

Robert C. Martin Series

Clean Architecture

A Craftsman's Guide to
Software Structure and Design

Robert C. Martin

With contributions by James Grenning and Simon Brown

Foreword by Kevlin Henney

Afterword by Jason Gorman



Acerca de este libro electrónico

EPUB es un formato abierto estándar de la industria para libros electrónicos. Sin embargo, la compatibilidad con EPUB y sus numerosas funciones varía según los dispositivos y aplicaciones de lectura. Utilice la configuración de su dispositivo o aplicación para personalizar la presentación a su gusto. Las configuraciones que puede personalizar a menudo incluyen fuente, tamaño de fuente, columna simple o doble, modo horizontal o vertical y figuras en las que puede hacer clic o tocar para ampliar. Para obtener información adicional sobre la configuración y las funciones de su dispositivo o aplicación de lectura, visite el sitio web del fabricante del dispositivo.

Muchos títulos incluyen código de programación o ejemplos de configuración. Para optimizar la presentación de estos elementos, vea el libro electrónico en modo horizontal de una sola columna y ajuste el tamaño de fuente al mínimo. Además de presentar el código y las configuraciones en formato de texto ajustable, hemos incluido imágenes del código que imitan la presentación que se encuentra en el libro impreso; por lo tanto, cuando el formato ajustable pueda comprometer la presentación de la lista de códigos, verá el enlace "Haga clic aquí para ver la imagen del código". Haga clic en el enlace para ver la imagen del código de fidelidad de impresión. Para regresar a la página anterior vista, haga clic en el botón Atrás en su dispositivo o aplicación.

Robert C. Martin Series



Visit informit.com/martinseries for a complete list of available publications.

The **Robert C. Martin Series** is directed at software developers, team-leaders, business analysts, and managers who want to increase their skills and proficiency to the level of a Master Craftsman. The series contains books that guide software professionals in the principles, patterns, and practices of programming, software project management, requirements gathering, design, analysis, testing, and others.



Make sure to connect with us!
informit.com/socialconnect



informIT.com

the trusted technology learning source



Arquitectura limpia

UNA GUÍA PARA EL ARTESANO SOBRE LA ESTRUCTURA DEL SOFTWARE
Y DISEÑO

Roberto C. Martín 
PRENTICE
HALL

Boston • Columbus • Indianápolis • Nueva York • San Francisco • Ámsterdam •
Ciudad del Cabo Dubái • Londres • Madrid • Milán • Múnich • París • Montreal •
Toronto • Delhi • Ciudad de México São Paulo • Sídney • Hong Kong • Seúl •
Singapur • Taipei • Tokio

Muchas de las designaciones utilizadas por fabricantes y vendedores para distinguir sus productos se consideran marcas comerciales. Cuando esas designaciones aparecen en este libro y el editor tenía conocimiento de un reclamo de marca registrada, las designaciones se imprimieron con letras mayúsculas iniciales o en mayúsculas.

El autor y el editor han tenido cuidado en la preparación de este libro, pero no ofrecen garantía expresa o implícita de ningún tipo y no asumen responsabilidad por errores u omisiones. No se asume ninguna responsabilidad por daños incidentales o consecuentes relacionados 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 corporativo, en corpsales@pearsoned.com o (800) 382-3419.

Para consultas sobre ventas gubernamentales, comuníquese con gobiernosales@pearsoned.com.

Si tiene preguntas sobre ventas fuera de EE. UU., comuníquese con intlcs@pearson.com

Visítanos en la Web: informit.com

Número de control de la Biblioteca del Congreso: 2017945537

Copyright © 2018 Pearson Education, Inc.

Reservados todos los derechos. Impreso en los Estados Unidos de América. Esta publicación está protegida por derechos de autor y se debe obtener permiso del editor antes de cualquier reproducción prohibida, almacenamiento en un sistema de recuperación o transmisión en cualquier forma o por cualquier medio, ya sea electrónico, mecánico, fotocopia, 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: 978-0-13-449416-6

ISBN-10: 0-13-449416-4

1 17

Este libro está dedicado a mi encantadora esposa, a mis cuatro espectaculares hijos y a sus familias, incluido mi carcaj lleno de cinco nietos, que son el postre de mi vida.

CONTENIDO

[Prefacio](#)

[Prefacio](#)

[Expresiones de gratitud](#)

[Sobre el Autor](#)

[PARTE I Introducción](#)

[Capítulo 1 ¿Qué es el diseño y la arquitectura?](#)

[¿La meta?](#)

[Conclusión del
estudio de caso](#)

[Capítulo 2 Una historia de dos valores](#)

[Comportamiento](#)

[Arquitectura](#)

[El mayor valor](#)

[La matriz de Eisenhower](#)

[Lucha por la arquitectura](#)

[PARTE II Comenzando con los ladrillos: paradigmas de programación](#)

[Capítulo 3 Descripción general del paradigma](#)

[Programación estructurada](#)

[Programación orientada a objetos](#)

[Programación funcional](#)

[Comida para el pensamiento](#)

Conclusión

Capítulo 4 Programación estructurada

Prueba

Una proclamación dañina

Descomposición funcional

Sin pruebas formales

Ciencia al rescate

Pruebas

Conclusión

Capítulo 5 ¿Encapsulación de programación orientada a

objetos ?

¿Herencia?

¿Polimorfismo?

Conclusión

Capítulo 6 Programación funcional

Cuadrados de números enteros

Inmutabilidad y arquitectura

Segregación de mutabilidad

Abastecimiento de eventos

Conclusión

PARTE III Principios de diseño

Capítulo 7 SRP: El principio de responsabilidad única

Síntoma 1: duplicación accidental

Síntoma 2: fusiones

Soluciones

Conclusión

Capítulo 8 OCP: El principio abierto-cerrado

Un experimento mental

Control direccional

Ocultación de información

Conclusión

[Capítulo 9 LSP: El principio de sustitución de Liskov](#)

[Orientar el uso de la herencia](#)

[El problema del cuadrado/rectángulo](#)

[LSP y Arquitectura](#)

[Ejemplo de infracción de LSP](#)

[Conclusión](#)

[Capítulo 10 ISP: El principio de segregación de interfaces](#)

[ISP y idioma](#)

[ISP y arquitectura](#)

[Conclusión](#)

[Capítulo 11 DIP: El principio de inversión de dependencia](#)

[Abstracciones estables](#)

[Fábricas](#)

[Componentes de hormigón](#)

[Conclusión](#)

[PARTE IV Principios de los componentes](#)

[Capítulo 12 Componentes](#)

[Una breve historia de los componentes](#)

[Reubicación](#)

[Enlazadores](#)

[Conclusión](#)

[Capítulo 13 Cohesión de componentes](#)

[El principio de equivalencia de reutilización/liberación](#)

[El principio de cierre común](#)

[El principio de reutilización común](#)

[El diagrama de tensión para la cohesión de los componentes](#)

[Conclusión](#)

[Capítulo 14 Acoplamiento de componentes](#)

[El principio de dependencias acíclicas](#)

[Diseño de arriba hacia abajo](#)

[El principio de dependencias estables](#)

[El principio de abstracciones estables](#)

[Conclusión](#)

[PARTE V Arquitectura](#)

[Capítulo 15 ¿Qué es la arquitectura?](#)

[Desarrollo](#)

[Implementación](#)

[Operación](#)

[Mantenimiento](#)

[Mantener las opciones abiertas](#)

[Independencia del dispositivo](#)

[Correo](#)

[basura Direcciónamiento](#)

[físico Conclusión](#)

[Capítulo 16 Independencia](#)

[Casos de uso](#)

[Operación](#)

[Desarrollo](#)

[Despliegue](#)

[Dejar opciones abiertas](#)

[Capas de desacoplamiento](#)

[Casos de uso de desacoplamiento](#)

[Modo de desacoplamiento](#)

[Capacidad de desarrollo independiente](#)

[Implementación independiente](#)

[Duplicación](#)

[Modos de desacoplamiento \(nuevamente\)](#)

[Conclusión](#)

[Capítulo 17 Límites: Dibujar líneas](#)

[Un par de historias tristes](#)

[FitNesse](#)

[¿Qué líneas dibujas y cuándo las dibujas?](#)

[¿Qué pasa con la entrada y la salida?](#)

[Arquitectura del complemento](#)

[El argumento del complemento](#)

Conclusión

Capítulo 18 Anatomía de límites

Cruce de límites

El temido monolito

Componentes de implementación

Hilos

Procesos Locales

Servicios

Conclusión

Capítulo 19 Política y nivel

Nivel

Conclusión

Capítulo 20 Reglas comerciales

Entidades

Casos de uso

Modelos de solicitud y respuesta

Conclusión

Capítulo 21 Arquitectura que grita El tema

de una arquitectura El propósito

de una arquitectura Pero ¿qué

pasa con la Web?

Los frameworks son herramientas, no formas de

vida. Arquitecturas

comprobables Conclusión

Capítulo 22 La arquitectura limpia

La regla de dependencia

Un escenario típico

Conclusión

Capítulo 23 Presentadores y objetos humildes

El patrón del objeto humilde

Presentadores y vistas

Pruebas y arquitectura

Puertas de enlace de bases de datos

Mapeadores de datos

Oyentes de servicio

Conclusión

Capítulo 24 Límites parciales

Saltar el último paso

Límites unidimensionales

Fachadas

Conclusión

Capítulo 25 ¿Capas y límites cazan la arquitectura

Limpia de Wumpus?

Cruzando las corrientes

Dividiendo las corrientes

Conclusión

Capítulo 26 El componente principal

El detalle definitivo

Conclusión

Capítulo 27 Servicios: ¿Arquitectura de servicios

grande y pequeña?

¿Beneficios del servicio?

The Kitty Problem se

opone a la conclusión de las

preocupaciones transversales sobre

los servicios basados en

componentes de rescate

Capítulo 28 El límite de la prueba

Pruebas como componentes del sistema

Diseño para la capacidad de prueba

La API de prueba

Conclusión

Capítulo 29 Arquitectura integrada limpia

Prueba de aptitud para la aplicación

El cuello de botella del hardware objetivo

Conclusión

PARTE VI Detalles

Capítulo 30 La base de datos es un detalle Bases

de datos relacionales ¿Por

qué son tan frecuentes los sistemas de bases de datos?

¿Qué pasaría si no hubiera disco?

Detalles

¿Pero qué pasa con el rendimiento?

Conclusión

de la anécdota

Capítulo 31 La Web es un detalle

El péndulo sin fin

El resultado

Conclusión

Capítulo 32 Los marcos son detalles

Autores del marco

Matrimonio asimétrico

Los riesgos

La solución

Ahora te declaro

...

Conclusión

Capítulo 33 Estudio de caso: Ventas de videos

El producto

Análisis de casos de uso

Arquitectura de componentes

Gestión de dependencias

Conclusión

Capítulo 34 El capítulo perdido

[Paquete por capa](#)
[Paquete por característica](#)
[Puertos y adaptadores](#)
[Paquete por componente](#)
[El diablo está en los detalles de implementación](#)
[Organización versus encapsulación](#)
[Otros modos de desacoplamiento](#)
[Conclusión: el consejo que falta](#)

[PARTE VII Apéndice](#)

[Apéndice A Arquitectura Arqueología](#)

[Índice](#)

PREFACIO

¿De qué hablamos cuando hablamos de arquitectura?

Como ocurre con cualquier metáfora, describir el software a través del lente de la arquitectura puede ocultar tanto como revelar. Puede prometer más de lo que puede cumplir y cumplir más de lo que promete.

El atractivo obvio de la arquitectura es la estructura, y la estructura es algo que domina los paradigmas y discusiones del desarrollo de software: componentes, clases, funciones, módulos, capas y servicios, micro o macro.

Pero la estructura general de tantos sistemas de software a menudo desafía la creencia o la comprensión: esquemas empresariales soviéticos destinados a ser heredados, improbables torres Jenga que se extienden hacia la nube, capas arqueológicas enterradas en una gran bola de barro. No es obvio que la estructura del software obedezca a nuestra intuición como lo hace la estructura de construcción.

Los edificios tienen una estructura física obvia, ya sea que estén arraigados en piedra u hormigón, ya sean arqueados o extendidos, grandes o pequeños, magníficos o mundanos. Sus estructuras no tienen más remedio que respetar la física de la gravedad y sus materiales. Por otro lado, excepto en su sentido de seriedad, el software tiene poco tiempo para la gravedad. ¿Y de qué está hecho el software? A diferencia de los edificios, que pueden estar hechos de ladrillos, hormigón, madera, acero y vidrio, el software está hecho de software. Las construcciones de software grandes están hechas de componentes de software más pequeños, que a su vez están hechos de componentes de software aún más pequeños, y así sucesivamente. Está codificando tortugas hasta el final.

Cuando hablamos de arquitectura de software, el software es de naturaleza recursiva y fractal, grabado y esbozado en código. Todo son detalles. Niveles entrelazados de

Los detalles también contribuyen a la arquitectura de un edificio, pero no tiene sentido hablar de escala física en el software. El software tiene estructura (muchas estructuras y muchos tipos de estructuras), pero su variedad eclipsa la gama de estructuras físicas que se encuentran en los edificios. Incluso se puede argumentar de manera bastante convincente que hay más actividad de diseño y enfoque en el software que en la arquitectura de edificios; en este sentido, no es descabellado considerar la arquitectura de software más arquitectónica que la arquitectura de edificios.

Pero la escala física es algo que los humanos comprenden y buscan en el mundo. Aunque atractivos y visualmente obvios, los cuadros de un diagrama de PowerPoint no son la arquitectura de un sistema de software. No hay duda de que representan una visión particular de una arquitectura, pero confundir las cajas con el panorama general (con la arquitectura) es perder el panorama general y la arquitectura: la arquitectura de software no se parece a nada. Una visualización particular es una elección, no un hecho. Es una elección basada en un conjunto adicional de opciones: qué incluir; qué excluir; qué enfatizar por forma o color; qué restar importancia mediante uniformidad u omisión. No hay nada natural o intrínseco en una visión sobre otra.

Aunque puede que no tenga sentido hablar de física y escala física en la arquitectura de software, apreciamos y nos preocupamos por ciertas limitaciones físicas. La velocidad del procesador y el ancho de banda de la red pueden dar un veredicto severo sobre el rendimiento de un sistema. La memoria y el almacenamiento pueden limitar las ambiciones de cualquier código base. El software puede ser el material con el que se fabrican los sueños, pero se ejecuta en el mundo físico.

Ésta es la monstruosidad del amor, señora, que la voluntad es infinita y la ejecución limitada; que el deseo es ilimitado y el acto esclavo del límite.

-William Shakespeare

El mundo físico es donde vivimos nosotros, nuestras empresas y nuestras economías. Esto nos da otra calibración mediante la cual podemos entender la arquitectura del software, otras fuerzas y cantidades menos físicas a través de las cuales podemos hablar y razonar.

La arquitectura representa las decisiones de diseño importantes que dan forma a un sistema, donde lo significativo se mide por el costo del cambio.

—Grady Booch

El tiempo, el dinero y el esfuerzo nos dan un sentido de escala para clasificar entre lo grande y lo pequeño, para distinguir el material arquitectónico del resto. Esta medida también nos dice cómo podemos determinar si una arquitectura es buena o no: una buena arquitectura no sólo satisface las necesidades de sus usuarios, desarrolladores y propietarios en un momento dado.

en un momento dado, pero también los cumple a lo largo del tiempo.

Si cree que la buena arquitectura es cara, pruebe con la mala arquitectura.

—Brian Foote y Joseph Yoder

Los tipos de cambios que normalmente experimenta el desarrollo de un sistema no deberían ser los cambios que son costosos, que son difíciles de realizar, que requieren proyectos gestionados propios en lugar de incorporarse al flujo de trabajo diario y semanal.

Ese punto nos lleva a un problema no tan pequeño relacionado con la física: el viaje en el tiempo. ¿Cómo sabemos cuáles serán esos cambios típicos para que podamos dar forma a esas decisiones importantes en torno a ellos? ¿Cómo podemos reducir el esfuerzo y los costos de desarrollo futuro sin bolas de cristal ni máquinas del tiempo?

La arquitectura son las decisiones que desearía poder acertar al principio de un proyecto, pero que no necesariamente es más probable que las acierte que cualquier otra.

—Ralph Johnson

Comprender el pasado ya es bastante difícil; nuestra comprensión del presente es, en el mejor de los casos, resbaladiza; predecir el futuro no es trivial.

Aquí es donde el camino se bifurca en muchos sentidos.

Por el camino más oscuro surge la idea de que la arquitectura fuerte y estable proviene de la autoridad y la rigidez. Si el cambio es costoso, se elimina: sus causas se atenúan o se dirigen a una fosa burocrática. El mandato del arquitecto es total y totalitario, y la arquitectura se convierte en una distopía para sus desarrolladores y una fuente constante de frustración para todos.

Por otro camino llega un fuerte olor a generalidad especulativa. Una ruta llena de conjeturas codificadas, innumerables parámetros, tumbas de código muerto y más complejidad accidental de la que se puede imaginar con un presupuesto de mantenimiento.

El camino que más nos interesa es el más limpio. Reconoce la suavidad del software y pretende preservarlo como una propiedad de primera clase del sistema. Reconoce que operamos con un conocimiento incompleto, pero también comprende que, como humanos, operar con un conocimiento incompleto es algo que hacemos, algo en lo que somos buenos. Juega más con nuestras fortalezas que con nuestras debilidades.

Creamos cosas y descubrimos cosas. Hacemos preguntas y realizamos experimentos. Una buena arquitectura surge de entenderla más como un viaje que como un destino, más como un proceso continuo de investigación que como un artefacto congelado.

La arquitectura es una hipótesis que necesita ser probada mediante implementación y medición.

—Tom Gilb

Recorrer este camino requiere cuidado y atención, pensamiento y observación, práctica y principios. Al principio esto puede parecer lento, pero todo depende de la forma en que caminas.

La única manera de ir rápido es hacerlo bien.

—Robert C. Martín

Disfruta el viaje.

—Kevlin Henney
mayo 2017

PREFACIO

El título de este libro es Arquitectura Limpia. Ese es un nombre audaz. Algunos incluso lo considerarían arrogante. Entonces, ¿por qué elegí ese título y por qué escribí este libro?

Escribí mi primera línea de código en 1964, a la edad de 12 años. Ahora es el año 2016, por lo que llevo más de medio siglo escribiendo código. En ese tiempo, aprendí algunas cosas sobre cómo estructurar sistemas de software, cosas que creo que otros probablemente encontrarían valiosas.

Aprendí estas cosas construyendo muchos sistemas, tanto grandes como pequeños. He construido pequeños sistemas integrados y grandes sistemas de procesamiento por lotes. He construido sistemas en tiempo real y sistemas web. He creado aplicaciones de consola, aplicaciones GUI, aplicaciones de control de procesos, juegos, sistemas de contabilidad, sistemas de telecomunicaciones, herramientas de diseño, aplicaciones de dibujo y muchas, muchas otras.

He creado aplicaciones de un solo subprocesso, aplicaciones de subprocessos múltiples, aplicaciones con pocos procesos pesados, aplicaciones con muchos procesos livianos, aplicaciones multiprocesador, aplicaciones de bases de datos, aplicaciones matemáticas, aplicaciones de geometría computacional y muchas, muchas otras.

He creado muchas aplicaciones. He construido muchos sistemas. Y de todos ellos, y tomándolos a todos en consideración, he aprendido algo sorprendente.

¡Las reglas de arquitectura son las mismas!

Esto es sorprendente porque los sistemas que he construido han sido todos radicalmente diferentes. ¿Por qué sistemas tan diferentes deberían compartir reglas de arquitectura similares? Mi conclusión es que las reglas de la arquitectura del software son

independiente de cualquier otra variable.

Esto es aún más sorprendente si se considera el cambio que se ha producido en el hardware durante el mismo medio siglo. Comencé a programar en máquinas del tamaño de refrigeradores de cocina que tenían tiempos de ciclo de medio megahercio, 4K de memoria central, 32K de memoria de disco y una interfaz de teletipo de 10 caracteres por segundo. Estoy escribiendo este prefacio en un autobús mientras estoy de gira por Sudáfrica. Estoy usando una MacBook con cuatro núcleos i7 que funcionan a 2,8 gigahercios cada uno. Tiene 16 gigabytes de RAM, un terabyte de SSD y una pantalla retina de 2880×1800 capaz de mostrar vídeo de altísima definición. La diferencia en el poder computacional es asombrosa. Cualquier análisis razonable mostrará que esta MacBook es al menos 10 veces más potente que aquellas primeras computadoras que comencé a usar hace medio siglo.

22

Veintidós órdenes de magnitud es un número muy grande. Es el número de angstroms desde la Tierra hasta Alpha-Centuri. Es la cantidad de electrones que hay en el cambio que llevas en el bolsillo o en el bolso. Y, sin embargo, ese número (al menos ese número) es el aumento de poder computacional que he experimentado en mi propia vida.

Y con todo ese enorme cambio en el poder computacional, ¿cuál ha sido el efecto en el software que escribo? Ciertamente se ha hecho más grande. Solía pensar que 2000 líneas era un gran programa. Después de todo, era una caja llena de cartas que pesaba 10 libras.

Ahora, sin embargo, un programa no es realmente grande hasta que supera las 100.000 líneas.

El software también se ha vuelto mucho más eficaz. Hoy podemos hacer cosas con las que apenas podíamos soñar en los años 1960. El Proyecto Forbin, La luna es una amante dura y 2001: Una odisea en el espacio intentaron imaginar nuestro futuro actual, pero fallaron de manera bastante significativa. Todos imaginaron máquinas enormes que adquirieron sensibilidad. Lo que tenemos en cambio son máquinas increíblemente pequeñas que todavía son... sólo máquinas.{xx}

Y hay una cosa más en el software que tenemos ahora, en comparación con el software de entonces: está hecho del mismo material. Está hecho de sentencias if , sentencias de asignación y bucles while .

Oh, podrías objetar y decir que tenemos lenguajes mucho mejores y paradigmas superiores. Después de todo, programamos en Java, C# o Ruby, y utilizamos diseño orientado a objetos. Es cierto y, sin embargo, el código sigue siendo sólo un conjunto de secuencia, selección e iteración, tal como lo era en las décadas de 1960 y 1950.

Cuando miras de cerca la práctica de programar computadoras, te das cuenta de que muy poco ha cambiado en 50 años. Los idiomas han mejorado un poco. Las herramientas han mejorado increíblemente. Pero los componentes básicos de un programa informático no han cambiado.

Si tomo a una programadora de computadoras desde 1966 en adelante hasta 2016 y le pongo¹ frente a mi MacBook con IntelliJ y le mostré Java, es posible que necesite 24 horas para recuperarse del shock. Pero entonces ella podría escribir el código. Java simplemente no es tan diferente de C, o incluso de Fortran.

Y si te transportara a 1966 y te mostrara cómo escribir y editar código PDP-8 perforando cinta de papel en un teletipo de 10 caracteres por segundo, es posible que necesites 24 horas para recuperarte de la decepción. Pero entonces podrás escribir el código. El código simplemente no ha cambiado mucho.

Ese es el secreto: esta inmutabilidad del código es la razón por la que las reglas de la arquitectura del software son tan consistentes en todos los tipos de sistemas. Las reglas de la arquitectura del software son las reglas para ordenar y ensamblar los componentes básicos de los programas. Y dado que esos bloques de construcción son universales y no han cambiado, las reglas para ordenarlos también son universales e inmutables.

Los programadores más jóvenes podrían pensar que esto es una tontería. Podrían insistir en que hoy en día todo es nuevo y diferente, que las reglas del pasado ya pasaron. Si eso es lo que piensan, están lamentablemente equivocados. Las reglas no han cambiado. A pesar de todos los nuevos lenguajes, y todos los nuevos marcos, y todos los paradigmas, las reglas son las mismas ahora que cuando Alan Turing escribió el primer código de máquina en 1946.

Pero una cosa ha cambiado: en aquel entonces no sabíamos cuáles eran las reglas. En consecuencia, los rompimos una y otra vez. Ahora, con medio siglo de experiencia a nuestras espaldas, comprendemos esas reglas.

Y son esas reglas (esas reglas eternas e inmutables) de las que trata este libro.

Registre su copia de Clean Architecture en el sitio de InformIT para acceder cómodamente a las actualizaciones y/o correcciones a medida que estén disponibles. Para iniciar el proceso de registro, ingresa a informit.com/register e iniciar sesión o crear una

cuenta. Ingrese el ISBN del producto (9780134494166) y haga clic en Enviar. Busque en la pestaña Productos registrados un enlace de acceso a contenido adicional junto a este producto y siga ese enlace para acceder a los materiales adicionales.

1. Y muy probablemente sería mujer ya que, en aquel entonces, las mujeres constitúan una gran fracción de los programadores.

EXPRESIONES DE GRATITUD

Las personas que participaron en la creación de este libro, sin ningún orden en particular:{xxiii}

Chris Guzikowski

Matt Heuser Jeff Overbey Micah Martin Justin Martin Carl Hickman James Grenning Simon Brown Kevlin Henney Jason Gorman Doug Bradbury Colin Jones Grady Booch

Martin Fowler Alistair Cockburn James O. Coplien Tim Conrad

Richard Lloyd Ken Buscador

Kris Iyer (CK) Mike Carew

Jerry Fitzpatrick Jim Newkirk Ed Thelen

jose mabel

Bill Degnan Y muchos otros, demasiados para nombrarlos.

En mi reseña final de este libro, mientras leía el capítulo sobre Arquitectura gritante, la sonrisa de ojos brillantes y la risa melódica de Jim Weirich resonaron en mi mente. ¡Buena suerte, Jim!

SOBRE EL AUTOR



Robert C. Martin (tío Bob) ha sido programador desde 1970. Es cofundador de cleancoders.com, que ofrece [capacitación en video](#) en línea para desarrolladores de software y es el fundador de Uncle Bob Consulting LLC, que ofrece servicios de consultoría, capacitación y desarrollo de habilidades de software a las principales corporaciones de todo el mundo. Se desempeñó como maestro artesano en 8th Light, Inc., una empresa de consultoría de software con sede en Chicago. Ha publicado decenas de artículos en diversas revistas especializadas y es ponente habitual en congresos y ferias comerciales internacionales. Se desempeñó durante tres años como editor en jefe del Informe C ++ y fue el primer presidente de Agile Alliance.

Martin es autor y editor de muchos libros, entre ellos The Clean Coder, Clean Código, UML para programadores Java, Desarrollo de software ágil, Extremo Programación en la práctica, más gemas de C++, lenguajes de programación de patrones Diseño 3 y diseño de aplicaciones C++ orientadas a objetos utilizando Booch

Método.

INTRODUCCIÓN

No se necesita una gran cantidad de conocimientos y habilidades para que un programa funcione. Los niños de secundaria lo hacen todo el tiempo. Hombres y mujeres jóvenes en la universidad inician negocios de miles de millones de dólares basándose en unas pocas líneas de PHP o Ruby. Hordas de programadores jóvenes en granjas de cubos de todo el mundo revisan enormes documentos de requisitos almacenados en enormes sistemas de seguimiento de problemas para lograr que sus sistemas "funcionen" mediante la pura fuerza bruta de la voluntad. El código que producen puede no ser bonito; pero funciona. Funciona porque hacer que algo funcione (una vez) no es tan difícil.

Hacerlo bien es otra cuestión completamente diferente. Obtener el software correcto es difícil. Se necesitan conocimientos y habilidades que la mayoría de los programadores jóvenes aún no han adquirido. Requiere pensamiento y conocimiento que la mayoría de los programadores no se toman el tiempo para desarrollar. Requiere un nivel de disciplina y dedicación que la mayoría de los programadores nunca imaginaron que necesitarían. Principalmente, se necesita pasión por el oficio y el deseo de ser un profesional.

Y cuando obtienes el software correcto, sucede algo mágico: no necesitas hordas de programadores para que siga funcionando. No necesita documentos de requisitos masivos ni sistemas de seguimiento de problemas enormes. No necesita granjas de cubos globales ni programación 24 horas al día, 7 días a la semana.

Cuando el software se hace bien, se requiere una fracción de los recursos humanos para crearlo y mantenerlo. Los cambios son simples y rápidos. Los defectos son pocos y espaciados. Se minimiza el esfuerzo y se maximiza la funcionalidad y la flexibilidad.

Sí, esta visión suena un poco utópica. Pero he estado allí; Lo he visto suceder. He trabajado en proyectos donde el diseño y la arquitectura del sistema hicieron que fuera fácil de escribir y de mantener. He experimentado proyectos que requirieron una fracción de los recursos humanos previstos. He trabajado en sistemas que tenían tasas de defectos extremadamente bajas. He visto el extraordinario efecto que una buena arquitectura de software puede tener en un sistema, un proyecto y un equipo. He estado en la tierra prometida.

Pero no confíes en mi palabra. Mira tu propia experiencia. ¿Has experimentado lo contrario? ¿Ha trabajado en sistemas que están tan interconectados y estrechamente acoplados que cada cambio, por trivial que sea, lleva semanas e implica enormes riesgos? ¿Ha experimentado la impedancia de un código incorrecto y un diseño deficiente? ¿El diseño de los sistemas en los que ha trabajado ha tenido un enorme efecto negativo en la moral del equipo, la confianza de los clientes y la paciencia de los gerentes? ¿Ha visto equipos, departamentos e incluso empresas que han sido derribados por la estructura podrida de su software? ¿Has estado en el infierno de la programación?

Yo lo he hecho y, hasta cierto punto, la mayoría de nosotros también lo hemos hecho. Es mucho más común abrirse camino a través de diseños de software terribles que disfrutar del placer de trabajar con uno bueno.

1

¿QUÉ ES DISEÑO Y ARQUITECTURA?



Ha habido mucha confusión sobre el diseño y la arquitectura a lo largo de los años. ¿Qué es el diseño? ¿Qué es la arquitectura? ¿Cuáles son las diferencias entre los dos?

Uno de los objetivos de este libro es romper con toda esa confusión y definir, de una vez por todas, qué son el diseño y la arquitectura. Para empezar, afirmaré que no hay diferencia entre ellos. Ninguno en absoluto.

La palabra "arquitectura" se usa a menudo en el contexto de algo de alto nivel que está divorciado de los detalles de nivel inferior, mientras que "diseño" parece implicar más a menudo estructuras y decisiones de un nivel inferior. Pero este uso no tiene sentido.

cualquier momento miras lo que hace un verdadero arquitecto.

Pensemos en el arquitecto que diseñó mi nuevo hogar. ¿Esta casa tiene una arquitectura? Claro que lo hace. ¿Y cuál es esa arquitectura? Bueno, es la forma de la casa, el aspecto exterior, las elevaciones y la distribución de los espacios y habitaciones. Pero cuando miro los diagramas que produjo mi arquitecto, veo una inmensa cantidad de detalles de bajo nivel. Veo dónde se colocarán cada toma de corriente, interruptor de luz y luz. Veo qué interruptores controlan qué luces. Veo dónde está colocada la caldera y el tamaño y la ubicación del calentador de agua y la bomba de sumidero. Veo representaciones detalladas de cómo se construirán las paredes, los techos y los cimientos.

En resumen, veo todos los pequeños detalles que respaldan todas las decisiones de alto nivel. También veo que esos detalles de bajo nivel y las decisiones de alto nivel son parte de todo el diseño de la casa.

Y lo mismo ocurre con el diseño de software. Los detalles de bajo nivel y la estructura de alto nivel son todos parte del mismo todo. Forman un tejido continuo que define la forma del sistema. No puedes tener uno sin el otro; de hecho, no hay una línea divisoria clara que los separe. Simplemente hay una serie de decisiones desde el nivel más alto hasta el más bajo.

¿LA META?

¿Y el objetivo de esas decisiones? ¿El objetivo del buen diseño de software? Ese objetivo es nada menos que mi descripción utópica:

El objetivo de la arquitectura de software es minimizar los recursos humanos necesarios para construir y mantener el sistema requerido.

La medida de la calidad del diseño es simplemente la medida del esfuerzo requerido para satisfacer las necesidades del cliente. Si ese esfuerzo es bajo y se mantiene bajo durante toda la vida útil del sistema, el diseño es bueno. Si ese esfuerzo crece con cada nuevo lanzamiento, el diseño es malo. Es tan simple como eso.

CASO DE ESTUDIO

Como ejemplo, considere el siguiente estudio de caso. Incluye datos reales de un

empresa real que desea permanecer en el anonimato.

Primero, veamos el crecimiento del personal de ingeniería. Estoy seguro de que estará de acuerdo en que esta tendencia es muy alentadora. ¡Un crecimiento como el que se muestra en [la Figura 1.1](#) debe ser una indicación de un éxito significativo!

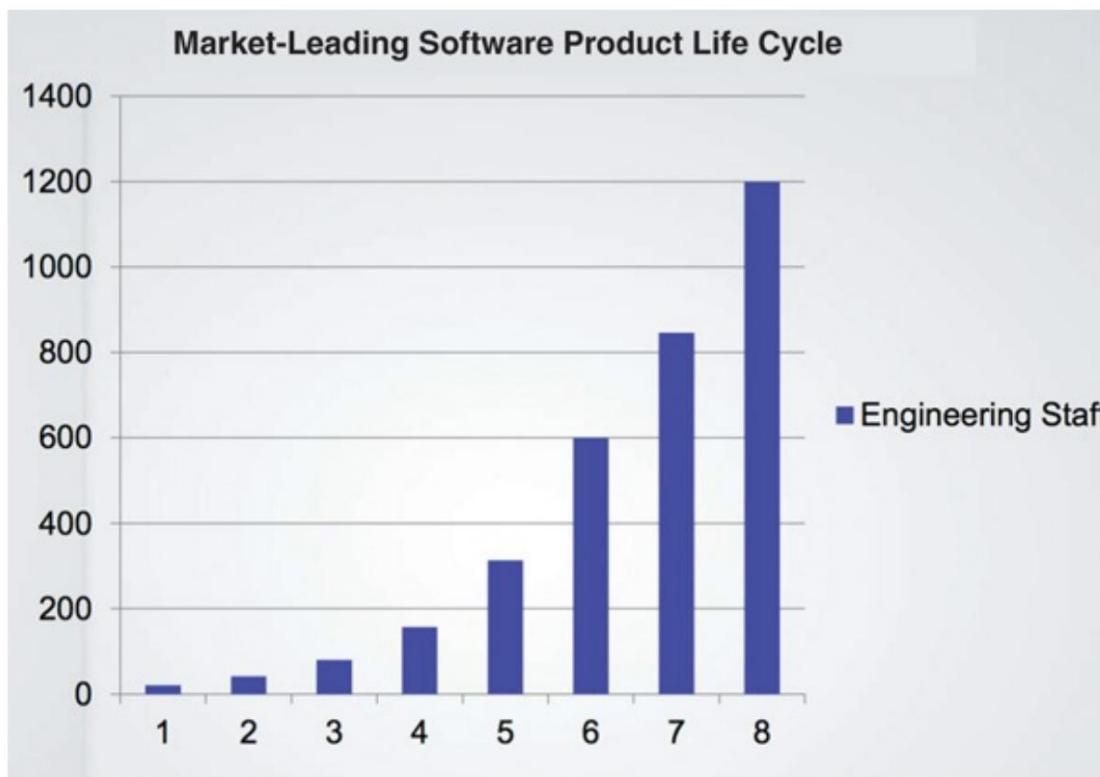


Figura 1.1 Crecimiento del personal de ingeniería

Reproducido con autorización de una presentación de diapositivas de Jason Gorman.

Ahora veamos la productividad de la empresa durante el mismo período de tiempo, medida con simples líneas de código ([Figura 1.2](#)).



Figura 1.2 Productividad durante el mismo período de tiempo

Es evidente que algo anda mal aquí. Aunque cada versión cuenta con el respaldo de un número cada vez mayor de desarrolladores, el crecimiento del código parece acercarse a una asíntota.

Ahora aquí está el gráfico realmente aterrador: [la Figura 1.3](#) muestra cómo el costo por línea de código ha cambiado con el tiempo.

Estas tendencias no son sostenibles. No importa cuán rentable pueda ser la empresa en este momento: esas curvas drenarán catastróficamente las ganancias del modelo de negocio y llevarán a la empresa a un estancamiento, si no a un colapso total.

¿Qué causó este notable cambio en la productividad? ¿Por qué fue 40 veces más caro producir el código en la versión 8 que en la versión 1?

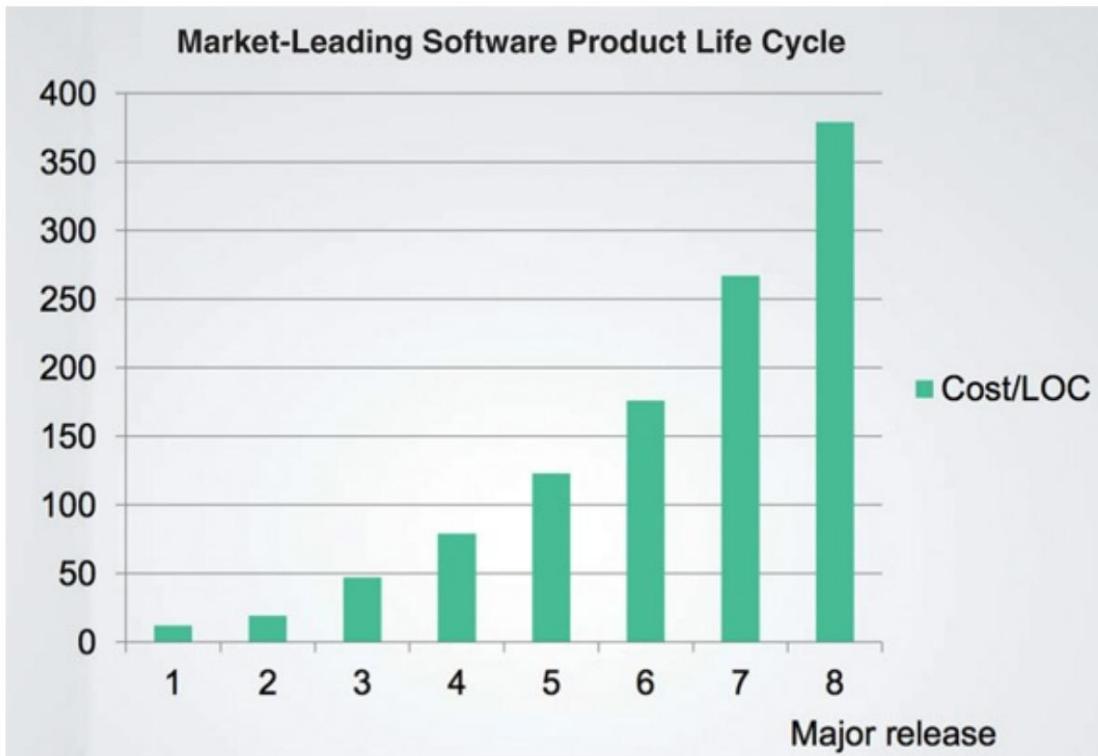


Figura 1.3 Costo por línea de código a lo largo del tiempo

LA FIRMA DE UN DESAFÍO

Lo que estás viendo es la firma de un desastre. Cuando los sistemas se ensamblan rápidamente, cuando el gran número de programadores es el único factor que impulsa el resultado y cuando se presta poca o ninguna atención a la limpieza del código o a la estructura del diseño, entonces puedes confiar en aprovechar este curva hasta su feo final.

[La Figura 1.4](#) muestra cómo ven esta curva a los desarrolladores. Comenzaron con una productividad de casi el 100%, pero con cada lanzamiento su productividad disminuyó. En la cuarta versión, estaba claro que su productividad iba a tocar fondo en un enfoque asintótico hacia cero.

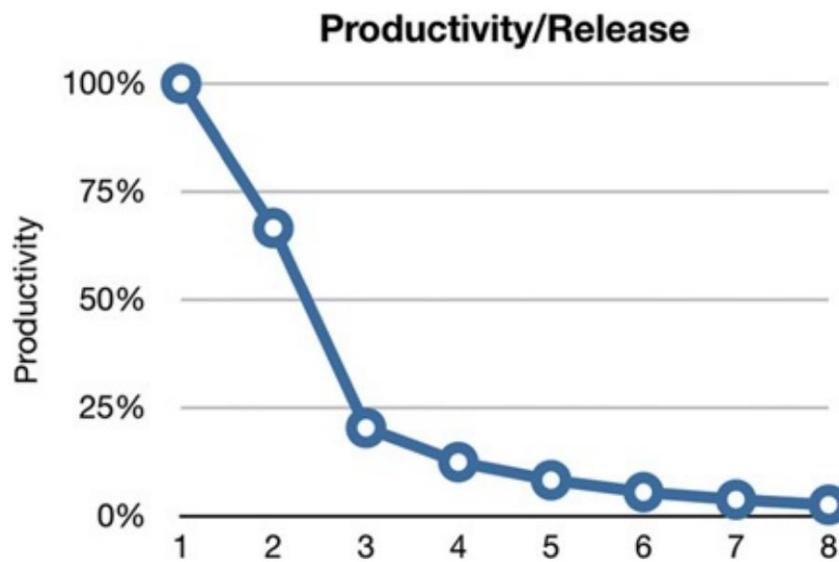


Figura 1.4 Productividad por lanzamiento

Desde el punto de vista de los desarrolladores, esto es tremadamente frustrante, porque todos están trabajando duro. Nadie ha disminuido su esfuerzo.

Y, sin embargo, a pesar de todos sus actos heroicos, horas extras y dedicación, simplemente ya no logran hacer mucho. Todo su esfuerzo se ha desviado de las funciones y ahora se dedica a gestionar el desorden. Su trabajo, tal como está, se ha convertido en mover el desorden de un lugar a otro, y al siguiente, y al siguiente, para poder agregar una pequeña característica más.

LA VISTA EJECUTIVA

Si crees que eso es malo, ¡imagínate cómo ven esta imagen a los ejecutivos! Considere [la Figura 1.5](#), que muestra la nómina de desarrollo mensual para el mismo período.

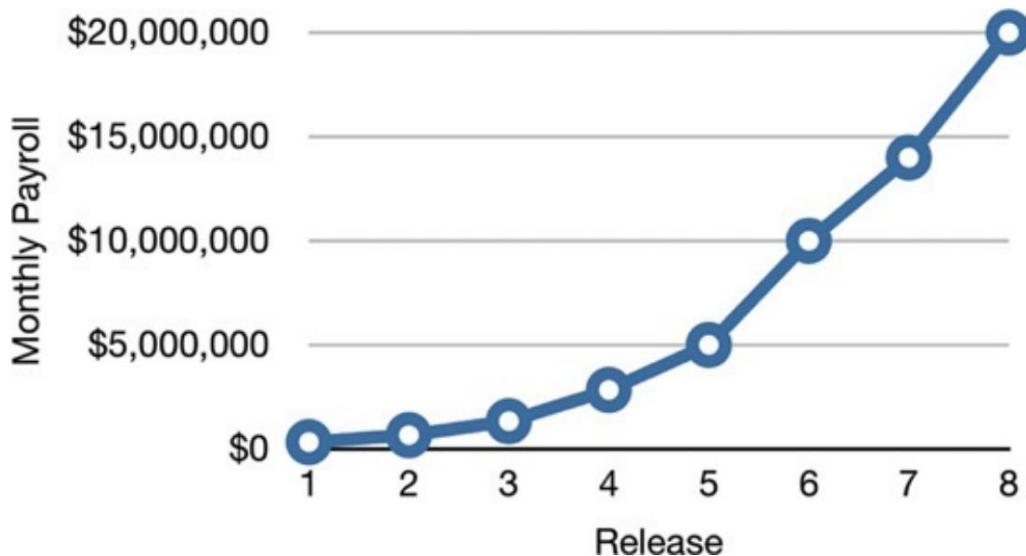


Figura 1.5 Nómina de desarrollo mensual por versión

La versión 1 se entregó con una nómina mensual de unos cientos de miles de dólares. La segunda versión costó unos cientos de miles más. En la octava publicación, la nómina mensual era de 20 millones de dólares y seguía aumentando.

Sólo este gráfico por sí solo da miedo. Es evidente que algo sorprendente está sucediendo. Se espera que los ingresos superen los costos y, por lo tanto, justifiquen el gasto. Pero no importa cómo se mire esta curva, es motivo de preocupación.

Pero ahora compare la curva de [la Figura 1.5](#) con las líneas de código escritas por versión en [la Figura 1.2](#). Esos primeros cientos de miles de dólares al mes permitieron adquirir muchas funciones, ¡pero los últimos 20 millones de dólares no permitieron casi nada! Cualquier director financiero miraría estos dos gráficos y sabría que es necesaria una acción inmediata para evitar el desastre.

¿Pero qué medidas se pueden tomar? ¿Qué ha salido mal? ¿Qué ha causado esta increíble caída de la productividad? ¿Qué pueden hacer los ejecutivos, aparte de patear y enojarse con los desarrolladores?

¿QUÉ SALIÓ MAL?

Hace casi 2600 años, Esopo contó la historia de la tortuga y la liebre. La moraleja de esa historia se ha expresado muchas veces de diferentes maneras:

- "Lento y constante gana la carrera."

- “La carrera no es de los veloces, ni la batalla de los fuertes”. • “Cuanto más prisa, menos velocidad”.

La historia en sí ilustra la tontería del exceso de confianza. La Liebre, tan confiada en su velocidad intrínseca, no se toma en serio la carrera y, por lo tanto, duerme una siesta mientras la Tortuga cruza la línea de meta.

Los desarrolladores modernos están en una carrera similar y muestran un exceso de confianza similar. Oh, no duermen, ni mucho menos. La mayoría de los desarrolladores modernos trabajan duro. Pero una parte de su cerebro sí duerme: la parte que sabe que un código bueno, limpio y bien diseñado es importante.

Estos desarrolladores creen en una mentira familiar: “Podemos limpiarlo más tarde; ¡Solo tenemos que llegar al mercado primero! Por supuesto, las cosas nunca se arreglan después, porque las presiones del mercado nunca disminuyen. Llegar primero al mercado simplemente significa que ahora tienes una horda de competidores detrás de ti y tienes que adelantarte a ellos corriendo lo más rápido que puedas.

Y por eso los desarrolladores nunca cambian de modo. No pueden regresar y limpiar las cosas porque tienen que terminar la siguiente función, y la siguiente, y la siguiente, y la siguiente. Y así el desorden crece y la productividad continúa su aproximación asintótica hacia cero.

Así como la Liebre confiaba demasiado en su velocidad, los desarrolladores confían demasiado en su capacidad para seguir siendo productivos. Pero el creciente desorden de código que mina su productividad nunca duerme y nunca cede. Si se le permite, reducirá la productividad a cero en cuestión de meses.

La mentira más grande que creen los desarrolladores es la noción de que escribir código desordenado los hace ir más rápido en el corto plazo y solo los ralentiza a largo plazo. Los desarrolladores que aceptan esta mentira exhiben el exceso de confianza de la liebre en su capacidad para cambiar de modo de crear desorden a limpiar desorden en algún momento en el futuro, pero también cometen un simple error de hecho. El hecho es que ensuciar siempre es más lento que mantenerse limpio, sin importar la escala de tiempo que estés usando.

Considere los resultados de un notable experimento realizado por Jason Gorman que se muestra en [la Figura 1.6](#). Jason realizó esta prueba durante un período de seis días. Cada día completó un programa sencillo para convertir números enteros en números romanos.

Sabía que su trabajo estaba completo cuando su conjunto predefinido de pruebas de aceptación

aprobado. Cada día la tarea tomó un poco menos de 30 minutos. Jason utilizó una disciplina de limpieza muy conocida llamada desarrollo basado en pruebas (TDD) en el primer, tercer y quinto día. Los otros tres días escribió el código sin esa disciplina.

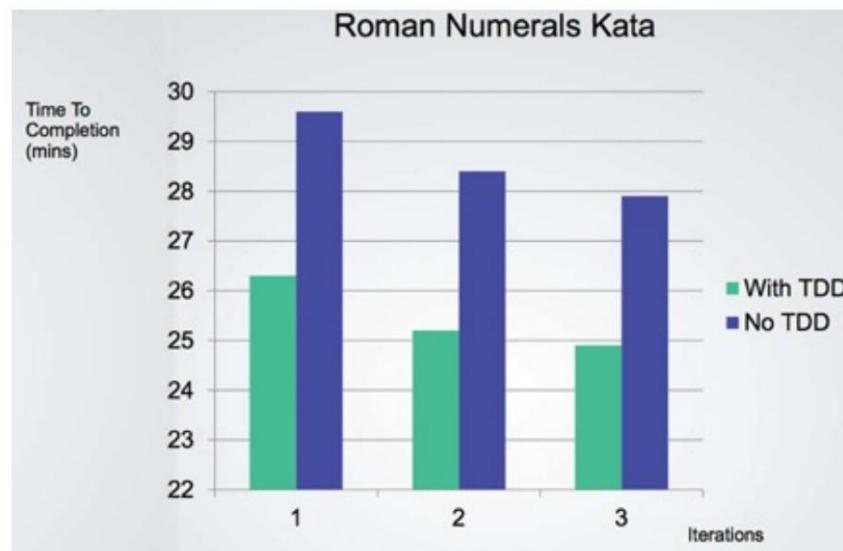


Figura 1.6 Tiempo de finalización por iteraciones y uso/no uso de TDD

Primero, observe la curva de aprendizaje que se muestra en [la Figura 1.6](#). El trabajo en los últimos días se completa más rápidamente que en los primeros. Observe también que el trabajo en los días TDD avanzó aproximadamente un 10% más rápido que el trabajo en los días sin TDD, y que incluso el día TDD más lento fue más rápido que el día más rápido sin TDD.

Algunas personas podrían ver ese resultado y pensar que es un resultado notable. Pero aquellos que no se han dejado engañar por el exceso de confianza de la Liebre, el resultado es esperado, porque conocen esta simple verdad del desarrollo de software:

La única manera de ir rápido es hacerlo bien.

Y esa es la respuesta al dilema del ejecutivo. La única manera de revertir la disminución de la productividad y el aumento de los costos es lograr que los desarrolladores dejen de pensar como la liebre demasiado confiada y comiencen a asumir la responsabilidad por el desastre que han causado.

Los desarrolladores pueden pensar que la respuesta es empezar de cero y rediseñar todo el sistema, pero eso es sólo la Liebre hablando otra vez. El mismo exceso de confianza que condujo al desastre ahora les dice que pueden construirlo mejor si tan solo pudieran comenzar la carrera de nuevo. La realidad es menos halagüeña:

Su exceso de confianza llevará el rediseño al mismo desastre que el proyecto original.

CONCLUSIÓN

En todos los casos, la mejor opción es que la organización de desarrollo reconozca y evite su propio exceso de confianza y comience a tomar en serio la calidad de su arquitectura de software.

Para tomarse en serio la arquitectura de software, es necesario saber qué es una buena arquitectura de software. Para construir un sistema con un diseño y una arquitectura que minimicen el esfuerzo y maximicen la productividad, es necesario saber qué atributos de la arquitectura del sistema conducen a ese fin.

De eso se trata este libro. Describe cómo son las arquitecturas y los diseños limpios y buenos, de modo que los desarrolladores de software puedan crear sistemas que tengan una vida útil prolongada y rentable.

2

UNA HISTORIA DE DOS VALORES



Cada sistema de software proporciona dos valores diferentes a las partes interesadas: comportamiento y estructura. Los desarrolladores de software son responsables de garantizar que ambos valores se mantengan altos. Desafortunadamente, a menudo se centran en uno excluyendo al otro. Desafortunadamente aún más, a menudo se centran en el menor de los dos valores, dejando al sistema de software finalmente sin valor.

COMPORTAMIENTO

El primer valor del software es su comportamiento. Se contratan programadores para hacer que las máquinas se comporten de una manera que genere o ahorre dinero para las partes interesadas. Hacemos esto ayudando a las partes interesadas a desarrollar una especificación funcional, o

documento de requisitos. Luego escribimos el código que hace que las máquinas de las partes interesadas satisfagan esos requisitos.

Cuando la máquina viola esos requisitos, los programadores sacan sus depuradores y solucionan el problema.

Muchos programadores creen que ese es todo su trabajo. Creen que su trabajo es hacer que la máquina implemente los requisitos y corregir cualquier error. Están lamentablemente equivocados.

ARQUITECTURA

El segundo valor del software tiene que ver con la palabra "software", una palabra compuesta de "software" y "ware". La palabra "artículos" significa "producto"; la palabra "suave" ... Bueno, ahí es donde reside el segundo valor.

El software fue inventado para ser "suave". Su objetivo era ser una forma de cambiar fácilmente el comportamiento de las máquinas. Si hubiéramos querido que el comportamiento de las máquinas fuera difícil de cambiar, lo habríamos llamado hardware.

Para cumplir su propósito, el software debe ser blando, es decir, debe ser fácil de cambiar. Cuando las partes interesadas cambian de opinión sobre una característica, ese cambio debería ser simple y fácil de realizar. La dificultad para realizar tal cambio debería ser proporcional sólo al alcance del cambio y no a la forma del cambio.

Es esta diferencia entre alcance y forma la que a menudo impulsa el crecimiento de los costos de desarrollo de software. Es la razón por la que los costos crecen desproporcionadamente con la magnitud de los cambios solicitados. Es la razón por la que el primer año de desarrollo es mucho más barato que el segundo, y el segundo año es mucho más barato que el tercero.

Desde el punto de vista de las partes interesadas, simplemente están proporcionando una corriente de cambios de alcance más o menos similar. Desde el punto de vista de los desarrolladores, las partes interesadas les están dando un flujo de piezas de rompecabezas que deben encajar en un rompecabezas de complejidad cada vez mayor. Cada nueva solicitud es más difícil de adaptar que la anterior, porque la forma del sistema no coincide con la forma de la solicitud.

Estoy usando la palabra "forma" aquí de una manera poco convencional, pero creo que la

La metáfora es adecuada. Los desarrolladores de software a menudo se sienten obligados a meter clavijas cuadradas en agujeros redondos.

El problema, por supuesto, es la arquitectura del sistema. Cuanto más prefiera esta arquitectura una forma sobre otra, más probable será que las nuevas características sean cada vez más difíciles de encasar en esa estructura. Por lo tanto, las arquitecturas deben ser tan independientes de la forma como sea práctico.

EL MAYOR VALOR

¿Función o arquitectura? ¿Cuál de estos dos proporciona el mayor valor? ¿Es más importante que el sistema de software funcione o es más importante que el sistema de software sea fácil de cambiar?

Si preguntas a los gerentes de negocios, a menudo te dirán que es más importante que el sistema de software funcione. Los desarrolladores, a su vez, suelen aceptar esta actitud. Pero es una actitud equivocada. Puedo demostrar que está mal con la simple herramienta lógica de examinar los extremos.

- Si me das un programa que funciona perfectamente pero que es imposible de cambiar, entonces no funcionará cuando cambien los requisitos y no podré hacerlo funcionar. Por lo tanto el programa quedará inútil.
- Si me dan un programa que no funciona pero que es fácil de cambiar, entonces puedo hacerlo funcionar y mantenerlo funcionando a medida que cambian los requisitos. Por lo tanto, el programa seguirá siendo continuamente útil.

Es posible que este argumento no le resulte convincente. Después de todo, no existe ningún programa que sea imposible de cambiar. Sin embargo, hay sistemas que son prácticamente imposibles de cambiar, porque el costo del cambio supera el beneficio del cambio. Muchos sistemas llegan a ese punto en algunas de sus características o configuraciones.

Si les pregunta a los gerentes de negocios si quieren poder realizar cambios, dirán que por supuesto que sí, pero luego matizarán su respuesta señalando que la funcionalidad actual es más importante que cualquier flexibilidad posterior. Por el contrario, si los gerentes de negocios le piden un cambio y sus costos estimados para ese cambio son inasequibles, los gerentes de negocios probablemente estarán furiosos porque usted permitió que el sistema llegara al punto en que el cambio no era práctico.

MATRIZ DE EISENHOWER

Consideremos la matriz de importancia versus urgencia del presidente Dwight D. Eisenhower ([Figura 2.1](#)). De esta matriz, Eisenhower dijo:

Tengo dos tipos de problemas, los urgentes y los importantes. Lo urgente no es importante y lo importante nunca es urgente.—



Figura 2.1 Matriz de Eisenhower

Hay mucho de cierto en este viejo dicho. Las cosas que son urgentes rara vez son de gran importancia, y las cosas que son importantes rara vez son de gran urgencia.

El primer valor del software –el comportamiento– es urgente pero no siempre particularmente importante.

El segundo valor del software, la arquitectura, es importante pero nunca particularmente urgente.

Por supuesto, algunas cosas son a la vez urgentes e importantes. Otras cosas no son urgentes ni importantes. En última instancia, podemos ordenar estos cuatro pareados en prioridades:

1. Urgente e importante 2.

No urgente e importante

3. Urgente y no importante 4. No urgente y no importante

Tenga en cuenta que la arquitectura del código (lo importante) se encuentra en las dos primeras posiciones de esta lista, mientras que el comportamiento del código ocupa la primera y tercera posiciones.

El error que suelen cometer los administradores y desarrolladores de negocios es elevar los elementos de la posición 3 a la posición 1. En otras palabras, no logran separar aquellas características que son urgentes pero no importantes de aquellas que realmente son urgentes e importantes. Este fallo lleva entonces a ignorar la arquitectura importante del sistema en favor de las características sin importancia del sistema.

El dilema para los desarrolladores de software es que los gerentes de negocios no están preparados para evaluar la importancia de la arquitectura. Para eso se contrató a los desarrolladores de software. Por lo tanto, es responsabilidad del equipo de desarrollo de software afirmar la importancia de la arquitectura sobre la urgencia de las funciones.

LUCHA POR LA ARQUITECTURA

Cumplir con esta responsabilidad significa lanzarse a una pelea, o tal vez una palabra mejor sea "lucha". Francamente, así es siempre como se hacen estas cosas. El equipo de desarrollo tiene que luchar por lo que cree que es mejor para la empresa, al igual que el equipo de gestión, el equipo de marketing, el equipo de ventas y el equipo de operaciones. Siempre es una lucha.

Los equipos de desarrollo de software eficaces abordan esa lucha de frente. Se pelean descaradamente con todos los demás interesados como iguales. Recuerde, como desarrollador de software, usted es una parte interesada. Usted tiene un interés en el software que necesita proteger. Eso es parte de su papel y parte de su deber. Y es una gran parte del motivo por el que te contrataron.

Este desafío es doblemente importante si eres arquitecto de software. Los arquitectos de software, en virtud de la descripción de su trabajo, se centran más en la estructura del sistema que en sus características y funciones. Los arquitectos crean una arquitectura que permite que esas características y funciones se desarrollen, modifiquen y amplíen fácilmente.

Sólo recuerde: si la arquitectura es lo último, entonces el desarrollo del sistema será cada vez más costoso y, eventualmente, el cambio será prácticamente imposible para una parte o la totalidad del sistema. Si se permite que eso suceda, significa que el equipo de desarrollo de software no luchó lo suficiente por lo que sabían que era necesario.

1. De un discurso pronunciado en la Universidad Northwestern en 1954.

||

COMENZANDO CON LOS LADRILLOS: PARADIGMAS DE PROGRAMACIÓN

La arquitectura de software comienza con el código, por lo que comenzaremos nuestra discusión sobre arquitectura analizando lo que hemos aprendido sobre el código desde que se creó el código escrito.

En 1938, Alan Turing sentó las bases de lo que se convertiría en la programación informática. No fue el primero en concebir una máquina programable, pero sí el primero en comprender que los programas eran simplemente datos. En 1945, Turing estaba escribiendo programas reales en computadoras reales en un código que podríamos reconocer (si entrecerrábamos los ojos lo suficiente). Esos programas utilizaban bucles, ramas, asignaciones, subrutinas, pilas y otras estructuras familiares. El lenguaje de Turing era binario.

Desde entonces se han producido varias revoluciones en la programación. Una revolución que todos conocemos muy bien es la revolución de los idiomas.

Primero, a finales de los años 1940, llegaron los ensambladores. Estos “lenguajes” aliviaron a los programadores de la monotonía de traducir sus programas al binario. En 1951, Grace Hopper inventó A0, el primer compilador. De hecho, ella acuñó el término compilador. Fortran se inventó en 1953 (el año después de mi nacimiento). Lo que siguió fue una avalancha incesante de nuevos lenguajes de programación: COBOL, PL/1, SNOBOL, C, Pascal, C++, Java, hasta el infinito.

Otra revolución, probablemente más significativa, se produjo en los paradigmas de programación. Los paradigmas son formas de programación, relativamente ajenas a los lenguajes. Un paradigma le dice qué estructuras de programación usar y cuándo usarlas.

Hasta la fecha, ha habido tres de esos paradigmas. Por razones que discutiremos más adelante, es poco probable que haya otros.

3

RESUMEN DEL PARADIGMA



Los tres paradigmas incluidos en este capítulo de descripción general son la programación estructurada, la programación orientada a objetos y la programación funcional.

PROGRAMACIÓN ESTRUCTURADA

El primer paradigma adoptado (pero no el primero inventado) fue la programación estructurada, descubierta por Edsger Wybe Dijkstra en 1968. Dijkstra demostró que el uso de saltos desenfrenados (declaraciones `goto`) es perjudicial para la estructura del programa. Como veremos en los capítulos siguientes, reemplazó esos saltos con las construcciones más familiares `if/then/else` y `do/ while/hasta`.

Podemos resumir el paradigma de programación estructurada de la siguiente manera:

La programación estructurada impone disciplina a la transferencia directa de control.

PROGRAMACIÓN ORIENTADA A OBJETOS

El segundo paradigma adoptado fue descubierto dos años antes, en 1966, por Ole Johan Dahl y Kristen Nygaard. Estos dos programadores notaron que el marco de la pila de llamadas a funciones en el lenguaje ALGOL se podía mover a un montón, permitiendo así que las variables locales declaradas por una función existieran mucho después de que la función regresara. La función se convirtió en constructor de una clase, las variables locales se convirtieron en variables de instancia y las funciones anidadas en métodos.

Esto condujo inevitablemente al descubrimiento del polimorfismo mediante el uso disciplinado de punteros de función.

Podemos resumir el paradigma de programación orientada a objetos de la siguiente manera:

La programación orientada a objetos impone disciplina a la transferencia indirecta de control.

PROGRAMACIÓN FUNCIONAL

El tercer paradigma, que recién ha comenzado a adoptarse, fue el primero en inventarse. De hecho, su invención es anterior a la propia programación informática.

La programación funcional es el resultado directo del trabajo de Alonzo Church, quien en 1936 inventó el cálculo λ mientras perseguía el mismo problema matemático que motivaba a Alan Turing al mismo tiempo. Su cálculo λ es la base del lenguaje LISP, inventado en 1958 por John McCarthy. Una noción fundamental del cálculo λ es la inmutabilidad, es decir, la noción de que los valores de los símbolos no cambian. Esto significa efectivamente que un lenguaje funcional no tiene declaración de asignación. De hecho, la mayoría de los lenguajes funcionales tienen algunos medios para alterar el valor de una variable, pero sólo bajo una disciplina muy estricta.

Podemos resumir el paradigma de programación funcional de la siguiente manera:

La programación funcional impone disciplina en la asignación.

COMIDA PARA EL PENSAMIENTO

Observe el patrón que he establecido deliberadamente al presentar estos tres paradigmas de programación: cada uno de los paradigmas elimina capacidades del programador.

Ninguno de ellos añade nuevas capacidades. Cada uno impone algún tipo de disciplina adicional que tiene una intención negativa . Los paradigmas nos dicen qué no hacer, más que qué hacer .

Otra forma de ver esta cuestión es reconocer que cada paradigma nos quita algo.

Los tres paradigmas juntos eliminan declaraciones goto , punteros de función y asignaciones. ¿Queda algo por llevar?

Probablemente no. Por lo tanto, es probable que estos tres paradigmas sean los únicos tres que veremos, al menos los únicos tres que son negativos. Una prueba más de que ya no existen tales paradigmas es que todos fueron descubiertos en los diez años transcurridos entre 1958 y 1968. En las muchas décadas que siguieron, no se agregaron nuevos paradigmas.

CONCLUSIÓN

¿Qué tiene que ver esta lección de historia sobre paradigmas con la arquitectura?

Todo. Utilizamos el polimorfismo como mecanismo para cruzar fronteras arquitectónicas; utilizamos programación funcional para imponer disciplina sobre la ubicación y el acceso a los datos; y utilizamos programación estructurada como base algorítmica de nuestros módulos.

Observe qué tan bien se alinean esos tres con las tres grandes preocupaciones de la arquitectura: función, separación de componentes y gestión de datos.

4

PROGRAMACIÓN ESTRUCTURADA



Edsger Wybe Dijkstra nació en Rotterdam en 1930. Sobrevivió al bombardeo de Rotterdam durante la Segunda Guerra Mundial, junto con la ocupación alemana de los Países Bajos, y en 1948 se graduó de la escuela secundaria con las mejores calificaciones posibles en matemáticas, física, química y biología. En marzo de 1952, a la edad de 21 años (y sólo 9 meses antes de que yo naciera), Dijkstra aceptó un trabajo en el Centro de Matemáticas de Ámsterdam como el primer programador de los Países Bajos.

En 1955, después de haber sido programador durante tres años y siendo aún estudiante, Dijkstra concluyó que el desafío intelectual de la programación era mayor que el desafío intelectual de la física teórica. Como resultado, eligió la programación como su carrera a largo plazo.

En 1957, Dijkstra se casó con María Debets. En aquella época, en los Países Bajos había que declarar su profesión como parte del rito matrimonial. Las autoridades holandesas no estaban dispuestas a aceptar la profesión de "programador" como profesión de Dijkstra; Nunca habían oído hablar de tal profesión. Para satisfacerlos, Dijkstra se decidió por "físico teórico" como título de trabajo.

Como parte de la decisión de hacer de la programación su carrera, Dijkstra consultó con su jefe, Adriaan van Wijngaarden. A Dijkstra le preocupaba que nadie hubiera identificado una disciplina o ciencia de la programación y que, por lo tanto, no lo tomaran en serio. Su jefe respondió que Dijkstra bien podría ser una de las personas que descubriría tales disciplinas, convirtiendo así el software en una ciencia.

Dijkstra comenzó su carrera en la era de los tubos de vacío, cuando las computadoras eran enormes, frágiles, lentas, poco confiables y (según los estándares actuales) extremadamente limitadas. En esos primeros años, los programas se escribían en binario o en un lenguaje ensamblador muy tosco. Los insumos tomaban la forma física de cinta de papel o tarjetas perforadas. El ciclo de edición/compilación/prueba duró horas, si no días.

Fue en este entorno primitivo donde Dijkstra hizo sus grandes descubrimientos.

PRUEBA

El problema que Dijkstra reconoció desde el principio fue que programar es difícil y que los programadores no lo hacen muy bien. Un programa de cualquier complejidad contiene demasiados detalles para que un cerebro humano pueda manejarlos sin ayuda.

Pasar por alto sólo un pequeño detalle da como resultado programas que pueden parecer funcionar, pero fallan de manera sorprendente.

La solución de Dijkstra fue aplicar la disciplina matemática de la prueba. Su visión era la construcción de una jerarquía euclidianas de postulados, teoremas, corolarios y lemas. Dijkstra pensó que los programadores podrían utilizar esa jerarquía como lo hacen los matemáticos. En otras palabras, los programadores usarían estructuras probadas y las unirían con código que luego ellos mismos demostrarían que son correctos.

Por supuesto, para poner esto en marcha, Dijkstra se dio cuenta de que tendría que demostrar la técnica para escribir pruebas básicas de algoritmos simples. Esto lo encontró ser

todo un reto.

Durante su investigación, Dijkstra descubrió que ciertos usos de las declaraciones goto evitan que los módulos se descompongan recursivamente en unidades cada vez más pequeñas, impidiendo así el uso del enfoque de divide y vencerás necesario para pruebas razonables.

Otros usos de goto, sin embargo, no tuvieron este problema. Dijkstra se dio cuenta de que estos “buenos” usos de goto correspondían a estructuras simples de control de selección e iteración como if/then/else y do/ while. Los módulos que utilizaban sólo ese tipo de estructuras de control podían subdividirse recursivamente en unidades demostrables.

Dijkstra sabía que esas estructuras de control, cuando se combinaban con la ejecución secuencial, eran especiales. Habían sido identificados dos años antes por Böhm y Jacopini, quienes demostraron que todos los programas pueden construirse a partir de sólo tres estructuras: secuencia, selección e iteración.

Este descubrimiento fue notable: las mismas estructuras de control que hacían que un módulo fuera demostrable eran el mismo conjunto mínimo de estructuras de control a partir del cual se pueden construir todos los programas. Así nació la programación estructurada.

Dijkstra demostró que se podía demostrar que las afirmaciones secuenciales eran correctas mediante una simple enumeración. La técnica rastreó matemáticamente las entradas de la secuencia hasta las salidas de la secuencia. Este enfoque no era diferente de cualquier prueba matemática normal.

Dijkstra abordó la selección mediante la reaplicación de la enumeración. Se enumeró cada camino a través de la selección. Si ambos caminos finalmente produjeron resultados matemáticos apropiados, entonces la prueba era sólida.

La iteración fue un poco diferente. Para demostrar que una iteración era correcta, Dijkstra tuvo que utilizar la inducción. Demostró el caso de 1 por enumeración. Luego demostró el caso de que si se suponía que N era correcto, N + 1 era correcto, nuevamente mediante enumeración. También demostró los criterios inicial y final de la iteración por enumeración.

Semejantes pruebas eran laboriosas y complejas, pero eran pruebas. Con su desarrollo, la idea de que se podía construir una jerarquía euclíadiana de teoremas parecía alcanzable.

UNA PROCLAMACIÓN DAÑINA

En 1968, Dijkstra escribió una carta al editor del CACM, que se publicó en el número de marzo. El título de esta carta era "Ir a la declaración considerada dañina". El artículo esboza su posición sobre las tres estructuras de control.

Y el mundo de la programación se incendió. En aquel entonces no teníamos Internet, por lo que la gente no podía publicar memes desagradables de Dijkstra y no podían criticarlo en línea. Pero podían escribir cartas a los editores de muchas revistas publicadas, y lo hicieron.

No todas esas cartas eran necesariamente educadas. Algunas eran intensamente negativas; otros expresaron un fuerte apoyo a su posición. Y así se inició la batalla, que finalmente duró aproximadamente una década.

Al final el argumento se apagó. La razón era sencilla: Dijkstra había ganado.

A medida que evolucionaron los lenguajes informáticos, la declaración goto fue retrocediendo cada vez más, hasta que prácticamente desapareció. La mayoría de los lenguajes modernos no tienen una declaración goto y, por supuesto, LISP nunca la tuvo.

Hoy en día todos somos programadores estructurados, aunque no necesariamente por elección propia. Lo que pasa es que nuestros idiomas no nos dan la opción de utilizar una transferencia de control directa e indisciplinada.

Algunos pueden señalar rupturas con nombre en Java o excepciones como análogos de Goto . De hecho, estas estructuras no son las transferencias de control absolutamente ilimitadas que alguna vez tuvieron lenguajes más antiguos como Fortran o COBOL. De hecho, incluso los idiomas que todavía admiten la palabra clave goto a menudo restringen el objetivo dentro del alcance de la función actual.

DESCOMPOSICIÓN FUNCIONAL

La programación estructurada permite que los módulos se descompongan de forma recursiva en unidades demostrables, lo que a su vez significa que los módulos se pueden descomponer funcionalmente. Es decir, se puede tomar el planteamiento de un problema a gran escala y descomponerlo en funciones de alto nivel. Luego, cada una de esas funciones se puede descomponer en funciones de nivel inferior, hasta el infinito. Además, cada una de esas funciones descompuestas se puede representar utilizando las estructuras de control restringidas.

de programación estructurada.

Sobre la base de esta base, disciplinas como el análisis estructurado y el diseño estructurado se hicieron populares a finales de los años 1970 y durante los años 1980.

Hombres como Ed Yourdon, Larry Constantine, Tom DeMarco y Meilir Page-Jones promovieron y popularizaron estas técnicas durante ese período. Siguiendo estas disciplinas, los programadores podrían descomponer grandes sistemas propuestos en módulos y componentes que podrían descomponerse en pequeñas funciones demostrables.

SIN PRUEBAS FORMALES

Pero las pruebas nunca llegaron. La jerarquía euclíadiana de teoremas nunca se construyó.

Y los programadores en general nunca vieron los beneficios de trabajar en el laborioso proceso de demostrar formalmente que todas y cada una de las pequeñas funciones son correctas. Al final, el sueño de Dijkstra se desvaneció y murió. Pocos de los programadores actuales creen que las pruebas formales sean una forma adecuada de producir software de alta calidad.

Por supuesto, las demostraciones matemáticas formales, de estilo euclíadiano, no son la única estrategia para demostrar que algo es correcto. Otra estrategia de gran éxito es el método científico.

CIENCIA AL RESCATE

La ciencia es fundamentalmente diferente de las matemáticas en que no se puede demostrar que las teorías y leyes científicas sean correctas. No puedo demostrarles que la segunda ley del movimiento de Newton, $F = ma$, o la ley de la gravedad, $F = Gm_1m_2/r^2$, les demuestren estas leyes, son correctos. Puedo y puedo hacer mediciones que las muestren correctas con muchos decimales, pero no podemos probarlos en el sentido de una prueba matemática. No importa cuántos experimentos realice o cuánta evidencia empírica recopile, siempre existe la posibilidad de que algún experimento demuestre que esas leyes del movimiento y la gravedad son incorrectas.

Ésa es la naturaleza de las teorías y leyes científicas: son falsificables pero no demostrables.

Y, sin embargo, apostamos nuestras vidas por estas leyes todos los días. Cada vez que te subes a un coche,

Puedes apostar tu vida a que $F = ma$ es una descripción confiable de cómo funciona el mundo. Cada vez que das un paso, apuestas tu salud y seguridad a que $F = Gm_1m_2/r^2$ es correcto.

La ciencia no funciona demostrando que afirmaciones sean verdaderas, sino demostrando que afirmaciones son falsas. Aquellas afirmaciones que no podemos demostrar que son falsas, después de mucho esfuerzo, las consideramos suficientemente ciertas para nuestros propósitos.

Por supuesto, no todas las afirmaciones son demostrables. La afirmación "Esto es mentira" no es ni verdadera ni falsa. Es uno de los ejemplos más simples de una afirmación que no es demostrable.

En última instancia, podemos decir que las matemáticas son la disciplina que consiste en demostrar que enunciados demostrables son verdaderos. La ciencia, por el contrario, es la disciplina que consiste en demostrar que enunciados demostrables son falsos.

PRUEBAS

Dijkstra dijo una vez: "Las pruebas muestran la presencia, no la ausencia, de errores". En otras palabras, se puede demostrar que un programa es incorrecto mediante una prueba, pero no se puede demostrar que sea correcto. Todo lo que las pruebas pueden hacer, después de un esfuerzo de prueba suficiente, es permitirnos considerar que un programa es lo suficientemente correcto para nuestros propósitos.

Las implicaciones de este hecho son asombrosas. El desarrollo de software no es una tarea matemática, aunque parezca manipular construcciones matemáticas. Más bien, el software es como una ciencia. Demostramos corrección al no demostrar lo incorrecto, a pesar de nuestros mejores esfuerzos.

Estas pruebas de incorrección sólo pueden aplicarse a programas demostrables. Un programa que no es demostrable (debido al uso desenfrenado de goto, por ejemplo) no puede considerarse correcto sin importar cuántas pruebas se le apliquen.

La programación estructurada nos obliga a descomponer recursivamente un programa en un conjunto de pequeñas funciones demostrables. Luego podemos usar pruebas para intentar demostrar que esas pequeñas funciones demostrables son incorrectas. Si dichas pruebas no demuestran que son incorrectas, entonces consideramos que las funciones son lo suficientemente correctas para nuestros propósitos.

CONCLUSIÓN

Es esta capacidad de crear unidades de programación falsificables lo que hace que la programación estructurada sea valiosa en la actualidad. Esta es la razón por la que los lenguajes modernos no suelen admitir declaraciones goto sin restricciones . Además, a nivel arquitectónico, es por eso que todavía consideramos la descomposición funcional como una de nuestras mejores prácticas.

En todos los niveles, desde la función más pequeña hasta el componente más grande, el software es como una ciencia y, por lo tanto, está impulsado por la falsabilidad. Los arquitectos de software se esfuerzan por definir módulos, componentes y servicios que sean fácilmente falsificables (comprobables). Para ello, emplean disciplinas restrictivas similares a la programación estructurada, aunque a un nivel mucho más alto.

Son esas disciplinas restrictivas las que estudiaremos con cierto detalle en los capítulos venir.

5

PROGRAMACIÓN ORIENTADA A OBJETOS



Como veremos, la base de una buena arquitectura es la comprensión y aplicación de los principios del diseño orientado a objetos (OO). Pero ¿qué es exactamente OO?

Una respuesta a esta pregunta es "La combinación de datos y función".

Aunque se cita con frecuencia, esta es una respuesta muy insatisfactoria porque implica que of() es de alguna manera diferente de f(o). Esto es absurdo. Los programadores pasaban estructuras de datos a funciones mucho antes de 1966, cuando Dahl y Nygaard trasladaron el marco de la pila de llamadas de funciones al montón e inventaron la OO.

Otra respuesta común a esta pregunta es "Una forma de modelar el mundo real".

Ésta es, en el mejor de los casos, una respuesta evasiva. ¿Qué significa realmente "modelar el mundo real"?

¿Qué significa y por qué es algo que querríamos hacer? Quizás esta afirmación pretenda dar a entender que la OO hace que el software sea más fácil de entender porque tiene una relación más estrecha con el mundo real, pero incluso esa afirmación es evasiva y está demasiado vagamente definida. No nos dice qué es OO.

Algunas personas recurren a tres palabras mágicas para explicar la naturaleza de la OO: encapsulación, herencia y polimorfismo. La implicación es que OO es la combinación adecuada de estas tres cosas, o al menos que un lenguaje OO debe soportar estas tres cosas.

Examinemos cada uno de estos conceptos por turno.

¿ENCAPSULACIÓN?

La razón por la que se cita la encapsulación como parte de la definición de OO es que los lenguajes OO proporcionan una encapsulación fácil y eficaz de datos y funciones. Como resultado, se puede trazar una línea alrededor de un conjunto cohesivo de datos y funciones. Fuera de esa línea, los datos están ocultos y sólo se conocen algunas de las funciones. Vemos este concepto en acción como los miembros de datos privados y las funciones de miembros públicos de una clase.

Esta idea ciertamente no es exclusiva de OO. De hecho, teníamos una encapsulación perfecta en C. Considere este sencillo programa en C:

[Haga clic aquí para ver la imagen del código](#)

punto.h

punto de
estructura; estructura Punto* makePoint(doble x, doble y);
doble distancia (estructura Punto *p1, estructura Punto *p2);

[Haga clic aquí para ver la imagen del código](#)

punto.c

```
#incluye "punto.h"  
#incluye <stdlib.h>  
#incluye <math.h>
```

```
punto de estructura  
{ doble x,y; };
```

```
Punto de estructura* makepoint(doble x, doble y) { Punto de
estructura* p = malloc(tamaño de(Punto de estructura)); p->x =
x; p->y = y;
devolver
p; }
```

```
doble distancia(estructura Punto* p1, estructura Punto* p2) { doble dx =
p1->x - p2->x; doble dy = p1->y -
p2->y; return sqrt(dx*dx+dy*dy); }
```

Los usuarios de point.h no tienen acceso alguno a los miembros de struct Point. Pueden llamar a la función makePoint() y a la función distancia(), pero no tienen absolutamente ningún conocimiento de la implementación de la estructura de datos Point o de las funciones.

Esta es una encapsulación perfecta, en un lenguaje que no es OO. Los programadores de C solían hacer este tipo de cosas todo el tiempo. Remitiríamos declarar estructuras de datos y funciones en archivos de encabezado y luego implementarlas en archivos de implementación. Nuestros usuarios nunca tuvieron acceso a los elementos de esos archivos de implementación.

Pero luego llegó OO en forma de C++ y se rompió la encapsulación perfecta de C.

El compilador de C++, por razones técnicas, debe [1](#) necesitaba las variables miembro de un declarar la clase en el archivo de encabezado de esa clase. Entonces nuestro programa Point cambió para verse así:

[Haga clic aquí para ver la imagen del código](#)

punto.h

```
punto de clase
{público:
Punto(doble x, doble y); doble
distancia(const Punto& p) const;

privado:
doble x;
doble y; };
```

[Haga clic aquí para ver la imagen del código](#)

punto.cc

```
#incluye "punto.h"
#include <mathematica.h>

Punto::Punto(doble x, doble y) : x(x), y(y) {}
```

```
punto doble::distancia(punto constante& p) const { doble dx =
xp.x; doble dy = yp.y;
devolver raíz cuadrada
(dx*dx + dy*dy); }
```

¡ Los clientes del archivo de encabezado point.h conocen las variables miembro xey !
 El compilador impedirá el acceso a ellos, pero el cliente aún sabe que existen.
 Por ejemplo, si se cambian los nombres de esos miembros, se debe volver a compilar el archivo point.cc . Se ha roto la encapsulación.

De hecho, la forma en que se repara parcialmente la encapsulación es introduciendo las palabras clave pública, privada y protegida en el lenguaje. Esto, sin embargo, fue un truco requerido por la necesidad técnica del compilador de ver esas variables en el archivo de encabezado.

Java y C# simplemente abolieron por completo la división encabezado/implementación, debilitando así aún más la encapsulación. En estos lenguajes, es imposible separar la declaración y la definición de una clase.

Por estas razones, es difícil aceptar que la OO dependa de una fuerte encapsulación. De hecho, muchos lenguajes OO [2](#) tienen poco o ningún cumplimiento encapsulan.

La OO ciertamente depende de la idea de que los programadores se comporten lo suficientemente bien como para no eludir los datos encapsulados. Aun así, los lenguajes que afirman proporcionar OO sólo han debilitado la encapsulación alguna vez perfecta que disfrutamos con C.

¿HERENCIA?

Si los lenguajes OO no nos dieron una mejor encapsulación, entonces ciertamente nos dieron

herencia.

Especie de. La herencia es simplemente la redeclaración de un grupo de variables y funciones dentro de un ámbito delimitado. Esto es algo que los programadores de C podían ³ fueron hacer manualmente mucho antes de que existiera un lenguaje OO.

Considere esta adición a nuestro programa point.h C original:

[Haga clic aquí para ver la imagen del código](#)

puntonombrado.h

estructura PuntoNombrado;

estructura NamedPoint* makeNamedPoint(doble x, doble y, char* nombre); void setName(struct NamedPoint* np, char* nombre); char* getName(struct NamedPoint* np);

[Haga clic aquí para ver la imagen del código](#)

puntonombrado.c

```
#include "namedPoint.h" #include
<stdlib.h>
```

```
estructura PuntoNombrado
{ double x,y;
  nombre del
  personaje; };
```

```
estructura NamedPoint* makeNamedPoint(doble x, doble y, char* nombre) { estructura NamedPoint* p =
  malloc(sizeof(struct NamedPoint)); p->x = x; p->y = y; p->nombre = nombre; devolver p; }
```

```
void setName(struct NamedPoint* np, char* nombre) { np->nombre = nombre; }
```

```
char* getName(struct NamedPoint* np) { return np->nombre; }
```

[Haga clic aquí para ver la imagen del código](#)

C Principal

```
#include "point.h"
#include "namedPoint.h"
#include <stdio.h>

int main(int ac, char** av) { estructura
NamedPoint* origen = makeNamedPoint(0.0, 0.0, "origen"); estructura NamedPoint*
superiorDerecha = makeNamedPoint (1.0, 1.0, "superiorDerecha"); printf("distancia=%f\n",
distanzia( (struct Point*)
origen,
(struct Point*) superiorDerecha)); }
```

Si observa detenidamente el programa principal , verá que la estructura de datos NamedPoint actúa como si fuera un derivado de la estructura de datos Point . Esto se debe a que el orden de los dos primeros campos en NamedPoint es el mismo que el de Point. En resumen, NamedPoint puede hacerse pasar por Point porque NamedPoint es un superconjunto puro de Point y mantiene el orden de los miembros que corresponden a Punto.

la aparición de OO. Este tipo de engaño era una [4](#) de los programadores antes de práctica común. De hecho, ese truco es cómo C++ implementa la herencia única.

Así, podríamos decir que tuvimos una especie de herencia mucho antes de que se inventaran los lenguajes OO. Sin embargo, esa afirmación no sería del todo cierta. Teníamos un truco, pero no es tan conveniente como la herencia verdadera. Además, la herencia múltiple es mucho más difícil de lograr mediante este tipo de engaños.

Tenga en cuenta también que en main.c, me vi obligado a convertir los argumentos de NamedPoint en Point. En un lenguaje OO real, dicha conversión estaría implícita.

Es justo decir que si bien los lenguajes OO no nos brindaron algo completamente nuevo, sí hicieron que el enmascaramiento de estructuras de datos fuera significativamente más conveniente.

En resumen: no podemos otorgar ningún punto a OO por encapsulación, y quizás medio punto por herencia. Hasta ahora, ese no es un puntaje tan bueno.

Pero hay un atributo más a considerar.

¿POLIMORFISMO?

¿Teníamos un comportamiento polimórfico antes de los lenguajes OO? Por supuesto que lo hicimos. Considere este sencillo programa de copia en C.

[Haga clic aquí para ver la imagen del código](#)

```
#incluir <stdio.h>
```

```
copia nula() { int
c;
mientras ((c=getchar()) != EOF)
putchar(c); }
```

La función getchar() lee desde STDIN. ¿ Pero qué dispositivo es STDIN? La función putchar() escribe en STDOUT. ¿Pero qué dispositivo es ese? Estas funciones son polimórficas: su comportamiento depende del tipo de STDIN y STDOUT.

Es como si STDIN y STDOUT fueran interfaces estilo Java que tienen implementaciones para cada dispositivo. Por supuesto, no hay interfaces en el programa C de ejemplo; entonces, ¿cómo se entrega realmente la llamada a getchar() al controlador del dispositivo que lee el carácter?

La respuesta a esa pregunta es bastante sencilla. El sistema operativo UNIX requiere que cada controlador de dispositivo IO proporcione cinco funciones estándar: [cinco](#) abrir, cerrar, leer, escribir y buscar. Las firmas de esas funciones deben ser idénticas para cada controlador IO.

La estructura de datos ARCHIVO contiene cinco punteros a funciones. En nuestro ejemplo, podría verse así:

[Haga clic aquí para ver la imagen del código](#)

```
estructura
ARCHIVO { void (*open)(char* nombre, modo
int); vacío (*cerrar)();
int (*leer)(); vacío
(*escribir)(char); void
(*seek)(índice largo, modo int); };
```

El controlador IO para la consola definirá esas funciones y cargará una estructura de datos de ARCHIVO con sus direcciones, algo como esto:

[Haga clic aquí para ver la imagen del código](#)

```
#incluir "archivo.h"

void open(char* nombre, modo int) {/*...*/} void close() {/
*...*/}; int read() {int c; /*...*/ return
c;} void write(char c) {/*...*/} void seek(índice largo,
modo int) {/*... */}
```

```
estructura ARCHIVO consola = {abrir, cerrar, leer, escribir, buscar};
```

Ahora bien, si STDIN se define como FILE*, y si apunta a la estructura de datos de la consola, entonces getchar() podría implementarse de esta manera:

[Haga clic aquí para ver la imagen del código](#)

```
estructura externa ARCHIVO* STDIN;
```

```
int getchar() { return
STDIN->read(); }
```

En otras palabras, getchar() simplemente llama a la función apuntada por el puntero de lectura de la estructura de datos FILE apuntada por STDIN.

Este sencillo truco es la base de todo polimorfismo en OO. En C++, por ejemplo, cada función virtual dentro de una clase tiene un puntero en una tabla llamada vtable, y todas las llamadas a funciones virtuales pasan por esa tabla. Los constructores de derivadas simplemente cargan sus versiones de esas funciones en la tabla virtual del objeto que se está creando.

La conclusión es que el polimorfismo es una aplicación de punteros a funciones.

Los programadores han estado utilizando punteros a funciones para lograr un comportamiento polimórfico desde que se implementaron por primera vez las arquitecturas de Von Neumann a finales de la década de 1940. En otras palabras, OO no ha aportado nada nuevo.

Ah, pero eso no es del todo correcto. Es posible que los lenguajes OO no nos hayan brindado polimorfismo, pero lo han hecho mucho más seguro y conveniente.

El problema de utilizar explícitamente punteros a funciones para crear un comportamiento polimórfico es que los punteros a funciones son peligrosos. Este uso está impulsado por un conjunto de convenciones manuales. Tienes que recordar seguir la convención para

inicializar esos punteros. Debe recordar seguir la convención para llamar a todas sus funciones a través de esos punteros. Si algún programador no recuerda estas convenciones, el error resultante puede ser tremadamente difícil de localizar y eliminar.

Los lenguajes OO eliminan estas convenciones y, por tanto, estos peligros. El uso de un lenguaje OO hace que el polimorfismo sea trivial. Este hecho proporciona un poder enorme con el que los viejos programadores de C sólo podían soñar. Sobre esta base, podemos concluir que OO impone disciplina a la transferencia indirecta de control.

EL PODER DEL POLIMORFISMO

¿Qué tiene de bueno el polimorfismo? Para apreciar mejor sus encantos, reconsideraremos el programa de copia de ejemplo . ¿Qué sucede con ese programa si se crea un nuevo dispositivo IO? Supongamos que queremos usar el programa de copia para copiar datos desde un dispositivo de reconocimiento de escritura a un dispositivo sintetizador de voz: ¿Cómo necesitamos cambiar el programa de copia para que funcione con esos nuevos dispositivos?

¡No necesitamos ningún cambio en absoluto! De hecho, ni siquiera necesitamos recompilar el programa de copia . ¿Por qué? Porque el código fuente del programa de copia no depende del código fuente de los controladores IO. Siempre que esos controladores IO implementen las cinco funciones estándar definidas por FILE, el programa de copia estará encantado de utilizarlas.

En resumen, los dispositivos IO se han convertido en complementos del programa de copia .

¿Por qué el sistema operativo UNIX creó complementos para dispositivos IO? Porque aprendimos, a finales de la década de 1950, que nuestros programas deberían ser independientes del dispositivo. ¿Por qué? Porque escribimos muchos programas que dependían del dispositivo, sólo para descubrir que realmente queríamos que esos programas hicieran el mismo trabajo pero usaran un dispositivo diferente.

Por ejemplo, a menudo escribíamos programas que leían datos de entrada de mazos de cartas y luego perforaban nuevos mazos de cartas como salida. Posteriormente, nuestros clientes dejaron de regalarnos barajas de cartas y empezaron a regalarnos bobinas de cinta magnética. Esto era muy inconveniente porque implicaba reescribir grandes porciones del programa original. Sería muy conveniente que un mismo programa funcionara indistintamente con tarjetas o cintas. 6

La arquitectura del complemento se inventó para admitir este tipo de independencia del dispositivo IO y se ha implementado en casi todos los sistemas operativos desde su introducción. Aun así, la mayoría de los programadores no extendieron la idea a sus propios programas, porque usar punteros a funciones era peligroso.

OO permite que la arquitectura del complemento se utilice en cualquier lugar y para cualquier cosa.

INVERSIÓN DE DEPENDENCIA

Imagínese cómo era el software antes de que estuviera disponible un mecanismo seguro y conveniente para el polimorfismo. En el árbol de llamadas típico, las funciones principales se denominan funciones de alto nivel, las cuales se denominan funciones de nivel medio y las que se denominan funciones de bajo nivel. Sin embargo, en ese árbol de llamadas, las dependencias del código fuente seguían inexorablemente el flujo de control (Figura 5.1).

