diversas trabajando en el mismo problema al mismo tiempo, con un mecanógrafo?

La programación mafiosa, a pesar del nombre, no implica antorchas ni horcas. Es una extensión de la programación en pareja que involucra a más de dos desarrolladores. Los defensores reportan grandes resultados usando mobs para resolver problemas difíciles. Las turbas pueden incluir fácilmente a personas que generalmente no se consideran parte del desarrollo.

equipo, incluidos los usuarios, los patrocinadores del proyecto y los evaluadores. De hecho, en nuestro primer proyecto "imposible" juntos, era común que uno de nosotros estuviera escribiendo mientras el otro discutía el problema con nuestro experto en negocios. Era una pequeña multitud de tres.

Puede pensar en la programación de la mafia como una estrecha colaboración con la codificación en vivo.

### ¿QUÉ TENGO QUE HACER?

Si actualmente solo está programando solo, tal vez intente programar en pareja. Déle un mínimo de dos semanas, solo unas pocas horas a la vez, ya que se sentirá extraño al principio. Para hacer una lluvia de ideas nuevas o diagnosticar problemas espinosos, tal vez pruebe una sesión de programación de la mafia.

Si ya estás emparejando o mobbing, ¿quién está incluido? ¿Son solo desarrolladores, o permites que participen miembros de tu equipo ampliado: usuarios, probadores, patrocinadores...?

Y como con toda colaboración, debe administrar los aspectos humanos y técnicos. Estos son solo algunos consejos para comenzar:

- Construye el código, no tu ego. No se trata de quién es más brillante; Todos tenemos nuestros momentos, buenos y malos.
- Empieza pequeño. Mob con solo 4-5 personas, o comience con solo unas pocas parejas, en sesiones cortas.
- Critica el código, no a la persona. “Veamos este bloque” suena mucho mejor que “estás equivocado”.
- Escuche y trate de comprender los puntos de vista de los demás. Diferente no es equivocado.

- Realice retrospectivas frecuentes para tratar de mejorar para la próxima vez.

Programar en la misma oficina o de forma remota, solo, en parejas o en grupos, son formas efectivas de trabajar juntos para resolver problemas. Si usted y su equipo solo lo han hecho de una manera, es posible que desee experimentar con un estilo diferente. Pero no se lance con un enfoque ingenuo: hay reglas, sugerencias y pautas para cada uno de estos estilos de desarrollo. Por ejemplo, con la programación de la mafia, cambias al mecanógrafo cada 5-10 minutos.

Lea e investigue un poco, tanto de los libros de texto como de los informes de experiencia, y obtenga una idea de las ventajas y las dificultades que puede encontrar. Es posible que desee comenzar codificando un ejercicio simple, y no simplemente saltar directamente a su código de producción más difícil.

Pero como sea que lo haga, permítanos sugerirle un consejo final:

Consejo 82

No entre solo en el código



Tema 48

## La esencia de la agilidad

Sigues usando esa palabra, no lo creo  
medio  
lo que piensas  
medio.

Íñigo Montoya, La princesa  
Novia

Ágil es un adjetivo: es cómo haces algo. Puedes ser un desarrollador ágil. Puede estar en un equipo que adopte prácticas ágiles, un equipo que responda a los cambios y contratiempos con agilidad. La agilidad es tu estilo, no tú.

Consejo 83

ágil no es un sustantivo; Ágil es cómo haces las cosas

Mientras escribimos esto, casi 20 años después del inicio de la Manifiesto para el desarrollo ágil de software, vemos<sup>[73]</sup> a muchos, muchos desarrolladores aplicando con éxito sus valores. Vemos muchos equipos fantásticos que encuentran formas de tomar estos valores y usarlos para guiar lo que hacen y cómo cambian lo que hacen.

Pero también vemos otro lado de la agilidad. Vemos equipos y empresas ansiosos por soluciones listas para usar: Agile-in-a-Box. Y vemos muchos consultores y empresas muy felices de venderles lo que quieren. Vemos empresas que adoptan más capas de gestión, informes más formales, desarrolladores más especializados y títulos de trabajo más elegantes que simplemente significan "alguien con un portapapeles y un cronómetro".<sup>[74]</sup>

Creemos que muchas personas han perdido de vista el verdadero significado de la agilidad y nos gustaría que la gente volviera a lo básico.

Recuerda los valores del manifiesto:

Estamos descubriendo mejores formas de desarrollar software haciéndolo y ayudando a otros a hacerlo. A través de este trabajo hemos llegado a valorar:

- Individuos e interacciones sobre procesos y herramientas.
- Software de trabajo sobre documentación completa
- Colaboración con el cliente sobre la negociación del contrato
- Responder al cambio sobre seguir un plan

Es decir, mientras hay valor en los elementos de la derecha, valoramos más los elementos de la izquierda.

Cualquiera que te venda algo que aumente la importancia de las cosas de la derecha sobre las de la izquierda claramente no valora las mismas cosas que nosotros y los otros escritores del manifiesto.

Y cualquiera que le venda una solución en una caja no ha leído la declaración introductoria. Los valores están motivados e informados por el acto continuo de descubrir mejores formas de producir software. Este no es un documento estático. Son sugerencias para un proceso generativo.

## NUNCA PUEDE HABER UN PROCESO ÁGIL

De hecho, cada vez que alguien dice “haz esto y serás ágil”, se equivoca. Por definición.

Porque la agilidad, tanto en el mundo físico como en el desarrollo de software, tiene que ver con responder al cambio, responder a las incógnitas que encuentras después de que te pones en marcha. Una gacela que corre no va en línea recta. Una gimnasta hace cientos de correcciones por segundo a medida que responde a cambios en su entorno y errores menores en la colocación de sus pies.

Lo mismo ocurre con los equipos y los desarrolladores individuales. No hay un plan único que pueda seguir cuando desarrolla software. Tres de los cuatro valores te dicen eso. Se trata de recopilar y responder a los comentarios.

Los valores no te dicen qué hacer. Le dicen qué buscar cuando decide por sí mismo qué hacer.

Estas decisiones siempre son contextuales: dependen de quién es usted, la naturaleza de su equipo, su aplicación, sus herramientas, su empresa, su cliente, el mundo exterior; un número increíblemente grande de factores, algunos importantes y otros triviales. Ningún plan fijo y estático puede sobrevivir a esta incertidumbre.

### ¿ASI QUE QUE HACEMOS?

Nadie puede decirte qué hacer. Pero creemos que podemos decirte algo sobre el espíritu con el que lo haces. Todo se reduce a cómo lidas con la incertidumbre. El manifiesto sugiere que hagas esto reuniendo y actuando sobre la retroalimentación. Así que aquí está nuestra receta para trabajar de forma ágil:

1. Averigüe dónde se encuentra.

2. Da el paso significativo más pequeño hacia donde quieras estar.

3. Evalúe dónde termina y arregle todo lo que haya roto.

Repita estos pasos hasta que haya terminado. Y úselos recursivamente, en todos los niveles de todo lo que hace.

A veces, incluso la decisión que parece más trivial se vuelve importante cuando recopilas comentarios.

“Ahora mi código necesita obtener el propietario de la cuenta.

```
let usuario = propietario de cuenta (ID de cuenta);
```

Hmmm... **usuario** es un nombre inútil. Lo haré **dueño**.

```
let propietario = propietario de cuenta (ID de cuenta);
```

Pero ahora eso se siente un poco redundante. ¿Qué estoy realmente tratando de hacer aquí? La historia dice que le estoy enviando un correo electrónico a esta persona, así que necesito encontrar su dirección de correo electrónico. Tal vez no necesito el propietario completo de la cuenta en absoluto.

```
let email = emailOfAccountOwner(cuentaID);
```

Al aplicar el ciclo de retroalimentación a un nivel realmente bajo (la denominación de una variable), en realidad hemos mejorado el diseño del sistema general, reduciendo el acoplamiento entre este código y el código que se ocupa de las cuentas.

El ciclo de retroalimentación también se aplica al nivel más alto de un proyecto. Parte de nuestro trabajo más exitoso sucedió cuando comenzamos a trabajar en los requisitos de un cliente, dimos un solo paso y nos dimos cuenta de que lo que estábamos a punto de hacer no era necesario, que la mejor solución ni siquiera involucraba software.

Este ciclo se aplica fuera del alcance de un solo proyecto. Los equipos deben aplicarlo para revisar su proceso y qué tan bien funcionó. Un equipo que no experimenta continuamente con su proceso no es un equipo ágil.

## Y ESTO IMPULSA EL DISEÑO

En el Tema 8, La esencia del buen diseño, afirmamos que la medida del diseño es qué tan fácil es cambiar el resultado de ese diseño: un buen diseño produce algo que es más fácil de cambiar que un mal diseño.

Y esta discusión sobre la agilidad explica por qué ese es el caso.

Haces un cambio y descubres que no te gusta. El paso 3 de nuestra lista dice que tenemos que ser capaces de arreglar lo que rompemos. Para que nuestro circuito de retroalimentación sea eficiente, esta solución debe ser lo menos dolorosa posible. Si no es así, estaremos tentados a encogernos de hombros y dejarlo sin arreglar. Hablamos de este efecto en el Tema 3, Entropía del software. Para hacer que todo esto ágil funcione, necesitamos practicar un buen diseño, porque un buen diseño hace que las cosas sean fáciles de cambiar. Y si es fácil de cambiar, podemos ajustar, en todos los niveles, sin dudarlo.

Eso es agilidad.

## LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 27, No superes tus faros
- Tema 40, Refactorización
- Tema 50, Los cocos no lo cortan

## RETOS

El ciclo de retroalimentación simple no es solo para el software.  
Piense en otras decisiones que haya tomado recientemente.  
¿Podría alguno de ellos haber mejorado al pensar en cómo  
podría deshacerlos si las cosas no lo llevaran en la dirección en la que se dirigía?  
¿Puedes pensar en formas en las que puedes mejorar lo que haces  
reuniendo y actuando en base a los comentarios?

---

notas al pie

[70] ¿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.

[71] <https://www.wired.com/1999/01/eno/>

[72] <https://www.psychologytoday.com/us/blog/your-brain-work/201209/stop-trying-solve-problems>

[73] <https://agilemanifesto.org>

[74] Para obtener más información sobre lo malo que puede ser ese enfoque, consulte *The Tyranny of Metrics* [Mul18].

## Capítulo 9

# 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 el Tema 49, Equipos pragmáticos, mostraremos cómo hacerlo respetando la filosofía pragmática.

El propósito de un método de desarrollo de software es ayudar a las personas a trabajar juntas. ¿Usted y su equipo están haciendo lo que funciona bien para ustedes, o solo están invirtiendo en los artefactos superficiales triviales y no obtienen los beneficios reales que se merecen? Veremos por qué el Tema 50, Los cocos no son suficientes y ofreceremos el verdadero secreto para éxito.

Y, por supuesto, nada de eso importa si no puede entregar software de manera consistente y confiable. Esa es la base del trío mágico de control de versiones, pruebas y automatización: el Topic 51, Pragmatic Starter Kit.

Sin embargo, en última instancia, el éxito está en el ojo del espectador: el patrocinador del proyecto. La percepción del éxito es lo que cuenta, y en el Tema 52, Deleite a sus usuarios, le mostraremos cómo deleitar al patrocinador de cada proyecto.

El último consejo del libro es una consecuencia directa de todo lo demás. En el Tema 53, Orgullo y prejuicio, le pedimos que firme su trabajo y que se enorgullezca de lo que hace.



Tema 49

## Equipos pragmáticos

En el Grupo L, Stof el supervisa seis equipos de primer nivel programadores, un desafío gerencial más o menos comparable a pastorear gatos.

el poste de washington  
Revista, 9 de junio de 1985

Incluso en 1985, la broma sobre el pastoreo de gatos se estaba volviendo vieja. En el momento de la primera edición a principios de siglo, era positivamente antiguo. Sin embargo, persiste, porque suena a verdad.

Los programadores son un poco como los gatos: inteligentes, de voluntad fuerte, obstinados, independientes y, a menudo, adorados por la red.

Hasta ahora en este libro hemos visto técnicas pragmáticas que ayudan a un individuo a ser un mejor programador. ¿Pueden estos métodos funcionar también para equipos, incluso para equipos de personas independientes y de voluntad fuerte? La respuesta es un rotundo “¡sí!” Hay ventajas en ser una persona pragmática, pero estas ventajas se multiplican si la persona trabaja en un equipo pragmático.

Un equipo, desde nuestro punto de vista, es una entidad propia pequeña, en su mayoría estable. Cincuenta personas no son un equipo, son una horda. Los equipos en los que los miembros están constantemente siendo llamados a otras asignaciones y nadie se conoce tampoco son un equipo, son simplemente extraños que comparten temporalmente una parada de autobús bajo la lluvia.<sup>[75]</sup>

Un equipo pragmático es pequeño, de menos de 10 a 12 miembros.

Los miembros van y vienen rara vez. Todos conocen bien a todos, confían unos en otros y dependen unos de otros.

Consejo 84

### Mantener equipos pequeños y estables

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 debe asumir la responsabilidad de la calidad del producto, apoyando a los desarrolladores que entienden la filosofía de ventanas rotas que describimos en el Tema 3, Entropía del software, y alejando a aquellos que aún no la han descubierto.

Algunas metodologías de equipo tienen un "oficial de calidad", alguien a quien el equipo delega la responsabilidad de la calidad del entregable. Esto es claramente ridículo: la calidad solo puede provenir de las contribuciones individuales de todos los miembros del equipo. La calidad se construye, no se atornilla.

## RANAS HERVIDAS

¿Recuerda la rana apócrifa en la olla con agua, en el Tema 4, Sopa de piedra y ranas hervidas? 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. Anime a todos a monitorear activamente el entorno en busca de cambios. Manténgase despierto y consciente de un mayor alcance, escalas de tiempo reducidas, características adicionales, nuevos entornos, cualquier cosa que no estaba en el entendimiento original. Mantenga métricas sobre los nuevos requisitos.<sup>[76]</sup> No es necesario que el equipo rechace los cambios sin más, simplemente debe ser consciente de que están ocurriendo. De lo contrario, estarás tú en el agua caliente.

## AGENDA TU PORTAFOLIO DE CONOCIMIENTOS

En el Tema 6, Su Portafolio de Conocimientos , analizamos las formas en que debe invertir en su Portafolio de Conocimientos personal en su propio tiempo. Los equipos que quieren tener éxito también deben considerar sus inversiones en conocimientos y habilidades.

Si su equipo se toma en serio la mejora y la innovación, debe programarlo. Tratar de hacer las cosas “siempre que haya un momento libre” significa que nunca sucederán. Cualquier tipo de

trabajo pendiente, lista de tareas o flujo con el que está trabajando, no lo reserve solo para el desarrollo de características. El equipo trabaja en algo más que nuevas funciones. Algunos ejemplos posibles incluyen:

#### Mantenimiento de sistemas

antiguos Si bien nos encanta trabajar en el nuevo y reluciente sistema, es probable que haya que realizar tareas de mantenimiento en el sistema anterior. Hemos conocido equipos que intentan y empujan este trabajo en la esquina. Si el equipo está encargado de hacer estas tareas, entonces hágalas, de verdad.

#### Reflexión y refinamiento del proceso

La mejora continua solo puede ocurrir cuando se toma el tiempo de mirar a su alrededor, descubrir qué funciona y qué no, y luego realizar cambios (consulte el Tema 48, *La esencia de la agilidad*). Demasiados equipos están tan ocupados sacando agua que no tienen tiempo para reparar la fuga. Programarla. Arreglalo.

#### Experimentos con

nuevas tecnologías No adopte nuevas tecnologías, marcos o bibliotecas simplemente porque “todo el mundo lo está haciendo”, o basándose en algo que vio en una conferencia o leyó en línea. Examine deliberadamente las tecnologías candidatas con prototipos. Ponga tareas en el cronograma para probar las cosas nuevas y analizar los resultados.

#### Mejoras en el aprendizaje y las

habilidades El aprendizaje y las mejoras personales son un gran comienzo, pero muchas habilidades son más efectivas cuando se distribuyen en todo el equipo. Planee hacerlo, ya sea en el almuerzo informal o en sesiones de capacitación más formales.

## Programarlo para que suceda

### COMUNICAR LA PRESENCIA DEL EQUIPO

Es obvio que los desarrolladores de un equipo deben hablar entre ellos. Dimos algunas sugerencias para facilitar esto en el Tema 7, ¡Comunícate!. Sin embargo, es fácil olvidar que el propio equipo tiene presencia dentro de la organización. El equipo como entidad necesita comunicarse claramente con el resto del mundo.

Para los extraños, los peores equipos de proyecto son aquellos que parecen hoscos y reticentes. Hacen reuniones sin estructura, donde nadie quiere hablar. Sus correos electrónicos y documentos de proyectos son un desastre: no hay dos 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 una actuación bien preparada que hará que todos se sientan bien. La documentación que producen es nítida, precisa y consistente. El equipo habla con una sola voz. Incluso pueden tener sentido del humor.<sup>[77]</sup>

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, jerbos, personajes de dibujos animados y ciudades míticas). Dedique 30 minutos a idear un logotipo estafalario y utilícelo. 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 SE REPITAN

En el Tema 9, SECO: Los males de la duplicación, 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. Los sistemas "Stovepipe" o "siloed" son comunes en estos equipos, con poco intercambio y mucha funcionalidad duplicada.

La buena comunicación es clave para evitar estos problemas. Y por "bueno" queremos decir instantáneo y sin fricciones.

Debería poder hacer una pregunta a los miembros del equipo y obtener una respuesta más o menos instantánea. Si el equipo está ubicado en el mismo lugar, esto podría ser tan simple como asomar la cabeza por encima de la pared del cubo o por el pasillo. Para los equipos remotos, es posible que deba confiar en una aplicación de mensajería u otros medios electrónicos.

Si tiene que esperar una semana para que la reunión del equipo haga su pregunta o comparta su estado, eso es mucha fricción.

[78]

Sin fricciones significa que es fácil y sencillo hacer preguntas, compartir su progreso, sus problemas, sus ideas y aprendizajes, y estar al tanto de lo que están haciendo sus compañeros de equipo.

Mantenga la conciencia para mantenerse SECO.

## BALAS TRAZADORAS DEL EQUIPO

Un equipo de proyecto tiene que realizar muchas tareas diferentes en diferentes áreas del proyecto, tocando muchas tecnologías diferentes. La comprensión de los requisitos, el diseño de la arquitectura, la codificación para la interfaz y el servidor, las pruebas, todo tiene que suceder. Pero es un error común pensar que estas actividades y tareas pueden ocurrir por separado, de forma aislada. no pueden

Algunas metodologías abogan por todo tipo de roles y títulos diferentes dentro del equipo, o crean equipos especializados separados por completo. Pero el problema con ese enfoque es que introduce puertas y handofs. Ahora, en lugar de un flujo fluido desde el equipo hasta el despliegue, tiene puertas artificiales donde se detiene el trabajo. Transferencias que tienen que esperar para ser aceptadas.