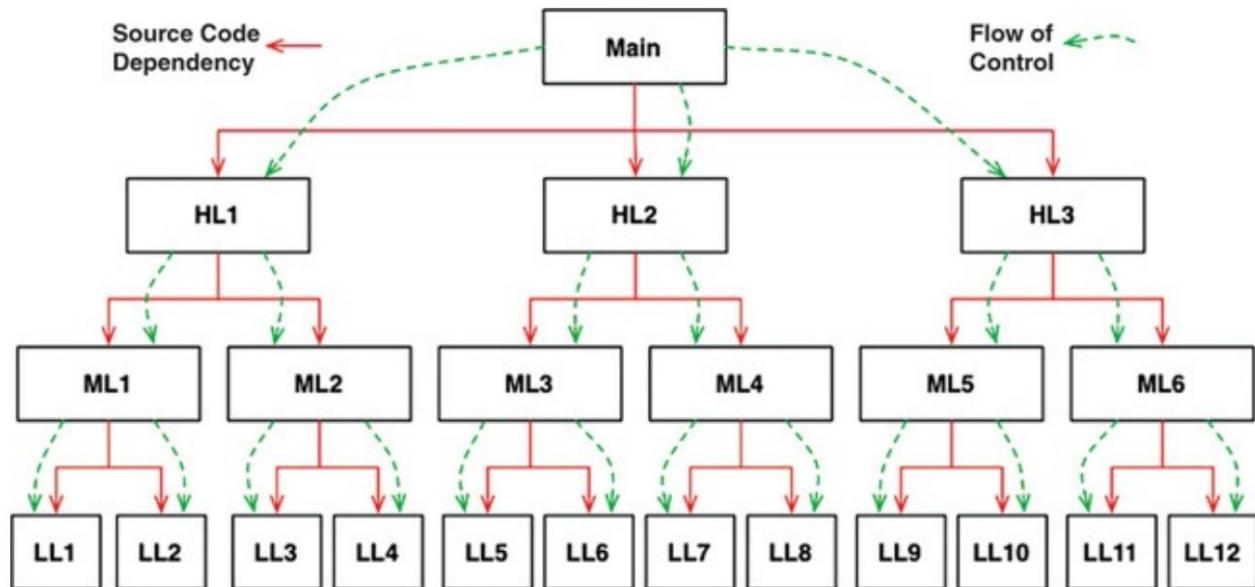


Figura 5.1 Dependencias del código fuente versus flujo de control

Para que main llame a una de las funciones de alto nivel, tenía que mencionar el nombre del módulo que contenía esa función. En C, esto era `#include .` En Java, era una declaración de importación `.` En C#, era una declaración de uso `.` De hecho, cada persona que llamaba se veía obligada a mencionar el nombre del módulo que contenía a la persona que llamaba.

Este requisito presentó al arquitecto de software pocas opciones, si es que alguna. El flujo de control estaba dictado por el comportamiento del sistema, y las dependencias del código fuente estaban dictadas por ese flujo de control.

Sin embargo, cuando entra en juego el polimorfismo, puede suceder algo muy diferente (Figura 5.2).

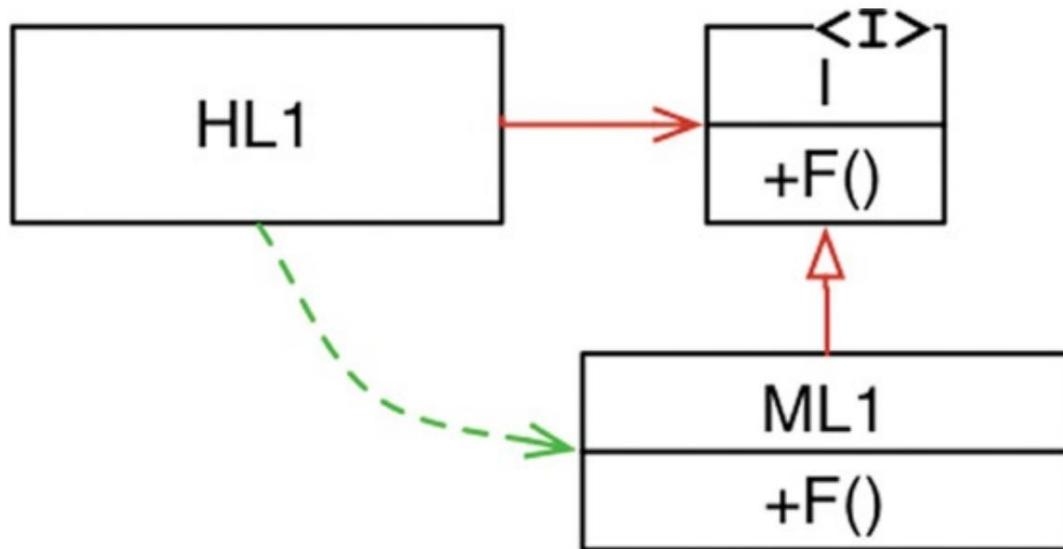


Figura 5.2 Inversión de dependencia

En [la Figura 5.2](#), el módulo HL1 llama a la función F() en el módulo ML1. El hecho de que llame a esta función a través de una interfaz es un invento del código fuente. En tiempo de ejecución, la interfaz no existe. HL1 simplemente llama a F() dentro de ML1. [7](#)

Sin embargo, tenga en cuenta que la dependencia del código fuente (la relación de herencia) entre ML1 y la interfaz I apunta en la dirección opuesta en comparación con el flujo de control. Esto se llama inversión de dependencia y sus implicaciones para el arquitecto de software son profundas.

El hecho de que los lenguajes OO proporcionen un polimorfismo seguro y conveniente significa que cualquier dependencia del código fuente, sin importar dónde se encuentre, se puede invertir.

Ahora mire nuevamente el árbol de llamadas de [la Figura 5.1](#) y sus numerosas dependencias del código fuente. Cualquiera de esas dependencias del código fuente se puede revertir insertando una interfaz entre ellas.

Con este enfoque, los arquitectos de software que trabajan en sistemas escritos en lenguajes OO tienen control absoluto sobre la dirección de todas las dependencias del código fuente en el sistema. No están obligados a alinear esas dependencias con el flujo de control. No importa qué módulo realiza la llamada y qué módulo se llama, el arquitecto de software puede señalar el código fuente

dependencia en cualquier dirección.

¡Eso es poder! Ese es el poder que proporciona OO. De eso se trata realmente la OO, al menos desde el punto de vista del arquitecto.

¿Qué puedes hacer con ese poder? Como ejemplo, puede reorganizar las dependencias del código fuente de su sistema para que la base de datos y la interfaz de usuario (UI) dependan de las reglas comerciales ([Figura 5.3](#)), y no al revés.

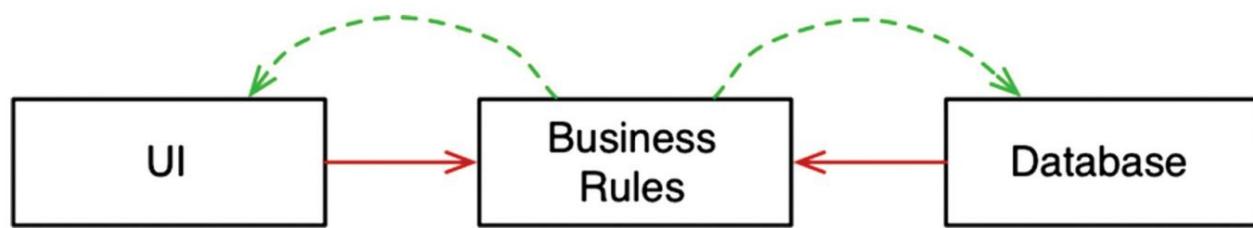


Figura 5.3 La base de datos y la interfaz de usuario dependen de las reglas comerciales

Esto significa que la interfaz de usuario y la base de datos pueden ser complementos de las reglas comerciales. Significa que el código fuente de las reglas comerciales nunca menciona la interfaz de usuario ni la base de datos.

Como consecuencia, las reglas de negocio, la interfaz de usuario y la base de datos se pueden compilar en tres componentes o unidades de implementación separados (por ejemplo, archivos jar, DLL o archivos Gem) que tienen las mismas dependencias que el código fuente. El componente que contiene las reglas comerciales no dependerá de los componentes que contienen la interfaz de usuario y la base de datos.

A su vez, las reglas de negocio se pueden implementar independientemente de la interfaz de usuario y la base de datos. Los cambios en la interfaz de usuario o la base de datos no tienen por qué tener ningún efecto en las reglas comerciales. Esos componentes se pueden implementar por separado e independientemente.

En resumen, cuando el código fuente de un componente cambia, solo es necesario volver a implementar ese componente. Esta es la capacidad de implementación independiente.

Si los módulos de su sistema se pueden implementar de forma independiente, entonces diferentes equipos pueden desarrollarlos de forma independiente. Eso es desarrollo independiente.

CONCLUSIÓN

¿Qué es OO? Hay muchas opiniones y muchas respuestas a esta pregunta. Para el arquitecto de software, sin embargo, la respuesta es clara: OO es la capacidad, mediante el uso de polimorfismo, de obtener control absoluto sobre cada dependencia del código fuente en el sistema. Permite al arquitecto crear una arquitectura de complementos, en la que los módulos que contienen políticas de alto nivel son independientes de los módulos que contienen detalles de bajo nivel. Los detalles de bajo nivel quedan relegados a módulos complementarios que se pueden implementar y desarrollar independientemente de los módulos que contienen políticas de alto nivel.

1. El compilador de C++ necesita saber el tamaño de las instancias de cada clase.

2. Por ejemplo, Smalltalk, Python, JavaScript, Lua y Ruby.

3. No sólo los programadores de C: la mayoría de los lenguajes de esa época tenían la capacidad de enmascarar una estructura de datos.
como otro.

4. De hecho, todavía lo es.

5. Los sistemas UNIX varían; este es sólo un ejemplo.

6. Tarjetas perforadas: tarjetas IBM Hollerith, 80 columnas de ancho. Estoy seguro de que muchos de ustedes nunca han visto uno.
de estos, pero eran comunes en los años cincuenta, sesenta e incluso setenta.

7. Aunque sea indirectamente.

6

PROGRAMACIÓN FUNCIONAL



En muchos sentidos, los conceptos de programación funcional son anteriores a la programación misma. Este paradigma está fuertemente basado en el cálculo λ inventado por Alonzo Church en la década de 1930.

CUADRADOS DE ENTEROS

Para explicar qué es la programación funcional, lo mejor es examinar algunos ejemplos. Investiguemos un problema simple: imprimir los cuadrados de los primeros 25 números enteros.

En un lenguaje como Java, podríamos escribir lo siguiente:

[Haga clic aquí para ver la imagen del código](#)

```
clase pública Entrecerrar los
ojos { public static void main(String args[]) { for (int i=0;
i<25; i++)
System.out.println(i*i); } }
```

En un lenguaje como Clojure, que es un derivado de Lisp y es funcional, podríamos implementar este mismo programa de la siguiente manera:

[Haga clic aquí para ver la imagen del código](#)

```
(println (tomar 25 (mapa (fn [x] (* xx)) (rango))))
```

Si no conoces Lisp, esto puede parecer un poco extraño. Déjame reformatearlo un poco y agregar algunos comentarios.

[Haga clic aquí para ver la imagen del código](#)

```
(println ;_____ Imprimir (tomar
25 ;_____ los primeros 25 (map (fn [x]
(* xx)) ;__ cuadrados (rango))) ;_____
de Enteros
```

Debe quedar claro que `println`, `take`, `map` y `range` son todas funciones. En Lisp, llamas a una función poniéndola entre paréntesis. Por ejemplo, `(rango)` llama a la función de `rango`.

La expresión `(fn [x] (* xx))` es una función anónima que llama a la función multiplicar y pasa su argumento de entrada dos veces. En otras palabras, calcula el cuadrado de su entrada.

Mirando todo nuevamente, es mejor comenzar con la llamada a función más interna.

- La función de `rango` devuelve una lista interminable de números enteros que comienzan con 0.
- Esta lista se pasa a la función de `mapa` , que llama a la función de cuadratura anónima en cada elemento, produciendo una nueva lista interminable de todos los cuadrados.
- La lista de cuadrados se pasa a la función `take` , que devuelve una nueva lista con sólo los primeros 25 elementos.

- La función `println` imprime su entrada, que es una lista de los primeros 25 cuadrados de números enteros.

Si le aterriza el concepto de listas interminables, no se preocupe.

En realidad, sólo se crean los primeros 25 elementos de esas listas interminables. Esto se debe a que ningún elemento de una lista interminable se evalúa hasta que se accede a él.

Si todo eso le resultó confuso, puede esperar pasar un momento glorioso aprendiendo todo sobre Clojure y la programación funcional. No es mi objetivo enseñarte sobre estos temas aquí.

En cambio, mi objetivo aquí es señalar algo muy dramático acerca de la diferencia entre los programas Clojure y Java. El programa Java utiliza una variable mutable, una variable que cambia de estado durante la ejecución del programa. Esa variable es `i`, la variable de control del bucle. No existe tal variable mutable en el programa Clojure. En el programa Clojure, variables como `x` se inicializan, pero nunca se modifican.

Esto nos lleva a una afirmación sorprendente: las variables en los lenguajes funcionales no varían.

INMUTABILIDAD Y ARQUITECTURA

¿Por qué es importante este punto como consideración arquitectónica? ¿Por qué un arquitecto estaría preocupado por la mutabilidad de las variables? La respuesta es absurdamente simple: todas las condiciones de carrera, condiciones de interbloqueo y problemas de actualización simultánea se deben a variables mutables. No puede tener una condición de carrera o un problema de actualización simultánea si nunca se actualiza ninguna variable. No se pueden tener puntos muertos sin bloqueos mutables.

En otras palabras, todos los problemas que enfrentamos en aplicaciones concurrentes (todos los problemas que enfrentamos en aplicaciones que requieren múltiples subprocessos y múltiples procesadores) no pueden ocurrir si no hay variables mutables.

Como arquitecto, debería estar muy interesado en las cuestiones de concurrencia. Quiere asegurarse de que los sistemas que diseñe sean robustos en presencia de múltiples subprocessos y procesadores. La pregunta que usted debe hacerse, entonces, es si la inmutabilidad es practicable.

La respuesta a esa pregunta es afirmativa, si tienes almacenamiento infinito y velocidad de procesador infinita. Al carecer de esos recursos infinitos, la respuesta es un poco más matizada. Sí, la inmutabilidad puede ser practicable, si se hacen ciertas concesiones.

Veamos algunos de esos compromisos.

SEGREGACIÓN DE LA MUTABILIDAD

Uno de los compromisos más comunes con respecto a la inmutabilidad es separar la aplicación, o los servicios dentro de la aplicación, en componentes mutables e inmutables. Los componentes inmutables realizan sus tareas de forma puramente funcional, sin utilizar variables mutables. Los componentes inmutables se comunican con uno o más componentes que no son puramente funcionales y permiten que el estado de las variables mute ([Figura 6.1](#)).

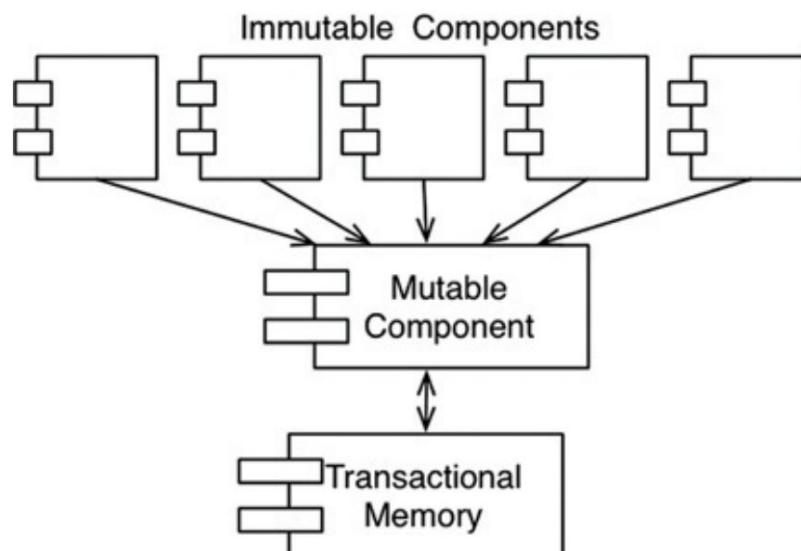


Figura 6.1 Estado mutante y memoria transaccional

Dado que el estado mutante expone esos componentes a todos los problemas de concurrencia, es una práctica común utilizar algún tipo de memoria transaccional para proteger las variables mutables de actualizaciones simultáneas y condiciones de carrera.

La memoria transaccional simplemente trata las variables en la memoria de la misma manera que una base de datos trata los registros en el disco. [1](#) Protege esas variables con una transacción o esquema basado en reintentos.

Un ejemplo sencillo de este enfoque es la instalación atómica de Clojure :

[Haga clic aquí para ver la imagen del código](#)

```
(contador def (átomo 0)); inicializar el contador a 0 (¡intercambiar! contador inc); incrementar el contador de forma segura.
```

En este código, la variable contador se define como un átomo. En Clojure, un átomo es un tipo especial de variable cuyo valor puede mutar bajo condiciones muy disciplinadas que se imponen mediante el intercambio. función.

¡ El intercambio! La función, que se muestra en el código anterior, toma dos argumentos: el átomo que se va a mutar y una función que calcula el nuevo valor que se almacenará en el átomo.

En nuestro código de ejemplo, el átomo contador se cambiará al valor calculado por la función inc , que simplemente incrementa su argumento.

La estrategia utilizada por swap! Es un algoritmo tradicional de comparación e intercambio . El valor del contador se lee y se pasa a inc. Cuando inc regresa, el valor del contador se bloquea y se compara con el valor que se pasó a inc. Si el valor es el mismo, entonces el valor devuelto por inc se almacena en el contador y se libera el bloqueo. De lo contrario, se libera el bloqueo y se vuelve a intentar la estrategia desde el principio.

La instalación atómica es adecuada para aplicaciones simples. Desafortunadamente, no puede proteger completamente contra actualizaciones simultáneas y bloqueos cuando entran en juego múltiples variables dependientes. En esos casos, se pueden utilizar instalaciones más elaboradas.

La cuestión es que las aplicaciones bien estructuradas se segregarán en aquellos componentes que no mutan las variables y los que sí lo hacen. Este tipo de segregación está respaldado por el uso de disciplinas apropiadas para proteger esas variables mutadas.

Los arquitectos harían bien en introducir tanto procesamiento como sea posible en los componentes inmutables y extraer la mayor cantidad de código posible de aquellos componentes que deben permitir la mutación.

OBTENCIÓN DE EVENTOS

Los límites del poder de almacenamiento y procesamiento se han ido alejando rápidamente de la vista. Hoy en día es habitual que los procesadores ejecuten miles de millones de instrucciones por segundo y tengan miles de millones de bytes de RAM. Cuanta más memoria tengamos y más rápidas sean nuestras máquinas, menos necesitamos estados mutables.

Como ejemplo sencillo, imagine una aplicación bancaria que mantiene los saldos de las cuentas de sus clientes. Muta esos saldos cuando se ejecutan transacciones de depósito y retiro.

Ahora imagine que en lugar de almacenar los saldos de las cuentas, almacenamos solo las transacciones. Siempre que alguien quiere saber el saldo de una cuenta, simplemente sumamos todas las transacciones de esa cuenta, desde el principio de los tiempos. Este esquema no requiere variables mutables.

Evidentemente, este enfoque suena absurdo. Con el tiempo, el número de transacciones crecería sin límites y la potencia de procesamiento necesaria para calcular los totales se volvería intolerable. Para que este esquema funcione para siempre, necesitaríamos almacenamiento infinito y potencia de procesamiento infinita.

Pero tal vez no tengamos que hacer que el plan funcione para siempre. Y tal vez tengamos suficiente almacenamiento y suficiente potencia de procesamiento para que el esquema funcione durante la vida útil razonable de la aplicación.

Esta es la idea detrás del abastecimiento de eventos.² El abastecimiento de eventos es una estrategia en la que almacenamos las transacciones, pero no el estado. Cuando se requiere estado, simplemente aplicamos todas las transacciones desde el principio de los tiempos.

Por supuesto, podemos tomar atajos. Por ejemplo, podemos calcular y guardar el estado cada medianoche. Luego, cuando se requiere la información del estado, necesitamos calcular solo las transacciones desde la medianoche.

Consideremos ahora el almacenamiento de datos requerido para este esquema: necesitaríamos mucho. Siendo realistas, el almacenamiento de datos fuera de línea ha crecido tan rápido que ahora consideramos que billones de bytes son pequeños, por lo que tenemos muchos.

Más importante aún, nunca se elimina ni actualiza nada de dicho almacén de datos. Como consecuencia, nuestras aplicaciones no son CRUD; son solo CR. Además, como no se producen actualizaciones ni eliminaciones en el almacén de datos, no puede haber problemas de actualización simultánea.

Si tenemos suficiente almacenamiento y suficiente potencia de procesador, podemos hacer que nuestras aplicaciones sean completamente inmutables y, por lo tanto, completamente funcionales.

Si esto todavía suena absurdo, podría ser útil recordar que así es precisamente como funciona su sistema de control de código fuente.

CONCLUSIÓN

Para resumir:

- La programación estructurada es una disciplina impuesta sobre la transferencia directa de control.
- La programación orientada a objetos es una disciplina impuesta sobre la transferencia indirecta de control.
- La programación funcional es una disciplina impuesta sobre la asignación de variables.

Cada uno de estos tres paradigmas nos ha quitado algo. Cada uno restringe algún aspecto de la forma en que escribimos código. Ninguno de ellos ha aumentado nuestro poder o nuestras capacidades.

Lo que hemos aprendido durante el último medio siglo es lo que no debemos hacer.

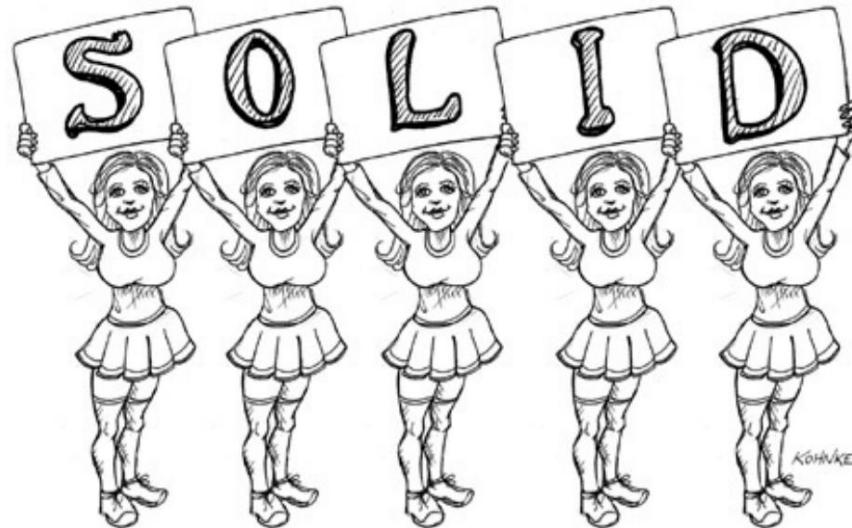
Al darnos cuenta de esto, tenemos que enfrentar un hecho desagradable: el software no es una tecnología que avanza rápidamente. Las reglas del software son las mismas hoy que en 1946, cuando Alan Turing escribió el primer código que se ejecutaría en una computadora electrónica. Las herramientas han cambiado y el hardware ha cambiado, pero la esencia del software sigue siendo la misma.

El software (el material de los programas de computadora) se compone de secuencia, selección, iteración y dirección indirecta. Nada mas. Nada menos.

1. Lo sé... ¿Qué es un disco?
2. Gracias a Greg Young por enseñarme sobre este concepto.

III

CRITERIOS DE DISEÑO



Los buenos sistemas de software comienzan con un código limpio. Por un lado, si los ladrillos no están bien hechos, la arquitectura del edificio no importa mucho. Por otro lado, puedes causar un gran desastre con ladrillos bien hechos. Aquí es donde entran los principios SÓLIDOS.

Los principios SOLID nos dicen cómo organizar nuestras funciones y estructuras de datos en clases, y cómo esas clases deben interconectarse. El uso de la palabra "clase" no implica que estos principios sean aplicables únicamente al software orientado a objetos. Una clase es simplemente una agrupación acoplada de funciones y datos. Cada sistema de software tiene tales agrupaciones, ya sea que se llamen clases o no. Los principios SOLID se aplican a esas agrupaciones.

El objetivo de los principios es la creación de estructuras de software de nivel medio que:

- Toleran el cambio,
- Son fáciles de entender, y • Son

la base de componentes que pueden usarse en muchos sistemas de software.

El término "nivel medio" se refiere al hecho de que estos principios los aplican los programadores que trabajan a nivel de módulo. Se aplican justo encima del nivel del código y ayudan a definir los tipos de estructuras de software utilizadas dentro de los módulos y componentes.

Así como es posible crear un desorden sustancial con ladrillos bien hechos, también es posible crear un desorden en todo el sistema con componentes de nivel medio bien diseñados. Por esta razón, una vez que hayamos cubierto los principios SOLID, pasaremos a sus contrapartes en el mundo de los componentes y luego a los principios de la arquitectura de alto nivel.

La historia de los principios SOLID es larga. Comencé a ensamblarlos a fines de la década de 1980, mientras debatía principios de diseño de software con otros en USENET (una de las primeras formas de Facebook). A lo largo de los años, los principios han cambiado y cambiado. Algunos fueron eliminados. Otros se fusionaron. Aún se agregaron otros. El grupo final se estabilizó a principios de la década de 2000, aunque los presenté en un orden diferente.

Alrededor de 2004, Michael Feathers me envió un correo electrónico diciendo que si reorganizaba los principios, sus primeras palabras formarían la palabra SÓLIDO, y así nacieron los principios SÓLIDOS.

Los capítulos que siguen describen cada principio más detalladamente. Aquí está el resumen ejecutivo:

- SRP: El principio de responsabilidad única Un corolario activo de la ley de Conway: la mejor estructura para un sistema de software está fuertemente influenciada por la estructura social de la organización que lo utiliza, de modo que cada módulo de software tiene una, y sólo una, razón para cambiar. •

OCP: El principio abierto-cerrado

Bertrand Meyer hizo famoso este principio en los años 1980. La esencia es que para que los sistemas de software sean fáciles de cambiar, deben diseñarse para permitir la

el comportamiento de esos sistemas se puede cambiar añadiendo nuevo código, en lugar de cambiar el código existente.

- LSP: El principio de sustitución de Liskov La famosa definición de subtipos de Barbara Liskov, de 1988. En resumen, este principio dice que para construir sistemas de software a partir de partes intercambiables, esas partes deben cumplir un contrato que permita sustituirlas una por otra.

- ISP: El principio de segregación de interfaces Este principio aconseja a los diseñadores de software evitar depender de cosas que no utilizan. •

DIP: El principio de inversión de dependencia El código que implementa políticas de alto nivel no debe depender del código que implementa detalles de bajo nivel. Más bien, los detalles deberían depender de las políticas.

Estos principios se han descrito en detalle en muchas publicaciones diferentes a lo largo de los ¹ años. Los capítulos que siguen se centrarán en las implicaciones arquitectónicas de estos principios en lugar de repetir esas discusiones detalladas. Si aún no está familiarizado con estos principios, lo que sigue es insuficiente para comprenderlos en detalle y haría bien en estudiarlos en los documentos con notas a pie de página.

¹. Por ejemplo, Principios, patrones y prácticas de desarrollo de software ágil , Robert C. Martin, Prentice Salón, 2002, <http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOOD>, y [https://en.wikipedia.org/wiki/SOLID_\(dise%C3%B1o_orientado_a_objetos\)](https://en.wikipedia.org/wiki/SOLID_(dise%C3%B1o_orientado_a_objetos)) (o simplemente google SÓLIDO).

7

SRP: LA ÚNICA RESPONSABILIDAD PRINCIPIO



De todos los principios SOLID, el Principio de Responsabilidad Única (SRP) podría ser el menos comprendido. Probablemente se deba a que tiene un nombre particularmente inapropiado. Es demasiado fácil para los programadores escuchar el nombre y luego asumir que significa que cada módulo debe hacer solo una cosa.

No se equivoque, existe un principio como ese. Una función debe hacer una y sólo una cosa. Usamos ese principio cuando refactorizamos funciones grandes en funciones más pequeñas; Lo usamos en los niveles más bajos. Pero no es uno de los principios SOLID, no es el SRP.

Históricamente, el SRP se ha descrito de esta manera:

Un módulo debe tener una, y sólo una, razón para cambiar.

Los sistemas de software se modifican para satisfacer a los usuarios y partes interesadas; esos usuarios y partes interesadas son la “razón para el cambio” de la que habla el principio.

De hecho, podemos reformular el principio para decir esto:

Un módulo debe ser responsable ante un, y sólo uno, usuario o parte interesada.

Desafortunadamente, las palabras "usuario" y "parte interesada" no son realmente las palabras correctas para usar aquí. Probablemente habrá más de un usuario o parte interesada que desee que el sistema cambie de la misma manera. Más bien, en realidad nos estamos refiriendo a un grupo: una o más personas que requieren ese cambio. Nos referiremos a ese grupo como un actor.

Así, la versión final del SRP es:

Un módulo debe ser responsable ante un, y sólo un, actor.

Ahora bien, ¿qué queremos decir con la palabra “módulo”? La definición más simple es simplemente un archivo fuente. La mayoría de las veces esa definición funciona bien. Sin embargo, algunos lenguajes y entornos de desarrollo no utilizan archivos fuente para contener su código.

En esos casos, un módulo es simplemente un conjunto cohesivo de funciones y estructuras de datos.

Esa palabra "cohesivo" implica el SRP. La cohesión es la fuerza que une el código responsable de un único actor.

Quizás la mejor manera de entender este principio sea observando los síntomas de su violación.

SÍNTOMA 1: ACCIDENTAL DUPLICACIÓN

Mi ejemplo favorito es la clase Empleado de una aplicación de nómina. Tiene tres métodos: calcularPago(), reportHours() y guardar() ([Figura 7.1](#)).

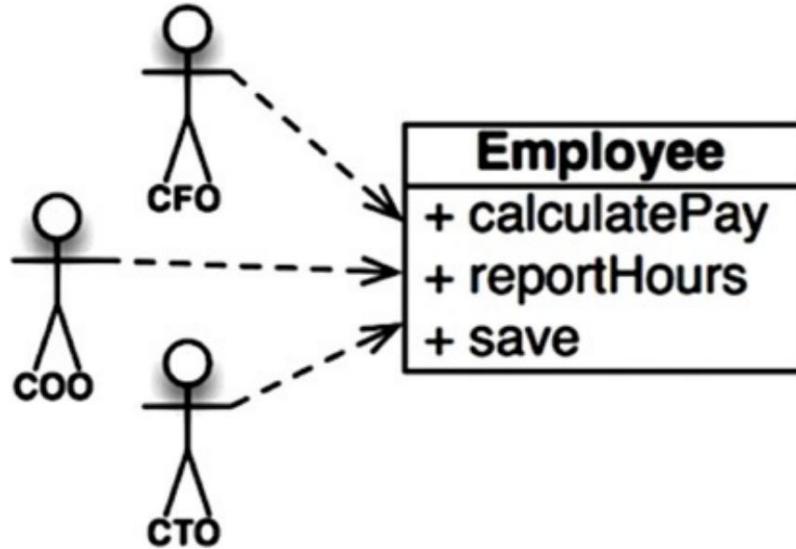


Figura 7.1 La clase Empleado

Esta clase viola el SRP porque esos tres métodos son responsables ante tres actores muy diferentes.

- El método `calculPay()` lo especifica el departamento de contabilidad, que depende del director financiero.
- El método `reportHours()` lo especifica y utiliza el departamento de recursos humanos, que depende del director de operaciones. • El método `save()` lo especifican los administradores de bases de datos (DBA), que reportan al CTO.

Al colocar el código fuente de estos tres métodos en una única clase de Empleado , los desarrolladores han acoplado cada uno de estos actores a los demás. Este acoplamiento puede hacer que las acciones del equipo del CFO afecten algo de lo que depende el equipo del COO.

Por ejemplo, supongamos que la función `calcularPago()` y la función `reportHours()` comparten un algoritmo común para calcular las horas que no son horas extra. Supongamos también que los desarrolladores, que tienen cuidado de no duplicar el código, colocan ese algoritmo en una función denominada `regularHours()` ([Figura 7.2](#)).

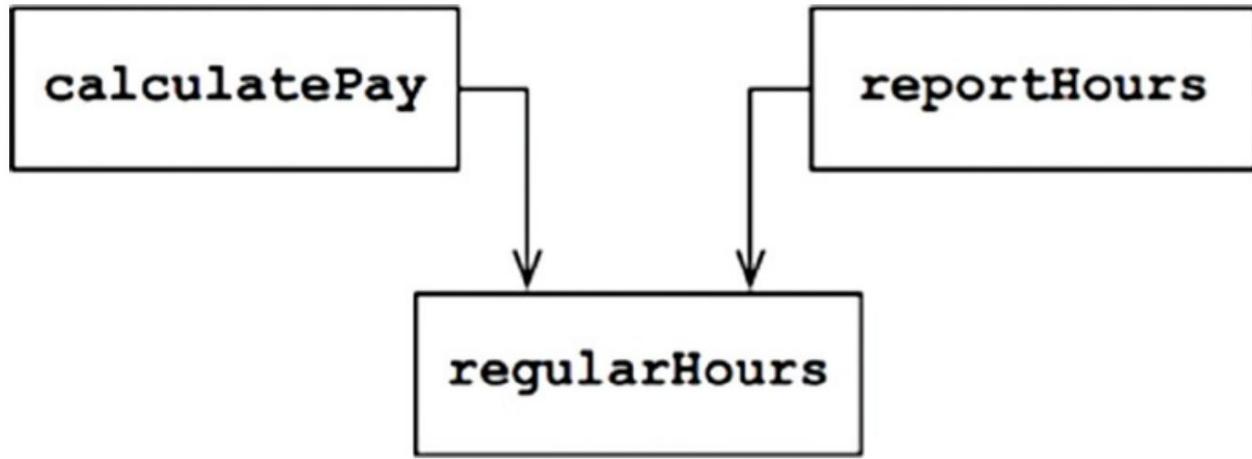


Figura 7.2 Algoritmo compartido

Ahora supongamos que el equipo del director financiero decide que es necesario modificar la forma en que se calculan las horas no extras. Por el contrario, el equipo de recursos humanos del director de operaciones no quiere ese ajuste en particular porque utilizan las horas que no son horas extras para un propósito diferente.

Un desarrollador tiene la tarea de realizar el cambio y ve la conveniente función `regularHours()` llamada por el método `calcularPay()`. Desafortunadamente, ese desarrollador no se da cuenta de que la función `reportHours()` también llama a la función.

El desarrollador realiza el cambio requerido y lo prueba cuidadosamente. El equipo del director financiero valida que la nueva función funcione según lo deseado y se implementa el sistema.

Por supuesto, el equipo del director de operaciones no sabe que esto está sucediendo. El personal de RR.HH. continúa utilizando los informes generados por la función `reportHours()`, pero ahora contienen números incorrectos. Finalmente se descubre el problema y el director de operaciones está furioso porque los datos incorrectos le han costado a su presupuesto millones de dólares.

Todos hemos visto suceder cosas como esta. Estos problemas ocurren porque ponemos el código del que dependen diferentes actores muy cerca. El SRP dice que se separe el código del que dependen los diferentes actores.

SÍNTOMA 2: FUSIONES

No es difícil imaginar que las fusiones serán comunes en archivos fuente que contienen muchos métodos diferentes. Esta situación es especialmente probable si esos métodos son responsables ante diferentes actores.

Por ejemplo, supongamos que el equipo de DBA del CTO decide que debe haber un cambio de esquema simple en la tabla Empleado de la base de datos. Supongamos también que el equipo de empleados de recursos humanos del director de operaciones decide que necesita un cambio en el formato del informe de horas.

Dos desarrolladores diferentes, posiblemente de dos equipos diferentes, revisan la clase Empleado y comienzan a realizar cambios. Desafortunadamente sus cambios chocan.

El resultado es una fusión.

Probablemente no necesito decíles que las fusiones son asuntos riesgosos. Nuestras herramientas son bastante buenas hoy en día, pero ninguna herramienta puede abordar todos los casos de fusión. Al final siempre existe el riesgo.

En nuestro ejemplo, la fusión pone en riesgo tanto al CTO como al COO. No es inconcebible que el director financiero también pueda verse afectado.

Hay muchos otros síntomas que podríamos investigar, pero todos implican que varias personas cambian el mismo archivo fuente por diferentes motivos.

Una vez más, la forma de evitar este problema es separar el código que admite diferentes actores.

SOLUCIONES

Hay muchas soluciones diferentes a este problema. Cada uno mueve las funciones a diferentes clases.

Quizás la forma más obvia de resolver el problema sea separar los datos de las funciones. Las tres clases comparten acceso a EmployeeData, que es una estructura de datos simple sin métodos ([Figura 7.3](#)). Cada clase contiene sólo el código fuente necesario para su función particular. Las tres clases no pueden conocerse entre sí. De este modo se evita cualquier duplicación accidental.

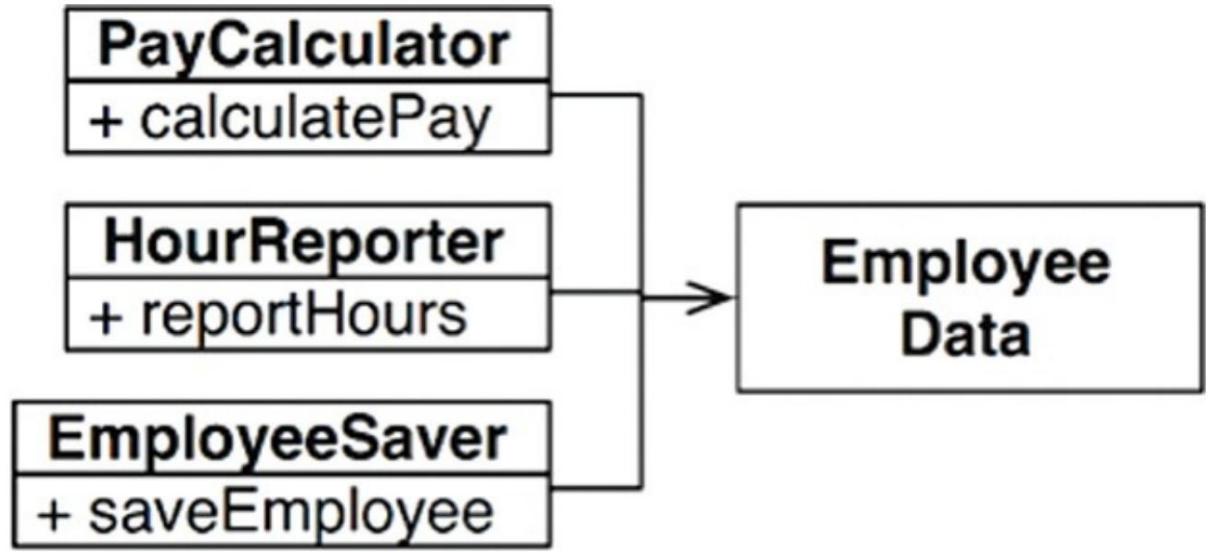


Figura 7.3 Las tres clases no se conocen entre sí

La desventaja de esta solución es que los desarrolladores ahora tienen tres clases de las que deben crear instancias y realizar un seguimiento. Una solución común a este dilema es utilizar el patrón Fachada ([Figura 7.4](#)).

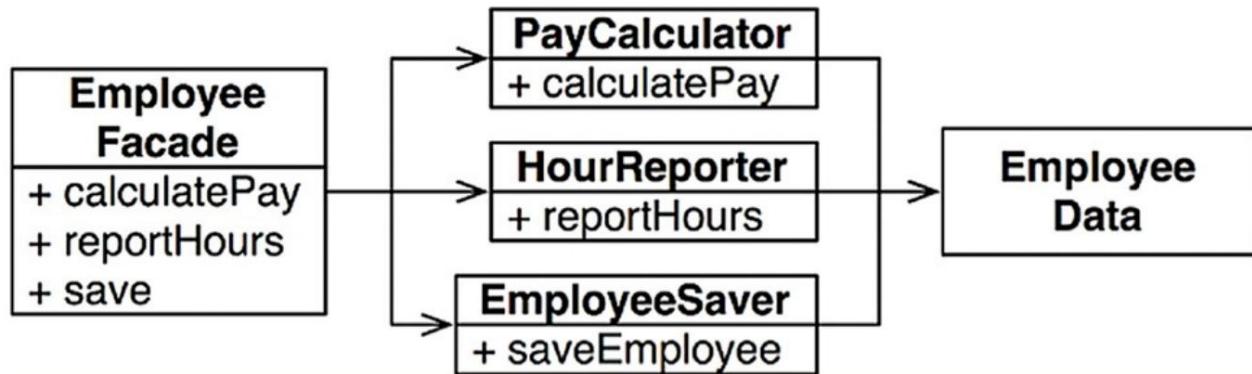


Figura 7.4 El patrón de fachada

EmployeeFacade contiene muy poco código. Se encarga de instanciar y delegar a las clases las funciones.

Algunos desarrolladores prefieren mantener las reglas comerciales más importantes más cerca de los datos. Esto se puede hacer manteniendo el método más importante en la clase Empleado original y luego usando esa clase como Fachada para las funciones menores ([Figura 7.5](#)).

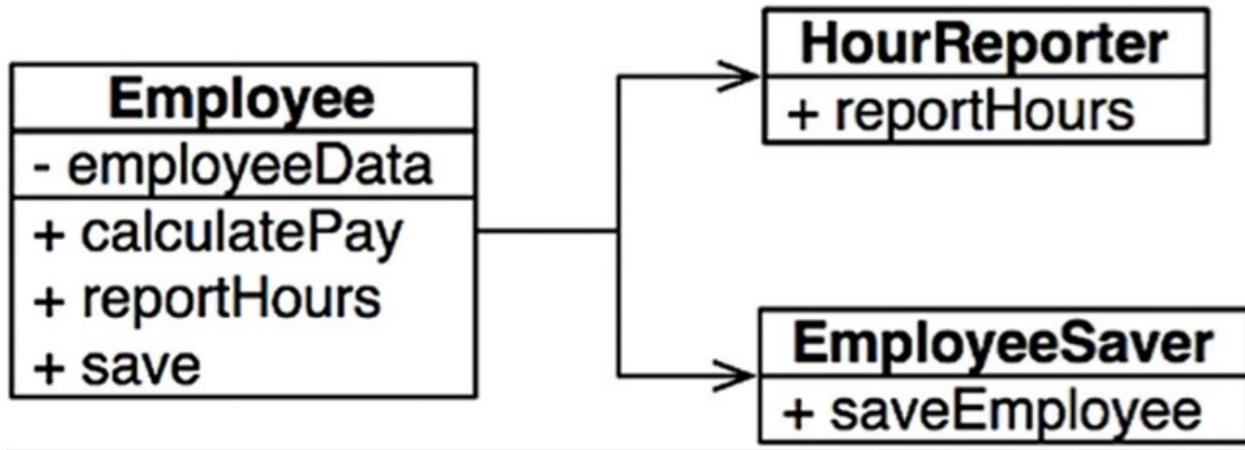


Figura 7.5 El método más importante se mantiene en la clase Empleado original y se usa como Fachada para las funciones menores.

Podría objetar estas soluciones basándose en que cada clase contendría solo una función. Este no es el caso. Es probable que la cantidad de funciones necesarias para calcular el salario, generar un informe o guardar los datos sea grande en cada caso. Cada una de esas clases tendría muchos métodos privados .

Cada una de las clases que contienen dicha familia de métodos es un alcance. Fuera de ese ámbito, nadie sabe que los miembros privados de la familia existen.

CONCLUSIÓN

El principio de responsabilidad única se refiere a funciones y clases, pero reaparece en una forma diferente en dos niveles más. A nivel de componentes, se convierte en el Principio de Cierre Común. A nivel arquitectónico, se convierte en el Eje de Cambio responsable de la creación de Límites Arquitectónicos. Estudiaremos todas estas ideas en los próximos capítulos.

8

OCP: EL PRINCIPIO ABIERTO-CERRADO



El principio abierto-cerrado (PCO) fue acuñado en 1988 por Bertrand Meyer. dice:

[1](#) Él

Un artefacto de software debe estar abierto a la extensión pero cerrado a la modificación.

En otras palabras, el comportamiento de un artefacto de software debería ser extensible, sin tener que modificar ese artefacto.

Esta, por supuesto, es la razón más fundamental por la que estudiamos la arquitectura de software. Claramente, si simples extensiones de los requisitos fuerzan cambios masivos en el software, entonces los arquitectos de ese sistema de software han cometido un fracaso espectacular.

La mayoría de los estudiantes de diseño de software reconocen el OCP como un principio que los guía en el diseño de clases y módulos. Pero el principio adquiere una importancia aún mayor cuando consideramos el nivel de los componentes arquitectónicos.

Un experimento mental aclarará esto.

UN EXPERIMENTO PENSAMIENTO

Imaginemos, por un momento, que tenemos un sistema que muestra un resumen financiero en una página web. Los datos de la página se pueden desplazar y los números negativos se muestran en rojo.

Ahora imagine que las partes interesadas piden que esta misma información se convierta en un informe para imprimir en una impresora en blanco y negro. El informe debe estar paginado correctamente, con encabezados de página, pies de página y etiquetas de columna adecuados. Los números negativos deben estar entre paréntesis.

Claramente, se debe escribir algún código nuevo. ¿Pero cuánto código antiguo habrá que cambiar?

Una buena arquitectura de software reduciría la cantidad de código modificado al mínimo. Lo ideal es cero.

¿Cómo? Separando adecuadamente las cosas que cambian por diferentes razones (el Principio de Responsabilidad Única) y luego organizando adecuadamente las dependencias entre esas cosas (el Principio de Inversión de Dependencia).

Al aplicar el SRP, podríamos obtener la vista del flujo de datos que se muestra en [la Figura 8.1](#). Algún procedimiento de análisis inspecciona los datos financieros y produce datos reportables, que luego son formateados apropiadamente por los dos procesos de reporte.



Figura 8.1 Aplicación del SRP

La idea esencial aquí es que generar el informe implica dos responsabilidades separadas: el cálculo de los datos reportados y la presentación de esos datos en un formato compatible con la web y para imprimir.

Una vez realizada esta separación, debemos organizar las dependencias del código fuente para garantizar que los cambios en una de esas responsabilidades no provoquen cambios en la otra. Además, la nueva organización debe garantizar que el comportamiento pueda ampliarse sin deshacer la modificación.

Logramos esto dividiendo los procesos en clases y separando esas clases en componentes, como se muestra en las líneas dobles en el diagrama de la [Figura 8.2](#). En esta figura, el componente en la parte superior izquierda es el Controlador. En la parte superior derecha tenemos el Interactor. En la parte inferior derecha está la Base de Datos. Finalmente, en la parte inferior izquierda, hay cuatro componentes que representan a los Presentadores y las Vistas.

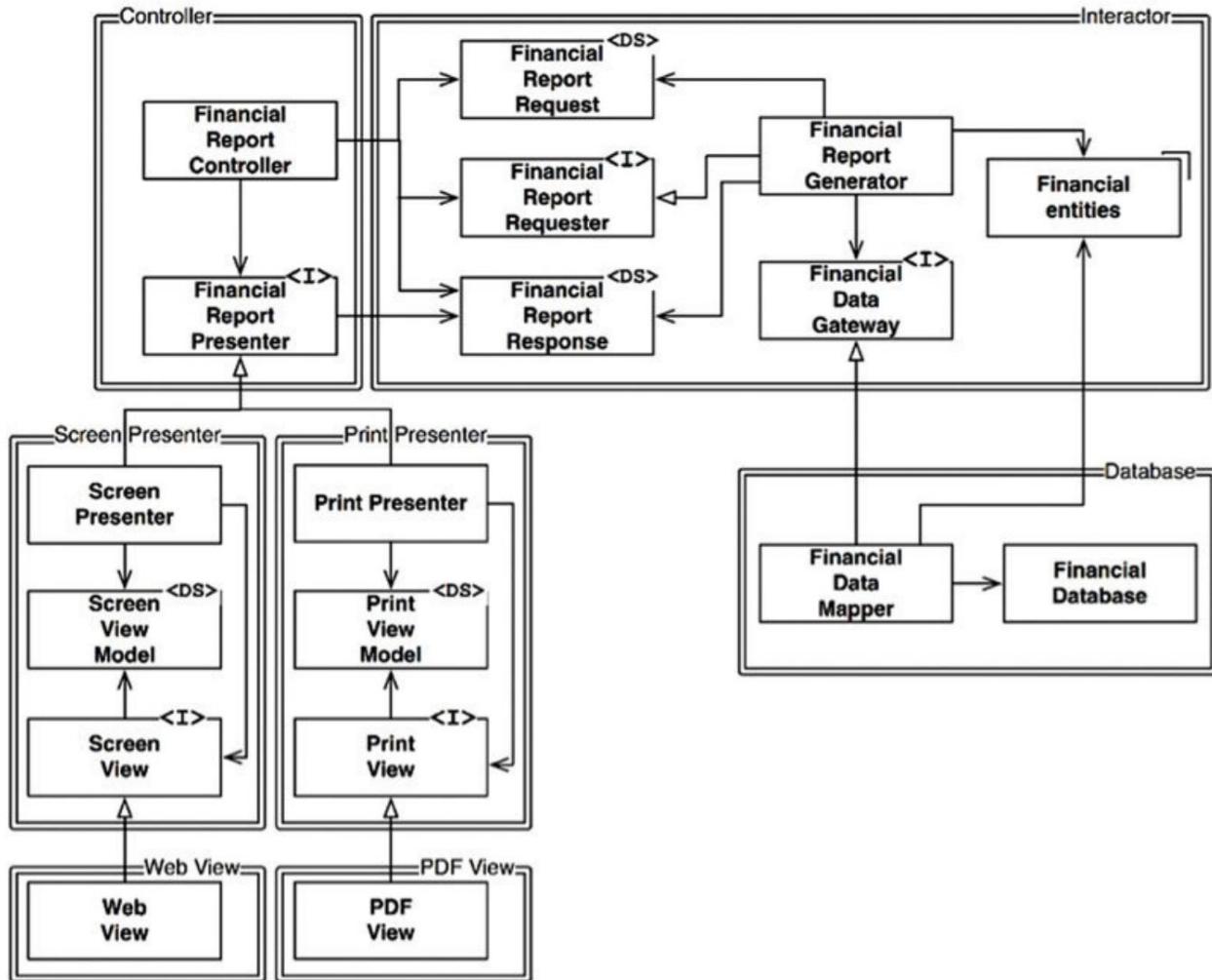


Figura 8.2 Partición de los procesos en clases y separación de las clases en componentes

Las clases marcadas con <I> son interfaces; los marcados con <DS> son estructuras de datos.

Las puntas de flecha abiertas utilizan relaciones. Las puntas de flecha cerradas son instrumentos o relaciones de herencia .

Lo primero que hay que notar es que todas las dependencias son dependencias del código fuente . Una flecha que apunta de la clase A a la clase B significa que el código fuente de la clase A menciona el nombre de la clase B, pero la clase B no menciona nada sobre la clase A. Por lo tanto, en la [Figura 8.2](#), **FinancialDataMapper** conoce **FinancialDataGateway** a través de una relación de implementos , pero **FinancialGateway** no sabe nada sobre **FinancialDataMapper**.

Lo siguiente que hay que notar es que cada línea doble se cruza en una sola dirección.

Esto significa que todas las relaciones de los componentes son unidireccionales, como se muestra en la

gráfico de componentes en [la Figura 8.3](#). Estas flechas apuntan hacia los componentes que queremos proteger del cambio.

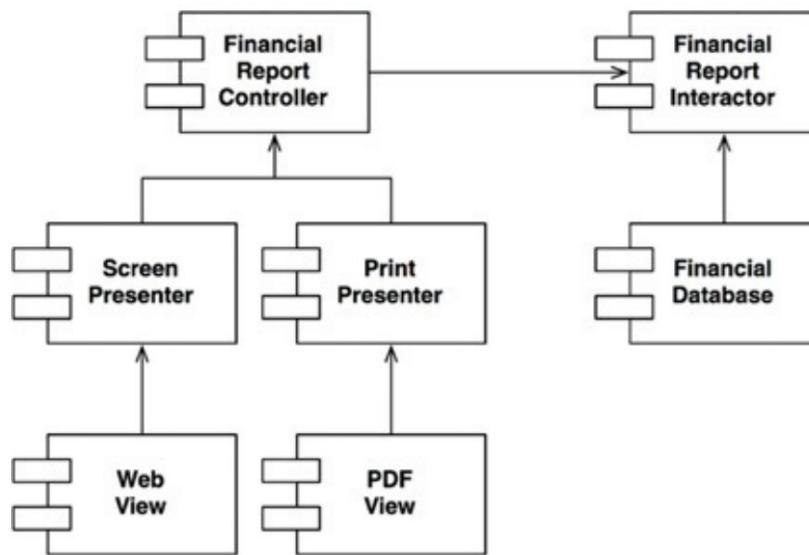


Figura 8.3 Las relaciones de los componentes son unidireccionales.

Permítanme decirlo nuevamente: si el componente A debe protegerse de los cambios en el componente B, entonces el componente B debería depender del componente A.

Queremos proteger al Controlador de cambios en los Presentadores. Queremos proteger a los Presentadores de cambios en las Vistas. Queremos proteger al Interactor de cambios en... bueno, cualquier cosa.

El Interactor está en la posición que mejor se ajusta al OCP. Los cambios en la Base de Datos, el Controlador, los Presentadores o las Vistas no tendrán ningún impacto en el Interactor.

¿Por qué debería el Interactor ocupar una posición tan privilegiada? Porque contiene las reglas de negocio. El Interactor contiene las políticas de más alto nivel de la aplicación. Todos los demás componentes se ocupan de preocupaciones periféricas. El Interactor se ocupa de la preocupación central.

Aunque el Controlador es periférico al Interactor, es central para los Presentadores y las Vistas. Y si bien los Presentadores pueden ser periféricos al Controlador, son fundamentales para las Vistas.

Observe cómo esto crea una jerarquía de protección basada en la noción de "nivel".

Los interactianos son el concepto de más alto nivel, por lo que son los más protegidos. Las vistas se encuentran entre los conceptos de nivel más bajo, por lo que son los menos protegidos. Los presentadores tienen un nivel superior a las Vistas, pero un nivel inferior al del Controlador o al Interactor.

Así funciona el OCP a nivel arquitectónico. Los arquitectos separan la funcionalidad según cómo, por qué y cuándo cambia, y luego organizan esa funcionalidad separada en una jerarquía de componentes. Los componentes de nivel superior en esa jerarquía están protegidos de los cambios realizados en los componentes de nivel inferior.

CONTROL DIRECCIONAL

Si retrocedió horrorizado ante el diseño de clases mostrado anteriormente, mire de nuevo. Gran parte de la complejidad de ese diagrama tenía como objetivo garantizar que las dependencias entre los componentes apuntaran en la dirección correcta.

Por ejemplo, la interfaz `FinancialDataGateway` entre `FinancialReportGenerator` y `FinancialDataMapper` existe para invertir la dependencia que de otro modo habría apuntado desde el componente `Interactor` al componente de base de datos . Lo mismo ocurre con la interfaz `FinancialReportPresenter` y las dos interfaces `View` .

OCULTACIÓN DE INFORMACIÓN

La interfaz `FinancialReportRequester` tiene un propósito diferente. Está ahí para proteger al `FinancialReportController` de saber demasiado sobre los aspectos internos del `Interactor`. Si esa interfaz no estuviera ahí, entonces el Controlador tendría dependencias transitivas de las Entidades Financieras.

Las dependencias transitivas son una violación del principio general de que las entidades de software no deben depender de cosas que no utilizan directamente. Nos encontraremos con ese principio nuevamente cuando hablemos del Principio de Segregación de Interfaz y el Principio de Reutilización Común.

Entonces, aunque nuestra primera prioridad es proteger el `Interactor` de cambios en el Controlador, también queremos proteger al Controlador de cambios en el `Interactor` ocultando las partes internas del `Interactor`.

CONCLUSIÓN

El OCP es una de las fuerzas impulsoras detrás de la arquitectura de los sistemas. El objetivo es hacer que el sistema sea fácil de ampliar sin incurrir en un gran impacto de cambio. Este objetivo se logra dividiendo el sistema en componentes y organizando esos componentes en una jerarquía de dependencia que protege los componentes de nivel superior de cambios en los componentes de nivel inferior.

1. Bertrand Meyer. Construcción de software orientado a objetos, Prentice Hall, 1988, p. 23.

9

LSP: LA SUSTITUCIÓN DE LISKOV PRINCIPIO



En 1988, Barbara Liskov escribió lo siguiente como una forma de definir los subtipos.

Lo que se busca aquí es algo como la siguiente propiedad de sustitución: si para cada objeto o1 de tipo S hay un objeto o2 de tipo T tal que para todos los programas P definidos en términos de T, el comportamiento **1** de P no cambia cuando o1 es sustituido por o2, entonces S es un subtipo de T.

Para entender esta idea, que se conoce como Principio de Sustitución de Liskov (LSP), veamos algunos ejemplos.

ORIENTAR EL USO DE LA HERENCIA

Imagine que tenemos una clase llamada `Licencia`, como se muestra en [la Figura 9.1](#). Esta clase tiene un método llamado `calcFee()`, al que llama la aplicación de facturación . Hay dos “subtipos” de `Licencia`: `PersonalLicense` y `BusinessLicense`. Utilizan diferentes algoritmos para calcular la tarifa de la licencia.

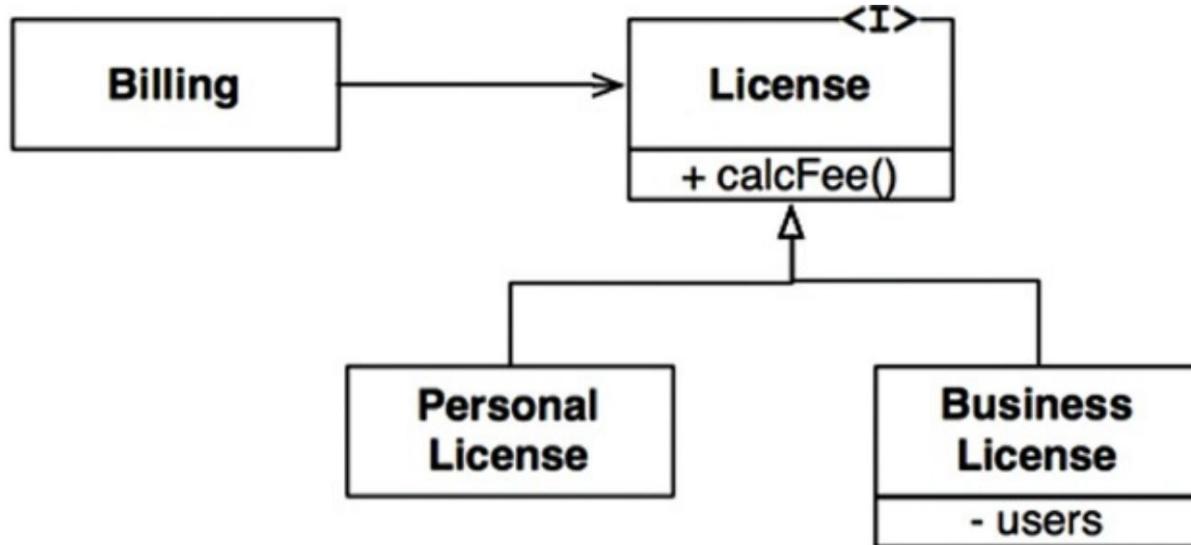


Figura 9.1 La licencia y sus derivados se ajustan a LSP

Este diseño se ajusta al LSP porque el comportamiento de la aplicación Facturación no depende, en modo alguno, de cuál de los dos subtipos utiliza. Ambos subtipos son sustituibles por el tipo de licencia .

EL PROBLEMA DEL CUADRADO/RECTÁNGULO

El ejemplo canónico de una violación del LSP es el famoso (o infame, según la perspectiva) problema del cuadrado/rectángulo ([Figura 9.2](#)).

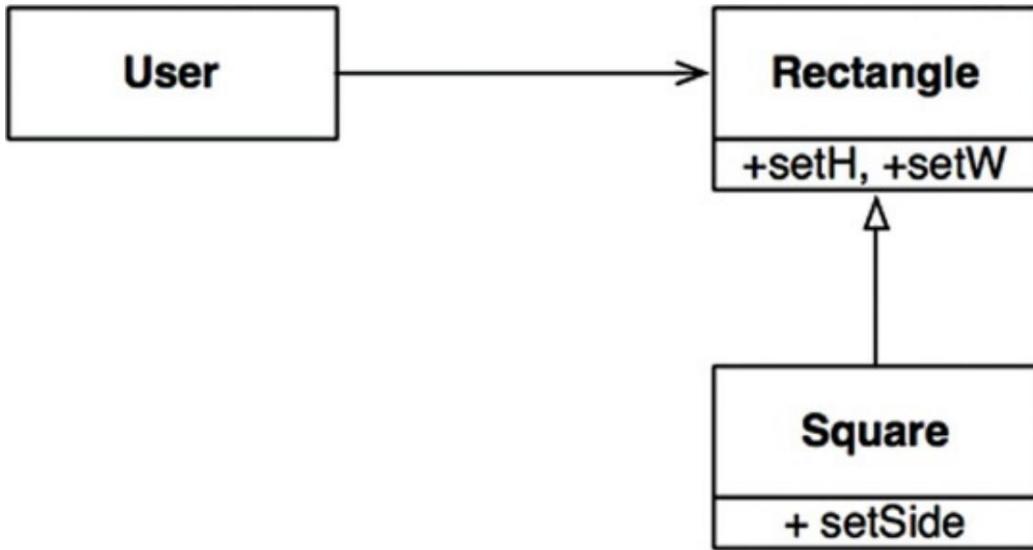


Figura 9.2 El infame problema del cuadrado/rectángulo

En este ejemplo, Cuadrado no es un subtipo adecuado de Rectángulo porque la altura y el ancho del Rectángulo son mutables de forma independiente; por el contrario, la altura y el ancho del cuadrado deben cambiar juntos. Dado que el Usuario cree que se está comunicando con un Rectángulo, fácilmente podría confundirse. El siguiente código muestra por qué:

[Haga clic aquí para ver la imagen del código](#)

```

Rectángulo r = ...
r.setW(5);
r.setH(2);
afirmar(r.area() == 10);
  
```

Si el ... Si el código produjera un cuadrado, entonces la afirmación fallaría.

La única forma de defenderse contra este tipo de violación de LSP es agregar mecanismos al Usuario (como una declaración if) que detecte si el Rectángulo es, de hecho, un Cuadrado. Dado que el comportamiento del Usuario depende de los tipos que utiliza, esos tipos no son sustituibles.

LSP Y ARQUITECTURA

En los primeros años de la revolución orientada a objetos, pensábamos en el LSP como una forma de guiar el uso de la herencia, como se muestra en las secciones anteriores. Sin embargo, a lo largo de los años, el LSP se ha transformado en un principio más amplio de diseño de software que pertenece a interfaces e implementaciones.

Las interfaces en cuestión pueden tener muchas formas. Podríamos tener una interfaz estilo Java, implementada por varias clases. O podríamos tener varias clases de Ruby que comparten las mismas firmas de métodos. O podríamos tener un conjunto de servicios que respondan todos a la misma interfaz REST.

En todas estas situaciones, y más, el LSP es aplicable porque hay usuarios que dependen de interfaces bien definidas y de la sustituibilidad de las implementaciones de esas interfaces.

La mejor manera de entender el LSP desde un punto de vista arquitectónico es observar qué sucede con la arquitectura de un sistema cuando se viola el principio.

EJEMPLO DE VIOLACIÓN DE LSP

Supongamos que estamos creando un agregador para muchos servicios de despacho de taxis. Los clientes utilizan nuestro sitio web para encontrar el taxi más adecuado, independientemente de la compañía de taxis. Una vez que el cliente toma una decisión, nuestro sistema despacha el taxi elegido utilizando un servicio tranquilo.

Ahora supongamos que el URI del servicio de despacho tranquilo es parte de la información contenida en la base de datos del conductor. Una vez que nuestro sistema ha elegido un controlador apropiado para el cliente, obtiene ese URI del registro del conductor y luego lo utiliza para enviar el controlador.

Supongamos que Driver Bob tiene un URI de envío similar a este:

[Haga clic aquí para ver la imagen del código](#)

purplecab.com/driver/Bob

Nuestro sistema agregaría la información de envío a este URI y la enviaría con un PUT, de la siguiente manera:

[Haga clic aquí para ver la imagen del código](#)

<purplecab.com/driver/Bob />
dirección de recogida/24 Maple St. /
hora de recogida/
153 /destino/ORD

Claramente, esto significa que todos los servicios de despacho, para todas las diferentes empresas, deben ajustarse a la misma interfaz REST. Deben tratar los campos pickupAddress, pickupTime y destino de manera idéntica.

Ahora supongamos que la compañía de taxis Acme contrató a algunos programadores que no leyeron las especificaciones con mucha atención. Abreviaron el campo de destino a simplemente destino. Acme es la compañía de taxis más grande de nuestra área, y la ex esposa del director ejecutivo de Acme es la nueva esposa de nuestro director ejecutivo, y... Bueno, ya se hace una idea. ¿Qué pasaría con la arquitectura de nuestro sistema?

Evidentemente, necesitaríamos añadir un caso especial. La solicitud de despacho para cualquier conductor de Acme tendría que elaborarse utilizando un conjunto de reglas diferente al de todos los demás conductores.

La forma más sencilla de lograr este objetivo sería agregar una declaración if al módulo que construyó el comando de envío:

[Haga clic aquí para ver la imagen del código](#)

```
si (driver.getDispatchUri().startsWith("acme.com"))...
```

Pero, por supuesto, ningún arquitecto que se precie permitiría que tal construcción existiera en el sistema. Poner la palabra "acme" en el código mismo crea una oportunidad para todo tipo de errores horribles y misteriosos, sin mencionar las violaciones de seguridad.

Por ejemplo, ¿qué pasaría si Acme tuviera aún más éxito y comprara la empresa Purple Taxi? ¿Qué pasaría si la empresa fusionada mantuviera marcas y sitios web separados, pero unificara todos los sistemas de las empresas originales? ¿Tendríamos que agregar otra declaración if para "púrpura"?

Nuestro arquitecto tendría que aislar el sistema de errores como este creando algún tipo de módulo de creación de comandos de envío impulsado por una base de datos de configuración codificada por el URI de envío. Los datos de configuración podrían verse así:

[Haga clic aquí para ver la imagen del código](#)

URI Formato de envío

<Acme.com> /direcciónderecogida/%s/horaderecogida/%s/dest/%s *.*
/direcciónderecogida/%s/horaderecogida/%s/destino/%s

Y por eso nuestro arquitecto ha tenido que añadir un mecanismo importante y complejo para hacer frente al hecho de que las interfaces de los servicios de descanso no son todas sustituibles.

CONCLUSIÓN

El LSP puede y debe ampliarse al nivel de la arquitectura. Una simple violación de la sustituibilidad puede causar que la arquitectura de un sistema se contamine con una cantidad significativa de mecanismos adicionales.

1. Barbara Liskov, "Data Abstraction and Hierarchy", SIGPLAN Notices 23, 5 (mayo de 1988).

10

ISP: LA SEGREGACIÓN DE LA INTERFAZ PRINCIPIO



El Principio de Segregación de Interfaz (ISP) deriva su nombre del diagrama que se muestra en [la Figura 10.1](#).

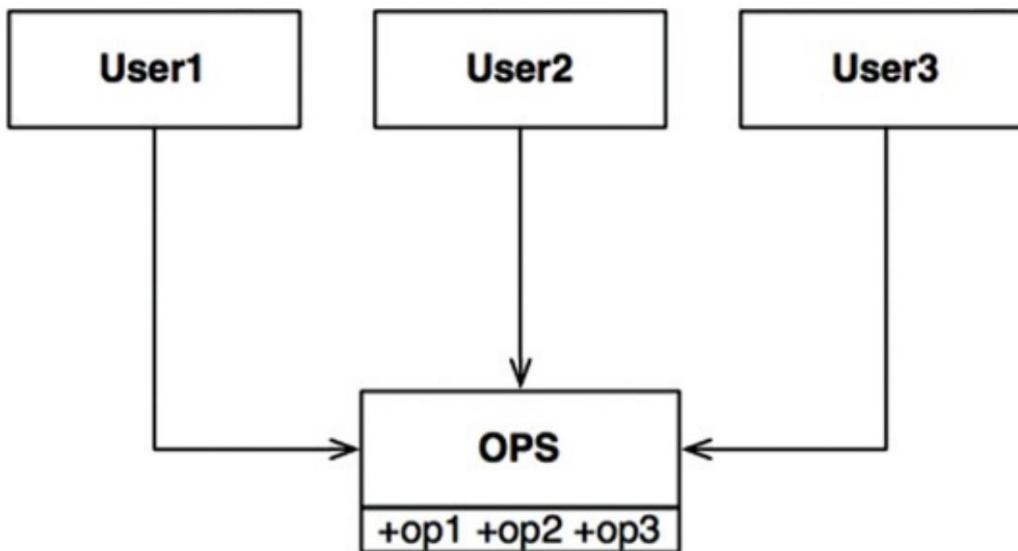


Figura 10.1 El principio de segregación de interfaz

En la situación ilustrada en [la Figura 10.1](#), hay varios usuarios que utilizan las operaciones de la clase OPS . Supongamos que el Usuario1 usa solo op1, el Usuario2 usa solo op2 y el Usuario3 usa solo op3.

Ahora imagine que OPS es una clase escrita en un lenguaje como Java. Claramente, en ese caso, el código fuente de Usuario1 dependerá inadvertidamente de op2 y op3, aunque no los llame. Esta dependencia significa que un cambio en el código fuente de op2 en OPS obligará al Usuario1 a ser recompilado y reimplementado, aunque nada de lo que le importaba haya cambiado realmente.

Este problema se puede resolver segregando las operaciones en interfaces como se muestra en [la Figura 10.2](#).

Nuevamente, si imaginamos que esto se implementa en un lenguaje de tipo estático como Java, entonces el código fuente de Usuario1 dependerá de U1Ops y op1, pero no dependerá de OPS. Por lo tanto, un cambio en OPS que al Usuario1 no le importe no hará que el Usuario1 sea recompilado y reimplementado.

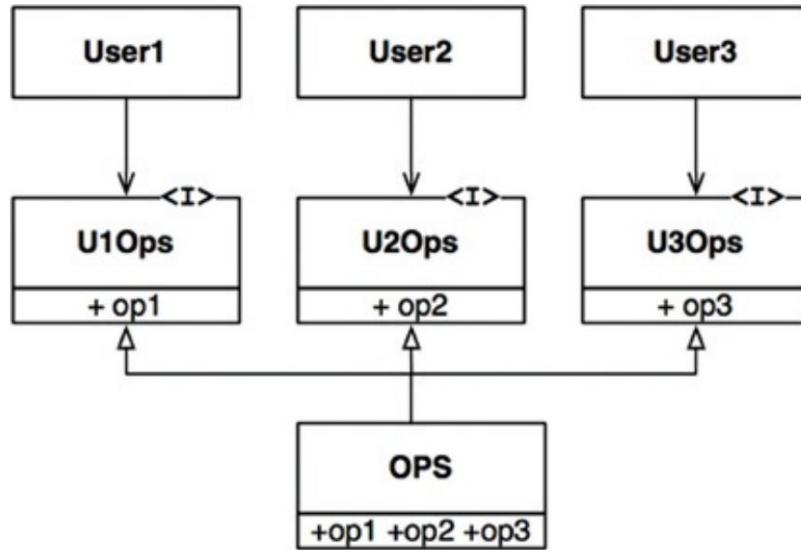


Figura 10.2 Operaciones segregadas

ISP E IDIOMA

Claramente, la descripción dada anteriormente depende críticamente del tipo de idioma. Los lenguajes de tipo estático como Java obligan a los programadores a crear declaraciones que los usuarios deben importar, usar o incluir de otra manera. Son estas declaraciones incluidas en el código fuente las que crean las dependencias del código fuente que fuerzan la recompilación y la reimplementación.

En lenguajes de tipo dinámico como Ruby y Python, este tipo de declaraciones no existen en el código fuente. En cambio, se infieren en tiempo de ejecución. Por lo tanto, no existen dependencias del código fuente que fuercen la recompilación y la redistribución. Esta es la razón principal por la que los lenguajes de tipado dinámico crean sistemas que son más flexibles y menos estrechamente acoplados que los lenguajes de tipado estático.

Este hecho podría llevarle a concluir que el ISP es una cuestión de idioma, más que de arquitectura.

ISP Y ARQUITECTURA

Si das un paso atrás y observas las motivaciones fundamentales del ISP, puedes ver una preocupación más profunda acechando allí. En general, es perjudicial depender de módulos que contienen más de lo necesario. Obviamente, esto es cierto para las dependencias del código fuente.

eso puede forzar una recompilación y redistribución innecesarias, pero también es cierto en un nivel arquitectónico mucho más elevado.

Consideremos, por ejemplo, un arquitecto que trabaja en un sistema S. Quiere incluir una determinada estructura, F, en el sistema. Ahora supongamos que los autores de F lo han vinculado a una base de datos particular, D. Entonces S depende de F, que depende de D ([Figura 10.3](#)).



Figura 10.3 Una arquitectura problemática

Ahora supongamos que D contiene características que F no utiliza y, por tanto, que a S no le importan. Los cambios en esas funciones dentro de D bien pueden forzar la redistribución de F y, por lo tanto, la redistribución de S. Peor aún, una falla de una de las funciones dentro de D puede causar fallas en F y S.

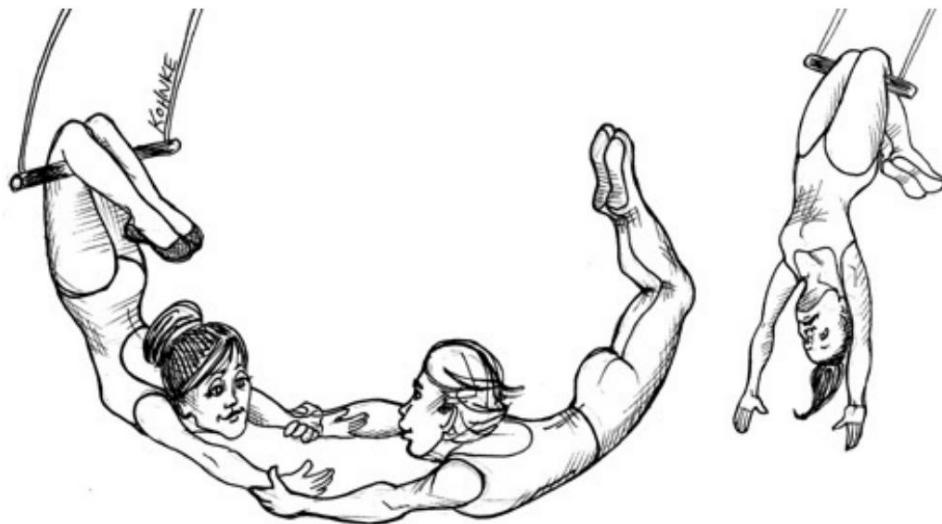
CONCLUSIÓN

La lección aquí es que depender de algo que lleva un equipaje que no necesitas puede causarte problemas que no esperabas.

Exploraremos esta idea con más detalle cuando analicemos el Principio de reutilización común en el [Capítulo 13](#), "Cohesión de componentes".

11

DIP: LA INVERSIÓN DE LA DEPENDENCIA PRINCIPIO



El Principio de Inversión de Dependencia (DIP) nos dice que los sistemas más flexibles son aquellos en los que las dependencias del código fuente se refieren sólo a abstracciones, no a concreciones.

En un lenguaje de tipo estático, como Java, esto significa que las declaraciones `use`, `import` e `include` deben referirse sólo a módulos fuente que contengan interfaces, clases abstractas o algún otro tipo de declaración abstracta. No se debe depender de nada concreto.

La misma regla se aplica a los lenguajes escritos dinámicamente, como Ruby y Python.

Las dependencias del código fuente no deben hacer referencia a módulos concretos. Sin embargo, en estos lenguajes es un poco más difícil definir qué es un módulo concreto. En particular, es cualquier módulo en el que se implementen las funciones que se llaman.

Claramente, tratar esta idea como una regla no es realista, porque los sistemas de software deben depender de muchas instalaciones concretas. Por ejemplo, la clase String en Java es concreta y no sería realista intentar forzarla a ser abstracta. La dependencia del código fuente del java.lang.String concreto no puede ni debe evitarse.

En comparación, la clase String es muy estable. Los cambios en esa clase son muy raros y están estrictamente controlados. Los programadores y arquitectos no tienen que preocuparse por cambios frecuentes y caprichosos en String.

Por estas razones, tendemos a ignorar el trasfondo estable del sistema operativo y las instalaciones de la plataforma cuando se trata de DIP. Toleramos esas dependencias concretas porque sabemos que podemos confiar en que no cambiarán.

Son los elementos concretos volátiles de nuestro sistema de los que queremos evitar depender. Esos son los módulos que estamos desarrollando activamente y que están experimentando cambios frecuentes.

ABSTRACCIONES ESTABLES

Cada cambio en una interfaz abstracta corresponde a un cambio en sus implementaciones concretas. Por el contrario, los cambios en implementaciones concretas no siempre, ni siquiera habitualmente, requieren cambios en las interfaces que implementan. Por tanto, las interfaces son menos volátiles que las implementaciones.

De hecho, los buenos diseñadores y arquitectos de software trabajan duro para reducir la volatilidad de las interfaces. Intentan encontrar formas de agregar funcionalidad a las implementaciones sin realizar cambios en las interfaces. Este es Diseño de software 101.

La implicación, entonces, es que las arquitecturas de software estables son aquellas que evitan depender de concreciones volátiles y que favorecen el uso de interfaces abstractas estables. Esta implicación se reduce a un conjunto de prácticas de codificación muy específicas:

- No haga referencia a clases de concreto volátil. En su lugar, consulte interfaces abstractas. Esta regla se aplica en todos los idiomas, ya sea de tipo estático o dinámico. Él

también impone severas restricciones a la creación de objetos y, en general, impone el uso de Abstract Factories.

- No deriven de clases concretas volátiles. Este es un corolario de la regla anterior, pero merece una mención especial. En los lenguajes de tipado estático, la herencia es la más fuerte y rígida de todas las relaciones del código fuente; en consecuencia, debe usarse con mucho cuidado. En los lenguajes tipados dinámicamente, la herencia es un problema menor, pero sigue siendo una dependencia, y la precaución es siempre la opción más inteligente.
- No anule funciones concretas. Las funciones concretas a menudo requieren dependencias del código fuente. Cuando anula esas funciones, no elimina esas dependencias; de hecho, las hereda . Para gestionar esas dependencias, debes hacer que la función sea abstracta y crear múltiples implementaciones.
- Nunca menciones el nombre de nada concreto y volátil.

En realidad, esto es sólo una reformulación del principio mismo.

FÁBRICAS

Para cumplir con estas reglas, la creación de objetos volátiles de hormigón requiere un manejo especial. Esta precaución está justificada porque, prácticamente en todos los lenguajes, la creación de un objeto requiere una dependencia del código fuente de la definición concreta de ese objeto.

En la mayoría de los lenguajes orientados a objetos, como Java, usaríamos una Fábrica Abstracta para gestionar esta dependencia no deseada.

El diagrama de [la Figura 11.1](#) muestra la estructura. La Aplicación utiliza `ConcreteImpl` a través de la interfaz del Servicio . Sin embargo, la Aplicación debe crear de alguna manera instancias de `ConcreteImpl`. Para lograr esto sin crear una dependencia del código fuente en `ConcreteImpl`, la aplicación llama al método `makeSvc` de la interfaz `ServiceFactory` . Este método lo implementa la clase `ServiceFactoryImpl` , que deriva de `ServiceFactory`. Esa implementación crea una instancia de `ConcreteImpl` y lo devuelve como un Servicio.

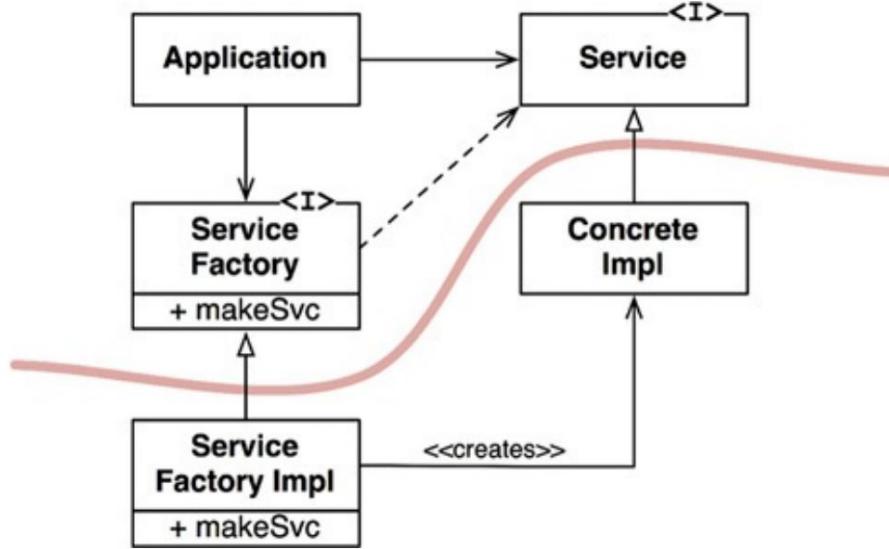


Figura 11.1 Uso del patrón Abstract Factory para gestionar la dependencia

La línea curva en [la Figura 11.1](#) es un límite arquitectónico. Separa lo abstracto de lo concreto. Todas las dependencias del código fuente cruzan esa línea curva que apunta en la misma dirección, hacia el lado abstracto.

La línea curva divide el sistema en dos componentes: uno abstracto y otro concreto. El componente abstracto contiene todas las reglas comerciales de alto nivel de la aplicación. El componente concreto contiene todos los detalles de implementación que manipulan esas reglas comerciales.

Tenga en cuenta que el flujo de control cruza la línea curva en la dirección opuesta a las dependencias del código fuente. Las dependencias del código fuente se invierten contra el flujo de control, razón por la cual nos referimos a este principio como Inversión de Dependencia.

COMPONENTES DE HORMIGÓN

El componente concreto de [la Figura 11.1](#) contiene una única dependencia, por lo que viola el DIP. Esto es típico. Las infracciones DIP no se pueden eliminar por completo, pero se pueden agrupar en una pequeña cantidad de componentes concretos y mantener separados del resto del sistema.

La mayoría de los sistemas contendrán al menos uno de esos componentes concretos, a menudo llamado principal porque contiene el componente principal de la función. En el caso ilustrado en [la Figura 11.1](#),

la función principal crearía una instancia de ServiceFactoryImpl y colocaría esa instancia en una variable global de tipo ServiceFactory. Luego, la Aplicación accedería a la fábrica a través de esa variable global.

CONCLUSIÓN

A medida que avancemos en este libro y cubramos principios arquitectónicos de nivel superior, el DIP aparecerá una y otra vez. Será el principio organizativo más visible en nuestros diagramas de arquitectura. La línea curva de [la Figura 11.1](#) se convertirá en los límites arquitectónicos en capítulos posteriores. La forma en que las dependencias cruzan esa línea curva en una dirección, y hacia entidades más abstractas, se convertirá en una nueva regla que llamaremos Regla de Dependencia.

¹ En otras palabras, la función que invoca el sistema operativo cuando se inicia la aplicación por primera vez.

arriba.

IV

PRINCIPIOS DE LOS COMPONENTES

Si los principios SÓLIDOS nos dicen cómo organizar los ladrillos en paredes y habitaciones, entonces los principios de los componentes nos dicen cómo organizar las habitaciones en edificios. Los grandes sistemas de software, al igual que los grandes edificios, se construyen a partir de componentes más pequeños.

En [la Parte IV](#), discutiremos qué son los componentes de software, qué elementos deberían componerlos y cómo deberían componerse en sistemas.

12 COMPONENTES



Los componentes son las unidades de despliegue. Son las entidades más pequeñas que se pueden implementar como parte de un sistema. En Java, son archivos jar. En Ruby, son archivos de gemas. En .Net, son DLL. En lenguajes compilados, son agregaciones de archivos binarios. En lenguajes interpretados, son agregaciones de archivos fuente. En todos los idiomas, son el gránulo de implementación.

Los componentes se pueden vincular entre sí en un único ejecutable. O se pueden agregar en un solo archivo, como un archivo .war . O se pueden implementar de forma independiente como complementos separados cargados dinámicamente, como archivos.jar , .dll o .exe . Independientemente de cómo se implementen finalmente, los componentes bien diseñados siempre conservan la capacidad de implementarse de forma independiente y, por lo tanto, desarrollarse de forma independiente.

UNA BREVE HISTORIA DE LOS COMPONENTES

En los primeros años del desarrollo de software, los programadores controlaban la ubicación de la memoria y el diseño de sus programas. Una de las primeras líneas de código de un programa sería la declaración de origen , que declaraba la dirección en la que se iba a cargar el programa.

Considere el siguiente programa sencillo PDP-8. Consiste en una subrutina llamada GETSTR que ingresa una cadena desde el teclado y la guarda en un búfer. También cuenta con un pequeño programa de pruebas unitarias para ejercitarse GETSTR.

[Haga clic aquí para ver la imagen del código](#)

```
*200
TLS
COMIENZO, CLA
TAD BUFR
JMS GETSTR
CLA
TAD BUFR
JMS PUTSTR
INICIO DEL JMP
BUFR, 3000

OBTENERSTR, 0
DCA PTR
NXTCH, KSF
JMP-1
KRB
DCA I PTR
TAD I PTR
Y K177
ISZ PTR
TAD MCR
SZA
JMP NXTCH

K177, 177
RCM, -15
```

Tenga en cuenta el comando *200 al inicio de este programa. Le dice al compilador que genere código que se cargará en la dirección 2008 .

Este tipo de programación es un concepto extraño para la mayoría de los programadores actuales. Rara vez tienen que pensar en dónde está cargado un programa en la memoria del

computadora. Pero al principio, esta era una de las primeras decisiones que debía tomar un programador. En aquella época los programas no eran reubicables.

¿Cómo accedías a una función de biblioteca en aquellos tiempos? El código anterior ilustra el enfoque utilizado. Los programadores incluyeron el código fuente de las funciones de la biblioteca con el código de su aplicación y los compilaron todos como un solo [programa](#).

- Las bibliotecas se mantuvieron en código fuente, no en binario.

El problema con este enfoque era que, durante esta época, los dispositivos eran lentos y la memoria cara y, por tanto, limitada. Los compiladores necesitaban realizar varias pasadas sobre el código fuente, pero la memoria era demasiado limitada para mantener todo el código fuente residente. En consecuencia, el compilador tuvo que leer el código fuente varias veces utilizando los dispositivos lentos.

Esto llevó mucho tiempo y cuanto más grande fuera su biblioteca de funciones, más tardaría el compilador. Compilar un programa grande podría llevar horas.

Para acortar los tiempos de compilación, los programadores separaron el código fuente de la biblioteca de funciones de las aplicaciones. Compilaron la biblioteca de funciones por separado y cargaron el binario en una dirección conocida, digamos 20008 . Crearon una tabla de símbolos para la biblioteca de funciones y la compilaron con el código de su aplicación. Cuando querían ejecutar una aplicación, cargaban la biblioteca de funciones binarias, que se muestra en [la Figura 12.1](#).

[2](#) y luego cargar la aplicación. La memoria se parecía al diseño.

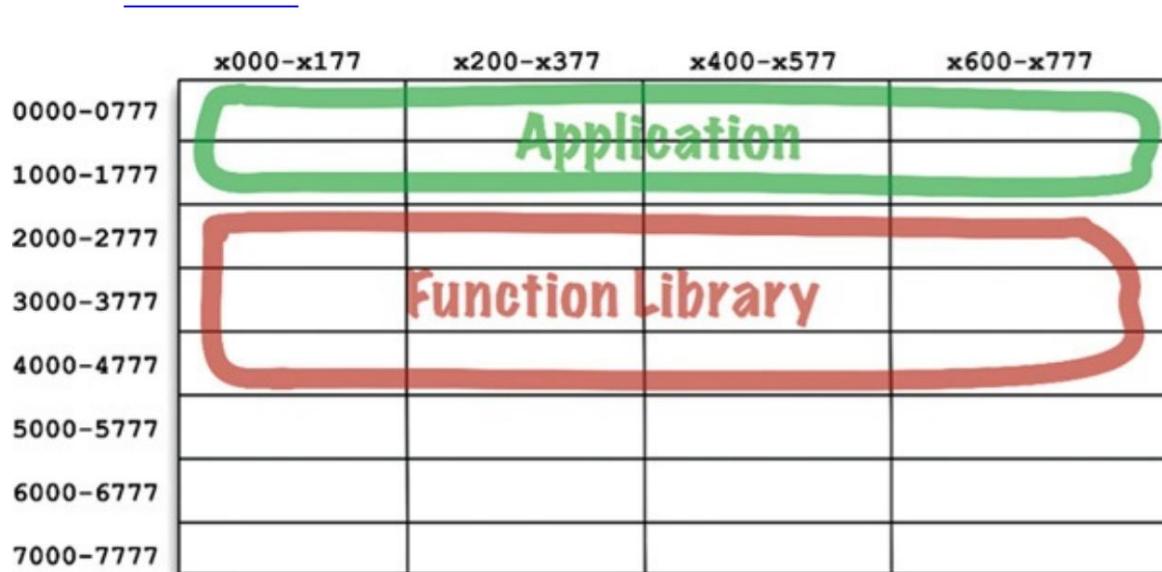


Figura 12.1 Diseño de la memoria inicial

Esto funcionó bien siempre que la aplicación pudiera caber entre las direcciones 00008 y 17778 .

Pero pronto las solicitudes crecieron siendo más grandes que el espacio asignado para ellas. En ese punto, los programadores tuvieron que dividir sus aplicaciones en dos segmentos de direcciones, saltando por la biblioteca de funciones ([Figura 12.2](#)).

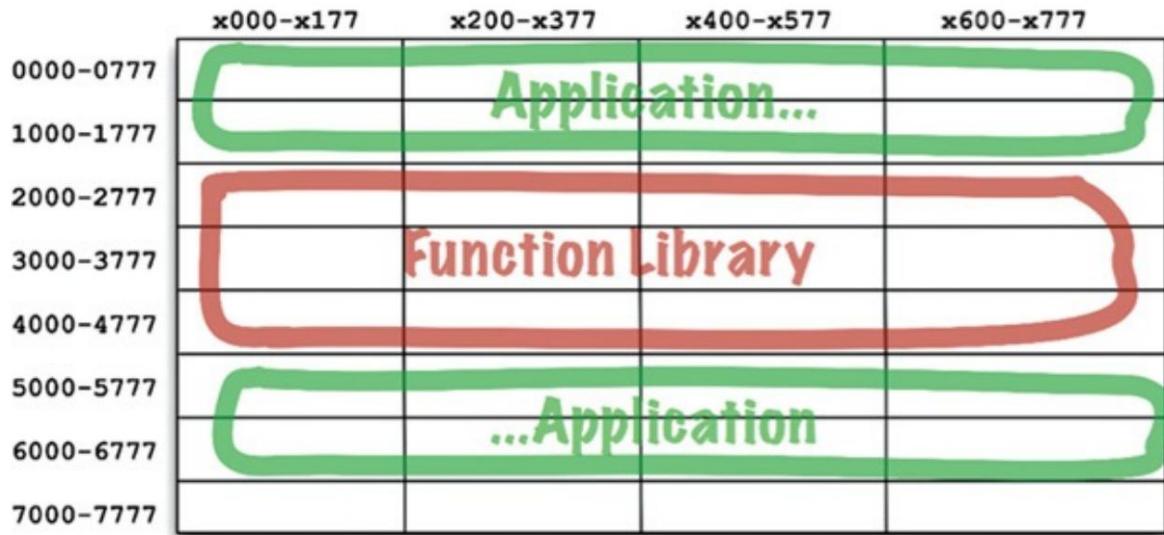


Figura 12.2 División de la aplicación en dos segmentos de dirección

Evidentemente, ésta no era una situación sostenible. A medida que los programadores agregaron más funciones a la biblioteca de funciones, esta excedió sus límites y tuvieron que asignarle más espacio (en este ejemplo, cerca de 70008). Esta fragmentación de programas y bibliotecas necesariamente continuó a medida que crecía la memoria de la computadora.