Aprobaciones. Papeleo. La gente Lean llama a esto desperdicio y se esfuerza por eliminarlo activamente.

Todos estos roles y actividades diferentes son en realidad puntos de vista diferentes del mismo problema, y separarlos artificialmente puede causar muchos problemas. Por ejemplo, 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.

Con el Tema 12, Tracer Bullets, recomendamos desarrollar funciones individuales, por pequeñas y limitadas que sean inicialmente, que abarquen todo el sistema. Eso significa que necesita todas las habilidades para hacer eso dentro del equipo: frontend, UI/UX, servidor, DBA, QA, etc., todos cómodos y acostumbrados a trabajar juntos. Con un enfoque de bala de seguimiento, puede implementar fragmentos muy pequeños de funcionalidad muy rápidamente y obtener comentarios inmediatos sobre qué tan bien se comunica y entrega su equipo. Eso crea un entorno en el que puede realizar cambios y ajustar su equipo y procesos de forma rápida y sencilla.

Consejo 86

### Organizar equipos completamente funcionales

Cree equipos para que pueda crear código de principio a fin, de forma incremental e iterativa.

## AUTOMATIZACIÓN

Una excelente manera de garantizar tanto la consistencia como la precisión es automatizar todo lo que hace el equipo. ¿Por qué luchar con los estándares de formato de código cuando su editor o IDE pueden hacerlo automáticamente? ¿Por qué hacer pruebas manuales cuando la compilación continua puede ejecutar pruebas automáticamente? ¿Por qué implementar a mano cuando la automatización puede hacerlo de la misma manera cada vez, de manera repetible y confiable?

La automatización es un componente esencial de cada equipo de proyecto. Asegúrese de que el equipo tenga habilidades en la creación de herramientas para construir e implementar las herramientas que automatizan el desarrollo del proyecto y la implementación de la producción.

## SABER CUANDO DEJAR DE AÑADIR PINTURA

Recuerda que los equipos están formados por individuos. Dale a cada miembro la capacidad de brillar a su manera. Ofrezcales la estructura suficiente para apoyarlos y garantizar que el proyecto genere valor. Luego, como el pintor del Tema 5, Software suficientemente bueno, resista la tentación de agregar más pintura.

## LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 2, [El gato se comió mi código fuente](#)
- Tema 7, [¡Comuníquese!](#)
- Tema 12, [Viñetas trazadoras](#)
- Tema 19, [Control de versiones](#)
- Tema 50, [Los cocos no lo cortan](#)
- Tema 51, [Kit de inicio pragmático](#)

## RETOS

- 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.
- Probablemente alguna vez le dieron 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 si los trabajadores estuvieran escribiendo código en su lugar? ¿En cuántos escenarios se reduce realmente el tiempo?
- Lea El mes del hombre mítico [Bro96] de Frederick Brooks. Para crédito adicional, compre dos copias para que pueda leerlo el doble de rápido.



Tema 50

Los cocos no lo cortan

Los isleños nativos nunca antes habían visto un avión ni conocido a personas como estos extraños. A cambio del uso de su tierra, los extraños proporcionaron pájaros mecánicos que entraban y salían volando durante todo el día en una “pista de aterrizaje”, trayendo una increíble riqueza material a su isla natal. Los extraños mencionaron algo sobre la guerra y la lucha. Un día todo terminó y todos se fueron, llevándose consigo sus extrañas riquezas.

Los isleños estaban desesperados por restaurar su buena fortuna y reconstruyeron un facsímil del aeropuerto, la torre de control y el equipo utilizando materiales locales: enredaderas, cáscaras de coco, hojas de palma y demás. Pero por alguna razón, a pesar de que tenían todo en su lugar, los aviones no llegaron. Habían imitado la forma, pero no el contenido. Los antropólogos llaman a esto un culto de carga.

Con demasiada frecuencia, somos los isleños.

Es fácil y tentador caer en la trampa del culto a la carga: al invertir y construir los artefactos fácilmente visibles, esperas atraer la magia subyacente que funciona. Pero al igual que con los cultos de carga originales de Melanesia, un aeropuerto falso hecho de cáscaras de coco <sup>[79]</sup> no sustituye al real.

Por ejemplo, hemos visto personalmente equipos que afirman estar usando Scrum. Pero, después de un examen más detallado, resultó que estaban haciendo una reunión diaria de pie una vez por semana, con iteraciones de cuatro semanas que a menudo se convertían en seis u ocho semanas.

iteraciones Sintieron que esto estaba bien porque estaban usando una popular herramienta de programación "ágil". Solo estaban invirtiendo en los artefactos superficiales, e incluso entonces, a menudo solo de nombre, como si "ponerse de pie" o "iteración" fuera una especie de encantamiento para los supersticiosos. Como era de esperar, ellos tampoco lograron atraer la verdadera magia.

## CUESTIONES DE CONTEXTO

¿Usted o su equipo han caído en esta trampa? Pregúntese, ¿por qué está usando ese método de desarrollo en particular? O ese marco? ¿O esa técnica de prueba? ¿Es realmente adecuado para el trabajo en cuestión? ¿Te funciona bien? ¿O fue adoptado simplemente porque estaba siendo utilizado por la última historia de éxito impulsada por Internet?

Existe una tendencia actual a adoptar las políticas y procesos de empresas exitosas como Spotify, Netflix, Stripe, GitLab y otras. Cada uno tiene su propia visión única del desarrollo y la gestión de software. Pero considere el contexto: ¿está usted en el mismo mercado, con las mismas limitaciones y oportunidades, experiencia y tamaño de organización similares, administración similar y cultura similar? ¿Base de usuarios y requisitos similares?

No se deje engañar. Los artefactos particulares, las estructuras superficiales, las políticas, los procesos y los métodos no son suficientes.

Consejo 87

Haz lo que funcione, no lo que esté de moda

¿Cómo sabes "lo que funciona"? Confías en la más fundamental de las técnicas pragmáticas:

Intentalo.

Pon a prueba la idea con un pequeño equipo o conjunto de equipos. Mantenga las partes buenas que parecen funcionar bien y deseche cualquier otra cosa como desperdicio o gastos generales. Nadie degradará su organización porque opera de manera diferente a Spotify o Netflix, porque incluso ellos no siguieron sus procesos actuales mientras crecían. Y dentro de unos años, a medida que esas empresas maduren, pivoten y sigan prosperando, volverán a hacer algo diferente.

Ese es el verdadero secreto de su éxito.

#### TALLA ÚNICA A NADIE LE QUEDA BIEN

El propósito de una metodología de desarrollo de software es ayudar a las personas a trabajar juntas. Como discutimos en el Tema 48, La esencia de la agilidad, no hay un plan único que pueda seguir cuando desarrolla software, especialmente un plan que se le ocurrió a otra persona en otra empresa.

Muchos programas de certificación son incluso peores que eso: se basan en que el estudiante pueda memorizar y seguir las reglas. Pero eso no es lo que quieres. Necesita la capacidad de ver más allá de las reglas existentes y explotar las posibilidades para obtener una ventaja. Esa es una mentalidad muy diferente de "pero Scrum/Lean/Kanban/XP/agile lo hace de esta manera..." y así sucesivamente.

En su lugar, desea tomar las mejores piezas de cualquier metodología en particular y adaptarlas para su uso. No hay una talla única para todos, y los métodos actuales están lejos de ser completos, por lo que deberá buscar más de un método popular.

Por ejemplo, Scrum define algunos proyectos de gestión

prácticas, pero Scrum por sí mismo no brinda suficiente orientación a nivel técnico para los equipos o a nivel de cartera/gobernanza para el liderazgo. Entonces, ¿por dónde empiezas?

¡Sé como ellos!

Con frecuencia escuchamos a los líderes de desarrollo de software decir a su personal: "Deberíamos operar como Netflix" (o una de estas otras empresas líderes). Por supuesto que podrías hacer eso.

Primero, consiga unos cientos de miles de servidores y decenas de millones de usuarios...

## EL VERDADERO OBJETIVO

El objetivo, por supuesto, no es "hacer Scrum", "hacer ágil", "hacer Lean" o lo que sea. El objetivo es estar en condiciones de ofrecer un software que funcione y que brinde a los usuarios alguna capacidad nueva en cualquier momento. No dentro de semanas, meses o años, sino ahora. Para muchos equipos y organizaciones, la entrega continua se siente como un objetivo elevado e inalcanzable, especialmente si tiene un proceso que restringe la entrega a meses o incluso semanas. Pero como con cualquier objetivo, la clave es seguir apuntando en la dirección correcta.

imágenes/horas\_de\_entrega.png

Si va a dar a luz en años, intente acortar el ciclo a meses. De meses, redúcelo a semanas. De un sprint de cuatro semanas, prueba dos. De un sprint de dos semanas, prueba uno. Entonces diariamente. Luego, finalmente, bajo demanda. Tenga en cuenta que poder entregar a pedido no significa que esté obligado a entregar cada minuto de cada día. Entrega cuando los usuarios lo necesitan, cuando tiene sentido comercial hacerlo.

Consejo 88

Entregar cuando los usuarios lo necesiten

Para pasar a este estilo de desarrollo continuo, necesita una infraestructura sólida, que analizamos en el siguiente tema, Tema 51, Kit de inicio pragmático. Usted realiza el desarrollo en el tronco principal de su sistema de control de versiones, no en las sucursales, y utiliza técnicas tales como cambios de funciones para implementar funciones de prueba para los usuarios de forma selectiva.

Una vez que su infraestructura esté en orden, debe decidir cómo

organizar el trabajo. Los principiantes pueden querer comenzar con Scrum para la gestión de proyectos, además de las prácticas técnicas de eXtreme Programming (XP). Los equipos más disciplinados y experimentados podrían recurrir a las técnicas Kanban y Lean, tanto para el equipo como para problemas de gobernanza más amplios.

Pero no confíe en nuestra palabra, investigue y pruebe estos enfoques usted mismo. Sin embargo, tenga cuidado al exagerar.

Invertir demasiado en una metodología en particular puede dejarlo ciego ante las alternativas. Te acostumbras. Pronto se hace difícil ver de otra manera. Te has calcificado y ahora ya no puedes adaptarte rápidamente.

Bien podría estar usando cocos.

## LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 12, [Viñetas trazadoras](#)
- Tema 27, [No superes tus faros](#)
- Tema 48, [La esencia de la agilidad](#)
- Tema 49, [Equipos Pragmáticos](#)
- Tema 51, [Kit de inicio pragmático](#)



Tema 51

## Kit de inicio pragmático

La civilización avanza ampliando el número de operaciones importantes que podemos realizar sin pensar.

Alfred North Whitehead

Antes, cuando los automóviles eran una novedad, las instrucciones para arrancar un Ford Modelo T tenían más de dos páginas. Con los automóviles modernos, solo presiona un botón: 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 el desarrollo de software 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 el procedimiento de compilación y lanzamiento, las pruebas, el papeleo del proyecto o cualquier otra tarea recurrente en el proyecto, debe ser automático y repetible en cualquier máquina capaz.

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.

Después de escribir la primera edición de *El programador pragmático*, queríamos crear más libros para ayudar a los equipos a desarrollar software. Pensamos que deberíamos empezar por el principio: ¿cuáles son los más

elementos básicos y más importantes que todo equipo necesita, independientemente de la metodología, el lenguaje o la tecnología. Y así nació la idea del Pragmatic Starter Kit , que cubre estos tres temas críticos e interrelacionados:

- Control de versiones
- Pruebas de regresión
- Automatización completa

Estas son las tres patas que sustentan todo proyecto. Así es cómo.

## ACCIONAMIENTO CON CONTROL DE VERSIÓN

Como dijimos en el Tema 19, [Control de versiones](#), desea mantener todo lo necesario para construir su proyecto bajo control de versiones.

Esa idea se vuelve aún más importante en el contexto del proyecto mismo.

Primero, permite que las máquinas de construcción sean efímeras. En lugar de una máquina sagrada y chirriante en la esquina de la oficina que todos tienen miedo de tocar, se crean máquinas y/o clústeres bajo demanda como instancias puntuales en la nube.<sup>[80]</sup>

La configuración de implementación también está bajo control de versiones, por lo que la liberación a producción se puede manejar automáticamente.

Y esa es la parte importante: a nivel de proyecto, el control de versiones impulsa el proceso de compilación y lanzamiento.

Consejo 89

Use el control de versiones para impulsar compilaciones, pruebas y Lanzamientos

Es decir, la compilación, la prueba y la implementación se activan a través de confirmaciones o

empuja al control de versiones y se integra en un contenedor en la nube.

El lanzamiento a ensayo o producción se especifica mediante el uso de una etiqueta en su sistema de control de versiones. Los lanzamientos se convierten entonces en una parte mucho más sencilla de la vida cotidiana: una verdadera entrega continua, no atada a ninguna máquina de construcción o máquina de desarrollador.

## PRUEBAS DESPIADADAS Y CONTINUAS

Muchos desarrolladores prueban suavemente, inconscientemente sabiendo dónde se romperá el código y evitando los puntos débiles. Los programadores pragmáticos son diferentes. Estamos impulsados a encontrar nuestros errores ahora, por lo que no tenemos que soportar la vergüenza de que otros encuentren nuestros errores más tarde.

Encontrar insectos es algo así como pescar con una red. 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.

Consejo 90

Pruebe temprano, pruebe con frecuencia, pruebe automáticamente

Queremos comenzar a probar tan pronto como tengamos el código. Esos pequeños pececillos tienen la desagradable costumbre de convertirse rápidamente en tiburones gigantes que comen hombres, y atrapar un tiburón es un poco más difícil. Así que escribimos pruebas unitarias. Muchas pruebas unitarias.

De hecho, un buen proyecto puede tener más código de prueba que código de producción. El tiempo que lleva producir este código de prueba vale la pena. Termina siendo mucho más barato a largo plazo, y en realidad tienes la oportunidad de producir un producto con

cerca de cero defectos.

Además, saber que pasó la prueba le brinda un alto grado de confianza de que una parte del código está "terminada".

Consejo 91

La codificación no se hace hasta que se ejecutan todas las pruebas

La compilación automática ejecuta todas las pruebas disponibles. Es importante apuntar a "probar de verdad", en otras palabras, el entorno de prueba debe coincidir estrechamente con el entorno de producción. Cualquier brecha es donde se reproducen los insectos.

La compilación puede cubrir varios tipos importantes de pruebas de software: pruebas unitarias; pruebas de integración; validación y verificación; y pruebas de rendimiento.

Esta lista no está completa y algunos proyectos especializados también requerirán otros tipos de pruebas. Pero nos da un buen punto de partida.

## Pruebas

unitarias Una prueba unitaria es un código que ejercita un módulo. Cubrimos esto en el Tema 41, Test to Code. Las pruebas unitarias son 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

Las pruebas de integración muestran 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 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 cómo se diferencian de los datos de prueba del desarrollador (para ver un ejemplo, consulte la historia sobre las pinceladas aquí).

## Pruebas de rendimiento

Las pruebas de rendimiento o estrés también pueden ser aspectos importantes 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 pruebas especializadas

hardware o software para simular la carga de forma realista.

### Probar las pruebas

Como 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, provoque el error deliberadamente y asegúrese de que la prueba se queje. Esto asegura que la prueba detectará el error si sucede de verdad.

Consejo 92

### Use saboteadores para probar sus pruebas

Si realmente se toma en serio las pruebas, tome una rama separada del árbol de código fuente, introduzca errores a propósito y verifique que las pruebas los atrapen. En un nivel superior, puedes usar algo como Chaos Monkey de Netflix para <sup>[81]</sup> interrumpir (es decir, "matar") los servicios y probar la resistencia de su aplicación.

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 ha creado, ¿cómo sabe si ha probado el código base lo suficientemente a fondo?

La respuesta corta es “no lo harás”, y nunca lo harás. Puede probar las herramientas de análisis de cobertura que vigilan su código durante

probar y realizar 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 % [82] .

-----

Incluso si llegas a cada línea de código, esa no es la imagen completa. Lo importante es el número de estados que puede tener su programa. Los estados no son equivalentes a 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:

```
int test(int a, int b)
{ return a / (a +
b); }
```

En teoría, esta función de tres líneas tiene 1.000.000 de estados lógicos, 999.999 de los cuales funcionarán correctamente y uno no (cuando *a + b* es 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, este es un problema realmente difícil . Duro como en, "El sol será un bullo frío y duro antes de que puedas resolverlo".

Consejo 93

### Probar la cobertura del estado, no la cobertura del código

Pruebas basadas en

propiedades Una excelente manera de explorar cómo su código maneja estados inesperados es hacer que una computadora genere esos estados.

Utilice técnicas de prueba basadas en propiedades para generar datos de prueba de acuerdo con los contratos e invariantes del código bajo prueba.

Cubrimos este tema en detalle en el Tema 42, Pruebas basadas en propiedades.

-----

## APRETAR 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.

Consejo 94

Encuentra errores una vez

Una vez que un probador humano encuentra un error, debería ser la última vez 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.

## AUTOMATIZACIÓN COMPLETA

Como dijimos al comienzo de esta sección, el desarrollo moderno se basa en procedimientos automáticos con guiones. Ya sea que use algo tan simple como scripts de shell con rsync y ssh, o soluciones completas como Ansible, Puppet, Chef o Salt, simplemente no confíe en ninguna intervención manual.

Érase 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

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.

Consejo 95

### No use procedimientos manuales

Las personas simplemente no son tan repetibles como las computadoras. Tampoco debemos esperar que lo sean. Un script o programa de shell ejecutará las mismas instrucciones, en el mismo orden, una y otra vez. Está bajo el control de versiones, por lo que también puede examinar los cambios en los procedimientos de compilación/lanzamiento a lo largo del tiempo ("pero solía funcionar ...").

Todo depende de la automatización. No puede compilar el proyecto en un servidor en la nube anónimo a menos que la compilación sea completamente automática. No puede implementar automáticamente si hay pasos manuales involucrados. Y una vez que introduce los pasos manuales ("solo para esta parte...") ha roto una ventana muy grande. [83]

Con estas tres patas de control de versiones, pruebas despiadadas y automatización completa, su proyecto tendrá la base firme que necesita para que pueda concentrarse en la parte difícil: deleitar a los usuarios.

## LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 11, Reversibilidad

- Tema 12, Viñetas trazadoras
- Tema 17, Juegos de conchas
- Tema 19, Control de versiones
- Tema 41, Prueba de código
- Tema 49, Equipos Pragmáticos
- Tema 50, Los cocos no lo cortan

## RETOS

- ¿Sus compilaciones nocturnas o continuas son automáticas, pero la implementación en producción no lo es? ¿Por qué? ¿Qué tiene de especial ese servidor?
- ¿Puedes probar automáticamente tu proyecto por completo? 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?