Era evidente que había que hacer algo.

REUBICABILIDAD

La solución fueron los binarios reubicables. La idea detrás de ellos era muy simple. El compilador se cambió para generar código binario que un cargador inteligente podría reubicar en la memoria. Se le indicará al cargador dónde cargar el código reubicable. El código reubicable estaba equipado con banderas que le indicaban al cargador qué partes de los datos cargados debían modificarse para cargarse en la dirección seleccionada. Por lo general, esto solo significaba agregar la dirección inicial a cualquier dirección de referencia de memoria en el binario.

Ahora el programador podría indicarle al cargador dónde cargar la biblioteca de funciones y

dónde cargar la aplicación. De hecho, el cargador aceptaría varias entradas binarias y simplemente las cargaría en la memoria una tras otra, reubicándolas a medida que las cargaba. Esto permitió a los programadores cargar solo aquellas funciones que necesitaban.

El compilador también se cambió para emitir los nombres de las funciones como metadatos en el binario reubicable. Si un programa llamara a una función de biblioteca, el compilador emitiría ese nombre como referencia externa. Si un programa definiera una función de biblioteca, el compilador emitiría ese nombre como una definición externa. Luego, el cargador podría vincular las referencias externas a las definiciones externas una vez que haya determinado dónde había cargado esas definiciones.

Y nació el cargador de enlaces.

ENLACEADORES

El cargador de enlaces permitió a los programadores dividir sus programas en segmentos compilables y cargables por separado. Esto funcionó bien cuando se vinculaban programas relativamente pequeños con bibliotecas relativamente pequeñas. Sin embargo, a finales de los años 1960 y principios de los 1970, los programadores se volvieron más ambiciosos y sus programas se hicieron mucho más grandes.

Al final, los cargadores de enlace fueron demasiado lentos para tolerarlos. Las bibliotecas de funciones se almacenaban en dispositivos lentos, como una cinta magnética. Incluso los discos en aquel entonces eran bastante lentos. Usando estos dispositivos relativamente lentos, los cargadores de enlace tuvieron que leer docenas, si no cientos, de bibliotecas binarias para resolver las referencias externas. A medida que los programas crecían cada vez más y se acumulaban más funciones de biblioteca en las bibliotecas, un cargador de enlaces podía tardar más de una hora simplemente en cargar el programa.

Finalmente, la carga y el enlace se separaron en dos fases.

Los programadores tomaron la parte lenta (la parte que realizaba el enlace) y la colocaron en una aplicación separada llamada enlazador . La salida del enlazador era un reubicable enlazado que un cargador reubicable podía cargar muy rápidamente. Esto permitió a los programadores preparar un ejecutable usando el enlazador lento, pero luego podían cargarlo rápidamente, en cualquier momento.

Luego vinieron los años 1980. Los programadores trabajaban en C o algún otro lenguaje de alto nivel. A medida que crecieron sus ambiciones, también lo hicieron sus programas. Programas que

Los cientos de miles de líneas de código numeradas no eran inusuales.

Los módulos fuente se compilaron a partir de archivos .c en archivos .o y luego se introdujeron en el vinculador para crear archivos ejecutables que pudieran cargarse rápidamente. Compilar cada módulo individual fue relativamente rápido, pero compilar todos los módulos tomó un poco de tiempo. El enlazador tardaría aún más tiempo. El tiempo de respuesta había vuelto a aumentar hasta una hora o más en muchos casos.

Parecía como si los programadores estuvieran condenados a perseguirse la cola sin cesar.

A lo largo de las décadas de 1960, 1970 y 1980, todos los cambios realizados para acelerar el flujo de trabajo se vieron frustrados por las ambiciones de los programadores y el tamaño de los programas que escribían. Parecía que no podían escapar de los plazos de entrega de una hora. El tiempo de carga siguió siendo rápido, pero los tiempos de compilación y enlace fueron el cuello de botella.

Por supuesto, estábamos experimentando la ley de Murphy del tamaño del programa:

Los programas crecerán para ocupar todo el tiempo disponible de compilación y enlace.

Murphy no fue el único contendiente en la ciudad. Llegó Moore, a finales de los 80, y los dos se [3](#) y en el Pero enfrentaron. Moore ganó esa batalla. Los discos comenzaron a reducirse y se volvieron significativamente más rápidos. La memoria de la computadora comenzó a volverse tan ridículamente barata que gran parte de los datos del disco podían almacenarse en la memoria RAM. Las velocidades de reloj de las computadoras aumentaron de 1 MHz a 100 MHz.

A mediados de la década de 1990, el tiempo dedicado a establecer vínculos había comenzado a reducirse más rápido de lo que nuestras ambiciones podían hacer crecer los programas. En muchos casos, el tiempo de enlace se redujo a cuestión de segundos. Para trabajos pequeños, la idea de un cargador de enlace volvió a ser viable.

Esta fue la era de Active-X, las bibliotecas compartidas y los inicios de los archivos .jar . Las computadoras y los dispositivos se habían vuelto tan rápidos que, una vez más, podíamos realizar la vinculación en el momento de la carga. Podríamos vincular varios archivos .jar , o varias bibliotecas compartidas en cuestión de segundos, y ejecutar el programa resultante. Y así nació la arquitectura de complementos de componentes.

Hoy en día enviamos habitualmente archivos .jar o DLL o bibliotecas compartidas como complementos para aplicaciones existentes. Si desea crear un mod para Minecraft, por ejemplo, simplemente incluya sus archivos .jar personalizados en una carpeta determinada. Si desea conectar Resharper a Visual Studio, simplemente incluya las DLL apropiadas.

CONCLUSIÓN

Estos archivos vinculados dinámicamente, que se pueden conectar entre sí en tiempo de ejecución, son los componentes de software de nuestras arquitecturas. Han sido necesarios 50 años, pero hemos llegado a un lugar donde la arquitectura de complementos de componentes puede ser un valor predeterminado casual en lugar del esfuerzo hercúleo que alguna vez fue.

1. Mi primer empleador guardaba varias docenas de paquetes del código fuente de la biblioteca de subrutinas en un estante. Cuando escribías un nuevo programa, simplemente tomabas uno de esos mazos y lo colocabas en el extremo del mazo.
2. En realidad, la mayoría de esas máquinas antiguas usaban memoria central, que no se borraba cuando encendías el computadora apagada. A menudo dejábamos la biblioteca de funciones cargada durante días seguidos.
3. Ley de Moore: la velocidad, la memoria y la densidad de la computadora se duplican cada 18 meses. Esta ley se sostuvo desde el 1950 a 2000, pero luego, al menos en lo que respecta a las velocidades de reloj, se detuvo en seco.

13

COHESIÓN DE COMPONENTES



¿Qué clases pertenecen a qué componentes? Esta es una decisión importante y requiere la guía de buenos principios de ingeniería de software. Desafortunadamente, a lo largo de los años, esta decisión se ha tomado de manera ad hoc y basada casi exclusivamente en el contexto.

En este capítulo discutiremos los tres principios de cohesión de componentes:

- REP: El principio de equivalencia de reutilización/ liberación
- CCP: El principio de cierre común
- CRP: El principio de reutilización común

LA EQUIVALENCIA DE REUTILIZACIÓN/LIBERACIÓN PRINCIPIO

El gránulo de reutilización es el gránulo de liberación.

La última década ha visto el surgimiento de una colección de herramientas de gestión de módulos, como Maven, Leiningen y RVM. Estas herramientas han ganado importancia porque, durante ese tiempo, se ha creado una gran cantidad de componentes y bibliotecas de componentes reutilizables. Ahora vivimos en la era de la reutilización del software, un cumplimiento de una de las promesas más antiguas del modelo orientado a objetos.

El Principio de Equivalencia de Reutilización/Liberación (REP) es un principio que parece obvio, al menos en retrospectiva. Las personas que quieran reutilizar componentes de software no pueden hacerlo, ni lo harán, a menos que se realice un seguimiento de esos componentes a través de un proceso de lanzamiento y se les proporcionen números de lanzamiento.

Esto no se debe simplemente a que, sin los números de versión, no habría forma de garantizar que todos los componentes reutilizados sean compatibles entre sí. Más bien, también refleja el hecho de que los desarrolladores de software necesitan saber cuándo llegarán nuevos lanzamientos y qué cambios traerán esos nuevos lanzamientos.

No es raro que los desarrolladores reciban alertas sobre una nueva versión y decidan, basándose en los cambios realizados en esa versión, continuar usando la versión anterior. Por lo tanto, el proceso de lanzamiento debe producir las notificaciones y la documentación de lanzamiento apropiadas para que los usuarios puedan tomar decisiones informadas sobre cuándo y si integrar el nuevo lanzamiento.

Desde el punto de vista del diseño y la arquitectura del software, este principio significa que las clases y módulos que se forman en un componente deben pertenecer a un grupo cohesivo. El componente no puede consistir simplemente en una mezcolanza aleatoria de clases y módulos; en cambio, debe haber algún tema o propósito general que todos esos módulos compartan.

Por supuesto, esto debería ser obvio. Sin embargo, hay otra forma de abordar esta cuestión que quizás no sea tan obvia. Las clases y módulos que están agrupados en un componente deben poder publicarse juntos. El hecho de que comparten el mismo número de versión y el mismo seguimiento de lanzamiento, y estén incluidos en la misma documentación de lanzamiento, debería tener sentido tanto para el autor como para el autor.
usuarios.

Este es un consejo débil: decir que algo debería “tener sentido” es sólo una forma de agitar las manos en el aire y tratar de parecer autoritario. El consejo es débil porque es difícil explicar con precisión el pegamento que mantiene unidos las clases y los módulos en un solo componente. Por débil que sea el consejo, el principio en sí es importante, porque las violaciones son fáciles de detectar: no “tienen sentido”. Si viola el REP, sus usuarios lo sabrán y no quedarán impresionados con sus habilidades arquitectónicas.

La debilidad de este principio queda más que compensada por la fuerza de los dos principios siguientes. De hecho, el PCC y el CRP definen firmemente este principio, pero en un sentido negativo.

EL PRINCIPIO DE CIERRE COMÚN

Reunir en componentes aquellas clases que cambian por los mismos motivos y en los mismos momentos.
Separar en diferentes componentes aquellas clases que cambian en diferentes momentos y para diferentes razones.

Este es el Principio de Responsabilidad Única reformulado para los componentes. Así como el SRP dice que una clase no debe contener múltiples razones para cambiar, el Principio de Cierre Común (CCP) dice que un componente no debe tener múltiples razones para cambiar.

Para la mayoría de las aplicaciones, la mantenibilidad es más importante que la reutilización. Si el código de una aplicación debe cambiar, preferiría que todos los cambios se produjeran en un componente, en lugar de distribuirse entre muchos componentes.
¹
Si los cambios se limitan a un solo componente, entonces necesitamos volver a implementar solo el componente modificado. No es necesario revalidar ni implementar otros componentes que no dependen del componente modificado.

El PCC nos insta a reunir en un solo lugar todas las clases que probablemente cambiarán por las mismas razones. Si dos clases están tan estrechamente ligadas, ya sea física o conceptualmente, que siempre cambian juntas, entonces pertenecen al mismo componente. Esto minimiza la carga de trabajo relacionada con la liberación, revalidación y reimplementación del software.

Este principio está estrechamente asociado con el principio abierto y cerrado (OCP). De hecho, lo que aborda el PCC es el “cierre” en el sentido de la palabra que da el PCC. La OCP establece que las clases deben estar cerradas para modificaciones pero abiertas para extensión.

Dado que no es posible lograr un cierre del 100%, el cierre debe ser estratégico. Diseñamos nuestras clases de manera que estén cerradas a los tipos de cambios más comunes que esperamos o hemos experimentado.

El PCC amplifica esta lección reuniendo en el mismo componente aquellas clases que están cerradas a los mismos tipos de cambios. Por lo tanto, cuando se produce un cambio en los requisitos, es muy probable que ese cambio se limite a un número mínimo de componentes.

SIMILARIDAD CON SRP

Como se indicó anteriormente, el PCC es el componente del SRP. El SRP nos dice que separemos los métodos en diferentes clases, si cambian por diferentes motivos. El PCC nos dice que separemos las clases en diferentes componentes, si cambian por diferentes motivos. Ambos principios se pueden resumir en el siguiente fragmento:

Reúne aquellas cosas que cambian al mismo tiempo y por los mismos motivos. Separa aquellas cosas que cambian en diferentes momentos o por diferentes motivos.

EL PRINCIPIO DE REUTILIZACIÓN COMÚN

No obligue a los usuarios de un componente a depender de cosas que no necesitan.

El Principio de Reutilización Común (CRP) es otro principio que nos ayuda a decidir qué clases y módulos deben colocarse en un componente. Afirma que las clases y módulos que tienden a reutilizarse juntos pertenecen al mismo componente.

Las clases rara vez se reutilizan de forma aislada. Lo más habitual es que las clases reutilizables colaboren con otras clases que forman parte de la abstracción reutilizable. El PCI establece que estas clases pertenecen juntas en el mismo componente. En un componente de este tipo, esperaríamos ver clases que tengan muchas dependencias entre sí.

Un ejemplo sencillo podría ser una clase contenedora y sus iteradores asociados. Estas clases se reutilizan juntas porque están estrechamente vinculadas entre sí. Por tanto deberían estar en el mismo componente.

Pero el CRP nos dice más que sólo juntar en un

componente: También nos dice qué clases no mantener juntas en un componente. Cuando un componente usa otro, se crea una dependencia entre los componentes. Quizás el componente que usa usa sólo una clase dentro del componente usado , pero eso aún no debilita la dependencia. El componente utilizado todavía depende del componente utilizado .

Debido a esa dependencia, cada vez que se cambia el componente utilizado , es probable que el componente utilizado necesite los cambios correspondientes. Incluso si no es necesario realizar cambios en el componente que lo utiliza , es probable que aún sea necesario volver a compilarlo, revalidarlo y volverlo a implementar. Esto es cierto incluso si al componente que lo utiliza no le importa el cambio realizado en el componente usado .

Por lo tanto, cuando dependemos de un componente, queremos asegurarnos de depender de cada clase de ese componente. Dicho de otra manera, queremos asegurarnos de que las clases que ponemos en un componente sean inseparables: que sea imposible depender de unas y no de otras. De lo contrario, estaremos redistribuyendo más componentes de los necesarios y desperdiando un esfuerzo significativo.

Por lo tanto, el CRP nos dice más sobre qué clases no deberían estar juntas que sobre qué clases deberían estar juntas. El CRP dice que las clases que no están estrechamente vinculadas entre sí no deberían estar en el mismo componente.

RELACIÓN CON EL ISP

El CRP es la versión genérica del ISP. El ISP nos aconseja no depender de clases que tengan métodos que no utilizamos. El CRP nos aconseja no depender de componentes que tengan clases que no utilizamos.

Todos estos consejos se pueden reducir a un solo fragmento:

No dependas de cosas que no necesitas.

EL DIAGRAMA DE TENSIONES PARA COHESIÓN DE COMPONENTES

Quizás ya te hayas dado cuenta de que los tres principios de cohesión tienden a luchar entre sí. El REP y el CCP son principios inclusivos : ambos tienden a agrandar los componentes. El CRP es un principio exclusivo que impulsa los componentes a ser

menor. Es la tensión entre estos principios lo que los buenos arquitectos buscan resolver.

[La figura 13.1](#) es un diagrama de tensión² que muestra cómo interactúan entre sí los tres principios de cohesión. Los bordes del diagrama describen el costo de abandonar el principio en el vértice opuesto.

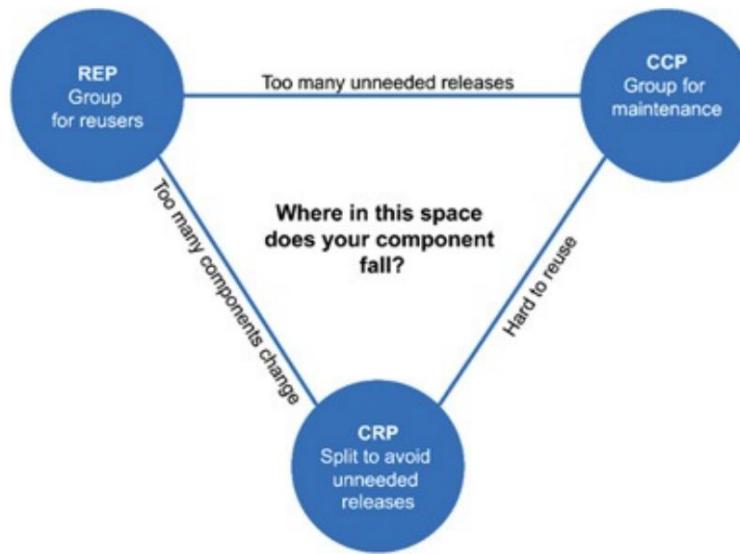


Figura 13.1 Diagrama de tensión de los principios de cohesión

Un arquitecto que se centra únicamente en REP y CRP encontrará que demasiados componentes se ven afectados cuando se realizan cambios simples. Por el contrario, un arquitecto que se centra demasiado en el CCP y el REP provocará que se generen demasiadas publicaciones innecesarias.

Un buen arquitecto encuentra una posición en ese triángulo de tensión que responde a las preocupaciones actuales del equipo de desarrollo, pero también es consciente de que esas preocupaciones cambiarán con el tiempo. Por ejemplo, al principio del desarrollo de un proyecto, el PCC es mucho más importante que el REP, porque la capacidad de desarrollo es más importante que la reutilización.

Generalmente, los proyectos tienden a comenzar en el lado derecho del triángulo, donde el único sacrificio es la reutilización. A medida que el proyecto madure y otros proyectos comiencen a aprovecharlo, el proyecto se deslizará hacia la izquierda. Esto significa que la estructura de los componentes de un proyecto puede variar con el tiempo y la madurez. Tiene más que ver con la forma en que se desarrolla y utiliza el proyecto que con lo que realmente hace.

CONCLUSIÓN

En el pasado, nuestra visión de la cohesión era mucho más simple de lo que implicaban el REP, el CCP y el CRP. Alguna vez pensamos que la cohesión era simplemente el atributo de que un módulo realiza una, y sólo una, función. Sin embargo, los tres principios de cohesión de componentes describen una variedad de cohesión mucho más compleja. Al elegir las clases para agrupar en componentes, debemos considerar las fuerzas opuestas involucradas en la reutilización y la capacidad de desarrollo. Equilibrar estas fuerzas con las necesidades de la aplicación no es trivial. Además, el equilibrio casi siempre es dinámico. Es decir, la partición que es apropiada hoy podría no serlo el año que viene. Como consecuencia, la composición de los componentes probablemente variará y evolucionará con el tiempo a medida que el enfoque del proyecto cambie de la capacidad de desarrollo a la reutilización.

1. Consulte la sección sobre "El problema de los gatitos" en [el Capítulo 27](#), "Servicios: grandes y pequeños". 2. Gracias a Tim Ottinger por esta idea.

14

ACOPLAMIENTO DE COMPONENTES



Los siguientes tres principios tratan de las relaciones entre componentes. Aquí nuevamente nos toparemos con la tensión entre capacidad de desarrollo y diseño lógico.

Las fuerzas que inciden sobre la arquitectura de una estructura componente son técnicas, políticas y volátiles.

LAS DEPENDENCIAS ACÍCLICAS PRINCIPIO

No permita ciclos en el gráfico de dependencia de componentes.

¿Alguna vez has trabajado todo el día, has conseguido que algunas cosas funcionen y luego te has ido a casa,

¿Solo para llegar a la mañana siguiente y descubrir que tus cosas ya no funcionan? ¿Por qué no funciona? ¡Porque alguien se quedó más tarde que tú y cambió algo de lo que dependes! Yo llamo a esto "el síndrome del día después".

El "síndrome del día después" ocurre en entornos de desarrollo donde muchos desarrolladores modifican los mismos archivos fuente. En proyectos relativamente pequeños con sólo unos pocos desarrolladores, no es un problema demasiado grande. Pero a medida que crece el tamaño del proyecto y el equipo de desarrollo, las mañanas siguientes pueden volverse una pesadilla. No es raro que pasen semanas sin que el equipo pueda construir una versión estable del proyecto. En cambio, todos siguen cambiando y cambiando su código tratando de que funcione con los últimos cambios que hizo otra persona.

En las últimas décadas, han surgido dos soluciones a este problema, ambas provenientes de la industria de las telecomunicaciones. El primero es "la construcción semanal" y el segundo es el Principio de Dependencias Acíclicas (ADP).

LA CONSTRUCCIÓN SEMANAL

La construcción semanal solía ser común en proyectos de tamaño mediano. Funciona así: todos los desarrolladores se ignoran entre sí durante los primeros cuatro días de la semana. Todos trabajan con copias privadas del código y no se preocupan por integrar su trabajo de forma colectiva. Luego, el viernes, integran todos sus cambios y construyen el sistema.

Este enfoque tiene la maravillosa ventaja de permitir a los desarrolladores vivir en un mundo aislado durante cuatro de cada cinco días. La desventaja, por supuesto, es la gran multa por integración que se paga el viernes.

Desafortunadamente, a medida que el proyecto crece, se vuelve menos factible terminar de integrarlo el viernes. La carga de la integración crece hasta que comienza a desbordarse hasta el sábado. Unos cuantos sábados de este tipo son suficientes para convencer a los desarrolladores de que la integración realmente debería comenzar el jueves, por lo que el inicio de la integración avanza lentamente hacia la mitad de la semana.

A medida que disminuye el ciclo de trabajo de desarrollo versus integración, también disminuye la eficiencia del equipo. Con el tiempo, esta situación se vuelve tan frustrante que los desarrolladores o los directores de proyecto declaran que el cronograma debería cambiarse a una construcción quincenal. Esto es suficiente por un tiempo, pero el tiempo de integración continúa

crecer con el tamaño del proyecto.

Al final, este escenario conduce a una crisis. Para mantener la eficiencia, el cronograma de construcción debe ampliarse continuamente, pero alargarlo aumenta los riesgos del proyecto. La integración y las pruebas se vuelven cada vez más difíciles de realizar y el equipo pierde el beneficio de una retroalimentación rápida.

ELIMINAR CICLOS DE DEPENDENCIA

La solución a este problema es dividir el entorno de desarrollo en componentes liberables. Los componentes se convierten en unidades de trabajo que pueden ser responsabilidad de un único desarrollador o de un equipo de desarrolladores. Cuando los desarrolladores consiguen que un componente funcione, lo liberan para que lo utilicen otros desarrolladores. Le asignan un número de versión y lo trasladan a un directorio para que lo utilicen otros equipos. Luego continúan modificando su componente en sus propias áreas privadas. Todos los demás usan la versión publicada.

A medida que aparecen nuevas versiones de un componente, otros equipos pueden decidir si adoptarán inmediatamente la nueva versión. Si deciden no hacerlo, simplemente continúan usando la versión anterior. Una vez que deciden que están listos, comienzan a utilizar la nueva versión.

Por tanto, ningún equipo está a merced de los demás. No es necesario que los cambios realizados en un componente tengan un efecto inmediato en otros equipos. Cada equipo puede decidir por sí mismo cuándo adaptar sus propios componentes a las nuevas versiones de los componentes. Además, la integración se produce en pequeños incrementos. No existe un momento único en el que todos los desarrolladores deban unirse e integrar todo lo que están haciendo.

Este es un proceso muy simple y racional, y se usa ampliamente. Sin embargo, para que funcione correctamente, debe gestionar la estructura de dependencia de los componentes. No puede haber ciclos. Si hay ciclos en la estructura de dependencia, entonces no se puede evitar el “síndrome del día después”.

Considere el diagrama de componentes de [la Figura 14.1](#). Muestra una estructura bastante típica de componentes ensamblados en una aplicación. La función de esta aplicación no es importante para los fines de este ejemplo. Lo importante es la estructura de dependencia de los componentes. Observe que esta estructura es un gráfico dirigido. Los componentes son los nodos y las relaciones de dependencia.

son los bordes dirigidos.

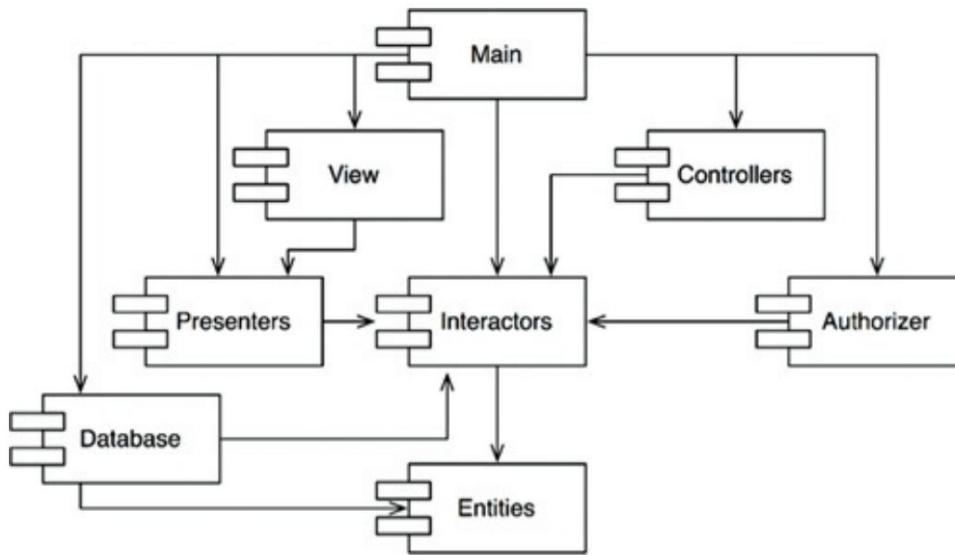


Figura 14.1 Diagrama de componentes típico

Observe una cosa más: independientemente del componente en el que comience, es imposible seguir las relaciones de dependencia y terminar en ese componente. Esta estructura no tiene ciclos. Es un gráfico acíclico dirigido (DAG).

Ahora considere lo que sucede cuando el equipo responsable de los Presentadores realiza una nueva versión de su componente. Es fácil saber quién se ve afectado por esta versión; simplemente sigue las flechas de dependencia hacia atrás. Por lo tanto, tanto **View** como **Main** se verán afectados. Los desarrolladores que actualmente trabajan en esos componentes tendrán que decidir cuándo integrar su trabajo con la nueva versión de **Presenters**.

Tenga en cuenta también que cuando se libera **Main**, no tiene ningún efecto en ninguno de los demás componentes del sistema. No conocen **Main** y no les importa cuándo cambia. Esto es bonito. Significa que el impacto de lanzar **Main** es relativamente pequeño.

Cuando los desarrolladores que trabajan en el componente **Presentadores** desean ejecutar una prueba de ese componente, solo necesitan crear su versión de **Presentadores** con las versiones de los componentes **Interactores** y **Entidades** que están usando actualmente. Ninguno de los otros componentes del sistema necesita estar involucrado. Esto es bonito. Significa que los desarrolladores que trabajan en **Presenters** tienen relativamente poco trabajo que hacer para configurar una prueba y que tienen relativamente pocas variables para

considerar.

Cuando llega el momento de liberar todo el sistema, el proceso continúa de abajo hacia arriba. Primero, se compila, prueba y lanza el componente Entidades . Luego se hace lo mismo para la base de datos y los interactores. A estos componentes les siguen Presentadores, Vista, Controladores y luego Autorizador. El principal va al final. Este proceso es muy claro y fácil de abordar. Sabemos cómo construir el sistema porque entendemos las dependencias entre sus partes.

EL EFECTO DE UN CICLO EN EL COMPONENTE GRÁFICO DE DEPENDENCIA

Supongamos que un nuevo requisito nos obliga a cambiar una de las clases en Entidades de modo que haga uso de una clase en Autorizador. Por ejemplo, digamos que la clase Usuario en Entidades usa la clase Permisos en Autorizador. Esto crea un ciclo de dependencia, como se muestra en la [Figura 14.2](#).

Este ciclo crea algunos problemas inmediatos. Por ejemplo, los desarrolladores que trabajan en el componente Base de datos saben que para lanzarlo, el componente debe ser compatible con Entidades. Sin embargo, con el ciclo implementado, el componente de la base de datos ahora también debe ser compatible con Authorizer. Pero el Autorizador depende de los interactantes. Esto hace que la base de datos sea mucho más difícil de publicar. Las entidades, el autorizador y los interactantes se han convertido, de hecho, en un gran componente, lo que significa que todos los desarrolladores que trabajan en cualquiera de esos componentes experimentarán el temido "síndrome del día después". Se pisarán unos sobre otros porque todos deben usar exactamente la misma versión de los componentes de cada uno.

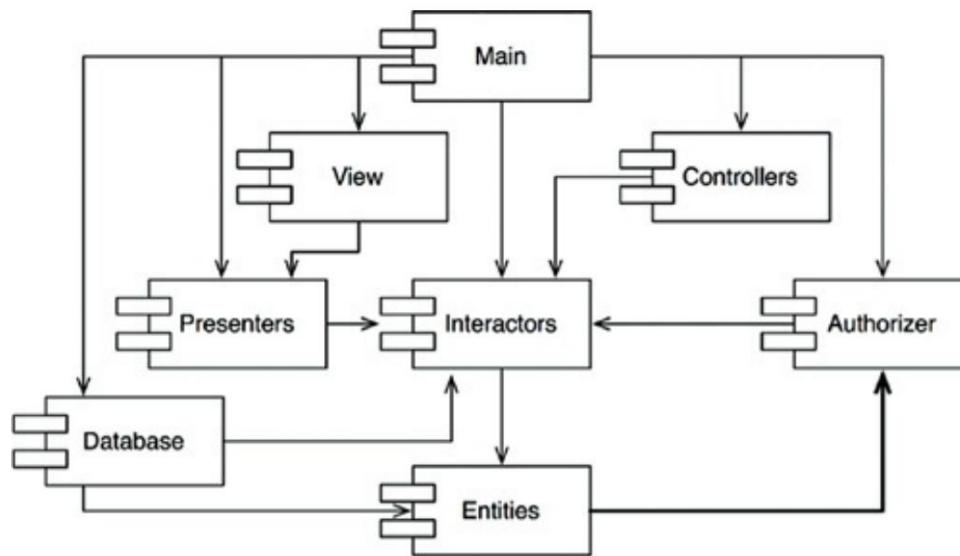


Figura 14.2 Un ciclo de dependencia

Pero esto es sólo parte del problema. Considere lo que sucede cuando queremos probar el componente Entidades . Para nuestro disgusto, descubrimos que debemos construir e integrar con Authorizer e Interactors. Este nivel de acoplamiento entre componentes es preocupante, si no intolerable.

Quizás te hayas preguntado por qué tienes que incluir tantas bibliotecas diferentes y tantas cosas de todos los demás, solo para ejecutar una prueba unitaria simple de una de tus clases. Si investigas un poco el asunto, probablemente descubras que hay ciclos en el gráfico de dependencia. Estos ciclos hacen que sea muy difícil aislar componentes. Las pruebas y lanzamientos unitarios se vuelven muy difíciles y propensos a errores. Además, los problemas de construcción crecen geométricamente con la cantidad de módulos.

Además, cuando hay ciclos en el gráfico de dependencia, puede resultar muy difícil determinar el orden en el que se deben construir los componentes. De hecho, probablemente no exista un orden correcto. Esto puede provocar problemas muy desagradables en lenguajes como Java que leen sus declaraciones de archivos binarios compilados.

ROMPIENDO EL CICLO

Siempre es posible romper un ciclo de componentes y restablecer el gráfico de dependencia como DAG. Hay dos mecanismos principales para hacerlo:

1. Aplicar el Principio de Inversión de Dependencia (DIP). En el caso de [la Figura 14.3](#), podríamos crear una interfaz que tenga los métodos que el Usuario necesita. Pudimos

Luego coloque esa interfaz en Entidades y heredéela en Authorizer. Esto invierte la dependencia entre las Entidades y el Autorizador, rompiendo así el ciclo.

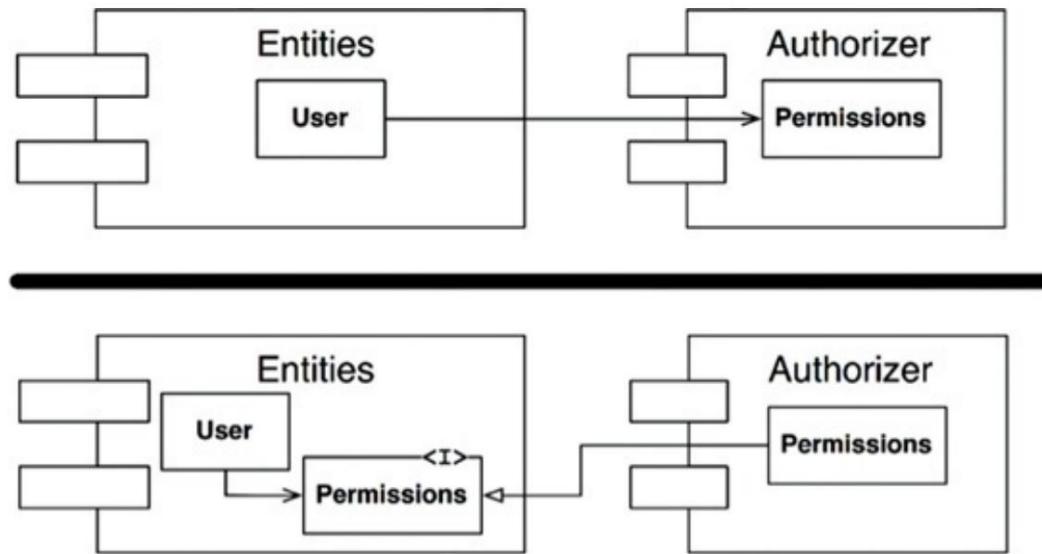


Figura 14.3 Inversión de la dependencia entre Entidades y Autorizador

2. Cree un nuevo componente del que dependan tanto las Entidades como el Autorizador .
Mueva las clases de las que ambos dependen a ese nuevo componente ([Figura 14.4](#)).

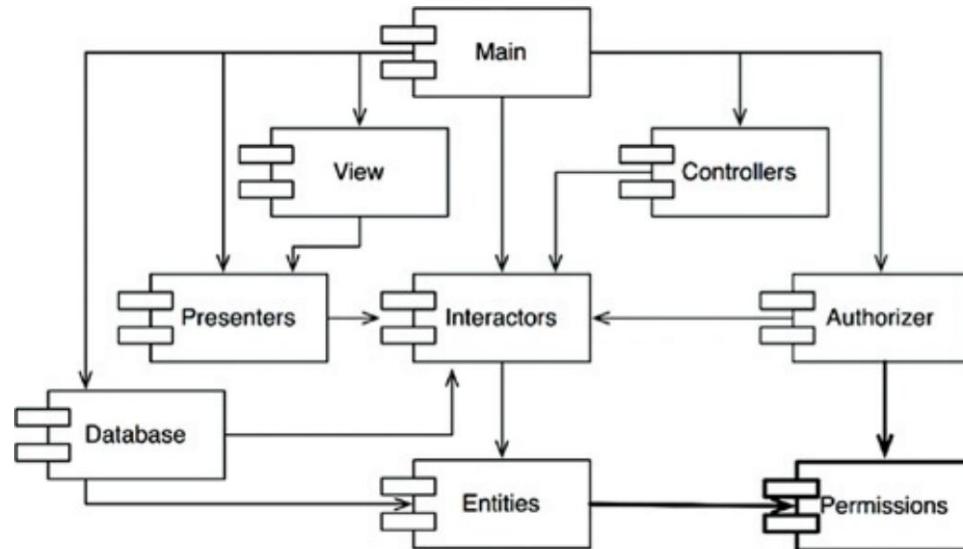


Figura 14.4 El nuevo componente del que dependen tanto las Entidades como el Autorizador

LOS “NERVIOSOS”

La segunda solución implica que la estructura del componente es volátil en presencia de requisitos cambiantes. De hecho, a medida que la aplicación crece, la estructura de dependencia de los componentes se tambalea y crece. Por lo tanto, la estructura de dependencia siempre debe ser monitoreada por ciclos. Cuando ocurren ciclos, es necesario romperlos de alguna manera. A veces esto significará crear nuevos componentes, haciendo crecer la estructura de dependencia.

DISEÑO DE ARRIBA ABAJO

Las cuestiones que hemos discutido hasta ahora conducen a una conclusión ineludible: la estructura de los componentes no se puede diseñar de arriba hacia abajo. No es una de las primeras cosas que se diseñan en el sistema, sino que evoluciona a medida que el sistema crece y cambia.

Algunos lectores pueden encontrar este punto contradictorio. Hemos llegado a esperar que las descomposiciones de grano grande, al igual que los componentes, también sean descomposiciones funcionales de alto nivel .

Cuando vemos una agrupación amplia, como una estructura de dependencia de componentes, creemos que los componentes deberían representar de alguna manera las funciones del sistema. Sin embargo, esto no parece ser un atributo de los diagramas de dependencia de componentes.

De hecho, los diagramas de dependencia de componentes tienen muy poco que ver con describir la función de la aplicación. En cambio, son un mapa de la capacidad de construcción y mantenimiento de la aplicación. Es por eso que no se diseñan al inicio del proyecto. No hay software que construir o mantener, por lo que no hay necesidad de un mapa de construcción y mantenimiento. Pero a medida que se acumulan más y más módulos en las primeras etapas de implementación y diseño, existe una creciente necesidad de gestionar las dependencias para que el proyecto pueda desarrollarse sin el "síndrome del día después". Además, queremos mantener los cambios lo más localizados posible, por lo que comenzamos a prestar atención al SRP y al CCP y a colocar las clases que probablemente cambien juntas.

Una de las principales preocupaciones de esta estructura de dependencia es el aislamiento de la volatilidad. No queremos que los componentes que cambian con frecuencia y por razones caprichosas afecten a componentes que de otro modo deberían ser estables. Por ejemplo, no queremos que los cambios cosméticos en la GUI tengan un impacto en nuestras reglas comerciales.

No queremos que la adición o modificación de informes tenga un impacto en nuestras políticas de más alto nivel. En consecuencia, los arquitectos crean y moldean el gráfico de dependencia de componentes para proteger los componentes estables de alto valor de los componentes volátiles.

A medida que la aplicación sigue creciendo, empezamos a preocuparnos por crear elementos reutilizables. En este punto, la PCR comienza a influir en la composición de los componentes. Finalmente, a medida que aparecen los ciclos, se aplica el ADP y el gráfico de dependencia de los componentes tiembla y crece.

Si intentáramos diseñar la estructura de dependencia de los componentes antes de diseñar cualquier clase, probablemente fracasaríamos bastante. No sabríamos mucho sobre el cierre común, no conoceríamos ningún elemento reutilizable y casi con seguridad crearíamos componentes que produjeran ciclos de dependencia. Así, la estructura de dependencia de los componentes crece y evoluciona con el diseño lógico del sistema.

LAS DEPENDENCIAS ESTABLES PRINCIPIO

Depende de la dirección de la estabilidad.

Los diseños no pueden ser completamente estáticos. Es necesaria cierta volatilidad si se quiere mantener el diseño. Al cumplir con el Principio de Cierre Común (PCC), creamos componentes que son sensibles a ciertos tipos de cambios pero inmunes a otros. Algunos de estos componentes están diseñados para ser volátiles. Esperamos que cambien.

Cualquier componente que esperemos que sea volátil no debería depender de un componente que sea difícil de cambiar. De lo contrario, el componente volátil también será difícil de cambiar.

Es la perversidad del software que un módulo que usted ha diseñado para que sea fácil de cambiar pueda resultar difícil de cambiar por otra persona que simplemente depende de él. No es necesario cambiar ni una línea de código fuente en su módulo, pero su módulo de repente será más difícil de cambiar. Al cumplir con el Principio de Dependencias Estables (SDP), nos aseguramos de que los módulos que están destinados a ser fáciles de cambiar no dependan de módulos que son más difíciles de cambiar.

cambiar.

ESTABILIDAD

¿Qué se entiende por "estabilidad"? Coloca una moneda de lado. ¿Es estable en esa posición? Probablemente dirías "no". Sin embargo, a menos que se le moleste, permanecerá en esa posición durante mucho tiempo. Por tanto, la estabilidad no tiene nada que ver directamente con la frecuencia del cambio. El centavo no cambia, pero es difícil considerarlo estable.

El Diccionario Webster dice que algo es estable si "no se mueve fácilmente". La estabilidad está relacionada con la cantidad de trabajo requerido para realizar un cambio. Por un lado, la moneda en pie no es estable porque requiere muy poco trabajo para derribarla. Por otro lado, una mesa es muy estable porque requiere un esfuerzo considerable darle la vuelta.

¿Cómo se relaciona esto con el software? Muchos factores pueden hacer que un componente de software sea difícil de cambiar; por ejemplo, su tamaño, complejidad y claridad, entre otras características. Ignoraremos todos esos factores y nos centraremos en algo diferente aquí. Una forma segura de hacer que un componente de software sea difícil de cambiar es hacer que muchos otros componentes de software dependan de él. Un componente con muchas dependencias entrantes es muy estable porque requiere mucho trabajo para conciliar cualquier cambio con todos los componentes dependientes.

El diagrama de [la figura 14.5](#) muestra X, que es un componente estable. Tres componentes dependen de X, por lo que tiene tres buenas razones para no cambiar. Decimos que X es responsable de esos tres componentes. Por el contrario, X no depende de nada, por lo que no tiene ninguna influencia externa que lo haga cambiar. Decimos que es independiente.

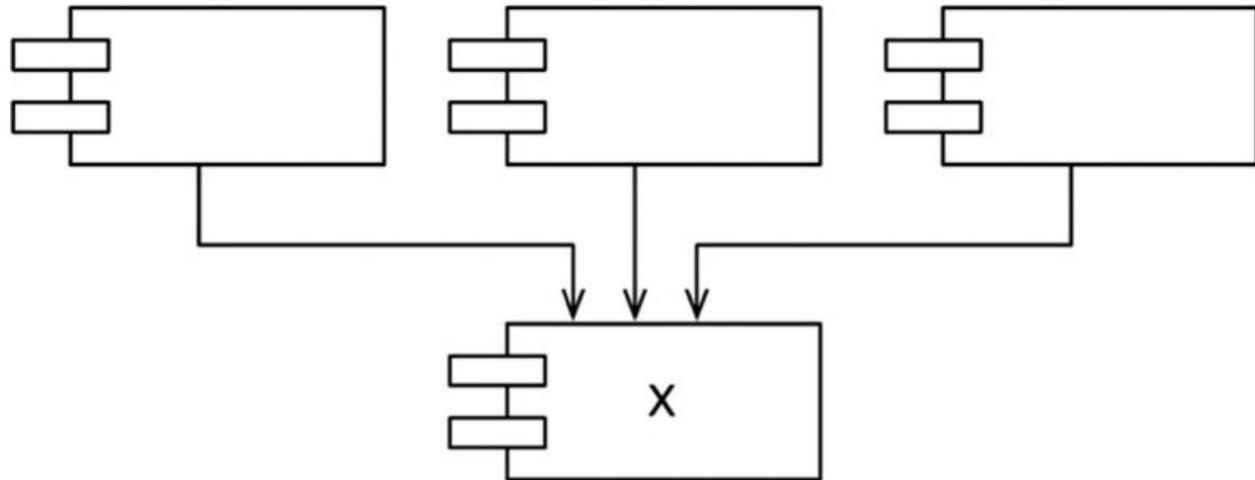


Figura 14.5 X: un componente estable

[La figura 14.6](#) muestra Y, que es un componente muy inestable. Ningún otro componente depende de Y, por lo que decimos que es irresponsable. Y también tiene tres componentes de los que depende, por lo que los cambios pueden provenir de tres fuentes externas. Decimos que Y es dependiente.

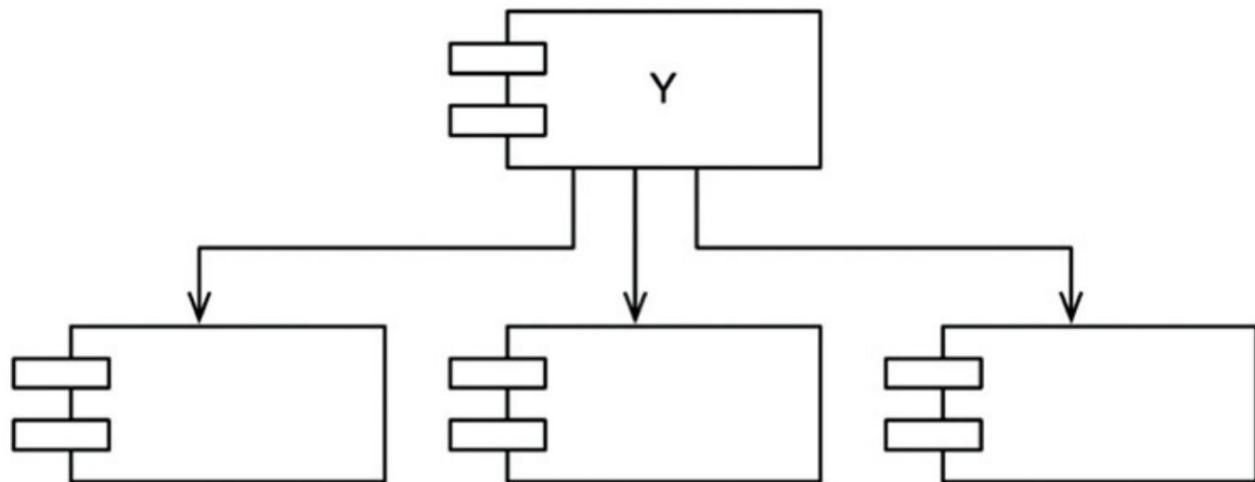


Figura 14.6 Y: un componente muy inestable

MÉTRICAS DE ESTABILIDAD

¿Cómo podemos medir la estabilidad de un componente? Una forma es contar la cantidad de dependencias que entran y salen de ese componente. Estos recuentos nos permitirán calcular la estabilidad posicional del componente.

- Fan-in: Dependencias entrantes. Esta métrica identifica el número de clases.

fuera de este componente que dependen de clases dentro del componente. • Fan-out: Dependencias salientes. Esta métrica identifica la cantidad de clases dentro de este componente que dependen de clases fuera del componente. • I: Inestabilidad: $I = \frac{\text{Fan-out}}{\text{Fan-in} + \text{Fan-out}}$. Esta métrica tiene el rango [0, 1]. $I = 0$ indica un componente máximamente estable. $I = 1$ indica un componente máximamente inestable.

Las métricas de entrada y salida [1](#) se calculan contando el número de clases fuera del componente en cuestión que tienen dependencias con las clases dentro del componente en cuestión. Considere el ejemplo de [la Figura 14.7](#).

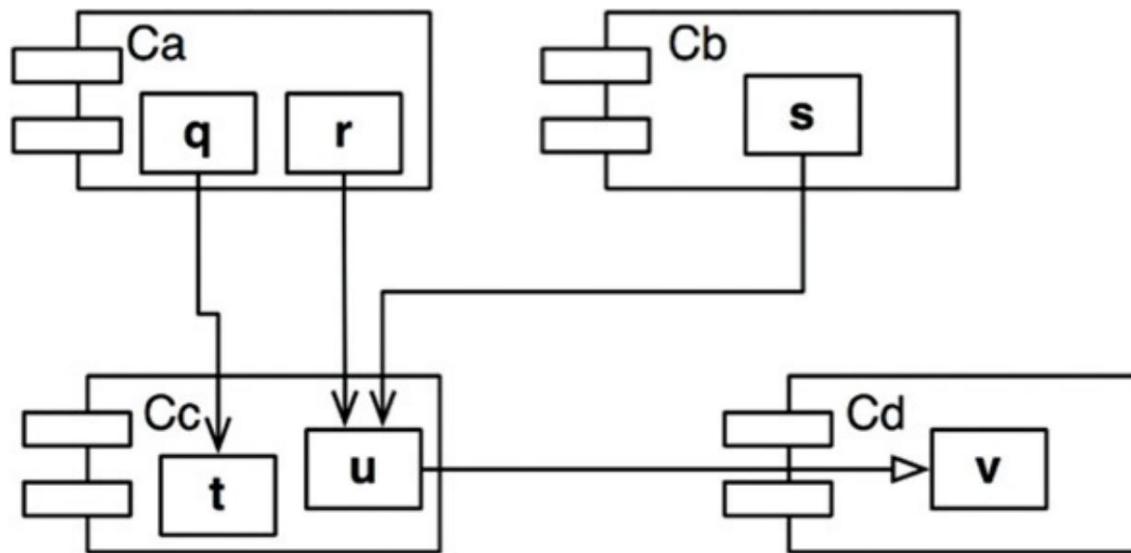


Figura 14.7 Nuestro ejemplo

Digamos que queremos calcular la estabilidad del componente Cc. Encontramos que hay tres clases fuera de Cc que dependen de las clases en Cc. Por lo tanto, Fan-in = 3. Además, hay una clase fuera de Cc de la que dependen las clases en Cc . Por lo tanto, Fan-out = 1 e $I = 1/4$.

En C++, estas dependencias normalmente se representan mediante declaraciones `#include` . De hecho, la métrica I es más fácil de calcular cuando has organizado tu código fuente de manera que haya una clase en cada archivo fuente. En Java, la métrica I se puede calcular contando declaraciones de importación y nombres calificados.

Cuando la métrica I es igual a 1, significa que ningún otro componente depende de este componente (Fan-in = 0), y este componente depende de otros componentes (Fan-out > 0). Esta situación es lo más inestable que puede llegar a ser un componente; es

irresponsable y dependiente. Su falta de dependientes no le da al componente ninguna razón para no cambiar, y los componentes de los que depende pueden darle amplias razones para cambiar.

Por el contrario, cuando la métrica I es igual a 0, significa que otros componentes dependen del componente ($\text{Fan-in} > 0$), pero no depende de ningún otro componente ($\text{Fan-out} = 0$). Este componente es responsable e independiente. Es lo más estable posible. Sus dependientes dificultan el cambio del componente y no tiene dependencias que puedan obligarlo a cambiar.

El SDP dice que la métrica I de un componente debe ser mayor que la métrica I de los componentes de los que depende. Es decir, las métricas I deberían disminuir en la dirección de la dependencia.

NO TODOS LOS COMPONENTES DEBEN SER ESTABLES

Si todos los componentes de un sistema fueran máximamente estables, el sistema sería inmutable. Ésta no es una situación deseable. De hecho, queremos diseñar nuestra estructura de componentes de modo que algunos componentes sean inestables y otros estables. El diagrama de [la Figura 14.8](#) muestra una configuración ideal para un sistema con tres componentes.

Los componentes cambiables están en la parte superior y dependen del componente estable en la parte inferior. Poner los componentes inestables en la parte superior del diagrama es una convención útil porque cualquier flecha que apunte hacia arriba viola el SDP (y, como veremos más adelante, el ADP).

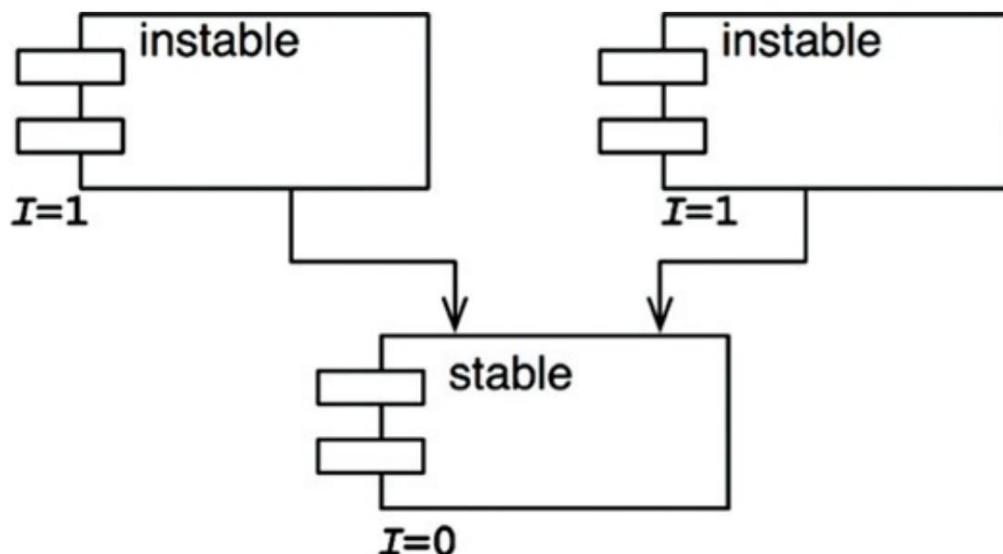


Figura 14.8 Una configuración ideal para un sistema con tres componentes

El diagrama de [la Figura 14.9](#) muestra cómo se puede violar el SDP.

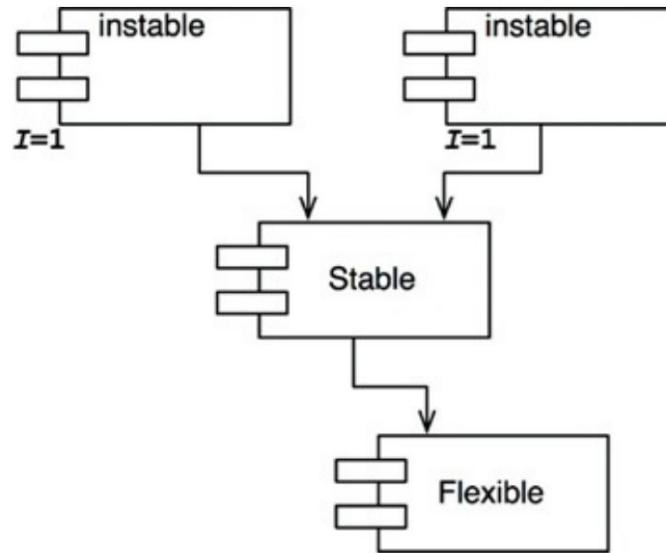


Figura 14.9 Violación del SDP

Flexible es un componente que hemos diseñado para que sea fácil de cambiar. Queremos que Flexible sea inestable. Sin embargo, algún desarrollador, que trabaja en el componente denominado Stable, ha dependido de Flexible. Esto viola el SDP porque la métrica I para Estable es mucho más pequeña que la métrica I para Flexible.

Como resultado, Flexible ya no será fácil de cambiar. Un cambio a Flexible nos obligará a tratar con Estable y todos sus dependientes.

Para solucionar este problema, de alguna manera tenemos que romper la dependencia de Estable respecto de Flexible. ¿Por qué existe esta dependencia? Supongamos que hay una clase C dentro de Flexible que otra clase U dentro de Estable necesita usar ([Figura 14.10](#)).

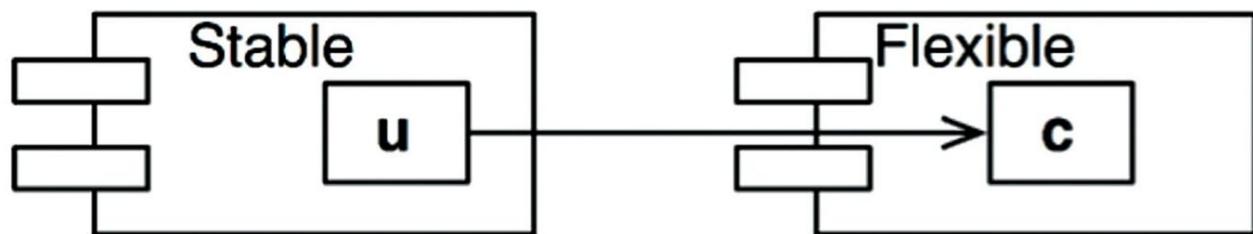


Figura 14.10 U dentro de Estable usa C dentro de Flexible

Podemos solucionar este problema empleando el DIP. Creamos una clase de interfaz llamada US y la colocamos en un componente llamado UServer. Nos aseguramos de que esta interfaz declare

todos los métodos que U necesita utilizar. Luego hacemos que C implemente esta interfaz como se muestra en la Figura 14.11. Esto rompe la dependencia de Estable respecto de Flexible y obliga a ambos componentes a depender de UServer. UServer es muy estable ($I = 0$) y Flexible conserva su inestabilidad necesaria ($I = 1$). Todas las dependencias fluyen ahora en la dirección de I decreciente .

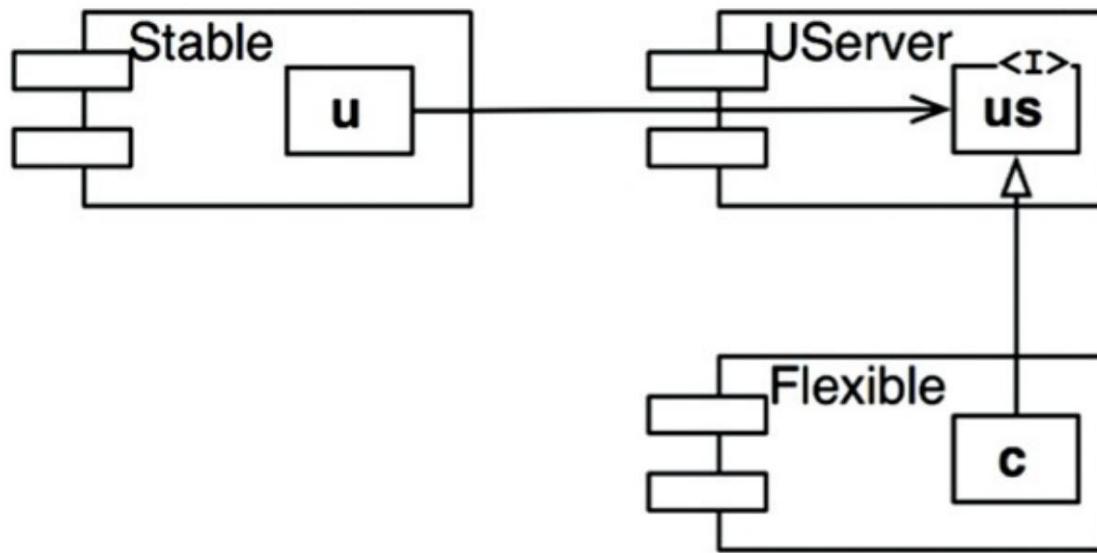


Figura 14.11 C implementa la clase de interfaz US

Componentes abstractos

Puede que le resulte extraño que creemos un componente (en este ejemplo, UService) que no contenga nada más que una interfaz. ¡Un componente de este tipo no contiene ningún código ejecutable! Sin embargo, resulta que esta es una táctica muy común y necesaria cuando se utilizan lenguajes de tipo estático como Java y C#. Estos componentes abstractos son muy estables y, por lo tanto, son objetivos ideales para que dependan los componentes menos estables.

Cuando se utilizan lenguajes de tipado dinámico como Ruby y Python, estos componentes abstractos no existen en absoluto, ni tampoco las dependencias que se habrían dirigido a ellos. Las estructuras de dependencia en estos lenguajes son mucho más simples porque la inversión de dependencia no requiere ni la declaración ni la herencia de interfaces.

LAS ABSTRACCIONES ESTABLES

PRINCIPIO

Un componente debe ser tan abstracto como estable.

¿DÓNDE PONDEMOS LA POLÍTICA DE ALTO NIVEL?

Parte del software del sistema no debería cambiar con mucha frecuencia. Este software representa decisiones políticas y de arquitectura de alto nivel. No queremos que estas decisiones comerciales y arquitectónicas sean volátiles. Por tanto, el software que encapsula las políticas de alto nivel del sistema debe ubicarse en componentes estables ($I = 0$). Los componentes inestables ($I = 1$) deben contener solo el software que es volátil: software que queremos poder cambiar rápida y fácilmente.

Sin embargo, si las políticas de alto nivel se colocan en componentes estables, entonces será difícil cambiar el código fuente que representa esas políticas. Esto podría hacer que la arquitectura general sea inflexible. ¿Cómo puede un componente que es máximamente estable ($I = 0$) ser lo suficientemente flexible como para resistir el cambio? La respuesta se encuentra en el OCP. Este principio nos dice que es posible y deseable crear clases que sean lo suficientemente flexibles como para ampliarlas sin necesidad de modificaciones. ¿Qué tipo de clases se ajustan a este principio? Clases abstractas .

INTRODUCCIÓN AL PRINCIPIO DE ABSTRACCIÓN ESTABLE

El Principio de Abstracciones Estables (SAP) establece una relación entre estabilidad y abstracción. Por un lado, dice que un componente estable también debe ser abstracto para que su estabilidad no impida su ampliación. Por otro lado, dice que un componente inestable debe ser concreto ya que su inestabilidad permite que el código concreto que contiene se pueda cambiar fácilmente.

Por lo tanto, para que un componente sea estable, debe constar de interfaces y clases abstractas para que pueda ampliarse. Los componentes estables que son extensibles son flexibles y no limitan demasiado la arquitectura.

El SAP y el SDP combinados equivalen al DIP para los componentes. Esto es cierto porque el SDP dice que las dependencias deben ir en dirección a la estabilidad, y el SAP dice que la estabilidad implica abstracción. Por tanto, las dependencias corren en dirección a la abstracción.

El DIP, sin embargo, es un principio que se ocupa de las clases, y con las clases no hay matices. O una clase es abstracta o no lo es. La combinación del SDP y el SAP se ocupa de componentes, y permite que un componente pueda ser parcialmente abstracto y parcialmente estable.

MEDICIÓN DE LA ABSTRACCIÓN

La métrica A es una medida de la abstracción de un componente. Su valor es simplemente la proporción de interfaces y clases abstractas en un componente con respecto al número total de clases en el componente.

- Nc: El número de clases en el componente.
- Na: El número de clases abstractas e interfaces en el componente.
- A: Abstracción. $A = Na \div Nc$.

La métrica A varía de 0 a 1. Un valor de 0 implica que el componente no tiene ninguna clase abstracta. Un valor de 1 implica que el componente no contiene más que clases abstractas.

LA SECUENCIA PRINCIPAL

Ahora estamos en condiciones de definir la relación entre estabilidad (I) y abstracción (A). Para hacerlo, creamos una gráfica con A en el eje vertical y I en el eje horizontal ([Figura 14.12](#)). Si trazamos los dos tipos "buenos" de componentes en este gráfico, encontraremos los componentes que son máximamente estables y abstractos en la parte superior izquierda en (0, 1). Los componentes que son máximamente inestables y concretos están en la parte inferior derecha en (1, 0).

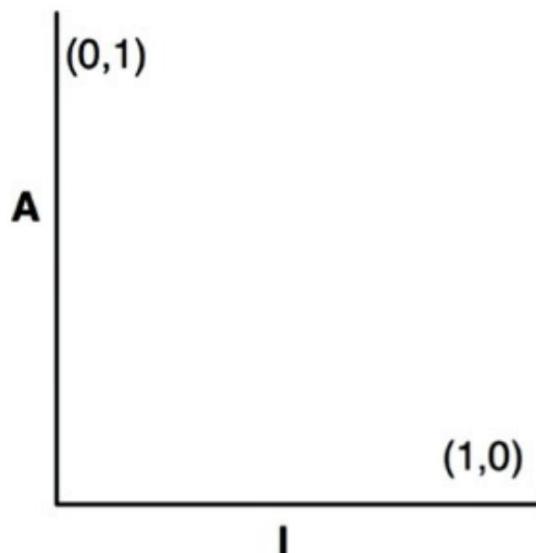


Figura 14.12 El gráfico I/A

No todos los componentes caen en una de estas dos posiciones, porque los componentes suelen tener grados de abstracción y estabilidad. Por ejemplo, es muy común que una clase abstracta derive de otra clase abstracta. La derivada es una abstracción que tiene una dependencia. Por tanto, aunque sea máximamente abstracto, no será máximamente estable. Su dependencia disminuirá su estabilidad.

Como no podemos imponer una regla de que todos los componentes se encuentren en (0, 1) o (1, 0), debemos suponer que hay un lugar geométrico de puntos en la gráfica A/I que define posiciones razonables para los componentes. Podemos inferir cuál es ese locus encontrando las áreas donde los componentes no deberían estar; en otras palabras, determinando las zonas de exclusión (Figura 11.13).

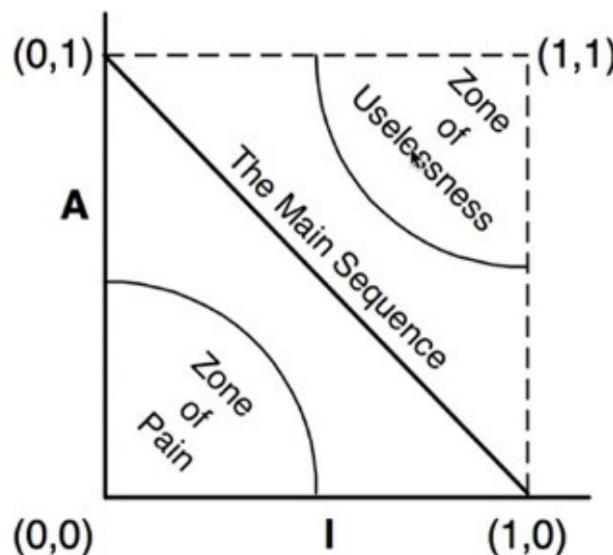


Figura 14.13 Zonas de exclusión

La zona del dolor

Considere un componente en el área de $(0, 0)$. Se trata de un componente muy estable y concreto. Un componente de este tipo no es deseable porque es rígido. No se puede ampliar porque no es abstracto y es muy difícil de cambiar debido a su estabilidad. Por lo tanto, normalmente no esperamos ver componentes bien diseñados cerca de $(0, 0)$. El área alrededor de $(0, 0)$ es una zona de exclusión llamada Zona del Dolor.

De hecho, algunas entidades de software se encuentran dentro de la Zona del Dolor. Un ejemplo sería un esquema de base de datos. Los esquemas de bases de datos son notoriamente volátiles, extremadamente concretos y muy dependientes de ellos. Ésta es una de las razones por las que la interfaz entre las aplicaciones OO y las bases de datos es tan difícil de gestionar, y por la que las actualizaciones de esquemas son generalmente dolorosas.

Otro ejemplo de software que se encuentra cerca del área de $(0, 0)$ es una biblioteca de utilidades concreta. Aunque dicha biblioteca tiene una métrica I de 1, en realidad puede ser no volátil. Considere el componente `String`, por ejemplo. Aunque todas las clases que contiene son concretas, se usa con tanta frecuencia que cambiarlo crearía caos. Por tanto, `String` no es volátil.

Los componentes no volátiles son inofensivos en la zona $(0, 0)$ ya que no es probable que se modifiquen. Por esta razón, sólo los componentes de software volátiles son problemáticos en la Zona del Dolor. Cuanto más volátil sea un componente en la Zona del Dolor, más “doloroso” será. De hecho, podríamos considerar la volatilidad como un tercer eje del gráfico. Teniendo esto en cuenta, la figura 14.13 muestra sólo el plano más doloroso, donde la volatilidad = 1.

La zona de la inutilidad

Considere un componente cerca de $(1, 1)$. Esta ubicación no es deseable porque es máximamente abstracta, pero no tiene dependientes. Estos componentes son inútiles. Por eso esta área se llama Zona de Inutilidad.

Las entidades de software que habitan esta región son una especie de detritos. A menudo son clases abstractas sobrantes que nadie nunca implementó. Los encontramos en los sistemas de vez en cuando, en la base del código, sin usar.

Un componente que tenga una posición profunda dentro de la Zona de Inutilidad debe contener una fracción significativa de dichas entidades. Claramente, la presencia de entidades tan inútiles es indeseable.

EVITANDO LAS ZONAS DE EXCLUSIÓN

Parece claro que nuestros componentes más volátiles deben mantenerse lo más lejos posible de ambas zonas de exclusión. El lugar geométrico de los puntos que están máximamente distantes de cada zona es la línea que conecta $(1, 0)$ y $(0, 1)$. A esta línea la llamo Secuencia Principal. [2](#)

Un componente que se encuentra en la secuencia principal no es "demasiado abstracto" para su estabilidad, ni "demasiado inestable" para su abstracción. No es inútil ni particularmente doloroso. Se depende de él en la medida en que es abstracto y depende de otros en la medida en que es concreto.

La posición más deseable para un componente es uno de los dos puntos finales de la secuencia principal. Los buenos arquitectos se esfuerzan por colocar la mayoría de sus componentes en esos puntos finales. Sin embargo, en mi experiencia, una pequeña fracción de los componentes de un sistema grande no son ni perfectamente abstractos ni perfectamente estables. Esos componentes tienen las mejores características si están en la secuencia principal o cerca de ella.

DISTANCIA DE LA SECUENCIA PRINCIPAL

Esto nos lleva a nuestra última métrica. Si es deseable que los componentes estén en la secuencia principal o cerca de ella, entonces podemos crear una métrica que mida qué tan lejos está un componente de este ideal.

- **D3:** Distancia. $D = |A+I-1|$. El rango de esta métrica es $[0, 1]$. Un valor de 0 indica que el componente está directamente en la secuencia principal. Un valor de 1 indica que el componente está lo más lejos posible de la Secuencia Principal.

Dada esta métrica, se puede analizar un diseño para determinar su conformidad general con la Secuencia Principal. Se puede calcular la métrica D para cada componente. Cualquier componente que tenga un valor D no cercano a cero puede reexaminarse y reestructurarse.

También es posible el análisis estadístico de un diseño. Podemos calcular la media y la varianza de todas las métricas D para los componentes dentro de un diseño. Esperaríamos que un diseño conforme tuviera una media y una varianza cercanas a cero.

La variación se puede utilizar para establecer "límites de control" con el fin de identificar componentes que son "excepcionales" en comparación con todos los demás.

En el diagrama de dispersión de [la figura 14.14](#), vemos que la mayor parte de los componentes se encuentran a lo largo de la secuencia principal, pero algunos de ellos están a más de una desviación estándar ($Z = 1$) de la media. Vale la pena examinar más de cerca estos componentes aberrantes. Por alguna razón, son muy abstractos con pocos dependientes o muy concretos con muchos dependientes.

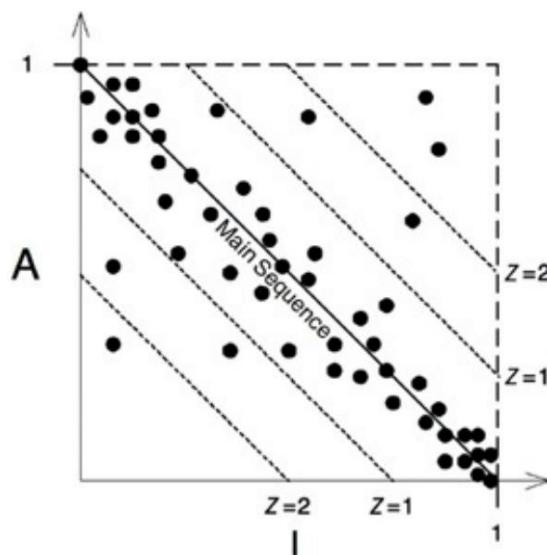


Figura 14.14 Diagrama de dispersión de los componentes

Otra forma de utilizar las métricas es trazar la métrica D de cada componente a lo largo del tiempo. El gráfico de [la figura 14.15](#) es una maqueta de dicho diagrama. Puede ver que algunas dependencias extrañas se han ido infiltrando en el componente Nómica en las últimas versiones. El gráfico muestra un umbral de control en $D = 0,1$. El punto R2.1 ha superado este límite de control, por lo que valdría la pena descubrir por qué este componente está tan lejos de la secuencia principal.

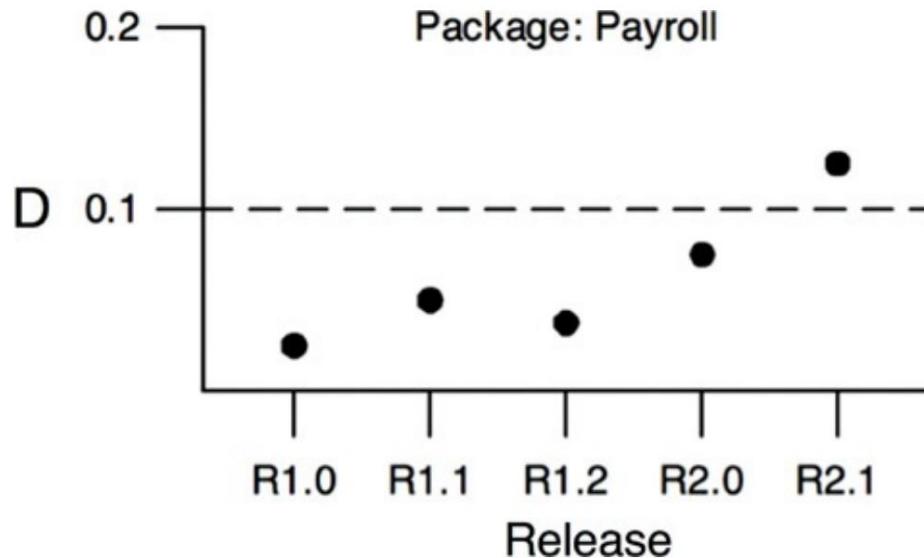


Figura 14.15 Gráfico de D para un solo componente a lo largo del tiempo

CONCLUSIÓN

Las métricas de gestión de dependencia descritas en este capítulo miden la conformidad de un diseño con un patrón de dependencia y abstracción que creo que es un patrón "bueno". La experiencia ha demostrado que ciertas dependencias son buenas y otras malas. Este patrón refleja esa experiencia. Sin embargo, una métrica no es un dios; es simplemente una medida frente a un estándar arbitrario. Estas métricas son imperfectas, en el mejor de los casos, pero espero que las encuentres útiles.

1. En publicaciones anteriores, utilicé los nombres Acoplamientos eferentes y aferentes (Ce y Ca) para Fan-out y Fan-in, respectivamente. Eso fue simplemente arrogancia de mi parte: me gustaba la metáfora del sistema nervioso central.
2. El autor pide indulgencia al lector por la arrogancia de tomar prestado un término tan importante de astronomía.
3. En publicaciones anteriores, llamé a esta métrica D'. No veo ninguna razón para continuar con esa práctica.

V

ARQUITECTURA

15

¿ QUÉ ES LA ARQUITECTURA?



La palabra "arquitectura" evoca visiones de poder y misterio. Nos hace pensar en decisiones importantes y profundas proezas técnicas. La arquitectura de software está en la cima de los logros técnicos. Cuando pensamos en un arquitecto de software, pensamos en alguien que tiene poder y que inspira respeto. ¿Qué joven aspirante a desarrollador de software no ha soñado con convertirse algún día en arquitecto de software?

Pero ¿qué es la arquitectura de software? ¿Qué hace un arquitecto de software y cuándo lo hace?

En primer lugar, un arquitecto de software es un programador; y sigue siendo programador. Nunca caiga en la mentira que sugiere que los arquitectos de software retroceden

del código para centrarse en cuestiones de nivel superior. ¡Ellos no! Los arquitectos de software son los mejores programadores y continúan asumiendo tareas de programación, mientras también guían al resto del equipo hacia un diseño que maximice la productividad.

Es posible que los arquitectos de software no escriban tanto código como otros programadores, pero continúan participando en tareas de programación. Lo hacen porque no pueden hacer su trabajo correctamente si no experimentan los problemas que están creando para el resto de los programadores.

La arquitectura de un sistema de software es la forma que le dan a ese sistema quienes lo construyen. La forma de esa forma está en la división de ese sistema en componentes, la disposición de esos componentes y las formas en que esos componentes se comunican entre sí.

El propósito de esa forma es facilitar el desarrollo, implementación, operación y mantenimiento del sistema de software contenido en ella.

La estrategia detrás de esa facilitación es dejar abiertas tantas opciones como sea posible, durante el mayor tiempo posible.

Quizás esta afirmación te haya sorprendido. Quizás pensó que el objetivo de la arquitectura de software era hacer que el sistema funcionara correctamente. Ciertamente queremos que el sistema funcione correctamente, y ciertamente la arquitectura del sistema debe respaldarlo como una de sus principales prioridades.

Sin embargo, la arquitectura de un sistema tiene muy poca relación con su funcionamiento. Hay muchos sistemas por ahí, con arquitecturas terribles, que funcionan bien. Sus problemas no residen en su funcionamiento; más bien, ocurren en su implementación, mantenimiento y desarrollo continuo.

Esto no quiere decir que la arquitectura no desempeñe ningún papel a la hora de respaldar el comportamiento adecuado del sistema. Ciertamente lo es, y ese papel es fundamental. Pero el papel es pasivo y cosmético, no activo ni esencial. Hay pocas opciones de comportamiento , si es que hay alguna, que la arquitectura de un sistema pueda dejar abiertas.

El propósito principal de la arquitectura es soportar el ciclo de vida del sistema. Una buena arquitectura hace que el sistema sea fácil de entender, de desarrollar, de mantener y de implementar. El objetivo final es minimizar el costo de vida útil del sistema y maximizar la productividad del programador.

DESARROLLO

Es poco probable que un sistema de software que sea difícil de desarrollar tenga una vida útil larga y saludable. Por lo tanto, la arquitectura de un sistema debe hacer que ese sistema sea fácil de desarrollar para los equipos que lo desarrollan.

Diferentes estructuras de equipo implican diferentes decisiones arquitectónicas. Por un lado, un pequeño equipo de cinco desarrolladores puede trabajar juntos de manera bastante efectiva para desarrollar un sistema monolítico sin componentes ni interfaces bien definidos. De hecho, un equipo así probablemente encontraría las restricciones de una arquitectura como un impedimento durante los primeros días de desarrollo. Esta es probablemente la razón por la que tantos sistemas carecen de una buena arquitectura: comenzaron sin ninguna, porque el equipo era pequeño y no quería el impedimento de una superestructura.

Por otro lado, un sistema desarrollado por cinco equipos diferentes, cada uno de los cuales incluye siete desarrolladores, no puede avanzar a menos que el sistema esté dividido en componentes bien definidos con interfaces estables y confiables. Si no se consideran otros factores, la arquitectura de ese sistema probablemente evolucionará hacia cinco componentes, uno para cada equipo.

No es probable que dicha arquitectura de componente por equipo sea la mejor arquitectura para la implementación, operación y mantenimiento del sistema. Sin embargo, es la arquitectura hacia la que gravitará un grupo de equipos si están impulsados únicamente por un cronograma de desarrollo.

DESPLIEGUE

Para ser eficaz, un sistema de software debe poder implementarse. Cuanto mayor sea el coste de implementación, menos útil será el sistema. Entonces, el objetivo de una arquitectura de software debería ser crear un sistema que pueda implementarse fácilmente con una sola acción.

Desafortunadamente, la estrategia de implementación rara vez se considera durante el desarrollo inicial. Esto conduce a arquitecturas que pueden hacer que el sistema sea fácil de desarrollar, pero que lo hacen muy difícil de implementar.

Por ejemplo, en el desarrollo inicial de un sistema, los desarrolladores pueden decidir utilizar una "arquitectura de microservicio". Pueden encontrar que este enfoque hace que el sistema sea muy fácil de desarrollar ya que los límites de los componentes son muy firmes y

las interfaces son relativamente estables. Sin embargo, cuando llega el momento de implementar el sistema, es posible que descubran que la cantidad de microservicios se ha vuelto abrumadora; La configuración de las conexiones entre ellos y el momento de su inicio también pueden resultar una gran fuente de errores.

Si los arquitectos hubieran considerado los problemas de implementación desde el principio, podrían haber optado por menos servicios, un híbrido de servicios y componentes en proceso, y un medio más integrado para gestionar las interconexiones.

OPERACIÓN

El impacto de la arquitectura en la operación del sistema tiende a ser menos dramático que el impacto de la arquitectura en el desarrollo, implementación y mantenimiento. Casi cualquier dificultad operativa se puede resolver agregando más hardware al sistema sin afectar drásticamente la arquitectura del software.

De hecho, hemos visto que esto sucede una y otra vez. Los sistemas de software que tienen arquitecturas ineficientes a menudo pueden funcionar de manera efectiva simplemente agregando más almacenamiento y más servidores. El hecho de que el hardware sea barato y las personas caras significa que las arquitecturas que impiden el funcionamiento no son tan costosas como las arquitecturas que impiden el desarrollo, la implementación y el mantenimiento.

Esto no quiere decir que una arquitectura que esté bien adaptada al funcionamiento del sistema no sea deseable. ¡Es! Lo que pasa es que la ecuación de costos se inclina más hacia el desarrollo, la implementación y el mantenimiento.

Dicho esto, la arquitectura desempeña otro papel en el funcionamiento del sistema: una buena arquitectura de software comunica las necesidades operativas del sistema.

Quizás una mejor manera de decir esto es que la arquitectura de un sistema hace que el funcionamiento del sistema sea fácilmente evidente para los desarrolladores. La arquitectura debe revelar la operación. La arquitectura del sistema debe elevar los casos de uso, las características y los comportamientos requeridos del sistema a entidades de primera clase que sean puntos de referencia visibles para los desarrolladores. Esto simplifica la comprensión del sistema y, por lo tanto, ayuda enormemente en el desarrollo y mantenimiento.

MANTENIMIENTO

De todos los aspectos de un sistema de software, el mantenimiento es el más costoso. El desfile interminable de nuevas funciones y el inevitable rastro de defectos y correcciones consumen grandes cantidades de recursos humanos.

El costo principal del mantenimiento está en la espeleología y el riesgo. La espeleología es el costo de investigar el software existente, tratando de determinar el mejor lugar y la mejor estrategia para agregar una nueva característica o reparar un defecto. Al realizar tales cambios, siempre existe la posibilidad de crear defectos involuntarios, lo que aumenta el costo del riesgo.

Una arquitectura cuidadosamente pensada mitiga enormemente estos costos. Al separar el sistema en componentes y aislar esos componentes a través de interfaces estables, es posible iluminar el camino para funciones futuras y reducir en gran medida el riesgo de rotura involuntaria.

MANTENER LAS OPCIONES ABIERTAS

Como describimos en un capítulo anterior, el software tiene dos tipos de valor: el valor de su comportamiento y el valor de su estructura. El segundo de ellos es el mayor de los dos porque es este valor el que hace que el software sea suave.

El software se inventó porque necesitábamos una forma de cambiar rápida y fácilmente el comportamiento de las máquinas. Pero esa flexibilidad depende fundamentalmente de la forma del sistema, la disposición de sus componentes y la forma en que esos componentes están interconectados.

La forma de mantener el software suave es dejar abiertas tantas opciones como sea posible, durante el mayor tiempo posible. ¿Cuáles son las opciones que debemos dejar abiertas? Son los detalles los que no importan.

Todos los sistemas de software se pueden descomponer en dos elementos principales: política y detalles. El elemento de política incorpora todas las reglas y procedimientos comerciales. La política es donde reside el verdadero valor del sistema.

Los detalles son aquellas cosas que son necesarias para permitir que los humanos, otros sistemas y programadores se comuniquen con la política, pero que no afectan la

comportamiento de la política en absoluto. Incluyen dispositivos IO, bases de datos, sistemas web, servidores, marcos, protocolos de comunicación, etc.

El objetivo del arquitecto es crear una forma para el sistema que reconozca la política como el elemento más esencial del sistema y al mismo tiempo haga que los detalles sean irrelevantes para esa política. Esto permite retrasar y aplazar las decisiones sobre esos detalles .

Por ejemplo:

- No es necesario elegir un sistema de base de datos en los primeros días de desarrollo, porque a la política de alto nivel no debería importarle qué tipo de base de datos se utilizará. De hecho, si el arquitecto es cuidadoso, a la política de alto nivel no le importará si la base de datos es relacional, distribuida, jerárquica o simplemente archivos planos.
- No es necesario elegir un servidor web en las primeras etapas del desarrollo, porque la política de alto nivel no debe saber que se está entregando a través de la web. Si la política de alto nivel desconoce HTML, AJAX, JSP, JSF o cualquier otra sopa de letras del desarrollo web, entonces no necesita decidir qué sistema web usar hasta mucho más adelante en el proyecto. De hecho, ni siquiera tiene que decidir si el sistema se entregará a través de la web.
- No es necesario adoptar REST en una fase temprana del desarrollo, porque la política de alto nivel debe ser independiente de la interfaz con el mundo exterior. Tampoco es necesario adoptar un marco de microservicios o un marco SOA. Una vez más, la política de alto nivel no debería preocuparse por estas cosas.
- No es necesario adoptar un marco de inyección de dependencia en las primeras etapas del desarrollo, porque a la política de alto nivel no debería importarle cómo se resuelven las dependencias.

Creo que entiendes el punto. Si se puede desarrollar una política de alto nivel sin comprometerse con los detalles que la rodean, se pueden retrasar y aplazar las decisiones sobre esos detalles durante mucho tiempo. Y cuanto más espere para tomar esas decisiones, más información tendrá para tomarlas correctamente.

Esto también te deja la opción de probar diferentes experimentos. Si tiene una parte de la política de alto nivel funcionando y es independiente de la base de datos, puede intentar conectarla a varias bases de datos diferentes para verificar la aplicabilidad y el rendimiento. Lo mismo ocurre con los sistemas web, los marcos web o incluso la propia web.

Cuento más tiempo deje las opciones abiertas, más experimentos podrá realizar, más cosas podrá probar y más información tendrá cuando llegue al punto en el que esas decisiones ya no puedan posponerse.

¿Qué pasa si las decisiones ya las ha tomado otra persona? ¿Qué pasa si su empresa se ha comprometido con una determinada base de datos, un determinado servidor web o un determinado marco? Un buen arquitecto finge que no se ha tomado la decisión y configura el sistema de manera que esas decisiones aún puedan aplazarse o cambiarse durante el mayor tiempo posible.

Un buen arquitecto maximiza el número de decisiones no tomadas.

INDEPENDENCIA DEL DISPOSITIVO

Como ejemplo de este tipo de pensamiento, retrocedamos a la década de 1960, cuando las computadoras eran adolescentes y la mayoría de los programadores eran matemáticos o ingenieros de otras disciplinas (y un tercio o más eran mujeres).

En aquellos días cometímos muchos errores. Por supuesto, en ese momento no sabíamos que eran errores. ¿Cómo podríamos?

Uno de esos errores fue vincular nuestro código directamente a los dispositivos IO. Si necesitábamos imprimir algo en una impresora, escribíamos código que usaba las instrucciones IO que controlarían la impresora. Nuestro código dependía del dispositivo.

Por ejemplo, cuando escribí programas PDP-8 que se imprimían en la teleimpresora, utilicé un conjunto de instrucciones de máquina que se parecía a esto:

[Haga clic aquí para ver la imagen del código](#)

```
PRTCHR, 0  
TSF  
JMP .-1  
TLS  
JMP I PRTCDH
```

PRTCHR es una subrutina que imprime un carácter en el teleimpresor. El cero inicial se utilizó como almacenamiento para la dirección del remitente. (No pregunte). La instrucción TSF omitió la siguiente instrucción si el teleimpresor estaba listo para imprimir un carácter. Si el teleimpresor estaba ocupado, entonces TSF simplemente pasó al JMP .-1

instrucción, que acaba de regresar a la instrucción TSF . Si el teleimpresor estaba listo, entonces TSF pasaría a la instrucción TLS , que enviaba el carácter del registro A al teleimpresor. Luego, la instrucción JMP I PRTCHR regresó a la persona que llama.

Al principio esta estrategia funcionó bien. Si necesitábamos leer tarjetas del lector de tarjetas, usábamos un código que se comunicaba directamente con el lector de tarjetas. Si necesitábamos perforar tarjetas, escribíamos código que manipulaba directamente la perforación. Los programas funcionaron perfectamente. ¿Cómo podríamos saber que esto fue un error?

Pero es difícil gestionar grandes lotes de tarjetas perforadas. Se pueden perder, mutilar, girar, barajar o dejar caer. Se pueden perder tarjetas individuales y se pueden insertar tarjetas adicionales. Entonces la integridad de los datos se convirtió en un problema importante.

La cinta magnética fue la solución. Podríamos mover las imágenes de las tarjetas a la cinta. Si se te cae una cinta magnética, los discos no se mezclan. No se puede perder accidentalmente un registro ni insertar un registro en blanco simplemente entregándole la cinta. La cinta es mucho más segura. También es más rápido de leer y escribir, y es muy sencillo realizar copias de seguridad.

Desafortunadamente, todo nuestro software fue escrito para manipular lectores de tarjetas y perforadoras de tarjetas. Esos programas tuvieron que reescribirse para utilizar cinta magnética. Ese fue un gran trabajo.

A finales de la década de 1960, habíamos aprendido la lección e inventamos la independencia de los dispositivos. Los sistemas operativos de la época abstraían los dispositivos IO en funciones de software que manejaban registros unitarios que parecían tarjetas. Los programas invocarían servicios del sistema operativo que se ocupaban de dispositivos abstractos de registro de unidades. Los operadores podrían decirle al sistema operativo si esos servicios abstractos deberían conectarse a lectores de tarjetas, cintas magnéticas o cualquier otro dispositivo de registro de unidades.

Ahora el mismo programa podía leer y escribir tarjetas, o leer y escribir cintas, sin ningún cambio. El principio abierto-cerrado nació (pero aún no tiene nombre).

CORREO BASURA

A finales de la década de 1960, trabajaba para una empresa que imprimía correo basura para clientes. El

los clientes nos enviaban cintas magnéticas con registros de unidades que contenían los nombres y direcciones de sus clientes, y escribíamos programas que imprimían bonitos anuncios personalizados.

Ya sabes el tipo:

Hola Sr. Martín,

¡Felicitaciones!

Te elegimos a TI entre todos los que viven en Witchwood Lane para participar en nuestra nueva y fantástica oferta única...

Los clientes nos enviaban enormes rollos de cartas modelo con todas las palabras excepto el nombre y la dirección, y cualquier otro elemento que quisieran que imprimiésemos.

Escribímos programas que extraían los nombres, direcciones y otros elementos de la cinta magnética e imprimímos esos elementos exactamente donde debían aparecer en los formularios.

Estos rollos de cartas modelo pesaban 500 libras y contenían miles de cartas. Los clientes nos enviarían cientos de estos rollos. Imprimíramos cada uno individualmente.

Al principio, teníamos una IBM 360 imprimiendo en su única impresora de línea. Podríamos imprimir unos pocos miles de letras por turno. Desafortunadamente, esto inmovilizó una máquina muy costosa durante mucho tiempo. En aquellos días, los IBM 360 se alquilaban por decenas de miles de dólares al mes.

Entonces le dijimos al sistema operativo que usara cinta magnética en lugar de la impresora de líneas. A nuestros programas no les importaba, porque habían sido escritos para utilizar las abstracciones IO del sistema operativo.

La 360 podría producir una cinta completa en aproximadamente 10 minutos, suficiente para imprimir varios rollos de cartas modelo. Las cintas se sacaron de la sala de ordenadores y se montaron en unidades de cinta conectadas a impresoras fuera de línea. Teníamos cinco y usábamos esas cinco impresoras las 24 horas del día, los siete días de la semana, imprimiendo cientos de miles de correo basura cada semana.