Tema 52

## Deleite a sus usuarios

Cuando encantas a las personas, tu objetivo no es ganar dinero con ellas o conseguir que hagan lo que tú quieras, sino llenarlas de gran deleite.

chico kawasaki

Nuestro objetivo como desarrolladores es deleitar a los usuarios. Por eso estamos aquí. No para minarlos en busca de sus datos, ni contar sus globos oculares ni vaciar sus billeteras. Dejando a un lado los objetivos nefastos, incluso entregar software que funcione de manera oportuna no es suficiente. Eso por sí solo no los deleitará.

Sus usuarios no están particularmente motivados por el código. En cambio, tienen un problema comercial que necesita resolverse dentro del contexto de sus objetivos y presupuesto. Su creencia es que al trabajar

con su equipo podrán hacer esto.

Sus expectativas no están relacionadas con el software. Ni siquiera están implícitos en ninguna especificación que le den (porque esa especificación estará incompleta hasta que su equipo la haya repetido varias veces).

Entonces, ¿cómo descubres sus expectativas? Haz una pregunta sencilla:

¿Cómo sabrá que todos hemos tenido éxito un mes (o un año, o lo que sea) después de que termine este proyecto?

Puede que te sorprenda la respuesta. Un proyecto para mejorar las recomendaciones de productos podría juzgarse en términos de retención de clientes; un proyecto para consolidar dos bases de datos podría juzgarse en términos de calidad de datos, o podría tratarse de ahorros de costos. Pero son estas expectativas de valor comercial las que realmente cuentan, no solo el proyecto de software en sí. El software es sólo un medio para estos fines.

Y ahora que ha sacado a la luz algunas de las expectativas subyacentes de valor detrás del proyecto, puede comenzar a pensar en cómo puede cumplir con ellas:

- Asegúrese de que todos en el equipo tengan totalmente claras estas expectativas.
- Al tomar decisiones, piense en qué camino se acerca más a esas expectativas.
- Analizar críticamente los requerimientos del usuario a la luz de las expectativas. En muchos proyectos, hemos descubierto que el "requisito" declarado era, de hecho, solo una suposición de lo que podría lograr la tecnología: en realidad, era un plan de implementación amateur disfrazado de documento de requisitos. No tenga miedo de hacer sugerencias que cambien el requisito si puede demostrar que acercarán el proyecto al objetivo.
- Continúe pensando en estas expectativas a medida que avanza en el proyecto.

Descubrimos que a medida que aumenta nuestro conocimiento del dominio, podemos hacer sugerencias sobre otras cosas que se podrían hacer para abordar los problemas comerciales subyacentes. Creemos firmemente que los desarrolladores, que están expuestos a muchos aspectos diferentes de una organización, a menudo pueden encontrar formas de unir diferentes partes del negocio que no siempre son obvias para los departamentos individuales.

Consejo 96

### Deleite a los usuarios, no se limite a entregar código

Si quieres deleitar a tu cliente, forja una relación con él en la que puedas ayudarlo activamente a resolver sus problemas. Aunque su título podría ser una variación de "Desarrollador de software" o "Ingeniero de software", en realidad debería ser "Solucionador de problemas".

Eso es lo que hacemos, y esa es la esencia de un programador pragmático.

Solucionamos problemas.

#### LAS SECCIONES RELACIONADAS INCLUYEN

- Tema 12, Viñetas trazadoras
- Tema 13, Prototipos y Post-it Notes
- Tema 45, El pozo de requisitos



Tema 53

## 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.

Consejo 97

## 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, eXtreme Programming de Kent Beck<sup>[84]</sup> 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 profesional.

Un programador pragmático.

Gracias.

imágenes/dave\_y\_andy.png

---

notas al pie

[75] A medida que crece el tamaño del equipo, las rutas de comunicación crecen  , dónde está el  a la velocidad de la cantidad de miembros del equipo. En equipos más grandes, la comunicación comienza a fallar y se vuelve ineficaz.

[76] Un gráfico de quemado es mejor para esto que el gráfico de quemado más habitual . Con un gráfico de quemado, puede ver claramente cómo las características adicionales mueven los postes de la portería.

[77] 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.

[78] Andy ha conocido equipos que realizan sus reuniones diarias de Scrum los viernes.

[79] Consulte [https://en.wikipedia.org/wiki/Cargo\\_cult](https://en.wikipedia.org/wiki/Cargo_cult).

[80] Hemos visto esto de primera mano más veces de lo que piensas.

[81] <https://netflix.github.io/chaosmonkey>

[82] Para un estudio interesante de la correlación entre la cobertura de prueba y los defectos, consulte Cobertura de prueba de unidad mítica [ADSS18].

[83] Recuerde siempre el Tema 3, Entropía del software. Siempre.

[84] <http://www.extremeprogramming.org>

---

Copyright © 2020 Pearson Educación, Inc.

A la larga, moldeamos  
nuestras vidas y nos moldeamos a  
nosotros mismos. El proceso  
nunca termina hasta que  
morimos. Y el  
las elecciones que hacemos son  
en última instancia, nuestra  
propia responsabilidad.

---

Eleanor Roosevelt

Machine Translated by Google



## Capítulo 10

### Posfacio

En los veinte años previos a la primera edición, fuimos parte de la evolución de la computadora desde una curiosidad periférica hasta un imperativo moderno para las empresas. En los veinte años transcurridos desde entonces, el software ha crecido más allá de las meras máquinas comerciales y realmente se ha apoderado del mundo. Pero, ¿qué significa eso realmente para nosotros?

En The Mythical Man-Month: Essays on Software Engineering

[Bro96], Fred Brooks dijo: “El programador, como el poeta, trabaja solo ligeramente alejado del pensamiento puro.

Construye sus castillos en el aire, desde el aire, creando mediante el ejercicio de la imaginación”. Comenzamos con una página en blanco y podemos crear prácticamente cualquier cosa que podamos imaginar. Y las cosas que creamos pueden cambiar el mundo.

Desde Twitter que ayuda a las personas a planificar revoluciones, hasta el procesador de su automóvil que trabaja para evitar que derrape, hasta el teléfono inteligente, lo que significa que ya no tenemos que recordar los molestos detalles diarios, nuestros programas están en todas partes. Nuestra imaginación está en todas partes.

Los desarrolladores somos increíblemente privilegiados. Realmente estamos construyendo el futuro. Es una cantidad extraordinaria de poder. Y con eso

El poder conlleva una extraordinaria responsabilidad.

¿Cuántas veces nos detenemos a pensar en eso? ¿Con qué frecuencia discutimos, tanto entre nosotros como con una audiencia más general, lo que esto significa?

Los dispositivos integrados usan un orden de magnitud más de computadoras que las que se usan en computadoras portátiles, computadoras de escritorio y centros de datos. Estas computadoras integradas a menudo controlan sistemas críticos para la vida, desde plantas de energía hasta automóviles y equipos médicos. Incluso un simple sistema de control de calefacción central o un electrodoméstico puede matar a alguien si está mal diseñado o implementado. Cuando desarrollas para estos dispositivos, asumes una responsabilidad asombrosa.

Muchos sistemas no integrados también pueden hacer tanto bien como mucho daño.

Las redes sociales pueden promover la revolución pacífica o fomentar el odio feo. Big data puede facilitar las compras y puede destruir cualquier vestigio de privacidad que pueda pensar que tiene.

Los sistemas bancarios toman decisiones de préstamo que cambian la vida de las personas. Y casi cualquier sistema puede usarse para espiar a sus usuarios.

Hemos visto indicios de las posibilidades de un futuro utópico y ejemplos de consecuencias no deseadas que conducen a distopías de pesadilla. La diferencia entre los dos resultados puede ser más sutil de lo que piensas. Y todo está en tus manos.

## La brújula moral

El precio de este poder inesperado es la vigilancia. Nuestras acciones afectan directamente a las personas. Ya no es el programa de pasatiempos en la CPU de 8 bits en el garaje, el proceso comercial por lotes aislado en el mainframe en el centro de datos, o incluso solo la PC de escritorio; nuestro software teje el tejido mismo de la vida moderna diaria.

Tenemos el deber de hacernos dos preguntas sobre cada pieza de código que entregamos:

1. ¿He protegido al usuario?
2. ¿Usaría esto yo mismo?

Primero, debe preguntar "¿He hecho todo lo posible para proteger a los usuarios de este código de daños?" ¿He hecho provisiones para aplicar parches de seguridad continuos a ese simple monitor de bebé? ¿Me he asegurado de que, aunque falle el termostato de calefacción central automático, el cliente seguirá teniendo el control manual? ¿Estoy almacenando solo los datos que necesito y encriptando algo personal?

Nadie es perfecto; todos extrañan cosas de vez en cuando. Pero si no puede decir con sinceridad que trató de enumerar todas las consecuencias y se aseguró de proteger a los usuarios de ellas, entonces tiene cierta responsabilidad cuando las cosas van mal.

Consejo 98

Primero, no hacer daño

En segundo lugar, hay un juicio relacionado con la regla de oro: ¿estaría feliz de ser un usuario de este software? quiero mis datos

¿compartido? ¿Quiero que mis movimientos sean entregados a puntos de venta? ¿Estaría feliz de ser conducido por este vehículo autónomo? ¿Me siento cómodo haciendo esto?