¡El valor de la independencia del dispositivo fue enorme! Podríamos escribir nuestros programas sin saber ni preocuparnos qué dispositivo se utilizaría. Podríamos probar esos programas usando la impresora de línea local conectada a la computadora. Entonces podríamos

dígale al sistema operativo que “imprima” en cinta magnética y ejecute cientos de miles de formularios.

Nuestros programas tenían una forma. Esa forma desconectó la política de los detalles. La política fue el formato de los registros de nombre y dirección. El detalle fue el dispositivo. Aplazamos la decisión sobre qué dispositivo usaríamos.

DIRECCIONAMIENTO FÍSICO

A principios de la década de 1970, trabajé en un gran sistema de contabilidad para un sindicato de camioneros local. Teníamos una unidad de disco de 25 MB en la que almacenábamos registros de agentes, empleadores y miembros. Los diferentes registros tenían diferentes tamaños, por lo que formateamos los primeros cilindros del disco para que cada sector tuviera el tamaño de un registro de Agente . Los siguientes cilindros fueron formateados para tener sectores que coincidieran con los registros del Empleador . Los últimos cilindros fueron formateados para ajustarse a los registros de miembros .

Escribimos nuestro software para conocer la estructura detallada del disco. Sabía que el disco tenía 200 cilindros y 10 cabezas, y que cada cilindro tenía varias docenas de sectores por cabeza. Sabía qué cilindros contenían a los Agentes, Empleadores y Miembros. Todo esto estaba integrado en el código.

Mantuvimos un índice en el disco que nos permitió buscar a cada uno de los Agentes, Empleadores y Miembros. Este índice estaba en otro conjunto de cilindros especialmente formateado en el disco. El índice de Agente estaba compuesto por registros que contenían la identificación de un agente y el número de cilindro, número de cabeza y número de sector de ese registro de Agente . Los empleadores y los miembros tenían índices similares.

Los miembros también se mantuvieron en una lista doblemente enlazada en el disco. Cada registro de Miembro contenía el cilindro, la culata y el número de sector del siguiente registro de Miembro y del registro de Miembro anterior .

¿Qué pasaría si necesitáramos actualizar a una nueva unidad de disco: una con más cabezales, otra con más cilindros o una con más sectores por cilindro?

Tuvimos que escribir un programa especial para leer los datos antiguos del disco antiguo y luego escribirlos en el disco nuevo, traduciendo todos los números de cilindro/culata/sector. También tuvimos que cambiar todo el cableado de nuestro código, ¡y ese cableado estaba en todas partes! Todas las reglas de negocio conocían en detalle el esquema cilindro/culata/sector.

Un día un programador con más experiencia se unió a nuestras filas. Cuando vio lo que habíamos hecho, la sangre se le fue de la cara y nos miró horrorizado, como si fuéramos extraterrestres de algún tipo. Luego nos aconsejó amablemente que cambiáramos nuestro esquema de direccionamiento para utilizar direcciones relativas.

Nuestro colega más sabio sugirió que consideráramos el disco como una enorme matriz lineal de sectores, cada uno de los cuales es direccionable por un número entero secuencial. Luego podríamos escribir una pequeña rutina de conversión que conociera la estructura física del disco y pudiera traducir la dirección relativa a un número de cilindro/culata/sector sobre la marcha.

Afortunadamente para nosotros, seguimos su consejo. Cambiamos la política de alto nivel del sistema para que sea independiente de la estructura física del disco. Eso nos permitió desacoplar la decisión sobre la estructura de la unidad de disco de la aplicación.

CONCLUSIÓN

Las dos historias de este capítulo son ejemplos, en pequeño, de un principio que los arquitectos emplean en lo grande. Los buenos arquitectos separan cuidadosamente los detalles de la política y luego desacoplan la política de los detalles tan completamente que la política no tiene conocimiento de los detalles y no depende de los detalles de ninguna manera. Los buenos arquitectos diseñan la política de manera que las decisiones sobre los detalles puedan retrasarse y aplazarse el mayor tiempo posible.

dieciséis

INDEPENDENCIA



Como dijimos anteriormente, una buena arquitectura debe soportar:

- Los casos de uso y funcionamiento del sistema.
- El mantenimiento del sistema. • El desarrollo del sistema. • El despliegue del sistema.

CASOS DE USO

El primer punto (casos de uso) significa que la arquitectura del sistema debe

apoyar la intención del sistema. Si el sistema es una aplicación de carrito de compras, entonces la arquitectura debe admitir casos de uso de carrito de compras. De hecho, ésta es la primera preocupación del arquitecto y la primera prioridad de la arquitectura. La arquitectura debe soportar los casos de uso.

Sin embargo, como comentamos anteriormente, la arquitectura no ejerce mucha influencia sobre el comportamiento del sistema. Hay muy pocas opciones de comportamiento que la arquitectura pueda dejar abiertas. Pero la influencia no lo es todo. Lo más importante que puede hacer una buena arquitectura para respaldar el comportamiento es aclarar y exponer ese comportamiento para que la intención del sistema sea visible a nivel arquitectónico.

Una aplicación de carrito de compras con una buena arquitectura se verá como una aplicación de carrito de compras. Los casos de uso de ese sistema serán claramente visibles dentro de la estructura de ese sistema. Los desarrolladores no tendrán que buscar comportamientos, porque esos comportamientos serán elementos de primera clase visibles en el nivel superior del sistema. Esos elementos serán clases, funciones o módulos que ocupan posiciones destacadas dentro de la arquitectura y tendrán nombres que describan claramente su función.

[El capítulo 21](#), “Arquitectura que grita”, aclarará mucho más este punto.

OPERACIÓN

La arquitectura juega un papel más sustancial y menos cosmético en el apoyo al funcionamiento del sistema. Si el sistema debe manejar 100.000 clientes por segundo, la arquitectura debe soportar ese tipo de rendimiento y tiempo de respuesta para cada caso de uso que lo exija. Si el sistema debe consultar grandes cubos de datos en milisegundos, entonces la arquitectura debe estructurarse para permitir este tipo de operación.

Para algunos sistemas, esto significará organizar los elementos de procesamiento del sistema en una serie de pequeños servicios que se pueden ejecutar en paralelo en muchos servidores diferentes. Para otros sistemas, significará una gran cantidad de pequeños subprocesos livianos que compartirán el espacio de direcciones de un solo proceso dentro de un solo procesador. Otros sistemas necesitarán sólo unos pocos procesos ejecutándose en espacios de direcciones aislados. Y algunos sistemas pueden incluso sobrevivir como simples programas monolíticos que se ejecutan en un solo proceso.

Por extraño que parezca, esta decisión es una de las opciones que un buen arquitecto deja abierta. Un sistema que está escrito como un monolito y que depende de esa estructura monolítica no puede actualizarse fácilmente a múltiples procesos, múltiples subprocessos o microservicios en caso de que surja la necesidad. En comparación, una arquitectura que mantiene el aislamiento adecuado de sus componentes y no asume los medios de comunicación entre esos componentes, será mucho más fácil de realizar la transición a través del espectro de subprocessos, procesos y servicios a medida que cambian las necesidades operativas del sistema. con el tiempo.

DESARROLLO

La arquitectura juega un papel importante en el soporte del entorno de desarrollo. Aquí es donde entra en juego la ley de Conway. La ley de Conway dice:

Cualquier organización que diseñe un sistema producirá un diseño cuya estructura es una copia de la estructura de comunicación de la organización.

Un sistema que debe ser desarrollado por una organización con muchos equipos y muchas preocupaciones debe tener una arquitectura que facilite acciones independientes por parte de esos equipos, de modo que los equipos no interfieran entre sí durante el desarrollo. Esto se logra dividiendo adecuadamente el sistema en componentes bien aislados y desarrollables de forma independiente. Luego, esos componentes se pueden asignar a equipos que puedan trabajar de forma independiente unos de otros.

DESPLIEGUE

La arquitectura también juega un papel muy importante a la hora de determinar la facilidad con la que se implementa el sistema. El objetivo es el “despliegue inmediato”. Una buena arquitectura no depende de docenas de pequeños scripts de configuración y ajustes en los archivos de propiedades. No requiere la creación manual de directorios o archivos que deban organizarse así. Una buena arquitectura ayuda a que el sistema se pueda implementar inmediatamente después de su construcción.

Nuevamente, esto se logra mediante la partición y el aislamiento adecuados de los componentes del sistema, incluidos los componentes maestros que unen todo el sistema y garantizan que cada componente se inicie, integre y supervise adecuadamente.

DEJANDO OPCIONES ABIERTAS

Una buena arquitectura equilibra todas estas preocupaciones con una estructura de componentes que las satisfaga a todas mutuamente. Suena fácil, ¿verdad? Bueno, es fácil para mí escribir eso.

La realidad es que lograr este equilibrio es bastante difícil. El problema es que la mayoría de las veces no sabemos cuáles son todos los casos de uso, ni conocemos las limitaciones operativas, la estructura del equipo o los requisitos de implementación.

Peor aún, incluso si los conociéramos, inevitablemente cambiarán a medida que el sistema avance en su ciclo de vida. En definitiva, los objetivos que debemos alcanzar son confusos e inconstantes. Bienvenido al mundo real.

Pero no todo está perdido: algunos principios de la arquitectura son relativamente económicos de implementar y pueden ayudar a equilibrar esas preocupaciones, incluso cuando no se tiene una idea clara de los objetivos que se deben alcanzar. Esos principios nos ayudan a dividir nuestros sistemas en componentes bien aislados que nos permiten dejar tantas opciones abiertas como sea posible, durante el mayor tiempo posible.

Una buena arquitectura hace que el sistema sea fácil de cambiar, en todas las formas en que debe cambiar, dejando opciones abiertas.

CAPAS DE DESACOPLAMIENTO

Considere los casos de uso. El arquitecto quiere que la estructura del sistema admita todos los casos de uso necesarios, pero no sabe cuáles son todos esos casos de uso.

Sin embargo, el arquitecto sí conoce la intención básica del sistema. Es un sistema de carrito de compras, o un sistema de lista de materiales, o un sistema de procesamiento de pedidos.

De modo que el arquitecto puede emplear el principio de responsabilidad única y el principio de cierre común para separar aquellas cosas que cambian por diferentes razones y recopilar aquellas cosas que cambian por las mismas razones, dado el contexto de la intención del sistema.

¿Qué cambia por diferentes motivos? Hay algunas cosas obvias. Las interfaces de usuario cambian por motivos que no tienen nada que ver con las reglas comerciales. Los casos de uso tienen elementos de ambos. Claramente, entonces, un buen arquitecto querrá separar las partes de la interfaz de usuario de un caso de uso de las partes de las reglas de negocio de tal manera que se puedan cambiar independientemente una de otra, manteniendo esos casos de uso.

visible y claro.

Las propias reglas comerciales pueden estar estrechamente vinculadas a la aplicación o pueden ser más generales. Por ejemplo, la validación de campos de entrada es una regla de negocio estrechamente ligada a la propia aplicación. Por el contrario, el cálculo de los intereses de una cuenta y el recuento del inventario son reglas de negocio que están más estrechamente asociadas con el dominio. Estos dos tipos diferentes de reglas cambiarán a ritmos diferentes y por razones diferentes, por lo que deben separarse para que puedan cambiarse de forma independiente.

La base de datos, el lenguaje de consulta e incluso el esquema son detalles técnicos que no tienen nada que ver con las reglas comerciales o la interfaz de usuario. Cambiarán a ritmos y por razones independientes de otros aspectos del sistema.

En consecuencia, la arquitectura debe separarlos del resto del sistema para que puedan modificarse de forma independiente.

Así, encontramos el sistema dividido en capas horizontales desacopladas: la interfaz de usuario, las reglas comerciales específicas de la aplicación, las reglas comerciales independientes de la aplicación y la base de datos, solo por mencionar algunas.

CASOS DE USO DE DESacoplamiento

¿Qué más cambia por diferentes motivos? ¡Los casos de uso en sí! Es casi seguro que el caso de uso para agregar un pedido a un sistema de ingreso de pedidos cambiará a un ritmo diferente y por razones diferentes que el caso de uso que elimina un pedido del sistema. Los casos de uso son una forma muy natural de dividir el sistema.

Al mismo tiempo, los casos de uso son cortes verticales estrechos que atraviesan las capas horizontales del sistema. Cada caso de uso utiliza alguna interfaz de usuario, algunas reglas comerciales específicas de la aplicación, algunas reglas comerciales independientes de la aplicación y algunas funciones de base de datos. Por lo tanto, a medida que dividimos el sistema en capas horizontales, también lo dividimos en casos de uso verticales delgados que atraviesan esas capas.

Para lograr este desacoplamiento, separamos la interfaz de usuario del caso de uso de orden de agregar de la interfaz de usuario del caso de uso de orden de eliminación. Hacemos lo mismo con las reglas de negocio y con la base de datos. Mantenemos los casos de uso separados a lo largo de la altura vertical del sistema.

Puedes ver el patrón aquí. Si desacopla los elementos del sistema que cambian por diferentes motivos, podrá continuar agregando nuevos casos de uso sin interferir con los antiguos. Si también agrupa la interfaz de usuario y la base de datos para admitir esos casos de uso, de modo que cada caso de uso utilice un aspecto diferente de la interfaz de usuario y la base de datos, es poco probable que agregar nuevos casos de uso afecte a los más antiguos.

MODO DE DESACOPLAMIENTO

Ahora piense en lo que significa todo ese desacoplamiento para el segundo punto: las operaciones. Si se separan los diferentes aspectos de los casos de uso, entonces aquellos que deben ejecutarse con un alto rendimiento probablemente ya estén separados de aquellos que deben ejecutarse con un rendimiento bajo. Si la interfaz de usuario y la base de datos se han separado de las reglas comerciales, entonces pueden ejecutarse en servidores diferentes. Aquellos que requieren mayor ancho de banda se pueden replicar en muchos servidores.

En resumen, el desacoplamiento que hicimos por el bien de los casos de uso también ayuda con las operaciones. Sin embargo, para aprovechar el beneficio operativo, el desacoplamiento debe tener el modo adecuado. Para ejecutarse en servidores separados, los componentes separados no pueden depender de estar juntos en el mismo espacio de direcciones de un procesador. Deben ser servicios independientes, que se comuniquen a través de una red de algún tipo.

Muchos arquitectos llaman a estos componentes “servicios” o “microservicios”, dependiendo de una vaga noción de número de líneas. De hecho, una arquitectura basada en servicios suele denominarse arquitectura orientada a servicios.

Si esa nomenclatura hizo sonar algunas alarmas en tu mente, no te preocupes. No les voy a decir que SoA sea la mejor arquitectura posible, o que los microservicios sean la ola del futuro. El punto que se señala aquí es que a veces tenemos que separar nuestros componentes hasta el nivel de servicio.

Recuerde, una buena arquitectura deja opciones abiertas. El modo de desacoplamiento es una de esas opciones.

Antes de explorar más ese tema, veamos los otros dos puntos.

CAPACIDAD DE DESARROLLO INDEPENDIENTE

El tercer punto fue el desarrollo. Claramente, cuando los componentes están fuertemente desacoplados, se mitiga la interferencia entre equipos. Si las reglas comerciales no conocen la interfaz de usuario, entonces un equipo que se centra en la interfaz de usuario no puede afectar mucho a un equipo que se centra en las reglas comerciales. Si los casos de uso en sí están desacoplados entre sí, entonces no es probable que un equipo que se centra en el caso de uso addOrder interfiera con un equipo que se centra en el caso de uso eliminarOrder .

Mientras las capas y los casos de uso estén desacoplados, la arquitectura del sistema respaldará la organización de los equipos, independientemente de si están organizados como equipos de funciones, equipos de componentes, equipos de capas o alguna otra variación.

IMPLEMENTACIÓN INDEPENDIENTE

El desacoplamiento de los casos de uso y las capas también ofrece un alto grado de flexibilidad en la implementación. De hecho, si el desacoplamiento se realiza bien, entonces debería ser posible intercambiar capas y casos de uso en caliente en sistemas en ejecución. Agregar un nuevo caso de uso podría ser tan simple como agregar algunos archivos jar o servicios nuevos al sistema y dejar el resto en paz.

DUPLICACIÓN

Los arquitectos a menudo caen en una trampa, una trampa que depende de su miedo a la duplicación.

La duplicación es generalmente algo malo en el software. No nos gusta el código duplicado. Cuando el código está verdaderamente duplicado, estamos obligados por honor como profesionales a reducirlo y eliminarlo.

Pero existen diferentes tipos de duplicación. Existe una verdadera duplicación, en la que cada cambio en una instancia requiere el mismo cambio en cada duplicado de esa instancia. Entonces hay una duplicación falsa o accidental. Si dos secciones de código aparentemente duplicadas evolucionan a lo largo de caminos diferentes (si cambian a diferentes ritmos y por diferentes razones), entonces no son verdaderos duplicados.

Vuelva a consultarlos dentro de unos años y descubrirá que son muy diferentes entre sí.

Ahora imagine dos casos de uso que tengan estructuras de pantalla muy similares. El

Es probable que los arquitectos se sientan fuertemente tentados a compartir el código de esa estructura. ¿Pero deberían hacerlo? ¿Es esa una verdadera duplicación? ¿O es accidental?

Lo más probable es que sea accidental. A medida que pasa el tiempo, lo más probable es que esas dos pantallas diverjan y eventualmente se vean muy diferentes. Por este motivo hay que tener cuidado de no unificarlos. De lo contrario, separarlos más adelante será un desafío.

Cuando separa verticalmente los casos de uso entre sí, se encontrará con este problema y la tentación será acoplar los casos de uso porque tienen estructuras de pantalla similares, algoritmos similares o consultas y/o esquemas de bases de datos similares. Ten cuidado. Resista la tentación de cometer el pecado de eliminar instintivamente la duplicación. Asegúrese de que la duplicación sea real.

Del mismo modo, cuando separa capas horizontalmente, puede notar que la estructura de datos de un registro de base de datos en particular es muy similar a la estructura de datos de una vista de pantalla en particular. Es posible que tenga la tentación de simplemente pasar el registro de la base de datos a la interfaz de usuario, en lugar de crear un modelo de vista que tenga el mismo aspecto y copiar los elementos.

Tenga cuidado: es casi seguro que esta duplicación es accidental. Crear el modelo de vista independiente no requiere mucho esfuerzo y le ayudará a mantener las capas correctamente desacopladas.

MODOS DE DESACOPLAMIENTO (OTRA VEZ)

Volver a los modos. Hay muchas formas de desacoplar capas y casos de uso. Se pueden desacoplar a nivel de código fuente, a nivel de código binario (implementación) y a nivel de unidad de ejecución (servicio).

- Nivel de fuente. Podemos controlar las dependencias entre los módulos de código fuente para que los cambios en un módulo no fueren cambios o recompilación de otros (por ejemplo, Ruby Gems).

En este modo de desacoplamiento, todos los componentes se ejecutan en el mismo espacio de direcciones y se comunican entre sí mediante llamadas a funciones simples. Hay un único ejecutable cargado en la memoria de la computadora. La gente suele llamar a esto monolítico. estructura.

- Nivel de implementación. Podemos controlar las dependencias entre unidades implementables, como archivos jar, DLL o bibliotecas compartidas, de modo que los cambios en el código fuente de un módulo no fueren la reconstrucción y reimplementación de otros.

Es posible que muchos de los componentes aún vivan en el mismo espacio de direcciones y se comuniquen a través de llamadas a funciones. Otros componentes pueden vivir en otros procesos en el mismo procesador y comunicarse a través de comunicaciones entre procesos, sockets o memoria compartida. Lo importante aquí es que los componentes desacoplados se partitionan en unidades desplegables de forma independiente, como archivos jar, archivos Gem o DLL.

- Nivel de servicio. Podemos reducir las dependencias hasta el nivel de las estructuras de datos y comunicarnos únicamente a través de paquetes de red, de modo que cada unidad de ejecución sea completamente independiente de los cambios fuente y binarios de otras (por ejemplo, servicios o microservicios).

¿Cuál es el mejor modo para usar?

La respuesta es que es difícil saber qué modo es mejor durante las primeras fases de un proyecto. De hecho, a medida que el proyecto madure, el modo óptimo puede cambiar.

Por ejemplo, no es difícil imaginar que un sistema que actualmente se ejecuta cómodamente en un servidor pueda crecer hasta el punto en que algunos de sus componentes deban ejecutarse en servidores separados. Mientras el sistema se ejecuta en un único servidor, el desacoplamiento a nivel de fuente puede ser suficiente. Sin embargo, más adelante podría ser necesario desacoplarlo en unidades desplegables, o incluso servicios.

Una solución (que parece ser popular en este momento) es simplemente desacoplar el nivel de servicio de forma predeterminada. Un problema con este enfoque es que es caro y fomenta un desacoplamiento generalizado. No importa cuán “micro” sean los microservicios, es probable que el desacoplamiento no sea lo suficientemente detallado.

Otro problema con el desacoplamiento de niveles de servicio es que es costoso, tanto en tiempo de desarrollo como en recursos del sistema. Lidiar con los límites de los servicios cuando no son necesarios es un desperdicio de esfuerzo, memoria y ciclos. Y sí, sé que los dos últimos son baratos, pero el primero no.

Mi preferencia es impulsar el desacoplamiento hasta el punto en que se pueda formar un servicio, si fuera necesario; pero luego dejar los componentes en el mismo espacio de direcciones el mayor tiempo posible. Esto deja abierta la opción de un servicio.

Con este enfoque, inicialmente los componentes se separan a nivel del código fuente. Esto puede ser suficiente durante toda la vida del proyecto. Sin embargo, si surgen problemas de implementación o desarrollo, impulsar algunas de las

el desacoplamiento a un nivel de implementación puede ser suficiente, al menos por un tiempo.

A medida que aumentan los problemas operativos, de implementación y de desarrollo, elijo cuidadosamente qué unidades desplegables convertir en servicios y cambio gradualmente el sistema en esa dirección.

Con el tiempo, las necesidades operativas del sistema pueden disminuir. Lo que antes requería un desacoplamiento a nivel de servicio ahora puede requerir solo un desacoplamiento a nivel de implementación o incluso a nivel de fuente.

Una buena arquitectura permitirá que un sistema nazca como un monolito, implementado en un solo archivo, pero luego crezca hasta convertirse en un conjunto de unidades implementables de forma independiente y luego hasta convertirse en servicios y/o microservicios independientes. Más adelante, a medida que las cosas cambien, debería permitir revertir esa progresión y deslizarse hacia abajo hasta convertirse en un monolito.

Una buena arquitectura protege la mayor parte del código fuente de esos cambios. Deja el modo de desacoplamiento abierto como una opción para que las implementaciones grandes puedan usar un modo, mientras que las implementaciones pequeñas pueden usar otro.

CONCLUSIÓN

Sí, esto es complicado. Y no estoy diciendo que el cambio de modos de desacoplamiento deba ser una opción de configuración trivial (aunque a veces eso es apropiado).

Lo que estoy diciendo es que el modo de desacoplamiento de un sistema es una de esas cosas que probablemente cambie con el tiempo, y un buen arquitecto prevé y facilita adecuadamente esos cambios.

17

LÍMITES: TRAZANDO LÍNEAS



La arquitectura de software es el arte de trazar líneas que yo llamo límites. Esos límites separan los elementos de software entre sí y restringen a los de un lado el conocimiento de los del otro. Algunas de esas líneas se trazan muy temprano en la vida de un proyecto, incluso antes de que se escriba cualquier código. Otros se dibujan mucho más tarde. Aquellos que se extraen temprano lo hacen con el propósito de aplazar las decisiones el mayor tiempo posible y evitar que esas decisiones contaminen la lógica empresarial central.

Recuerde que el objetivo de un arquitecto es minimizar los recursos humanos necesarios para construir y mantener el sistema requerido. ¿Qué es lo que debilita este tipo de poder popular? Acoplamiento, y especialmente acoplamiento a decisiones prematuras.

¿Qué tipo de decisiones son prematuras? Decisiones que no tienen nada que ver con los requisitos comerciales (los casos de uso) del sistema. Estas incluyen decisiones sobre marcos, bases de datos, servidores web, bibliotecas de utilidades, inyección de dependencias y similares. Una buena arquitectura de sistema es aquella en la que decisiones como estas se vuelven auxiliares y aplazables. Una buena arquitectura de sistema no depende de esas decisiones. Una buena arquitectura del sistema permite que esas decisiones se tomen en el último momento posible, sin un impacto significativo.

UN PAR DE HISTORIAS TRISTE

Aquí está la triste historia de la empresa P, que sirve de advertencia sobre la toma de decisiones prematuras. En la década de 1980, los fundadores de P escribieron una aplicación de escritorio monolítica y sencilla. Disfrutaron de un gran éxito y desarrollaron el producto durante la década de 1990 hasta convertirlo en una aplicación GUI de escritorio popular y exitosa.

Pero entonces, a finales de los años 1990, la red surgió como una fuerza. De repente, todo el mundo necesitaba tener una solución web y P no fue la excepción. Los clientes de P clamaban por una versión del producto en la web. Para satisfacer esta demanda, la empresa contrató a un grupo de programadores Java de veintitantos años y se embarcó en un proyecto para publicar su producto en la web.

Los chicos de Java soñaban con granjas de servidores bailando en sus cabezas, por lo que adoptaron una rica “arquitectura” de granjas [1](#) que podrían distribuir a través de tales de tres niveles. Habría servidores para la GUI, servidores para el middleware y servidores para la base de datos. Por supuesto.

Los programadores decidieron, desde el principio, que todos los objetos del dominio tendrían tres instancias: una en el nivel de GUI, otra en el nivel de middleware y otra en el nivel de base de datos. Dado que estas instancias vivían en diferentes máquinas, se creó un rico sistema de comunicaciones entre procesadores y entre niveles. Las invocaciones de métodos entre niveles se convirtieron en objetos, se serializaron y se ordenaron a través del cable.

Ahora imagine lo que se necesitó para implementar una característica simple como agregar un nuevo campo a un registro existente. Ese campo tuvo que agregarse a las clases en los tres niveles y a varios de los mensajes entre niveles. Dado que los datos viajaban en ambas direcciones, fue necesario diseñar cuatro protocolos de mensajes. Cada protocolo tenía un lado de envío y un lado de recepción, por lo que se requerían ocho controladores de protocolo. Tres ejecutables tenían

por construir, cada uno con tres objetos comerciales actualizados, cuatro mensajes nuevos y ocho controladores nuevos.

Y piense en lo que tuvieron que hacer esos ejecutables para implementar las funciones más simples. Piense en todas las instancias de objetos, todas las serializaciones, toda la clasificación y desclasificación, toda la creación y análisis de mensajes, todas las comunicaciones de socket, administradores de tiempo de espera, escenarios de reintento y todas las demás cosas adicionales que tiene que hacer. para hacer una cosa simple.

Por supuesto, durante el desarrollo los programadores no tenían una granja de servidores. De hecho, simplemente ejecutaron los tres ejecutables en tres procesos diferentes en una sola máquina. Se desarrollaron de esta manera durante varios años. Pero estaban convencidos de que su arquitectura era la correcta. Y así, aunque se estaban ejecutando en una sola máquina, continuaron con todas las instancias de objetos, todas las serializaciones, toda la clasificación y desclasificación, toda la creación y análisis de mensajes, todas las comunicaciones de socket y todo el material adicional. en una sola máquina.

La ironía es que la empresa P nunca vendió un sistema que requiriera una granja de servidores. Cada sistema que implementaron fue un único servidor. Y en ese único servidor, los tres ejecutables continuaron con todas las instancias de objetos, todas las serializaciones, toda la clasificación y desclasificación, toda la creación y análisis de mensajes, todas las comunicaciones de socket y todo el material adicional, en anticipación de un servidor. granja que nunca existió y que nunca existiría.

La tragedia es que los arquitectos, al tomar una decisión prematura, multiplicaron enormemente el esfuerzo de desarrollo.

La historia de P no es aislada. Lo he visto muchas veces y en muchos lugares.

De hecho, P es una superposición de todos esos lugares.

Pero hay destinos peores que los de P.

Considere W, una empresa local que gestiona flotas de vehículos de empresa. Recientemente contrataron a un “arquitecto” para controlar su variado esfuerzo de software. Y déjame decirte que control era el segundo nombre de este tipo. Rápidamente se dio cuenta de que lo que esta pequeña operación necesitaba era una “ARQUITECTURA” completa, a escala empresarial y orientada a servicios . Creó un enorme modelo de dominio de todos los diferentes "objetos" del negocio, diseñó un conjunto de servicios para administrar estos objetos de dominio y puso a todos los desarrolladores en el camino al infierno. Como ejemplo sencillo,

Supongamos que desea agregar el nombre, la dirección y el número de teléfono de una persona de contacto a un registro de ventas. Tenías que ir al ServiceRegistry y solicitar el ID del servicio ContactService. Luego tenías que enviar un mensaje CreateContact al ContactService. Por supuesto, este mensaje tenía docenas de campos y todos debían contener datos válidos, datos a los que el programador no tenía acceso, ya que todo lo que tenía era un nombre, una dirección y un número de teléfono.

Después de falsificar los datos, el programador tuvo que introducir la identificación del contacto recién creado en el registro de ventas y enviar el mensaje UpdateContact al SaleRecordService.

Por supuesto, para probar cualquier cosa, había que activar todos los servicios necesarios, uno por uno, y activar el bus de mensajes y el servidor BPEL, y fueron los retrasos ... Y luego, allí en la propagación cuando estos mensajes rebocaban de un servicio a otro y esperaban en cola. después de la cola.

Y luego, si desea agregar una nueva característica, bueno, puede imaginar el acoplamiento entre todos esos servicios y el gran volumen de WSDL que necesitaban cambios, y todas las reimplementaciones que esos cambios requirieron. ...

El infierno empieza a parecer un lugar agradable en comparación.

No hay nada intrínsecamente malo en un sistema de software estructurado en torno a servicios. El error en W fue la adopción y aplicación prematuras de un conjunto de herramientas que prometían SoA, es decir, la adopción prematura de un conjunto masivo de servicios de objetos de dominio. El costo de esos errores fueron puras horas-persona (horas-persona en masa) arrojadas al vórtice de SoA.

Podría seguir describiendo un fallo arquitectónico tras otro. Pero hablemos más bien de un éxito arquitectónico.

APTITUD

Mi hijo Micah y yo comenzamos a trabajar en FitNesse en 2001. La idea era crear un wiki simple que incluyera la herramienta FIT de Ward Cunningham para escribir aceptación. pruebas.

Esto fue en los días anteriores a que Maven "resolviera" el problema del archivo jar. Me mantuve firme en que cualquier cosa que produjéramos no debería requerir que la gente descargara más

más de un archivo jar. Llamé a esta regla "Descargar y listo". Esta regla impulsó muchas de nuestras decisiones.

Una de las primeras decisiones fue escribir nuestro propio servidor web, específico para las necesidades de FitNesse. Esto puede parecer absurdo. Incluso en 2001 había muchos servidores web de código abierto que podríamos haber utilizado. Sin embargo, escribir el nuestro resultó ser una muy buena decisión porque un servidor web básico es un software muy sencillo de escribir y nos permitió posponer cualquier decisión sobre el marco web [2](#) hasta mucho más tarde.

Otra decisión temprana fue evitar pensar en una base de datos. Teníamos MySQL en el fondo de nuestras mentes, pero retrasamos intencionalmente esa decisión empleando un diseño que hizo que la decisión fuera irrelevante. Ese diseño consistía simplemente en establecer una interfaz entre todos los accesos a los datos y el propio depósito de datos.

Colocamos los métodos de acceso a datos en una interfaz llamada WikiPage. Esos métodos proporcionaron toda la funcionalidad que necesitábamos para buscar, recuperar y guardar páginas. Por supuesto, al principio no implementamos esos métodos; simplemente los eliminamos mientras trabajábamos en funciones que no implicaban buscar y guardar los datos.

De hecho, durante tres meses simplemente trabajamos en traducir texto wiki a HTML. Esto no requirió ningún tipo de almacenamiento de datos, por lo que creamos una clase llamada MockWikiPage que simplemente dejó los métodos de acceso a datos bloqueados.

Con el tiempo, esos resguardos se volvieron insuficientes para las funciones que queríamos escribir. Necesitábamos acceso a datos reales, no talones. Entonces creamos un nuevo derivado de WikiPage llamado InMemoryPage. Este derivado implementó el método de acceso a datos para administrar una tabla hash de páginas wiki, que guardamos en la RAM.

Esto nos permitió escribir artículo tras artículo durante un año completo. De hecho, conseguimos que toda la primera versión del programa FitNesse funcionara de esta manera. Podríamos crear páginas, vincular a otras páginas, realizar todos los formatos sofisticados de wiki e incluso ejecutar pruebas con FIT. Lo que no pudimos hacer fue guardar nada de nuestro trabajo.

Cuando llegó el momento de implementar la persistencia, pensamos nuevamente en MySQL, pero decidimos que no era necesario en el corto plazo, porque sería muy fácil escribir las tablas hash en archivos planos. Entonces implementamos FileSystemWikiPage, que simplemente trasladó la funcionalidad a archivos planos, y luego continuamos desarrollando más funciones.

Tres meses después, llegamos a la conclusión de que la solución para limas planas era suficientemente buena; Decidimos abandonar por completo la idea de MySQL. Aplazamos esa decisión hasta convertirla en inexistente y nunca miramos atrás.

Ese sería el final de la historia si no fuera por uno de nuestros clientes que decidió que necesitaba poner el wiki en MySQL para sus propios fines. Le mostramos la arquitectura de WikiPages que nos había permitido aplazar la decisión. Regresó un día después con todo el sistema funcionando en MySQL.

Simplemente escribió un derivado de MySqlWikiPage y lo puso en funcionamiento.

Solíamos incluir esa opción en un paquete con FitNesse, pero nadie más la usó, así que finalmente la abandonamos. Incluso el cliente que escribió el derivado finalmente lo abandonó.

Al principio del desarrollo de FitNesse, trazamos una línea divisoria entre las reglas comerciales y las bases de datos. Esa línea impidió que las reglas comerciales supieran algo sobre la base de datos, aparte de los métodos simples de acceso a los datos. Esa decisión nos permitió aplazar la elección y la implementación de la base de datos durante más de un año. Nos permitió probar la opción del sistema de archivos y nos permitió cambiar de dirección cuando vimos una solución mejor. Sin embargo, no impidió, ni siquiera impidió, avanzar en la dirección original (MySQL) cuando alguien así lo deseaba.

El hecho de que no tuviéramos una base de datos ejecutándose durante 18 meses de desarrollo significó que, durante 18 meses, no tuvimos problemas de esquema, problemas de consulta, problemas de servidor de base de datos, problemas de contraseña, problemas de tiempo de conexión y todos los demás problemas desagradables que levantan sus feas cabezas cuando inicias una base de datos. También significó que todas nuestras pruebas se ejecutaron rápidamente, porque no había ninguna base de datos que las ralentizara.

En resumen, trazar las líneas divisorias nos ayudó a retrasar y aplazar decisiones y, en última instancia, nos ahorró una enorme cantidad de tiempo y dolores de cabeza. Y eso es lo que debería hacer una buena arquitectura.

¿QUÉ LÍNEAS DIBUJAS Y CUÁNDΟ LAS DIBUJAS?

Trazas líneas entre las cosas que importan y las que no. La GUI no tiene importancia para las reglas comerciales, por lo que debería haber una línea entre ellas. El

La base de datos no le importa a la GUI, por lo que debe haber una línea entre ellos. La base de datos no tiene importancia para las reglas comerciales, por lo que debe haber una línea entre ellas.

Es posible que algunos de ustedes hayan rechazado una o más de esas declaraciones, especialmente la parte sobre las reglas comerciales que no se preocupan por la base de datos. A muchos de nosotros nos han enseñado a creer que la base de datos está inextricablemente conectada a las reglas comerciales. Algunos de nosotros incluso estamos convencidos de que la base de datos es la encarnación de las reglas comerciales.

Pero, como veremos en otro capítulo, esta idea es errónea. La base de datos es una herramienta que las reglas de negocio pueden utilizar indirectamente. Las reglas de negocio no necesitan conocer el esquema, ni el lenguaje de consulta, ni ningún otro detalle sobre la base de datos. Todo lo que las reglas comerciales deben saber es que existe un conjunto de funciones que se pueden usar para recuperar o guardar datos. Esto nos permite poner la base de datos detrás de una interfaz.

Puedes ver esto claramente en la [Figura 17.1](#). Las BusinessRules utilizan DatabaseInterface para cargar y guardar datos. DatabaseAccess implementa la interfaz y dirige el funcionamiento de la base de datos real .

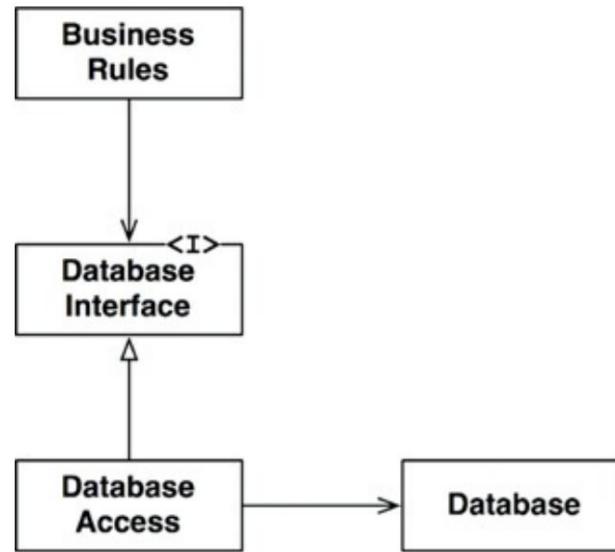


Figura 17.1 La base de datos detrás de una interfaz

Las clases e interfaces en este diagrama son simbólicas. En una aplicación real, habría muchas clases de reglas de negocio, muchas clases de interfaz de base de datos y muchas implementaciones de acceso a bases de datos. Todos ellos, sin embargo, seguirían

aproximadamente el mismo patrón.

¿Dónde está la línea límite? El límite se traza a través de la relación de herencia, justo debajo de la interfaz de la base de datos ([Figura 17.2](#)).

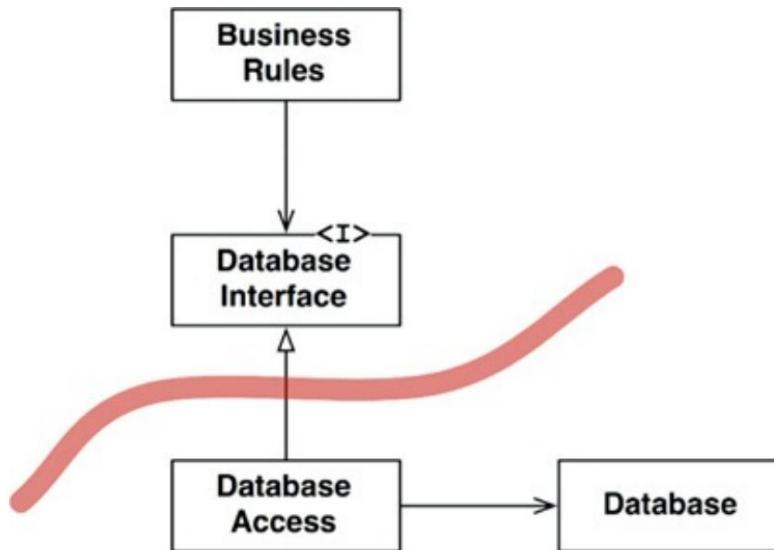


Figura 17.2 La línea límite

Tenga en cuenta las dos flechas que salen de la clase DatabaseAccess . Esas dos flechas apuntan en dirección opuesta a la clase DatabaseAccess . Eso significa que ninguna de estas clases sabe que existe la clase DatabaseAccess .

Ahora retrocedamos un poco. Veremos el componente que contiene muchas reglas de negocio y el componente que contiene la base de datos y todas sus clases de acceso ([Figura 17.3](#)).

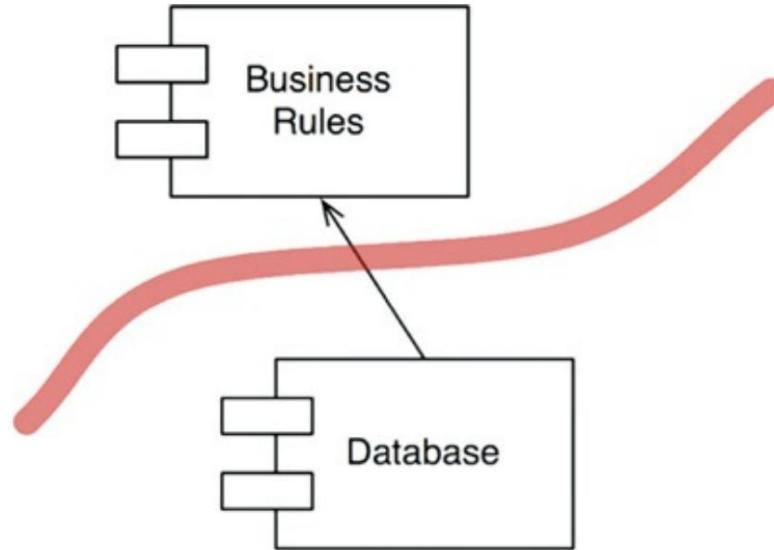


Figura 17.3 Las reglas de negocio y los componentes de la base de datos.

Tenga en cuenta la dirección de la flecha. La Base de Datos conoce las BusinessRules.

Las BusinessRules no conocen la Base de Datos. Esto implica que las clases DatabaseInterface residen en el componente BusinessRules , mientras que las clases DatabaseAccess residen en el componente Database .

La dirección de esta línea es importante. Muestra que la base de datos no le importa a las BusinessRules, pero la base de datos no puede existir sin las BusinessRules.

Si esto le parece extraño, recuerde este punto: el componente Base de datos contiene el código que traduce las llamadas realizadas por BusinessRules al lenguaje de consulta de la base de datos. Es ese código de traducción el que conoce las BusinessRules.

Habiendo dibujado esta línea límite entre los dos componentes y habiendo establecido la dirección de la flecha hacia BusinessRules, ahora podemos ver que BusinessRules podría usar cualquier tipo de base de datos. El componente Base de datos podría reemplazarse con muchas implementaciones diferentes; las BusinessRules no cuidado.

La base de datos podría implementarse con Oracle, MySQL, Couch, Datomic o incluso archivos planos. A las reglas del negocio no les importa en absoluto. Y eso significa que la decisión sobre la base de datos se puede aplazar y usted puede concentrarse en escribir y probar las reglas comerciales antes de tener que tomar la decisión sobre la base de datos.

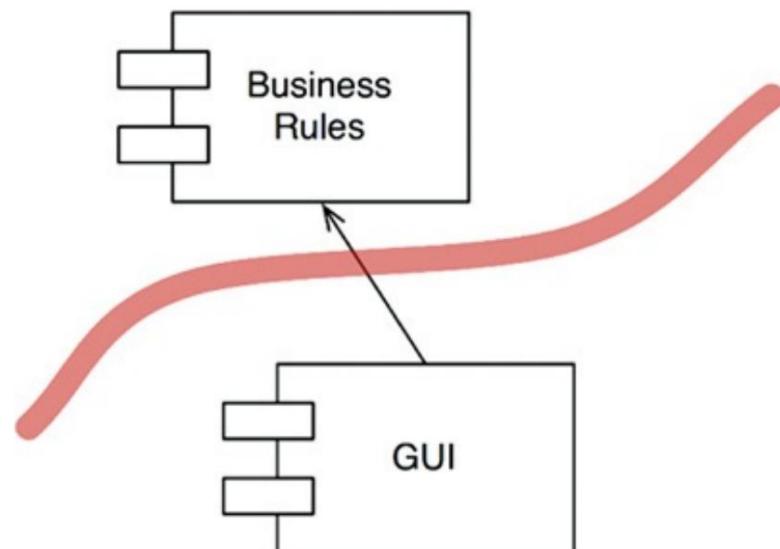
¿Qué pasa con las entradas y salidas?

Los desarrolladores y los clientes a menudo se confunden acerca de qué es el sistema. Ven la GUI y piensan que la GUI es el sistema. Definen un sistema en términos de la GUI, por lo que creen que deberían ver que la GUI comienza a funcionar inmediatamente.

No logran darse cuenta de un principio de importancia crítica: la OI es irrelevante.

Esto puede resultar difícil de entender al principio. A menudo pensamos en el comportamiento del sistema en términos del comportamiento de IO. Consideremos, por ejemplo, un videojuego. Tu experiencia está dominada por la interfaz: la pantalla, el mouse, los botones y los sonidos. Olvidas que detrás de esa interfaz hay un modelo (un conjunto sofisticado de estructuras y funciones de datos) que la impulsa. Más importante aún, ese modelo no necesita la interfaz. Felizmente ejecutaría sus tareas, modelando todos los eventos del juego, sin que el juego se muestre nunca en la pantalla. La interfaz no le importa al modelo: las reglas de negocio.

Y así, una vez más, vemos los componentes GUI y BusinessRules separados por una línea límite ([Figura 17.4](#)). Una vez más, vemos que el componente menos relevante depende del componente más relevante. Las flechas muestran qué componente conoce al otro y, por tanto, qué componente se preocupa por el otro. La GUI se preocupa por las BusinessRules.



[Figura 17.4 El límite entre los componentes GUI y BusinessRules](#)

Habiendo trazado este límite y esta flecha, ahora podemos ver que la GUI podría reemplazarse por cualquier otro tipo de interfaz, y las BusinessRules no.

cuidado.

ARQUITECTURA DE COMPLEMENTOS

En conjunto, estas dos decisiones sobre la base de datos y la GUI crean una especie de patrón para la adición de otros componentes. Ese patrón es el mismo que utilizan los sistemas que permiten complementos de terceros.

De hecho, la historia de la tecnología de desarrollo de software es la historia de cómo crear complementos de manera conveniente para establecer una arquitectura de sistema escalable y mantible. Las reglas de negocio centrales se mantienen separadas e independientes de aquellos componentes que son opcionales o que pueden implementarse de muchas formas diferentes ([Figura 17.5](#)).

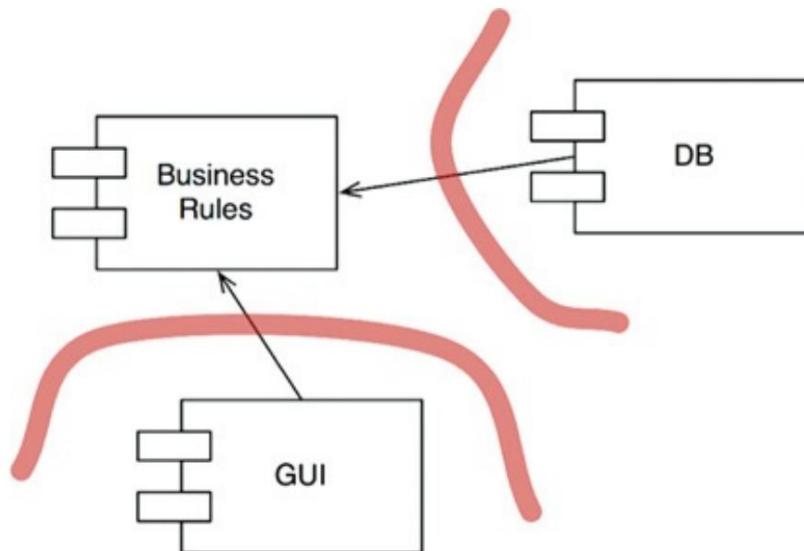


Figura 17.5 Conexión a las reglas de negocio

Debido a que la interfaz de usuario en este diseño se considera un complemento, hemos hecho posible conectar muchos tipos diferentes de interfaces de usuario. Pueden estar basados en web, cliente/servidor, SOA, consola o cualquier otro tipo de tecnología de interfaz de usuario.

Lo mismo ocurre con la base de datos. Dado que hemos elegido tratarlo como un complemento, podemos reemplazarlo con cualquiera de las diversas bases de datos SQL, una base de datos NOSQL, una base de datos basada en un sistema de archivos o cualquier otro tipo de tecnología de base de datos que consideremos necesaria en el futuro. .

Es posible que estos reemplazos no sean triviales. Si la implementación inicial de nuestro sistema se basó en la web, entonces escribir el complemento para una interfaz de usuario cliente-servidor podría ser un desafío. Es probable que sea necesario reelaborar algunas de las comunicaciones entre las reglas comerciales y la nueva interfaz de usuario. Aun así, al comenzar con la presunción de una estructura de complemento, al menos hemos hecho que ese cambio sea práctico.

EL ARGUMENTO DEL COMPLEMENTO

Considere la relación entre ReSharper y Visual Studio. Estos componentes son producidos por equipos de desarrollo completamente diferentes en empresas completamente diferentes. De hecho, JetBrains, el fabricante de ReSharper, vive en Rusia. Microsoft, por supuesto, reside en Redmond, Washington. Es difícil imaginar dos equipos de desarrollo más separados.

¿Qué equipo puede dañar al otro? ¿Qué equipo es inmune al otro? La estructura de dependencia cuenta la historia ([Figura 17.6](#)). El código fuente de ReSharper depende del código fuente de Visual Studio. Por lo tanto, el equipo de ReSharper no puede hacer nada para molestar al equipo de Visual Studio. Pero el equipo de Visual Studio podría desactivar completamente el equipo de ReSharper si así lo deseara.

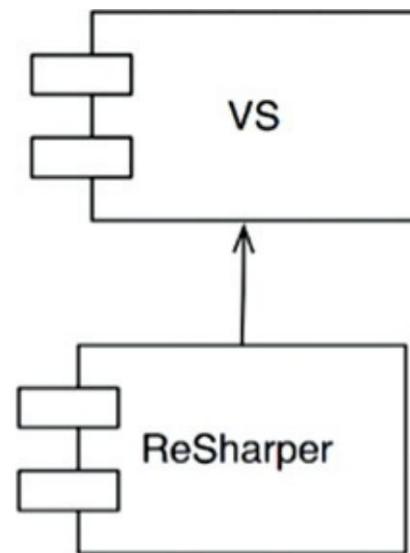


Figura 17.6 ReSharper depende de Visual Studio

Esa es una relación profundamente asimétrica y es una que deseamos tener en nuestros propios sistemas. Queremos que ciertos módulos sean inmunes a otros. Para

Por ejemplo, no queremos que las reglas comerciales se rompan cuando alguien cambia el formato de una página web o cambia el esquema de la base de datos. No queremos que los cambios en una parte del sistema provoquen que otras partes no relacionadas del sistema fallen. No queremos que nuestros sistemas muestren ese tipo de fragilidad.

Organizar nuestros sistemas en una arquitectura de complementos crea firewalls a través de los cuales los cambios no se pueden propagar. Si la GUI se conecta a las reglas comerciales, los cambios en la GUI no pueden afectar esas reglas comerciales.

Los límites se trazan donde hay un eje de cambio. Los componentes de un lado de la frontera cambian a ritmos diferentes y por diferentes razones que los componentes del otro lado de la frontera.

Las GUI cambian en diferentes momentos y a diferentes ritmos que las reglas comerciales, por lo que debe haber un límite entre ellas. Las reglas comerciales cambian en diferentes momentos y por diferentes razones que los marcos de inyección de dependencia, por lo que debe haber un límite entre ellas.

Esto es simplemente, nuevamente, el Principio de Responsabilidad Única. El SRP nos dice dónde trazar nuestros límites.

CONCLUSIÓN

Para dibujar líneas divisorias en una arquitectura de software, primero se divide el sistema en componentes. Algunos de esos componentes son reglas comerciales fundamentales; otros son complementos que contienen funciones necesarias que no están directamente relacionadas con el negocio principal. Luego organiza el código en esos componentes de manera que las flechas entre ellos apunten en una dirección: hacia el negocio principal.

Debe reconocer esto como una aplicación del Principio de Inversión de Dependencia y el Principio de Abstracciones Estables. Las flechas de dependencia están dispuestas para señalar desde los detalles de nivel inferior hasta las abstracciones de nivel superior.

1. La palabra “arquitectura” aparece aquí entre comillas porque tres niveles no es una arquitectura; es una topología.

Es exactamente el tipo de decisión que una buena arquitectura se esfuerza por aplazar.

2. Muchos años después pudimos incorporar el marco Velocity a FitNesse.

18

ANATOMÍA DE LÍMITES



La arquitectura de un sistema está definida por un conjunto de componentes de software y los límites que los separan. Esos límites se presentan de muchas formas diferentes. En este capítulo veremos algunos de los más comunes.

CRUCE DE LÍMITES

En tiempo de ejecución, un cruce de límites no es más que una función en un lado del límite que llama a una función en el otro lado y pasa algunos datos.

El truco para crear un cruce de límites adecuado es gestionar las dependencias del código fuente.

¿Por qué código fuente? Porque cuando un módulo de código fuente cambia, es posible que sea necesario cambiar o volver a compilar otros módulos de código fuente y luego volver a implementarlos. Gestionar y construir cortafuegos contra este cambio es de lo que se tratan los límites.

EL TEMIDO MONOLITO

El más simple y común de los límites arquitectónicos no tiene una representación física estricta. Es simplemente una segregación disciplinada de funciones y datos dentro de un único procesador y un único espacio de direcciones. En un capítulo anterior, llamé a esto modo de desacoplamiento a nivel de fuente.

Desde el punto de vista de la implementación, esto equivale a nada más que un único archivo ejecutable: el llamado monolito. Este archivo puede ser un proyecto C o C++ vinculado estáticamente, un conjunto de archivos de clase Java unidos en un archivo jar ejecutable, un conjunto de archivos binarios .NET vinculados a un único archivo .EXE , etc.

El hecho de que los límites no sean visibles durante el despliegue de un monolito no significa que no estén presentes y no sean significativos. Incluso cuando está vinculado estáticamente a un único ejecutable, la capacidad de desarrollar y organizar de forma independiente los diversos componentes para el ensamblaje final es inmensamente valiosa.

Estas arquitecturas casi siempre dependen de algún tipo de polimorfismo dinámico¹ para gestionar sus dependencias internas. Ésta es una de las razones por las que el desarrollo orientado a objetos se ha convertido en un paradigma tan importante en las últimas décadas. Sin OO, o una forma equivalente de polimorfismo, los arquitectos deben recurrir a la peligrosa práctica de utilizar punteros a funciones para lograr el desacoplamiento apropiado. La mayoría de los arquitectos consideran que el uso prolífico de punteros a funciones es demasiado arriesgado, por lo que se ven obligados a abandonar cualquier tipo de partición de componentes.

El cruce de límites más simple posible es una llamada de función desde un cliente de bajo nivel a un servicio de nivel superior. Tanto la dependencia en tiempo de ejecución como la dependencia en tiempo de compilación apuntan en la misma dirección, hacia el componente de nivel superior.

En [la Figura 18.1](#), el flujo de control cruza el límite de izquierda a derecha. El Cliente llama a la función f() en el Servicio. Pasa una instancia de datos.

El marcador <DS> simplemente indica una estructura de datos. Los datos pueden transmitirse como

argumento de función o por algún otro medio más elaborado. Tenga en cuenta que la definición de los datos está en el lado llamado del límite.

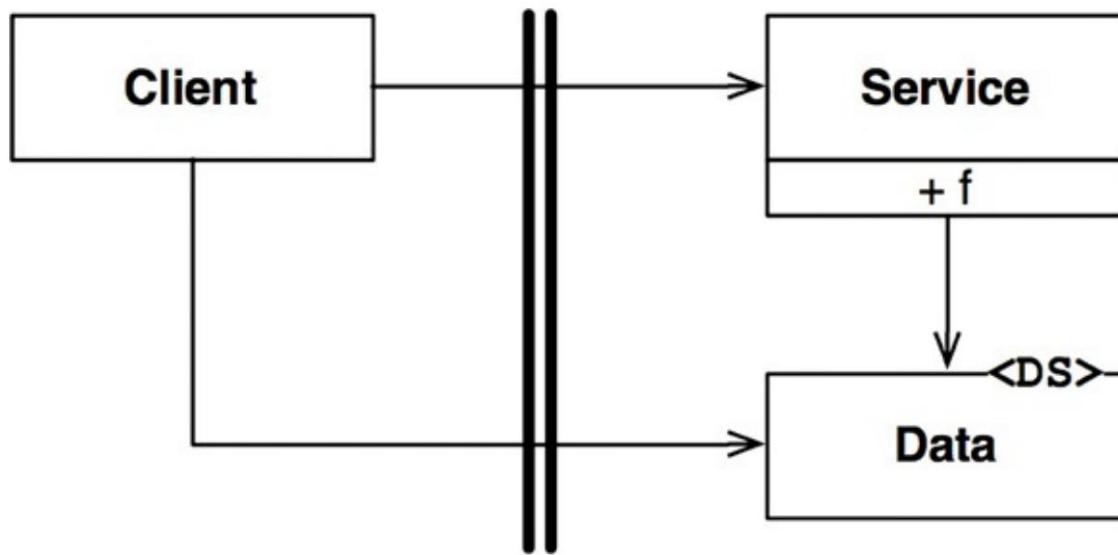


Figura 18.1 El flujo de control cruza la frontera de un nivel inferior a un nivel superior

Cuando un cliente de alto nivel necesita invocar un servicio de nivel inferior, se utiliza el polimorfismo dinámico para invertir la dependencia frente al flujo de control. La dependencia del tiempo de ejecución se opone a la dependencia del tiempo de compilación.

En [la Figura 18.2](#), el flujo de control cruza el límite de izquierda a derecha como antes. El Cliente de alto nivel llama a la función f () del ServicImpl de nivel inferior a través de la interfaz de Servicio . Sin embargo, tenga en cuenta que todas las dependencias cruzan el límite de derecha a izquierda hacia el componente de nivel superior. Tenga en cuenta también que la definición de la estructura de datos está en el lado llamante del límite.

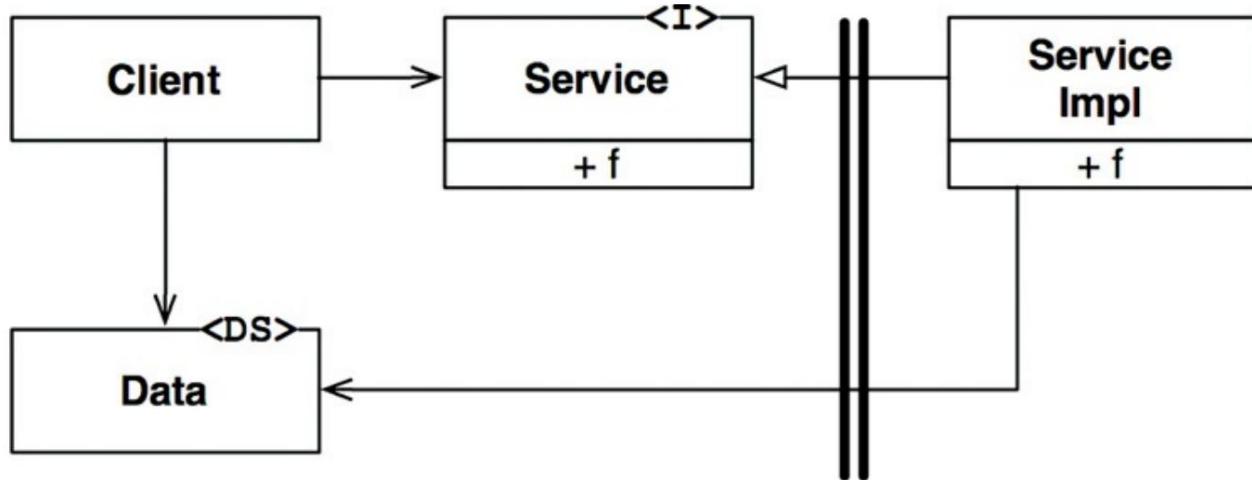


Figura 18.2 Cruzando el límite contra el flujo de control

Incluso en un ejecutable monolítico y vinculado estáticamente, este tipo de partición disciplinada puede ayudar enormemente en el trabajo de desarrollo, prueba e implementación del proyecto. Los equipos pueden trabajar independientemente unos de otros en sus propios componentes sin pisarse los pies unos a otros. Los componentes de alto nivel siguen siendo independientes de los detalles de nivel inferior.

Las comunicaciones entre componentes de un monolito son muy rápidas y económicas. Por lo general, son solo llamadas a funciones. En consecuencia, las comunicaciones a través de límites desacoplados a nivel de fuente pueden ser muy comunicativas.

Dado que la implementación de monolitos generalmente requiere compilación y vinculación estática, los componentes de estos sistemas generalmente se entregan como código fuente.

COMPONENTES DE IMPLEMENTACIÓN

La representación física más simple de un límite arquitectónico es una biblioteca vinculada dinámicamente como una DLL .Net, un archivo jar de Java, una Ruby Gem o una biblioteca compartida de UNIX. La implementación no implica compilación. En cambio, los componentes se entregan en binario o en alguna forma implementable equivalente. Este es el modo de desacoplamiento a nivel de implementación. El acto de implementación es simplemente reunir estas unidades desplegables en alguna forma conveniente, como un archivo WAR o incluso simplemente un directorio.

Con esa única excepción, los componentes a nivel de implementación son lo mismo que los monolitos. Generalmente todas las funciones existen en el mismo procesador y dirección.

espacio. Las estrategias para separar los componentes y gestionar sus dependencias son las mismas.

[2](#)

Al igual que con los monolitos, las comunicaciones a través de los límites de los componentes de implementación son solo llamadas de función y, por lo tanto, son muy económicas. Puede haber un éxito puntual en el enlace dinámico o la carga en tiempo de ejecución, pero las comunicaciones a través de estos límites aún pueden ser muy comunicativas.

HILOS

Tanto los monolitos como los componentes de implementación pueden utilizar subprocessos. Los subprocessos no son límites arquitectónicos ni unidades de implementación, sino más bien una forma de organizar el cronograma y el orden de ejecución. Pueden estar contenidos completamente dentro de un componente o distribuidos entre muchos componentes.

PROCESOS LOCALES

Un límite arquitectónico físico mucho más fuerte es el proceso local. Normalmente, un proceso local se crea desde la línea de comando o una llamada al sistema equivalente.

Los procesos locales se ejecutan en el mismo procesador o en el mismo conjunto de procesadores dentro de un núcleo múltiple, pero se ejecutan en espacios de direcciones separados. La protección de la memoria generalmente evita que dichos procesos comparten memoria, aunque a menudo se utilizan particiones de memoria compartida.

Muy a menudo, los procesos locales se comunican entre sí mediante sockets o algún otro tipo de instalación de comunicación del sistema operativo, como buzones de correo o colas de mensajes.

Cada proceso local puede ser un monolito vinculado estáticamente o puede estar compuesto por componentes de implementación vinculados dinámicamente. En el primer caso, varios procesos monolíticos pueden tener los mismos componentes compilados y vinculados. En este último caso, pueden compartir los mismos componentes de implementación vinculados dinámicamente.

Piense en un proceso local como una especie de super componente: el proceso consta de componentes de nivel inferior que gestionan sus dependencias mediante polimorfismo dinámico.

La estrategia de segregación entre procesos locales es la misma que para monolitos y componentes binarios. Las dependencias del código fuente apuntan en la misma dirección a través del límite y siempre hacia el componente de nivel superior.

Para los procesos locales, esto significa que el código fuente de los procesos de nivel superior no debe contener los nombres, direcciones físicas o claves de búsqueda de registro de procesos de nivel inferior. Recuerde que el objetivo arquitectónico es que los procesos de nivel inferior sean complementos de procesos de nivel superior.

La comunicación a través de los límites de los procesos locales implica llamadas al sistema operativo, clasificación y decodificación de datos y cambios de contexto entre procesos, que son moderadamente costosos. La charlatanería debe limitarse cuidadosamente.

SERVICIOS

El límite más fuerte es un servicio. Un servicio es un proceso, generalmente iniciado desde la línea de comando o mediante una llamada al sistema equivalente. Los servicios no dependen de su ubicación física. Dos servicios de comunicación pueden operar o no en el mismo procesador físico o multinúcleo. Los servicios asumen que todas las comunicaciones se realizan a través de la red.

Las comunicaciones a través de los límites de los servicios son muy lentas en comparación con las llamadas a funciones. Los tiempos de respuesta pueden variar desde decenas de milisegundos hasta segundos. Se debe tener cuidado de evitar chatear siempre que sea posible. Las comunicaciones a este nivel deben lidiar con altos niveles de latencia.

De lo contrario, se aplican a los servicios las mismas reglas que a los procesos locales. Los servicios de nivel inferior deben “conectarse” a los servicios de nivel superior. El código fuente de los servicios de nivel superior no debe contener ningún conocimiento físico específico (por ejemplo, un URI) de ningún servicio de nivel inferior.

CONCLUSIÓN

La mayoría de los sistemas, excepto los monolitos, utilizan más de una estrategia de límites. Un sistema que hace uso de límites de servicios también puede tener algunos límites de procesos locales. De hecho, un servicio es a menudo sólo una fachada para un conjunto de procesos locales que interactúan. Es casi seguro que un servicio o un proceso local será una

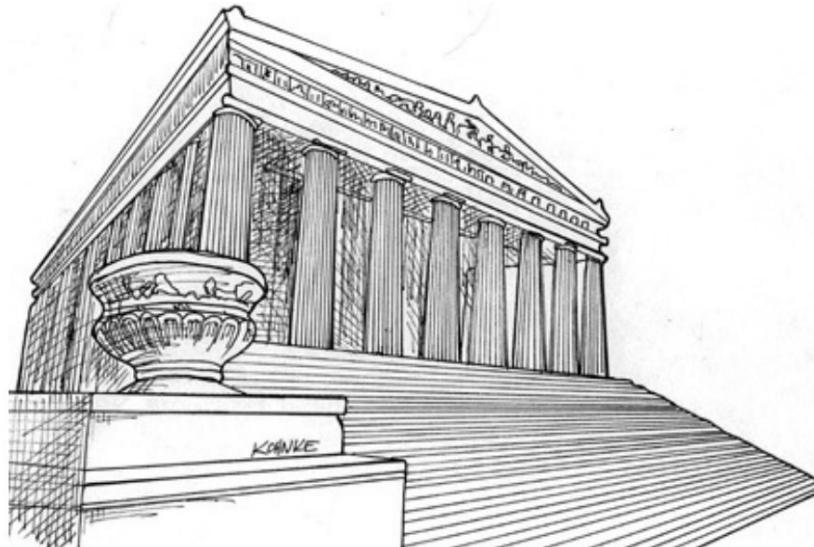
Monolito compuesto por componentes de código fuente o un conjunto de componentes de implementación vinculados dinámicamente.

Esto significa que los límites de un sistema a menudo serán una mezcla de límites locales hablados y límites que están más relacionados con la latencia.

1. El polimorfismo estático (p. ej., genéricos o plantillas) a veces puede ser un medio viable de dependencia gestión en sistemas monolíticos, especialmente en lenguajes como C++. Sin embargo, el desacoplamiento que ofrecen los genéricos no puede protegerlo de la necesidad de recompilación y redistribución como lo hace el polimorfismo dinámico.
2. Aunque el polimorfismo estático no es una opción en este caso.

19

POLÍTICA Y NIVEL



Los sistemas de software son declaraciones de política. De hecho, en esencia, eso es todo lo que es un programa de computadora. Un programa de computadora es una descripción detallada de la política mediante la cual los insumos se transforman en productos.

En la mayoría de los sistemas no triviales, esa política se puede dividir en muchas declaraciones de política más pequeñas y diferentes. Algunas de esas declaraciones describirán cómo se deben calcular reglas comerciales particulares. Otros describirán cómo se formatearán ciertos informes. Otros describirán cómo se validarán los datos de entrada.

Parte del arte de desarrollar una arquitectura de software consiste en separar cuidadosamente esas políticas entre sí y reagruparlas según la forma en que se aplican.

cambiar. Las políticas que cambian por las mismas razones y en los mismos momentos están al mismo nivel y pertenecen juntas al mismo componente. Las políticas que cambian por diferentes razones, o en diferentes momentos, se encuentran en diferentes niveles y deben separarse en diferentes componentes.

El arte de la arquitectura a menudo implica formar los componentes reagrupados en un gráfico acíclico dirigido. Los nodos del gráfico son los componentes que contienen políticas al mismo nivel. Los bordes dirigidos son las dependencias entre esos componentes. Conectan componentes que se encuentran en diferentes niveles.

Esas dependencias son código fuente, dependencias en tiempo de compilación. En Java, son declaraciones de importación . En C#, utilizan declaraciones . En Ruby, son declaraciones obligatorias . Son las dependencias necesarias para que el compilador funcione.

En una buena arquitectura, la dirección de esas dependencias se basa en el nivel de los componentes que conectan. En todos los casos, los componentes de bajo nivel están diseñados para que dependan de componentes de alto nivel.

NIVEL

Una definición estricta de "nivel" es "la distancia desde las entradas y salidas". Cuanto más alejada esté una política tanto de los insumos como de los productos del sistema, mayor será su nivel. Las políticas que gestionan las entradas y salidas son las políticas de nivel más bajo del sistema.

El diagrama de flujo de datos de [la Figura 19.1](#) muestra un programa de cifrado simple que lee caracteres de un dispositivo de entrada, traduce los caracteres usando una tabla y luego escribe los caracteres traducidos en un dispositivo de salida. Los flujos de datos se muestran como flechas sólidas curvas. Las dependencias del código fuente diseñadas correctamente se muestran como líneas discontinuas rectas.

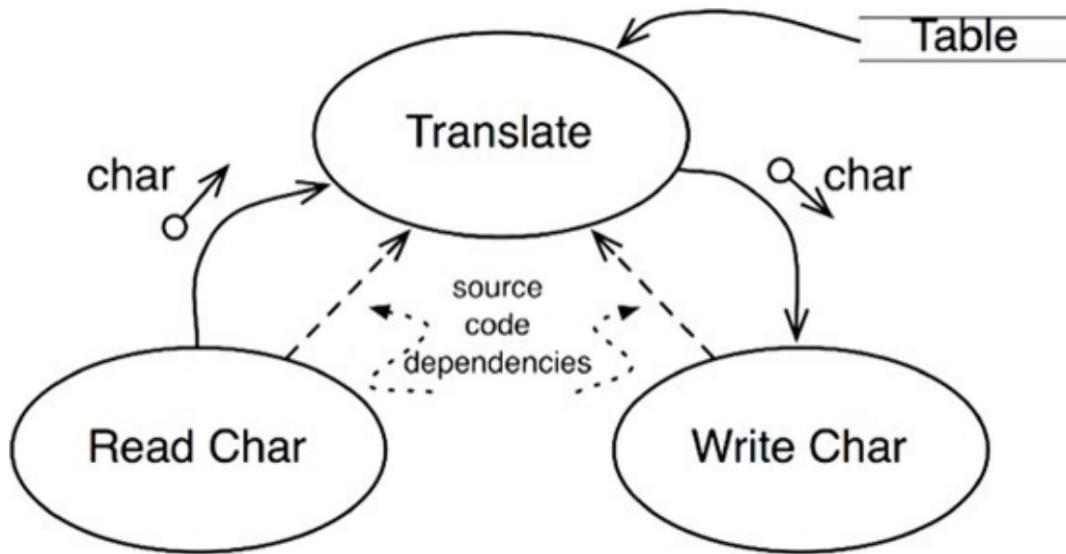


Figura 19.1 Un programa de cifrado simple

El componente Traducir es el componente de más alto nivel en este sistema porque es el [1](#) componente que está más alejado de las entradas y salidas.

Tenga en cuenta que los flujos de datos y las dependencias del código fuente no siempre apuntan en la misma dirección. Esto, nuevamente, es parte del arte de la arquitectura de software. Queremos que las dependencias del código fuente se desacoplen del flujo de datos y se acoplen al nivel.

Sería fácil crear una arquitectura incorrecta escribiendo el programa de cifrado de esta manera:

[Haga clic aquí para ver la imagen del código](#)

```
función cifrar() { while(true)
    writeChar(traducir(readChar())); }
```

Esta es una arquitectura incorrecta porque la función de cifrado de alto nivel depende de las funciones readChar y writeChar de nivel inferior .

En el diagrama de clases de [la Figura 19.2 se muestra una mejor arquitectura para este sistema.](#) Observe el borde discontinuo que rodea la clase Encrypt y las interfaces CharWriter y CharReader . Todas las dependencias que cruzan esa frontera apuntan hacia el interior. Esta unidad es el elemento de más alto nivel del sistema.

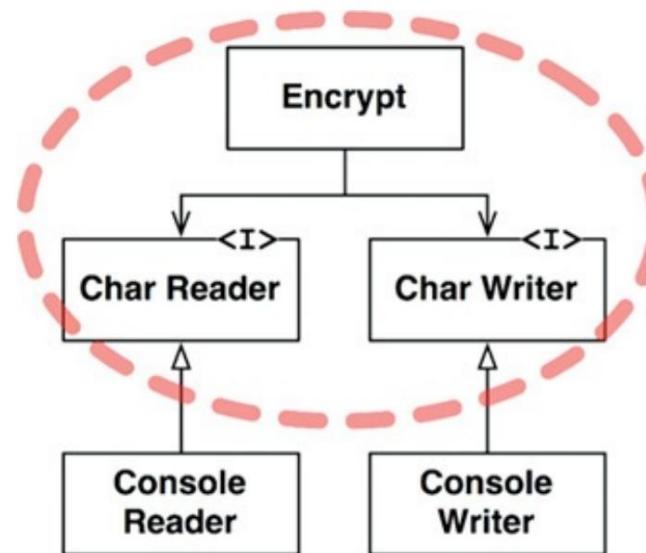


Figura 19.2 Diagrama de clases que muestra una mejor arquitectura para el sistema

ConsoleReader y **ConsoleWriter** se muestran aquí como clases. Son de bajo nivel porque están cerca de las entradas y salidas.

Observe cómo esta estructura desacopla la política de cifrado de alto nivel de las políticas de entrada/salida de nivel inferior. Esto hace que la política de cifrado sea utilizable en una amplia gama de contextos. Cuando se realizan cambios en las políticas de entrada y salida, no es probable que afecten la política de cifrado.

Recuerde que las políticas se agrupan en componentes según la forma en que cambian. Las políticas que cambian por las mismas razones y en los mismos momentos son agrupadas por el SRP y el CCP. Las políticas de nivel superior (aquellas que están más alejadas de los insumos y productos) tienden a cambiar con menos frecuencia y por razones más importantes que las políticas de nivel inferior. Las políticas de nivel inferior (aquellas que están más cercanas a los insumos y productos) tienden a cambiar con frecuencia y con mayor urgencia, pero por razones menos importantes.

Por ejemplo, incluso en el ejemplo trivial del programa de cifrado, es mucho más probable que cambien los dispositivos IO que el algoritmo de cifrado. Si el algoritmo de cifrado cambia, probablemente será por una razón más importante que un cambio en uno de los dispositivos IO.

Mantener estas políticas separadas, con todas las dependencias del código fuente apuntando en la dirección de las políticas de nivel superior, reduce el impacto del cambio. Los cambios triviales pero urgentes en los niveles más bajos del sistema tienen poco o ningún impacto en

los niveles más altos y más importantes.

Otra forma de ver este problema es observar que los componentes de nivel inferior deben ser complementos de los componentes de nivel superior. El [diagrama de componentes de la Figura 19.3](#) muestra esta disposición. El componente Encryption no sabe nada del componente IODevices ; el componente IODevices depende del componente Encryption .

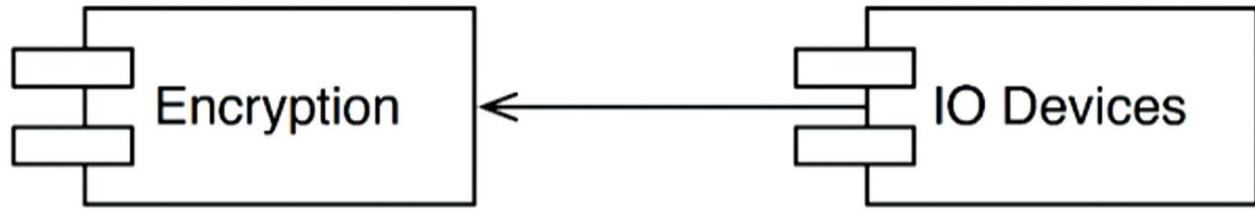


Figura 19.3 Los componentes de nivel inferior deben conectarse a los componentes de nivel superior

CONCLUSIÓN

En este punto, esta discusión de políticas ha involucrado una mezcla del Principio de Responsabilidad Única, el Principio Abierto-Cerrado, el Principio de Cierre Común, el Principio de Inversión de Dependencia, el Principio de Dependencias Estables y el Principio de Abstracciones Estables. Mire hacia atrás y vea si puede identificar dónde se utilizó cada principio y por qué.

1. Meilir Page-Jones llamó a este componente la "Transformación central" en su libro *La guía práctica para Diseño de sistemas estructurados*, 2^a ed. (Prensa Yourdon, 1988).

20

REGLAS DEL NEGOCIO



Si vamos a dividir nuestra aplicación en reglas de negocio y complementos, será mejor que comprendamos bien qué son realmente las reglas de negocio. Resulta que hay varios tipos diferentes.

Estrictamente hablando, las reglas comerciales son reglas o procedimientos que generan o ahorran dinero a la empresa. En sentido estricto, estas reglas generarían o ahorrarían dinero a la empresa, independientemente de si se implementaran en una computadora. Ganarían o ahorrarían dinero incluso si se ejecutaran manualmente.

El hecho de que un banco cobre N% de interés por un préstamo es una regla comercial que hace que el banco gane dinero. No importa si un programa de computadora calcula el interés o si un empleado con un ábaco calcula el interés.

Llamaremos a estas reglas Reglas de Negocio Críticas, porque son críticas para el negocio en sí y existirían incluso si no existiera un sistema para automatizarlas.

Las reglas comerciales críticas generalmente requieren algunos datos para trabajar. Por ejemplo, nuestro préstamo requiere un saldo de préstamo, una tasa de interés y un calendario de pagos.

A estos datos los llamaremos datos comerciales críticos. Estos son los datos que existirían incluso si el sistema no estuviera automatizado.

Las reglas críticas y los datos críticos están indisolublemente ligados, por lo que son un buen candidato para un objeto. A este tipo de objeto lo llamaremos Entidad. [1](#)

ENTIDADES

Una entidad es un objeto dentro de nuestro sistema informático que incorpora un pequeño conjunto de reglas comerciales críticas que operan sobre datos comerciales críticos. El objeto Entidad contiene los datos comerciales críticos o tiene muy fácil acceso a esos datos.

La interfaz de la Entidad consta de las funciones que implementan las Reglas de Negocio Críticas que operan sobre esos datos.

Por ejemplo, [la Figura 20.1](#) muestra cómo podría verse nuestra entidad Préstamo como clase en UML. Tiene tres datos comerciales críticos y presenta tres reglas comerciales críticas relacionadas en su interfaz.

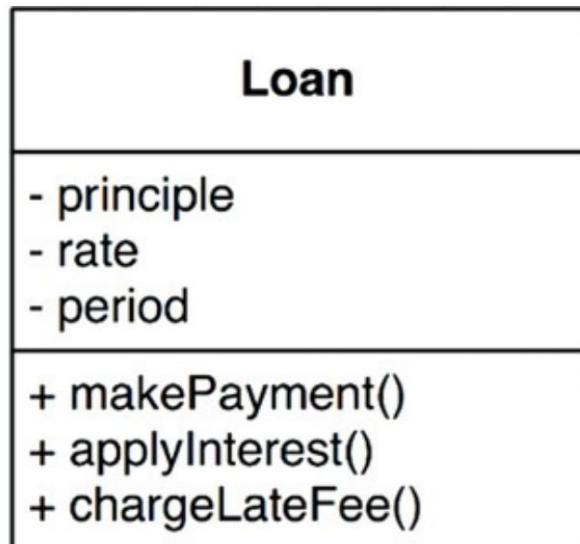


Figura 20.1 Entidad de préstamo como clase en UML

Cuando creamos este tipo de clase, reunimos el software que implementa un concepto que es crítico para el negocio y lo sepáramos de cualquier otra preocupación en el sistema automatizado que estamos construyendo. Esta clase es la única representante del negocio. Está libre de preocupaciones sobre bases de datos, interfaces de usuario o marcos de trabajo de terceros. Podría servir al negocio en cualquier sistema, independientemente de cómo se presentara ese sistema, de cómo se almacenaran los datos o de cómo estuvieran dispuestas las computadoras en ese sistema. La Entidad es puro negocio y nada más.

A algunos de ustedes les puede preocupar que lo haya llamado clase. No lo seas. No es necesario utilizar un lenguaje orientado a objetos para crear una Entidad. Todo lo que se requiere es vincular los datos comerciales críticos y las reglas comerciales críticas en un módulo de software único e independiente.

CASOS DE USO

No todas las reglas de negocio son tan puras como las Entidades. Algunas reglas comerciales generan o ahorrarán dinero para la empresa al definir y restringir la forma en que opera un sistema automatizado. Estas reglas no se utilizarían en un entorno manual, porque sólo tienen sentido como parte de un sistema automatizado.

Por ejemplo, imagine una aplicación que utilizan los funcionarios bancarios para crear un nuevo préstamo. El banco puede decidir que no quiere que los oficiales de crédito ofrezcan estimaciones de pago del préstamo hasta que primero hayan recopilado y validado la información de contacto y se hayan asegurado de que el puntaje crediticio del candidato sea 500 o superior. Por este motivo, el banco puede especificar que el sistema no pasará a la pantalla de estimación de pago hasta que se haya completado y verificado la pantalla de información de contacto y se haya confirmado que la puntuación crediticia es mayor que el límite.

[2](#) Este es un caso de uso. Un caso de uso es una descripción de la forma en que se utiliza un sistema automatizado. Especifica la entrada que debe proporcionar el usuario, la salida que se devolverá al usuario y los pasos de procesamiento involucrados en la producción de esa salida.

Un caso de uso describe reglas comerciales específicas de la aplicación en contraposición a las reglas comerciales críticas dentro de las Entidades.

[La Figura 20.2](#) muestra un ejemplo de un caso de uso. Observe que en la última línea menciona al Cliente. Esta es una referencia a la entidad Cliente, que

contiene las Reglas Comerciales Críticas que rigen la relación entre el banco y sus clientes.



Figura 20.2 Caso de uso de ejemplo

Los casos de uso contienen las reglas que especifican cómo y cuándo se invocan las reglas comerciales críticas dentro de las entidades. Los casos de uso controlan la danza de las Entidades.

Observe también que el caso de uso no describe la interfaz de usuario más que para especificar informalmente los datos que entran desde esa interfaz y los datos que salen a través de esa interfaz. A partir del caso de uso, es imposible saber si la aplicación se entrega en la web, en un cliente pesado, en una consola o es un servicio puro.

Esto es muy importante. Los casos de uso no describen cómo aparece el sistema ante el usuario. En cambio, describen las reglas específicas de la aplicación que gobiernan la interacción entre los usuarios y las Entidades. La forma en que los datos entran y salen del sistema es irrelevante para los casos de uso.

Un caso de uso es un objeto. Tiene una o más funciones que implementan las reglas comerciales específicas de la aplicación. También tiene elementos de datos que incluyen los datos de entrada, los datos de salida y las referencias a las Entidades apropiadas con las que interactúa.

Las entidades no tienen conocimiento de los casos de uso que las controlan. Este es otro ejemplo de la dirección de las dependencias siguiendo la Dependencia

Principio de inversión. Los conceptos de alto nivel, como las Entidades, no saben nada de los conceptos de nivel inferior, como los casos de uso. En cambio, los casos de uso de nivel inferior conocen las Entidades de nivel superior.

¿Por qué las Entidades son de alto nivel y los casos de uso de menor nivel? Porque los casos de uso son específicos de una única aplicación y, por tanto, están más cerca de las entradas y salidas de ese sistema. Las entidades son generalizaciones que se pueden utilizar en muchas aplicaciones diferentes, por lo que están más alejadas de las entradas y salidas del sistema. Los casos de uso dependen de las entidades; Las entidades no dependen de los casos de uso.

MODELOS DE SOLICITUD Y RESPUESTA

Los casos de uso esperan datos de entrada y producen datos de salida. Sin embargo, un objeto de caso de uso bien formado no debería tener idea de la forma en que los datos se comunican al usuario o a cualquier otro componente. ¡Ciertamente no queremos que el código dentro de la clase de caso de uso sepa sobre HTML o SQL!

La clase de caso de uso acepta estructuras de datos de solicitud simples como entrada y devuelve estructuras de datos de respuesta simples como salida. Estas estructuras de datos no dependen de nada. No se derivan de interfaces de marco estándar como `HttpRequest` y `HttpResponse`. No saben nada de la web, ni comparten ninguna de las características de cualquier interfaz de usuario que pueda existir.

Esta falta de dependencias es crítica. Si los modelos de solicitud y respuesta no son independientes, entonces los casos de uso que dependen de ellos estarán indirectamente vinculados a cualquier dependencia que los modelos lleven consigo.

Es posible que tenga la tentación de que estas estructuras de datos contengan referencias a objetos de entidad. Podría pensar que esto tiene sentido porque las Entidades y los modelos de solicitud/respuesta comparten muchos datos. ¡Evita esta tentación! El propósito de estos dos objetos es muy diferente. Con el tiempo cambiarán por razones muy diferentes, por lo que vincularlos de cualquier manera viola los principios de cierre común y responsabilidad única. El resultado sería una gran cantidad de datos vagabundos y muchos condicionales en su código.

CONCLUSIÓN

Las reglas comerciales son la razón por la que existe un sistema de software. Son la funcionalidad principal. Llevan el código que genera o ahorra dinero. Son las joyas de la familia.

Las reglas de negocio deben permanecer impecables, inmaculadas por preocupaciones más básicas como la interfaz de usuario o la base de datos utilizada. Idealmente, el código que representa las reglas de negocio debería ser el corazón del sistema, con preocupaciones menores conectadas a él. Las reglas de negocio deben ser el código más independiente y reutilizable del sistema.

1. Este es el nombre que le dio Ivar Jacobson a este concepto (I. Jacobson et al., Object Oriented Software Engineering, Addison-Wesley, 1992).

2. Ibídem.

21

ARQUITECTURA GRITANTE



Imagina que estás mirando los planos de un edificio. Este documento, elaborado por un arquitecto, proporciona los planos del edificio. ¿Qué te dicen estos planes?

Si los planos que está viendo son para una residencia unifamiliar, probablemente verá una entrada principal, un vestíbulo que conduce a una sala de estar y tal vez a un comedor. Probablemente habrá una cocina a poca distancia, cerca del comedor. Quizás haya un área de comedor al lado de la cocina, y probablemente una sala familiar cerca de ella. Cuando mirara esos planos, no habría duda de que estaba ante una casa unifamiliar. La arquitectura gritaría: "INICIO".

Ahora suponga que está mirando la arquitectura de una biblioteca. Probablemente vea una gran entrada, un área para los empleados de registro de entrada y salida, áreas de lectura, pequeñas salas de conferencias y una galería tras otra capaz de albergar estanterías para todos los libros de la biblioteca. Esa arquitectura gritaría: "BIBLIOTECA".

Entonces, ¿qué grita la arquitectura de tu aplicación? Cuando observa la estructura de directorios de nivel superior y los archivos fuente en el paquete de nivel más alto, ¿gritan "Sistema de atención médica", "Sistema de contabilidad" o "Sistema de gestión de inventario"? ¿O gritan "Rails", "Spring/Hibernate" o "ASP"?

EL TEMA DE UNA ARQUITECTURA

Regrese y lea el trabajo fundamental de Ivar Jacobson sobre arquitectura de software: Ingeniería de software orientada a objetos. Observe el subtítulo del libro: Un enfoque basado en casos de uso. En este libro, Jacobson señala que las arquitecturas de software son estructuras que soportan los casos de uso del sistema. Así como los planos de una casa o una biblioteca gritan sobre los casos de uso de esos edificios, la arquitectura de una aplicación de software debería gritar sobre los casos de uso de la aplicación.

Las arquitecturas no se tratan (o no deberían tratarse) de marcos. Las arquitecturas no deben ser proporcionadas por marcos. Los marcos son herramientas para utilizar, no arquitecturas a las que adaptarse. Si su arquitectura se basa en marcos, entonces no puede basarse en sus casos de uso.

EL PROPÓSITO DE UNA ARQUITECTURA

Las buenas arquitecturas se centran en casos de uso para que los arquitectos puedan describir de forma segura las estructuras que respaldan esos casos de uso sin comprometerse con marcos, herramientas y entornos. Nuevamente, considere los planos de una casa. La primera preocupación del arquitecto es asegurarse de que la casa sea utilizable, no asegurarse de que esté hecha de ladrillos. De hecho, el arquitecto se esfuerza por garantizar que el propietario pueda tomar decisiones sobre el material exterior (ladrillos, piedra o cedro) más adelante, después de que los planos garanticen que se cumplen los casos

Una buena arquitectura de software permite decisiones sobre frameworks, bases de datos, web.

servidores y otras cuestiones y herramientas ambientales que se pospondrán y retrasarán. Los marcos son opciones que deben dejarse abiertas. Una buena arquitectura hace innecesario decidirse por Rails, Spring, Hibernate, Tomcat o MySQL, hasta mucho más adelante en el proyecto. Una buena arquitectura también hace que sea fácil cambiar de opinión sobre esas decisiones. Una buena arquitectura enfatiza los casos de uso y los desvincula de las preocupaciones periféricas.

¿PERO Y LA WEB?

¿Es la web una arquitectura? ¿El hecho de que su sistema se entregue en la web dicta la arquitectura de su sistema? ¡Por supuesto que no! La web es un mecanismo de entrega (un dispositivo IO) y la arquitectura de su aplicación debe tratarlo como tal. El hecho de que su aplicación se entregue a través de la web es un detalle y no debe dominar la estructura de su sistema. De hecho, la decisión de que su solicitud se entregará a través de la web es algo que debe posponer. La arquitectura de su sistema debe ser lo más ignorante posible sobre cómo se entregará. Debería poder entregarlo como una aplicación de consola, una aplicación web, una aplicación de cliente pesado o incluso una aplicación de servicio web, sin complicaciones excesivas ni cambios en la arquitectura fundamental.

LOS MARCOS SON HERRAMIENTAS, NO MANERAS DE VIDA

Los marcos pueden ser muy poderosos y muy útiles. Los autores de marcos a menudo creen profundamente en sus marcos. Los ejemplos que escriben sobre cómo utilizar sus marcos se cuentan desde el punto de vista de un verdadero creyente. Otros autores que escriben sobre el marco también tienden a ser discípulos de la verdadera creencia. Le muestran la forma de utilizar el marco. A menudo asumen una posición omniabarcante, omnipresente, de dejar que el marco lo haga todo.

Ésta no es la posición que usted desea adoptar.

Mire cada marco con ojos cansados. Míralo con escepticismo. Sí, podría ayudar, pero ¿a qué precio? Pregúntese cómo debe utilizarlo y cómo debe protegerse de él. Piense en cómo puede preservar el énfasis en los casos de uso de su arquitectura. Desarrollar una estrategia que evite que el marco

de apoderarse de esa arquitectura.

ARQUITECTURAS PROBABLES

Si la arquitectura de su sistema tiene que ver con los casos de uso, y si ha mantenido sus marcos a distancia, entonces debería poder realizar pruebas unitarias de todos esos casos de uso sin ninguno de los marcos implementados. No debería necesitar que el servidor web esté en ejecución para ejecutar sus pruebas. No debería necesitar la base de datos conectada para ejecutar sus pruebas. Sus objetos de Entidad deben ser objetos antiguos y simples que no dependan de marcos o bases de datos u otras complicaciones. Los objetos de su caso de uso deben coordinar sus objetos de Entidad. Finalmente, todos ellos juntos deberían poder probarse *in situ*, sin ninguna de las complicaciones de las estructuras.