Algunas ideas ingeniosas comienzan a eludir los límites del comportamiento ético, y si estás involucrado en ese proyecto, eres tan responsable como los patrocinadores. No importa cuántos grados de separación pueda racionalizar, una regla sigue siendo cierta:

Consejo 99

No habilitar Scumbags

## Imagina el futuro que quieres

Tu decides. Es su imaginación, sus esperanzas, sus preocupaciones las que proporcionan el material de pensamiento puro que construye los próximos veinte años y más allá.

Estáis construyendo el futuro, para vosotros y para vuestra descendencia. Tu deber es convertirlo en un futuro en el que todos queramos habitar. Reconoce cuando estás haciendo algo en contra de este ideal y ten el coraje de decir "¡no!" Visualiza el futuro que podríamos tener y ten el coraje de crearlo. Construye castillos en el aire todos los días.

Todos tenemos una vida increíble.

Consejo 100

Es tu vida.  
Compártelo. Celebrarlo. Constrúyelo.  
¡Y DIVERTIRSE!

# Apéndice 1

## Bibliografía

- [ADSS18] Vard Antinyan, Jesper Derehag, Anna Sandberg y Miroslaw Staron. Cobertura de prueba de unidad mítica. *IEEE Software*. 35:73-79, 2018.
- [Y10] Jackie Andrade. ¿Qué hace garabatear? *Psicología Cognitiva Aplicada*. 24(1):100-106, 2010, enero.
- joe armstrong *Programando Erlang: Software para un Mundo Concurrente*. La estantería pragmática, Raleigh, NC, 2007.
- [BR89] Albert J. Bernstein y Sydney Craft Rozen. *Cerebros de dinosaurio: tratar con todas esas personas imposibles en el trabajo*. John Wiley & Sons, Nueva York, NY, 1989.
- [Hermano96] Frederick P. Brooks, Jr. *The Mythical Man-Month: ensayos sobre ingeniería de software*. Addison-Wesley, Reading, MA, Aniversario, 1996.
- [CN91] Brad J. Cox y Andrew J. Novobilski. *Programación orientada a objetos: un enfoque evolutivo*. Addison Wesley, Reading, MA, Segundo, 1991.
- [Con68] Melvin E. Conway. ¿Cómo inventan los comités? *Datamación*. 14(5):28-31, 1968, abril.
- [de 98] Gavin de Becker. *El regalo del miedo: y otras señales de supervivencia que nos protegen de la violencia*. Dell Publishing, Ciudad de Nueva York, 1998.
- [DL13] Tom De Macro y Tim Lister. *Peopleware: Proyectos y Equipos Productivos*. Addison-Wesley, Boston, MA, tercero, 2013.
- Martín Fowler. *UML destilado: una breve guía de Lenguaje estándar de modelado de objetos*. Addison- [Fow00]

- Wesley, Boston, MA, Segundo, 2000.
- Martín Fowler. [UML destilado: una breve guía de \[Fow04\]](#)  
[Lenguaje estándar de modelado de objetos.](#) Addison Wesley, Boston, MA, tercero, 2004.
- [Adelante19] Martín Fowler. [Refactorización: mejora del diseño del código existente.](#) Addison-Wesley, Boston, MA, Segundo, 2019.
- Vlissides. Erich Gamma, Richard Helm, Ralph Johnson y John [GHJV9]  
[Patrones de Diseño: Elementos de Reutilizable 5\]](#)  
[Software Orientado a Objetos.](#) Addison-Wesley, Reading, MA, 1995.
- [Hol92] Michael Holt. [Rompecabezas y juegos matemáticos.](#) Dorset House, Nueva York, NY, 1992.
- andy caza. [Pensamiento pragmático y aprendizaje: \[Hun08\]](#)  
[Refactorice su Wetware. La estantería pragmática,](#) Raleigh, NC, 2008.
- [Joi94] Carpintero TE. Depresión contagiosa: existencia, especificidad de los síntomas depresivos y el papel de la búsqueda de tranquilidad.  
[Revista de Personalidad y Psicología Social.](#)  
 67(2):287-296, 1994, agosto.
- [Knu11] Donald E. Knuth. [El arte de la programación informática, Volumen 4A: Algoritmos combinatorios, Parte 1.](#)  
 Addison-Wesley, Boston, MA, 2011.
- Donald E. Knuth. [El arte de la programación informática, \[Knu98\]](#)  
[Volumen 1: Algoritmos fundamentales.](#) Addison-Wesley, Reading, MA, Tercero, 1998.
- Volumen 2: Donald E. Knuth. [El arte de la programación informática, \[Knu98a\]](#)  
[2: Algoritmos semiméricos.](#) Addison ] Wesley, Reading, MA, Tercero, 1998.
- Volumen 3: Donald E. Knuth. [El arte de la programación informática, \[Knu98b\]](#)  
[clasificación y búsqueda.](#) Addison Wesley, ] Lectura, MA, Segundo, 1998.
- [KP99] Brian W. Kernighan y Rob Pike. [La práctica de la programación.](#) Addison-Wesley, Reading, MA, 1999.
- [Mey97] Bertrand Meyer. [Construcción de Software Orientado a Objetos.](#) Prentice Hall, Upper Saddle River, NJ, Segundo, 1997.
- [Mu18] Jerry Z. Muller. [La tiranía de las métricas.](#) Prensa de la Universidad de Princeton, Princeton NJ, 2018.

- [SF13] Robert Sedgewick y Philippe Flajolet. [Una introducción al análisis de algoritmos](#). Addison-Wesley, Boston, MA, Segundo, 2013.
- [Str35] James Ridley Stroop. Los estudios de interferencia en las reacciones verbales de serie. [Revista de Psicología Experimental](#). 18:643--662, 1935.
- [SW11] Robert Sedgewick y Kevin Wayne. [Algoritmos](#). Addison Wesley, Boston, MA, Cuarto, 2011.
- [Tal10] Nassim Nicolás Taleb. [El Cisne Negro: Segunda Edición: El Impacto de lo Altamente Improbable](#). Random House, Nueva York, NY, Segundo, 2010.
- [WH82] James Q. Wilson y George Helling. La policía y la seguridad vecinal. [El Atlántico Mensual](#). 249[3]:29- -38, 1982, marzo.
- [YC79] Edward Yourdon y Larry L. Constantine. [Diseño estructurado: fundamentos de una disciplina de diseño de sistemas y programas informáticos](#). Prentice Hall, Englewood Cliffs, Nueva Jersey, 1979.
- [Tú95] Eduardo Yourdon. Cuando un software lo suficientemente bueno es lo mejor. [Software IEEE](#). 1995, mayo.

Copyright © 2020 Pearson Educación, Inc.

Yo preferiría tener  
preguntas que no se  
pueden responder que  
respuestas que no pueden ser  
cuestionado

ricardo feynman

## Apéndice 2

# Possible respuestas a la Ejercicios

Respuesta 1 (del ejercicio 1)

A nuestro modo de pensar, la clase [Split2](#) es más ortogonal. Se concentra en su propia tarea, dividir líneas e ignora detalles como de dónde provienen las líneas. Esto no solo hace que el código sea más fácil de desarrollar, sino que también lo hace más flexible. [Split2](#) puede dividir líneas leídas de un archivo, generadas por otra rutina o pasadas a través del entorno.

### Respuesta 2 (del ejercicio 2)

---

Comencemos con una afirmación: puede escribir un buen código ortogonal en casi cualquier idioma. Al mismo tiempo, cada idioma tiene tentaciones: características que pueden conducir a un mayor acoplamiento y una menor orthogonalidad.

En los lenguajes orientados a objetos, 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. También hay una especie de acoplamiento porque una clase acopla código a datos. Esto normalmente es algo bueno (cuando el acoplamiento es bueno, lo llamamos cohesión). Pero si no hace que sus clases estén lo suficientemente enfocadas, puede llevar a algunas interfaces bastante desagradables.

En los lenguajes funcionales, se le recomienda que escriba muchas funciones pequeñas y desacopladas y que las combine de diferentes maneras para resolver su problema. En teoría, esto suena bien. En la práctica, a menudo lo es. Pero hay una forma de acoplamiento que también puede ocurrir aquí. Estas funciones suelen transformar datos, lo que significa que el resultado de una función puede convertirse en la entrada de otra. Si no tiene cuidado, hacer un cambio en el formato de datos que genera una función puede provocar una falla en algún lugar del flujo de transformación. Los lenguajes con buenos sistemas tipográficos pueden ayudar a mitigar esto.

### Respuesta 3 (del ejercicio 3)

¡Baja tecnología al rescate! Dibuje algunas caricaturas con marcadores en una pizarra: un automóvil, un teléfono y una casa. No tiene que ser un gran arte; los contornos de figuras de palo 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.

### Respuesta 4 (del ejercicio 4)

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.

lang/tortuga.c

```

estructura
typedef { char cmd; /* la letra de comando */
    int hasArg; /* requiere un argumento */ void (*func)
    (int, int); /* rutina a llamar */
} Dominio;

Comando estático cmds[] = {
    { 'P', ARG, doSelectPen },
    { 'U', NO_ARG, doPenUp },
    { 'D', NO_ARG, doPenDown },
    { 'N', ARG, doPenDir },
    { 'E', ARG, doPenDir },
    { 'S', ARG, doPenDir },
    { 'W', ARG, doPenDir };

```

El programa principal es bastante simple: lea una línea, busque el comando, obtenga el argumento si es necesario, luego llame al controlador

función.

lang/tortuga.c

```
while (fgets(buff, sizeof(buff), stdin)) {

    Comando *cmd = findCommand(*buff);

    si (cmd)
    { int arg = 0;

        if (cmd->hasArg && !getArg(buff+1, &arg))
        { fprintf(stderr, "%c necesita un argumento\n", *buff);
        continuar; }

        cmd->func(*buff, arg); } }
```

La función que busca un comando realiza una búsqueda lineal de la tabla y devuelve la entrada coincidente o NULL.

lang/tortuga.c

```
Comando *findCommand(int cmd) { int i;

for (i = 0; i < ARRAY_SIZE(cmds); i++) { if
(cmds[i].cmd == cmd) return
cmds + i;
}

fprintf(stderr, "Comando desconocido '%c'\n", cmd);
devolver
0; }
```

Finalmente, leer el argumento numérico es bastante simple usando escanear

lang/tortuga.c

```
int getArg(const char *buff, int *result)
{ return sscanf(buff, "%d", resultado) ==
1; }
```

### Respuesta 5 (del ejercicio 5)

En realidad, ya resolvió este problema en el ejercicio anterior, donde escribió un intérprete para el idioma externo, contendrá el intérprete interno. En el caso de nuestro código de muestra, estas son las funciones [doXxx](#).

### Respuesta 6 (del ejercicio 6)

Usando BNF, una especificación de tiempo podría ser

tiempo ::= hora amperio | hora : minuto ampm | hora : minuto

ampm ::= am | pm

hora ::= dígito | dígito dígito

minuto ::= dígito dígito

dígito ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Una mejor definición de hora y minuto tendría en cuenta que una hora solo puede ser de 00 a 23, y un minuto de 00 a 59:

|           |     |                                       |
|-----------|-----|---------------------------------------|
| hora      | ::= | dígito h-decenas   dígito             |
| minuto    | ::= | dígito m-decenas                      |
| h-decenas | ::= | 0   1                                 |
| m-decenas | ::= | 0   1   2   3   4   5                 |
| dígito    | ::= | 0   1   2   3   4   5   6   7   8   9 |

### Respuesta 7 (del ejercicio 7)

Aquí está el analizador escrito usando la biblioteca JavaScript de Pegjs:

lang/peg\_parser/time\_parser.pegjs

```
time
= h:hora offset:ampm { return h + offset } / h:hora ":" 
m:minuto offset:ampm { return h + m + offset } / h:hora ":" m:minuto
{ return h + m }

ampm
= "am" {retornar 0} /
"pm" {retornar 12*60}

hora
= h:dos_dígitos_hora { return h*60 } /
h:dígito { return h*60 }
```

```

minuto
= d1:[0-5] d2:[0-9] { return parseInt(d1+d2, 10); }

dígito = dígito:[0-9] { return parseInt(dígito, 10); }

dos_dígitos_hora
= d1:[01] d2:[0-9] { return parseInt(d1+d2, 10); } / d1:
[2] d2:[0-3] { return parseInt(d1+d2, 10); }

```

Las pruebas lo muestran en uso:

### lang/peg\_parser/test\_time\_parser.js

```

let test = require('cinta');
let time_parser = require('./time_parser.js');

// hora ::= hora
//         | hora : minuto ampm |
//         hora : minuto

ampm // // // // ampm ::=
am | pm // // hora ::= dígito |

dígito dígito // // minuto ::=
dígito dígito // // dígito ::= 0 |1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

const h = (val) => val*60;
const m = (valor) =>
valor; const am = (val)
=> val; const pm = (val) => val + h(12);

dejar pruebas =

```

- "1am": h(1),
- "1pm": pm(h(1)),
- "2:30": h(2) + m(30),
- "14:30": p.m.(h(2)) + m(30),
- "14:30": p.m.(h(2)) + m (30),

```
}
```

```
test('análisis de tiempo', function (t)
```

```
{ for ( cadena constante en
```

```
pruebas) { let result =
```

```
time_parser.parse(cadena) t.equal(result,
```

```
tests[string], string); } t.end() } );
```

## Respuesta 8 (del ejercicio 8)

Aquí hay una posible solución en Ruby:

## lang/re\_parser/time\_parser.rb

```
TIME_RE = %r{ (?  
<dígito>[0-9])  
<h_diez>[0-1]  
<m_diez>[0-6]  
ampm> am | pm  
<hora> (\g<h_diez> \g<dígito>) | \g<dígito>{0} (?<minuto>  
\g<m_diez> \g<dígito >){0}  
  
\A( ( \g<hora> \g<amperios> )  
| ( \g<hora> : \g<minuto> \g<amperios> ) |  
( \g<hora> : \g<minuto> )  
)Z  
}X  
  
def parse_time(cadena)  
resultado = TIME_RE.match(cadena)  
if  
    resultado resultado[:hora].to_i  
    * 60 + (resultado[:minuto] ||  
    "0").to_i + (resultado[:ampm] == "pm" ?  
    12*60 : 0) fin fin
```

(Este código utiliza el truco de definir patrones con nombre al comienzo de la expresión regular y luego hacer referencia a ellos como subpatrones en la coincidencia real).

#### Respuesta 9 (del ejercicio 9)

Nuestra respuesta debe basarse en varios supuestos:

- El dispositivo de almacenamiento contiene la información que necesitamos transferir.
- 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 el dispositivo de almacenamiento.
- La sobrecarga de almacenar datos es aproximadamente igual a la sobrecarga de enviarlos a través de una línea de comunicaciones.

#### Respuesta 10 (del ejercicio 10)

Sujeto a las advertencias de la respuesta anterior: una cinta de 1 TGB contiene  $8 \times 2^{40}$ , o 2<sup>43</sup> bits, por lo que una línea de 1 Gbps tendría que bombejar datos durante aproximadamente 9000 segundos, o aproximadamente 2½ horas, para transferir la cantidad equivalente de información. Si la persona camina a una velocidad constante de 3 ½ mph, nuestras dos máquinas tendrían que estar separadas casi 9 millas para que la línea de comunicaciones supere a nuestro servicio de mensajería. De lo contrario, la persona gana.

#### Respuesta 14 (del ejercicio 14)

Mostraremos las firmas de funciones en Java, con las condiciones previas y posteriores en los comentarios.

Primero, el invariante para la clase:

```
/***
 * @invariant getSpeed() > 0
 *      implica isFull()           // No te quedes vacío
 *
 * @invariant 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 // Comprobación de rango * @post
 * getSpeed() == x / Respetar la velocidad solicitada */
public void setSpeed(final int x)

/***
 * @pre !isFull() *                  // No lo llenes dos veces
 * @post isFull() */                // Asegurarse de que se haya hecho

void fill()

/***
 * @pre isFull() *                  // No lo vacíes dos veces
 * @post !isFull() */              // Asegurarse de que se haya hecho

void vacío()
```

Respuesta 15 (del ejercicio 15)

Hay 21 términos en la serie. Si dijo 20, acaba de experimentar un error de poste de la cerca (sin saber si contar los postes de la cerca o los espacios entre ellos).

Respuesta 16 (del ejercicio 16)

- Septiembre de 1752 tuvo sólo 19 días. Esto se hizo para sincronizar

calendarios como parte de la Reforma Gregoriana.

- El directorio podría haber sido eliminado por otro proceso, es posible que no tenga permiso para leerlo, es posible que la unidad no esté montada, ...; te dan la imagen.
- Disimuladamente no especificamos los tipos de `a` y `b`. La sobrecarga de operadores podría haber definido `+`, `=` o `! =` para tener un comportamiento inesperado. Además, `a` y `b` pueden ser alias para la misma variable, por lo que la segunda asignación sobrescribirá el valor almacenado en la primera. Además, si el programa es concurrente y está mal escrito, es posible que se haya actualizado para cuando se realice la adición.
- En la geometría no euclíadiana, la suma de los ángulos de un triángulo no suman  $180^\circ$ . Piense en un triángulo trazado en la superficie de una esfera.
- Los minutos bisiestos pueden tener 61 o 62 segundos.
- Según el idioma, el desbordamiento numérico puede dejar el resultado de `a+1` negativo.

### Respuesta 17 (del ejercicio 17)

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 configurar el puntero en `NULL`, los programadores esperan evitar estas referencias no autorizadas; en la mayoría de los casos, eliminar la referencia de un puntero `NULL` generará un error de tiempo de ejecución.

### Respuesta 18 (del ejercicio 18)

Al establecer la referencia en `NULL`, reduce el número de punteros al objeto al que se hace referencia en uno. Una vez que esto cuenta

llega a cero, el objeto es elegible para la recolección de elementos no utilizados. Establecer las referencias en `NULL` puede ser importante para los programas de ejecución prolongada, en los que los programadores deben asegurarse de que la utilización de la memoria no aumente con el tiempo.

### Respuesta 19 (del ejercicio 19)

Una implementación simple podría ser:

evento/cadenas\_ex\_1.rb

```
class FSM
  def initialize(transiciones, estado_inicial)
    @transiciones = transiciones
    @estado = estado_inicial
  end
  def accept(evento)
    @estado, acción = TRANSICIONES[@estado][evento] ||
      TRANSICIONES[@estado][:predeterminado]
  end
end
```

(Descargue este archivo para obtener el código actualizado que usa esta nueva clase `FSM`).

### Respuesta 20 (del ejercicio 20)

- ...tres eventos de inactividad de la interfaz de red en cinco minutos Esto podría implementarse usando una máquina de estado, pero sería más complicado de lo que parece a primera vista: si obtiene eventos en los minutos 1, 4, 7 y 8, entonces debería activar la advertencia en el cuarto evento, lo que significa que la máquina de estado debe poder manejar el reinicio.