CONCLUSIÓN

Su arquitectura debe informar a los lectores sobre el sistema, no sobre los marcos que utilizó en su sistema. Si está construyendo un sistema de atención médica, cuando los nuevos programadores miren el repositorio de código fuente, su primera impresión debería ser: "Oh, este es un sistema de atención médica". Esos nuevos programadores deberían poder aprender todos los casos de uso del sistema, pero aún no saber cómo se entrega el sistema. Es posible que se acerquen a usted y le digan:

"Vemos algunas cosas que parecen modelos, pero ¿dónde están las vistas y los controladores?"

Y deberías responder:

"Oh, esos son detalles que no necesitan preocuparnos en este momento. Decidiremos sobre ellos más tarde".

22

LA ARQUITECTURA LIMPIA



Durante las últimas décadas hemos visto toda una gama de ideas sobre la arquitectura de sistemas. Éstas incluyen:

- Arquitectura Hexagonal (también conocida como Puertos y Adaptadores), desarrollada por Alistair Cockburn, y adoptado por Steve Freeman y Nat Pryce en su maravilloso libro *Growing Object Oriented Software with Tests*.
- DCI de James Coplien y Trygve Reenskaug • BCE, presentado por Ivar Jacobson de su libro *Object Oriented Software Ingeniería: un enfoque basado en casos de uso*

Aunque todas estas arquitecturas varían algo en sus detalles, son muy

similar. Todos tienen el mismo objetivo que es la separación de preocupaciones.

Todos logran esta separación dividiendo el software en capas. Cada uno tiene al menos una capa para reglas comerciales y otra capa para interfaces de usuario y sistema.

Cada una de estas arquitecturas produce sistemas que tienen las siguientes características:

- Independiente de los marcos. La arquitectura no depende de la existencia de alguna biblioteca de software cargado de funciones. Esto le permite utilizar dichos marcos como herramientas, en lugar de obligarlo a meter su sistema en sus limitaciones limitadas.
- Comprobable. Las reglas comerciales se pueden probar sin la interfaz de usuario, la base de datos, el servidor web o cualquier otro elemento externo.
- Independiente de la UI. La interfaz de usuario puede cambiar fácilmente, sin cambiar el resto del sistema. Una interfaz de usuario web podría reemplazarse por una interfaz de usuario de consola, por ejemplo, sin cambiar las reglas comerciales.
- Independiente de la base de datos. Puede cambiar Oracle o SQL Server por Mongo, BigTable, CouchDB u otra cosa. Sus reglas comerciales no están vinculadas a la base de datos.
- Independiente de cualquier agencia externa. De hecho, sus reglas de negocio no saben nada sobre las interfaces con el mundo exterior.

El diagrama de [la Figura 22.1](#) es un intento de integrar todas estas arquitecturas en una única idea viable.

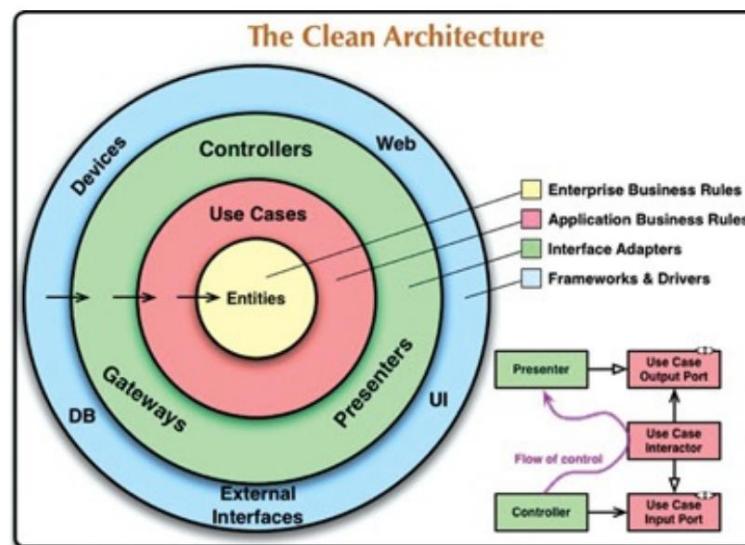


Figura 22.1 La arquitectura limpia

LA REGLA DE DEPENDENCIA

Los círculos concéntricos de [la Figura 22.1](#) representan diferentes áreas del software. En general, cuanto más avanzas, más alto es el nivel del software. Los círculos exteriores son mecanismos. Los círculos internos son las políticas.

La regla primordial que hace que esta arquitectura funcione es la regla de dependencia:

Las dependencias del código fuente deben apuntar sólo hacia adentro, hacia políticas de nivel superior.

Nada en un círculo interno puede saber nada acerca de algo en un círculo externo. En particular, el nombre de algo declarado en un círculo exterior no debe mencionarse en el código en un círculo interior. Eso incluye funciones, clases, variables o cualquier otra entidad de software nombrada.

Del mismo modo, los formatos de datos declarados en un círculo exterior no deben ser utilizados por un círculo interior, especialmente si esos formatos son generados por un marco en un círculo exterior. No queremos que nada en un círculo exterior impacte los círculos internos.

ENTIDADES

Las entidades encapsulan reglas comerciales críticas para toda la empresa. Una entidad puede ser un objeto con métodos o puede ser un conjunto de estructuras de datos y funciones. No importa siempre que las entidades puedan ser utilizadas por muchas aplicaciones diferentes en la empresa.

Si no tiene una empresa y está escribiendo solo una aplicación, entonces estas entidades son los objetos comerciales de la aplicación. Encapsulan las reglas más generales y de alto nivel. Son los que tienen menos probabilidades de cambiar cuando algo externo cambia.

Por ejemplo, no esperaría que estos objetos se vieran afectados por un cambio en la seguridad o la navegación de la página. Ningún cambio operativo en ninguna aplicación en particular debería afectar la capa de entidad.

CASOS DE USO

El software en la capa de casos de uso contiene reglas comerciales específicas de la aplicación . Encapsula e implementa todos los casos de uso del sistema. Estos usan

Los casos orquestan el flujo de datos hacia y desde las entidades, y dirigen a esas entidades a utilizar sus reglas comerciales críticas para lograr los objetivos del caso de uso.

No esperamos que los cambios en esta capa afecten a las entidades. Tampoco esperamos que esta capa se vea afectada por cambios en externalidades como la base de datos, la interfaz de usuario o cualquiera de los marcos comunes. La capa de casos de uso está aislada de tales preocupaciones.

Sin embargo, esperamos que los cambios en el funcionamiento de la aplicación afecten los casos de uso y, por lo tanto, el software en esta capa. Si los detalles de un caso de uso cambian, entonces parte del código de esta capa seguramente se verá afectado.

ADAPTADORES DE INTERFAZ

El software en la capa de adaptadores de interfaz es un conjunto de adaptadores que convierten datos del formato más conveniente para los casos de uso y entidades al formato más conveniente para alguna agencia externa como la base de datos o la web. Es esta capa, por ejemplo, la que contendrá por completo la arquitectura MVC de una GUI.

Los presentadores, vistas y controladores pertenecen a la capa de adaptadores de interfaz. Es probable que los modelos sean solo estructuras de datos que se pasan de los controladores a los casos de uso y luego de los casos de uso a los presentadores y las vistas.

De manera similar, los datos se convierten, en esta capa, desde la forma más conveniente para entidades y casos de uso, a la forma más conveniente para cualquier marco de persistencia que se esté utilizando (es decir, la base de datos). Ningún código dentro de este círculo debería saber nada sobre la base de datos. Si la base de datos es una base de datos SQL, entonces todo SQL debe restringirse a esta capa y, en particular, a las partes de esta capa que tienen que ver con la base de datos.

También en esta capa se necesita cualquier otro adaptador para convertir datos de algún formulario externo, como un servicio externo, al formulario interno utilizado por los casos de uso y entidades.

MARCOS Y CONDUCTORES

La capa más externa del modelo en [la Figura 22.1](#) generalmente se compone de marcos y herramientas como la base de datos y el marco web. Generalmente no se escribe mucho código en esta capa, aparte del código adhesivo que se comunica con el siguiente círculo hacia adentro.

La capa de marcos y controladores es donde van todos los detalles. La web es un detalle.
La base de datos es un detalle. Mantenemos estas cosas en el exterior, donde pueden causar poco daño.

¿SOLO CUATRO CÍRCULOS?

Los círculos de [la Figura 22.1](#) pretenden ser esquemáticos: es posible que necesite algo más que estos cuatro. No hay ninguna regla que diga que siempre debes tener solo estos cuatro. Sin embargo, siempre se aplica la regla de dependencia. Las dependencias del código fuente siempre apuntan hacia adentro. A medida que uno avanza hacia adentro, aumenta el nivel de abstracción y política. El círculo más externo consta de detalles de hormigón de bajo nivel. A medida que avanza, el software se vuelve más abstracto y encapsula políticas de nivel superior. El círculo más interno es el nivel más general y más elevado.

CRUZANDO LÍMITES

En la parte inferior derecha del diagrama de [la Figura 22.1](#) hay un ejemplo de cómo cruzamos los límites del círculo. Muestra a los controladores y presentadores comunicándose con los casos de uso en la siguiente capa. Tenga en cuenta el flujo de control: comienza en el controlador, avanza a través del caso de uso y luego termina ejecutándose en el presentador. Tenga en cuenta también las dependencias del código fuente: cada una apunta hacia los casos de uso.

Normalmente resolvemos esta aparente contradicción utilizando el principio de inversión de dependencia. En un lenguaje como Java, por ejemplo, organizariamos interfaces y relaciones de herencia de manera que las dependencias del código fuente se opongan al flujo de control justo en los puntos correctos a través de la frontera.

Por ejemplo, supongamos que el caso de uso necesita llamar al presentador. Esta llamada no debe ser directa porque violaría la regla de dependencia: ningún nombre en un círculo externo puede ser mencionado por un círculo interno. Entonces, tenemos el caso de uso que llama a una interfaz (que se muestra en [la Figura 22.1](#) como “puerto de salida del caso de uso”) en el círculo interior, y hacemos que el presentador en el círculo exterior lo implemente.

Se utiliza la misma técnica para cruzar todos los límites de las arquitecturas. Aprovechamos el polimorfismo dinámico para crear dependencias de código fuente que se oponen al flujo de control para que podamos cumplir con la regla de dependencia, sin importar en qué dirección viaje el flujo de control.

QUÉ DATOS SUPERAN LOS LÍMITES

Normalmente, los datos que cruzan los límites consisten en estructuras de datos simples. Puede utilizar estructuras básicas u objetos de transferencia de datos simples si lo desea. O los datos pueden ser simplemente argumentos en llamadas a funciones. O puede empaquetarlo en un mapa hash o convertirlo en un objeto. Lo importante es que las estructuras de datos simples y aisladas crucen los límites. No queremos hacer trampa y pasar objetos de entidad o filas de bases de datos. No queremos que las estructuras de datos tengan ningún tipo de dependencia que viole la regla de dependencia.

Por ejemplo, muchos marcos de bases de datos devuelven un formato de datos conveniente en respuesta a una consulta. Podríamos llamar a esto una "estructura de filas". No queremos pasar esa estructura de filas hacia adentro a través de un límite. Hacerlo violaría la regla de dependencia porque obligaría a un círculo interno a saber algo sobre un círculo externo.

Por lo tanto, cuando pasamos datos a través de un límite, siempre es en la forma más conveniente para el círculo interno.

UN ESCENARIO TÍPICO

El diagrama de [la Figura 22.2](#) muestra un escenario típico para un sistema Java basado en web que utiliza una base de datos. El servidor web recopila los datos de entrada del usuario y los entrega al Controlador en la parte superior izquierda. El controlador empaqueta esos datos en un objeto Java antiguo y simple y pasa este objeto a través del límite de entrada al UseCaseInteractor. El UseCaseInteractor interpreta esos datos y los utiliza para controlar el baile de las Entidades. También utiliza DataAccessInterface para traer los datos utilizados por esas Entidades a la memoria desde la Base de Datos. Al finalizar, UseCaseInteractor recopila datos de las Entidades y construye OutputData como otro antiguo objeto Java. Luego, OutputData se pasa a través de la interfaz OutputBoundary al presentador.

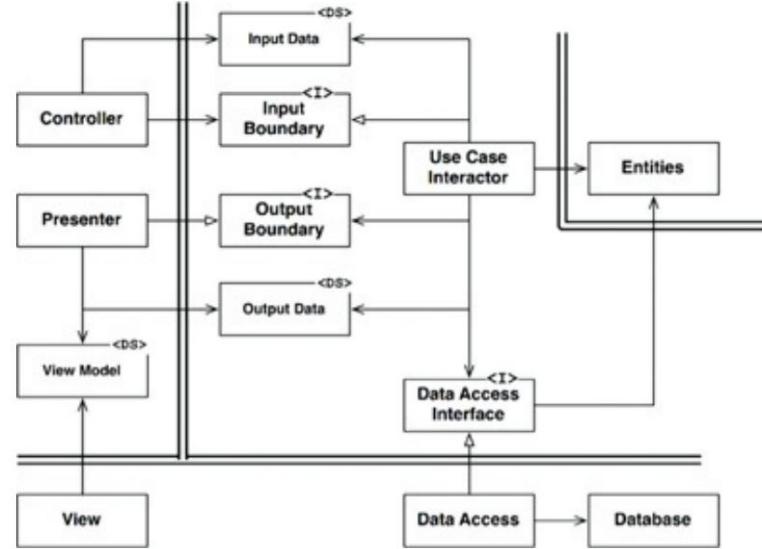


Figura 22.2 Un escenario típico para un sistema Java basado en web que utiliza una base de datos

El trabajo del Presentador es volver a empaquetar `OutputData` en un formato visible como `ViewModel`, que es otro objeto Java simple y antiguo. `ViewModel` contiene principalmente cadenas y indicadores que la vista utiliza para mostrar los datos.

Mientras que `OutputData` puede contener objetos `Date`, Presentador cargará `ViewModel` con las cadenas correspondientes ya formateadas correctamente para el usuario.

Lo mismo se aplica a los objetos de moneda o cualquier otro dato relacionado con el negocio. Los nombres de los botones y elementos de menú se colocan en `ViewModel`, al igual que las banderas que le indican a la vista si esos botones y elementos de menú deben ser grises.

Esto deja a la Vista sin casi nada que hacer más que mover los datos del `ViewModel` a la página HTML .

Tenga en cuenta las direcciones de las dependencias. Todas las dependencias cruzan las líneas fronterizas que apuntan hacia adentro, siguiendo la Regla de Dependencia.

CONCLUSIÓN

Cumplir con estas sencillas reglas no es difícil y le ahorrará muchos dolores de cabeza en el futuro.

Al separar el software en capas y ajustarse a la Regla de Dependencia, creará un sistema intrínsecamente comprobable, con todos los beneficios que eso implica. Cuando alguna de las partes externas del sistema se vuelve obsoleta, como la base de datos o el marco web, puede reemplazar esos elementos obsoletos con un mínimo de complicaciones.

23

PRESENTADORES Y OBJETOS HUMILDES



En [el capítulo 22](#) introdujimos la noción de presentadores. Los presentadores son una forma del patrón Objeto Humilde , que nos ayuda a identificar y proteger los límites arquitectónicos. En realidad, la Arquitectura Limpia del último capítulo estaba llena de implementaciones de Objetos Humildes .

EL PATRÓN DEL OBJETO HUMILDE

El patrón Objeto Humilde es un patrón¹ de diseño que se identificó originalmente como una forma de ayudar a los evaluadores unitarios a separar comportamientos que son difíciles de probar de aquellos que son fáciles de probar. La idea es muy simple: dividir los comportamientos en dos

módulos o clases. Uno de esos módulos es humilde; contiene todos los comportamientos difíciles de probar reducidos a su esencia más pura. El otro módulo contiene todos los comportamientos comprobables que fueron eliminados del humilde objeto.

Por ejemplo, las GUI son difíciles de realizar pruebas unitarias porque es muy difícil escribir pruebas que puedan ver la pantalla y verificar que se muestren los elementos apropiados allí. Sin embargo, la mayor parte del comportamiento de una GUI es, de hecho, fácil de probar. Usando el patrón Objeto Humilde , podemos separar estos dos tipos de comportamientos en dos clases diferentes llamadas Presentador y Vista.

PRESENTADORES Y VISTAS

La Vista es el objeto humilde que es difícil de probar. El código de este objeto se mantiene lo más simple posible. Mueve datos a la GUI pero no los procesa.

El presentador es el objeto comprobable. Su trabajo es aceptar datos de la aplicación y formatearlos para su presentación de modo que la Vista pueda simplemente moverlos a la pantalla. Por ejemplo, si la aplicación quiere que se muestre una fecha en un campo, le entregará al presentador un objeto Fecha . Luego, el presentador formateará esos datos en una cadena adecuada y los colocará en una estructura de datos simple llamada modelo de vista, donde la vista puede encontrarlos.

Si la aplicación quiere mostrar dinero en la pantalla, puede pasar un objeto Moneda al Presentador. El presentador formateará ese objeto con los decimales y marcadores de moneda apropiados, creando una cadena que puede colocar en el modelo de vista. Si el valor de esa moneda se vuelve rojo si es negativo, entonces se establecerá apropiadamente un indicador booleano simple en el modelo de Vista.

Cada botón en la pantalla tendrá un nombre. Ese nombre será una cadena en el modelo de vista, colocada allí por el presentador. Si esos botones están atenuados, el presentador establecerá un indicador booleano apropiado en el modelo de vista. Cada nombre de elemento de menú es una cadena en el modelo de Vista, cargada por el Presentador. El presentador carga los nombres de cada botón de opción, casilla de verificación y campo de texto en cadenas y valores booleanos apropiados en el modelo de vista. El presentador carga las tablas de números que deben mostrarse en la pantalla en tablas de cadenas con el formato adecuado en el modelo de vista.

Cualquier cosa que aparezca en pantalla, y que la aplicación tenga

algún tipo de control se representa en el modelo de vista como una cadena, un valor booleano o una enumeración. A la Vista no le queda nada más que cargar los datos del Modelo de vista en la pantalla. Por eso la Vista es humilde.

PRUEBAS Y ARQUITECTURA

Se sabe desde hace mucho tiempo que la capacidad de prueba es un atributo de las buenas arquitecturas. El patrón Humble Object es un buen ejemplo, porque la separación de los comportamientos en partes comprobables y no comprobables a menudo define un límite arquitectónico. El límite Presentador/Vista es uno de estos límites, pero hay muchos otros.

PASARELAS DE BASE DE DATOS

Entre los interactuantes del caso de uso y la base de datos se encuentran las puertas de enlace de la base de datos.² Estas puertas de enlace son interfaces polimórficas que contienen métodos para cada operación de creación, lectura, actualización o eliminación que puede realizar la aplicación en la base de datos. Por ejemplo, si la aplicación necesita saber los apellidos de todos los usuarios que iniciaron sesión ayer, entonces la interfaz UserGateway tendrá un método llamado `getLastNamesofUsersWhoLoggedInAfter` que toma una fecha como argumento y devuelve una lista de apellidos.

Recuerde que no permitimos SQL en la capa de casos de uso; en su lugar, utilizamos interfaces de puerta de enlace que tienen métodos apropiados. Esas puertas de enlace se implementan mediante clases en la capa de base de datos. Esta implementación es el humilde objeto. Simplemente utiliza SQL, o cualquiera que sea la interfaz de la base de datos, para acceder a los datos requeridos por cada uno de los métodos. Los interactuantes, por el contrario, no son humildes porque encapsulan reglas de negocio específicas de la aplicación. Aunque no son humildes, esos interactuantes se pueden probar, porque las puertas de enlace se pueden reemplazar con stubs y dobles de prueba apropiados.

MAPEADORES DE DATOS

Volviendo al tema de las bases de datos, ¿a qué capa crees que pertenecen los ORM como Hibernate?

Primero, dejemos algo claro: no existe un mapeador relacional de objetos (ORM). La razón es simple: los objetos no son estructuras de datos. Al menos, no son estructuras de datos desde el punto de vista de sus usuarios. Los usuarios de un objeto no pueden ver los datos, ya que todos son privados. Esos usuarios solo ven los métodos públicos de ese objeto. Entonces, desde el punto de vista del usuario, un objeto es simplemente un conjunto de operaciones.

Una estructura de datos, por el contrario, es un conjunto de variables de datos públicos que no tienen ningún comportamiento implícito. Los ORM serían mejor llamados “mapeadores de datos”, porque cargan datos en estructuras de datos desde tablas de bases de datos relacionales.

¿Dónde deberían residir tales sistemas ORM? En la capa de base de datos, por supuesto.

De hecho, los ORM forman otro tipo de límite de objeto humilde entre las interfaces de la puerta de enlace y la base de datos.

SERVICIO DE OYENTES

¿Qué pasa con los servicios? Si su aplicación debe comunicarse con otros servicios, o si su aplicación proporciona un conjunto de servicios, ¿encontraremos el patrón de Objeto Humilde creando un límite de servicio?

¡Por supuesto! La aplicación cargará datos en estructuras de datos simples y luego pasará esas estructuras a través del límite a módulos que formatean adecuadamente los datos y los envían a servicios externos. En el lado de entrada, los oyentes del servicio recibirán datos de la interfaz del servicio y los formatearán en una estructura de datos simple que puede ser utilizada por la aplicación. Luego, esa estructura de datos cruza el límite del servicio.

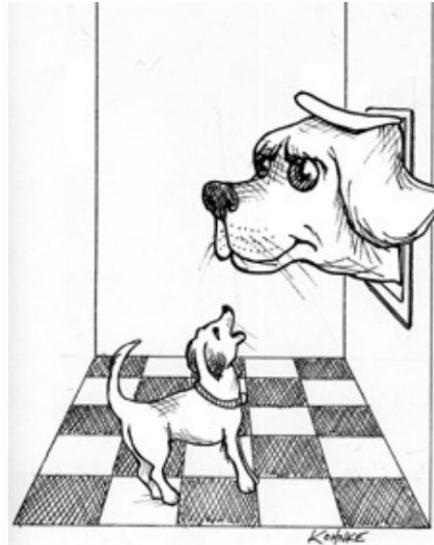
CONCLUSIÓN

En cada límite arquitectónico, es probable que encontraremos el patrón del Objeto Humilde acechando en algún lugar cercano. La comunicación a través de ese límite casi siempre implicará algún tipo de estructura de datos simple, y el límite frecuentemente dividirá algo que es difícil de probar de algo que es fácil de probar. El uso de este patrón en los límites arquitectónicos aumenta enormemente la capacidad de prueba de todo el sistema.

1. Patrones xUnit, Meszaros, Addison-Wesley, 2007, pág. 695.
2. Patrones de arquitectura de aplicaciones empresariales, Martin Fowler, et. al., Addison-Wesley, 2003, pág. 466.

24

LÍMITES PARCIALES



Los límites arquitectónicos completos son caros. Requieren interfaces de límites polimórficas recíprocas , estructuras de datos de entrada y salida y toda la gestión de dependencias necesaria para aislar los dos lados en componentes compilables e implementables de forma independiente.

Eso requiere mucho trabajo.

También es mucho trabajo de mantenimiento.

En muchas situaciones, un buen arquitecto podría juzgar que el gasto de tal límite es demasiado alto, pero aun así podría querer reservar un lugar para dicho límite en caso de que sea necesario más adelante.

Este tipo de diseño anticipatorio a menudo es mal visto por muchos en la comunidad ágil como una violación de YAGNI: "No lo vas a necesitar". arquitectos,

sin embargo, a veces miramos el problema y pensamos: "Sí, pero podría hacerlo". En ese caso, podrán implementar un límite parcial.

SALTAR EL ÚLTIMO PASO

Una forma de construir un límite parcial es hacer todo el trabajo necesario para crear componentes compilables e implementables de forma independiente y luego simplemente mantenerlos juntos en el mismo componente. Las interfaces recíprocas están ahí, las estructuras de datos de entrada/salida están ahí y todo está configurado, pero los compilamos e implementamos todos como un solo componente.

Obviamente, este tipo de límite parcial requiere la misma cantidad de código y trabajo de diseño preparatorio que un límite completo. Sin embargo, no requiere la administración de múltiples componentes. No hay seguimiento del número de versión ni carga de gestión de lanzamientos. Esa diferencia no debe tomarse a la ligera.

Esta fue la estrategia inicial detrás de FitNesse. El componente del servidor web de FitNesse fue diseñado para ser separable de la wiki y la parte de prueba de FitNesse. La idea era que podríamos querer crear otras aplicaciones basadas en web utilizando ese componente web. Al mismo tiempo, no queríamos que los usuarios tuvieran que descargar dos componentes. Recuerde que uno de nuestros objetivos de diseño era "descargar y listo". Nuestra intención era que los usuarios descargaran un archivo jar y lo ejecutaran sin tener que buscar otros archivos jar, resolver compatibilidades de versiones, etc.

La historia de FitNesse también señala uno de los peligros de este enfoque. Con el tiempo, cuando quedó claro que nunca habría necesidad de un componente web separado, la separación entre el componente web y el componente wiki comenzó a debilitarse. Las dependencias empezaron a cruzar la línea en la dirección equivocada. Hoy en día, sería una tarea ardua volver a separarlos.

LÍMITES UNIDIMENSIONALES

El límite arquitectónico completo utiliza interfaces de límites recíprocos para mantener el aislamiento en ambas direcciones. Mantener la separación en ambas direcciones es costoso tanto en la configuración inicial como en el mantenimiento continuo.

En la Figura 24.1 se muestra una estructura más simple que sirve para mantener el lugar para una posterior extensión a un límite completo . Ejemplifica el patrón de estrategia tradicional . Los clientes utilizan una interfaz ServiceBoundary y la implementan mediante clases ServiceImpl .

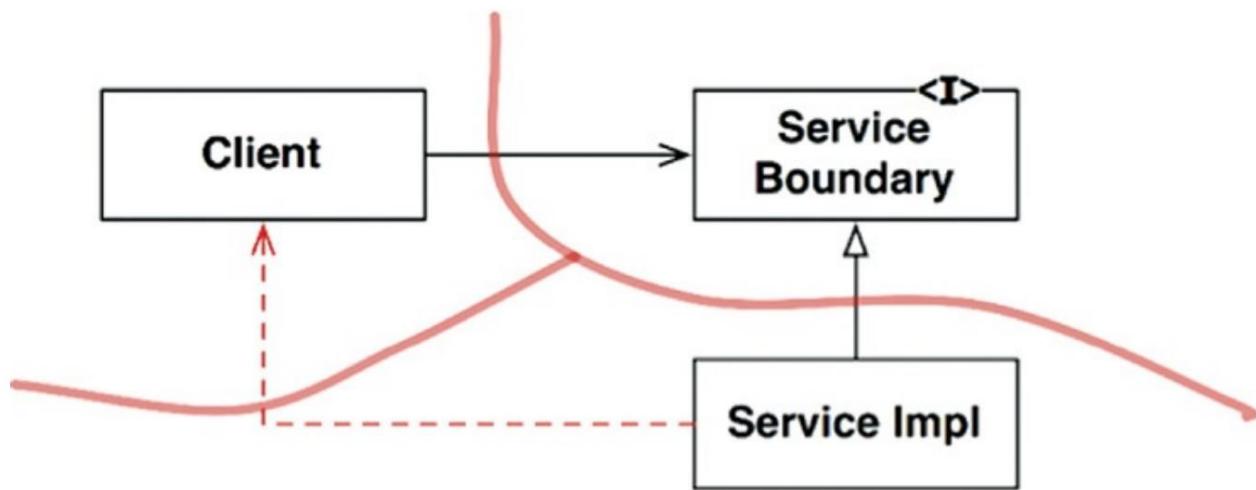


Figura 24.1 El patrón Estrategia

Debe quedar claro que esto prepara el escenario para una futura frontera arquitectónica. La inversión de dependencia necesaria está implementada en un intento de aislar al Cliente del ServiceImpl. También debería quedar claro que la separación puede degradarse con bastante rapidez, como lo muestra la desagradable flecha punteada en el diagrama. Sin interfaces recíprocas, nada impide este tipo de canal secundario excepto la diligencia y disciplina de los desarrolladores y arquitectos.

FACHADAS

Un límite aún más simple es el patrón Fachada , ilustrado en la Figura 24.2. En este caso, incluso se sacrifica la inversión de dependencia. El límite lo define simplemente la clase Facade , que enumera todos los servicios como métodos e implementa las llamadas de servicio a clases a las que se supone que el cliente no debe acceder.

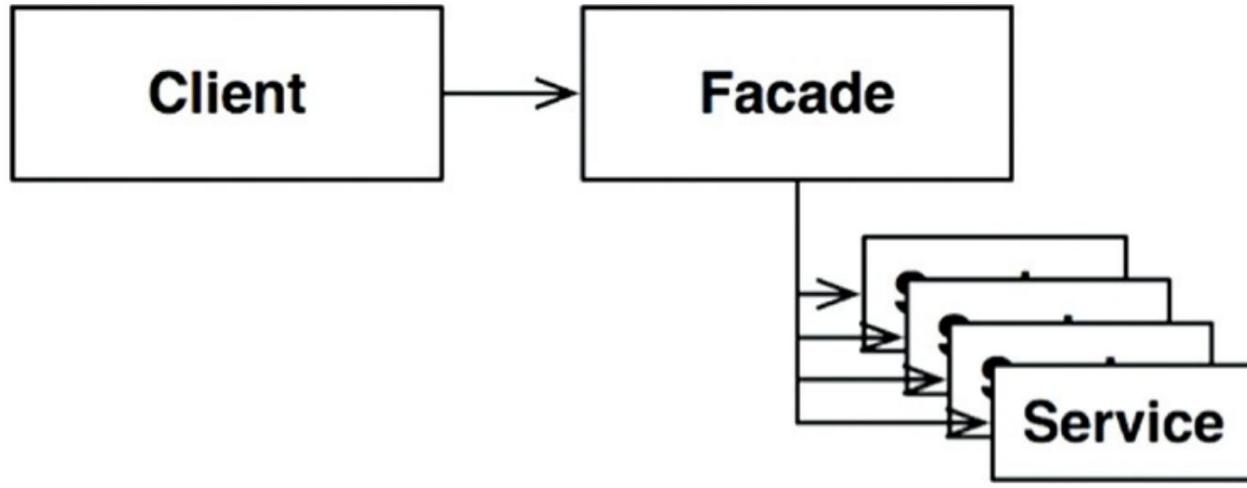


Figura 24.2 El patrón de fachada

Tenga en cuenta, sin embargo, que el Cliente tiene una dependencia transitiva de todas esas clases de servicios. En lenguajes estáticos, un cambio en el código fuente en una de las clases de Servicio obligará al Cliente a volver a compilar. Además, puedes imaginar lo fácil que es crear canales secundarios con esta estructura.

CONCLUSIÓN

Hemos visto tres formas sencillas de implementar parcialmente un límite arquitectónico. Por supuesto, hay muchos otros. Estas tres estrategias se ofrecen simplemente como ejemplos.

Cada uno de estos enfoques tiene su propio conjunto de costos y beneficios. Cada uno de ellos es apropiado, en ciertos contextos, como marcador de posición para un eventual límite pleno. Cada uno de ellos también puede degradarse si ese límite nunca se materializa.

Una de las funciones de un arquitecto es decidir dónde podría existir algún día un límite arquitectónico y si implementarlo total o parcialmente.

25

CAPAS Y LÍMITES



Es fácil pensar que los sistemas se componen de tres componentes: interfaz de usuario, reglas comerciales y base de datos. Para algunos sistemas simples, esto es suficiente. Sin embargo, para la mayoría de los sistemas, la cantidad de componentes es mayor.

Consideremos, por ejemplo, un sencillo juego de ordenador. Es fácil imaginar los tres componentes. La interfaz de usuario maneja todos los mensajes del jugador sobre las reglas del juego. Las reglas del juego almacenan el estado del juego en algún tipo de estructura de datos persistente.
¿Pero es eso todo lo que hay?

CAZAR LOS WUMPUS

Pongámosle un poco de carne a estos huesos. Supongamos que el juego es el venerable juego de aventuras Hunt the Wumpus de 1972. Este juego basado en texto utiliza comandos muy simples como IR AL ESTE y DISPARAR AL OESTE. El jugador ingresa un comando y la computadora responde con lo que el jugador ve, huele, oye y experimenta. El jugador está buscando un Wumpus en un sistema de cavernas y debe evitar trampas, pozos y otros peligros que lo acechan. Si estás interesado, las reglas del juego son fáciles de encontrar en la web.

Supongamos que mantendremos la interfaz de usuario basada en texto, pero la desacoplaremos de las reglas del juego para que nuestra versión pueda usar diferentes idiomas en diferentes mercados. Las reglas del juego se comunicarán con el componente de la interfaz de usuario mediante una API independiente del idioma, y la interfaz de usuario traducirá la API al lenguaje humano apropiado.

Si las dependencias del código fuente se administran adecuadamente, como se muestra en [la Figura 25.1](#), entonces cualquier número de componentes de la interfaz de usuario pueden reutilizar las mismas reglas del juego. Las reglas del juego no saben ni les importa qué lenguaje humano se está utilizando.

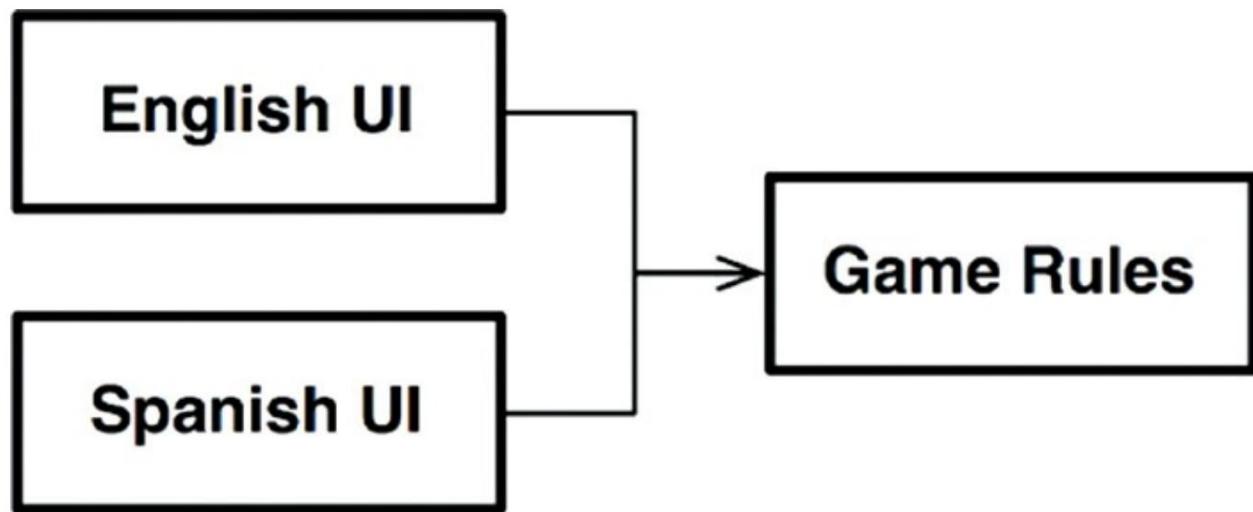


Figura 25.1 Cualquier cantidad de componentes de la interfaz de usuario pueden reutilizar las reglas del juego

Supongamos también que el estado del juego se mantiene en algún almacén persistente, tal vez en flash, o quizás en la nube, o tal vez solo en RAM. En cualquiera de esos casos, no queremos que las reglas del juego conozcan los detalles. Entonces, nuevamente, crearemos una API que las reglas del juego puedan usar para comunicarse con el componente de almacenamiento de datos.

No queremos que las reglas del juego sepan nada sobre los diferentes tipos de datos.

almacenamiento, por lo que las dependencias deben dirigirse adecuadamente siguiendo la Regla de Dependencia, como se muestra en [la Figura 25.2](#).

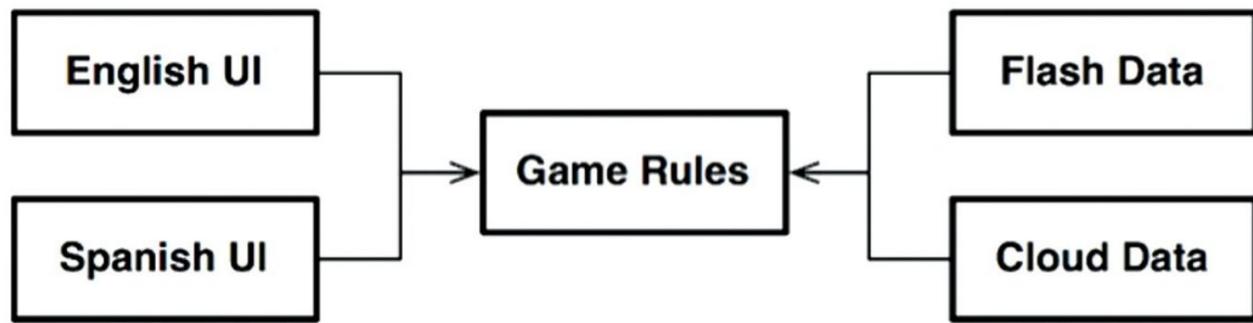


Figura 25.2 Siguiendo la regla de dependencia

¿ARQUITECTURA LIMPIA?

Debería quedar claro que podríamos aplicar fácilmente el enfoque de arquitectura limpia en este contexto, con todos los casos de uso, límites, entidades y estructuras de datos correspondientes. ¹ ¿Pero hemos encontrado realmente todos los límites arquitectónicos importantes?

Por ejemplo, el idioma no es el único eje de cambio para la interfaz de usuario. También es posible que deseemos variar el mecanismo mediante el cual comunicamos el texto. Por ejemplo, es posible que queramos utilizar una ventana de shell normal, mensajes de texto o una aplicación de chat. Hay muchas posibilidades diferentes.

Eso significa que existe un límite arquitectónico potencial definido por este eje de cambio. Quizás deberíamos construir una API que cruce esa frontera y aísle el lenguaje del mecanismo de comunicación; esa idea se ilustra en [la figura 25.3](#).

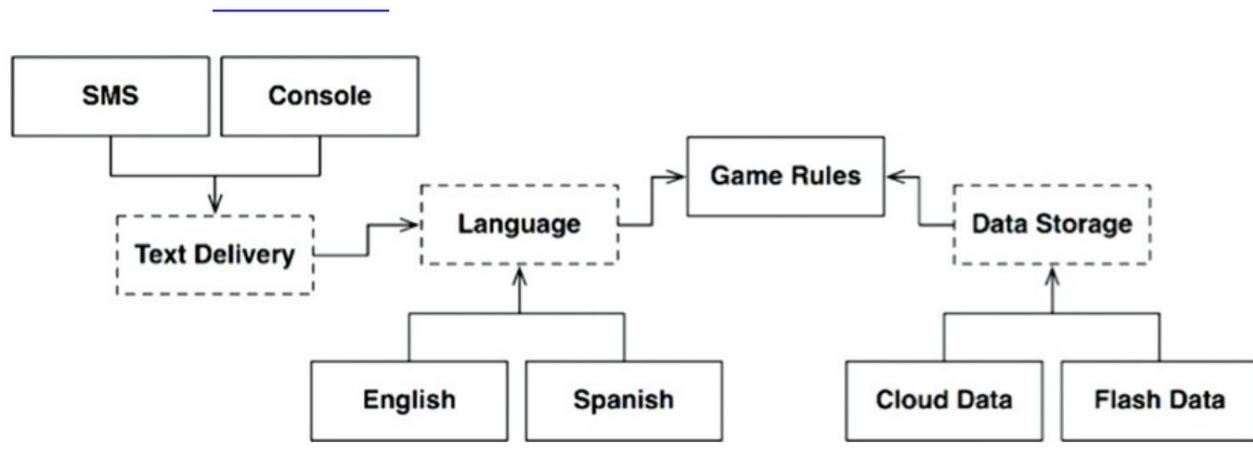


Figura 25.3 El diagrama revisado

El diagrama de [la figura 25.3](#) se ha vuelto un poco complicado, pero no debería contener sorpresas. Los contornos discontinuos indican componentes abstractos que definen una API implementada por los componentes encima o debajo de ellos. Por ejemplo, la API de idioma está implementada en inglés y español.

GameRules se comunica con Language a través de una API que GameRules define e implementa Language . Language se comunica con TextDelivery mediante una API que Language define pero TextDelivery implementa. La API la define y es propiedad del usuario, no del implementador.

Si miráramos dentro de GameRules, encontraríamos interfaces de límites polimórficas utilizadas por el código dentro de GameRules e implementadas por el código dentro del componente Idioma . También encontraríamos interfaces de límites polimórficas utilizadas por Language e implementadas mediante código dentro de GameRules.

Si miráramos dentro de Language, encontraríamos lo mismo: interfaces de límite polimórficas implementadas por el código dentro de TextDelivery, e interfaces de límite polimórficas utilizadas por TextDelivery e implementadas por Language.

En cada caso, la API definida por esas interfaces de límite es propiedad del componente ascendente.

Las variaciones, como English, SMS y CloudData, se proporcionan mediante interfaces polimórficas definidas en el componente API abstracto e implementadas por los componentes concretos que las sirven. Por ejemplo, esperaríamos que las interfaces polimórficas definidas en Idioma se implementaran en inglés y español.

Podemos simplificar este diagrama eliminando todas las variaciones y centrándonos solo en los componentes de la API. [La figura 25.4](#) muestra este diagrama.

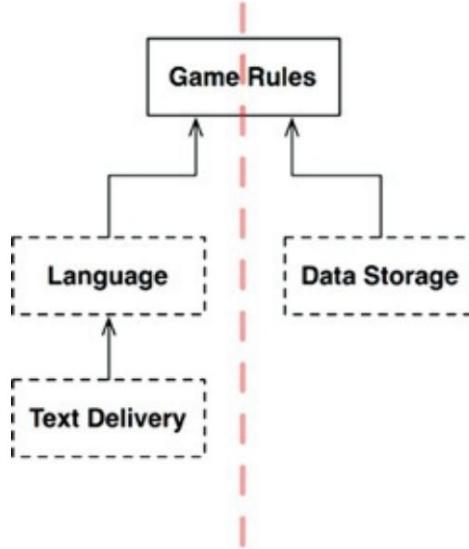


Figura 25.4 Diagrama simplificado

Observe que el diagrama está orientado en la [Figura 25.4](#) de modo que todas las flechas apunten hacia arriba. Esto coloca a GameRules en la cima. Esta orientación tiene sentido porque GameRules es el componente que contiene las políticas de más alto nivel.

Considere la dirección del flujo de información. Toda la entrada proviene del usuario a través del componente TextDelivery en la parte inferior izquierda. Esa información aumenta a través del componente Idioma y se traduce en comandos para GameRules. GameRules procesa la entrada del usuario y envía los datos apropiados a DataStorage en la parte inferior derecha.

Luego, GameRules envía la salida nuevamente a Idioma, que traduce la API nuevamente al idioma apropiado y luego entrega ese idioma al usuario a través de TextDelivery.

Esta organización divide efectivamente el flujo de datos en dos flujos. La secuencia de la izquierda se ocupa de la comunicación con el usuario y la secuencia de la derecha se ocupa de la persistencia de los datos. Ambas corrientes se encuentran² en la GameRules superior, que es el procesador final de los datos que pasan por ambas corrientes.

CRUZANDO LOS ARROYOS

¿Hay siempre dos flujos de datos como en este ejemplo? No, en absoluto. Imaginar

que nos gustaría jugar Hunt the Wumpus en la red con varios jugadores. En este caso, necesitaríamos un componente de red, como el que se muestra en [la Figura 25.5](#).

Esta organización divide el flujo de datos en tres flujos, todos controlados por GameRules.

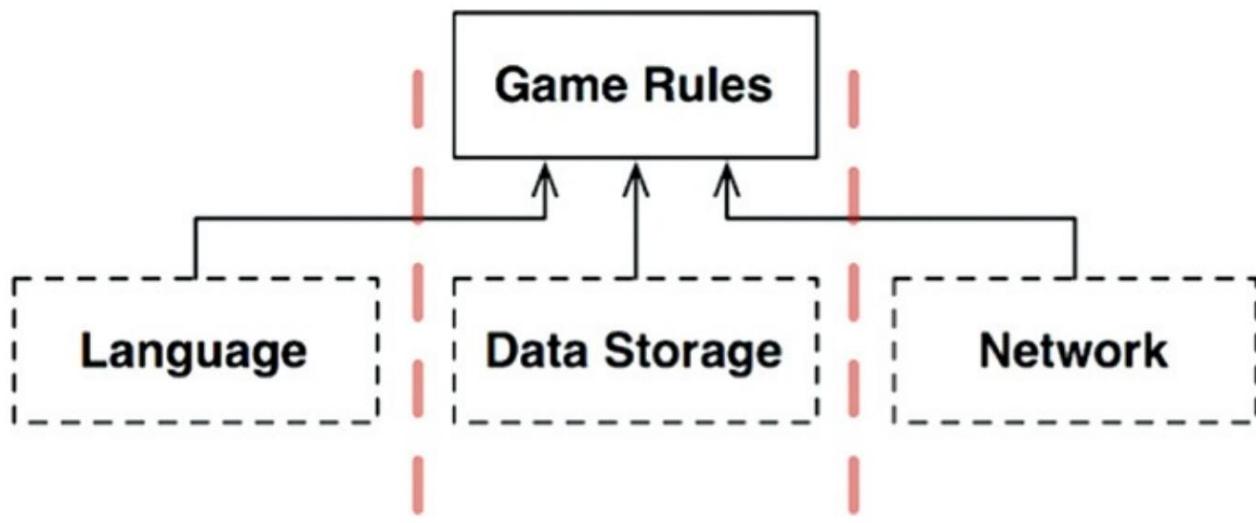


Figura 25.5 Agregar un componente de red

Entonces, a medida que los sistemas se vuelven más complejos, la estructura de los componentes puede dividirse en muchas de esas corrientes.

DIVIDIENDO LAS CORRIENTES

En este punto usted puede estar pensando que todas las corrientes finalmente se encuentran en la parte superior en un solo componente. ¡Si la vida fuera tan sencilla! La realidad, por supuesto, es mucho más compleja.

Considere el componente GameRules para Hunt the Wumpus. Parte de las reglas del juego tratan de la mecánica del mapa. Saben cómo están conectadas las cavernas y qué objetos se encuentran en cada caverna. Saben cómo mover al jugador de una caverna en otra y cómo determinar los eventos con los que debe lidiar el jugador.

Pero hay otro conjunto de políticas a un nivel aún más alto: políticas que conocen la salud del jugador y el costo o beneficio de un evento en particular. Estas políticas podrían hacer que el jugador pierda salud gradualmente o la gane al descubrir comida. La política de mecánicas de nivel inferior declararía eventos a este

política de nivel superior, como FoundFood o FellInPit. La política de nivel superior entonces gestionaría el estado del jugador (como se muestra en la [Figura 25.6](#)). Con el tiempo, esa política decidiría si el jugador gana o pierde.

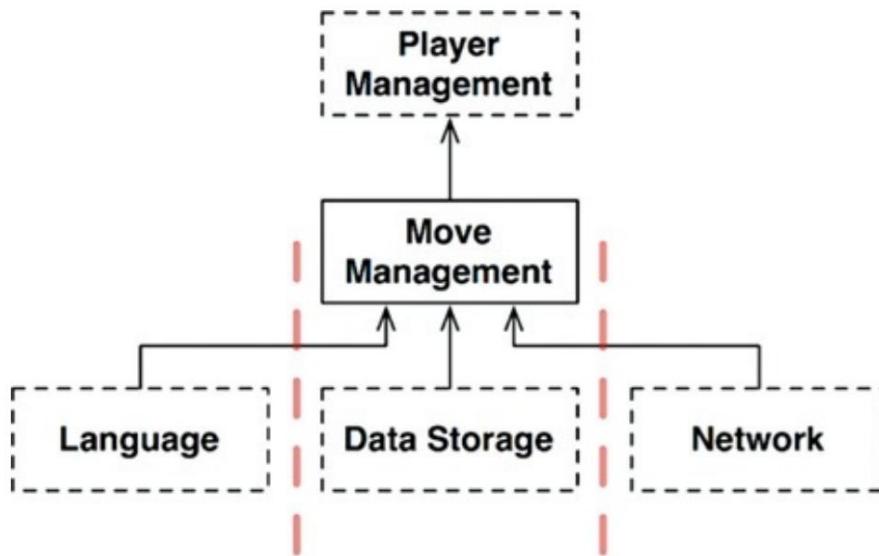


Figura 25.6 La política de nivel superior gestiona al jugador

¿Es este un límite arquitectónico? ¿Necesitamos una API que separe MoveManagement de PlayerManagement? Bueno, hagamos esto un poco más interesante y agreguemos microservicios.

Supongamos que tenemos una versión multijugador masiva de Hunt the Wumpus. MoveManagement se maneja localmente dentro de la computadora del jugador, pero PlayerManagement lo maneja un servidor. PlayerManagement ofrece una API de microservicio para todos los componentes de MoveManagement conectados .

El diagrama de [la Figura 25.7](#) representa este escenario de forma algo abreviada. Los elementos de la Red son un poco más complejos de lo que se muestran, pero probablemente aún puedas hacerte una idea. En este caso existe una frontera arquitectónica completa entre MoveManagement y PlayerManagement .

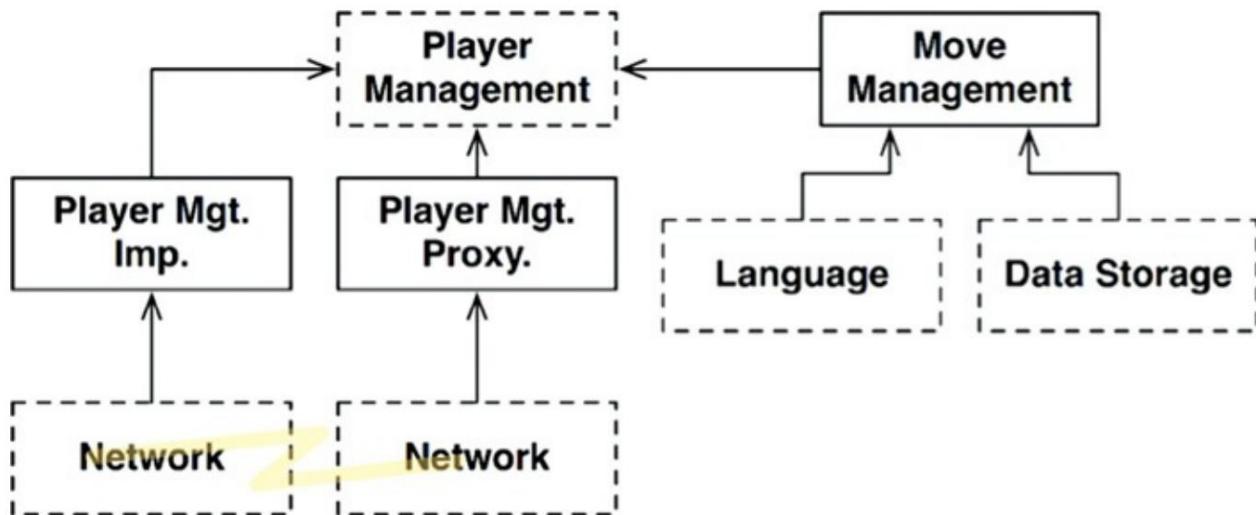


Figura 25.7 Agregar una API de microservicio

CONCLUSIÓN

¿Qué significa todo esto? ¿Por qué tomé este programa absurdamente simple, que podría implementarse en 200 líneas de Kornshell, y lo extrapolé con todos estos límites arquitectónicos locos?

Este ejemplo pretende mostrar que los límites arquitectónicos existen en todas partes.

Nosotros, como arquitectos, debemos tener cuidado de reconocer cuándo son necesarios. También debemos ser conscientes de que esos límites, cuando se implementan plenamente, resultan costosos. Al mismo tiempo, debemos reconocer que cuando se ignoran esos límites, resulta muy costoso agregarlos más adelante, incluso en presencia de conjuntos de pruebas integrales y disciplina de refactorización.

Entonces, ¿qué hacen los arquitectos? La respuesta es insatisfactoria. Por un lado, algunas personas muy inteligentes nos han dicho, a lo largo de los años, que no debemos anticipar la necesidad de la abstracción. Esta es la filosofía de YAGNI: "No lo vas a necesitar". Hay sabiduría en este mensaje, ya que un exceso de ingeniería es a menudo mucho peor que una falta de ingeniería. Por otro lado, cuando descubre que realmente necesita un límite arquitectónico donde no existe ninguno, los costos y riesgos pueden ser muy altos para agregar dicho límite.

Ahí lo tienes. Oh arquitecto de software, debes ver el futuro. Debes adivinar... intelligentemente.

Debe sopesar los costos y determinar dónde se encuentran los límites arquitectónicos, cuáles deben implementarse por completo y cuáles.

deberían implementarse parcialmente y cuáles deberían ignorarse.

Pero esta no es una decisión única. Al inicio de un proyecto no se decide simplemente qué límites implementar y cuáles ignorar. Más bien, miras.

Prestas atención a medida que el sistema evoluciona. Observa dónde pueden ser necesarios límites y luego observa atentamente el primer indicio de fricción porque esos límites no existen.

En ese punto, sopesas los costos de implementar esos límites versus el costo de ignorarlos y revisas esa decisión con frecuencia. Su objetivo es implementar los límites justo en el punto de inflexión donde el costo de implementarlos es menor que el costo de ignorarlos.

Se necesita una mirada atenta.

1. Debería quedar igualmente claro que no aplicaríamos el enfoque de arquitectura limpia a algo tan trivial como este juego. Después de todo, el programa completo probablemente pueda escribirse en 200 líneas de código o menos. En este caso, utilizamos un programa simple como proxy para un sistema mucho más grande con límites arquitectónicos importantes.
2. Si te confunde la dirección de las flechas, recuerda que apuntan en la dirección de la fuente. dependencias de código, no en la dirección del flujo de datos.
3. En el pasado, habríamos llamado a ese componente superior Transformación Central. Ver Guía Práctica al diseño de sistemas estructurados, 2^a ed., Meilir Page-Jones, 1988.

26

EL COMPONENTE PRINCIPAL



En todo sistema hay al menos un componente que crea, coordina y supervisa a los demás. A este componente lo llamo Principal.

EL ÚLTIMO DETALLE

El componente principal es el detalle último: la política de nivel más bajo. Es el punto de entrada inicial del sistema. Nada más que el sistema operativo depende de ello. Su trabajo es crear todas las Fábricas, Estrategias y otras instalaciones globales, y luego entregar el control a las partes abstractas de alto nivel del sistema.

Es en este componente principal donde un marco de inyección de dependencias debe inyectar las dependencias. Una vez que se inyectan en Main, Main debería distribuir esas dependencias normalmente, sin utilizar el marco.

Piense en Main como el más sucio de todos los componentes sucios.

Considere el siguiente componente principal de una versión reciente de Hunt the Wumpus. Observe cómo carga todas las cadenas que no queremos que conozca el cuerpo principal del código.

[Haga clic aquí para ver la imagen del código](#)

```
clase pública Principal implementa HtwMessageReceiver { juego
privado estático HuntTheWumpus; puntos de
impacto int estáticos privados = 10; cavernas
privadas estáticas finales List<String> = new ArrayList<>(); entornos privados
estáticos finales String[] = new String[] { "brillante", "húmedo", "seco",
"espeluznante",
"feo", "brumoso", "caliente", "frío", "con corrientes de aire", "terrible" };

cadena final estática privada [] formas = nueva cadena [] { "redondo",
"cuadrado",
"ovalado",

"irregular",
"largo",
"escarpado",
"áspero",
"alto",
"estrecho" };

private static final String[] cavernTypes = new String[] { "caverna", "habitación",
"cámara",

"catacumba",
"grieta",
"celda", "túnel",
```

```

"pasillo", "pasillo",
"extensión" };

private static final String[] adornos = new String[] { "oliendo a azufre", "con
grabados en las paredes",
"con el piso lleno de baches", "", "lleno
de basura", "salpicado de
guano", " con montones de
excrementos de Wumpus",
"con huesos esparcidos", "con un cadáver
en el suelo", "que parece vibrar", "que
se siente tapado", "que te llena de
pavor" };

```

Ahora aquí está la función principal . Observe cómo utiliza HtwFactory para crear el juego. Pasa en el nombre de la clase, htw.game.HuntTheWumpusFacade, porque esa clase es incluso más sucia que Main. Esto evita que los cambios en esa clase hagan que Main se vuelva a compilar/implementar.

[Haga clic aquí para ver la imagen del código](#)

```

public static void main(String[] args) lanza IOException { juego =
HtwFactory.makeGame("htw.game.HuntTheWumpusFacade", new Main());
crearMapa();
BufferedReader
br = nuevo BufferedReader
(nuevo InputStreamReader (System.in));
juego.makeRestCommand().execute();
mientras
(verdadero) { System.out.println(game.getPlayerCavern()); " "
System.out.println("Salud: " + hitPoints + game.getQuiver()); " flechas: " +
HuntTheWumpus.Command
c = juego.makeRestCommand(); System.out.println(">"); Comando
de cadena = br.readLine(); if
(command.equalsIgnoreCase("e")) c =
juego.makeMoveCommand(ESTE); else if
(command.equalsIgnoreCase("w")) c =
game.makeMoveCommand(OESTE); else if
(command.equalsIgnoreCase("n")) c =
game.makeMoveCommand(NORTE); else if
(command.equalsIgnoreCase("s")) c =
game.makeMoveCommand(SUR); de lo contrario si
(command.equalsIgnoreCase("r"))

```

```

c = juego.makeRestCommand();
else if (command.equalsIgnoreCase("sw")) c =
game.makeShootCommand(OESTE); else
if (command.equalsIgnoreCase("se")) c =
game.makeShootCommand(ESTE); else if
(command.equalsIgnoreCase("sn")) c =
game.makeShootCommand(NORTE); else
if (command.equalsIgnoreCase("ss")) c =
game.makeShootCommand(SUR); de lo
contrario, si (command.equalsIgnoreCase("q"))
regresa;

c.ejecutar(); } }

```

Observe también que main crea el flujo de entrada y contiene el bucle principal del juego, interpretando los comandos de entrada simples, pero luego difiere todo el procesamiento a otros componentes de nivel superior.

Finalmente, observe que main crea el mapa.

[Haga clic aquí para ver la imagen del código](#)

```

vacío estático privado createMap() { int
nCaverns = (int) (Math.random() * 30.0 + 10.0); while (nCaverns-- >
0) cavernas.add(makeName());

for (String caverna: cavernas) {tal
vezConnectCavern(caverna, NORTE); tal
vezConnectCavern(caverna, SUR); tal
vezConnectCavern(caverna, ESTE); tal
vezConnectCavern(caverna, OESTE); }

String playerCavern = cualquierCavern();
game.setPlayerCavern(jugadorCavern);
game.setWumpusCavern(cualquierOtro(playerCavern));
game.addBatCavern(cualquierOtro(playerCavern));
game.addBatCavern(cualquierOtro(playerCavern));
game.addBatCavern(cualquierOtro(playerCavern));

game.addPitCavern(cualquierOtro(jugadorCavern));
game.addPitCavern(cualquierOtro(jugadorCavern));
game.addPitCavern(cualquierOtro(jugadorCavern));

juego.setQuiver(5); }

```

```
// se eliminó mucho código... }
```

El punto es que Main es un módulo sucio de bajo nivel en el círculo más externo de la arquitectura limpia. Carga todo para el sistema de alto nivel y luego le entrega el control.

CONCLUSIÓN

Piense en Main como un complemento para la aplicación: un complemento que establece las condiciones y configuraciones iniciales, reúne todos los recursos externos y luego entrega el control a la política de alto nivel de la aplicación. Al ser un complemento, es posible tener muchos componentes principales , uno para cada configuración de su aplicación.

Por ejemplo, podría tener un complemento principal para Dev, otro para Test y otro más para Production. También podría tener un complemento principal para cada país en el que implemente, cada jurisdicción o cada cliente.

Cuando piensas en Main como un componente de complemento, ubicado detrás de un límite arquitectónico, el problema de la configuración se vuelve mucho más fácil de resolver.

27

SERVICIOS: GRANDES Y PEQUEÑOS



Las “arquitecturas” orientadas a servicios y las “arquitecturas” de microservicios se han vuelto muy populares últimamente. Las razones de su popularidad actual incluyen las siguientes:

- Los servicios parecen estar fuertemente desvinculados unos de otros. Como veremos, esto es sólo parcialmente cierto.
- Los servicios parecen respaldar la independencia de desarrollo e implementación.
Nuevamente, como veremos, esto es sólo parcialmente cierto.

¿ARQUITECTURA DEL SERVICIO?

Primero, consideremos la noción de que el uso de servicios, por su naturaleza, es una arquitectura. Esto es evidentemente falso. La arquitectura de un sistema está definida por límites que separan la política de alto nivel de los detalles de bajo nivel y siguen la regla de dependencia. Los servicios que simplemente separan los comportamientos de las aplicaciones son poco más que costosas llamadas a funciones y no son necesariamente significativos desde el punto de vista arquitectónico.

Esto no quiere decir que todos los servicios deban ser arquitectónicamente significativos. A menudo existen beneficios sustanciales al crear servicios que separan la funcionalidad entre procesos y plataformas, ya sea que obedezcan la regla de dependencia o no. Lo que pasa es que los servicios, en sí mismos, no definen una arquitectura.

Una analogía útil es la organización de funciones. La arquitectura de un sistema monolítico o basado en componentes se define mediante ciertas llamadas a funciones que cruzan los límites arquitectónicos y siguen la regla de dependencia. Sin embargo, muchas otras funciones en esos sistemas simplemente separan un comportamiento de otro y no son significativas desde el punto de vista arquitectónico.

Lo mismo ocurre con los servicios. Después de todo, los servicios son solo llamadas a funciones a través de los límites del proceso y/o la plataforma. Algunos de esos servicios son arquitectónicamente importantes y otros no. Nuestro interés, en este capítulo, está en el primero.

¿BENEFICIOS DEL SERVICIO?

El signo de interrogación en el encabezado anterior indica que esta sección desafiará la ortodoxia popular actual de la arquitectura de servicios. Abordemos los beneficios uno por uno.

La falacia del desacoplamiento

Uno de los grandes supuestos beneficios de dividir un sistema en servicios es que los servicios están fuertemente desacoplados entre sí. Después de todo, cada servicio se ejecuta en un proceso diferente, o incluso en un procesador diferente; por lo tanto, esos servicios no tienen acceso a las variables de cada uno. Es más, la interfaz de cada servicio debe estar bien definida.

Ciertamente hay algo de verdad en esto, pero no mucha verdad. Sí, los servicios están desacoplados a nivel de variables individuales. Sin embargo, todavía se pueden acoplar.

por recursos compartidos dentro de un procesador o en la red. Es más, están fuertemente vinculados por los datos que comparten.

Por ejemplo, si se agrega un nuevo campo a un registro de datos que se pasa entre servicios, entonces se deben cambiar todos los servicios que operan en el nuevo campo. Los servicios también deben estar totalmente de acuerdo sobre la interpretación de los datos en ese campo. Por tanto, esos servicios están fuertemente acoplados al registro de datos y, por tanto, indirectamente acoplados entre sí.

En cuanto a que las interfaces estén bien definidas, eso es ciertamente cierto, pero no lo es menos en el caso de las funciones. Las interfaces de servicio no son más formales, más rigurosas ni mejor definidas que las interfaces de funciones. Es evidente, entonces, que este beneficio es algo así como una ilusión.

LA FALACIA DEL DESARROLLO INDEPENDIENTE Y DESPLIEGUE

Otro de los supuestos beneficios de los servicios es que pueden ser propiedad de un equipo dedicado y ser operados por ellos. Ese equipo puede ser responsable de escribir, mantener y operar el servicio como parte de una estrategia de operaciones de desarrollo. Se supone que esta independencia de desarrollo y despliegue es escalable. Se cree que se pueden crear grandes sistemas empresariales a partir de docenas, cientos o incluso miles de servicios que se pueden desarrollar e implementar de forma independiente.

El desarrollo, mantenimiento y operación del sistema se pueden dividir entre un número similar de equipos independientes.

Hay algo de verdad en esta creencia, pero sólo algo. En primer lugar, la historia ha demostrado que los sistemas empresariales grandes pueden construirse a partir de monolitos y sistemas basados en componentes, así como sistemas basados en servicios. Por tanto, los servicios no son la única opción para construir sistemas escalables.

En segundo lugar, la falacia del desacoplamiento significa que los servicios no siempre pueden desarrollarse, desplegarse y operarse de forma independiente. En la medida en que estén acoplados por datos o comportamiento, se debe coordinar el desarrollo, despliegue y operación.

EL PROBLEMA DEL GATITO

Como ejemplo de estas dos falacias, veamos nuevamente nuestro sistema de agregación de taxis. Recuerde, este sistema conoce muchos proveedores de taxis en una ciudad determinada y permite a los clientes solicitar viajes. Supongamos que los clientes seleccionan los taxis basándose en una serie de criterios, como la hora de recogida, el coste, el lujo y la experiencia del conductor.

Queríamos que nuestro sistema fuera escalable, por lo que decidimos construirlo a partir de muchos pequeños microservicios. Subdividimos a nuestro personal de desarrollo en muchos equipos pequeños, cada uno de los cuales es responsable de desarrollar, mantener y operar una pequeña cantidad de servicios. ¹ correspondientemente

El diagrama de [la Figura 27.1](#) muestra cómo nuestros arquitectos ficticios organizaron servicios para implementar esta aplicación. El servicio TaxiUI se ocupa de los clientes, que utilizan dispositivos móviles para solicitar taxis. El servicio TaxiFinder examina los inventarios de los distintos TaxiSupliers y determina qué taxis son posibles candidatos para el usuario. Los deposita en un registro de datos a corto plazo adjunto a ese usuario. El servicio TaxiSelector toma en consideración los criterios del usuario en cuanto a coste, tiempo, lujo, etc., y elige el taxi adecuado entre los candidatos. Entrega ese taxi al servicio TaxiDispatcher, que solicita el taxi adecuado.

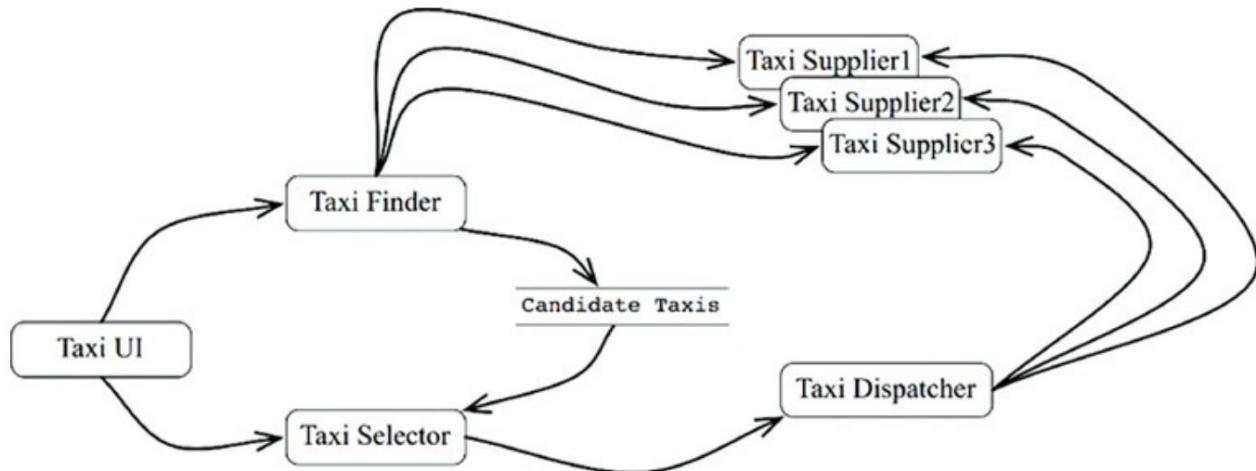


Figura 27.1 Servicios dispuestos para implementar el sistema agregador de taxis

Supongamos ahora que este sistema lleva más de un año en funcionamiento.

Nuestro personal de desarrolladores ha estado felizmente desarrollando nuevas funciones mientras mantiene y opera todos estos servicios.

Un día brillante y alegre, el departamento de marketing mantiene una reunión con el

Equipo de desarrollo. En este encuentro anuncian sus planes de ofrecer un servicio de entrega de gatitos a la ciudad. Los usuarios pueden solicitar la entrega de gatitos en su domicilio o en su lugar de trabajo.

La empresa instalará varios puntos de recogida de gatitos en toda la ciudad. Cuando se realiza un pedido de gatitos, se seleccionará un taxi cercano para recoger un gatito en uno de esos puntos de recogida y luego entregarlo en la dirección correspondiente.

Uno de los proveedores de taxis ha aceptado participar en este programa. Es probable que otros sigan sus pasos. Otros más pueden declinar.

Por supuesto, algunos conductores pueden ser alérgicos a los gatos, por lo que esos conductores nunca deben ser seleccionados para este servicio. Además, algunos clientes sin duda tendrán alergias similares, por lo que un vehículo que se haya utilizado para entregar gatitos en los últimos 3 días no debe seleccionarse para los clientes que declaran dichas alergias.

Mire ese diagrama de servicios. ¿Cuántos de esos servicios tendrán que cambiar para implementar esta característica? Todos ellos. Claramente, el desarrollo y la implementación de la función Kitty tendrán que coordinarse con mucho cuidado.

En otras palabras, todos los servicios están acoplados y no pueden desarrollarse, implementarse ni mantenerse de forma independiente.

Éste es el problema de las preocupaciones transversales. Todo sistema de software debe afrontar este problema, esté orientado a servicios o no. Las descomposiciones funcionales, del tipo que se muestra en el diagrama de servicios de la [figura 27.1](#), son muy vulnerables a nuevas características que atraviesan todos esos comportamientos funcionales.

OBJETOS AL RESCATE

¿Cómo hubiéramos solucionado este problema en una arquitectura basada en componentes? Una consideración cuidadosa de los principios de diseño SOLID nos habría impulsado a crear un conjunto de clases que podrían extenderse polimórficamente para manejar nuevas características.

El diagrama de [la Figura 27.2](#) muestra la estrategia. Las clases en este diagrama corresponden aproximadamente a los servicios que se muestran en [la Figura 27.1](#). Sin embargo, tenga en cuenta los límites. Tenga en cuenta también que las dependencias siguen la regla de dependencia.

Gran parte de la lógica de los servicios originales se conserva dentro de las clases base del modelo de objetos. Sin embargo, esa parte de la lógica que era específica de las atracciones se extrajo en un componente de atracciones . La nueva característica para gatitos se ha colocado en un componente Gatitos . Estos dos componentes anulan las clases base abstractas en los componentes originales utilizando un patrón como Método de plantilla o Estrategia.

Tenga en cuenta nuevamente que los dos nuevos componentes, atracciones y gatitos, siguen la regla de dependencia. Tenga en cuenta también que las clases que implementan esas características son creadas por fábricas bajo el control de la interfaz de usuario.

Claramente, en este esquema, cuando se implementa la función Kitty, TaxiUI debe cambiar. Pero no es necesario cambiar nada más. Más bien, se agrega un nuevo archivo jar, Gem o DLL al sistema y se carga dinámicamente en tiempo de ejecución.

Por lo tanto, la función Kitty está desacoplada y se puede desarrollar y desplegar de forma independiente.

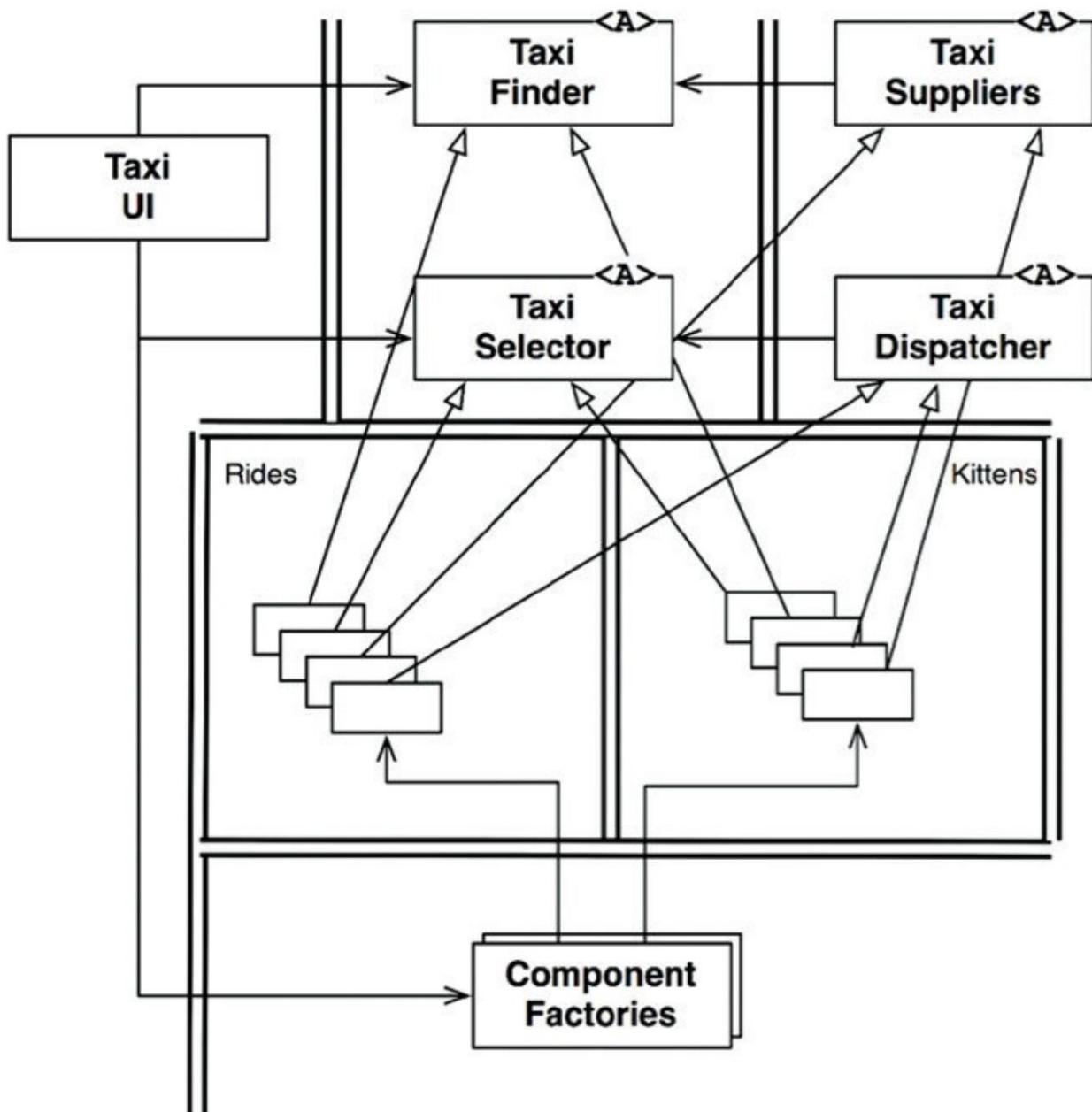


Figura 27.2 Uso de un enfoque orientado a objetos para abordar cuestiones transversales

SERVICIOS BASADOS EN COMPONENTES

La pregunta obvia es: ¿podemos hacer eso con los servicios? Y la respuesta es, por supuesto: ¡Sí! Los servicios no tienen por qué ser pequeños monolitos. En cambio, los servicios pueden diseñarse utilizando los principios SOLID y recibir una estructura de componentes para que se les puedan agregar nuevos componentes sin cambiar los componentes existentes dentro del servicio.

Piense en un servicio en Java como un conjunto de clases abstractas en uno o más archivos jar. Piense en cada característica nueva o extensión de característica como otro archivo jar que contiene clases que amplían las clases abstractas en los primeros archivos jar. Entonces, implementar una nueva característica no es una cuestión de volver a implementar los servicios, sino más bien de simplemente agregar los nuevos archivos jar a las rutas de carga de esos servicios. En otras palabras, agregar nuevas funciones se ajusta al principio abierto-cerrado.

El diagrama de servicio en [la Figura 27.3](#) muestra la estructura. Los servicios siguen existiendo como antes, pero cada uno tiene su propio diseño de componentes internos, lo que permite agregar nuevas funciones como nuevas clases derivadas. Esas clases de derivados viven dentro de sus propios componentes.

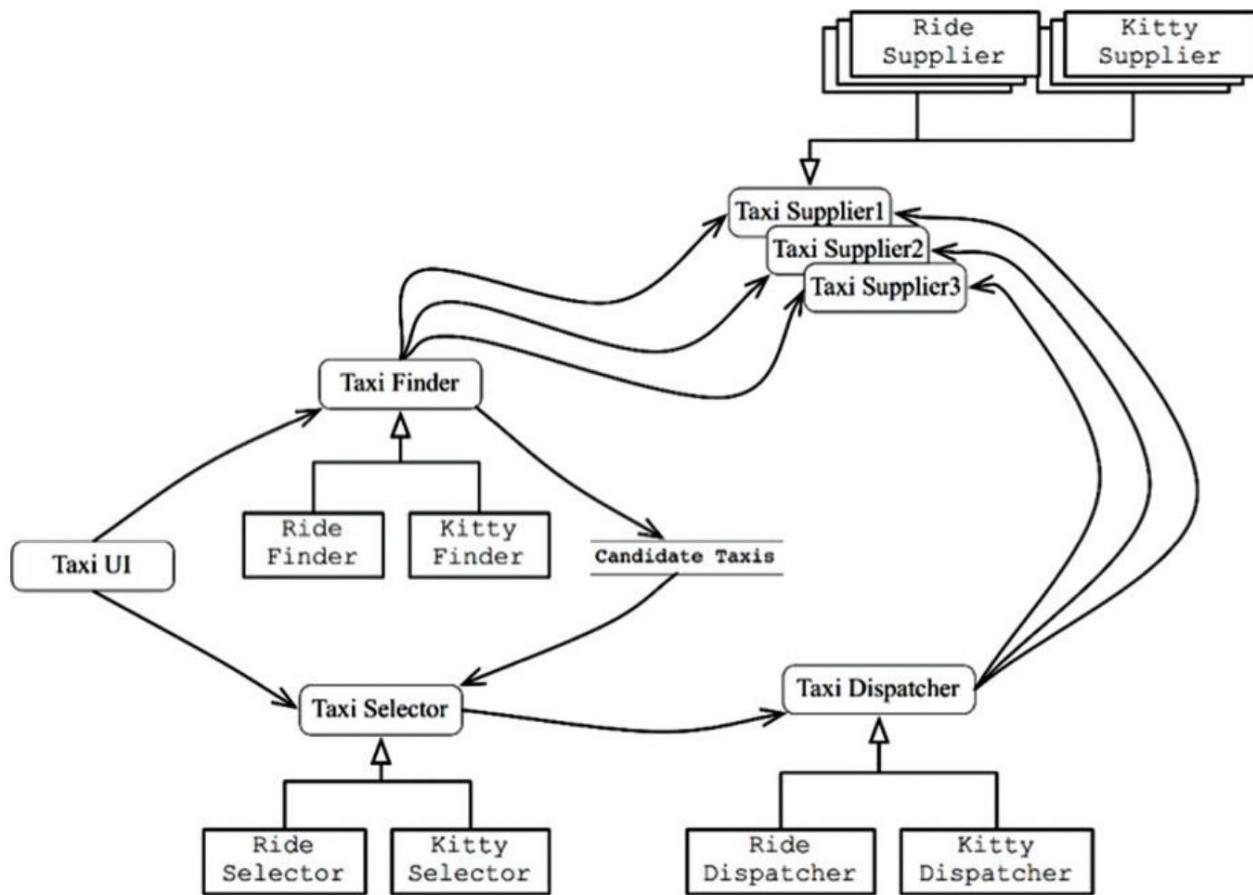


Figura 27.3 Cada servicio tiene su propio diseño de componentes internos, lo que permite agregar nuevas características como nuevas clases derivadas.

PREOCUPACIONES TRANSVERSALES

Lo que hemos aprendido es que los límites arquitectónicos no se encuentran entre los servicios. Más bien, esos límites atraviesan los servicios, dividiéndolos en componentes.

Para abordar las preocupaciones transversales que enfrentan todos los sistemas importantes, los servicios deben diseñarse con arquitecturas de componentes internos que sigan la regla de dependencia, como se muestra en el [diagrama de la Figura 27.4](#). Esos servicios no definen los límites arquitectónicos del sistema; en cambio, lo hacen los componentes dentro de los servicios.

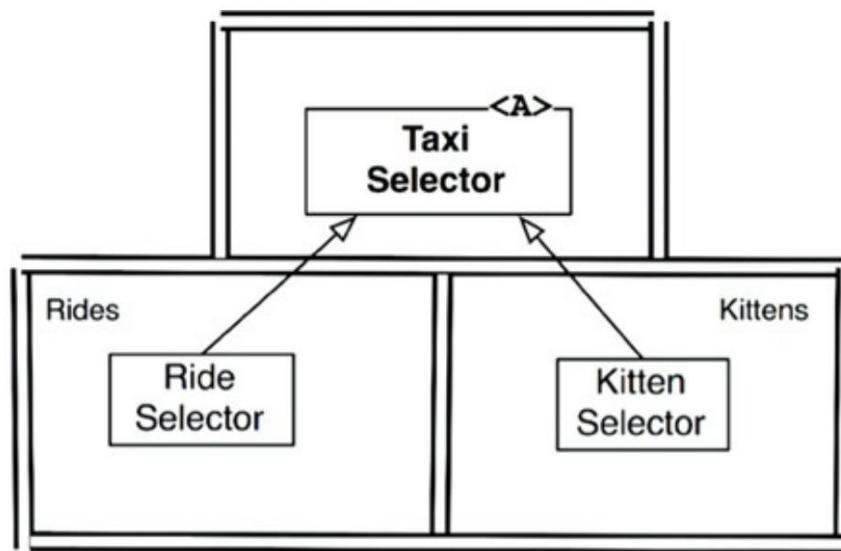


Figura 27.4 Los servicios deben diseñarse con arquitecturas de componentes internos que sigan la regla de dependencia

CONCLUSIÓN

Por muy útiles que sean los servicios para la escalabilidad y la capacidad de desarrollo de un sistema, no son, en sí mismos, elementos arquitectónicamente significativos. La arquitectura de un sistema está definida por los límites trazados dentro de ese sistema y por las dependencias que cruzan esos límites. Esa arquitectura no está definida por los mecanismos físicos mediante los cuales los elementos se comunican y ejecutar.

Un servicio puede ser un componente único, completamente rodeado por un límite arquitectónico. Alternativamente, un servicio podría estar compuesto por varios componentes separados por límites arquitectónicos. En raro ² casos, clientes y

Los servicios pueden estar tan acoplados que no tengan importancia arquitectónica alguna.

1. Por lo tanto, la cantidad de microservicios será aproximadamente igual a la cantidad de programadores.
2. Esperamos que sean raros. Desgraciadamente, la experiencia sugiere lo contrario.

28

EL LÍMITE DE LA PRUEBA



Sí, es cierto: las pruebas son parte del sistema y participan en la arquitectura al igual que cualquier otra parte del sistema. En cierto modo, esa participación es bastante normal. En otros sentidos, puede ser bastante único.

PRUEBAS COMO COMPONENTES DEL SISTEMA

Existe una gran confusión acerca de las pruebas. ¿Son parte del sistema? ¿Están separados del sistema? ¿Qué tipos de pruebas existen? ¿Las pruebas unitarias y las pruebas de integración son cosas diferentes? ¿Qué pasa con las pruebas de aceptación, pruebas funcionales, pruebas de pepino, pruebas TDD, pruebas BDD, pruebas de componentes, etc.?

No es el papel de este libro involucrarse en ese debate en particular y, afortunadamente, no es necesario. Desde un punto de vista arquitectónico, todas las pruebas son iguales. Ya sean las pequeñas pruebas creadas por TDD o las pruebas grandes de FitNesse, Cucumber, SpecFlow o JBehave, son arquitectónicamente equivalentes.

Las pruebas, por su propia naturaleza, siguen la Regla de Dependencia; son muy detallados y concretos; y siempre dependen del código que se está probando. De hecho, puedes pensar en las pruebas como el círculo más externo de la arquitectura.

Nada dentro del sistema depende de las pruebas, y las pruebas siempre dependen internamente de los componentes del sistema.

Las pruebas también se pueden implementar de forma independiente. De hecho, la mayoría de las veces se implementan en sistemas de prueba, en lugar de en sistemas de producción. Por lo tanto, incluso en sistemas donde la implementación independiente no es necesaria, las pruebas se seguirán implementando de forma independiente.

Las pruebas son el componente del sistema más aislado. No son necesarios para el funcionamiento del sistema. Ningún usuario depende de ellos. Su función es apoyar el desarrollo, no la operación. Y, sin embargo, no son menos que cualquier otro componente del sistema. De hecho, en muchos sentidos representan el modelo que todos los demás componentes del sistema deberían seguir.

DISEÑO PARA LA TESTABILIDAD

El aislamiento extremo de las pruebas, combinado con el hecho de que no suelen implementarse, a menudo hace que los desarrolladores piensen que las pruebas quedan fuera del diseño del sistema. Este es un punto de vista catastrófico. Las pruebas que no están bien integradas en el diseño del sistema tienden a ser frágiles y hacen que el sistema sea rígido y difícil de cambiar.

La cuestión, por supuesto, es el acoplamiento. Las pruebas que están fuertemente acopladas al sistema deben cambiar junto con el sistema. Incluso el cambio más trivial en un componente del sistema puede provocar que muchas pruebas acopladas fallen o requieran cambios.

Esta situación puede volverse aguda. Los cambios en los componentes comunes del sistema pueden provocar que cientos, o incluso miles, de pruebas fallen. Esto se conoce como el problema de las pruebas frágiles.

No es difícil ver cómo puede suceder esto. Imagine, por ejemplo, un conjunto de pruebas que utilizan la GUI para verificar las reglas comerciales. Dichas pruebas pueden comenzar en la pantalla de inicio de sesión y luego navegar a través de la estructura de la página hasta que puedan verificar reglas comerciales particulares. Cualquier cambio en la página de inicio de sesión o en la estructura de navegación puede provocar la falla de una enorme cantidad de pruebas.

Las pruebas frágiles a menudo tienen el efecto perverso de volver rígido el sistema. Cuando los desarrolladores se dan cuenta de que cambios simples en el sistema pueden causar fallas masivas en las pruebas, es posible que se resistan a realizar esos cambios. Por ejemplo, imagine la conversación entre el equipo de desarrollo y un equipo de marketing que solicita un cambio simple en la estructura de navegación de la página que provocará que se interrumpan 1000 pruebas.

La solución es diseñar para la capacidad de prueba. La primera regla del diseño de software, ya sea por capacidad de prueba o por cualquier otro motivo, es siempre la misma: no dependa de cosas volátiles. Las GUI son volátiles. Los conjuntos de pruebas que operan el sistema a través de la GUI deben ser frágiles. Por lo tanto, diseñe el sistema y las pruebas de modo que las reglas comerciales puedan probarse sin utilizar la GUI.

LA API DE PRUEBA

La forma de lograr este objetivo es crear una API específica que las pruebas puedan utilizar para verificar todas las reglas comerciales. Esta API debe tener superpoderes que permitan que las pruebas eviten restricciones de seguridad, eviten recursos costosos (como bases de datos) y fuercen al sistema a estados comprobables particulares. Esta API será un superconjunto del conjunto de interactores y adaptadores de interfaz que utiliza la interfaz de usuario.

El propósito de la API de prueba es desacoplar las pruebas de la aplicación. Este desacoplamiento abarca más que simplemente separar las pruebas de la interfaz de usuario: el objetivo es desacoplar la estructura de las pruebas de la estructura de la aplicación.

ACOPLAMIENTO ESTRUCTURAL

El acoplamiento estructural es una de las formas más fuertes e insidiosas de acoplamiento de prueba. Imagine un conjunto de pruebas que tenga una clase de prueba para cada clase de producción y un conjunto de métodos de prueba para cada método de producción. Un conjunto de pruebas de este tipo está profundamente acoplado a la estructura de la aplicación.

Cuando uno de esos métodos o clases de producción cambia, también deben cambiar una gran cantidad de pruebas. En consecuencia, las pruebas son frágiles y hacen rígido el código de producción.

La función de la API de prueba es ocultar la estructura de la aplicación a las pruebas. Esto permite refactorizar y evolucionar el código de producción de manera que no afecte las pruebas. También permite refactorizar y evolucionar las pruebas de manera que no afecten el código de producción.

Esta separación de la evolución es necesaria porque a medida que pasa el tiempo las pruebas tienden a volverse cada vez más concretas y específicas. Por el contrario, el código de producción tiende a volverse cada vez más abstracto y general. Un fuerte acoplamiento estructural impide (o al menos impide) esta necesaria evolución e impide que el código de producción sea tan general y flexible como podría ser.

SEGURIDAD

Los superpoderes de la API de prueba podrían ser peligrosos si se implementaran en sistemas de producción. Si esto le preocupa, entonces la API de prueba y las partes peligrosas de su implementación deben mantenerse en un componente separado que se pueda implementar de forma independiente.

CONCLUSIÓN

Las pruebas no están fuera del sistema; más bien, son partes del sistema que deben estar bien diseñadas para que proporcionen los beneficios deseados de estabilidad y regresión.

Las pruebas que no están diseñadas como parte del sistema tienden a ser frágiles y difíciles de mantener. Estas pruebas a menudo terminan en el suelo de la sala de mantenimiento, desecharadas porque son demasiado difíciles de mantener.

29

ARQUITECTURA INTEGRADA LIMPIA



Por James Grenning

Hace un tiempo leí un artículo titulado "La creciente importancia de sostener [1](#) Software para el Departamento de Defensa" – en el blog de Doug Schmidt. Doug hizo lo siguiente afirmación:

"Aunque el software no se desgasta, el firmware y el hardware se vuelven obsoletos, por lo que es necesario realizar modificaciones en el software".

Fue un momento clarificador para mí. Doug mencionó dos términos que habría pensado que eran obvios, pero tal vez no. El software es algo que puede tener una larga vida útil, pero el firmware quedará obsoleto a medida que el hardware evolucione. Si ha dedicado algún tiempo al desarrollo de sistemas integrados, conoce el hardware.

está en continua evolución y mejora. Al mismo tiempo, se añaden funciones al nuevo “software” y su complejidad crece continuamente.

Me gustaría agregar a la declaración de Doug:

Aunque el software no se desgasta, puede destruirse desde dentro debido a dependencias no gestionadas del firmware y el hardware.

No es raro que al software integrado se le niegue una vida potencialmente larga debido a que está infectado con dependencias del hardware.

Me gusta la definición de firmware de Doug, pero veamos qué otras definiciones existen. Encontré estas alternativas:

- “El firmware se guarda en dispositivos de memoria no volátil como ROM, EPROM o memoria flash”. (<https://en.wikipedia.org/wiki/Firmware>) • “El firmware es un programa de software o un conjunto de instrucciones programadas en un dispositivo de hardware”. (<https://techterms.com/definition/firmware>) • “El firmware es software integrado en una pieza de hardware”. (<https://www.lifewire.com/what-is-firmware-2625881>) • Firmware es “software (programas o datos) que se ha escrito en una memoria de sólo lectura (ROM)”. (<http://www.webopedia.com/TERM/F/firmware.html>)
-

La declaración de Doug me hace darme cuenta de que estas definiciones aceptadas de firmware son incorrectas o al menos obsoletas. El firmware no significa que el código viva en la ROM. No es firmware por el lugar donde está almacenado; más bien, es firmware por lo que depende de él y lo difícil que es cambiarlo a medida que evoluciona el hardware. El hardware evoluciona (haga una pausa y mire su teléfono en busca de evidencia), por lo que debemos estructurar nuestro código integrado teniendo esa realidad en mente.

No tengo nada en contra del firmware ni de los ingenieros de firmware (se sabe que yo mismo escribo algunos firmware). Pero lo que realmente necesitamos es menos firmware y más software. De hecho, ¡me decepciona que los ingenieros de firmware escriban tanto firmware!

¡Los ingenieros no integrados también escriben firmware! Ustedes, los desarrolladores no integrados, esencialmente escriben firmware cada vez que incluyen SQL en su código o cuando distribuyen dependencias de plataforma en todo su código. Los desarrolladores de aplicaciones de Android escriben firmware cuando no separan su lógica empresarial de la API de Android.

He estado involucrado en muchos esfuerzos donde la línea entre el código del producto (el software) y el código que interactúa con el hardware del producto (el firmware) es borrosa hasta el punto de no existir. Por ejemplo, a finales de la década de 1990 me divertí ayudando a rediseñar un subsistema de comunicaciones que estaba pasando de la multiplexación por división de tiempo (TDM) a la voz sobre IP (VOIP).

VOIP es la forma en que se hacen las cosas ahora, pero TDM se consideró lo último en las décadas de 1950 y 1960, y se implementó ampliamente en las décadas de 1980 y 1990.

Cada vez que teníamos una pregunta para el ingeniero de sistemas sobre cómo debería reaccionar una llamada ante una situación determinada, él desaparecía y poco después emergía con una respuesta muy detallada. “¿De dónde sacó esa respuesta?” preguntamos. “Del código del producto actual”, respondía. ¡El código heredado enredado era la especificación del nuevo producto! La implementación existente no tenía separación entre TDM y la lógica empresarial de realizar llamadas. Todo el producto dependía del hardware/tecnología de arriba a abajo y no podía desenredarse.

Básicamente, todo el producto se había convertido en firmware.

Considere otro ejemplo: los mensajes de comando llegan a este sistema a través del puerto serie. Como era de esperar, hay un procesador/despachador de mensajes. El procesador de mensajes conoce el formato de los mensajes, puede analizarlos y luego enviar el mensaje al código que pueda manejar la solicitud. Nada de esto es sorprendente, excepto que el procesador/despachador de mensajes reside en el mismo archivo [2](#) que el código que interactúa con un UART contaminado con hardware. El procesador de mensajes es detalles de UART. El procesador de mensajes podría haber sido un software con una vida útil potencialmente larga, pero en cambio es firmware. Al procesador de mensajes se le niega la oportunidad de convertirse en software, ¡y eso simplemente no está bien!

Conozco y entiendo la necesidad de separar el software del hardware desde hace mucho tiempo, pero las palabras de Doug aclararon cómo utilizar los términos software y firmware en relación entre sí.

Para los ingenieros y programadores, el mensaje es claro: dejen de escribir tanto firmware y denle a su código la oportunidad de tener una larga vida útil. Por supuesto, exigirlo no hará que sea así. Veamos cómo podemos mantener limpia la arquitectura del software integrado para darle al software la oportunidad de tener una vida larga y útil.

APLICACIÓN-TEST DE ACTITUD

¿Por qué tanto software integrado potencial se convierte en firmware? Parece que la mayor parte del énfasis está en lograr que el código incorporado funcione, y no tanto en estructurarlo para que tenga una larga vida útil. Kent Beck describe tres actividades en la creación de software (el texto citado son palabras de Kent y las cursivas son mi comentario):

1. "Primero haz que funcione". Estás fuera del negocio si no funciona.
2. "Entonces hazlo bien". Refactorice el código para que usted y otros puedan comprenderlo y evolucionarlo a medida que las necesidades cambien o se comprendan mejor.
3. "Entonces hazlo rápido". Refactorice el código para obtener el rendimiento "necesario".

Gran parte del software de sistemas integrados que veo en el mercado parece haber sido escrito con el objetivo de "hacerlo funcionar" en mente, y quizás también con una obsesión por el objetivo de "hacerlo rápido", que se logra agregando microoptimizaciones en cada oportunidad. . En The Mythical Man-Month, Fred Brooks sugiere que "planeemos tirar uno". Kent y Fred dan prácticamente el mismo consejo: aprenda qué funciona y luego cree una solución mejor.

El software integrado no es especial cuando se trata de estos problemas. La mayoría de las aplicaciones no integradas están diseñadas sólo para funcionar, sin tener en cuenta que el código sea adecuado para una larga vida útil.

Hacer que una aplicación funcione es lo que yo llamo la prueba App-titude para un programador. Los programadores, integrados o no, que sólo se preocupan por hacer que su aplicación funcione no están haciendo ningún favor a sus productos y a sus empleadores. La programación implica mucho más que simplemente hacer que una aplicación funcione.

Como ejemplo de código producido al pasar la prueba App-titude, consulte estas funciones ubicadas en un archivo de un pequeño sistema integrado:

[Haga clic aquí para ver la imagen del código](#)

```
ISR(TIMER1_vect) { ... }  
ISR(INT2_vect) { ... } void  
btn_Handler(void) { ... } float  
calc_RPM(void) { ... } static char  
Read_RawData(void) { ... } void Do_Average(void)  
{ ... }
```

```

void Get_Next_Measurement(void) { ... } void
Zero_Sensor_1(void) { ... } void
Zero_Sensor_2(void) { ... } void
Dev_Control(char Activación) { ... } char
Load_FLASH_Setup(void) { ... } void
Save_FLASH_Setup(void) { ... } void
Store_DataSet(void) { ... } float
bytes2float(char bytes[4]) { ... } void Recall_DataSet(void)
{ ... } void Sensor_init(void) { ... } vacío
uC_Sleep(vacío) { ... }

```

Esa lista de funciones está en el orden en que las encontré en el archivo fuente. Ahora los separaré y agruparé por preocupación:

- Funciones que tienen lógica de dominio

[Haga clic aquí para ver la](#)

```

imagen del código float calc_RPM(void)
{ ... } void Do_Average(void) { ... } void
Get_Next_Measurement(void) { ... } void
Zero_Sensor_1(void) { ... } void
Zero_Sensor_2(void ) {...}

```

- Funciones que configuran la plataforma de hardware

[Haga clic aquí para ver la](#)

```
imagen del código ISR(TIMER1_vect)
```

```
{ ... }* ISR(INT2_vect) { ... } void
```

```
uC_Sleep(void) { ... }
```

Funciones que reaccionan al presionar el botón de encendido

```
y apagado void btn_Handler(void) { ... }
```

```
void Dev_Control(char Activation) { ... }
```

Una función que puede obtener lecturas de entrada A/D del carácter estático del hardware Read_RawData(void) {...}

- Funciones que almacenan valores en el almacenamiento persistente

[Haga clic aquí para ver la](#)

```
imagen del código char Load_FLASH_Setup(void)
```

```
{ ... } void Save_FLASH_Setup(void) { ... } void
```

```
Store_DataSet(void) { ... } float bytes2float(char
bytes[4]) { ... } vacío Recall_DataSet (vacío) {...}
```

- Función que no hace lo que su nombre implica

[Haga clic aquí para ver la imagen del código](#)

```
vacío Sensor_init(vacío) {...}
```

Al observar algunos de los otros archivos de esta aplicación, encontré muchos impedimentos para comprender el código. También encontré una estructura de archivos que implicaba que la única forma de probar este código es en el destino incrustado. Prácticamente cada parte de este código sabe que está en una arquitectura de microprocesador especial, que utiliza C "extendido".

construye [3](#) que vinculan el código a una cadena de herramientas y un microprocesador en particular. No hay forma de que este código tenga una vida útil prolongada a menos que el producto nunca necesite trasladarse a un entorno de hardware diferente.

Esta aplicación funciona: el ingeniero pasó la prueba App-titude. Pero no se puede decir que la aplicación tenga una arquitectura integrada limpia.

EL HARDWARE OBJETIVO EMBOTELLAMIENTO

Hay muchas preocupaciones especiales que los desarrolladores integrados tienen que afrontar y que los desarrolladores no integrados no tienen: por ejemplo, espacio de memoria limitado, limitaciones y plazos en tiempo real, E/S limitadas, interfaces de usuario no convencionales y sensores y conexiones con el mundo real. La mayoría de las veces, el hardware se desarrolla simultáneamente con el software y el firmware. Como ingeniero que desarrolla código para este tipo de sistema, es posible que no tenga un lugar para ejecutar el código.

Si eso no es lo suficientemente malo, una vez que obtenga el hardware, es probable que tenga sus propios defectos, lo que hará que el progreso del desarrollo del software sea incluso más lento de lo habitual.

Sí, integrado es especial. Los ingenieros integrados son especiales. Pero el desarrollo integrado no es tan especial como para que los principios de este libro no sean aplicables a los sistemas integrados.

Uno de los problemas especiales incorporados es el cuello de botella del hardware de destino. Cuando el código incrustado se estructura sin aplicar principios y prácticas de arquitectura limpia, a menudo se enfrentará al escenario en el que puede probar su código solo en el destino. Si el objetivo es el único lugar donde es posible realizar pruebas, el cuello de botella del hardware objetivo lo ralentizará.

UNA ARQUITECTURA INTEGRADA LIMPIA ES COMPROBABLE ARQUITECTURA EMPOTRADA

Veamos cómo aplicar algunos de los principios arquitectónicos al software y firmware integrados para ayudarle a eliminar el cuello de botella del hardware de destino.

Capas

Las capas vienen en muchos sabores. Comencemos con tres capas, como se muestra en [la Figura 29.1](#). En la parte inferior está el hardware. Como nos advierte Doug, debido a los avances tecnológicos y la ley de Moore, el hardware cambiará. Las piezas se vuelven obsoletas y las piezas nuevas consumen menos energía, proporcionan un mejor rendimiento o son más baratas. Cualquiera sea el motivo, como ingeniero integrado, no quiero tener un trabajo más grande del necesario cuando finalmente ocurra el inevitable cambio de hardware.

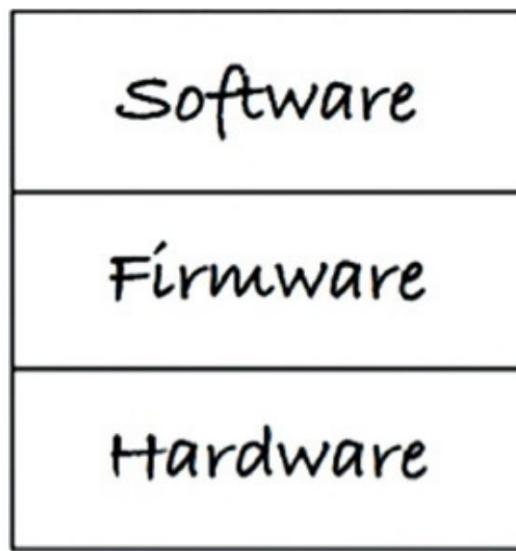


Figura 29.1 Tres capas

La separación entre el hardware y el resto del sistema es un hecho, al menos una vez que se define el hardware ([Figura 29.2](#)). Aquí es donde suelen comenzar los problemas cuando intentas pasar la prueba de App-titude. No hay nada que impida que el conocimiento del hardware contamine todo el código. Si no tiene cuidado con dónde coloca las cosas y qué puede saber un módulo sobre otro módulo, será muy difícil cambiar el código. No me refiero sólo a cuándo cambia el hardware, sino a cuándo el usuario solicita un cambio o cuándo es necesario corregir un error.

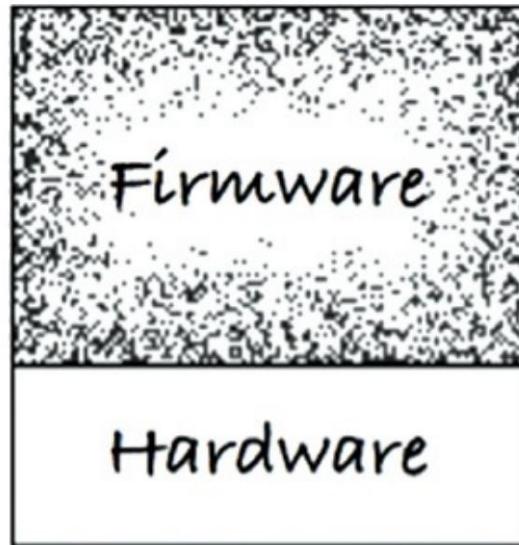


Figura 29.2 El hardware debe estar separado del resto del sistema.

La mezcla de software y firmware es un antipatrón. El código que exhiba este antipatrón resistirá los cambios. Además, los cambios serán peligrosos y a menudo tendrán consecuencias no deseadas. Se necesitarán pruebas de regresión completas de todo el sistema para cambios menores. Si no ha creado pruebas instrumentadas externamente, espere aburrirse con las pruebas manuales y luego podrá esperar nuevos informes de errores.

El hardware es un detalle

La línea entre software y firmware normalmente no está tan bien definida como la línea entre código y hardware, como se muestra en [la Figura 29.3.](#)

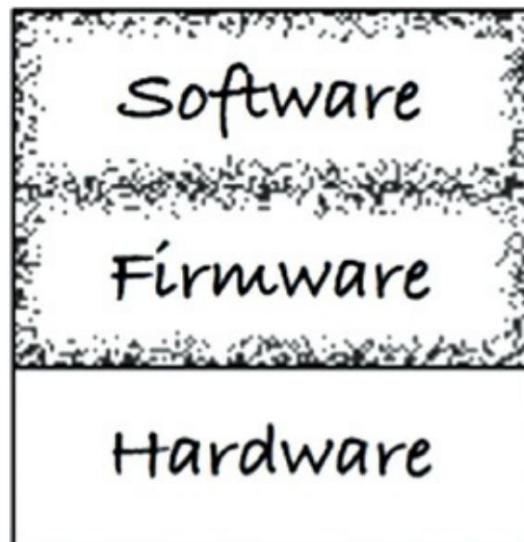


Figura 29.3 La línea entre software y firmware es un poco más borrosa que la línea entre código y hardware

Uno de sus trabajos como desarrollador de software integrado es reafirmar esa línea. El nombre del límite entre el software y el firmware es capa de abstracción de hardware (HAL) ([Figura 29.4](#)). Esta no es una idea nueva: ha estado en las PC desde los días anteriores a Windows.

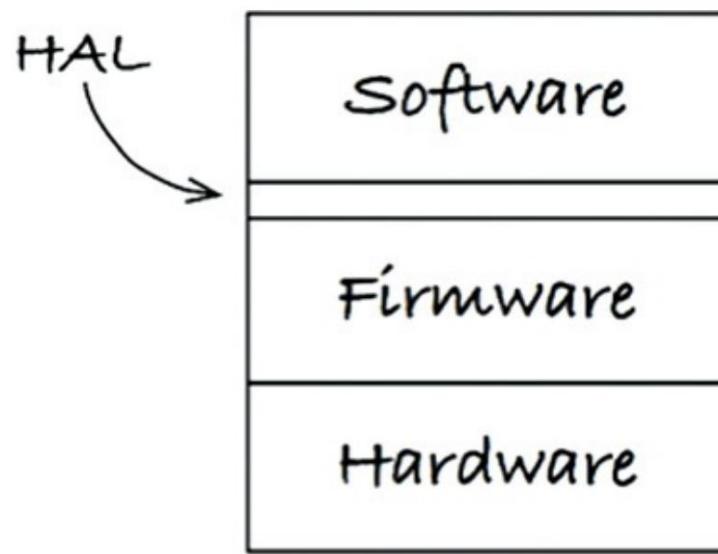


Figura 29.4 La capa de abstracción de hardware

HAL existe para el software que se encuentra encima de él y su API debe adaptarse a las necesidades de ese software. Por ejemplo, el firmware puede almacenar bytes y matrices de bytes en la memoria flash. Por el contrario, la aplicación necesita almacenar y leer pares de nombre/valor en algún mecanismo de persistencia. El software no debe preocuparse de que los pares nombre/valor se almacenen en la memoria flash, un disco giratorio, la nube o la memoria central. HAL proporciona un servicio y no revela al software cómo lo hace. La implementación flash es un detalle que debería ocultarse al software.

Como otro ejemplo, un LED está vinculado a un bit GPIO. El firmware podría proporcionar acceso a los bits GPIO, donde un HAL podría proporcionar `Led_TurnOn(5)`. Esa es una capa de abstracción de hardware de bastante bajo nivel. Consideremos elevar el nivel de abstracción desde la perspectiva del hardware a la perspectiva del software/producto. ¿Qué indica el LED? Supongamos que indicaba batería baja. En algún nivel, el firmware (o un paquete de soporte de placa) podría proporcionar `Led_TurnOn(5)`, mientras que HAL proporciona `Indicate_LowBattery()`. Puedes ver

HAL expresa los servicios que necesita la aplicación. También puedes ver que las capas pueden contener capas. Es más un patrón fractal repetitivo que un conjunto limitado de capas predefinidas. Las asignaciones GPIO son detalles que deben ocultarse al software.

NO REVELAR DETALLES DE HARDWARE AL USUARIO DE EL SALÓN

El software de una arquitectura integrada limpia se puede probar fuera del hardware de destino. Un HAL exitoso proporciona esa costura o conjunto de puntos de sustitución que facilitan las pruebas fuera del objetivo.

El procesador es un detalle

Cuando su aplicación integrada utiliza una cadena de herramientas especializada, a menudo proporcionará archivos de encabezado para ⁴ Estos compiladores a menudo se toman libertades <i>ayudarle</i>. con el lenguaje C, agregando nuevas palabras clave para acceder a las funciones de su procesador. El código se verá como C, pero ya no es C.

A veces, los compiladores de C proporcionados por los proveedores proporcionan lo que parecen variables globales para dar acceso directamente a los registros del procesador, puertos IO, temporizadores de reloj, bits IO, controladores de interrupciones y otras funciones del procesador. Es útil tener acceso a estas cosas fácilmente, pero tenga en cuenta que cualquier código que utilice estas útiles funciones ya no es C. No se compilará para otro procesador, o tal vez incluso con un compilador diferente para el mismo procesador.

Odiaría pensar que el proveedor de silicio y herramientas sea cínico y vincule su producto al compilador. Demos al proveedor el beneficio de la duda asumiendo que realmente está tratando de ayudar. Pero ahora depende de usted utilizar esa ayuda de una manera que no resulte perjudicial en el futuro. Tendrá que limitar qué archivos pueden conocer las extensiones C.

Veamos este archivo de encabezado diseñado para la familia de DSP ACME, ya sabes, los utilizados por Wile E. Coyote:

[Haga clic aquí para ver la imagen del código](#)

```
#ifndef _ACME_STD_TYPES  
#define _ACME_STD_TYPES
```

[Haga clic aquí para ver la imagen del código](#)

```
#si está definido (_ACME_X42)
typedef unsigned int Uint_32; typedef corto
sin firmar Uint_16; typedef carácter sin firmar
Uint_8;

typedefint Int_32; typedef
corto Int_16; typedef char Int_8;

#elif definido (_ACME_A42) typedef
unsigned long Uint_32; typedef unsigned int
Uint_16; typedef carácter sin firmar Uint_8;

typedef largo Int_32; typedef
int Int_16; typedef char
Int_8; #else #error

<acmetypes.h> no es compatible con este entorno #endif
```

#terminara si

El archivo de encabezado acmetypes.h no debe usarse directamente. Si lo hace, su código queda vinculado a uno de los DSP de ACME. Estás usando un DSP ACME, dices, entonces, ¿cuál es el daño? No puede compilar su código a menos que incluya este encabezado. Si usa el encabezado y define _ACME_X42 o _ACME_A42, sus números enteros tendrán el tamaño incorrecto si intenta probar su código fuera del objetivo. Si eso no es suficientemente malo, un día querrás portar tu aplicación a otro procesador, y habrás hecho esa tarea mucho más difícil al no elegir la portabilidad y al no limitar lo que los archivos conocen sobre ACME.

En lugar de utilizar acmetypes.h, debería intentar seguir una ruta más estandarizada y utilizar stdint.h. Pero, ¿qué pasa si el compilador de destino no proporciona stdint.h?

Puede escribir este archivo de encabezado. El stdint.h que usted escribe para las compilaciones de destino usa acmetypes.h para compilaciones de destino como esta:

[Haga clic aquí para ver la imagen del código](#)

```
#ifndef _STDINT_H_
#define _STDINT_H_

#include <tiposacme.h>

typedef Uint_32 uint32_t;
```

```

typedef Uint_16 uint16_t; typedef
Uint_8 uint8_t;

typedef Int_32 int32_t; typedef
Int_16 int16_t; typedef Int_8 int8_t;

```

#terminara si

Hacer que su software y firmware integrados utilicen stdint.h ayuda a mantener su código limpio y portátil. Ciertamente, todo el software debe ser independiente del procesador, pero no todo el firmware puede serlo. El siguiente fragmento de código aprovecha extensiones especiales de C que le dan a su código acceso a los periféricos del microcontrolador. Es probable que su producto utilice este microcontrolador para que pueda utilizar sus periféricos integrados. Esta función genera una línea que dice "hola" al puerto de salida serie. (Este ejemplo se basa en código real salvaje).

[Haga clic aquí para ver la imagen del código](#)

```

vacío decir_hola() {

    Es decir = 0b11000000;
    SBUF0 = (0x68);
    mientras(TI_0 == 0);
    TI_0 = 0;
    SBUF0 = (0x69);
    mientras(TI_0 == 0);
    TI_0 = 0;
    SBUF0 = (0x0a);
    mientras(TI_0 == 0);
    TI_0 = 0;
    SBUF0 = (0x0d);
    mientras(TI_0 == 0);
    TI_0 = 0;
    Es decir = 0b11010000; }

```

Hay muchos problemas con esta pequeña función. Una cosa que podría llamar la atención es la presencia de 0b11000000. Esta notación binaria es genial; ¿Puede C hacer eso? Lamentablemente no. Algunos otros problemas se relacionan con este código directamente usando las extensiones C personalizadas:

IE: bits de habilitación de interrupción.

SBUF0: Búfer de salida serie.

TI_0: Interrupción vacía del búfer de transmisión en serie. Leer un 1 indica que el búfer está vacío.

Las variables en mayúsculas en realidad acceden a los periféricos integrados del microcontrolador. Si desea controlar interrupciones y caracteres de salida, debe utilizar estos periféricos. Sí, esto es conveniente, pero no es C.

Una arquitectura integrada limpia utilizaría estos registros de acceso a dispositivos directamente en muy pocos lugares y los limitaría totalmente al firmware. Todo lo que sabe acerca de estos registros se convierte en firmware y, en consecuencia, está vinculado al silicio. Vincular el código al procesador le perjudicará cuando quiera que el código funcione antes de tener un hardware estable. También le perjudicará cuando mueva su aplicación integrada a un nuevo procesador.

Si utiliza un microcontrolador como este, su firmware podría aislar estas funciones de bajo nivel con algún tipo de capa de abstracción del procesador (PAL).

El firmware por encima del PAL podría probarse fuera del objetivo, haciéndolo un poco menos firme.

El sistema operativo es un detalle

Un HAL es necesario, pero ¿es suficiente? En sistemas integrados básicos, un HAL puede ser todo lo que necesita para evitar que su código se vuelva demasiado adicto al entorno operativo. Pero ¿qué pasa con los sistemas integrados que utilizan un sistema operativo en tiempo real (RTOS) o alguna versión integrada de Linux o Windows?

Para darle a su código incrustado una buena oportunidad de tener una larga vida, debe tratar el sistema operativo como un detalle y protegerlo contra las dependencias del sistema operativo.

El software accede a los servicios del entorno operativo a través del sistema operativo.

El sistema operativo es una capa que separa el software del firmware ([Figura 29.5](#)). Usar un sistema operativo directamente puede causar problemas. Por ejemplo, ¿qué pasa si otra empresa compra su proveedor de RTOS y las regalías aumentan o la calidad disminuye?

¿Qué pasa si sus necesidades cambian y su RTOS no tiene las capacidades que necesita ahora?

Tendrás que cambiar mucho código. Estos no serán sólo simples cambios sintácticos debido a la API del nuevo sistema operativo, sino que probablemente tendrán que adaptarse semánticamente a las diferentes capacidades y primitivas del nuevo sistema operativo.



Figura 29.5 Agregar un sistema operativo

Una arquitectura integrada limpia aísla el software del sistema operativo, a través de una capa de abstracción del sistema operativo (OSAL) ([Figura 29.6](#)). En algunos casos, implementar esta capa puede ser tan sencillo como cambiar el nombre de una función. En otros casos, podría implicar agrupar varias funciones.

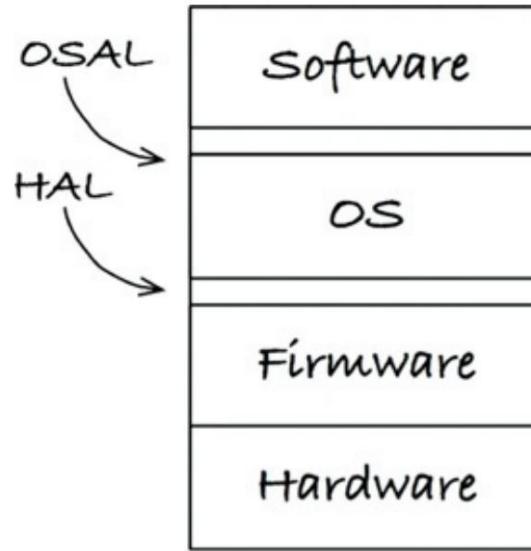


Figura 29.6 La capa de abstracción del sistema operativo

Si alguna vez ha movido su software de un RTOS a otro, sabe que es doloroso. Si su software dependiera de un OSAL en lugar del sistema operativo directamente, en gran medida estaría escribiendo un nuevo OSAL que sea compatible con el antiguo OSAL. ¿Qué preferirías hacer: modificar un montón de código complejo existente o escribir

¿Nuevo código para una interfaz y un comportamiento definidos? Ésta no es una pregunta capciosa. Elijo este último.

Es posible que ahora empieces a preocuparte por el exceso de código. Sin embargo, en realidad la capa se convierte en el lugar donde se aísla gran parte de la duplicación relacionada con el uso de un sistema operativo. Esta duplicación no tiene por qué imponer grandes gastos generales. Si define un OSAL, también puede fomentar que sus aplicaciones tengan una estructura común. Puede proporcionar mecanismos de paso de mensajes, en lugar de que cada hilo elabore su modelo de concurrencia.

OSAL puede ayudar a proporcionar puntos de prueba para que el valioso código de la aplicación en la capa de software pueda probarse fuera del objetivo y fuera del sistema operativo. El software de una arquitectura integrada limpia se puede probar fuera del sistema operativo de destino. Un OSAL exitoso proporciona esa costura o conjunto de puntos de sustitución que facilitan las pruebas fuera del objetivo.

PROGRAMACIÓN A INTERFACES Y SUSTITUTABILIDAD

Además de agregar una HAL y potencialmente una OSAL dentro de cada una de las capas principales (software, sistema operativo, firmware y hardware), puede (y debe) aplicar los principios descritos a lo largo de este libro. Estos principios fomentan la separación de preocupaciones, la programación de interfaces y la sustituibilidad.

La idea de una arquitectura en capas se basa en la idea de programación para interfaces. Cuando un módulo interactúa con otro a través de una interfaz, puede sustituir un proveedor de servicios por otro. Muchos lectores habrán escrito su propia versión pequeña de printf para implementarla en el destino. Siempre que la interfaz de su printf sea la misma que la versión estándar de printf, puede anular el servicio uno por el otro.

Una regla básica es utilizar archivos de encabezado como definiciones de interfaz. Sin embargo, cuando lo haga, deberá tener cuidado con lo que se incluye en el archivo de encabezado. Limite el contenido del archivo de encabezado a declaraciones de funciones, así como a las constantes y nombres de estructuras que necesita la función.

No satura los archivos de encabezado de la interfaz con estructuras de datos, constantes y definiciones de tipos que solo son necesarios para la implementación. No es sólo una cuestión de desorden: ese desorden conducirá a dependencias no deseadas. Limitar la visibilidad de

los detalles de implementación. Espere que los detalles de implementación cambien. Cuantos menos lugares donde el código conozca los detalles, menos lugares habrá que rastrear y modificar el código.

Una arquitectura integrada limpia se puede probar dentro de las capas porque los módulos interactúan a través de interfaces. Cada interfaz proporciona esa costura o punto de sustitución que facilita las pruebas fuera del objetivo.

DIRECTIVAS DE COMPILACIÓN CONDICIONALES EN SECO

Un uso de la sustituibilidad que a menudo se pasa por alto se relaciona con cómo los programas integrados C y C++ manejan diferentes objetivos o sistemas operativos. Existe una tendencia a utilizar la compilación condicional para activar y desactivar segmentos de código. Recuerdo un caso especialmente problemático en el que la declaración `#ifdef BOARD_V2` se mencionó varios miles de veces en una aplicación de telecomunicaciones.

Esta repetición de código viola el principio No repetirse (DRY). vea `#ifdef BOARD_V2` una [5](#) Si I vez, en realidad no es un problema. Seis mil veces es un problema extremo. La compilación condicional que identifica el tipo de hardware de destino se repite a menudo en los sistemas integrados. Pero, ¿Qué más podemos hacer?

¿Qué pasa si hay una capa de abstracción de hardware? El tipo de hardware pasaría a ser un detalle oculto bajo el HAL. Si HAL proporciona un conjunto de interfaces, en lugar de usar compilación condicional, podríamos usar el vinculador o alguna forma de enlace de tiempo de ejecución para conectar el software al hardware.

CONCLUSIÓN

Las personas que desarrollan software integrado tienen mucho que aprender de experiencias fuera del software integrado. Si es un desarrollador integrado y ha leído este libro, encontrará una gran cantidad de sabiduría sobre el desarrollo de software en las palabras y las ideas.

Dejar que todo el código se convierta en firmware no es bueno para la salud a largo plazo de su producto. Poder realizar pruebas solo en el hardware de destino no es bueno para la salud a largo plazo de su producto. Una arquitectura integrada limpia es buena para la salud a largo plazo de su producto.

1. https://insights.sei.cmu.edu/sei_blog/2011/08/the-growing-importance-of-sustaining-software-for-eldod.html
2. El dispositivo de hardware que controla el puerto serie.
3. Algunos proveedores de silicio agregan palabras clave al lenguaje C para facilitar el acceso a los registros y puertos IO. simple de C. Desafortunadamente, una vez hecho esto, el código ya no es C.
4. Esta declaración utiliza HTML intencionalmente.
5. Hunt y Thomas, El programador pragmático.

VI

DETALLES

30

LA BASE DE DATOS ES UN DETALLE



Desde un punto de vista arquitectónico, la base de datos no es una entidad: es un detalle que no alcanza el nivel de un elemento arquitectónico. Su relación con la arquitectura de un sistema de software es similar a la relación del pomo de una puerta con la arquitectura de su hogar.

Me doy cuenta de que son palabras de lucha. Créame, he tenido la pelea. Permítanme ser claro: no me refiero al modelo de datos. La estructura que le dé a los datos dentro de su aplicación es muy importante para la arquitectura de su sistema. Pero la base de datos no es el modelo de datos. La base de datos es una pieza de software. La base de datos es una utilidad que proporciona acceso a los datos. Desde el punto de vista de la arquitectura, esa utilidad es irrelevante porque es un detalle de bajo nivel: un mecanismo. Y un buen arquitecto no permite bajos niveles.

mecanismos para contaminar la arquitectura del sistema.

BASES DE DATOS RELACIONALES

Edgar Codd definió los principios de las bases de datos relacionales en 1970. A mediados de la década de 1980, el modelo relacional había crecido hasta convertirse en la forma dominante de almacenamiento de datos. Había una buena razón para esta popularidad: el modelo relacional es elegante, disciplinado y robusto. Es una excelente tecnología de acceso y almacenamiento de datos.

Pero no importa cuán brillante, útil y matemáticamente sólida sea una tecnología, sigue siendo solo una tecnología. Y eso significa que es un detalle.

Si bien las tablas relacionales pueden ser convenientes para ciertas formas de acceso a los datos, no hay nada significativo desde el punto de vista arquitectónico en organizar los datos en filas dentro de las tablas. Los casos de uso de su aplicación no deberían conocer ni preocuparse por estos asuntos. De hecho, el conocimiento de la estructura tabular de los datos debería restringirse a las funciones de utilidad de nivel más bajo en los círculos externos de la arquitectura.

Muchos marcos de acceso a datos permiten que las filas y tablas de la base de datos pasen por el sistema como objetos. Permitir esto es un error arquitectónico. Combina los casos de uso, las reglas comerciales y, en algunos casos, incluso la interfaz de usuario con la estructura relacional de los datos.

¿POR QUÉ SON TAN PREVALENTES LOS SISTEMAS DE BASES DE DATOS?

¿Por qué los sistemas de software y las empresas de software están dominados por sistemas de bases de datos? ¿A qué se debe la preeminencia de Oracle, MySQL y SQL Server? En una palabra: discos.

El disco magnético giratorio fue el pilar del almacenamiento de datos durante cinco décadas. Varias generaciones de programadores no han conocido otra forma de almacenamiento de datos. La tecnología de discos ha pasado de enormes pilas de enormes platos de 48 pulgadas de diámetro que pesaban miles de libras y contenían 20 megabytes, hasta

círculos delgados, de 3 pulgadas de diámetro, que pesan solo unos pocos gramos y contienen un terabyte o más. Ha sido un viaje salvaje. Y a lo largo de ese viaje, los programadores se han visto afectados por un rasgo fatal de la tecnología de discos: los discos son lentos.

En un disco, los datos se almacenan en pistas circulares. Esas pistas se dividen en sectores que contienen una cantidad conveniente de bytes, a menudo 4K. Cada plato puede tener cientos de pistas y puede haber aproximadamente una docena de platos. Si desea leer un byte particular del disco, debe mover el cabezal a la pista adecuada, esperar a que el disco gire al sector adecuado, leer todos los 4K de ese sector en la RAM y luego indexar en ese búfer de RAM. para obtener el byte que desea. Y todo eso lleva tiempo, milisegundos de tiempo.

Puede que milisegundos no parezca mucho, pero un milisegundo es un millón de veces más que el tiempo de ciclo de la mayoría de los procesadores. Si esos datos no estuvieran en un disco, se podría acceder a ellos en nanosegundos, en lugar de milisegundos.

Para mitigar el retraso impuesto por los discos, necesita índices, cachés y esquemas de consulta optimizados; y necesita algún tipo de medio regular para representar los datos para que estos índices, cachés y esquemas de consulta sepan con qué están trabajando. En resumen, necesita un sistema de gestión y acceso a datos. A lo largo de los años, estos sistemas se han dividido en dos tipos distintos: sistemas de archivos y sistemas de gestión de bases de datos relacionales (RDBMS).

Los sistemas de archivos se basan en documentos. Proporcionan una forma natural y cómoda de almacenar documentos completos. Funcionan bien cuando necesitas guardar y recuperar un conjunto de documentos por nombre, pero no ofrecen mucha ayuda cuando buscas el contenido de esos documentos. Es fácil encontrar un archivo llamado login.c, pero es difícil y lento encontrar cada archivo .c que tenga una variable llamada x .

Los sistemas de bases de datos se basan en contenido. Proporcionan una forma natural y cómoda de buscar registros según su contenido. Son muy buenos para asociar varios registros en función de algún contenido que todos comparten. Desafortunadamente, son bastante pobres a la hora de almacenar y recuperar documentos opacos.

Ambos sistemas organizan los datos en el disco para que puedan almacenarse y recuperarse de la manera más eficiente posible, dadas sus necesidades particulares de acceso.

Cada uno tiene su propio esquema para indexar y organizar los datos. Además, cada uno de ellos lleva los datos relevantes a la RAM, donde pueden manipularse rápidamente.

¿Y SI NO HUBIERA DISCO?

Por muy frecuentes que alguna vez fueran los discos, ahora son una especie en extinción.

Pronto habrán seguido el camino de las unidades de cinta, de disquete y de CD. Están siendo reemplazados por RAM.

Hágase esta pregunta: cuando se acaben todos los discos y todos sus datos estén almacenados en la RAM, ¿cómo organizará esos datos? ¿Lo organizarás en tablas y accederás a él con SQL? ¿Lo organizarás en archivos y accederás a él a través de un directorio?

Por supuesto que no. Lo organizará en listas vinculadas, árboles, tablas hash, pilas, colas o cualquiera de las otras innumerables estructuras de datos, y accederá a él mediante punteros o referencias, porque eso es lo que hacen los programadores.

De hecho, si piensas detenidamente en este tema, te darás cuenta de que esto es lo que ya haces. Aunque los datos se guardan en una base de datos o en un sistema de archivos, usted los lee en la RAM y luego los reorganiza, para su propia conveniencia, en listas, conjuntos, pilas, colas, árboles o cualquier estructura de datos que se le antoje. Es muy poco probable que dejes los datos en forma de archivos o tablas.

DETALLES

Esta realidad es por la que digo que la base de datos es un detalle. Es solo un mecanismo que utilizamos para mover los datos hacia adelante y hacia atrás entre la superficie del disco y la RAM.

En realidad, la base de datos no es más que un gran cubo de bits donde almacenamos nuestros datos a largo plazo. Pero rara vez utilizamos los datos en esa forma.

Por tanto, desde un punto de vista arquitectónico, no deberíamos preocuparnos por la forma que adoptan los datos mientras están en la superficie de un disco magnético giratorio. De hecho, no deberíamos reconocer que el disco existe en absoluto.

¿PERO QUÉ PASA CON EL RENDIMIENTO?

¿No es el rendimiento una preocupación arquitectónica? Por supuesto que lo es, pero cuando se trata de almacenamiento de datos, es una preocupación que puede encapsularse por completo y separarse de las reglas comerciales. Sí, necesitamos ingresar y sacar los datos del almacén de datos.

rápidamente, pero esa es una preocupación de bajo nivel. Podemos abordar esa preocupación con mecanismos de acceso a datos de bajo nivel. No tiene nada que ver con la arquitectura general de nuestros sistemas.

ANÉCDOTA

A finales de la década de 1980, dirigí un equipo de ingenieros de software en una nueva empresa que intentaba construir y comercializar un sistema de gestión de redes que midiera la integridad de las comunicaciones de las líneas de telecomunicaciones T1. El sistema recuperó datos de los dispositivos en los puntos finales de esas líneas y luego ejecutó una serie de algoritmos predictivos para detectar e informar problemas.

Usábamos plataformas UNIX y almacenábamos nuestros datos en archivos simples de acceso aleatorio. No necesitábamos una base de datos relacional porque nuestros datos tenían pocas relaciones basadas en contenido. Era mejor guardarlo en árboles y listas vinculadas en esos archivos de acceso aleatorio. En resumen, mantuvimos los datos en la forma que fuera más conveniente para cargarlos en la RAM, donde podían manipularse.

Contratamos a un gerente de marketing para esta startup, un tipo amable y conocedor. Pero inmediatamente me dijo que teníamos que tener una base de datos relacional en el sistema. No era una opción ni una cuestión de ingeniería: era una cuestión de marketing.

Esto no tuvo sentido para mí. ¿Por qué querría reorganizar mis listas y árboles vinculados en un montón de filas y tablas a las que se accede a través de SQL? ¿Por qué introduciría todos los gastos generales y gastos de un RDBMS masivo cuando un simple sistema de archivos de acceso aleatorio era más que suficiente? Así que luché contra él, con uñas y dientes.

Teníamos un ingeniero de hardware en esta empresa que adoptó el lema RDBMS. Se convenció de que nuestro sistema de software necesitaba un RDBMS por razones técnicas. Se reunía a mis espaldas con los ejecutivos de la empresa, dibujaba muñecos de palitos en la pizarra de una casa balanceándose sobre un poste, y les preguntaba a los ejecutivos: "¿Construirían una casa sobre un poste?". Su mensaje implícito fue que un RDBMS que mantiene sus tablas en archivos de acceso aleatorio era de alguna manera más confiable que los archivos de acceso aleatorio que estábamos usando.

Luché contra él. Luché con el tipo de marketing. Me apegué a mis principios de ingeniería en

El rostro de una ignorancia increíble. Luché, luché y luché.

Al final, el desarrollador de hardware fue ascendido por encima de mí para convertirse en administrador de software. Al final, pusieron un RDBMS en ese pobre sistema. Y, al final, ellos tenían toda la razón y yo estaba equivocado.

No por razones de ingeniería, claro está: tenía razón en eso. Hice bien en luchar contra la instalación de un RDBMS en el núcleo arquitectónico del sistema. La razón por la que me equivoqué fue porque nuestros clientes esperaban que tuviéramos una base de datos relacional. No sabían qué harían con él. No tenían ninguna forma realista de utilizar los datos relacionales en nuestro sistema. Pero no importaba: nuestros clientes esperaban un RDBMS. Se había convertido en una casilla de verificación que todos los compradores de software tenían en su lista. No había ninguna lógica de ingeniería; la racionalidad no tenía nada que ver con eso. Era una necesidad irracional, externa y totalmente infundada, pero no menos real.

¿De dónde surgió esa necesidad? Se originó a partir de las campañas de marketing altamente efectivas empleadas por los proveedores de bases de datos en ese momento. Habían logrado convencer a ejecutivos de alto nivel de que sus “activos de datos” corporativos necesitaban protección y que los sistemas de bases de datos que ofrecían eran el medio ideal para brindar esa protección.

Hoy vemos el mismo tipo de campañas de marketing. La palabra “empresa” y la noción de “Arquitectura Orientada a Servicios” tienen mucho más que ver con el marketing que con la realidad.

¿Qué debería haber hecho en ese escenario de hace mucho tiempo? Debería haber instalado un RDBMS en el costado del sistema y proporcionarle un canal de acceso a datos estrecho y seguro, mientras mantenía los archivos de acceso aleatorio en el núcleo del sistema. ¿Qué hice? Lo dejé y me convertí en consultor.

CONCLUSIÓN

La estructura organizativa de los datos, el modelo de datos, es arquitectónicamente significativo. Las tecnologías y sistemas que mueven datos dentro y fuera de una superficie magnética giratoria no lo son. Los sistemas de bases de datos relacionales que obligan a organizar los datos en tablas y acceder a ellos con SQL tienen mucho más que ver con lo segundo que con lo primero. El dato es significativo. La base de datos es un detalle.

31

LA WEB ES UN DETALLE



¿Eras desarrollador en la década de 1990? ¿Recuerdas cómo la web lo cambió todo? ¿Recuerda cómo mirábamos con desdén nuestras antiguas arquitecturas cliente-servidor frente a la nueva y brillante tecnología de la Web?

En realidad la web no cambió nada. O, al menos, no debería haberlo hecho. La Web es sólo la última de una serie de oscilaciones que nuestra industria ha atravesado desde los años 1960. Estas oscilaciones van y vienen entre poner toda la potencia de la computadora en los servidores centrales y apagar toda la energía de la computadora en las terminales.

Hemos visto varias de estas oscilaciones apenas en la última década desde que la Web se hizo prominente. Al principio pensamos que toda la potencia de la computadora estaría en

granjas de servidores y los navegadores serían estúpidos. Luego comenzamos a poner subprogramas en los navegadores. Pero eso no nos gustó, así que volvimos a trasladar el contenido dinámico a los servidores. Pero eso no nos gustó, así que inventamos la Web 2.0 y trasladamos gran parte del procesamiento al navegador con Ajax y JavaScript. Llegamos incluso a crear aplicaciones enormes escritas para ejecutarse en los navegadores. Y ahora estamos todos entusiasmados por volver a incorporar ese JavaScript al servidor con Node.

(Suspiro.)

EL PÉNDULO SIN FIN

Por supuesto, sería incorrecto pensar que esas oscilaciones comenzaron con la red. Antes de la web, existía una arquitectura cliente-servidor. Antes de eso, había minicomputadoras centrales con conjuntos de terminales tontas. Antes de eso, existían mainframes con terminales inteligentes de pantalla verde (que eran muy análogos a los navegadores modernos). Antes había salas de ordenadores y tarjetas perforadas.

...

Y así va la historia. Parece que no podemos determinar dónde queremos la potencia de la computadora. Vamos y venimos entre centralizarlo y distribuirlo. Y me imagino que esas oscilaciones continuarán durante algún tiempo.

Cuando lo miras en el ámbito general de la historia de TI, la web no cambió nada en absoluto. La red fue simplemente una de las muchas oscilaciones en una lucha que comenzó antes de que la mayoría de nosotros naciéramos y continuará mucho después de que la mayoría de nosotros nos hayamos retirado.

Sin embargo, como arquitectos tenemos que mirar a largo plazo. Esas oscilaciones son sólo cuestiones de corto plazo que queremos alejar del núcleo central de nuestras reglas comerciales.

Déjame contarte la historia de la empresa Q. La empresa Q creó un sistema de finanzas personales muy popular. Era una aplicación de escritorio con una GUI muy útil. Me encantó usarlo.

Luego vino la web. En su siguiente versión, la empresa Q cambió la GUI para que se vea y se comporte como un navegador. ¡Me quedé atónito! ¿Qué genio del marketing decidió que el software de finanzas personales, ejecutado en una computadora de escritorio, debería tener la apariencia y

sensación de un navegador web?

Por supuesto, odié la nueva interfaz. Al parecer, todos los demás también lo hicieron, porque después de algunos lanzamientos, la empresa Q eliminó gradualmente la sensación de navegador y volvió a convertir su sistema de finanzas personales en una GUI de escritorio normal.

Ahora imagina que eres arquitecto de software en Q. Imagina que algún genio del marketing convence a la alta dirección de que toda la interfaz de usuario tiene que cambiar para parecerse más a la web. ¿A qué te dedicas? O, mejor dicho, ¿qué deberías haber hecho antes de este punto para proteger tu aplicación de ese genio del marketing?

Debería haber desacoplado sus reglas comerciales de su interfaz de usuario. No sé si los arquitectos Q lo habían hecho. Algún día me encantaría escuchar su historia. Si hubiera estado allí en ese momento, ciertamente habría presionado mucho para aislar las reglas comerciales de la GUI, porque nunca se sabe qué harán a continuación los genios del marketing.

Consideremos ahora la empresa A, que fabrica un bonito teléfono inteligente. Recientemente lanzó una versión mejorada de su “sistema operativo” (es tan extraño que podamos hablar del sistema operativo dentro de un teléfono). Entre otras cosas, esa actualización del “sistema operativo” cambió por completo la apariencia de todas las aplicaciones. ¿Por qué? Supongo que lo dijo algún genio del marketing.

No soy un experto en el software de ese dispositivo, por lo que no sé si ese cambio causó dificultades importantes a los programadores de las aplicaciones que se ejecutan en el teléfono de la empresa A. Espero que los arquitectos de A y los arquitectos de las aplicaciones mantengan su interfaz de usuario y sus reglas comerciales aisladas entre sí, porque siempre hay genios del marketing esperando para atacar el siguiente acoplamiento que usted cree.

EL RESULTADO

El resultado es simplemente este: la GUI es un detalle. La web es una GUI. Entonces la web es un detalle. Y, como arquitecto, desea poner detalles como ese detrás de límites que los mantengan separados de su lógica empresarial principal.

Piénselo de esta manera: la WEB es un dispositivo IO. En la década de 1960, aprendimos el valor de escribir aplicaciones independientes del dispositivo. La motivación para

esa independencia no ha cambiado. La web no es una excepción a esa regla.

¿O es eso? Se puede argumentar que una GUI, como la web, es tan única y rica que es absurdo buscar una arquitectura independiente del dispositivo. Cuando piensas en las complejidades de la validación de JavaScript o las llamadas AJAX de arrastrar y soltar, o cualquiera de la gran cantidad de otros widgets y gadgets que puedes poner en una página web, es fácil argumentar que la independencia del dispositivo no es práctica.

Hasta cierto punto esto es cierto. La interacción entre la aplicación y la GUI es "conversacional" en formas que son bastante específicas para el tipo de GUI que tiene. El baile entre un navegador y una aplicación web es diferente del baile entre una GUI de escritorio y su aplicación. Parece poco probable que sea posible intentar abstraer ese baile, de la misma manera que se abstraen los dispositivos de UNIX.

Pero se puede abstraer otro límite entre la interfaz de usuario y la aplicación. La lógica empresarial puede considerarse como un conjunto de casos de uso, cada uno de los cuales realiza alguna función en nombre de un usuario. Cada caso de uso se puede describir en función de los datos de entrada, el procesamiento realizado y los datos de salida.

En algún punto del baile entre la interfaz de usuario y la aplicación, se puede decir que los datos de entrada están completos, lo que permite ejecutar el caso de uso. Al finalizar, los datos resultantes se pueden retroalimentar al baile entre la interfaz de usuario y la aplicación.

Los datos de entrada completos y los datos de salida resultantes se pueden colocar en estructuras de datos y usarse como valores de entrada y valores de salida para un proceso que ejecuta el caso de uso. Con este enfoque, podemos considerar que cada caso de uso opera el dispositivo IO de la interfaz de usuario de manera independiente del dispositivo.

CONCLUSIÓN

Este tipo de abstracción no es fácil y probablemente serán necesarias varias iteraciones para lograrlo correctamente. Pero es posible. Y dado que el mundo está lleno de genios del marketing, no es difícil argumentar que a menudo es muy necesario.

32

LOS MARCOS SON DETALLES



Los marcos se han vuelto bastante populares. En términos generales, esto es algo bueno. Existen muchos marcos que son gratuitos, potentes y útiles.

Sin embargo, los marcos no son arquitecturas, aunque algunos intentan serlo.

AUTORES DEL MARCO

La mayoría de los autores de marcos ofrecen su trabajo de forma gratuita porque quieren ser útiles para la comunidad. Quieren retribuir. Esto es loable. Sin embargo, independientemente de sus motivos altruistas, esos autores no tienen en cuenta sus mejores intereses. No pueden, porque no te conocen y no saben

tus problemas.

Los autores del marco conocen sus propios problemas y los problemas de sus compañeros de trabajo y amigos. Y escriben sus marcos para resolver esos problemas, no los tuyos.

Por supuesto, es probable que sus problemas se superpongan bastante con esos otros problemas. Si este no fuera el caso, los frameworks no serían tan populares. En la medida en que exista dicha superposición, los marcos pueden resultar muy útiles.

MATRIMONIO ASIMÉTRICO

La relación entre usted y el autor del marco es extraordinariamente asimétrica. Debes asumir un gran compromiso con el marco, pero el autor del marco no se compromete contigo en absoluto.

Piense detenidamente en este punto. Cuando utiliza un marco, lee la documentación que proporciona el autor de ese marco. En esa documentación, el autor y otros usuarios de ese marco le asesoran sobre cómo integrar su software con el marco. Normalmente, esto significa envolver su arquitectura alrededor de ese marco. El autor recomienda derivar las clases base del marco e importar las funciones del marco a sus objetos comerciales. El autor le insta a acoplar su aplicación al marco lo más estrechamente posible.

Para el autor del marco, acoplarse a su propio marco no es un riesgo. El autor quiere acoplarse a ese marco, porque tiene control absoluto sobre ese marco.

Es más, el autor quiere que usted se acople al marco, porque una vez acoplado de esta manera, es muy difícil separarse. Nada resulta más válido para un autor de framework que un grupo de usuarios dispuestos a derivar inextricablemente de las clases base del autor.

En efecto, el autor le pide que se case con el marco, que haga un compromiso enorme y de largo plazo con ese marco. Y, sin embargo, bajo ninguna circunstancia el autor asumirá un compromiso correspondiente con usted. Es un matrimonio unidireccional. Asumes todo el riesgo y la carga; el autor del marco asume

nada en absoluto.

LOS RIESGOS

¿Cuáles son los riesgos? Éstos son sólo algunos para que los considere.

- La arquitectura del marco a menudo no es muy clara. Los marcos tienden a violar la regla de dependencia. Le piden que herede su código en sus objetos comerciales: ¡sus Entidades! Quieren que su estructura se acople a ese círculo más interno. Una vez dentro, ese marco no volverá a salir. El anillo de bodas está en tu dedo; y va a quedarse allí. • El marco puede ayudarle con algunas de las primeras características de su aplicación.

Sin embargo, a medida que su producto madure, es posible que supere las capacidades del marco. Si te has puesto ese anillo de bodas, encontrarás que la estructura te lucha cada vez más a medida que pasa el tiempo.

- El marco puede evolucionar en una dirección que no le resulte útil. Es posible que se quede atascado al actualizar a nuevas versiones que no le ayudan. Incluso puede encontrar funciones antiguas que utilizó, desapareciendo o cambiando de maneras que le resulten difíciles de seguir.
- Es posible que aparezca un marco nuevo y mejor que le gustaría poder cambiar.
 - a.

LA SOLUCIÓN

¿Cuál es la solución?

¡No te cases con el marco!

Oh, puedes usar el marco, pero no lo conectes. Manténlo al alcance de la mano.

Trate el marco como un detalle que pertenece a uno de los círculos exteriores de la arquitectura. No lo dejes entrar en los círculos internos.

Si el marco quiere que usted derive sus objetos comerciales de sus clases base, ¡dígale que no! En su lugar, derive proxies y manténgalos en componentes que sean complementos de sus reglas comerciales.

No permita que los marcos entren en su código central. En lugar de ello, intégrelos en

componentes que se conectan a su código central, siguiendo la regla de dependencia.

Por ejemplo, tal vez te guste la primavera. Spring es un buen marco de inyección de dependencia. Tal vez uses Spring para conectar automáticamente tus dependencias. Está bien, pero no debe esparcir anotaciones @Autowired en todos sus objetos comerciales. Sus objetos comerciales no deberían conocer Spring.

En su lugar, puedes usar Spring para inyectar dependencias en tu componente principal . Está bien que Main conozca Spring, ya que Main es el componente más sucio y de nivel más bajo de la arquitectura.

AHORA TE PRONUNCIO ...

Hay algunos marcos con los que simplemente debes casarte. Si está utilizando C++, por ejemplo, probablemente tendrá que utilizar STL; es difícil evitarlo. Si está utilizando Java, es casi seguro que tendrá que utilizar la biblioteca estándar.

Eso es normal, pero aun así debería ser una decisión. Debe comprender que cuando une un marco a su aplicación, quedará atrapado con ese marco por el resto del ciclo de vida de esa aplicación. Para bien o para mal, en la enfermedad y en la salud, en la riqueza, en la pobreza, abandonando a todos los demás, estarás utilizando ese marco. Este no es un compromiso que deba asumirse a la ligera.

CONCLUSIÓN

Cuando te enfrentes a un marco, trata de no casarte con él de inmediato. Vea si no hay formas de salir con él por un tiempo antes de dar el paso. Mantenga la estructura detrás de un límite arquitectónico, si es posible, durante el mayor tiempo posible.

Quizás puedas encontrar una manera de conseguir la leche sin comprar la vaca.

33

ESTUDIO DE CASO : VENTAS DE VIDEO



Ahora es el momento de reunir estas reglas y pensamientos sobre la arquitectura en un estudio de caso. Este estudio de caso será breve y simple, pero describirá tanto el proceso que utiliza un buen arquitecto como las decisiones que toma.

EL PRODUCTO

Para este estudio de caso, elegí un producto con el que estoy bastante familiarizado: el software para un sitio web que vende videos. Por supuesto, recuerda a cleancoders.com, el sitio donde vendo mis videos tutoriales de software.

La idea básica es trivial. Tenemos un lote de videos que queremos vender. Vendemos

ellos, en la web, tanto a particulares como a empresas. Las personas pueden pagar un precio para transmitir los videos y otro precio más alto para descargar esos videos y poseerlos permanentemente. Las licencias comerciales solo se transmiten por streaming y se compran en lotes que permiten descuentos por cantidad.

Los individuos suelen actuar como espectadores y compradores. Las empresas, por el contrario, suelen tener personas que compran los videos que otras personas verán.

Los autores de videos deben proporcionar sus archivos de video, descripciones escritas y archivos auxiliares con exámenes, problemas, soluciones, código fuente y otros materiales.

Los administradores deben agregar nuevas series de videos, agregar y eliminar videos desde y hacia la serie y establecer precios para varias licencias.

Nuestro primer paso para determinar la arquitectura inicial del sistema es identificar los actores y los casos de uso.

ANÁLISIS DE CASO DE USO

[La Figura 33.1](#) muestra un análisis de caso de uso típico.

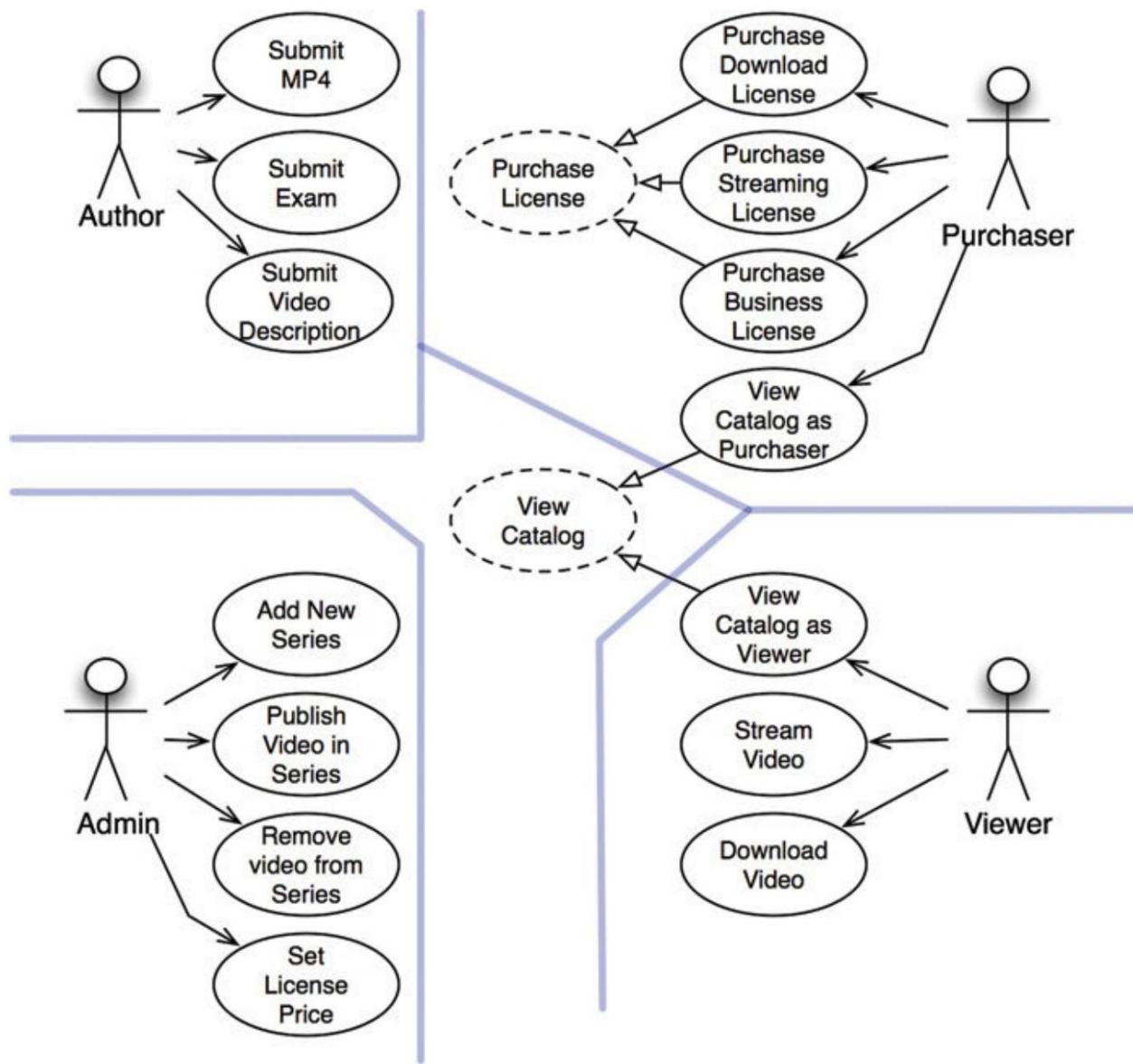


Figura 33.1 Un análisis de caso de uso típico

Los cuatro actores principales son evidentes. Según el Principio de Responsabilidad Única, estos cuatro actores serán las cuatro fuentes principales de cambio para el sistema. Cada vez que se agrega alguna característica nueva, o se cambia alguna característica existente, se tomará ese paso para servir a uno de estos actores. Por lo tanto, queremos dividir el sistema de manera que un cambio en un actor no afecte a ninguno de los demás actores.

Los casos de uso que se muestran en la Figura 33.1 no son una lista completa. Por ejemplo, no encontrará casos de uso de inicio o cierre de sesión. La razón de esta omisión es simplemente gestionar la magnitud del problema en este libro. Si tuviera que incluir todos los diferentes

casos de uso, entonces este capítulo tendría que convertirse en un libro por derecho propio.

Observe los casos de uso discontinuos en el centro de [la Figura 33.1](#). Son casos abstractos¹.

Un caso de uso abstracto es aquel que establece una política general que otro caso de uso desarrollará. Como puede ver, los casos de uso Ver catálogo como visor y Ver catálogo como comprador heredan del caso de uso abstracto Ver catálogo .

Por un lado, no me era estrictamente necesario crear esa abstracción. Podría haber dejado el caso de uso abstracto fuera del diagrama sin comprometer ninguna de las características del producto general. Por otro lado, estos dos casos de uso son tan similares que pensé que sería prudente reconocer la similitud y encontrar una manera de unificarla al principio del análisis.

ARQUITECTURA DE COMPONENTES

Ahora que conocemos los actores y los casos de uso, podemos crear una arquitectura de componentes preliminar ([Figura 33.2](#)).

Las líneas dobles en el dibujo representan límites arquitectónicos como de costumbre. Puede ver la partición típica de vistas, presentadores, interactuantes y controladores.

También puedes ver que he dividido cada una de esas categorías según sus actores correspondientes.

Cada uno de los componentes de [la Figura 33.2](#) representa un archivo .jar o un archivo .dll potencial . Cada uno de esos componentes contendrá las vistas, presentadores, interactuantes y controladores que se le han asignado.

Tenga en cuenta los componentes especiales para la Vista de catálogo y el Presentador de catálogo. Así es como traté del caso de uso abstracto de Ver catálogo . Supongo que esas vistas y presentadores se codificarán en clases abstractas dentro de esos componentes, y que los componentes heredados contendrán clases de vista y presentador que heredarán de esas clases abstractas.

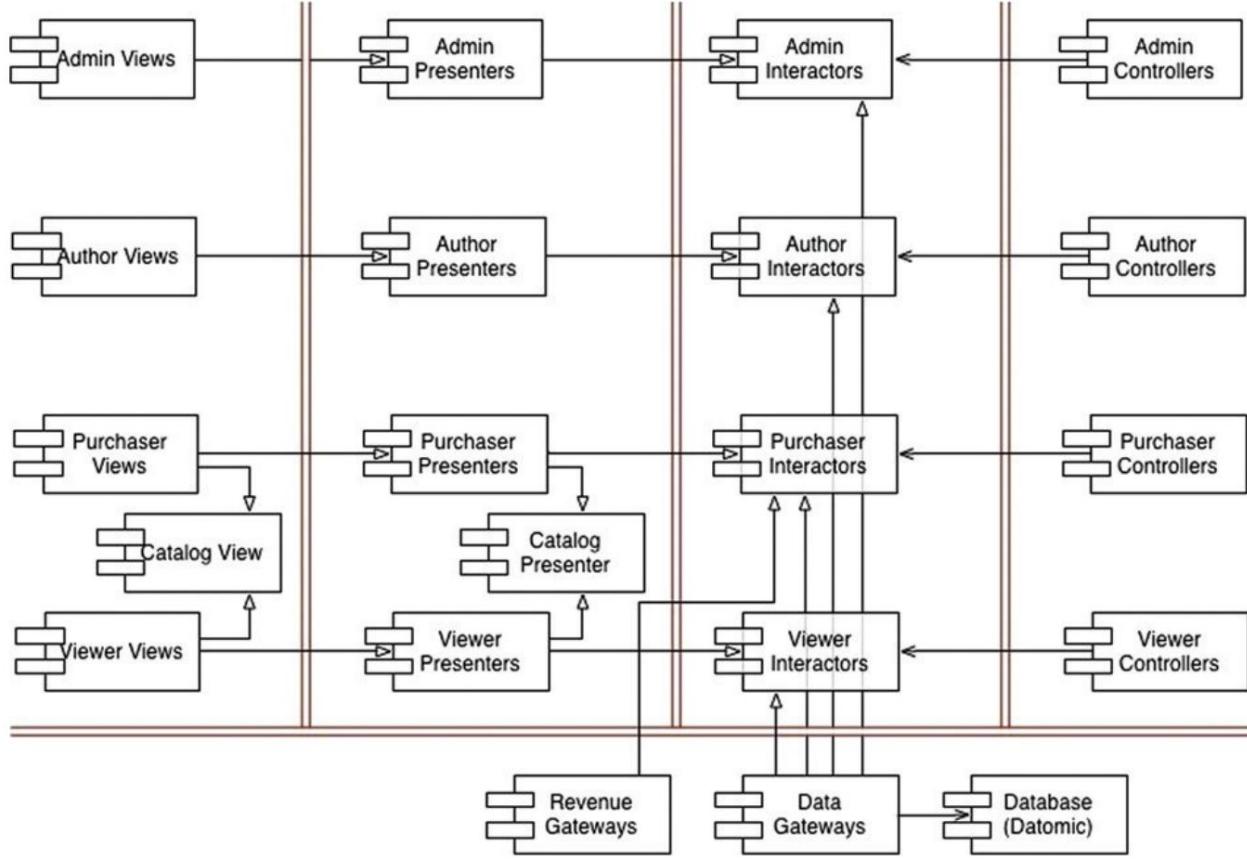


Figura 33.2 Una arquitectura de componentes preliminar

¿Realmente dividiría el sistema en todos estos componentes y los entregaría como archivos .jar o .dll? Si y no. Ciertamente rompería el entorno de compilación y construcción de esta manera, para poder crear entregables independientes como ese.

También me reservaría el derecho de combinar todos esos entregables en un número menor de entregables si fuera necesario. Por ejemplo, dada la partición de la Figura 33.2, sería fácil combinarlos en cinco archivos .jar : uno para vistas, presentadores, interactuantes, controladores y utilidades, respectivamente. Luego podría implementar de forma independiente los componentes que tienen más probabilidades de cambiar independientemente unos de otros.

Otra posible agrupación sería juntar las vistas y los presentadores en el mismo archivo .jar y colocar los interactuantes, controladores y utilidades en su propio archivo .jar . Otra agrupación aún más primitiva sería crear dos archivos .jar , con vistas y presentadores en un archivo y todo lo demás en el otro.

Mantener estas opciones abiertas nos permitirá adaptar la forma en que implementamos el sistema.

en función de cómo cambia el sistema a lo largo del tiempo.

GESTIÓN DE DEPENDENCIA

El flujo de control en [la Figura 33.2](#) procede de derecha a izquierda. La entrada se produce en los controladores y los interactuantes procesan esa entrada en un resultado. Luego, los presentadores dan formato a los resultados y las vistas muestran esas presentaciones.

Observe que no todas las flechas fluyen de derecha a izquierda. De hecho, la mayoría apunta de izquierda a derecha. Esto se debe a que la arquitectura sigue la regla de dependencia. Todas las dependencias cruzan las líneas fronterizas en una dirección y siempre apuntan hacia los componentes que contienen la política de nivel superior.

Observe también que las relaciones de uso (flechas abiertas) apuntan con el flujo de control y que las relaciones de herencia (flechas cerradas) apuntan contra el flujo de control. Esto describe nuestro uso del principio abierto-cerrado para garantizar que las dependencias fluyan en la dirección correcta y que los cambios en los detalles de bajo nivel no se extiendan hacia arriba y afecten las políticas de alto nivel.

CONCLUSIÓN

El diagrama de arquitectura de [la Figura 33.2](#) incluye dos dimensiones de separación.

El primero es la separación de actores basada en el Principio de Responsabilidad Única; la segunda es la Regla de Dependencia. El objetivo de ambos es separar componentes que cambian por diferentes razones y a diferentes ritmos. Los distintos motivos corresponden a los actores; las diferentes tasas corresponden a los diferentes niveles de política.

Una vez que haya estructurado el código de esta manera, puede mezclar y combinar cómo desea implementar el sistema. Puede agrupar los componentes en entregables implementables de cualquier forma que tenga sentido y cambiar fácilmente esa agrupación cuando cambien las condiciones.

¹. Esta es mi propia notación para casos de uso "abstractos". Habría sido más estándar usar un estereotipo UML como <<abstract>>, pero hoy en día no encuentro muy útil adherirse a tales estándares.

34

EL CAPÍTULO QUE FALTA



Por Simón Brown

Todos los consejos que ha leído hasta ahora le ayudarán sin duda a diseñar un mejor software, compuesto de clases y componentes con límites bien definidos, responsabilidades claras y dependencias controladas. Pero resulta que el diablo está en los detalles de implementación, y es muy fácil caer en el último obstáculo si no piensas en eso también.

Imaginemos que estamos construyendo una librería en línea y uno de los casos de uso que nos pidieron implementar es que los clientes puedan ver el estado de sus pedidos. Aunque este es un ejemplo de Java, los principios se aplican igualmente a otros lenguajes de programación. Dejemos de lado la Arquitectura Limpia por un momento.

momento y observe una serie de enfoques para el diseño y la organización del código.

PAQUETE POR CAPA

El primer enfoque de diseño, y quizás el más simple, es la arquitectura tradicional en capas horizontales, donde separamos nuestro código en función de lo que hace desde una perspectiva técnica. A esto se le suele denominar “paquete por capa”. [La Figura 34.1 muestra cómo se vería esto como un diagrama de clases UML](#).

En esta arquitectura en capas típica, tenemos una capa para el código web, una capa para nuestra “lógica empresarial” y una capa para la persistencia. En otras palabras, el código se divide horizontalmente en capas, que se utilizan como una forma de agrupar tipos similares de cosas. En una “arquitectura de capas estricta”, las capas deberían depender únicamente de la siguiente capa inferior adyacente. En Java, las capas normalmente se implementan como paquetes. Como puede ver en [la Figura 34.1](#), todas las dependencias entre capas (paquetes) apuntan hacia abajo. En este ejemplo, tenemos los siguientes tipos de Java:

- OrdersController: un controlador web, algo así como un controlador Spring MVC, que maneja solicitudes de la web.
- OrdersService: una interfaz que define la “lógica de negocios” relacionada con pedidos.
- OrdersServiceImpl: La implementación del servicio de pedidos. • [1](#)
- OrdersRepository: una interfaz que define cómo obtenemos acceso a la información persistente de los pedidos.
- JdbcOrdersRepository: una implementación de la interfaz del repositorio.

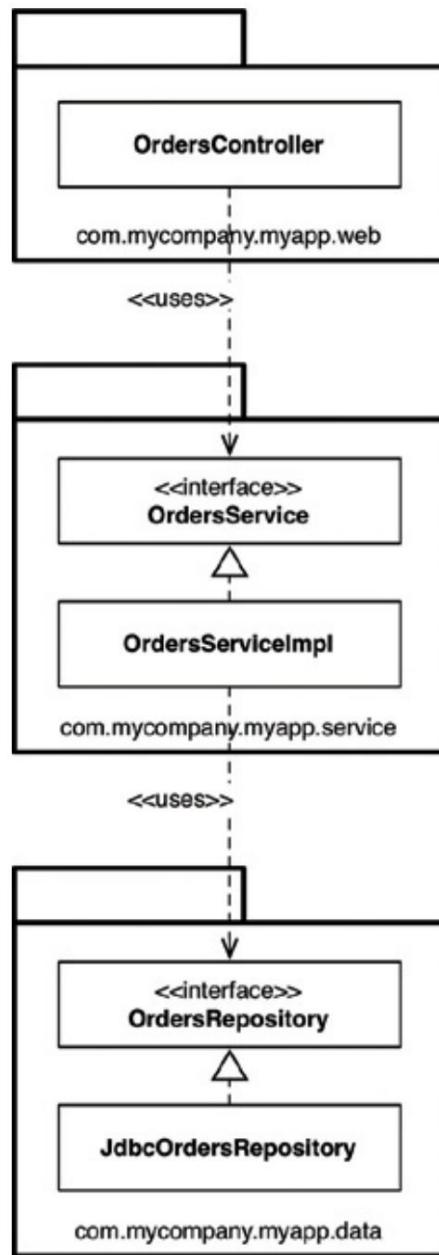


Figura 34.1 Paquete por capa

En "Capas de datos del dominio de presentación"², Martin Fowler afirma que adoptar una arquitectura en capas de este tipo es una buena forma de empezar. No está solo. Muchos de los libros, tutoriales, cursos de capacitación y códigos de muestra que encontrará también le indicarán el camino para crear una arquitectura en capas. Es una forma muy rápida de poner algo en funcionamiento sin una gran complejidad. El problema, como señala Martin, es que una vez que su software crece en escala y complejidad, rápidamente descubrirá que tener tres grandes depósitos de código no es suficiente y tendrá que pensar en modularizar aún más.

Otro problema es que, como ya ha dicho el tío Bob, una arquitectura en capas no dice nada sobre el ámbito empresarial. Coloque el código para arquitecturas de dos capas, de dos dominios comerciales muy diferentes, uno al lado del otro y probablemente se verán inquietantemente similares: web, servicios y repositorios. También hay otro gran problema con las arquitecturas en capas, pero hablaremos de eso más adelante.

PAQUETE POR CARACTERÍSTICA

Otra opción para organizar su código es adoptar un estilo de "paquete por característica". Se trata de una división vertical, basada en características relacionadas, conceptos de dominio o raíces agregadas (para usar terminología de diseño basada en dominios). En las implementaciones típicas que he visto, todos los tipos se colocan en un único paquete Java, cuyo nombre refleja el concepto que se está agrupando.

Con este enfoque, como se muestra en [la Figura 34.2](#), tenemos las mismas interfaces y clases que antes, pero todas están ubicadas en un único paquete Java en lugar de dividirse en tres paquetes. Esta es una refactorización muy simple del estilo "paquete por capa", pero la organización de nivel superior del código ahora grita algo sobre el dominio empresarial. Ahora podemos ver que esta base de código tiene algo que ver con los pedidos más que con la web, los servicios y los repositorios.

Otro beneficio es que es potencialmente más fácil encontrar todo el código que necesita modificar en caso de que cambie el caso de uso "ver pedidos". Todo está contenido en un único paquete Java en lugar de estar distribuido. [3](#)

A menudo veo que los equipos de desarrollo de software se dan cuenta de que tienen problemas con las capas horizontales ("paquete por capa") y en su lugar cambian a las capas verticales ("paquete por característica"). En mi opinión, ambos son subóptimos. Si ha leído este libro hasta ahora, quizá esté pensando que podemos hacerlo mucho mejor, y tiene razón.

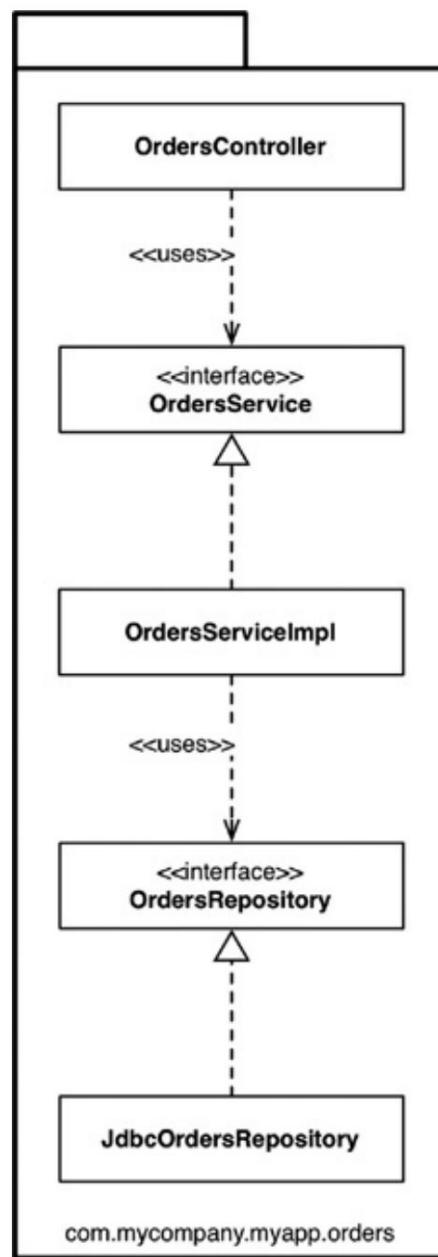


Figura 34.2 Paquete por característica

PUERTOS Y ADAPTADORES

Como ha dicho el tío Bob, enfoques como “puertos y adaptadores”, la “arquitectura hexagonal”, “límites, controladores, entidades”, etc., tienen como objetivo crear arquitecturas en las que el código centrado en el negocio/dominio sea independiente y esté separado de lo técnico. Detalles de implementación, como marcos y bases de datos. En resumen, a menudo se ve que dichas bases de código se componen de un “interno”

(dominio) y un “exterior” (infraestructura), como se sugiere en [la Figura 34.3.](#)

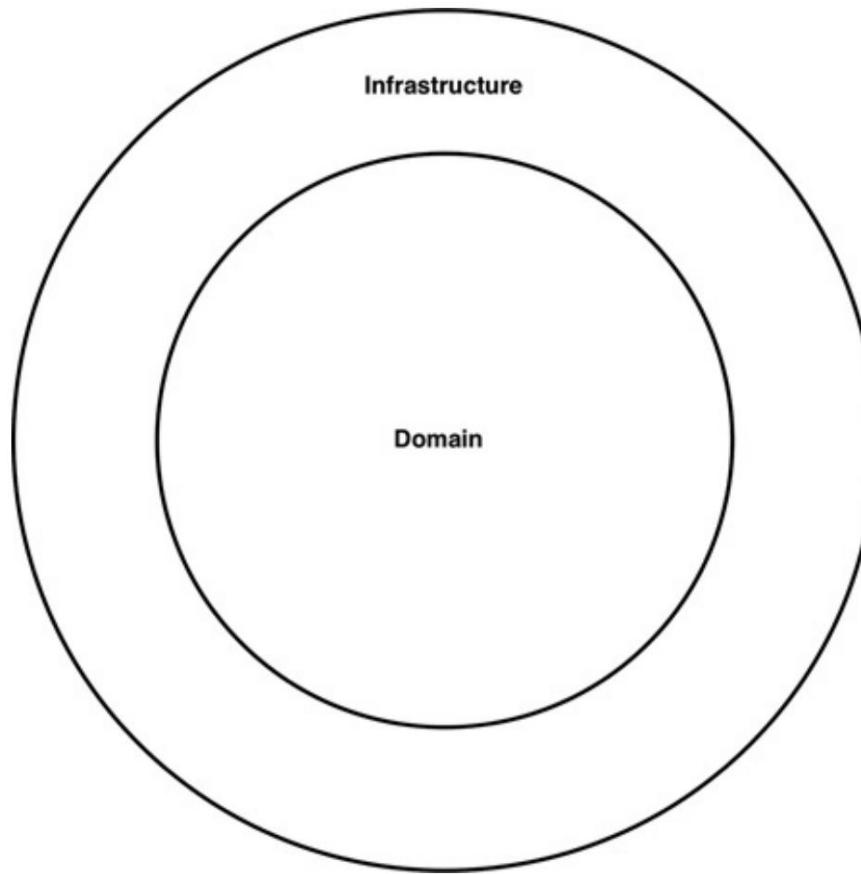


Figura 34.3 Una base de código con un interior y un exterior

La región “interior” contiene todos los conceptos de dominio, mientras que la región “exterior” contiene las interacciones con el mundo exterior (por ejemplo, interfaces de usuario, bases de datos, integraciones de terceros). La regla principal aquí es que el “exterior” depende del “interior”, y nunca al revés. [La Figura 34.4](#) muestra una versión de cómo se podría implementar el caso de uso “ver pedidos”.

El paquete com.mycompany.myapp.domain aquí es el “interior” y los otros paquetes son el “exterior”. Observe cómo las dependencias fluyen hacia el “adentro”. El lector atento notará que el Repositorio de Órdenes de los diagramas anteriores ha sido renombrado simplemente como Órdenes. Esto proviene del mundo del diseño basado en dominios, donde el consejo es que la denominación de todo lo que está “en el interior” debe expresarse en términos del “lenguaje de dominio ubicuo”. Para decirlo de otra manera, hablamos de “órdenes” cuando hablamos sobre el dominio, no del “repositorio de órdenes”.

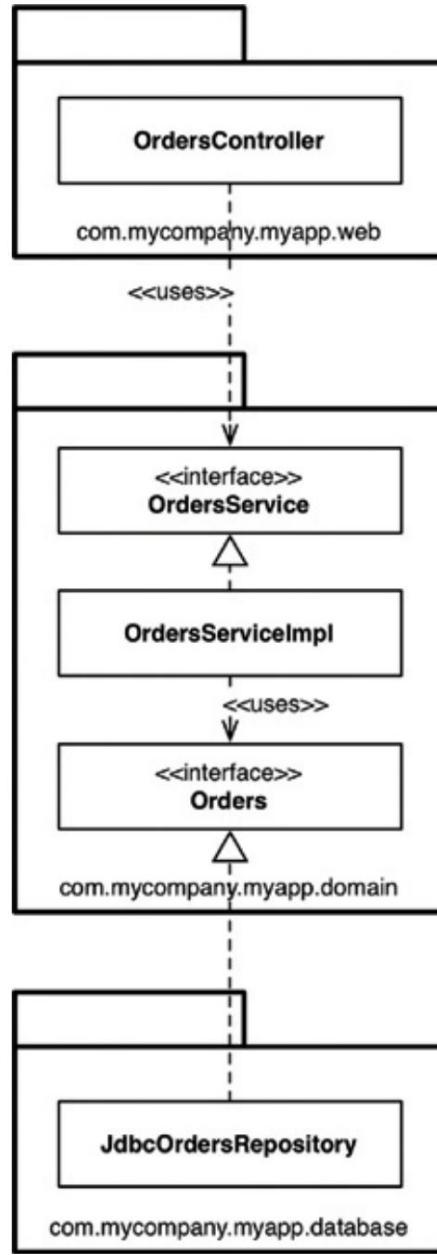


Figura 34.4 Caso de uso de Ver pedidos

También vale la pena señalar que esta es una versión simplificada de cómo podría verse el diagrama de clases UML, porque le faltan elementos como interactores y objetos para ordenar los datos a través de los límites de dependencia.

PAQUETE POR COMPONENTE

Aunque estoy totalmente de acuerdo con las discusiones sobre SOLID, REP, CCP,

y CRP y la mayoría de los consejos de este libro, llego a una conclusión ligeramente diferente sobre cómo organizar el código. Así que aquí presentaré otra opción, a la que llamo "paquete por componente". Para brindarles algunos antecedentes, he pasado la mayor parte de mi carrera desarrollando software empresarial, principalmente en Java, en varios dominios comerciales diferentes. Esos sistemas de software también han variado inmensamente. Un gran número se han basado en la web, pero otros se han diferenciado por [4](#) –, distribuido, basado en mensajes o algo más. Aunque el tecnologías cliente-servidor; el tema común era que la arquitectura de la mayoría de estos sistemas de software se basaba en una arquitectura en capas tradicional.

Ya mencioné un par de razones por las que las arquitecturas en capas deberían considerarse malas, pero esa no es toda la historia. El propósito de una arquitectura en capas es separar el código que tiene el mismo tipo de función. Los elementos web están separados de la lógica empresarial, que a su vez está separada del acceso a los datos. Como vimos en el diagrama de clases UML, desde una perspectiva de implementación, una capa normalmente equivale a un paquete Java. Desde una perspectiva de accesibilidad del código, para que OrdersController pueda tener una dependencia de la interfaz OrdersService , la interfaz OrdersService debe estar marcada como pública, porque están en paquetes diferentes. Del mismo modo, la interfaz OrdersRepository debe marcarse como pública para que la clase OrdersServiceImpl pueda verla fuera del paquete del repositorio .

En una arquitectura en capas estricta, las flechas de dependencia siempre deben apuntar hacia abajo, y las capas dependen únicamente de la siguiente capa inferior adyacente. Esto se reduce a crear un gráfico de dependencia acíclico, limpio y agradable, que se logra introduciendo algunas reglas sobre cómo los elementos en una base de código deben depender entre sí. El gran problema aquí es que podemos hacer trampa introduciendo algunas dependencias no deseadas y aun así crear un bonito gráfico de dependencia a

Supongamos que contrata a alguien nuevo que se une a su equipo y le da al recién llegado otro caso de uso relacionado con los pedidos para implementar. Dado que la persona es nueva, quiere causar una gran impresión e implementar este caso de uso lo más rápido posible. Después de sentarse con una taza de café durante unos minutos, el recién llegado descubre una clase OrdersController existente , por lo que decide que ahí es donde debe ir el código de la nueva página web relacionada con pedidos. Pero necesita algunos datos de pedidos de la base de datos. El recién llegado tiene una epifanía: "Oh, también hay una interfaz OrdersRepository ya creada. Puedo simplemente inyectar dependencias de la implementación en mi controlador. ¡Perfecto!" Después de unos minutos más de piratería, la página web está funcionando. Pero el UML resultante

El diagrama se parece a [la Figura 34.5.](#)

Las flechas de dependencia todavía apuntan hacia abajo, pero OrdersController ahora además omite OrdersService para algunos casos de uso. Esta organización a menudo se denomina arquitectura en capas relajada, ya que a las capas se les permite saltar alrededor de sus vecinos adyacentes. En algunas situaciones, este es el resultado deseado, por ejemplo, si estás intentando seguir el patrón CQRS. En muchos otros casos, no es deseable pasar por alto la capa de lógica empresarial, especialmente si esa lógica empresarial es responsable de garantizar el acceso autorizado a registros individuales, por ejemplo.⁵

Si bien el nuevo caso de uso funciona, quizás no se implemente de la manera que esperábamos. Veo que esto sucede mucho con los equipos que visito como consultor y, por lo general, se revela cuando los equipos comienzan a visualizar cómo se ve realmente su código base, a menudo por primera vez.

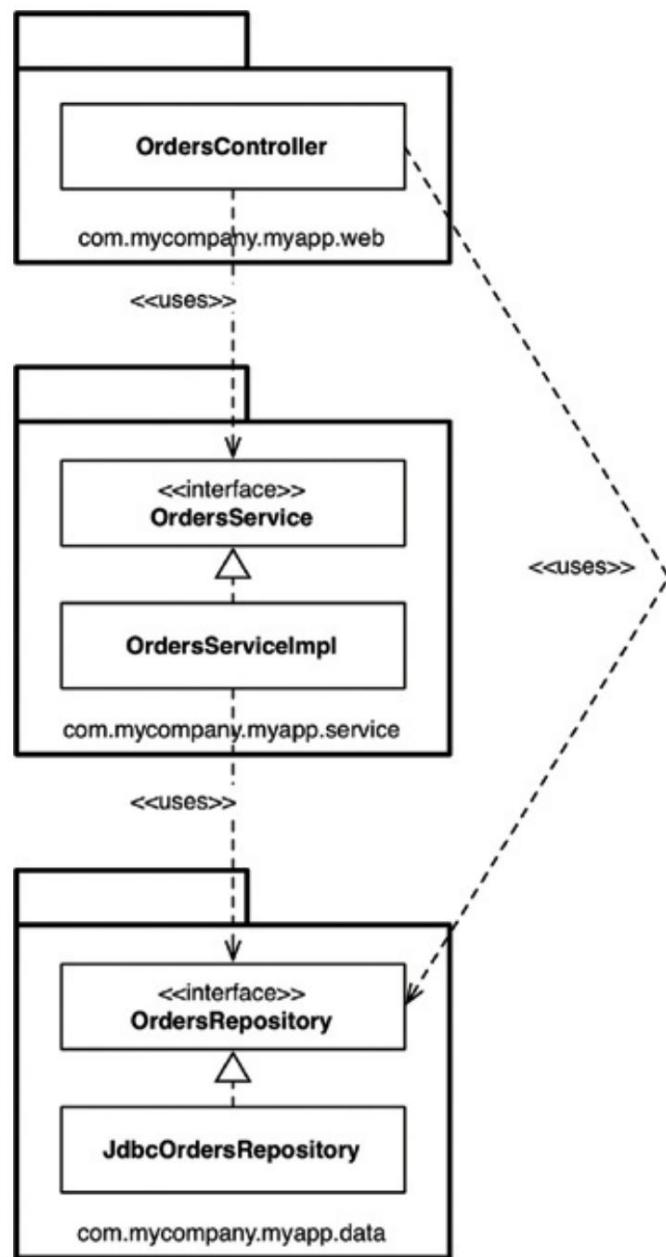


Figura 34.5 Arquitectura en capas relajada

Lo que necesitamos aquí es una guía (un principio arquitectónico) que diga algo como: "Los controladores web nunca deben acceder a los repositorios directamente". La cuestión, por supuesto, es la aplicación de la ley. Muchos equipos que he conocido simplemente dicen: "Aplicamos este principio mediante una buena disciplina y revisiones de código, porque confiamos en nuestros desarrolladores". Es fantástico escuchar esta confianza, pero todos sabemos lo que sucede cuando los presupuestos y los plazos comienzan a acercarse cada vez más.

Un número mucho menor de equipos me dice que utilizan herramientas de análisis estático (p. ej.,

NDepend, Structure101, Checkstyle) para verificar y aplicar automáticamente las violaciones de la arquitectura en el momento de la compilación. Es posible que usted mismo haya visto esas reglas; normalmente se manifiestan como expresiones regulares o cadenas comodín que indican "los tipos del paquete **/web no deben acceder a los tipos de **/data"; y se ejecutan después del paso de compilación.

Este enfoque es un poco tosco, pero puede funcionar: informar violaciones de los principios de la arquitectura que ha definido como equipo y (esperamos) fallar la compilación. El problema con ambos enfoques es que son falibles y el ciclo de retroalimentación es más largo de lo que debería ser. Si no se controla, esta práctica puede convertir una base de código en una "gran bola de barro". [6](#) Personalmente, me gustaría usar el compilador para hacer cumplir mi arquitectura, si es posible.

Esto nos lleva a la opción "paquete por componente". Es un enfoque híbrido para todo lo que hemos visto hasta ahora, con el objetivo de agrupar todas las responsabilidades relacionadas con un único componente de grano grueso en un único paquete Java. Se trata de adoptar una visión centrada en el servicio de un sistema de software, que es algo que también estamos viendo con las arquitecturas de microservicios. De la misma manera que los puertos y adaptadores tratan la web como un mecanismo de entrega más, el "paquete por componente" mantiene la interfaz de usuario separada de estos componentes generales. [La Figura 34.6](#) muestra cómo podría ser el caso de uso "ver pedidos".

En esencia, este enfoque agrupa la "lógica empresarial" y el código de persistencia en una sola cosa, a la que llamo "componente". El tío Bob presentó su definición de "componente" anteriormente en el libro, diciendo:

Los componentes son las unidades de despliegue. Son las entidades más pequeñas que se pueden implementar como parte de un sistema. En Java, son archivos jar.