Por esta razón, los flujos de eventos parecen ser la tecnología preferida. Hay una función reactiva llamada `buffer` con parámetros `de tamaño` y `compensación` que le permitiría devolver cada grupo de tres eventos entrantes. A continuación, puede consultar las marcas de tiempo del primer y último evento de un grupo para determinar si se debe activar la alarma.

- ...después de la puesta del sol, y se detecta movimiento en la parte inferior de las escaleras seguido de movimiento detectado en la parte superior de las escaleras...

Esto probablemente podría implementarse usando una combinación de pubsub y máquinas de estado. Puede usar pubsub para diseminar eventos a cualquier número de máquinas de estado y luego hacer que las máquinas de estado determinen qué hacer.

- ...notifica a varios sistemas de informes que se completó un pedido.

Esto probablemente se maneja mejor usando pubsub. Es posible que desee utilizar secuencias, pero eso requeriría que los sistemas a los que se notifique también se basen en secuencias.

- ...tres servicios backend y espera las respuestas.

Esto es similar a nuestro ejemplo que usó flujos para obtener datos de usuario.

## Respuesta 21 (del ejercicio 21)

1. Los impuestos de envío y ventas se agregan a un pedido:

pedido básico → pedido finalizado

En el código convencional, es probable que tenga una función que calcule los costos de envío y otra que calcule los impuestos. Pero aquí estamos pensando en transformaciones, por lo que transformamos un pedido con solo artículos en un nuevo tipo de cosa: un pedido que se puede enviar.

2. Su aplicación carga información de configuración de un nombre archivo:

nombre de archivo → estructura de configuración

3. Alguien inicia sesión en una aplicación web:

credenciales de usuario → sesión

## Respuesta 22 (del ejercicio 22)

La transformación de alto nivel:

contenido del campo como cadena

```
→ [validar y convertir] →
{:ok, valor} | {:error, motivo}
```

podría desglosarse en:

```
contenido del campo
como cadena → [convertir
cadena a entero] →
[verificar valor >= 18] →
[verificar valor <= 150] → {:ok, valor} | {:error, motivo}
```

Esto supone que tiene una canalización de manejo de errores.

Respuesta 23 (del ejercicio 23)

Respondamos primero a la segunda parte: preferimos la primera pieza de código.

En el segundo fragmento de código, cada paso devuelve un objeto que implementa la siguiente función que llamamos: el objeto devuelto por `content_of` debe implementar `find_matching_lines`, y así sucesivamente.

Esto significa que el objeto devuelto por `content_of` está acoplado a nuestro código. Imagine que el requisito cambió y tenemos que ignorar las líneas que comienzan con un carácter `#`. En el estilo de transformación, eso sería fácil:

```
const contenido = Archivo.leer(nombre_archivo);
const no_comments = remove_comments(contenido)
const lines = find_matching_lines(no_comments, pattern) const result
= truncate_lines(lines)
```

Podríamos intercambiar el orden de `remove_comments` y `find_matching_lines` y aún funcionaría.

Pero en el estilo encadenado, esto sería más difícil. ¿Dónde debería vivir nuestro método `remove_comments` : en el objeto devuelto por

`content_of` o el objeto devuelto por `find_matching_lines?` ¿Y qué otro código romperemos si cambiamos ese objeto? Este acoplamiento es la razón por la que el estilo de encadenamiento de métodos a veces se denomina choque de trenes.

### Respuesta 24 (del ejercicio 24)

Procesamiento de imágenes.

Para la programación simple de una carga de trabajo entre los procesos paralelos, una cola de trabajo compartida puede ser más que adecuada. Es posible que desee considerar un sistema de pizarra si hay comentarios involucrados, es decir, si los resultados de un fragmento procesado afectan a otros fragmentos, como en aplicaciones de visión artificial o transformaciones de deformación de imágenes 3D complejas.

Calendario de grupo

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 venir y ir.

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

necesidad de 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 constriñe artificialmente.

#### Herramienta de monitoreo

de red Esto es muy similar al programa de solicitud de hipoteca/préstamo. Tiene informes de problemas enviados por 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 solo 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.

#### Respuesta 25 (del ejercicio 25)

La suposición con una lista de pares clave-valor es generalmente que la clave es única, y las bibliotecas hash normalmente lo imponen por el comportamiento del hash en sí o con mensajes de error explícitos para claves duplicadas. Sin embargo, una matriz generalmente no tiene esas restricciones y felizmente almacenará claves duplicadas a menos que lo codifique específicamente para que no lo haga. Entonces, en este caso, la primera clave encontrada que coincide con **DepositAccount** gana, y las entradas coincidentes restantes se ignoran. El orden de las entradas no está garantizado, por lo que a veces funciona ya veces no.

¿Y qué hay de la diferencia en las máquinas de desarrollo y producción?  
Es solo una coincidencia.

#### Respuesta 26 (del ejercicio 26)

El hecho de que un campo puramente numérico funcione en EE. UU., Canadá y el Caribe es una coincidencia. Según las especificaciones de la UIT, el formato de llamada internacional comienza con un signo + literal . El carácter \* también se usa en algunos lugares y, más comúnmente, los ceros iniciales pueden ser parte del número. Nunca almacene un número de teléfono en un campo numérico.

Respuesta 27 (del ejercicio 27)

Depende de donde estés. En los EE. UU., las medidas de volumen se basan en el galón, que es el volumen de un cilindro de 6 pulgadas de alto y 7 pulgadas de diámetro, redondeado a la pulgada cúbica más cercana.

En Canadá, "una taza" en una receta podría significar cualquiera de

- 1/5 de cuarto imperial, o 227ml
- 1/4 de cuarto de galón estadounidense, o 236 ml
- 16 cucharadas métricas, o 240ml
- 1/4 de litro, o 250ml

A menos que esté hablando de una olla arrocera, en cuyo caso "una taza" son 180 ml. Eso se deriva del koku, que era el volumen estimado de arroz seco necesario para alimentar a una persona durante un año: aparentemente, alrededor de 180 litros. Las tazas de la olla arrocera son 1 gō, que es 1/1000 de un koku. Entonces, aproximadamente la cantidad de arroz que una persona comería en una sola comida.

[85]

Respuesta 28 (del ejercicio 28)

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 si los procesos en segundo plano le quitan periódicamente ciclos a sus programas. También es posible que desee verificar la memoria: si la aplicación comienza a usar el espacio de intercambio, el rendimiento se desplomará.

Aquí hay un gráfico de los resultados de ejecutar el código en una de nuestras máquinas:

imágenes/alg-speed-rust-results.png

### Respuesta 29 (del ejercicio 29)

Hay un par de maneras de llegar allí. Una es darle la vuelta al problema. 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 donde está el número de iteraciones.

,

Si lleva registros a la base 2 de cada lado, obtiene  , que por definición de registros se convierte en

### Respuesta 30 (del ejercicio 30)

Probablemente esto sea demasiado como un recuerdo de las matemáticas de la escuela secundaria, pero la fórmula para convertir un logaritmo en  base a uno en base  es:

Debido a que  es una constante, podemos ignorarla dentro de un resultado Big-O.

### Respuesta 31 (del ejercicio 31)

Una propiedad que podemos probar es que un pedido tiene éxito si el almacén tiene suficientes artículos disponibles. Podemos generar pedidos para cantidades aleatorias de artículos y verificar que se devuelva una tupla "OK" si el almacén tenía stock.

### Respuesta 32 (del ejercicio 32)

Este es un buen uso de las pruebas basadas en propiedades. Las pruebas unitarias pueden centrarse en casos individuales en los que ha obtenido el resultado

algunos otros medios, y las pruebas de propiedad pueden enfocarse en cosas como:

- ¿Se superponen dos cajas?
- ¿Alguna parte de alguna caja excede el ancho o largo del camión?
- ¿La densidad de empaque (área utilizada por las cajas dividida por el área de la caja del camión) es menor o igual a 1?
- Si es parte del requisito, ¿la densidad de empaque excede la densidad mínima aceptable?

### Respuesta 33 (del ejercicio 33)

1. Esta declaración suena como un requisito real: puede haber restricciones impuestas a la aplicación por su entorno.
2. Por sí sola, esta declaración no es realmente un requisito. Pero para saber lo que realmente se requiere, debe hacer la pregunta mágica: "¿Por qué?"  
Puede ser que se trate de un estándar corporativo, en cuyo caso el requisito real debería ser algo así como "todos los elementos de la interfaz de usuario deben cumplir con los estándares de interfaz de usuario de MegaCorp, V12.76".  
Puede ser que este sea un color que le guste al equipo de diseño. En ese caso, debe pensar en la forma en que al equipo de diseño también le gusta cambiar de opinión y expresar el requisito como "el color de fondo de todas las ventanas modales debe ser configurable. 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".

O simplemente puede significar que el usuario necesita poder distinguir ventanas modales y no modales. Si ese es el caso, se necesitan más discusiones.

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. ¿Es esto un problema de escala? ¿O el rendimiento? ¿Costo?  
¿Seguridad? Las respuestas informarán su diseño.
4. El requisito subyacente es probablemente algo más cercano a "El sistema evitará que el usuario realice entradas no válidas en los campos,

y advertirá al usuario cuando se realicen estas entradas." 5. Esta declaración es probablemente un requisito difícil, basado en algunos limitación de hardware.

Y aquí hay una solución al problema de los cuatro puntos:



---

notas al pie

[85] Gracias por esta trivia para Avi Bryant (@avibryant)