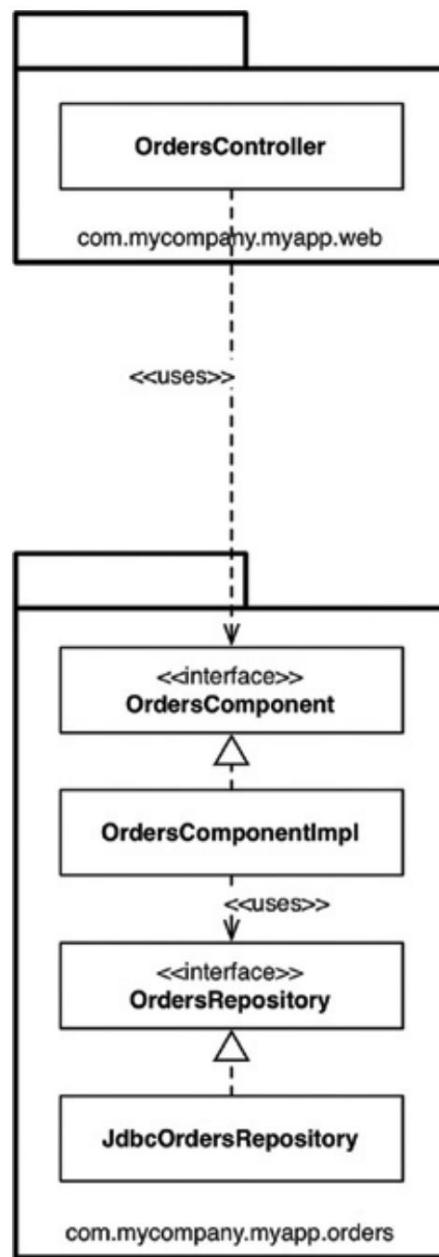


Figura 34.6 Caso de uso de Ver pedidos

Mi definición de componente es ligeramente diferente: "Una agrupación de funcionalidades relacionadas detrás de una interfaz limpia y agradable, que reside dentro de un entorno de ejecución como una aplicación". Esta definición proviene de mi "modelo de arquitectura de software⁷ que es una forma jerárquica simple de pensar en lo estático C4", estructuras de un sistema de software en términos de contenedores, componentes y clases (o código). Dice que un sistema de software se compone de uno o más contenedores (por ejemplo, aplicaciones web, aplicaciones móviles, aplicaciones independientes, bases de datos, sistemas de archivos), cada uno de los cuales contiene uno o más componentes, que a su vez son

implementado por una o más clases (o código). Si cada componente reside en un archivo jar separado es una preocupación ortogonal.

Un beneficio clave del enfoque "paquete por componente" es que si está escribiendo código que necesita hacer algo con los pedidos, solo hay un lugar al que acudir: OrdersComponent. Dentro del componente, la separación de preocupaciones aún se mantiene, por lo que la lógica empresarial está separada de la persistencia de los datos, pero ese es un detalle de implementación del componente que los consumidores no necesitan conocer. Esto es similar a lo que podría terminar si adoptara un microservicio o una arquitectura orientada a servicios: un OrdersService separado que encapsule todo lo relacionado con el manejo de pedidos. La diferencia clave es el modo de desacoplamiento. Puede pensar en componentes bien definidos en una aplicación monolítica como un trampolín hacia una arquitectura de microservicios.

EL DIABLO ESTA EN EL DETALLES DE IMPLEMENTACION

A primera vista, los cuatro enfoques parecen diferentes formas de organizar el código y, por lo tanto, podrían considerarse estilos arquitectónicos diferentes. Sin embargo, esta percepción comienza a desmoronarse muy rápidamente si se equivocan en los detalles de implementación.

Algo que veo regularmente es un uso excesivamente liberal del modificador de acceso público en lenguajes como Java. Es casi como si nosotros, como desarrolladores, usáramos instintivamente la palabra clave pública sin pensar. Está en nuestra memoria muscular. Si no me cree, eche un vistazo a los ejemplos de código de libros, tutoriales y marcos de código abierto en GitHub. Esta tendencia es evidente, independientemente del estilo arquitectónico que pretenda adoptar una base de código: capas horizontales, capas verticales, puertos y adaptadores, o cualquier otra cosa. Marcar todos sus tipos como públicos significa que no está aprovechando las facilidades que brinda su lenguaje de programación con respecto a la encapsulación. En algunos casos, literalmente no hay nada que impida que alguien escriba algún código para crear una instancia de una clase de implementación concreta directamente, violando el estilo de arquitectura previsto.

ORGANIZACIÓN VERSUS

ENCAPSULACIÓN

Mirando este tema de otra manera, si hace públicos todos los tipos en su aplicación Java, los paquetes son simplemente un mecanismo de organización (una agrupación, como carpetas), en lugar de usarse para encapsulación. Dado que los tipos públicos se pueden usar desde cualquier lugar de una base de código, puede ignorar efectivamente los paquetes porque proporcionan muy poco valor real. El resultado neto es que si ignora los paquetes (porque no proporcionan ningún medio de encapsulación y ocultación), realmente no importa qué estilo arquitectónico aspira a crear. Si volvemos a mirar los diagramas UML de ejemplo, los paquetes Java se vuelven un detalle irrelevante si todos los tipos están marcados como públicos. En esencia, los cuatro enfoques arquitectónicos presentados anteriormente en este capítulo son exactamente iguales cuando abusamos de esta designación ([Figura 34.7](#)).

Observe de cerca las flechas entre cada uno de los tipos en [la Figura 34.7: todos son](#) idénticos independientemente del enfoque arquitectónico que intente adoptar. Conceptualmente los enfoques son muy diferentes, pero sintácticamente son idénticos. Además, se podría argumentar que cuando hace públicos todos los tipos, lo que realmente tiene son sólo cuatro formas de describir una arquitectura tradicional en capas horizontales. Este es un buen truco y, por supuesto, nadie haría públicos todos sus tipos de Java. Excepto cuando lo hacen. Y lo he visto.

modificadores de acceso en Java no son perfectos, [8](#) pero ignorarlos es solo pedir Los problema. La forma en que se colocan los tipos de Java en los paquetes puede marcar una gran diferencia en cuanto a cuán accesibles (o inaccesibles) pueden ser esos tipos cuando los modificadores de acceso de Java se aplican adecuadamente. Si vuelvo a traer los paquetes y marco (difuminando gráficamente) aquellos tipos en los que el modificador de acceso puede hacerse más restrictivo, la imagen se vuelve bastante interesante ([Figura 34.8](#)).

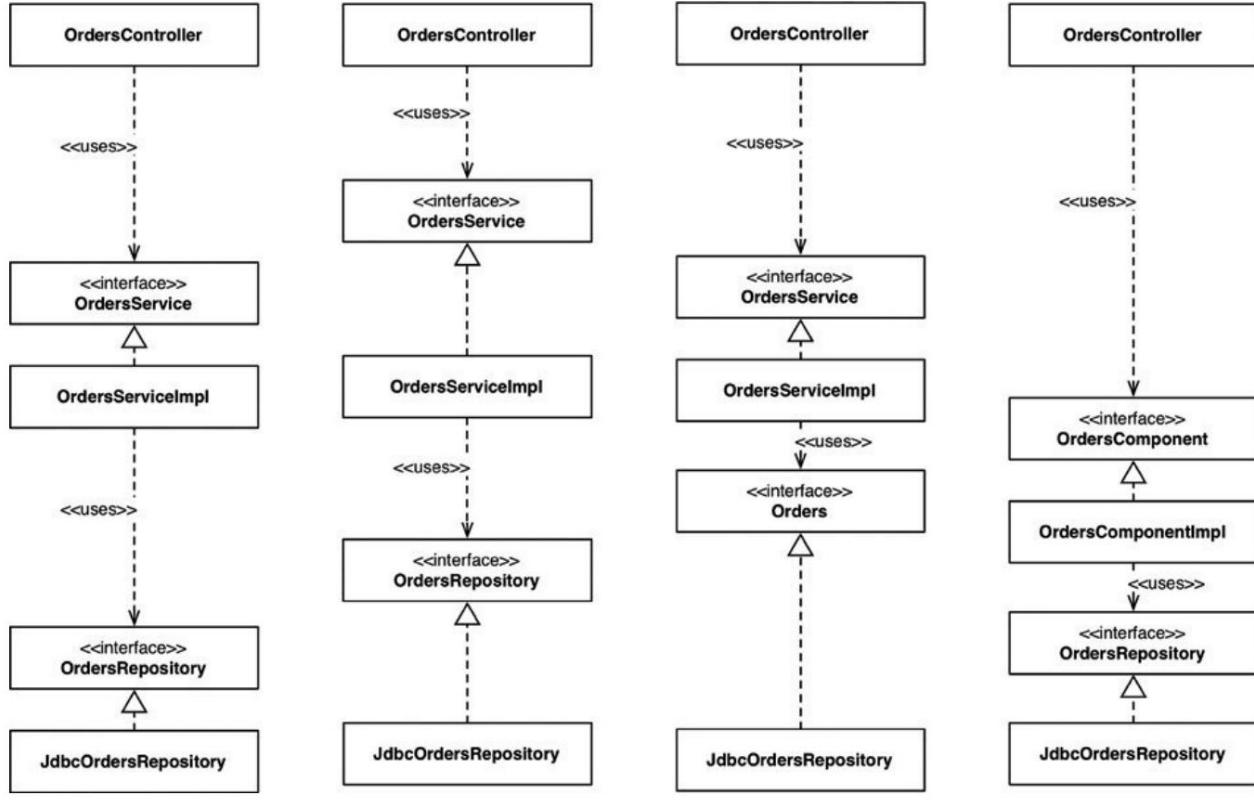


Figura 34.7 Los cuatro enfoques arquitectónicos son iguales

Moviéndose de izquierda a derecha, en el enfoque "paquete por capa", las interfaces OrdersService y OrdersRepository deben ser públicas, porque tienen dependencias entrantes de clases fuera de su paquete de definición. Por el contrario, las clases de implementación (OrdersServiceImpl y JdbcOrdersRepository) se pueden hacer más restrictivas (paquete protegido). Nadie necesita saber sobre ellos; son un detalle de implementación.

En el enfoque de "paquete por característica", OrdersController proporciona el único punto de entrada al paquete, por lo que todo lo demás puede protegerse como paquete. La gran advertencia aquí es que nada más en el código base, fuera de este paquete, puede acceder a la información relacionada con los pedidos a menos que pasen por el controlador. Esto puede ser deseable o no.

En el enfoque de puertos y adaptadores, las interfaces OrdersService y Orders tienen dependencias entrantes de otros paquetes, por lo que deben hacerse públicas. Nuevamente, las clases de implementación se pueden proteger con paquetes y se pueden inyectar dependencias en tiempo de ejecución.

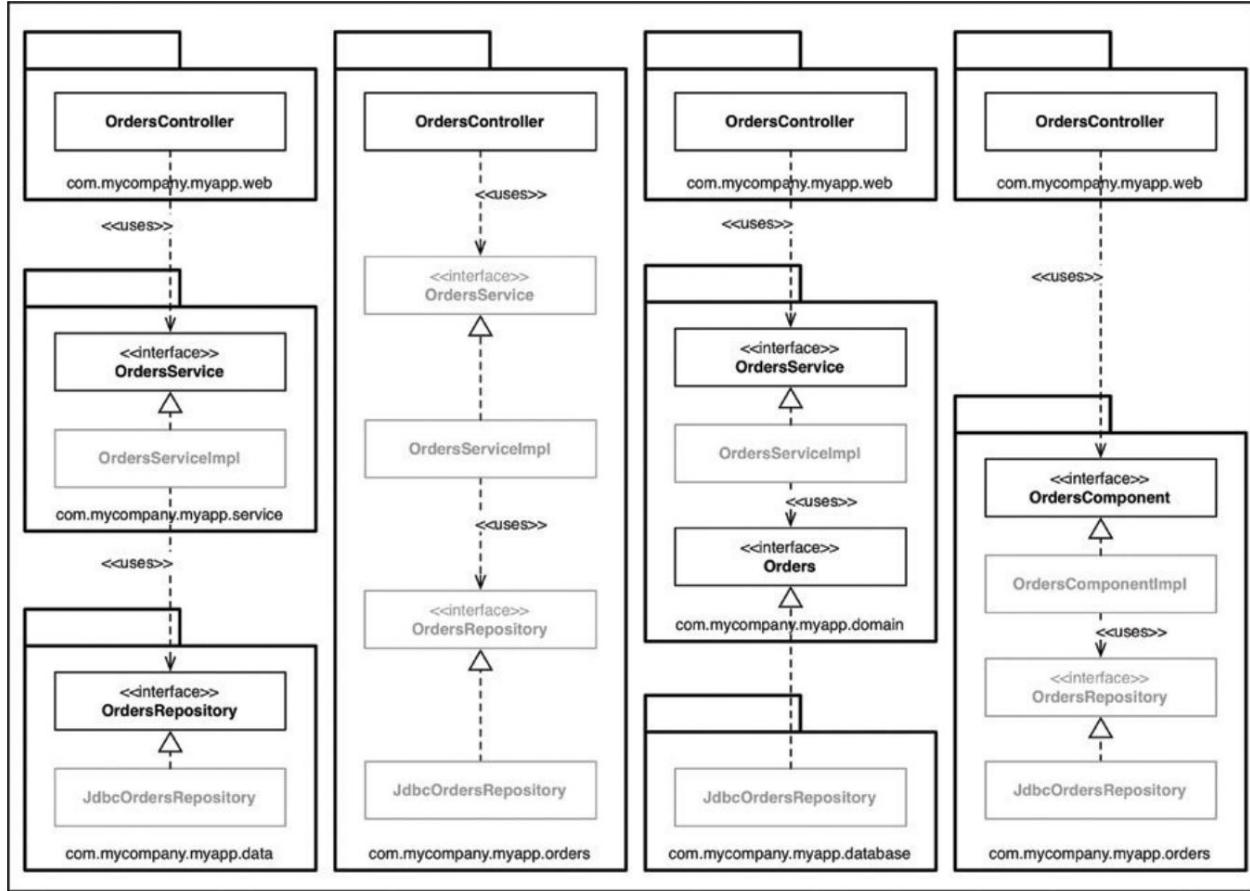


Figura 34.8 Los tipos atenuados son donde el modificador de acceso se puede hacer más restrictivo

Finalmente, en el enfoque de “paquete por componente”, la interfaz OrdersComponent tiene una dependencia entrante del controlador, pero todo lo demás se puede proteger mediante paquetes. Cuantos menos tipos públicos tenga, más pequeños serán los [9](#) que codifiquen dependencias potenciales. Ahora no hay forma de que el paquete [fuerza](#) este número de pueda usar la interfaz o implementación de OrdersRepository directamente, por lo que podemos confiar en el compilador para hacer cumplir este principio arquitectónico. Puede hacer lo mismo en .NET con la palabra clave interna , aunque necesitará crear un ensamblaje separado para cada componente.

Para que quede absolutamente claro, lo que he descrito aquí se relaciona con una aplicación monolítica, donde todo el código reside en un único árbol de código fuente. Si está creando una aplicación de este tipo (y muchas personas lo están), sin duda le recomiendo que se apoye en el compilador para hacer cumplir sus principios arquitectónicos, en lugar de depender de la autodisciplina y las herramientas posteriores a la compilación.

OTROS MODOS DE DESACOPLAMIENTO

Además del lenguaje de programación que estás utilizando, a menudo existen otras formas de desacoplar las dependencias del código fuente. Con Java, tiene marcos de módulos como OSGi y el nuevo sistema de módulos Java 9. Con los sistemas de módulos, cuando se usan correctamente, se puede hacer una distinción entre los tipos que son públicos y los tipos que se publican. Por ejemplo, podría crear un módulo de Pedidos donde todos los tipos estén marcados como públicos, pero publicar solo un pequeño subconjunto de esos tipos para consumo externo. Ha tardado mucho en llegar, pero estoy entusiasmado de que el sistema de módulos Java 9 nos brinde otra herramienta para crear un mejor software y despierte el interés de la gente en el pensamiento de diseño una vez más.

Otra opción es desacoplar sus dependencias a nivel del código fuente, dividiendo el código en diferentes árboles de código fuente. Si tomamos el ejemplo de puertos y adaptadores, podríamos tener tres árboles de código fuente:

- El código fuente para el negocio y el dominio (es decir, todo lo que es independiente de la tecnología y las opciones de marco): OrdersService, OrdersServiceImpl y Orders.
- El código fuente para la web: OrdersController.
- El código fuente para la persistencia de datos: JdbcOrdersRepository

Los dos últimos árboles de código fuente dependen en tiempo de compilación del código comercial y de dominio, que a su vez no sabe nada sobre la web o el código de persistencia de datos. Desde una perspectiva de implementación, puede hacerlo configurando módulos o proyectos separados en su herramienta de compilación (por ejemplo, Maven, Gradle, MSBuild). Lo ideal sería repetir este patrón, teniendo un árbol de código fuente separado para todos y cada uno de los componentes de su aplicación. Sin embargo, esta es una solución muy idealista, porque existen problemas de rendimiento, complejidad y mantenimiento en el mundo real asociados con dividir el código fuente de esta manera.

Un enfoque más simple que algunas personas siguen para el código de sus puertos y adaptadores es tener solo dos árboles de código fuente:

- Código de dominio (el “interior”)
- Código de infraestructura (el “exterior”)

Esto se corresponde muy bien con el diagrama ([Figura 34.9](#)) que mucha gente usa para resumir la arquitectura de puertos y adaptadores, y existe una dependencia en tiempo de compilación desde la infraestructura hasta el dominio.

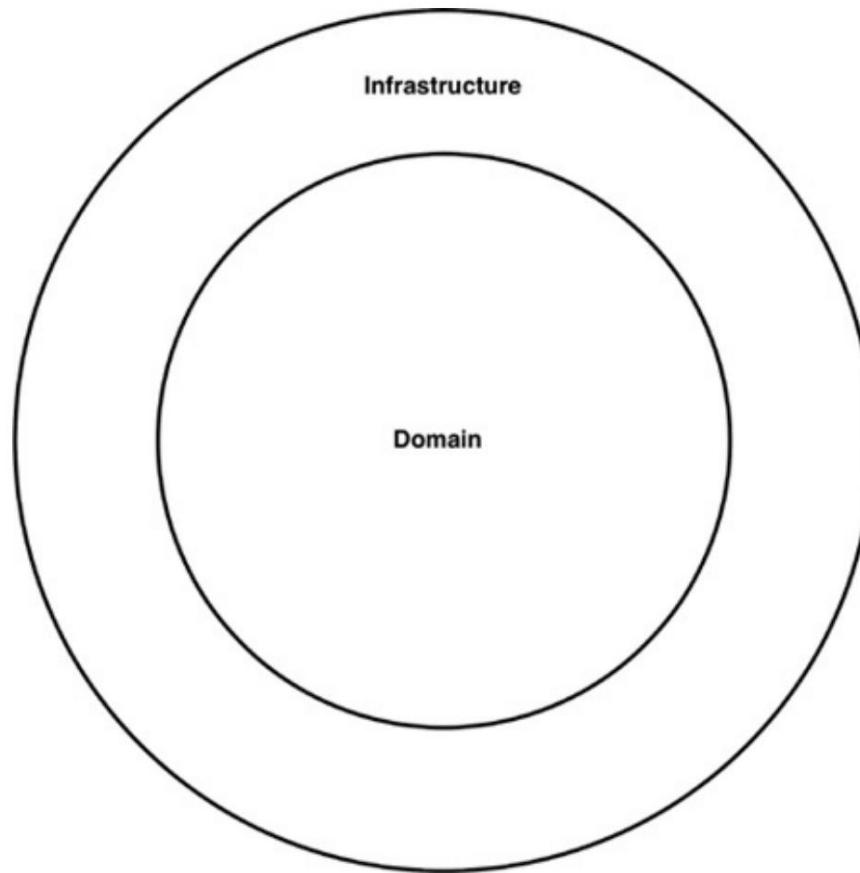


Figura 34.9 Código de dominio y de infraestructura

Este enfoque para organizar el código fuente también funcionará, pero tenga en cuenta las posibles compensaciones. Es lo que yo llamo el “antipatrón periférico de puertos y adaptadores”. La ciudad de París, Francia, cuenta con una vía de circunvalación llamada Boulevard Périphérique, que permite circumnavegar París sin adentrarse en las complejidades de la ciudad. Tener todo su código de infraestructura en un único árbol de código fuente significa que es potencialmente posible que el código de infraestructura en un área de su aplicación (por ejemplo, un controlador web) llame directamente al código en otra área de su aplicación (por ejemplo, un repositorio de base de datos). , sin navegar por el dominio. Esto es especialmente cierto si olvidó aplicar los modificadores de acceso apropiados a ese código.

CONCLUSIÓN: EL CONSEJO QUE FALTA

El objetivo de este capítulo es resaltar que sus mejores intenciones de diseño pueden destruirse en un instante si no considera las complejidades de la estrategia de implementación. Piense en cómo asignar el diseño deseado a las estructuras de código, cómo organizar ese código y qué modos de desacoplamiento aplicar durante el tiempo de ejecución y el tiempo de compilación. Deje opciones abiertas cuando corresponda, pero sea pragmático y tenga en cuenta el tamaño de su equipo, su nivel de habilidad y la complejidad de la solución junto con sus limitaciones de tiempo y presupuesto. Piense también en utilizar su compilador para ayudarle a aplicar el estilo arquitectónico elegido y tenga cuidado con el acoplamiento en otras áreas, como los modelos de datos. El diablo está en los detalles de implementación.

1. Podría decirse que esta es una forma horrible de nombrar una clase, pero como veremos más adelante, quizás en realidad no importe. 2. <https://martinfowler.com/bliki/PresentationDomainDataLayering.html>.
3. Este beneficio es mucho menos relevante con las funciones de navegación de los IDE modernos, pero parece que ha habido Ha habido un renacimiento al volver a los editores de texto livianos, por razones que claramente soy demasiado mayor para entender.
4. Mi primer trabajo después de graduarme de la universidad en 1996 fue crear aplicaciones de escritorio cliente-servidor. con una tecnología llamada PowerBuilder, un 4GL superproductivo que se destacó en la creación de aplicaciones basadas en bases de datos. Un par de años más tarde, estaba creando aplicaciones cliente-servidor con Java, donde teníamos que crear nuestra propia conectividad de base de datos (esto era antes de JDBC) y nuestros propios kits de herramientas GUI además de AWT. ¡Eso es "progreso" para usted!
5. En el patrón Segregación de responsabilidad de consulta de comando , tiene patrones separados para actualizar y leyendo datos.
6. <http://www.laputan.org/mud/> 7. Ver <https://www.structurizr.com/help/c4> para más información.
8. En Java, por ejemplo, aunque tendemos a pensar que los paquetes son jerárquicos, no es posible crear restricciones de acceso basadas en una relación de paquete y subpaquete. Cualquier jerarquía que cree está en el nombre de esos paquetes y en la estructura de directorios en el disco únicamente.
9. A menos que hagas trampa y uses el mecanismo de reflexión de Java, ¡pero no hagas eso!

VII

Apéndice

A

ARQUITECTURA ARQUEOLOGÍA



Para descubrir los principios de la buena arquitectura, hagamos un viaje de 45 años a través de algunos de los proyectos en los que he trabajado desde 1970. Algunos de estos proyectos son interesantes desde el punto de vista arquitectónico. Otros son interesantes por las lecciones aprendidas y por cómo alimentaron proyectos posteriores.

Este apéndice es algo autobiográfico. He tratado de mantener la discusión relevante al tema de la arquitectura; pero, como ocurre con todo lo autobiográfico, a veces intervienen otros factores. ;-)

SISTEMA DE CONTABILIDAD DE LA UNIÓN

A finales de la década de 1960, una empresa llamada ASC Tabulated firmó un contrato con el Local 705 del Teamsters Union para proporcionar un sistema de contabilidad. La computadora en la que ASC eligió implementar este sistema fue una GE Datanet 30, como se muestra en la Figura A.1.



Figura A.1 GE Datanet 30

Cortesía de Ed Thelen, ed-thelen.org

Como se puede ver en la imagen, esta era una habitación ¹ máquina. Llenó una habitación y enorme y necesitaba estrictos controles ambientales.

Esta computadora fue construida en la época anterior a los circuitos integrados. Fue construido con transistores discretos. Incluso había algunas válvulas de vacío (aunque sólo en el sentido de amplificadores de las unidades de cinta).

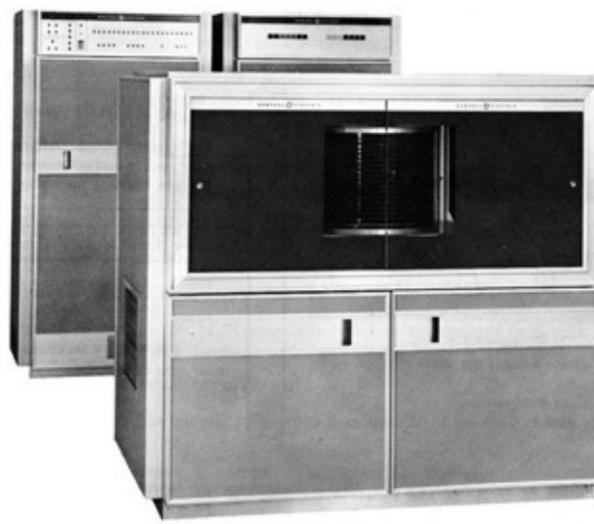
Según los estándares actuales, la máquina era enorme, lenta, pequeña y primitiva. Tenía 16K × 18 bits de núcleo, con un tiempo de ciclo de unos 7 microsegundos. Sala ² Llenó un gran ambientalmente controlada. Tenía unidades de cinta magnética de 7 pistas y una unidad de disco con una capacidad de aproximadamente 20 megabytes.

Ese disco era un monstruo. Puedes verlo en la imagen de la Figura A.2, pero eso no te da la escala de la bestia. La parte superior de ese gabinete estaba sobre mi cabeza. Los platos tenían 36 pulgadas de diámetro y 3/8 de pulgada de grosor. Uno de los platos se muestra en la Figura A.3.

Ahora cuenta los platos en esa primera imagen. Hay más de una docena. Cada

uno tenía su propio brazo de búsqueda individual impulsado por actuadores neumáticos. Se podía ver cómo esas cabezas buscadoras se movían por los platos. El tiempo de búsqueda fue probablemente de medio segundo a un segundo.

Cuando se encendía esta bestia, sonaba como un motor a reacción. El suelo retumbaría y temblaría hasta que alcanzara la velocidad. [3](#)



MASS RANDOM ACCESS DATA STORAGE UNIT

Figura A.2 La unidad de almacenamiento de datos con sus platos.

Cortesía de Ed Thelen, ed-thelen.org

El gran reclamo a la fama del Datanet 30 fue su capacidad para controlar una gran cantidad de terminales asíncronos a una velocidad relativamente alta. Eso es exactamente lo que necesitaba ASC.

ASC tenía su sede en Lake Bluff, Illinois, 30 millas al norte de Chicago. La oficina del Local 705 estaba en el centro de Chicago. El sindicato quería que aproximadamente una docena de sus empleados de entrada de [4](#) terminales ([Figura A.4](#)) para ingresar datos en el sistema. Los datos usarán CRT. Imprimirán informes en teletipos ASR35 ([Figura A.5](#)).



Figura A.3 Un plato de ese disco: 3/8 de pulgada de espesor, 36 pulgadas de diámetro

Cortesía de Ed Thelen, ed-thelen.org

Los terminales CRT funcionaban a 30 caracteres por segundo. Esta era una tasa bastante buena para finales de la década de 1960 porque los módems eran relativamente poco sofisticados en aquellos días.

ASC alquiló una docena de líneas telefónicas dedicadas y el doble de módems de 300 baudios de la compañía telefónica para conectar el Datnet 30 a estos terminales.

Estas computadoras no venían con sistemas operativos. Ni siquiera venían con sistemas de archivos. Lo que obtuviste fue un ensamblador.

Si necesitaba almacenar datos en el disco, almacenó datos en el disco. No en un archivo. No en un directorio. Descubriste en qué pista, plato y sector colocar los datos y luego operaste el disco para colocar los datos allí. Sí, eso significa que escribimos nuestro propio controlador de disco.



Figura A.4 Terminal CRT de punto de datos

Cortesía de Bill Degnan, vintagecomputer.net

El sistema de contabilidad sindical tenía tres tipos de registros: agentes, empleadores y miembros. El sistema era un sistema CRUD para estos registros, pero también incluía operaciones para contabilizar cuotas, calcular cambios en el libro mayor, etc.

El sistema original fue escrito en ensamblador por un consultor que de alguna manera logró meter todo en 16K.

Como se puede imaginar, esa gran Datanet 30 era una máquina costosa de operar y mantener. El consultor de software que mantenía el software en funcionamiento también era caro. Es más, las minicomputadoras se estaban volviendo populares y eran mucho más baratas.



Figura A.5 Teletipo ASR35

Joe Mabel, con permiso

En 1971, cuando tenía 18 años, ASC nos contrató a mí y a dos de mis amigos geek para reemplazar todo el sistema de contabilidad del sindicato por uno basado en una minicomputadora Varian 620/f ([Figura A.6](#)). La computadora era barata. Éramos tacaños. Por tanto, parecía un buen negocio para ASC.

La máquina Varian tenía un bus de 16 bits y una memoria de $32K * 16$ núcleos. Tenía un tiempo de ciclo de aproximadamente 1 microsegundo. Era mucho más potente que el Datanet 30. Utilizaba la exitosa tecnología de disco 2314 de IBM, lo que nos permitía almacenar 30 megabytes en platos que tenían sólo 14 pulgadas de diámetro y que no podían explotar a través de paredes de bloques de hormigón.

Por supuesto, todavía no teníamos sistema operativo. Sin sistema de archivos. Sin lenguaje de alto nivel. Todo lo que teníamos era un ensamblador. Pero nos las arreglamos.

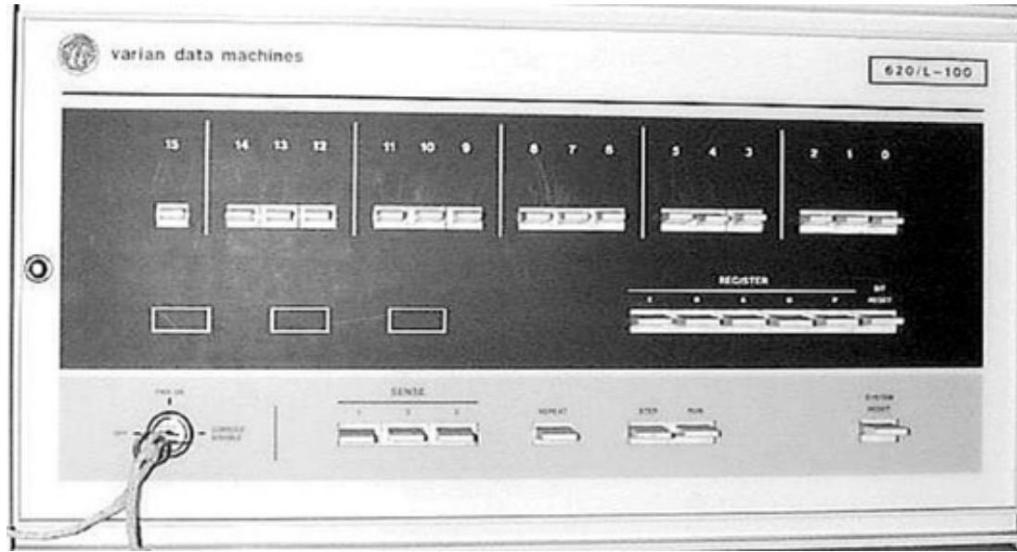


Figura A.6 Minicomputadora Varian 620/f

El orfanato de minicomputadoras

En lugar de intentar meter todo el sistema en 32K, creamos un sistema superpuesto. Las aplicaciones se cargarían desde el disco en un bloque de memoria dedicado a superposiciones. Se ejecutarían en esa memoria y, de forma preventiva, se volverían a intercambiar en el disco, con su RAM local, para permitir la ejecución de otros programas.

Los programas se intercambiarían en el área de superposición, se ejecutarían lo suficiente para llenar los búferes de salida y luego se intercambiarían para poder intercambiar otro programa.

Por supuesto, cuando su interfaz de usuario se ejecuta a 30 caracteres por segundo, sus programas pasan mucho tiempo esperando. Tuvimos mucho tiempo para intercambiar los programas dentro y fuera del disco para mantener todos los terminales funcionando lo más rápido posible. Nadie se quejó nunca de problemas de tiempo de respuesta.

Escribimos un supervisor preventivo que gestionaba las interrupciones y las IO. Escribimos las aplicaciones; escribimos los controladores de disco, los controladores de terminal, los controladores de cinta y todo lo demás en ese sistema. No hubo un solo bit en ese sistema que no escribíramos. Aunque fue una lucha que implicó demasiadas semanas de 80 horas, pusimos la bestia en funcionamiento en cuestión de 8 o 9 meses.

La arquitectura del sistema era simple ([Figura A.7](#)). Cuando se iniciaba una aplicación, generaba resultados hasta que el búfer de su terminal particular estaba lleno.

Luego, el supervisor cambiaría la aplicación e introduciría una nueva aplicación. El supervisor continuaría goteando el contenido del buffer del terminal a 30 cps hasta que estuviera casi vacío. Luego volvería a cambiar la aplicación para llenar el búfer nuevamente.

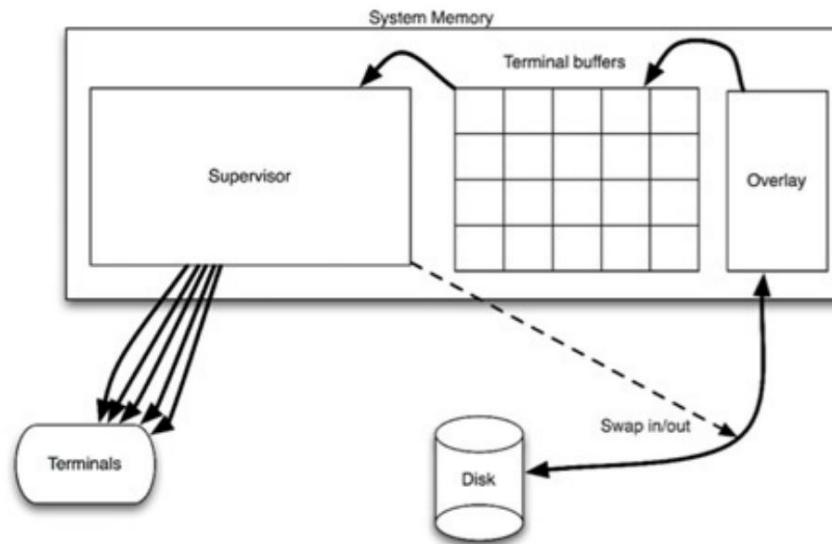


Figura A.7 La arquitectura del sistema

Hay dos límites en este sistema. El primero es el límite de salida de caracteres. Las aplicaciones no tenían idea de que su salida iba a una terminal de 30 cps. De hecho, la salida de caracteres era completamente abstracta desde el punto de vista de las aplicaciones. Las aplicaciones simplemente pasaban cadenas al supervisor, y el supervisor se encargaba de cargar los buffers, enviar los caracteres a las terminales e intercambiar las aplicaciones dentro y fuera de la memoria.

Este límite era la dependencia normal, es decir, dependencias apuntadas con el flujo de control. Las aplicaciones tenían dependencias en tiempo de compilación del supervisor y el flujo de control pasaba de las aplicaciones al supervisor. El límite impedía que las aplicaciones supieran a qué tipo de dispositivo se dirigía la salida.

El segundo límite fue la dependencia invertida. El supervisor podía iniciar las aplicaciones, pero no dependía de ellas en tiempo de compilación. El flujo de control pasó del supervisor a las aplicaciones. La interfaz polimórfica que invirtió la dependencia era simplemente esta: cada aplicación se iniciaba saltando exactamente a la misma dirección de memoria dentro del área de superposición. El límite impidió que el supervisor supiera algo sobre el

aplicaciones distintas al punto de partida.

RECORTE LÁSER

En 1973, me uní a una empresa en Chicago llamada Teradyne Applied Systems (TAS). Esta era una división de Teradyne Inc., que tenía su sede en Boston. Nuestro producto era un sistema que utilizaba láseres de potencia relativamente alta para recortar componentes electrónicos con tolerancias muy finas.

En aquellos días, los fabricantes serigrafiaban componentes electrónicos sobre sustratos cerámicos. Esos sustratos eran del orden de 1 pulgada cuadrada. Los componentes eran típicamente resistencias, dispositivos que resisten el flujo de corriente.

La resistencia de una resistencia depende de varios factores, entre ellos su composición y su geometría. Cuanto más ancha es la resistencia, menos resistencia tiene.

Nuestro sistema colocaría el sustrato cerámico en un arnés que tenía sondas que hacían contacto con las resistencias. El sistema mediría la resistencia de las resistencias y luego usaría un láser para quemar partes de la resistencia, haciéndola cada vez más delgada hasta alcanzar el valor de resistencia deseado dentro de una décima parte de un por ciento más o menos.

Vendimos estos sistemas a los fabricantes. También utilizamos algunos sistemas internos para recortar lotes relativamente pequeños para pequeños fabricantes.

La computadora era una M365. Esto fue en la época en que muchas empresas construían sus propias computadoras: Teradyne construyó el M365 y lo suministró a todas sus divisiones. El M365 era una versión mejorada de un PDP-8, un miniordenador popular de la época.

El M365 controlaba la mesa de posicionamiento, que movía los sustratos cerámicos debajo de las sondas. Controlaba el sistema de medición y el láser. El láser se posicionó mediante espejos XY que podían girar bajo el control del programa. La computadora también podría controlar la configuración de potencia del láser.

El entorno de desarrollo del M365 era relativamente primitivo. No había ningún disco. El almacenamiento masivo se hacía en cartuchos de cinta que parecían viejos cassetes de cinta de audio de 8 pistas. Las cintas y unidades fueron fabricadas por Tri-Data.

Al igual que los casetes de audio de 8 pistas de la época, la cinta estaba orientada en bucle. La unidad movía la cinta en una sola dirección: ¡no había rebobinado! Si querías colocar la cinta al principio, tenías que enviarla hacia adelante hasta su "punto de carga".

La cinta se movía a una velocidad de aproximadamente 1 pie por segundo. Por lo tanto, si el bucle de cinta tuviera 25 pies de largo, podría tomar hasta 25 segundos enviarlo al punto de carga. Por esta razón, Tridata fabricó cartuchos en varias longitudes, desde 10 a 100 pies.

El M365 tenía un botón en el frente que cargaba la memoria con un pequeño programa de arranque y lo ejecutaba. Este programa leería el primer bloque de datos de la cinta y lo ejecutaría. Normalmente, este bloque contenía un cargador que cargaba el sistema operativo que se encontraba en el resto de la cinta.

El sistema operativo solicitará al usuario el nombre de un programa para ejecutar. Esos programas se almacenaron en la cinta, justo después del sistema operativo. Escribiríamos el nombre del programa (por ejemplo, ED-402 Editor) y el sistema operativo buscaría ese programa en la cinta, lo cargaría y lo ejecutaría.

La consola era un CRT ASCII con fósforos verdes, 72 caracteres de ancho y 5 por 24 líneas. Todos los caracteres estaban en mayúsculas.

Para editar un programa, cargaría el editor ED-402 y luego insertaría la cinta que contenía su código fuente. Leerías un bloque de cinta de ese código fuente en la memoria y se mostraría en la pantalla. El bloque de cinta puede contener 50 líneas de código. Realizaría sus ediciones moviendo el cursor por la pantalla y escribiendo de manera similar a vi. Cuando haya terminado, escribirá ese bloque en una cinta diferente y leerá el siguiente bloque de la cinta fuente. Seguiste haciendo esto hasta que terminaste.

No era posible retroceder a los bloques anteriores. Editaste tu programa en línea recta, de principio a fin. Volver al principio te obligaba a terminar de copiar el código fuente en la cinta de salida y luego iniciar una nueva sesión de edición en esa cinta. Quizás no sea sorprendente que, dadas estas limitaciones, imprimimos nuestros programas en papel, marcamos todas las ediciones a mano con tinta roja y luego editamos nuestros programas bloque por bloque consultando nuestras marcas en la lista.

Una vez editado el programa volvimos al SO e invocamos al ensamblador.

El ensamblador leyó la cinta del código fuente y escribió una cinta binaria, al mismo tiempo que producía un listado en nuestra impresora de línea de productos de datos.

Las cintas no eran 100% confiables, por lo que siempre escribíamos dos cintas al mismo tiempo.

De esa forma, al menos uno de ellos tenía una alta probabilidad de estar libre de errores.

Nuestro programa tenía aproximadamente 20.000 líneas de código y tardó casi 30 minutos en compilarse. Las probabilidades de que obtuviéramos un error de lectura de cinta durante ese tiempo eran aproximadamente de 1 en 10. Si el ensamblador recibía un error de cinta, sonaría el timbre de la consola y luego comenzaría a imprimir una serie de errores en la impresora. Se podía oír esta enloquecedora campana por todo el laboratorio. También se podían escuchar las maldiciones del pobre programador que acababa de enterarse de que era necesario comenzar la compilación de 30 minutos. encima.

La arquitectura del programa era típica de aquellos días. Existía un Programa Maestro Operativo, apropiadamente llamado "el MOP". Su trabajo consistía en gestionar funciones de E/S básicas y proporcionar los rudimentos de un "shell" de consola. Muchas de las divisiones de Teradyne compartían el código fuente de MOP, pero cada una lo había bifurcado para sus propios usos. En consecuencia, nos enviaríamos actualizaciones del código fuente entre nosotros en forma de listados marcados que luego integraríamos manualmente (y con mucho cuidado).

Una capa de servicios especiales controlaba el hardware de medición, las mesas de posicionamiento y el láser. El límite entre esta capa y el MOP era, en el mejor de los casos, confuso. Si bien la capa de servicios públicos se llamaba MOP, el MOP se había modificado específicamente para esa capa y, a menudo, se le volvía a llamar. De hecho, realmente no pensamos en estas dos capas como capas separadas. Para nosotros, era sólo un código que agregamos al MOP de una manera altamente acoplada.

Luego vino la capa de aislamiento. Esta capa proporcionaba una interfaz de máquina virtual para los programas de aplicación, que estaban escritos en un lenguaje basado en datos (DSL) específico de un dominio completamente diferente. El lenguaje tenía operaciones para mover el láser, mover la mesa, realizar cortes, realizar mediciones, etc. Nuestros clientes escribirían sus programas de aplicación de corte por láser en este lenguaje y la capa de aislamiento los ejecutaría.

Este enfoque no pretendía crear un lenguaje de corte láser independiente de la máquina. De hecho, el idioma tenía muchas idiosincrasias que eran profundamente

acoplado a las capas inferiores. Más bien, este enfoque proporcionó a los programadores de aplicaciones un lenguaje "más simple" que el ensamblador M356 para programar sus trabajos de recorte.

Los trabajos de recorte podrían cargarse desde la cinta y ejecutarse mediante el sistema. Básicamente, nuestro sistema era un sistema operativo para aplicaciones de recorte.

El sistema fue escrito en ensamblador M365 y compilado en una única unidad de compilación que producía código binario absoluto.

Los límites en esta aplicación eran, en el mejor de los casos, suaves. Incluso no se cumplió bien el límite entre el código del sistema y las aplicaciones escritas en DSL.

Había acoplamientos por todas partes.

Pero eso era típico del software de principios de los años 1970.

MONITOREO DE ALUMINIO FUNDIDO A PRESIÓN

A mediados de la década de 1970, mientras la OPEP imponía un embargo al petróleo y la escasez de gasolina provocaba que los conductores enojados se pelearan en las gasolineras, comencé a trabajar en Outboard Marine Corporation (OMC). Esta es la empresa matriz de las cortadoras de césped Johnson Motors y Lawnboy.

OMC mantenía una enorme instalación en Waukegan, Illinois, para crear piezas de aluminio fundido a presión para todos los motores y productos de la empresa. El aluminio se fundía en enormes hornos y luego se transportaba en grandes cubos a docenas y docenas de máquinas de fundición de aluminio operadas individualmente. Cada máquina tenía un operador humano responsable de colocar los moldes, hacer funcionar la máquina y extraer las piezas recién fundidas. A estos operadores se les pagaba en función de la cantidad de piezas que producían.

Me contrataron para trabajar en un proyecto de automatización de un taller. OMC había comprado un IBM System/7, que era la respuesta de IBM a la minicomputadora. Ataron esta computadora a todas las máquinas de fundición en el piso, para que pudieramos contar y cronometrar los ciclos de cada máquina. Nuestra función era recopilar toda esa información y presentarla en 3270 pantallas verdes.

El lenguaje era ensamblador. Y, nuevamente, cada fragmento de código que se ejecutó en esta computadora fue código que escribimos nosotros. No había ningún sistema operativo, ninguna subrutina.

bibliotecas y sin marco. Era sólo código sin formato.

También era un código en tiempo real controlado por interrupciones. Cada vez que una máquina de fundición funcionaba, teníamos que actualizar un lote de estadísticas y enviar mensajes a un gran IBM 370 en el cielo, ejecutando un programa CICS-COBOL que presentaba esas estadísticas en las pantallas verdes.

Odiaba este trabajo. Oh, vaya, sí. ¡Oh, el trabajo fue divertido! Pero la cultura dice ... Satisfacer que debía usar corbata.

Ah, lo intenté. Realmente lo hice. Pero era evidente que no estaba contento trabajando allí y mis compañeros lo sabían. Lo sabían porque no podía recordar fechas críticas ni lograr levantarme lo suficientemente temprano para asistir a reuniones importantes. Este fue el único trabajo de programación del que me despidieron y me lo merecía.

Desde un punto de vista arquitectónico, no hay mucho que aprender aquí excepto una cosa. El System/7 tenía una instrucción muy interesante llamada establecer interrupción de programa (SPI). Esto le permitió activar una interrupción del procesador, permitiéndole manejar cualquier otra interrupción en cola de menor prioridad. Hoy en día, en Java lo llamamos Thread.yield().

4-TELÉFONO

En octubre de 1976, después de haber sido despedido de OMC, regresé a una división diferente de Teradyne, una división en la que permanecería durante 12 años. El producto en el que trabajé se llamó 4-TEL. Su propósito era probar todas las líneas telefónicas en un área de servicio telefónico, todas las noches, y producir un informe de todas las líneas que requerían reparación. También permitió al personal de pruebas telefónicas probar líneas telefónicas específicas en detalle.

Este sistema comenzó su vida con el mismo tipo de arquitectura que el sistema Laser Trim. Era una aplicación monolítica escrita en lenguaje ensamblador sin límites significativos. Pero cuando me uní a la empresa, eso estaba a punto de cambiar.

El sistema fue utilizado por probadores ubicados en un centro de servicio (SC). Un centro de servicios cubría muchas oficinas centrales (CO), cada una de las cuales podía manejar hasta 10.000 líneas telefónicas. El hardware de marcación y medición tenía que estar ubicado dentro del CO. Ahí es donde se colocaron las computadoras M365. los llamamos

computa los probadores de línea de la oficina central (COLT). Se colocó otro M365 en el SC; se llamó computadora del área de servicio (SAC). El SAC tenía varios módems que podía utilizar para marcar los COLT y comunicarse a 300 baudios (30 cps).

Al principio, las computadoras COLT hacían todo, incluidas todas las comunicaciones, menús e informes de la consola. El SAC era simplemente un multiplexor simple que tomaba la salida de los COLT y la colocaba en una pantalla.

El problema con esta configuración fue que 30 cps es realmente lento. A los evaluadores no les gustó ver a los personajes deslizándose por la pantalla, especialmente porque solo estaban interesados en unos pocos datos clave. Además, en aquellos días, la memoria central del M365 era cara y el programa era grande.

La solución fue separar la parte del software que marcaba y medía líneas de la parte que analizaba los resultados e imprimía los informes. Estos últimos pasarían al SAC y los primeros permanecerían en los COLT. Esto permitiría que el COLT fuera una máquina más pequeña, con mucha menos memoria, y aceleraría mucho la respuesta en el terminal, ya que los informes se generarían en el SAC.

El resultado fue notablemente exitoso. Las actualizaciones de pantalla fueron muy rápidas (una vez que se marcó el COLT apropiado) y la huella de memoria de los COLT se redujo mucho.

La frontera estaba muy limpia y muy desacoplada. Se intercambiaron paquetes de datos muy cortos entre el SAC y COLT. Estos paquetes eran una forma muy simple de DSL y representaban comandos primitivos como "DIAL XXXX" o "MEASURE".

El M365 se cargó desde una cinta. Esas unidades de cinta eran caras y no muy fiables, especialmente en el entorno industrial de una central telefónica. Además, el M365 era una máquina cara en comparación con el resto de la electrónica del COLT. Entonces nos embarcamos en un proyecto para reemplazar el M365 con una microcomputadora basada en un procesador 8085 μ.

La nueva computadora estaba compuesta por una placa procesadora que contenía el 8085, una placa RAM que contenía 32K de RAM y tres placas ROM que contenían 12K de memoria de sólo lectura cada una. Todas estas placas encajan en el mismo chasis que el hardware de medición, eliminando así el voluminoso chasis adicional que tenía

albergaba el M365.

Las placas ROM contenían 12 chips Intel 2708 EPROM (memoria de solo lectura programable y borrable). esos chips ⁶ [La figura A.8](#) muestra un ejemplo de dicho chip. cargamos con software insertándolos en dispositivos especiales llamados quemadores PROM que fueron impulsados por nuestro entorno de desarrollo. Las fichas podrían ser borrados exponiéndolos a luz ultravioleta de alta intensidad. ⁷

Mi amigo CK y yo tradujimos el programa de lenguaje ensamblador M365 para COLT al lenguaje ensamblador 8085. Esta traducción se hizo a mano y nos llevó unos 6 meses. El resultado final fue aproximadamente 30K de código 8085.

Nuestro entorno de desarrollo tenía 64K de RAM y ninguna ROM, por lo que pudimos descargar rápidamente nuestros archivos binarios compilados en la RAM y probarlos.

Una vez que el programa funcionó, pasamos a usar las EPROM. Quemamos 30 chips y los insertamos en las ranuras correctas de las tres placas ROM. Cada chip estaba etiquetado para que pudiéramos saber qué chip iba en cada ranura.

El programa de 30K era un binario único, de 30K de longitud. Para grabar los chips, simplemente dividimos esa imagen binaria en 30 segmentos diferentes de 1K y grabamos cada segmento en el chip etiquetado apropiadamente.

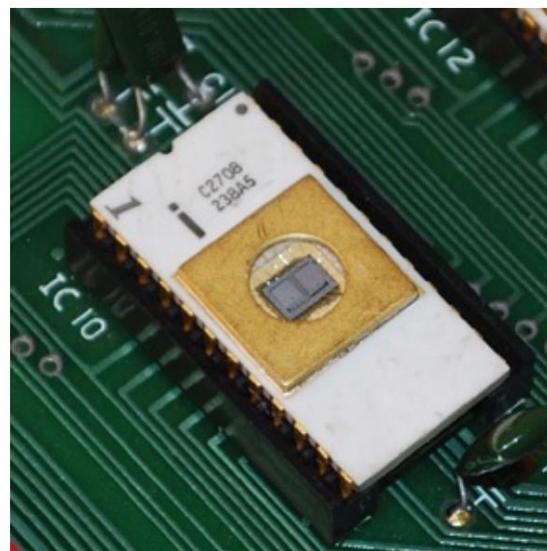


Figura A.8 Chip EPROM

Esto funcionó muy bien y comenzamos a producir en masa el hardware y a implementarlo.

el sistema en el campo.

Pero el software es blando. Era necesario agregar [8](#) funciones. Era necesario corregir errores. Y a medida que la base instalada crecía, la logística de actualizar el software quemando 30 chips por instalación y haciendo que el personal de servicio de campo reemplazara los 30 chips en cada sitio se convirtió en una pesadilla.

Hubo todo tipo de problemas. A veces los chips estaban mal etiquetados o las etiquetas se caían. A veces, el ingeniero de servicio de campo reemplazaba por error el chip equivocado. A veces, el ingeniero de servicio de campo, sin darse cuenta, rompía un pasador de uno de los nuevos chips. En consecuencia, los ingenieros de campo tuvieron que llevar consigo extras de los 30 chips.

¿Por qué tuvimos que cambiar los 30 chips? Cada vez que agregamos o eliminamos código de nuestro ejecutable de 30K, cambiaba las direcciones en las que se cargaba cada instrucción. También cambió las direcciones de las subrutinas y funciones que llamamos. Así que todos los chips se vieron afectados, sin importar cuán trivial fuera el cambio.

Un día, mi jefe vino a verme y me pidió que resolviera ese problema. Dijo que necesitábamos una forma de realizar cambios en el firmware sin reemplazar los 30 chips cada vez. Hicimos una lluvia de ideas sobre este tema durante un tiempo y luego nos embarcamos en el proyecto "Vectorización". Me tomó tres meses.

La idea era maravillosamente simple. Dividimos el programa de 30K en 32 archivos fuente compilables de forma independiente, cada uno de menos de 1K. Al comienzo de cada archivo fuente, le indicamos al compilador en qué dirección cargar el programa resultante (por ejemplo, ORG C400 para el chip que se iba a insertar en la posición C4).

Además, al comienzo de cada archivo fuente, creamos una estructura de datos simple de tamaño fijo que contenía todas las direcciones de todas las subrutinas en ese chip. Esta estructura de datos tenía 40 bytes de longitud, por lo que no podía contener más de 20 direcciones. Esto significaba que ningún chip podía tener más de 20 subrutinas.

A continuación, creamos un área especial en la RAM conocida como vectores. Contenía 32 tablas de 40 bytes: RAM exactamente suficiente para contener los punteros al inicio de cada chip.

Finalmente, cambiamos cada llamada a cada subrutina en cada chip a una llamada indirecta a través del vector RAM apropiado.

Cuando nuestro procesador arrancaba, escaneaba cada chip y cargaba la tabla de vectores al inicio de cada chip en los vectores de RAM. Luego saltaría al programa principal.

Esto funcionó muy bien. Ahora, cuando solucionamos un error o agregamos una función, simplemente podemos recomilar uno o dos chips y enviar solo esos chips a los ingenieros de servicio de campo.

Hicimos que los chips se pudieran implementar de forma independiente. Habíamos inventado el envío polimórfico. Habíamos inventado objetos.

Esta era una arquitectura de complemento, literalmente. Conectamos esos chips. Finalmente lo diseñamos para que se pudiera instalar una característica en nuestros productos conectando el chip con esa característica en uno de los zócalos de chip abiertos. El control del menú aparecería automáticamente y la vinculación a la aplicación principal se produciría automáticamente.

Por supuesto, en ese momento no sabíamos acerca de los principios orientados a objetos y no sabíamos nada sobre cómo separar la interfaz de usuario de las reglas comerciales. Pero los rudimentos estaban ahí y eran muy poderosos.

Un beneficio secundario inesperado de este enfoque fue que pudimos parchear el firmware a través de una conexión de acceso telefónico. Si encontramos un error en el firmware, podríamos marcar nuestros dispositivos y usar el programa de monitor integrado para alterar el vector de RAM para que la subrutina defectuosa apunte a un poco de RAM vacía. Luego ingresaríamos la subrutina reparada en esa área de RAM, escribiéndola en código de máquina, en hexadecimal.

Esto fue una gran ayuda para nuestra operación de servicio de campo y para nuestros clientes. Si tenían un problema, no necesitaban que enviáramos nuevos chips ni programaramos una llamada urgente de servicio de campo. Se podría parchear el sistema y se podría instalar un nuevo chip en la próxima visita de mantenimiento programada regularmente.

LA COMPUTADORA DEL ÁREA DE SERVICIO

La computadora del área de servicio (SAC) 4-TEL se basó en una minicomputadora M365. Este sistema se comunicaba con todos los COLT en el campo, ya sea a través de módems dedicados o de acceso telefónico. Ordenaría a esos COLT que midieran

líneas telefónicas, recibiría los resultados sin procesar y luego realizaría un análisis complejo de esos resultados para identificar y localizar cualquier falla.

DETERMINACIÓN DE ENVÍO

Una de las bases económicas de este sistema se basó en la correcta asignación de los profesionales de la reparación. Los oficios de reparación se dividieron, según las reglas sindicales, en tres categorías: oficina central, cable y acometida. Los artesanos de CO solucionaron problemas dentro de la oficina central. Los artesanos del cable solucionaron problemas en la planta de cables que conectaba el CO con el cliente. Los artesanos de la caída solucionaron problemas dentro de las instalaciones del cliente y en las líneas que conectan el cable externo a esas instalaciones (la "caída").

Cuando un cliente se quejaba de un problema, nuestro sistema podía diagnosticarlo y determinar qué tipo de técnico enviar. Esto ahorró mucho dinero a las compañías telefónicas porque los despachos incorrectos significaban retrasos para el cliente y viajes desperdiciados para los artesanos.

El código que tomó esta determinación de envío fue diseñado y escrito por alguien que era muy brillante, pero un terrible comunicador. El proceso de escribir el código se ha descrito como "Tres semanas de mirar al techo y dos días de código saliendo de cada orificio de su cuerpo, después de lo cual renunció".

Nadie entendió este código. Cada vez que intentábamos agregar una característica o corregir un defecto, lo rompíamos de alguna manera. Y dado que uno de los principales beneficios económicos de nuestro sistema dependía de este código, cada nuevo defecto resultaba profundamente embarazoso para la empresa.

Al final, nuestra gerencia simplemente nos dijo que bloqueáramos ese código y nunca lo modificáramos. Ese código se volvió oficialmente rígido.

Esta experiencia me dejó claro el valor de un código bueno y limpio.

ARQUITECTURA

El sistema fue escrito en 1976 en ensamblador M365. Era un programa único y monolítico de aproximadamente 60.000 líneas. El sistema operativo era un comutador de tareas no preventivo y de cosecha propia basado en encuestas. Lo llamamos MPS por sistema multiprocesamiento. La computadora M365 no tenía una pila incorporada, por lo que las tareas

Las variables específicas se mantuvieron en un área especial de la memoria y se intercambiaron en cada cambio de contexto. Las variables compartidas se gestionaron con cerraduras y semáforos. Los problemas de reentrada y las condiciones de carrera fueron problemas constantes.

No hubo aislamiento de la lógica de control del dispositivo, o lógica de la interfaz de usuario, de las reglas comerciales del sistema. Por ejemplo, el código de control del módem podría encontrarse borroso en la mayor parte de las reglas comerciales y el código de la interfaz de usuario. No hubo ningún intento de reunirlo en un módulo o abstraer la interfaz. Los módems estaban controlados, a nivel de bits, mediante un código que estaba disperso por todo el sistema.

Lo mismo ocurrió con la interfaz de usuario del terminal. Los mensajes y el código de control de formato no fueron aislados. Se extendieron a lo largo y ancho de la base de código de 60.000 líneas.

Los módulos de módem que estábamos usando fueron diseñados para montarse en placas de PC. Compramos esas unidades a un tercero y las integraron con otros circuitos en una placa que encaja en nuestro backplane personalizado. Estas unidades eran caras. Entonces, después de unos años, decidimos diseñar nuestros propios módems. Nosotros, en el grupo de software, le rogamos al diseñador de hardware que utilizara los mismos formatos de bits para controlar el nuevo módem. Le explicamos que el código de control del módem estaba manchado por todas partes y que nuestro sistema tendría que lidiar con ambos tipos de módems en el futuro. Entonces, suplicamos y engatusamos: "Por favor, haga que el nuevo módem se vea igual que el antiguo desde el punto de vista del control del software".

Pero cuando obtuvimos el nuevo módem, la estructura de control era completamente diferente. No fue sólo un poco diferente. Era total y completamente diferente.

Gracias, ingeniero de hardware.

¿Qué íbamos a hacer? No estábamos simplemente reemplazando todos los módems antiguos por módems nuevos. En cambio, estábamos mezclando módems nuevos y antiguos en nuestros sistemas. El software debía poder manejar ambos tipos de módems al mismo tiempo.

¿Estábamos condenados a rodear cada lugar del código que manipulaba los módems con banderas y casos especiales? ¡Había cientos de lugares así!

Al final optamos por una solución aún peor.

Una subrutina en particular escribía datos en el bus de comunicación serie que se utilizaba para controlar todos nuestros dispositivos, incluidos nuestros módems. Modificamos esa subrutina para reconocer los patrones de bits que eran específicos del módem anterior y

traducirlos a los patrones de bits que necesita el nuevo módem.

Esto no fue sencillo. Los comandos al módem consistían en secuencias de escrituras en diferentes direcciones IO en el bus serie. Nuestro truco tuvo que interpretar estos comandos, en secuencia, y traducirlos a una secuencia diferente usando diferentes direcciones IO, tiempos y posiciones de bits.

Lo hicimos funcionar, pero fue el peor truco imaginable. Fue gracias a este fiasco que aprendí el valor de aislar el hardware de las reglas comerciales y de abstraer las interfaces.

EL GRAN REDISEÑO EN EL CIELO

Cuando llegó la década de 1980, la idea de producir su propia minicomputadora y su propia arquitectura de computadora estaba comenzando a pasar de moda. Había muchas microcomputadoras en el mercado, y hacerlas funcionar era más barato y más estándar que seguir dependiendo de arquitecturas informáticas patentadas de finales de la década de 1960. Eso, sumado a la horrible arquitectura del software SAC, indujo a nuestra dirección técnica a iniciar una re-arquitectura completa del sistema SAC.

El nuevo sistema debía escribirse en C utilizando un sistema operativo UNIX en disco, ejecutándose en una microcomputadora Intel 8086. Nuestros chicos de hardware comenzaron a trabajar en el nuevo hardware de la computadora y un grupo selecto de desarrolladores de software, "The Tiger Team", recibió el encargo de reescribirlo.

No los aburriré con los detalles del fiasco inicial. Baste decir que el primer Tiger Team fracasó por completo después de gastar dos o tres años-hombre en un proyecto de software que nunca entregó nada.

Uno o dos años después, probablemente en 1982, se reanudó el proceso. El objetivo era el rediseño total y completo del SAC en C y UNIX en nuestro propio hardware 80286, de nuevo diseño e impresionantemente potente. A esa computadora la llamamos "Pensamiento profundo".

Fueron necesarios años, luego más años y luego incluso más años. No sé cuándo se implementó finalmente el primer SAC basado en UNIX; Creo que para entonces ya había dejado la empresa (1988). De hecho, no estoy del todo seguro de que alguna vez se haya implementado.

¿Por qué el retraso? En resumen, es muy difícil para un equipo de rediseño ponerse al día con un

Gran plantilla de programadores que mantienen activamente el antiguo sistema. Éste es sólo un ejemplo de las dificultades que encontraron.

EUROPA

Aproximadamente al mismo tiempo que se rediseñaba el SAC en C, la empresa comenzó a expandir sus ventas en Europa. No podían esperar a que estuviera terminado el software rediseñado y, por supuesto, implementaron los antiguos sistemas M365 en Europa.

El problema era que los sistemas telefónicos en Europa eran muy diferentes a los sistemas telefónicos en Estados Unidos. La organización del oficio y de las burocracias también era diferente. Entonces, uno de nuestros mejores programadores fue enviado al Reino Unido para liderar un equipo de desarrolladores del Reino Unido para modificar el software SAC para abordar todos estos problemas europeos.

Por supuesto, no se hizo ningún intento serio de integrar estos cambios en el software con sede en Estados Unidos. Esto fue mucho antes de que las redes hicieran posible transmitir grandes bases de códigos a través del océano. Estos desarrolladores del Reino Unido simplemente bifurcaron el código basado en Estados Unidos y lo modificaron según fuera necesario.

Esto, por supuesto, causó dificultades. Se encontraron errores a ambos lados del Atlántico que necesitaban reparación en el otro lado. Pero los módulos habían cambiado significativamente, por lo que era muy difícil determinar si la solución realizada en Estados Unidos funcionaría en el Reino Unido.

Después de algunos años de acidez y de la instalación de una línea de alto rendimiento que conecta las oficinas de EE. UU. y el Reino Unido, se hizo un intento serio de integrar nuevamente estas dos bifurcaciones, haciendo que las diferencias sean una cuestión de configuración. Este esfuerzo fracasó la primera, segunda y tercera vez que se intentó. Las dos bases de código, aunque notablemente similares, todavía eran demasiado diferentes para reintegrarse, especialmente en el entorno de mercado rápidamente cambiante que existía en ese momento.

Mientras tanto, el “Equipo Tigre”, que intentaba reescribir todo en C y UNIX, se dio cuenta de que también tenía que lidiar con esta dicotomía entre Europa y Estados Unidos. Y, por supuesto, eso no hizo nada para acelerar su progreso.

CONCLUSIÓN DEL SAC

Hay muchas otras historias que podría contarte sobre este sistema, pero es demasiado deprimente para continuar. Baste decir que muchas de las duras lecciones de mi vida software las aprendí mientras estaba inmerso en el horrible código ensamblador del SAC.

IDIOMA C

El hardware informático 8085 que utilizamos en el proyecto 4-Tel Micro nos brindó una plataforma informática de costo relativamente bajo para muchos proyectos diferentes que podrían integrarse en entornos industriales. Podíamos cargarlo con 32K de RAM y otros 32K de ROM, y teníamos un esquema extremadamente flexible y potente para controlar periféricos. Lo que no teníamos era un lenguaje flexible y conveniente para programar la máquina. Simplemente no era divertido escribir código en el ensamblador 8085.

Además de eso, el ensamblador que estábamos usando fue escrito por nuestros propios programadores. Se ejecutó en nuestras computadoras M365, utilizando el sistema operativo de cinta de cartucho descrito en la sección "Recorte láser".

Quiso el destino que nuestro ingeniero jefe de hardware convenciera a nuestro director ejecutivo de que necesitábamos una computadora real. En realidad no sabía qué haría con ello, pero tenía mucha influencia política. Entonces compramos un PDP-11/60.

Yo, un modesto desarrollador de software en ese momento, estaba extasiado. Sabía exactamente lo que quería hacer con esa computadora. Estaba decidido a que esta iba a ser mi máquina.

Cuando llegaron los manuales, muchos meses antes de la entrega de la máquina, me los llevé a casa y los devoré. Cuando me entregaron la computadora, sabía cómo operar tanto el hardware como el software a un nivel íntimo, al menos tan íntimo como el estudio en casa puede lograr.

Ayudé a redactar la orden de compra. En particular, especifiqué el almacenamiento en disco que tendría la nueva computadora. Decidí que deberíamos comprar dos unidades de disco que pudieran admitir paquetes de discos extraíbles de 25 megabytes cada uno. 9 –

¡Cincuenta megas! ¡El número parecía infinito! Recuerdo caminar por los pasillos de la oficina, a altas horas de la noche, riendo como la Malvada Bruja del Oeste:

“¡Cincuenta megas! ¡Jajajajajajajajaja!”

Le pedí al administrador de instalaciones que construyera una pequeña habitación que albergaría seis terminales VT100. Lo decoré con fotografías del espacio. Nuestros desarrolladores de software utilizarían esta sala para escribir y compilar código.

Cuando llegó la máquina, pasé varios días configurándola, cableando todos los terminales y haciendo que todo funcionara. Fue una alegría, un trabajo de amor.

Compramos ensambladores estándar para el 8085 en Boston Systems Office y tradujimos el código de 4-Tel Micro a esa sintaxis. Construimos un sistema de compilación cruzada que nos permitió descargar binarios compilados desde el PDP-11 a nuestros entornos de desarrollo 8085 y grabadoras de ROM. Y Bob es tu tío, todo funcionó como un campeón.

C

Pero eso nos dejó con el problema de seguir usando el ensamblador 8085. Esa no era una situación con la que estuviera contento. Había oído que existía este “nuevo” lenguaje que se usaba mucho en Bell Labs. Lo llamaron “C”. Entonces compré una copia de The C Programming Language de Kernighan y Ritchie. Al igual que los manuales del PDP-11 unos meses antes, inhalé este libro.

Me sorprendió la simple elegancia de este lenguaje. No sacrificó nada del poder del lenguaje ensamblador y proporcionó acceso a ese poder con una sintaxis mucho más conveniente. Me vendieron.

Compré un compilador de C en Whitesmiths y lo ejecuté en el PDP-11. El resultado del compilador era una sintaxis de ensamblador compatible con el compilador Boston Systems Office 8085. ¡Así que teníamos un camino para pasar de C al hardware 8085! Estábamos en el negocio.

Ahora el único problema era convencer a un grupo de programadores en lenguaje ensamblador incorporado de que deberían usar C. Pero esa es una historia de pesadilla para otro. tiempo ...

JEFE

Nuestra plataforma 8085 no tenía sistema operativo. Mi experiencia con el sistema MPS del M365 y los primitivos mecanismos de interrupción del IBM System 7 me convencieron de que necesitábamos un conmutador de tareas simple para el 8085. Así que concebí BOSS: Sistema operativo básico y programador. [10](#) —

La gran mayoría de BOSS fue escrita en C. Brindó la capacidad de crear tareas simultáneas. Esas tareas no eran preventivas: el cambio de tareas no se producía en función de interrupciones. En cambio, y al igual que con el sistema MPS del M365, las tareas se cambiaron basándose en un simple mecanismo de sondeo. La encuesta se realizaba cada vez que una tarea se bloqueaba para un evento.

La llamada de BOSS para bloquear una tarea tenía este aspecto:

```
bloquear(eventCheckFunction);
```

Esta llamada suspendió la tarea actual, colocó eventCheckFunction en la lista de sondeo y la asoció con la tarea recién bloqueada. Luego esperó en el ciclo de sondeo, llamando a cada una de las funciones en la lista de sondeo hasta que una de ellas resultó verdadera. Luego se permitió que se ejecutara la tarea asociada con esa función.

En otras palabras, como dije antes, era un cambio de tareas simple y no preventivo.

Este software se convirtió en la base de una gran cantidad de proyectos durante los años siguientes. Pero uno de los primeros fue el pCCU.

pCCU

Los finales de los años 1970 y principios de los 1980 fueron una época tumultuosa para las compañías telefónicas. Una de las fuentes de ese tumulto fue la revolución digital.

Durante el siglo anterior, la conexión entre la central de conmutación y el teléfono del cliente era un par de cables de cobre. Estos cables estaban agrupados en cables que se extendían formando una enorme red por todo el campo. A veces los transportaban en postes y otras los enterraban bajo tierra.

El cobre es un metal precioso y la compañía telefónica tenía toneladas (literalmente toneladas) de él cubriendo todo el país. La inversión de capital fue enorme. Gran parte de ese capital podría recuperarse trasladando la conversación telefónica a lo digital.

conexiones. Un solo par de cables de cobre podría transmitir cientos de conversaciones en formato digital.

En respuesta, las compañías telefónicas se embarcaron en el proceso de reemplazar sus antiguos equipos de conmutación central analógicos por modernos conmutadores digitales.

Nuestro producto 4-Tel probó cables de cobre, no conexiones digitales. Todavía había muchos cables de cobre en el entorno digital, pero eran mucho más cortos que antes y estaban localizados cerca de los teléfonos de los clientes. La señal se transportaría digitalmente desde la oficina central a un punto de distribución local, donde se convertiría nuevamente en una señal analógica y se distribuiría al cliente a través de cables de cobre estándar. Esto significaba que nuestro dispositivo de medición debía estar ubicado donde comenzaban los cables de cobre, pero nuestro dispositivo de marcación debía permanecer en la oficina central. El problema era que todos nuestros COLT incorporaban tanto la marcación como la medición en el mismo dispositivo. (¡Podríamos habernos ahorrado una fortuna si hubiéramos reconocido ese límite arquitectónico obvio unos años antes!)

Así concebimos una nueva arquitectura de producto: la CCU/CMU (la unidad de control COLT y la unidad de medición COLT). La CCU estaría ubicada en la oficina central de conmutación y se encargaría de marcar las líneas telefónicas que se probarían. La CMU estaría ubicada en los puntos de distribución locales y mediría los cables de cobre que conducían al teléfono del cliente.

El problema era que por cada CCU había muchas CMU. La información sobre qué CMU debería usarse para cada número de teléfono la guardaba el propio conmutador digital. Por lo tanto, la CCU tuvo que interrogar al interruptor digital para determinar con qué CMU comunicarse y controlar.

Prometimos a las compañías telefónicas que tendríamos esta nueva arquitectura funcionando a tiempo para su transición. Sabíamos que faltaban meses, si no años, por lo que no nos sentimos apurados. También sabíamos que se necesitarían varios años-hombre para desarrollar este nuevo hardware y software CCU/CMU.

LA TRAMPA DEL HORARIO

Con el paso del tiempo, descubrimos que siempre había asuntos urgentes que requerían que pospusiéramos el desarrollo de la arquitectura CCU/CMU. Nos sentimos seguros con esta decisión porque las compañías telefónicas retrasaban constantemente la entrega.

Implementación de conmutadores digitales. Al mirar sus cronogramas, nos sentimos seguros de que teníamos mucho tiempo, por lo que retrasamos constantemente nuestro desarrollo.

Entonces llegó el día en que mi jefe me llamó a su oficina y me dijo: "Uno de nuestros clientes implementará un conmutador digital el próximo mes. Para entonces debemos tener una CCU/CMU en funcionamiento".

¡Estaba horrorizado! ¿Cómo podríamos lograr años-hombre de desarrollo en un mes?

Pero mi jefe tenía un plan...

De hecho, no necesitábamos una arquitectura CCU/CMU completa. La compañía telefónica que estaba implementando el conmutador digital era pequeña. Tenían sólo una oficina central y sólo dos puntos de distribución local. Más importante aún, los puntos de distribución "locales" no eran particularmente locales. De hecho, tenían interruptores analógicos antiguos que conmutaban a varios cientos de clientes. Mejor aún, esos interruptores eran de un tipo que podía ser activado por un COLT normal. Mejor aún, el número de teléfono del cliente contenía toda la información necesaria para decidir qué punto de distribución local utilizar. Si el número de teléfono tenía un 5, 6 o 7 en una determinada posición, pasaba al punto de distribución 1; en caso contrario, pasaba al punto de distribución 2.

Entonces, como me explicó mi jefe, en realidad no necesitábamos una CCU/CMU. Lo que necesitábamos era una computadora simple en la oficina central conectada por líneas de módem a dos COLT estándar en los puntos de distribución. El SAC se comunicaría con nuestra computadora en la oficina central, y esa computadora decodificaría el número de teléfono y luego transmitiría los comandos de marcación y medición al COLT en el punto de distribución apropiado.

Así nació la pCCU.

Este fue el primer producto escrito en C y usando BOSS que se implementó para un cliente. Me tomó alrededor de una semana desarrollarlo. Esta historia no tiene un significado arquitectónico profundo, pero constituye un bonito prefacio para el próximo proyecto.

DLU/DRU

A principios de la década de 1980, uno de nuestros clientes era una compañía telefónica en Texas.

Tenían grandes áreas geográficas que cubrir. De hecho, las áreas eran tan grandes que una sola área de servicio requería varias oficinas diferentes desde las cuales enviar a los artesanos. Esas oficinas tenían técnicos de prueba que necesitaban terminales en nuestro SAC.

Podrías pensar que se trata de un problema sencillo de resolver, pero recuerda que esta historia tiene lugar a principios de los años ochenta. Los terminales remotos no eran muy comunes. Para colmo, el hardware del SAC presumía que todas las terminales eran locales. En realidad, nuestros terminales se encontraban en un bus serie propietario de alta velocidad.

Teníamos capacidad de terminal remota, pero estaba basada en módems, y a principios de los años 1980 los módems estaban generalmente limitados a 300 bits por segundo. Nuestros clientes no estaban contentos con esa baja velocidad.

Había módems de alta velocidad disponibles, pero eran muy caros y debían funcionar con conexiones permanentes "condicionadas". La calidad del acceso telefónico definitivamente no era lo suficientemente buena.

Nuestros clientes exigieron una solución. Nuestra respuesta fue DLU/DRU.

DLU/DRU significaba "Unidad local de visualización" y "Unidad remota de visualización". La DLU era una placa de computadora que se conectaba al chasis de la computadora SAC y pretendía ser una placa administradora de terminales. Sin embargo, en lugar de controlar el bus serie para terminales locales, tomó el flujo de caracteres y lo multiplexó a través de un único enlace de módem condicionado de 9600 bps.

La DRU era una caja colocada en la ubicación remota del cliente. Se conectaba al otro extremo del enlace de 9600 bps y tenía el hardware para controlar los terminales en nuestro bus serie propietario. Demultiplexó los caracteres recibidos del enlace de 9600 bps y los envió a los terminales locales apropiados.

Extraño, ¿no? Tuvimos que diseñar una solución que hoy en día es tan omnipresente que ni siquiera pensamos en ella. Pero en aquel entonces ...

Incluso tuvimos que inventar nuestro propio protocolo de comunicaciones porque, en aquellos días, los protocolos de comunicaciones estándar no eran shareware de código abierto. De hecho, esto fue mucho antes de que tuviéramos algún tipo de conexión a Internet.

ARQUITECTURA

La arquitectura de este sistema era muy simple, pero hay algunas peculiaridades interesantes que quiero resaltar. Primero, ambas unidades usaron nuestra tecnología 8085, y ambas fueron escritas en C y usaron BOSS. Pero ahí terminó la similitud.

Éramos dos en el proyecto. Yo era el líder del proyecto y Mike Carew era mi colaborador más cercano. Asumí el diseño y codificación del DLU; Mike hizo la DRU.

La arquitectura de la DLU se basó en un modelo de flujo de datos. Cada tarea hizo un trabajo pequeño y enfocado, y luego pasó su resultado a la siguiente tarea en línea, usando una cola. Piense en un modelo de tuberías y filtros en UNIX. La arquitectura era compleja. Una tarea podría alimentar una cola que muchas otras atenderían. Otras tareas alimentarían una cola que solo una tarea atendería.

Piense en una línea de montaje. Cada puesto en la línea de montaje tiene un trabajo único, simple y altamente enfocado que realizar. Luego el producto pasa a la siguiente posición en la línea. A veces la línea de montaje se divide en muchas líneas. A veces esas líneas vuelven a fusionarse en una sola línea. Ese fue el DLU.

La DRU de Mike utilizó un esquema notablemente diferente. Creó una tarea por terminal y simplemente hizo todo el trabajo para esa terminal en esa tarea. Sin colas. Sin flujo de datos. Sólo muchas tareas grandes idénticas, cada una administrando su propia terminal.

Esto es lo opuesto a una línea de montaje. En este caso, la analogía son muchos constructores expertos, cada uno de los cuales construye un producto completo.

En ese momento pensé que mi arquitectura era superior. Mike, por supuesto, pensó que el suyo era mejor. Tuvimos muchas discusiones entretenidas sobre esto. Al final, por supuesto, ambos funcionaron bastante bien. Y me quedé con la comprensión de que las arquitecturas de software pueden ser tremadamente diferentes, pero igualmente efectivas.

VRS

A medida que avanzaba la década de 1980, aparecieron tecnologías cada vez más nuevas. Una de esas tecnologías fue el control informático de la voz.

Una de las características del sistema 4-Tel era la capacidad del artesano de localizar un fallo en un cable. El procedimiento fue el siguiente:

- El probador, en la oficina central, usaría nuestro sistema para determinar la distancia aproximada, en pies, hasta la falla. Esto tendría una precisión de aproximadamente el 20%. El probador enviaría a un técnico reparador de cables a un punto de acceso apropiado cerca de esa posición.
- El técnico reparador de cables, al llegar, llamaría al probador y le pediría que comenzara el proceso de localización de la avería. El probador invocaría la función de localización de fallos del sistema 4-Tel. El sistema comenzaría a medir las características electrónicas de esa línea defectuosa, e imprimiría mensajes en la pantalla solicitando que se realizaran determinadas operaciones, como abrir el cable o cortocircuitarlo. • El evaluador le diría al técnico qué operaciones deseaba el sistema, y el técnico le indicaría cuando se completara la operación. Luego, el evaluador le diría al sistema que la operación se completó y el sistema continuaría con la prueba.

- Después de dos o tres de estas interacciones, el sistema calcularía una nueva distancia hasta la falla. Luego, el artesano del cable se dirigía hasta ese lugar y comenzaba el proceso nuevamente.

Imagínese lo mejor que sería si los artesanos del cable, subidos al poste o parados en un pedestal, pudieran operar el sistema ellos mismos. Y eso es exactamente lo que nos permitieron hacer las nuevas tecnologías de voz. Los artesanos del cable podían llamar directamente a nuestro sistema, dirigir el sistema mediante tonos y escuchar los resultados que se les leían con una voz agradable.

EL NOMBRE

La empresa realizó un pequeño concurso para seleccionar un nombre para el nuevo sistema. Uno de los nombres sugeridos más creativos fue SAM CARP. Esto significaba “Otra manifestación más de la avaricia capitalista que reprime al proletariado”.

No hace falta decir que eso no fue seleccionado.

Otro fue el sistema de prueba interactivo Teradyne. Ese tampoco fue seleccionado.

Otro más fue la Red de acceso a pruebas del área de servicio. Éste tampoco fue seleccionado.

El ganador al final fue VRS: Voice Response System.

ARQUITECTURA

No trabajé en este sistema, pero me enteré de lo que pasó. La historia que les voy a contar es de segunda mano, pero en su mayor parte creo que es correcta.

Eran los días embriagadores de las microcomputadoras, los sistemas operativos UNIX, las bases de datos C y SQL. Estábamos decididos a utilizarlos todos.

De los muchos proveedores de bases de datos que existen, finalmente elegimos UNIFY. UNIFY era un sistema de base de datos que funcionaba con UNIX, lo cual era perfecto para nosotros.

UNIFY también admitió una nueva tecnología llamada SQL integrado. Esta tecnología nos permitió incrustar comandos SQL, como cadenas, directamente en nuestro código C. Y así lo hicimos, en todas partes.

Quiero decir, fue genial poder poner tu SQL directamente en tu código, en cualquier lugar que quisieras. ¿Y dónde queríamos llegar? ¡En todos lados! Y entonces había SQL manchado por todo el cuerpo de ese código.

Por supuesto, en aquellos días SQL no era un estándar sólido. Había muchas peculiaridades especiales de cada proveedor. Por lo tanto, las llamadas especiales de SQL y API UNIFY también estaban manchadas en todo el código.

¡Esto funcionó muy bien! El sistema fue un éxito. Los artesanos lo utilizaban y a las compañías telefónicas les encantaba. La vida era todo sonrisas.

Luego se canceló el producto UNIFY que estábamos usando.

Oh. Oh.

Entonces decidimos cambiarnos a SyBase. ¿O fue Ingreso? No lo recuerdo. Basta decir que tuvimos que buscar en todo ese código C, encontrar todos los SQL incorporados y las llamadas API especiales, y reemplazarlos con los gestos correspondientes para el nuevo proveedor.

Después de aproximadamente tres meses de esfuerzo, nos dimos por vencidos. No pudimos hacerlo funcionar. Estábamos tan acoplados a UNIFY que no había ninguna esperanza seria de reestructurar el código a ningún costo práctico.

Entonces, contratamos a un tercero para que mantuviera UNIFY por nosotros, según un contrato de mantenimiento. Y, por supuesto, las tasas de mantenimiento aumentaron año tras año tras año.

CONCLUSIÓN DE VRS

Esta es una de las formas en que aprendí que las bases de datos son detalles que deben aislarse del propósito comercial general del sistema. Esta es también una de las razones por las que no me gusta acoplarne demasiado a sistemas de software de terceros.

LA RECEPCIONISTA ELECTRÓNICA

En 1983, nuestra empresa se encontraba en la confluencia de sistemas informáticos, sistemas de telecomunicaciones y sistemas de voz. Nuestro director ejecutivo pensó que ésta podría ser una posición fértil desde la cual desarrollar nuevos productos. Para lograr este objetivo, encargó a un equipo de tres (entre los que me incluía a mí) concebir, diseñar e implementar un nuevo producto para la empresa.

No nos tomó mucho tiempo crear The Electronic Receptionist (ER).

La idea era sencilla. Cuando llamabas a una empresa, Emergencias te respondía y te preguntaba con quién querías hablar. Usaría tonos al tacto para deletrear el nombre de esa persona y luego ER lo conectaría. Los usuarios de ER podrían marcar y, mediante simples comandos de tonos, indicar a qué número de teléfono se puede localizar a la persona deseada, en cualquier parte del mundo. De hecho, el sistema podría enumerar varios números alternativos.

Cuando llamaba a emergencias y marcaba RMART (mi código), emergencias llamaba al primer número de mi lista. Si no respondía ni me identificaba, llamaba al siguiente número y al siguiente. Si todavía no me contactaban, Emergencias grabaría un mensaje de la persona que llama.

Luego, Emergencias, periódicamente, intentaba encontrarme para entregar ese mensaje y cualquier otro mensaje que me dejara cualquier otra persona.

Este fue el primer sistema de correo de voz que existió y [11](#) tenía la patente.

Construimos todo el hardware para este sistema: la placa de la computadora, la memoria

tablero, los tableros de voz/telecomunicaciones, todo. La placa principal de la computadora era Deep Thought, el procesador Intel 80286 que mencioné anteriormente.

Cada uno de los paneles de voz admitía una línea telefónica. Consistían en una interfaz telefónica, un codificador/decodificador de voz, algo de memoria y una microcomputadora Intel 80186.

El software para la placa principal de la computadora estaba escrito en C. El sistema operativo era MP/M-86, uno de los primeros sistemas operativos de disco multiprocesamiento controlados por línea de comandos. MP/M era el UNIX del pobre.

El software de las placas de voz estaba escrito en ensamblador y no tenía sistema operativo. La comunicación entre Deep Thought y los tableros de voz se produjo a través de la memoria compartida.

La arquitectura de este sistema hoy se llamaría orientada a servicios. Cada línea telefónica fue monitoreada por un proceso de escucha que se ejecuta bajo MP/M. Cuando entraba una llamada, se iniciaba un proceso de controlador inicial y se le pasaba la llamada.

A medida que la llamada avanzaba de un estado a otro, se iniciaría el proceso de controlador apropiado y tomaría el control.

Los mensajes se pasaban entre estos servicios a través de archivos de disco. El servicio actualmente en ejecución determinaría cuál debería ser el próximo servicio; escribiría la información de estado necesaria en un archivo de disco; emitiría la línea de comando para iniciar ese servicio; y luego saldría.

Esta fue la primera vez que construí un sistema como este. De hecho, esta era la primera vez que era el arquitecto principal de un producto completo. Todo lo que tenía que ver con software era mío y funcionó como un campeón.

No diría que la arquitectura de este sistema fuera “limpia” en el sentido de este libro; no era una arquitectura de “complemento”. Sin embargo, definitivamente mostró signos de verdaderos límites. Los servicios se podían desplegar de forma independiente y vivían dentro de su propio ámbito de responsabilidad. Había procesos de alto nivel y procesos de bajo nivel, y muchas de las dependencias iban en la dirección correcta.

FALLECIMIENTO DE Urgencias

Lamentablemente, la comercialización de este producto no fue muy bien. Teradyne era una empresa que vendía equipos de prueba. No sabíamos cómo irrumpir en el

mercado de equipos de oficina.

Después de repetidos intentos durante dos años, nuestro director ejecutivo se dio por vencido y, desafortunadamente, abandonó la solicitud de patente. La patente fue recogida por la empresa que la presentó tres meses después de que la presentáramos; por lo tanto, entregamos todo el mercado de correo de voz y desvío electrónico de llamadas.

¡Ay!

Por otro lado, no puedes culparme por esas molestas máquinas que ahora plagan nuestra existencia.

SISTEMA DE DESPACHO DE EMBARCACIONES

ER había fracasado como producto, pero todavía teníamos todo este hardware y software que podíamos utilizar para mejorar nuestras líneas de productos existentes. Además, nuestro éxito de marketing con VRS nos convenció de que deberíamos ofrecer un sistema de respuesta de voz para interactuar con los técnicos telefónicos que no dependiera de nuestros sistemas de prueba.

Así nació CDS, el Craft Dispatch System. CDS era esencialmente ER, pero se centraba específicamente en el ámbito muy limitado de gestionar el despliegue de reparadores de teléfonos en el campo.

Cuando se descubría un problema en una línea telefónica, se creaba un ticket de problema en el centro de servicio. Los tickets de problemas se mantuvieron en un sistema automatizado. Cuando un reparador en el campo terminaba un trabajo, llamaba al centro de servicio para la siguiente tarea. El operador del centro de servicio sacaría el siguiente ticket de problema y se lo leería al técnico.

Nos propusimos automatizar ese proceso. Nuestro objetivo era que el técnico de campo llamara a CDS y preguntara por la siguiente tarea. CDS consultaría el sistema de tickets de problemas y leería los resultados. CDS realizaría un seguimiento de qué técnico fue asignado a cada ticket de problema e informaría al sistema de tickets de problema sobre el estado de la reparación.

Había bastantes características interesantes de este sistema que tenían que ver con la interacción con el sistema de notificación de problemas, el sistema de gestión de la planta y cualquier sistema de prueba automatizado.

La experiencia con la arquitectura orientada a servicios de ER me hizo querer probar la misma idea de manera más agresiva. La máquina de estado para un ticket de problema era mucho más complicada que la máquina de estado para manejar una llamada con ER. Me propuse crear lo que ahora se llamaría una arquitectura de microservicios.

Cada cambio de estado de cualquier llamada, por insignificante que fuera, hacía que el sistema iniciara un nuevo servicio. De hecho, la máquina de estados se externalizó en un archivo de texto que leía el sistema. Cada evento que ingresaba al sistema desde una línea telefónica se convertía en una transición en esa máquina de estados finitos. El proceso existente iniciaría un nuevo proceso dictado por la máquina de estados para manejar ese evento; entonces el proceso existente saldría o esperaría en una cola.

Esta máquina de estados externalizada nos permitió cambiar el flujo de la aplicación sin cambiar ningún código (el principio abierto-cerrado). Podríamos agregar fácilmente un nuevo servicio, independientemente de cualquiera de los demás, e incorporarlo al flujo modificando el archivo de texto que contenía la máquina de estados. Incluso podríamos hacer esto mientras el sistema estaba funcionando. En otras palabras, teníamos intercambio en caliente y un BPEL (Business Process Execution Language) eficaz.

El antiguo enfoque de ER de utilizar archivos de disco para comunicarse entre servicios era demasiado lento para este cambio mucho más rápido de servicios, por lo que inventamos un mecanismo de memoria compartida al que llamamos 3DBB. ser ¹² Los datos permitidos por 3DBB accedido por nombre; los nombres que utilizamos fueron nombres asignados a cada instancia de máquina de estado.

El 3DBB era excelente para almacenar cadenas y constantes, pero no podía usarse para contener estructuras de datos complejas. La razón de esto es técnica pero fácil de entender. Cada proceso en MP/M vivía en su propia partición de memoria. Los punteros a datos en una partición de memoria no tenían significado en otra partición de memoria. Como consecuencia, los datos del 3DBB no podían contener punteros. Las cadenas estaban bien, pero los árboles, las listas enlazadas o cualquier estructura de datos con punteros no funcionarían.

Los tickets de problemas del sistema de tickets de problemas procedían de muchas fuentes diferentes. Algunos eran automatizados y otros eran manuales. Las entradas manuales fueron creadas por operadores que hablaban con los clientes sobre sus problemas. A medida que los clientes describían sus problemas, los operadores escribían sus quejas y observaciones en un flujo de texto estructurado. Se parecía a esto:

[Haga clic aquí para ver la imagen del código](#)

/pno 8475551212 /ruido /llamadas-cortadas

Entiendes la idea. El personaje / inició un nuevo tema. Después de la barra diagonal había un código, y después del código estaban los parámetros. Había miles de códigos y un ticket de problema individual podía tener docenas de ellos en la descripción.

Peor aún, como se ingresaban manualmente, a menudo estaban mal escritos o formateados incorrectamente. Estaban destinados a que los humanos los interpretaran, no a que las máquinas los procesaran.

Nuestro problema era decodificar estas cadenas de forma semilibre, interpretar y corregir cualquier error, y luego convertirlas en salida de voz para poder leérsela al técnico, subido a un poste, escuchando con un teléfono. Esto requería, entre otras cosas, una técnica de análisis y representación de datos muy flexible. Esa representación de datos tuvo que pasar a través de 3DBB, que solo podía manejar cadenas.

Y así, en un avión, volando entre visitas de clientes, inventé un esquema al que llamé FLD: Field Labeled Data. Hoy en día lo llamaríamos XML o JSON.

El formato era diferente, pero la idea era la misma. Los FLD eran árboles binarios que asociaban nombres con datos en una jerarquía recursiva. Los FLD se podían consultar mediante una API simple y se podían traducir hacia y desde un formato de cadena conveniente que era ideal para 3DBB.

Entonces, los microservicios se comunican a través de un análogo de memoria compartida de sockets usando un análogo XML, en 1985.

No hay nada nuevo bajo el sol.

COMUNICACIONES CLARAS

En 1988, un grupo de empleados de Teradyne dejaron la empresa para formar una startup llamada Clear Communications. Me uní a ellos unos meses después. Nuestra misión era crear el software para un sistema que monitorearía la calidad de las comunicaciones de las líneas T1, las líneas digitales que transportaban comunicaciones de larga distancia en todo el país. La visión era un monitor enorme con un mapa de Estados Unidos atravesado por líneas T1 que parpadeaban en rojo si se estaban degradando.

Recuerde, las interfaces gráficas de usuario eran completamente nuevas en 1988. Apple

Macintosh tenía sólo cinco años. Windows era una broma en aquel entonces. Pero Sun Microsystems estaba construyendo Sparcstations que tenían GUI X-Windows creíbles. Así que optamos por Sun y, por tanto, por C y UNIX.

Esta fue una startup. Trabajamos de 70 a 80 horas por semana. Tuvimos la visión. Teníamos la motivación. Teníamos la voluntad. Teníamos la energía. Teníamos la experiencia. Teníamos equidad. Soñábamos con ser millonarios. Estábamos llenos de mierda.

El código C brotó de cada orificio de nuestro cuerpo. Lo golpeamos aquí y lo empujamos allí. Construimos enormes castillos en el aire. Teníamos procesos, colas de mensajes y arquitecturas grandiosas y superlativas. Escribimos una pila de comunicaciones ISO completa de siete capas desde cero, hasta la capa de enlace de datos.

Escribimos código GUI. ¡CÓDIGO PEGAJO! ¡DIOS MÍO! Escribimos código GOOOOOEY.

Yo personalmente escribí una función C de 3000 líneas llamada gi(); su nombre significa intérprete gráfico. Fue una obra maestra de goo. No fue el único mensaje que escribí en Clear, pero fue el más infame.

¿Arquitectura? ¿Estás bromeando? Esta fue una startup. No teníamos tiempo para la arquitectura. ¡Solo codifica, maldita sea! ¡Código para vuestras vidas!

Entonces codificamos. Y codificamos. Y codificamos. Pero, después de tres años, lo que no conseguimos fue vender. Oh, tuvimos una instalación o dos. Pero el mercado no estaba particularmente interesado en nuestra gran visión y nuestros financieros de capital de riesgo estaban bastante hartos.

Odiaba mi vida en este momento. Vi todo mi esfuerzo y sueños derrumbarse. Tuve conflictos en el trabajo, conflictos en casa por el trabajo y conflictos conmigo mismo.

Y luego recibí una llamada telefónica que lo cambió todo.

LA PUESTA EN MARCHA

Dos años antes de esa llamada telefónica, sucedieron dos cosas importantes.

Primero, logré configurar una conexión uucp con una empresa cercana que tenía una conexión uucp con otra instalación que estaba conectada a Internet. Estas conexiones eran por discado, por supuesto. Nuestra Sparcstation principal (la que está en mi escritorio) usaba un módem de 1200 bps para llamar a nuestro host uucp dos veces al día. esto nos dio

correo electrónico y Netnews (una de las primeras redes sociales donde la gente discutía temas interesantes).

En segundo lugar, Sun lanzó un compilador de C++. Me había interesado C++ y OO desde 1983, pero era difícil conseguir compiladores. Entonces, cuando se presentó la oportunidad, cambié de idioma de inmediato. Dejé atrás las funciones C de 3000 líneas y comencé a escribir código C++ en Clear. y aprendí ...

Leo libros. Por supuesto, leí El lenguaje de programación C++ y El manual de referencia anotado de C++ (ARM) de Bjarne Stroustrup. Leí el encantador libro de Rebecca Wirfs-Brock sobre diseño impulsado por la responsabilidad: Designing Object Oriented Software. Leí OOA , OOD y OOP de Peter Coad. Leí Smalltalk-80 de Adele Goldberg. Leí Modismos y estilos de programación avanzados en C++ de James O. Coplien. Pero quizás lo más significativo de todo es que leí Diseño orientado a objetos con aplicaciones de Grady Booch.

¡Que nombre! Grady Booch. ¿Cómo podría alguien olvidar un nombre así? Es más, ¡era el científico jefe de una empresa llamada Rational! ¡Cómo quería ser científico jefe! Y entonces leí su libro. Y aprendí, y aprendí, y aprendí ...

Como aprendí, también comencé a debatir en Netnews, como la gente debate ahora en Facebook. Mis debates fueron sobre C++ y OO. Durante dos años, alivié las frustraciones que se estaban acumulando en el trabajo debatiendo con cientos de personas en Usenet sobre las mejores características del lenguaje y los mejores principios de diseño. Después de un tiempo, incluso comencé a tener cierto sentido.

Fue en uno de esos debates donde se sentaron las bases de los principios SOLID.

Y todo ese debate, y tal vez incluso algo de sentido común, me hicieron notar. ...

TÍO BOB

Uno de los ingenieros de Clear era un joven llamado Billy Vogel. Billy le puso apodos a todo el mundo. Me llamó tío Bob. Sospecho que, a pesar de que mi nombre es Bob, estaba haciendo una referencia informal a JR "Bob". Dobbs (ver <https://en.wikipedia.org/wiki/File:Bobdobbs.png>).

Al principio lo toleré. Pero a medida que pasaban los meses, su incesante parloteo de

"Tío Bob, ... Uncle Bob", en el contexto de las presiones y decepciones de la startup, comenzó a debilitarse bastante.

Y entonces, un día, sonó el teléfono.

LA LLAMADA TELEFÓNICA

Fue un reclutador. Obtuvo mi nombre porque conocía C++ y el diseño orientado a objetos. No estoy seguro de cómo, pero sospecho que tuvo algo que ver con mi presencia en Netnews.

Dijo que tuvo una oportunidad en Silicon Valley, en una empresa llamada Rational. [13](#) Buscaban ayuda para construir un CASO [herramienta](#).

La sangre desapareció de mi cara. Sabía lo que era esto. No sé cómo lo supe, pero lo sabía. Esta era la empresa de Grady Booch . ¡Vi ante mí la oportunidad de unir fuerzas con Grady Booch!

ROSA

Me uní a Rational, como programador contratado, en 1990. Estaba trabajando en el producto ROSE. Esta era una herramienta que permitía a los programadores dibujar diagramas de Booch, los diagramas sobre los que Grady había escrito en Análisis y diseño orientado a objetos con aplicaciones ([la figura A.9 muestra un ejemplo](#)).

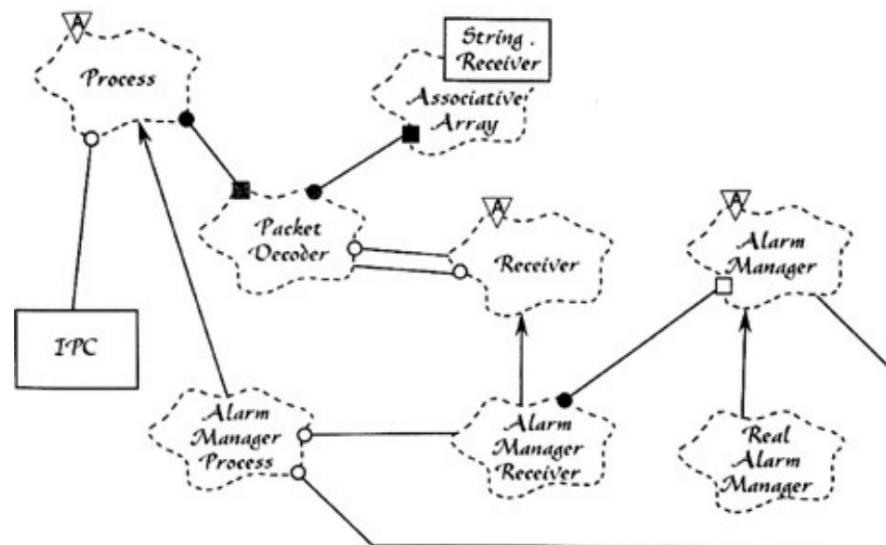


Figura A.9 Un diagrama de Booch

La notación de Booch era muy poderosa. Presagiaba notaciones como UML.

ROSE tenía una arquitectura, una arquitectura real . Se construyó en capas verdaderas y las dependencias entre capas se controlaron adecuadamente. La arquitectura lo hizo liberable, desarrollable y desplegable de forma independiente.

Oh, no fue perfecto. Había muchas cosas que todavía no entendíamos sobre los principios arquitectónicos. Por ejemplo, no creamos una verdadera estructura de complementos.

También caímos en una de las modas más desafortunadas del momento: utilizamos la llamada base de datos orientada a objetos.

Pero, en general, la experiencia fue estupenda. Pasé un maravilloso año y medio trabajando con el equipo de Rational en ROSE. Esta fue una de las experiencias más estimulantes intelectualmente de mi vida profesional.

LOS DEBATES CONTINUARON

Eso sí, no paré de debatir en Netnews. De hecho, aumenté drásticamente mi presencia en la red. Empecé a escribir artículos para C++ Report. Y, con la ayuda de Grady, comencé a trabajar en mi primer libro: Diseño de aplicaciones C++ orientadas a objetos utilizando el método Booch.

Una cosa me molestó. Era perverso, pero era cierto. Nadie me llamaba "tío Bob". Descubrí que me lo perdí. Así que cometí el error de poner "Tío Bob" en mi correo electrónico y en mis firmas de Netnews. Y el nombre se quedó. Al final me di cuenta de que era una marca bastante buena.

... POR CUALQUIER OTRO NOMBRE

ROSE era una aplicación C++ gigantesca. Estaba compuesto por capas, con una regla de dependencia estrictamente aplicada. Esa regla no es la regla que he descrito en este libro. No apuntamos nuestras dependencias hacia políticas de alto nivel. Más bien, dirigimos nuestras dependencias en la dirección más tradicional del control de flujo. La GUI apuntaba a la representación, que apuntaba a las reglas de manipulación, que apuntaban a la base de datos. Al final, fue esta incapacidad para orientar nuestras dependencias hacia la política lo que contribuyó a la eventual desaparición del producto.

La arquitectura de ROSE era similar a la arquitectura de un buen compilador.

La notación gráfica fue "analizada" en una representación interna; esa representación luego fue manipulada mediante reglas y almacenada en una base de datos orientada a objetos.

Las bases de datos orientadas a objetos eran una idea relativamente nueva y el mundo OO estaba alborotado con sus implicaciones. Todo programador orientado a objetos quería tener una base de datos orientada a objetos en su sistema. La idea era relativamente simple y profundamente idealista. La base de datos almacena objetos, no tablas. Se suponía que la base de datos debía parecerse a la RAM. Cuando accedías a un objeto, simplemente aparecía en la memoria. Si ese objeto apuntara a otro objeto, el otro objeto aparecería en la memoria tan pronto como accediera a él. Fue como magia.

Esa base de datos fue probablemente nuestro mayor error práctico. Queríamos la magia, pero lo que obtuvimos fue un marco de terceros grande, lento, intrusivo y costoso que hizo de nuestras vidas un infierno al impedir nuestro progreso en casi todos los niveles.

Esa base de datos no fue el único error que cometimos. De hecho, el mayor error fue la arquitectura excesiva. Había muchas más capas de las que he descrito aquí, y cada una tenía su propio tipo de comunicaciones generales. Esto sirvió para reducir significativamente la productividad del equipo.

De hecho, después de muchos años de trabajo, inmensas luchas y dos lanzamientos tibios, toda la herramienta fue descartada y reemplazada por una pequeña y linda aplicación escrita por un pequeño equipo en Wisconsin.

Y así aprendí que las grandes arquitecturas a veces conducen a grandes fracasos.

La arquitectura debe ser lo suficientemente flexible para adaptarse al tamaño del problema.

La arquitectura para la empresa, cuando todo lo que realmente necesita es una pequeña y linda herramienta de escritorio, es una receta para el fracaso.

EXAMEN DE REGISTRO DE ARQUITECTOS

A principios de los años 1990, me convertí en un verdadero consultor. Viajé por el mundo enseñando a la gente qué era esta nueva cosa de OO. Mi consultoría estuvo fuertemente enfocada al diseño y arquitectura de sistemas orientados a objetos.

Uno de mis primeros clientes de consultoría fue Educational Testing Service (ETS). Fue

bajo contrato con la Junta de Registro del Consejo Nacional de Arquitectos (NCARB) para realizar los exámenes de registro para nuevos candidatos a arquitectos.

Cualquiera que desee ser arquitecto registrado (el tipo de arquitecto que diseña edificios) en los Estados Unidos o Canadá debe aprobar el examen de registro. Este examen implicó que el candidato resolviera una serie de problemas arquitectónicos relacionados con el diseño de edificios. Al candidato se le puede dar un conjunto de requisitos para una biblioteca pública, un restaurante o una iglesia, y luego pedirle que dibuje los diagramas arquitectónicos apropiados.

Los resultados se recopilarían y guardarían hasta el momento en que un grupo de arquitectos experimentados pudiera reunirse como jurado para calificar las presentaciones. Estas reuniones fueron eventos grandes y costosos y causaron mucha ambigüedad y retrasos.

NCARB quería automatizar el proceso haciendo que los candidatos realizaran los exámenes usando una computadora y luego otra computadora hiciera la evaluación y calificación. NCARB le pidió a ETS que desarrollara ese software y ETS me contrató para reunir un equipo de desarrolladores para producir el producto.

ETS había dividido el problema en 18 viñetas de prueba individuales. Cada uno requeriría una aplicación GUI similar a CAD que el candidato usaría para expresar su solución. Una aplicación de puntuación separada aceptaría las soluciones y produciría puntuaciones.

Mi socio, Jim Newkirk, y yo nos dimos cuenta de que estas 36 aplicaciones tenían grandes similitudes. Las 18 aplicaciones GUI utilizaron gestos y mecanismos similares. Todas las 18 aplicaciones de puntuación utilizaron las mismas técnicas matemáticas. Teniendo en cuenta estos elementos compartidos, Jim y yo estábamos decididos a desarrollar un marco reutilizable para las 36 aplicaciones. De hecho, vendimos esta idea a ETS diciendo que pasaríamos mucho tiempo trabajando en la primera aplicación, pero que luego el resto aparecería cada pocas semanas.

En este punto deberías estar dándote palmadas en la cara o golpeándote la cabeza con este libro. Aquellos de ustedes que tengan edad suficiente tal vez recuerden la promesa de “reutilización” de OO. En aquel entonces, todos estábamos convencidos de que si escribías código C++ limpio y orientado a objetos, naturalmente producirías muchísimo código reutilizable.

Así que nos dispusimos a escribir la primera aplicación, que era la más complicada del lote. Se llamaba Viñeta Grande.

Los dos trabajamos a tiempo completo en Vignette Grande con miras a crear un marco reutilizable. Nos llevó un año. A finales de ese año teníamos 45.000 líneas de código marco y 6.000 líneas de código de aplicación. Entregamos este producto a ETS y ellos nos contrataron para redactar las otras 17 solicitudes rápidamente.

Entonces Jim y yo reclutamos un equipo de otros tres desarrolladores y comenzamos a trabajar en las siguientes viñetas.

Pero algo salió mal. Descubrimos que el marco reutilizable que habíamos creado no era particularmente reutilizable. No encajaba bien en las nuevas aplicaciones que se estaban escribiendo. Hubo fricciones sutiles que simplemente no funcionaron.

Esto fue profundamente desalentador, pero creímos que sabíamos qué hacer al respecto. Fuimos a ETS y les dijimos que habría un retraso: que era necesario reescribir el marco de 45.000 líneas, o al menos reajustarlo. Les dijimos que tomaría un poco más de tiempo lograrlo.

No necesito decirles que ETS no estaba particularmente contento con esta noticia.

Así que empezamos de nuevo. Dejamos a un lado el viejo marco y comenzamos a escribir cuatro nuevas viñetas simultáneamente. Tomaríamos prestadas ideas y códigos del marco anterior, pero los reelaboraríamos para que encajen en los cuatro sin modificaciones.

Este esfuerzo tomó otro año. Produjo otro marco de 45.000 líneas, más cuatro viñetas de entre 3.000 y 6.000 líneas cada una.

No hace falta decir que la relación entre las aplicaciones GUI y el marco seguía la regla de dependencia. Las viñetas eran complementos del marco. Toda la política de GUI de alto nivel estaba en el marco. El código de la viñeta era sólo pegamento.

La relación entre las aplicaciones de puntuación y el marco era un poco más compleja. La política de puntuación de alto nivel estaba en la viñeta. El marco de puntuación se conectó con la viñeta de puntuación.

Por supuesto, ambas aplicaciones eran aplicaciones C++ vinculadas estéticamente, por lo que la noción de complemento no estaba en ninguna parte de nuestras mentes. Y, sin embargo, la forma en que funcionaban las dependencias era consistente con la Regla de Dependencia.

Una vez entregadas esas cuatro solicitudes, comenzamos con las siguientes cuatro. Y esto

Es hora de que comenzaran a aparecer por la parte trasera cada pocas semanas, tal como lo habíamos predicho. El retraso nos había costado casi un año de nuestro cronograma, por lo que contratamos a otro programador para acelerar el proceso.

Cumplimos con nuestras fechas y nuestros compromisos. Nuestro cliente quedó contento. Éramos felices. La vida era buena.

Pero aprendimos una buena lección: no se puede crear un marco reutilizable hasta que primero se crea un marco utilizable. Los marcos reutilizables requieren que los construyas junto con varias aplicaciones de reutilización.

CONCLUSIÓN

Como dije al principio, este apéndice es un tanto autobiográfico. He llegado a los puntos destacados de los proyectos que sentí que tuvieron un impacto arquitectónico. Y, por supuesto, mencioné algunos episodios que no eran exactamente relevantes para el contenido técnico de este libro, pero que de todos modos fueron significativos.

Por supuesto, esta fue una historia parcial. Hubo muchos otros proyectos en los que trabajé a lo largo de décadas. También detuve deliberadamente esta historia a principios de los años 1990, porque tengo otro libro que escribir sobre los acontecimientos de finales de los años 1990.

Espero que hayas disfrutado de este pequeño viaje al pasado de mis recuerdos; y que pudiste aprender algunas cosas en el camino.

1. Una de las historias que escuchamos sobre esta máquina en particular en ASC fue que fue enviada en un camión semirremolque grande junto con una casa de muebles. En el camino, el camión chocó a gran velocidad contra un puente. La computadora estaba bien, pero se deslizó hacia adelante y aplastó los muebles.

2. Hoy diríamos que tenía una frecuencia de reloj de 142 kHz.

3. Imagina la masa de ese disco. ¡Imagínate la energía cinética! Un día entramos y vimos poco metal. virutas que caen del botón del gabinete. Llamamos al hombre de mantenimiento. Nos aconsejó que apagáramos la unidad. Cuando vino a repararlo dijo que uno de los rodamientos se había desgastado. Luego nos contó historias sobre cómo estos discos, si no se reparaban, podían soltarse de sus amarres, atravesar paredes de bloques de concreto e incrustarse en los autos en el estacionamiento.

4. Tubo de rayos catódicos: monocromático, pantalla verde, pantallas ASCII.

5. El número mágico 72 provino de las tarjetas perforadas de Hollerith, que contenían 80 caracteres cada una. los ultimos 8 Los caracteres estaban "reservados" para números de secuencia en caso de que se le cayera la baraja.

6. Sí, entiendo que es un oxímoron.

7. Tenían una pequeña ventana de plástico transparente que permitía ver el chip de silicona en el interior y dejaba pasar la luz ultravioleta. para borrar los datos.

8. Sí, sé que cuando el software se graba en la ROM, se llama firmware, pero incluso el firmware es realmente

todavía suave.

9. RKO7.

10. Posteriormente pasó a llamarse El único software exitoso de Bob.

11. Nuestra empresa poseía la patente. Nuestro contrato de trabajo dejaba claro que todo lo que inventáramos pertenecía a nuestra empresa. Mi jefe me dijo: "Nos lo vendiste por un dólar y no te pagamos ese dólar".

12. Pizarra negra tridimensional. Si nació en la década de 1950, probablemente obtenga esta referencia: Llovizna, Llovizna, Llovizna, Zángano.

13. Ingeniería de software asistida por computadora

ÍNDICE

Números

- Sistema de memoria compartida 3DBB, proyecto de arqueología Craft Dispatch System, [363](#) [_____](#)
- 4-TEL, proyectos de arqueología
- JEFE, [351–352](#) [_____](#)
 - Lenguaje C, [349–351](#) [_____](#)
 - DLU/DRU, [354–356](#) [_____](#)
 - descripción general de, [339](#)–[_____](#)
 - 344 pCCU, [352–354](#)
 - SAC (computadora del área de servicio), [344–349](#)
 - VRS, [357–359](#) [_____](#)
- 8085 computadora, proyectos arqueológicos
- 4-TELÉFONO, [341](#)
 - JEFE, [351](#) [_____](#)
 - Lenguaje C y, [349–351](#) [_____](#)
 - DLU/DRU, [356](#) [_____](#)
- Microcomputadora Intel 8086, proyecto de arqueología SAC, [347–348](#) [_____](#) [_____](#)

A

clases abstractas

- conclusión, [132](#) [_____](#)
- Principio de inversión de dependencia y, [87](#) [_____](#)
- sobrantes en Zona de inutilidad, [129–130](#) [colocación](#)
- de políticas de alto nivel, [126–128](#) [servicios](#)
- en Java como conjunto de, [246](#) [_____](#)

Componentes abstractos, [125–126](#)
Fábricas abstractas, [89–90](#)
Principio de estabilidad de las abstracciones. Consulte las dependencias del código fuente SAP (principio de abstracciones estables) y, [87](#)
estable, [88–89](#) modificadores de acceso, paquetes [_____](#)
arquitectónicos, [316–319](#) duplicación [_____](#)
accidental, [154–](#)
155 actores, [62–65](#) segmentos de direcciones, binarios [_____](#)
reubicables, [99–100](#) ADP (acíclico Principio de dependencias) ciclo de [_____](#)
ruptura, [117–118](#) gráfico de dependencia de [_____](#)
componentes afectado por, [118](#) efecto del ciclo en el gráfico de [_____](#)
dependencia de componentes, [115–117](#) [_____](#)
eliminación [_____](#)
de ciclos de [_____](#)
dependencia, [113–115](#) [_____](#)
nerviosismo, [118](#) descripción general de, [112](#) construcción semanal, [112–](#)
113 Die- monitoreo de [_____](#)
fundición, proyecto de [_____](#)
arqueología, [338–339](#) API, pruebas, [252–253](#) Prueba de aptitud de [_____](#)
aplicación,
258–261 Reglas comerciales específicas de la [_____](#)
aplicación, casos de uso, [192–193](#), [204](#) Objetivo de [_____](#)
los arquitectos para minimizar los [_____](#)
recursos
humanos, [160](#) examen de registro [_____](#)
proyecto de arqueología , [370–373](#) detalles separados de la [_____](#)
política, [142](#) [_____](#)
Arquitectura limpia. Consulte Arquitectura [_____](#)
limpia integrada limpia. Consulte Diseño de arquitectura integrada [_____](#)
limpia versus 4 en el proyecto [_____](#)
de arqueología DLU/ [_____](#)
DRU, [356](#) Matriz de importancia versus [_____](#)
urgencia de Eisenhower, [16–17](#) acertar con el software, [2](#) inmutabilidad y [52](#) independencia. Ver Independencia

LSP y 80 —
complementos, 170—
171 en el producto de arqueología ROSE, 368—
370 en el proyecto de arqueología SAC, 345—
347 como senior para —
funcionar, 18 como valor de —
software, 14–15 —
estabilidad, —
122–126 pruebas, 213 tres —
grandes preocupaciones en , 24 —
valor de función versus 15–16 en el proyecto de arqueología VRS, 358–359

Proyectos de arqueología de arquitectura.

4-TEL, 339–344 —
monitoreo de fundición a presión de aluminio, 338—
339 examen de registro de arquitectos, —
370–373 por autor desde 1970, 325–326
Sistema operativo básico y programador, 351–352 — —
Lenguaje C, 349–351 —
conclusión, 373 —
Sistema de despacho de embarcaciones, 361–367
DLU/DRU, 354–356 —
Recepcionista electrónica, 359–361 —
Ajuste láser, 334–338 —
pCCU, 352–354 —
Producto ROSE, computadora
del área de servicio 368–370 , 344–349
Sistema de contabilidad sindical, 326–334 —
VRS, 357–359 —

Arquitectura, conclusión

definitoria, 146 —
implementación, 138 —
desarrollo, 137–138 —
independencia del dispositivo, 142—
143 ejemplo de correo basura, 144—
145 mantener abiertas las opciones, —
140–142 mantenimiento, 139–140

operación, 138–139
ejemplo de direccionamiento físico, 145–146
comprensión, 135–137

Arquitectura, gritos
acerca de la web, 197–198
conclusión, 199
marcos como herramientas, no formas de vida,
198 descripción general
de, 195–196
propósito de, 197 arquitecturas
comprobables, 198 tema, 196–197

Archivos, como agregación de componentes, 96.

Artefactos, OCP, 70

ASC Tabulación, proyecto de arqueología de Union Accounting, 326–334

Asignación y programación funcional, 23.

Matrimonio asimétrico, a los autores del marco, 292–293

Autores, marco, 292–293

Sistemas automatizados, reglas comerciales, 191–192

B

Clases base, marcos, 293.

Arquitectura del sistema BCE, 202

Beck, Kent, 258–261

Comportamiento (función)
la arquitectura respalda el sistema, 137, 148
Matriz de importancia versus urgencia de Eisenhower, 16–17 lucha
por la antigüedad de la arquitectura sobre la función, 18 mantener
abiertas las opciones, 140–142 como
valor del software, 14
valor de la función versus la arquitectura, 15–16

Binarios , reubicación, 99–100 Booch,

Grady introducción
a, 366 trabajando para,
367 trabajando en
el producto ROSE, 368–369

Proyecto de arqueología BOSS (Sistema operativo básico y programador), [351–352](#)
en proyecto de arqueología DLU/DRU, [356](#)

Límites en

proyecto de arqueología 4-TEL, [340–341](#)
conclusión, [173](#)
división de servicios en componentes, [246](#)
en proyecto de arqueología Recepcionista electrónica, [361](#) FitNesse programa, [163–](#)
165 entradas y salidas, [169–](#)
170 en el proyecto de arqueología Laser
Trim, [338](#) capas y. Consulte Descripción general de [capas y](#)
[límites](#), [159–160](#)
parcial, [218–220](#) arquitectura
de complemento, [170–171](#)
argumento de complemento, [172–173](#) historias
tristes de fallas arquitectónicas, [160–163](#)
servicios como
llamadas de función, [240](#) prueba, [249–253](#) Contabilidad sindical
proyecto de arqueología del sistema, [333–](#)
334 qué líneas dibujar
y cuándo, [165–169](#)
Anatomía de
límites cruce de límites, [176](#) conclusión,
[181](#) componentes de
despliegue, [178–179](#) temido
monolito, [176–178](#)
procesos
locales, [179–180](#)
servicios, [180–181](#)
subprocesos, [179](#) Cruce de límites en
arquitectura limpia, [206](#)
escenario de arquitectura limpia, [207–](#)
208 creación apropiada, [176](#) Regla de dependencia para [datos](#),
[207](#) Rotura de ciclo, Principio de dependencias acíclicas, [117–118](#) Gerentes de negocios

Matriz de importancia frente a urgencia de Eisenhower, [17](#) _____
preferencia por la función frente a la arquitectura, [15-16](#)
Reglas de negocio
límites entre GUI y, 169–170 arquitectura limpia
para, 202–203 conclusión, [194](#) _____
creación de _____
entidades, 190–191 _____
desacoplamiento de la interfaz de _____
usuario, 287–289 desacoplamiento
de capas, 152–153 _____
desacoplamiento de casos de _____
uso, [153](#) diseño para capacidad de prueba, [251](#) en Hunt
the Juego de aventuras Wumpus, _____
222–223 desarrollo _____
independiente, [47](#) _____
mantenerse cerca de los datos, [67](#) _____
conectarse, 170–173 cálculo de declaraciones
de políticas, [184](#) modelos de solicitud/ _____
respuesta y, 193–194 en el proyecto de arqueología SAC, [346](#)–
347 separar componentes _____
con líneas divisorias, [165–169](#) comprensión, 189–190 casos de uso para, [191–193](#), [204](#)

C

Herencia del
lenguaje C++ en, [40](#)
aprendizaje, _____
[366](#) casamiento con el marco STL en, _____
[293](#) polimorfismo en, [42](#)
Aplicación de ROSE, [369–370](#) _____
debilitamiento de la encapsulación, [36–37](#)

lenguaje c
Proyecto de arqueología BOSS utilizando, [351–352](#)
Proyecto de arqueología DLU/DRU utilizando, [356](#)
encapsulación en, [34–36](#) _____
herencia en, [38–40](#) _____

polimorfismo en, 40–42 rediseño
de SAC en, 347–348

Lenguaje C, proyecto de arqueología, 349–351

lenguaje de programación c#
componentes abstractos en, 125
inversión de dependencia, 45 uso
de declaraciones para dependencias, 184 debilitamiento
de la encapsulación, 36–37 Lenguaje de
programación C (Kernighan & Ritchie), 351 Modelo de arquitectura de
software C4, 314–315 Carew, Mike, 356 CASE (Ingeniería
de software asistida por
computadora) herramienta, 368 Estudio de caso. Ver [vídeo de estudio de caso](#)
[de ventas](#) Terminales de tubo de rayos catódicos
(CRT), proyecto de arqueología de Union Accounting,
328–329

Capas de desacoplamiento de CCP (principio
de cierre común), 152 ,
agrupación de políticas en componentes, 186–187 ,
mantenimiento de los cambios localizados,
118 descripción general de, 105–107
Principio de dependencias estables y diagrama de
tensión 120 , 108-110

CCU/CMU (unidad de control COLT/unidad de medición COLT), proyecto de arqueología pCCU, 353–354

CDS (Craft Dispatch System), descripción general del proyecto de
arqueología, 361–363

Probadores de línea de oficina central. Ver [COLT \(probadores de línea de oficina central\)](#)

Oficinas centrales (CO), proyecto de arqueología 4-TEL, 339–340 Cambio,
facilidad de software, 14–15 Church, Alonzo,
22–23, 50 programa CICS-COBOL,
proyecto de arqueología de aluminio fundido a presión, 339 Resumen de clases. Consulte el
Principio
de reutilización común [de las clases](#)
[abstractas](#) , 107–108 DIP y 89 Uso de LSP para
guiar la herencia,

procesos de partición en, 71–72
Principio de equivalencia de reutilización/liberación, [105](#)
Ejemplos de SRP, [67](#)
Características de la
arquitectura limpia, 201–[203](#)
conclusión, [209](#)
Regla de dependencia, 203–[207](#) los
marcos tienden a violar, [293](#) escenario
típico, [208](#) uso de capas y
límites, 223–226 Prueba de aptitud de [aplicación](#)
de arquitectura integrada limpia, 258–
261 conclusión, [273](#) don No
revela detalles de
hardware al usuario de HAL, 265–269 directivas de compilación
condicional DRY, [272](#) el hardware es un detalle, 263–[264](#)
es una arquitectura integrada
comprobable, [262](#) capas, 262–263 el sistema
operativo es un
detalle, 269–271 descripción general de, 255–
258 programación para
interfaces y sustituibilidad, 271–272 cuello de botella de hardware de
destino, [261](#) Cleancoders.com, [297](#) Clear
[Communications](#), 364–367
llamada telefónica, [367](#) configuración, [366](#)
Uncle Bob, [367](#)
Clojure, [50](#)–
[51](#), 53–54 Codd,
Edgar, [278](#) Código en
 proyecto de arqueología
de
fundición a presión de aluminio, 338–339 disminución de la
productividad/aumento del costo de , 5–7 tontería por exceso
de confianza, [9](#)–[12](#) costos crecientes de la
nómina de desarrollo, 8–9 en el proyecto de
arqueología SAC, [345](#) firma de desordenado,
7–8

Dependencias del código fuente. Ver [Dependencias del código fuente](#)

Conclusión de la

organización del

código, [321 el diablo está en los detalles](#),

[315–316 otros modos de desacoplamiento](#),

[319–320 descripción general](#)

de, [303–304 paquete por componente](#), [310–](#)

[315 paquete por característica](#), [306–](#)

[307 paquete por capa](#), [304–306](#)

puertos y adaptadores, [308–310 vs](#)

encapsulación, [316–319 Cohesión](#),

principio de responsabilidad única, [63 COLT \(probadores de](#)

línea de oficina central) en proyecto de

arqueología 4-TEL, [340–344 proyecto de arqueología](#)

pCCU, [352–354 en área de servicio proyecto de](#)

arqueología informática, [344–349 Principio de cierre común](#). Ver [PCC](#)

(Principio de Cierre Común)

Principio de reutilización común. Ver [CRP \(Principio Común de Reutilización\)](#)

Comunicaciones

a través de los límites de los componentes de implementación,

[179 a través de los límites de los procesos locales](#),

[180 a través de los límites de los servicios](#), [180–](#)

[181 a través de los límites desacoplados a nivel de fuente](#), [178](#)

Ley de Conway, [149](#)

como llamadas a funciones entre componentes en monolitos, [178 en tipos](#)

de modos de desacoplamiento, [155–157](#)

Algoritmo de comparación e intercambio, [54](#)

Idiomas compilados, [96](#)

Los

compiladores aplican principios arquitectónicos con [319](#)

ubicación del código fuente, [97–98](#)

binarios reubicables, [99–100](#)

Arquitectura de componentes, estudio de caso de ventas de videos, [300–302](#)

Sistemas basados en componentes

que construyen escalables,

[241 diseñan servicios usando SOLID](#), [245–246](#)

llamadas de función, [240](#)
Enfoque OO para preocupaciones transversales, [244–245](#) Principio
de cierre común de cohesión
de componentes, [105–107](#) Principio de [reutilización](#)
común, [107–108](#) conclusión, [110](#) descripción
general de, [104](#) _____
Principio de _____
equivalencia de reutilización/liberación, [104–105](#) diagrama de
tensión, [108–110](#) Acoplamiento de
componentes ADP. Consulte
la conclusión [del ADP \(Principio de dependencias acíclicas\)](#), [132](#)
Problema de [pruebas](#)
frágiles, [251](#) descripción general de,
[111](#) Principio de _____
abstracciones estables. Ver [SAP \(Principio de abstracciones estables\)](#)

Principio de dependencias estables, [120–126](#) diseño
de arriba hacia abajo, [118–119](#) _____
Gráfico de dependencia de componentes
romper ciclo de componentes/restablecer como DAG, [117–118](#) efecto del
ciclo de entrada, [115–117](#) _____
Arquitectura de componente por equipo, [137–138](#) _____
Componentes
concretos, [91](#) _____
implementación de, [178–179](#) _____
historia de, [96–99](#) _____
enlazadores, [100–102](#)
descripción general _____
de, [96](#) paquete por, [313–](#)
315 procesos de partición en clases/separación de clases en, [71–72](#) principios, [93](#) _____
reubicabilidad, [99–](#)
[100](#) pruebas como sistema,
[250](#) _____
Herramienta de ingeniería de software asistida por computadora (CASE), [368](#)
Componentes concretos, Principio de inversión de dependencia, [91](#) _____
Tareas concurrentes, proyecto de arqueología BOSS, [351–352](#) _____
Actualizaciones simultáneas, [52–53](#)

Constantine, Larry, 29 —
Control, flujo de. Consulte [Flujo de control](#)
Estructuras de control, programa, 27–28 —
Control, transferencia de, —
22
 Controladores en arquitectura limpia,
 203, 205 escenario de arquitectura limpia,
 207–208 cruce de límites circulares,
 206 Ley de Conway,
 149 Cables de cobre, proyecto de arqueología pCCU, 352 –354
Código central, evitar marcos, 293 CO —
(oficinas centrales), proyecto de arqueología 4-TEL, 339–340 — —
Acoplamiento. Véase también [El acoplamiento de componentes](#) evita permitir el marco,
 293 a decisiones prematuras, 160
Sistema de despacho de embarcaciones. Véase [CDS \(Craft Dispatch System\)](#), proyecto de arqueología Critical Business Data, 190—
191 Critical Business Rules, 190–193 —
Preocupaciones transversales
 en el diseño de servicios a tratar, 247 —
 Enfoque orientado a objetos, 244–245 Cruce
de flujos de datos, 226 CRP —
(Principio de reutilización común) que
 influye en la composición de los componentes, 118 —
 descripción general de,
 107–108 diagrama de tensión, 108–110
Terminales CRT (tubo de rayos catódicos), proyecto de arqueología de Union Accounting, 328–329 —

Rompiendo ciclos, 117—
 118 efecto del gráfico de dependencia, 115–117
 eliminando la dependencia, 113–115 —
 problemas de compilación semanales, 112–113

D
Métrica D , distancia desde la secuencia principal, 130–132 —

Marco arquitectónico de políticas DAG

(gráficos acíclicos dirigidos), 184 ciclo de ruptura de componentes, 117–118 definido, 114

Dahl, Ole Johan, 22

Escenario de arquitectura de limpieza de datos, 207–208

Regla de dependencia para cruzar fronteras, 207 preocupaciones

de gestión en arquitectura, 24 mapeadores, 214–215

separación de funciones,

66

Modelo de datos, base de datos versus,

278

Almacenamiento de datos en el proyecto de

arqueología Laser Trim, 335 prevalencia de sistemas de bases de datos

debido a discos, 279–280 en el proyecto de arqueología de Union

Accounting,

327–328 Arquitectura limpia de base de datos

independiente de, 202 escenario de arquitectura

limpia, 207–208 crear arquitectura comprobable sin, 198

capas de desacoplamiento,

153 casos de uso de

desacoplamiento, 153 regla de

dependencia, 205 trazar una línea límite entre las reglas de negocio y, 165

puertas de enlace,

214 en el juego de aventuras Hunt the Wumpus, 222–223

desarrollo independiente, 47 dejar

opciones abiertas en desarrollo, 141, 197 arquitectura de

complementos, 171 relacional,

278 esquema en

Zone of Pain, 129 componentes

separadores con líneas límite, 165–169

La base de datos es detalle

anécdota, 281–283

conclusión, 283

detalles, 281

si no hubiera discos, 280–281 _____
descripción general de, _____
277–278 rendimiento, _____
281 bases de datos _____
relacionales, **278** por qué los sistemas de bases de datos _____
son tan frecuentes, 279–280 _____
arquitectura del sistema DCI, **202** Puntos _____
muertos,
debido a variables mutables, **52** _____
Desacoplamiento como falacia de _____
servicios, 240–241 implementación independiente, _____
154, 241 desarrollo independiente, **153**–
154, 241 ejemplo _____
de problema de gatito, _____
242–243 capas, 151–152 modos, **153**, 155–158 Enfoque _____
OO para inquietudes transversales, _____
244–245 propósito de las pruebas _____
API, 252–253 _____
dependencias de _____
código fuente,
319 casos de uso, **152** DeMarco, Tom, **29** dependencias _____
ADP. Consulte el marco arquitectónico de _____
políticas ADP (Principio de dependencias
acíclicas), **184** cálculo de métricas de _____
estabilidad, **123** estudio de caso. Consulte el _____
estudio de caso de ventas en vídeo Principio común de _____
reutilización y 107–108 DIP. Consulte **DIP** _____
(Principio de inversión de _____ _____
dependencia) en el _____
proyecto de arqueología **Laser Trim**, **338** _____
gestión de software no deseado, 89–90 ejemplo _____
de OCP, **72** en paquete por capa, **304**–**306**, 310–311
software _____
destruido por software no _____
administrado, **256** estable. Consulte **SDP** (Principio de _____ _____
dependencias estables) transitivo, **75** componente de comprensión, **121** en el proyecto de arqueología de Union A _____

Marco de inyección de dependencia, componente principal, [232](#)

Inversión de dependencia, [44–47](#)

Gestión de dependencia

métrica. Consulte [ADP \(Principio de dependencias acíclicas\)](#)

a través de límites arquitectónicos completos, [218](#) a

través de polimorfismo en sistemas monolíticos, [177](#)

estudio de caso de ventas de

video, [302](#) arquitectura

limpia de reglas de dependencia y,

203–206 escenario de arquitectura limpia,

207–208 límites cruzados,

[206 definidos](#),

[91](#) gestión de dependencias, [302](#)

diseño de servicios a seguir, [247](#)

entidades, [204](#)

marcos y controladores, [205](#)

marcos que tienden a violarse, [293](#) en el

juego de aventuras Hunt the Wumpus, [223](#)

adaptadores de interfaz,

[205](#) enfoque OO para preocupaciones transversales, [244–](#)

245 servicios pueden seguir,

[240](#) pruebas siguientes,

[250](#) casos de

uso, [204](#) qué datos cruzan fronteras, [207](#)

La arquitectura

de implementación determina la facilidad

de, [150](#) componentes,

178–180 componentes como

unidades de, [96](#) impacto de la

arquitectura en, [138](#) pruebas

usan independiente, [250](#) Desacoplamiento a nivel de

implementación modo, [156–157](#), [178–179](#)

Enfoques de diseño

para. Consulte Arquitectura de organización de código vs.,

[4](#) disminución de la productividad/aumento del costo del código, 5–7 hacerlo bien, [2](#)

objetivo del bien,[4–](#)
5 reducir la volatilidad de las interfaces,[_____](#)
[88](#) firma de un desorden,[7–](#)
8 principios SOLID de, [57–59](#) [_____](#)
para la capacidad [de](#)
prueba, [251](#) Diseño de aplicaciones C++ orientadas a objetos utilizando el método Booch,[_____](#)
[369](#) La
base de datos detallada es. Consulte [La](#)
[base de datos es un detalle](#), no revela el hardware, para
el usuario de HAL, [265–269](#)
el marco es, [291–295](#) el
hardware es, [263–264](#) se separa de la
política, [140–142](#) historia de éxito arquitectónico,
[163–165](#) la web es,
285–289 Los
desarrolladores disminuyen la productividad/aumentan el costo
del código, [5–7](#) Matriz de importancia versus urgencia de _____
Eisenhower, [17](#) la tontería del exceso de _____
confianza, [9–12](#) preferencia por la función versus la _____
arquitectura, [15–16](#) alcance versus forma para determinar _____
el costo del cambio, [15](#) firma
de un desastre, [8–9](#) _____
como partes
interesadas, [18](#) Impacto de la arquitectura
en el desarrollo, [137–138](#) independientes. Consulte
Función de desarrollo independiente de la _____
arquitectura en el soporte, [149–](#)
150 función de prueba
para soportar, [250](#)
Definición de independencia del _____
dispositivo, [142–143](#) dispositivo _____
IO de UI como, [288–289](#) ejemplo de correo _____
basura, [144–145](#) _____
ejemplo de direccionamiento físico, [145–146](#) en programación, _____
[44](#) Revolución digital y compañías telefónicas, [352–354](#) Dijkstra, Edsger Wybe
aplicando la disciplina de la prueba a la programación, [27](#)

descubrimiento de la programación estructurada, 22
historia de, 26
proclamación sobre declaraciones goto, 28–29 sobre
pruebas, 31

DIP (principio de inversión de dependencia)
romper el ciclo de componentes, 117–118 conclusión,
91 componentes
concretos, 91 cruzar límites
circulares, 206 definidos, 59 dibujar líneas
límite, 173

Entidades sin conocimiento de casos
de uso como, 193 fábricas, 89–90 en buena arquitectura de
software, 71 no todos
los componentes deben ser estables, 125
descripción general de, 87–88 abstracciones estables,
88–89 Principio de
abstracciones estables, 127 Gráfico
acíclico dirigido. Ver DAG (gráficos acíclicos)

dirigidos)

Control direccional, principio abierto-cerrado, 74 Discos si no
los
hubiera, 280–281 prevalencia de
sistemas de bases de datos debido a, 279–280 en el proyecto de
arqueología de Union Accounting, 326–330

Código de despacho, proyecto informático del área de servicio, 345

Mostrar proyecto de arqueología de unidad local/unidad remota (DLU/DRU),
354–356

DLU/DRU (unidad local de visualización/unidad remota de visualización), proyecto de arqueología,
354–356

Declaraciones hacer/mientras/hasta, 22, 27

Principio de no repetirse (DRY), directivas de compilación condicional, 272 Líneas de dibujo. Consulte

Controladores de límites , regla de
dependencia, principio 205 DRY (no se
repita), directivas de compilación condicional, 272 DSL (lenguaje basado en datos de dominio específico),
 proyecto de arqueología Laser Trim,
337

Duplicación

accidental, 63–65 [_____](#)

verdadero versus accidental, 154–

155 Polimorfismo dinámico, 177–178, 206 [_____](#)

Bibliotecas vinculadas dinámicamente, como límites arquitectónicos, 178–179 [_____](#)

Lenguajes tipados dinámicamente

DIP y, 88 ISP [_____](#)

y, 85 [_____](#)

E

Edición, proyecto de arqueología Laser Trim, 336 [_____](#)

Educational Testing Service (ETS), 370–372 [_____](#)

Eisenhower, matriz de importancia versus urgencia, 16–17 [_____](#)

Arquitectura integrada. Consulte Encapsulación [de arquitectura](#)

inteligencia limpia

al definir la programación [_____](#)

orientada a objetos, 35–37 [_____](#)

organización versus, 316–319 [_____](#)

uso excesivo de público y, 316 entidades

reglas de negocio y, 190–191 [_____](#)

escenario de arquitectura limpia, 207–208 [_____](#)

creación de arquitectura comprobable, [_____](#)

198 Regla de dependencia [_____](#)

para, 204 riesgos de marcos,

293 casos de uso [versus](#),

191–193 Enumeración, prueba de Dijkstra para secuencia/selección,

28 EPROM (borrable Chips de memoria de solo lectura programables, proyecto de arqueología 4-TEL, 341–343 [_____](#)

ER (receptor electrónico)

proyecto de arqueología, 359–361 [_____](#)

El sistema de despacho de embarcaciones fue, 362–364 [_____](#)

ETS (Servicio de pruebas educativas), 370–372 [_____](#)

Europa, rediseñando SAC para EE. UU. y, 347–348 [_____](#)

Abastecimiento de eventos, almacenamiento de transacciones, 54–55 [_____](#)

Ejecutables

despliegue de monolitos, 176–178 [_____](#)

- vincular componentes como, [96](#)
 - Agencia externa, arquitectura limpia independencia de, [202](#)
 - Definición externa, compiladores, [100](#)
 - Referencia externa, compiladores, [100](#)
- F
- Patrón de fachada, límites parciales, [220](#)
 - Métricas de distribución de entrada/salida, estabilidad de componentes, [122–123](#)
 - Plumas, Michael, [58](#)
 - Sistemas de archivos, mitigación del retraso de tiempo, [279–280](#)
 - Cortafuegos, cruces de límites vía, [176](#)
 - Firmware en
 - el proyecto de arqueología 4-TEL, [343–344](#)
 - definiciones de, [256–257](#)
 - eliminando el cuello de botella del hardware objetivo, [262–263](#)
 - línea difusa entre software y, [263–264](#) obsoleto a
 - medida que el hardware evoluciona, [256](#)
 - deja de escribir tanto, [257–258](#)
 - Descripción general
 - del programa FitNesse, [163–](#)
 - 165 límite parcial, [218–219](#)
 - FLD (Field Labeled Data), proyecto de arqueología del Craft Dispatch System, [363](#)
 - flujo de control
 - cruce de límites circulares, [206](#) gestión
 - de dependencia, estudio de caso, [302](#) polimorfismo
 - dinámico, [177–178](#) en el proyecto de
 - arqueología de Union Accounting, [334](#)
 - Fowler, Martín, [305–306](#)
 - Problema de pruebas frágiles, [251](#)
 - Los marcos
 - evitan basar la arquitectura en, [197](#)
 - arquitectura limpia independiente de, [202](#) crear
 - arquitectura comprobable sin, [198](#)
 - Regla de dependencia para [205](#)
 - como opción que se dejará abierta, [197](#)

como herramientas, no como una forma de vida, [198](#)

Los marcos son detalles del

matrimonio asimétrico y, 292–293 conclusión, [295](#)

autores del marco, [292](#)

marcos con los que simplemente

debes casarte, [295](#) popularidad de, [292](#) riesgos, 293–294

solución, [294](#)

Llamadas a funciones, servicios como, [240](#)

Mejores prácticas de programación de

descomposición funcional, [32](#) en programación

estructurada, [29](#)

Punteros funcionales, programación orientada a objetos, [22, 23](#)

Conclusión de la programación

funcional, [56](#) origen

de eventos, 54–55 historia de, [22–](#)

23 inmutabilidad, [52](#)

descripción general de,

[50](#) segregación de

mutabilidad, 52–54 ejemplo de cuadrados de

números enteros, 50–51

Las funciones

evitan anular lo concreto, [89](#) dividirse en

partes (descomposición funcional), [29](#) una de las tres grandes preocupaciones en

arquitectura, [24](#) principio de hacer una cosa, [62](#) separarse de los

datos, [66](#)

Ejemplos de SRP, [67](#)

GRAMO

Puertas de enlace, base de datos, [214](#)

Computadora GE Datanet [30](#), proyecto de arqueología de Union Accounting, 326–330

Ir a declaraciones

Dijkstra reemplaza con estructuras de control de iteración, [27](#)

Proclamación de Dijkstra sobre la nocividad de, 28–29 [historia](#)
de la programación estructurada, 22 [eliminada](#)
en la programación estructurada, 23

Software orientado a objetos en crecimiento con pruebas (Freeman & Pryce), 202 [GUI](#)
(interfaz gráfica de usuario). Consulte también [UI \(interfaz de usuario\)](#) que
desacopla las reglas comerciales de, 287–289 [diseño](#)
para capacidad de prueba, 251

examen de registro de arquitectos en desarrollo, 371–372

líneas de entrada/salida y límites, 169–170 [arquitectura](#)
de complemento, 170–171 [argumento](#)
de complemento, 172–173

separándose de las reglas comerciales con límites, 165–169 [pruebas](#)
unitarias, 212 [web](#)
is, 288

h

HAL (capa de abstracción de hardware)
evita revelar detalles del hardware al usuario, 265–269 [como línea](#)
divisoria entre software/firmware, 264

Directivas de compilación condicional DRY, 272 [el sistema](#)
operativo es detallado y, 269–271

Hardware

que elimina el cuello de botella del hardware objetivo con capas, 262–263 [el](#)
firmware se vuelve obsoleto a través de la evolución de, 256 [en el](#)
proyecto de arqueología SAC, 346–347

Archivos de encabezado, programación para interfaces con, 272

Arquitectura Hexagonal (Puertos y Adaptadores), 202

Desacoplamiento de

políticas de alto nivel de políticas de entrada/salida de nivel inferior, 185–186
separación de detalles, 140–142 [división](#)
de flujos de datos, 227–228 [dónde](#)
colocar, 126

Recursos humanos, objetivo del arquitecto minimizar, 160

Patrón de objeto humilde

mapeadores de datos, 214–215

escapadas de bases de datos, [214](#)
Presentadores y opiniones, [212–213](#)
Presentadores como forma de,
[212](#) pruebas y arquitectura, [213](#)
comprensión, [212](#)
Caza el juego Wumpus
capas y límites. Ver [Capas y límites Componente principal](#)
de, [232–237](#)

I
IBM System/7, proyecto de arqueología de fundición a presión de [aluminio](#),
338–339 Declaraciones [If/then/else](#),
[22, 27](#) Inmutabilidad, [52–](#)
54 Estrategia de implementación. Consulte [Organización](#)
del código Importancia, urgencia frente a la matriz de [Eisenhower](#),
16–17 Dependencias entrantes, métricas de estabilidad, [122–](#)
123 Conclusión
de independencia,
[158](#) capas de desacoplamiento,
151–152 modo de
desacoplamiento, [153](#) casos
de uso de
desacoplamiento, [152](#)
implementación, [150](#)
desarrollo, 149–150 duplicación, [154–](#)
155 implementabilidad independiente, [154](#)
desarrollo independiente, [153–154](#)
dejar opciones
abiertas, [150–151](#)
operación, [149](#) descripción general de, [147–](#)
148 tipos de
modos de desacoplamiento,
155–158 casos de uso, [148](#)
Componentes
independientes que calculan métricas de estabilidad, [123](#) comprensión, [121](#) Implementación independiente

en el proyecto de arqueología 4-TEL, 344 como
falacia de los servicios, 241 ejemplo
del problema del gatito, 242–243 en el _____
enfoque OO para preocupaciones transversales, 244–245 descripción
general de, 154 _____

Desarrollabilidad independiente como
falacia de los servicios, 241 ejemplo
del problema del gatito, 242–243 en el _____
enfoque OO para preocupaciones transversales, 244–245 descripción
general de, 153–154 de la _____
interfaz de usuario y la base de datos, 47

Inducción, prueba de Dijkstra relacionada con la iteración, 28 _____

Ocultamiento de información, principio abierto-cerrado, 74–75 _____

Relaciones de herencia que
cruzan los límites del círculo, 206 definen _____
la programación orientada a _____
objetos, 37–40 inversión de _____
dependencia, 46 gestión de dependencia, _____
302 guían el uso de, 78 _____

Reglas de
negocio de entrada/salida para casos de uso, 193–
194 desvinculando la política de nivel superior del nivel inferior, 185–187 nivel _____
de política definido como la distancia desde, 184 _____
separando componentes con líneas límite, 169–170 _____

Enteros, ejemplo de programación funcional, 50–51 Integración, _____
problemas de compilación semanales, 112–113 _____

Adaptadores de interfaz, Regla de dependencia para, 205 _____

Principio de segregación de interfaz. Ver ISP (Principio de segregación de interfaz) _____

Funciones
UNIX del dispositivo IO, 41–44 _____
web is, 288–289 _____

Aislamiento, prueba, 250–251 _____

Arquitectura ISP (Principio de segregación de
interfaz) y, 86 Principio de _____
reutilización común en comparación con, 108 conclusión, 86 _____

definido, 59 —

tipo de lenguaje y, 85 —

descripción general de, 84–85

Iteración, 27–28

J Jacobson, Ivar, 196, 202 —

Arquitectura de componentes de —

archivos Jar, 301 —

componentes como, 96 creación de —

límites parciales, 219 definición de funciones —

de componentes, 313 diseño de servicios basados en —

componentes, 245–246 regla Descargar

e Ir para, 163 en desacoplamiento a nivel de —

fuente modo, 176 componentes —

abstractos de Java en, 125 enfoques de organización de código en —

Consulte Componentes de —

organización —

de código como archivos jar en, 96 DIP y 87 —

declaraciones de — —

importación para dependencias, 184 ejemplo de ISP, —

84–85 uniendo el marco de —

biblioteca estándar en, 293 marcos —

de módulos en, 319 paquete por capa en, —

304–306 cuadrados de números enteros,

ejemplo en, 50–51 debilitamiento de la encapsulación,

36–37 Nerviosismo, interrupción del ciclo de componentes, 118 Ejemplo de correo basura, 144–145

k

Ejemplo de problema de Kitty, 242–245

|

Idiomas

arquitectura limpia y, 223–226 _____
Caza el juego de aventuras Wumpus, 222–223 _____
Laser Trim, proyecto de arqueología
Proyecto 4-TEL, 339 _____
descripción general de, 334–338
Paquete de arquitectura
en capas por organización de código de capas, 304–306 _____
relajado, 311–312 por _____
qué se considera malo, 310–311 Enfoque de _____
capas
para la organización del código, 304–306 uso de _____
arquitectura limpia, 202–203 desacoplamiento, 151–152 duplicación de, 155 eliminación del _____
cuello de botella del hardware de destino, 262–263 capacidad _____
de desarrollo independiente, 154 _____
Arquitectura limpia de capas
y límites, 223–226 conclusión, 228 _____
cruce de corrientes, 226 Juego de aventuras _____
Hunt the Wumpus, 222–223 descripción general de, 221–222 división de corrientes, 227–228 Herramienta Leiningen, _____
gestión de módulos, 104 Jerarquía de niveles de _____

protección y, 74 política y, 184–187 _____
Ubicación de bibliotecas _____
del código
fuente, 97–98 binarios reubicables, 99–100 Ciclo de vida, sistema de soporte _____
de arquitectura, 137 Enlazadores, separación de _____
cargadores, 100 –102 Liskov, Barbara, 78 Principio de _____
sustitución de Liskov _____
(LSP). Ver LSP (Principio de sustitución de Liskov) _____
Lenguaje LISP, programación funcional, 23 Lenguaje _____
Lisp, ejemplo de cuadrados de números enteros, 50–51 _____

Enlace de

cargadores, 100–102

binarios reubicables, 99–100

Límites del proceso local, 179–180

Arquitectura LSP (Principio de sustitución de Liskov)

y, 80 conclusión, 82 definido,

59 uso guía de la

herencia, 78

descripción general de, 78 problema de

ejemplo de cuadrado/

rectángulo, 79 violación de, 80–82

METRO

computadora m365

Proyecto de arqueología 4-TEL, 340–341

Proyecto de arqueología Laser Trim, 335–338

Proyecto de arqueología SAC, 345–347

Buzones de correo, los procesos locales se comunican a través de, 180

Conclusión del

componente principal,

237 como componente concreto, 91

definido, 232

programación orientada a objetos, 40

polimorfismo, 45 pequeño

impacto de la liberación, 115 como detalle

final, 232–237

Secuencia principal

evitando zonas de exclusión vía, 130 definiendo la

relación entre abstracción/estabilidad, 127–128 midiendo la distancia desde, 130–132

Zona de Dolor, 129

Zona de inutilidad, 129–130

Mantenimiento, impacto de la arquitectura en, 139–140

Campañas de marketing, proveedor de bases de datos, 283

Programa operativo maestro (MOP), proyecto de arqueología Laser Trim, [336 Matemáticas](#) que contrastan

la ciencia con, [30 disciplina de](#) _____

prueba, [27–28 herramienta](#) _____

Maven, gestión de módulos, [104 McCarthy](#), _____

John, [23 Diseño inicial](#) _____

de memoria

de, [98–99 procesos locales](#)

y, [179 RAM](#). Consulte [Fusiones de RAM](#), [ejemplos](#) _____

de SRP, [65 colas de mensajes](#), _____

procesos locales a través de los cuales se comunican, [180 métricas](#) _____

abstracción, [127](#) _____

distancia de la secuencia principal, [130–132](#) _____

Meyer, Bertrand, [70 años](#)

Arquitectura de microservicio

en el proyecto de arqueología Craft Dispatch System, [362–363 modo](#)

de desacoplamiento, [153](#) _____

estrategia de implementación,

[138 popularidad de](#), _____

[239 Módems](#), proyecto de arqueología SAC, [346–347](#) _____

Principio

de reutilización común de módulos, [107–108](#)

definidos, [62](#) _____

herramientas de gestión, [104](#)

tipos públicos vs. tipos publicados, [319](#) _____

Principio de equivalencia de reutilización/liberación,

[105 Monolitos](#)

que construyen sistemas escalables, _____

[241 componentes](#) a nivel de implementación _____

versus, [179 implementación](#) _____

de, [176–178 llamadas](#) _____

a funciones, [240 procesos locales vinculados](#) _____

estáticamente, _____

[180 subprocessos](#), [179 Ley de Moore](#), [101](#)

MOP (Master Operating Program), proyecto de arqueología Laser Trim, 336

Síndrome del día después

eliminar ciclos de dependencia para resolver, 113–115 [administrar](#)

dependencias para prevenir, 118 [descripción general](#)

de, 112 [problemas de](#)

compilación semanales, 112–113

MPS (sistema multiprocesamiento), proyecto de arqueología SAC, 345–346

Mutabilidad, 52–54

Variables mutables, 51, 54–55

N

Junta de Registro del Consejo Nacional de Arquitectos (NCARB), 370–372 .NET

componentes como DLL, 96 [NetNews](#),

presencia del autor en, 367–369 Newkirk, Jim, 371–372

Nygaard, Kristen, 22

oh

Bases de datos orientadas a objetos, producto ROSE, 368–370

Diseño orientado a objetos con aplicaciones (Booch), 366, 368

Conclusión de la programación orientada

a objetos, 47 para

preocupaciones transversales, 244–245

inversión de dependencia, 44–47

implementación de monolitos, 177

encapsulación, 35–37

historia de, 22

herencia, 37–40

descripción general de,

34–35 polimorfismo, 40–43

poder del polimorfismo, 43–44

Ingeniería de software orientada a objetos (Jacobson), 196, 202

Mapeadores relacionales de objetos (ORM), 214–215

Objetos inventados en el proyecto de arqueología 4-TEL, 344

OCP (principio abierto-cerrado)

nacimiento de, [142](#)

Principio de cierre común en comparación con, [106](#)

conclusión, [75](#)

en el proyecto de arqueología Craft Dispatch System, [363](#)

definido, [59](#)

gestión de dependencia, [302](#) diseño

de servicios basados en componentes, [246](#)

control direccional, [74](#)

ocultación de información, [74](#)–

75 descripción

general de, [70](#) experimento mental, [71](#)–[74](#)

OMC (Outboard Marine Corporation), proyecto de arqueología en fundición de aluminio, [338](#)–[339](#)

Límites unidimensionales, [219](#) Principio

abierto-cerrado. Ver [OCP \(Principio Abierto-Cerrado\)](#)

Capa de abstracción del sistema operativo (OSAL), arquitectura integrada limpia, [270](#)–[271](#)

Sistema operativo (SO), es un detalle, [269](#)–[271](#)

La

arquitectura de operaciones admite el sistema, [138](#)–

[139](#), [149](#) casos de uso de

desacoplamiento, [153](#) casos de uso

afectados por cambios en,

[204](#) Opciones, mantener abierta una buena arquitectura hace que el sistema sea fácil de cambiar, [150](#)–

[151](#) arquitectura operativa, [149](#) propósito de

la arquitectura, [140](#)–[142](#), [197](#)

mediante modo de desacoplamiento, [153](#)

Organización versus encapsulación, [316](#)–[319](#) ORM

(mapeadores relacionales de objetos), [214](#)–[215](#)

SO (sistema operativo), es detalle, [269](#) –[271](#) OSAL (capa de abstracción del sistema operativo), arquitectura integrada limpia, [270](#)–[271](#)

Oscilaciones, red como una de muchas, [285](#)–[289](#)

Dependencias salientes, métricas de estabilidad, [122](#)–[123](#)

Exceso de confianza, necesidad de, [9](#)–[12](#)

PAG

Paquete por componente, 310–315, 318 Paquete por
característica, 306–307, 317 Paquete por _____
modificadores de acceso
a capas, 317–318 capas horizontales _____
de código, 304–306 por qué se considera malo, 310–
311 Paquetes, organización versus encapsulación,
316–319 Page-Jones, Meilir, 29 Conclusión sobre límites parciales, 220
fachadas, 220 límites _____
unidimensionales, 219
razones para _____
implementar, 217–
218 omitir el último paso, 218–219 _____
_____ _____
_____ _____

Parches, en proyecto de arqueología 4-TEL, 344 PCCU, proyecto
de arqueología, 352–354 Computadora PDP-11/60, _____
349–351 Rendimiento, como preocupación _____
de bajo nivel, 281 Antipatrón periférico de puertos y _____
adaptadores, 320–321 Físico ejemplo de dirección, 145–146 Arquitectura de _____
complementos en el proyecto de arqueología 4-TEL, 344
para la independencia del
dispositivo, 44 trazar límites para el eje de cambio, _____
173 de componentes de nivel inferior en _____
componentes de nivel superior, 187 Componente principal _____
como, 237 comienzan con la presunción de , 170–171 Consejos para crear un _____
comportamiento polimórfico, 43 _____
funcional, 22–23 Política en arquitectura limpia, 203
de alto
nivel. Consulte Descripción general de políticas de alto
nivel de, 183–184 _____
sistemas
de software como declaraciones de, _____
183 _____
_____ _____

dividir flujos de datos, 227–228 _____

Despacho polimórfico, proyecto de arqueología 4-TEL, 344 _____

Polimorfismo que

- cruza límites circulares con dinámica, 206 inversión de _____
- dependencia, 44–47 flujo de control _____
- en dinámica, 177–178 en programación _____
- orientada a objetos, 22, 40–43 poder de, 43–44 _____

_____ _____

Modificadores de

- acceso a puertos y _____
- adaptadores, 318 enfoque para la organización del _____
- código, 308–310 desacoplar dependencias con árboles de código fuente, 319–320
- Antipatrón periférico de, 320–321 _____
- Estabilidad posicional, componente, 122–123 _____
- Decisiones prematuras, acoplamiento a, 160–163 _____
- “Capas de datos de dominio de presentación” (Fowler), 305–306 _____

Presentadores

- en arquitectura limpia, 203, 205 _____
- escenario de arquitectura limpia, 207–208 _____
- arquitectura de componentes, 301 _____
- cruzando límites circulares, 206 _____

Conclusión de presentadores y objetos

- humildes, 215 _____
- mapeadores de datos, 214–
- 215 escapadas de bases de datos, 214
- Patrón de objeto humilde, 212 _____
- descripción general de, 211–212
- Presentadores y vistas, 212–213 oyentes
- de servicios, 215 pruebas _____
- y arquitectura, 213 _____

Procesos, partición en clases/separación de clases, 71–72 _____

El procesador

- es detalle, 265–269 _____
- mutabilidad y, 52 _____

Producto, estudio de caso de ventas de videos, 298

Productividad

costo decreciente y creciente del código, 5–7 —
firma de un desastre, 8–9 —

Componentes abstractos de
lenguajes de programación, 125–126 —
componentes, 96 —
tipificados dinámicamente, 88 —
ISP y, 85 —
tipificados —
estáticamente, 87 variables en lenguajes funcionales, 51 —

Paradigmas de programación
programación funcional. Consulte Historia de la programación funcional de 19–20 —
programación orientada a objetos. Consulte Descripción general de la programación orientada a objetos, 21–24 programación estructurada. Consulte —

Programación estructurada
Disciplina de prueba de, 27–28 falta programación estructurada, 30–31 —
Proxies, uso con marcos, 293 Uso indebido —
de tipos
públicos, 315–316 —
versus tipos que se publican en módulos, 319 Python —
DIP y,
88 ISP y, 85 —

R

Condiciones de
carrera debido a variables —
mutables, 52 protección contra actualizaciones simultáneas y, 53 —

RAM
Proyecto de arqueología 4-TEL, 341, 343–344 —
reemplazo de discos, 280–281 —

Racional (empresa), 367, 368 —

RDBMS (sistemas de gestión de bases de datos relacionales), 279–283 —

El sistema operativo en tiempo real (RTOS) es detallado, 269–271
Sistemas de administración de bases de datos relacionales (RDBMS), 279–283
Bases de datos relacionales, 278, 281–283
Arquitectura en capas relajada, 311–312 *Libera el efecto*
del ciclo en el gráfico de dependencia de componentes, 115 –117
eliminación de ciclos de dependencia, 113–115
numeración de nuevos componentes, 113
Principio de equivalencia de reutilización/liberación para nuevos, 104–105
105 Terminales remotas, proyecto de arqueología DLU/DRU, 354–356 REP
(Principio de equivalencia de reutilización/liberación), 104–105, 108–110 Modelos
de solicitud, reglas comerciales, 193–194 ReSharper,
argumento del complemento, 172–173 Modelos
de respuesta, reglas comerciales, 193–194 REST,
dejar opciones abiertas en desarrollo, 141 Reutilizabilidad.
Ver CRP (Principio Común de Reutilización)
Principio de equivalencia de reutilización/liberación (REP), 104–105, 108–110 La
arquitectura de riesgos debe mitigar los costos de, 139–140 de
los marcos, 293–294
Placas ROM, proyecto de arqueología 4-TEL, 341
Producto ROSE, proyecto de arqueología, 368–370
RTOS (sistema operativo en tiempo real) es un detalle, 269–271

Componentes de Ruby como archivos de gemas, 96
inmersión y, 88
ISP y, 85
Herramienta RVM, gestión de módulos, 104

S
SAC (computadora de área de servicio), proyecto de arqueología
4-TEL usando, 340–341
arquitectura, 345–347
conclusión, 349
determinación de despacho, 345

Proyecto de arqueología DLU/DRU, 354–356
Europa, 348–349
gran rediseño, 347–348
descripción general
de, 344 SAP (principio de abstracciones estables) que evita zonas de exclusión,
130 distancia de la secuencia principal, 130–
132 trazar líneas límite, 173
introducción a, 126–127
secuencia principal, 127–130
medición de la abstracción, 127
dónde colocar la política de alto nivel, 126
SC (centro de servicios), proyecto de arqueología 4-TEL, 339–340
Problema y
servicios del kit de escalabilidad, 242–243
servicios no única opción para construir, 241
Schmidt, Doug, 256–258
Métodos científicos, prueba de afirmaciones falsas, 30–31
Alcance de la arquitectura cambiante, 15
Arquitectura gritadora. Consulte [Arquitectura, gritando](#)
componentes abstractos de SDP (principio de dependencias estables), 125–126
no todos los componentes deberían serlo, 124–
125 no todos los componentes deberían ser estables,
123–125 descripción general de, 120
estabilidad, 120–121 métricas
de estabilidad, 122–123 Principio de abstracciones estables, 127
Seguridad, prueba de API, 253 Selección, como estructura de control del programa, 27–28 Separación de componentes, como gran preocupación en la arquitectura, 24
Secuencia, como estructura de control del programa, 27–28 Bus de comunicación en serie, proyecto de arqueología SAC, 347 Computadora del área de servicio. Consulte [SAC \(computadora del área de servicio\)](#), proyecto de arqueología Centro de servicio (SC), proyecto de arqueología 4-TEL, 339–340 Modo de desacoplamiento de niveles

Servicios

basados en componentes, [245](#)—

246 conclusión, [247](#)

preocupaciones transversales, [246–247](#)

falacia de desacoplamiento, [240](#)—

241 como llamadas a funciones versus [_____](#)

arquitectura, [240](#) Límites de objetos humildes para,

214–215 falacia de desarrollo/implementación independiente, [_____](#)

[241 problema del gatito](#), [242](#)—

243 objetos al rescate, [244–245](#) [_____](#)

descripción general

de, [239](#) como límite más fuerte, [180](#)—

181 Establecer instrucción de interrupción del programa (SPI), proyecto de arqueología de fundición a presión de aluminio, [339](#)

Forma, de cambio, [15](#) [_____](#)

Principio de Responsabilidad Única. Ver [SRP \(Principio de Responsabilidad Única\)](#)

Modo de desacoplamiento SOA (arquitectura

orientada a servicios), [153](#)

en el proyecto de arqueología de recepcionista electrónica, [360–361](#)

razones de popularidad, [239](#) [Sockets](#),

los procesos locales se comunican a través de, [180](#) [Software](#)

La arquitectura integrada limpia aísla el sistema operativo de [270](#)

componentes. Consulte [Componentes](#)

que eliminan el cuello de botella del hardware de destino con capas, [262](#)—

263 línea difusa entre firmware y, [263–264](#) [hacerlo](#)

bien, 1–2 principios [_____](#)

SÓLIDOS, [58](#) valor de la

arquitectura frente al comportamiento, [14–18](#) [_____](#)

Desarrollo de software que

lucha por la arquitectura sobre la función, [18](#) como

una ciencia, [31](#) [_____](#)

Reutilización de software

Principio de reutilización común, [107–108](#)

componentes reutilizables y [104](#) [_____](#)

Principio de equivalencia de reutilización/liberación, [104–105](#)

Principios SÓLIDOS

Principio de inversión de dependencia. Consulte [DIP \(Principio de inversión de dependencia\)](#) que diseña servicios basados en componentes utilizando, 245–246 historia de, 57–59

Principio de segregación

de interfaz. Ver [ISP \(Principio de segregación de interfaz\)](#)

Principio de sustitución de Liskov. Ver [LSP \(Principio de sustitución de Liskov\)](#)

Enfoque OO para preocupaciones transversales, 244–245 Principio abierto-cerrado. Ver [OCP \(Principio Abierto-Cerrado\)](#)

Principio de Responsabilidad Única. Ver [SRP \(Principio de Responsabilidad Única\)](#)

Código fuente, compilación, 97–98

Dependencias del código fuente que

crean cruce de límites, 176 cruce de límites

circulares, 206 desacoplamiento, 184–185,

319 inversión de dependencia, 44–47

procesos locales como, 180

Ejemplo de OCP, 72 que

se refiere solo a abstracciones, 87–88

Los componentes de la interfaz de usuario reutilizan las reglas del juego vía, 222–223

Árboles de código fuente, dependencias de desacoplamiento, 319–321

Modo de desacoplamiento a nivel de fuente, 155–157, 176–178

Espeleología, la arquitectura mitiga los costos de, 139–140

Instrucción SPI (establecer interrupción del programa), proyecto de arqueología de fundición a presión de aluminio, 339

División de flujos de datos, 227–228

Problema de cuadrado/rectángulo, LSP, 79

Cuadrados de números enteros, programación funcional, 50–51

SRP (Principio de Responsabilidad Única)

ejemplo de duplicación accidental, 63–65 Principio de

cierre común versus, 106–107 conclusión, 66–67 capas de

desacoplamiento, 152

definidas, 59 gestión de

dependencias,

302 en una buena arquitectura de software,

71 políticas de agrupación en componentes,

186–187

mantener los cambios localizados, [118](#)
fusiones, [65](#)
descripción general de,
61–63 soluciones,
66–67 análisis de casos de
uso, [299](#) dónde trazar límites, [172–173](#)
Estabilidad, componente
midiendo, [122–123](#)
relación entre abstracción y, 127–130 SAP. Consulte [Comprensión de SAP \(principio de abstracciones estables\)](#), [120–121](#)
Principio de abstracciones estables. Ver [SAP \(Principio de abstracciones estables\)](#)
Los componentes
estables abstraen componentes como,
125–126 como inofensivos en Zone of Pain, [129](#) no todos los componentes deberían serlo, 123–125 colocando políticas de alto nivel en, [126](#) Principio de abstracciones estables, 126–127
Principio de dependencias estables. Ver [SDP \(Principio de Dependencias Estables\)](#)
Alcance versus
forma de las partes interesadas para el costo del cambio, [15](#) Antigüedad de la arquitectura sobre la función, [18](#) valores proporcionados por los sistemas de software, [14](#) Problemas de concurrencia estatal debido a la mutación, [53](#) almacenar transacciones pero no, [54–55](#) Herramientas de análisis estático, violaciones de la arquitectura, [313](#) Estático vs. .polimorfismo dinámico, [177](#) Patrón de estrategia que crea límites unidimensionales, [219](#) Enfoque OO para preocupaciones transversales, [244–245](#) Corrientes, arquitectura limpia de datos y, 224–226 cruce, [226](#) división, 227–228 Acoplamiento estructural, pruebas API, [252](#)

Estructura. Ver [Arquitectura](#)

Programación estructurada

Proclamación de Dijkstra sobre declaraciones goto, [28–29](#)
disciplina de la prueba, [27–28](#)
descomposición funcional en, [29](#)
historia de, [22](#)
falta de pruebas formales, [30](#)
descripción
general de, [26](#) papel de la
ciencia en, [30–31](#)
papel de las
pruebas en,
[31](#) valor de, [31–32](#) Sustitución LSP. Consulte
Programación LSP ([Principio de sustitución de](#)
[Liskov](#)) para interfaces y, [271–272](#) Subtipos, definición, [78](#)

t

Cuello de botella de hardware objetivo, [261](#), [262–272](#)
TAS (Sistemas aplicados Teradyne), [334–338](#), [339–344](#)
Patrón de método de plantilla, enfoque OO para inquietudes transversales, [244–245](#)

Conclusión de los

límites de la prueba,
[253](#) diseño para la capacidad de prueba, [251](#)
Problema de pruebas frágiles, [251](#)
descripción general de, [249–](#)
[250](#) pruebas de API, [252–](#)
[253](#) pruebas como componentes del sistema, [250](#)

Arquitectura comprobable

creación de arquitectura limpia, [202](#)
arquitectura integrada limpia como, [262–272](#)
descripción general de, [198](#)

Pruebas

y arquitectura, [213](#).
Presentadores y vistas, [212–213](#) en
programación estructurada, [31](#)

unidad. Consulte [Pruebas unitarias](#) a través del patrón de objeto [humilde](#),

[212 Mutabilidad de](#) [subprocesos](#) y [52 programación/orden de ejecución](#), [179 “arquitectura” de tres niveles \(como topología\)](#), [161 Diseño de arriba hacia abajo, estructura de componentes](#), [118–119 Memoria transaccional](#), [53 Transacciones, almacenamiento](#), [54–55 Dependencias transitivas, violación de los principios del software](#), [75 Tickets de problemas, proyecto de arqueología CDS](#), [362–364 Duplicación verdadera](#), [154–155 Turning, Alan](#), [23](#)

U

UI (interfaz de usuario). Véase también [GUI \(interfaz gráfica de usuario\)](#)

aplicar LSP a, [80](#)

arquitectura limpia independiente de, [202 cruzar límites circulares](#), [206 desacoplar reglas de negocio de](#), [287–289 desacoplar capas](#), [152–153 desacoplar casos de uso](#), [153](#)

Juego de aventuras Hunt the Wumpus, [222–223](#)

desarrollo independiente, [47, 154](#)

Principio de segregación de interfaz, [84](#)

programación para, [271–272](#)

reducción de la volatilidad de, [88](#)

Proyecto de arqueología SAC, [346](#)

Paquete de diagrama de

clases UML por capa, [304–305, 310](#)

puertos y adaptadores, [308–310](#)

arquitectura en capas relajada, [311–312](#)

Tío Bob, [367, 369](#)

Sistema de base de datos UNIFY, proyecto de arqueología VRS, [358–359](#)

Sistema de contabilidad sindical, proyecto de arqueología, [326–334](#)

Pruetas

unitarias que crean una arquitectura _____
comprobable, 198 efecto del ciclo en el gráfico de dependencia de _____
componentes, 116–117 a través del patrón
de objeto humilde, 212 UNIX, funciones del controlador
de dispositivo IO, 41–44 Actualizaciones, _____
riesgos de los marcos, 293 Urgencia, matriz de importancia de Eisenhower
vs. 16–17

Casos de uso que la arquitectura debe
soportar, 148 reglas de negocio _____
para, 191–194 escenario de arquitectura limpia,
207–208 acoplamiento a decisiones prematuras con, _____
160 creación de arquitectura comprobable, _____
198 cruce de límites circulares, 206 _____
desacoplamiento, _____
152 modo de desacoplamiento, _____
153 Regla de dependencia para, _____
204 duplicación de, 155
buena arquitectura centrada en, 196, 197 desarrollo
independiente y, 154 estudio de caso de ventas
de video, 298–300 Interfaz de usuario _____

GUI. Ver GUI (interfaz gráfica de usuario) _____

Interfaz de usuario. Ver UI (interfaz de usuario)

Biblioteca de servicios públicos, Zona del _____
Dolor, conexión Uucp 129 , 366

V

Valores, arquitectura (estructura)
del sistema de software, 14–15 _____
comportamiento, _____
14 Matriz de importancia versus urgencia de Eisenhower, 16–17 lucha _____
por la antigüedad de la arquitectura, 18 función _____
versus arquitectura, 15–16 descripción _____
general de, 14 _____

Variables, lenguaje funcional, 51. —

Minicomputadora Varian 620/f, proyecto de arqueología de Union Accounting, 331–334 — —

Arquitectura de componentes

- de estudio de caso de ventas de videos, —
- 300–302 conclusión, —
- 302 gestión de dependencias, 302 —
- sobre procesos/decisiones de un buen arquitecto, 297–298 —
- producto, 298 —
- análisis de casos de uso, 298–300 —

Ver modelo, presentadores y vistas, 213 —

Vistas

- de arquitectura de componentes, 301 —
- Presentadores y, 212–213 —

Vignette Grande, examen de registro de arquitectos, 371–372 —

Visual Studio, argumento del complemento, 172–173 —

Tecnologías de voz, proyectos de arqueología.

- Repcionista electrónica, 359–361 —
- Sistema de respuesta de voz, 357–359 —

Gráfico de dependencia

- de componentes volátiles y, 118 —
- diseño para capacidad de prueba,
- 251 colocación en software volátil, 126 —
- como problemático en Zone of Pain, 129 —
- Principio de dependencias estables y, 120 —

von Neumann, 34 años —

VRS (Sistema de respuesta de voz), proyectos de arqueología, 357–359, 362–363 — —

W.

Web

- como sistema de entrega de su aplicación, 197–198 — —
- La regla de dependencia para,
- 205 es detallada, 285–289 —

Servidores web

- que crean una arquitectura comprobable sin, 198 como
- opción para dejarlos abiertos, 141, 197 —

escribiendo propio, [163-165](#)

Construcción semanal, [112-113](#)

Texto wiki, caso de éxito arquitectónico, [164](#) —

Y

Yourdon, Ed, [29 años](#)

Z

Evitar zonas de

exclusión, [130](#)

relación entre abstracción/estabilidad, [128](#)

Zona de Dolor, [129](#)

Zona de inutilidad, [129-130](#) —



Register Your Product at informit.com/register
Access additional benefits and **save 35%** on your next purchase

- Automatically receive a coupon for 35% off your next purchase, valid for 30 days. Look for your code in your InformIT cart or the Manage Codes section of your account page.
- Download available product updates.
- Access bonus material if available.
- Check the box to hear from us and receive exclusive offers on new editions and related products.

InformIT.com—The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's foremost education company. At InformIT.com, you can:

- Shop our books, eBooks, software, and video training
- Take advantage of our special offers and promotions (informit.com/promotions)
- Sign up for special offers and content newsletter (informit.com/newsletters)
- Access thousands of free chapters and video lessons

Connect with InformIT—Visit informit.com/community



Addison-Wesley • Adobe Press • Cisco Press • Microsoft Press • Pearson IT Certification • Prentice Hall • Que • Sams • Peachpit Press



Fragmentos de código

point.h

```
struct Point;
struct Point* makePoint(double x, double y);
double distance (struct Point *p1, struct Point *p2);
```

point.c

```
#include "point.h"
#include <stdlib.h>
#include <math.h>

struct Point {
    double x,y;
};

struct Point* makepoint(double x, double y) {
    struct Point* p = malloc(sizeof(struct Point));
    p->x = x;
    p->y = y;
    return p;
}

double distance(struct Point* p1, struct Point* p2) {
    double dx = p1->x - p2->x;
    double dy = p1->y - p2->y;
    return sqrt(dx*dx+dy*dy);
}
```

point.h

```
class Point {  
public:  
    Point(double x, double y);  
    double distance(const Point& p) const;  
  
private:  
    double x;  
    double y;  
};
```

point.cc

```
#include "point.h"
#include <math.h>

Point::Point(double x, double y)
: x(x), y(y)
{ }

double Point::distance(const Point& p) const {
    double dx = x-p.x;
    double dy = y-p.y;
    return sqrt(dx*dx + dy*dy);
}
```

namedPoint.h

```
struct NamedPoint;  
  
struct NamedPoint* makeNamedPoint(double x, double y, char* name);  
void setName(struct NamedPoint* np, char* name);  
char* getName(struct NamedPoint* np);
```

namedPoint.c

```
#include "namedPoint.h"
#include <stdlib.h>

struct NamedPoint {
    double x,y;
    char* name;
};

struct NamedPoint* makeNamedPoint(double x, double y, char* name) {
    struct NamedPoint* p = malloc(sizeof(struct NamedPoint));
    p->x = x;
    p->y = y;
    p->name = name;
    return p;
}

void setName(struct NamedPoint* np, char* name) {
    np->name = name;
}

char* getName(struct NamedPoint* np) {
    return np->name;
}
```

main.c

```
#include "point.h"
#include "namedPoint.h"
#include <stdio.h>

int main(int ac, char** av) {
    struct NamedPoint* origin = makeNamedPoint(0.0, 0.0, "origin");
    struct NamedPoint* upperRight = makeNamedPoint
        (1.0, 1.0, "upperRight");
    printf("distance=%f\n",
        distance(
            (struct Point*) origin,
            (struct Point*) upperRight));
}
```

```
#include <stdio.h>

void copy() {
    int c;
    while ((c=getchar()) != EOF)
        putchar(c);
}
```

```
struct FILE {
    void (*open)(char* name, int mode);
    void (*close)();
    int (*read)();
    void (*write)(char);
    void (*seek)(long index, int mode);
};
```

```
#include "file.h"

void open(char* name, int mode) {/*...*/}
void close() {/*...*/};
int read() {int c; /*...*/ return c;};
void write(char c) {/*...*/}
void seek(long index, int mode) {/*...*/}

struct FILE console = {open, close, read, write, seek};
```

```
extern struct FILE* STDIN;

int getchar() {
    return STDIN->read();
}
```

Fragmentos de código

```
public class Squint {  
    public static void main(String args[]) {  
        for (int i=0; i<25; i++)  
            System.out.println(i*i);  
    }  
}
```

```
(println (take 25 (map (fn [x] (* x x)) (range))))
```

```
(println ;_____ Print
(take 25 ;_____ the first 25
(map (fn [x] (* x x)) ;__ squares
(range)))) ;_____ of Integers
```

```
(def counter (atom 0)) ; initialize counter to 0
(swap! counter inc)    ; safely increment counter.
```

Fragmentos de código

```
Rectangle r = ...  
r.setW(5);  
r.setH(2);  
assert(r.area() == 10);
```

purplecab.com/driver/Bob

purplecab.com/driver/Bob
/pickupAddress/24 Maple St.
/pickupTime/153
/destination/ORD

```
if (driver.getDispatchUri().startsWith("acme.com")) ...
```

URI	Dispatch Format
Acme.com	/pickupAddress/%s/pickupTime/%s/dest/%s
.	/pickupAddress/%s/pickupTime/%s/destination/%s

Fragmentos de código

	* 200
	TLS
START,	CLA
	TAD BUFR
	JMS GETSTR
	CLA
	TAD BUFR
	JMS PUTSTR
	JMP START
BUFR,	3000
GETSTR,	O
	DCA PTR
NXTCH,	KSF
	JMP -1
	KRB
	DCA I PTR
	TAD I PTR
	AND K177
	ISZ PTR
	TAD MCR
	SZA
	JMP NXTCH
K177,	177
MCR,	-15

Fragmentos de código

PRTCHR, 0

TSF

JMP . -1

TLS

JMP I PRTCHR

Fragmentos de código

```
function encrypt() {  
    while(true)  
        writeChar(translate(readChar()));  
}
```

Fragmentos de código

```
public class Main implements HtwMessageReceiver {  
    private static HuntTheWumpus game;  
    private static int hitPoints = 10;  
    private static final List<String> caverns = new  
    ArrayList<>();  
    private static final String[] environments = new String[] {  
        "bright",  
        "humid",  
        "dry",  
        "creepy",  
        "ugly",  
        "foggy",  
        "hot",  
    };
```

```
    "cold",
    "drafty",
    "dreadful"
};

private static final String[] shapes = new String[] {
    "round",
    "square",
    "oval",
    "irregular",
    "long",
    "craggy",
    "rough",
    "tall",
    "narrow"
};
```

```
private static final String[] cavernTypes = new String[] {  
    "cavern",  
    "room",  
    "chamber",  
    "catacomb",  
    "crevasse",  
    "cell",  
    "tunnel",  
    "passageway",  
    "hall",  
    "expanse"  
};  
  
private static final String[] adornments = new String[] {
```

"smelling of sulfur",
"with engravings on the walls",
"with a bumpy floor",
"
",
"littered with garbage",
"spattered with guano",
"with piles of Wumpus droppings",
"with bones scattered around",
"with a corpse on the floor",
"that seems to vibrate",
"that feels stuffy",
"that fills you with dread"
};

```
public static void main(String[] args) throws IOException {
    game = HtwFactory.makeGame("htw.game.HuntTheWumpusFacade",
                               new Main());
    createMap();
    BufferedReader br =
        new BufferedReader(new InputStreamReader(System.in));
    game.makeRestCommand().execute();
    while (true) {
        System.out.println(game.getPlayerCavern());
        System.out.println("Health: " + hitPoints + " arrows: " +
                           game.getQuiver());
        HuntTheWumpus.Command c = game.makeRestCommand();
```

Machine Translated by Google

```
System.out.println(">");

String command = br.readLine();
if (command.equalsIgnoreCase("e"))
    c = game.makeMoveCommand(EAST);
else if (command.equalsIgnoreCase("w"))
    c = game.makeMoveCommand(WEST);
else if (command.equalsIgnoreCase("n"))
    c = game.makeMoveCommand(NORTH);
else if (command.equalsIgnoreCase("s"))
    c = game.makeMoveCommand(SOUTH);
else if (command.equalsIgnoreCase("r"))
    c = game.makeRestCommand();
else if (command.equalsIgnoreCase("sw"))
    c = game.makeShootCommand(WEST);
else if (command.equalsIgnoreCase("se"))
    c = game.makeShootCommand(EAST);
else if (command.equalsIgnoreCase("sn"))
    c = game.makeShootCommand(NORTH);
else if (command.equalsIgnoreCase("ss"))
    c = game.makeShootCommand(SOUTH);
else if (command.equalsIgnoreCase("q"))
    return;

c.execute();

}
```

Machine Translated by Google

```
private static void createMap() {  
    int nCaverns = (int) (Math.random() * 30.0 + 10.0);  
    while (nCaverns-- > 0)  
        caverns.add(makeName());  
  
    for (String cavern : caverns) {  
        maybeConnectCavern(cavern, NORTH);  
        maybeConnectCavern(cavern, SOUTH);  
        maybeConnectCavern(cavern, EAST);  
        maybeConnectCavern(cavern, WEST);  
    }  
  
    String playerCavern = anyCavern();  
    game.setPlayerCavern(playerCavern);  
    game.setWumpusCavern(anyOther(playerCavern));  
    game.addBatCavern(anyOther(playerCavern));  
    game.addBatCavern(anyOther(playerCavern));  
    game.addBatCavern(anyOther(playerCavern));  
  
    game.addPitCavern(anyOther(playerCavern));  
    game.addPitCavern(anyOther(playerCavern));  
    game.addPitCavern(anyOther(playerCavern));  
  
    game.setQuiver(5);  
}  
  
// much code removed...  
}
```

Fragmentos de código

```
ISR(TIMER1_vect) { ... }

ISR(INT2_vect) { ... }

void btn_Handler(void) { ... }

float calc_RPM(void) { ... }

static char Read_RawData(void) { ... }

void Do_Average(void) { ... }

void Get_Next_Measurement(void) { ... }

void Zero_Sensor_1(void) { ... }

void Zero_Sensor_2(void) { ... }

void Dev_Control(char Activation) { ... }

char Load_FLASH_Setup(void) { ... }

void Save_FLASH_Setup(void) { ... }

void Store_DataSet(void) { ... }

float bytes2float(char bytes[4]) { ... }

void Recall_DataSet(void) { ... }

void Sensor_init(void) { ... }

void uC_Sleep(void) { ... }
```

```
float calc_RPM(void) { ... }

void Do_Average(void) { ... }

void Get_Next_Measurement(void) { ... }

void Zero_Sensor_1(void) { ... }

void Zero_Sensor_2(void) { ... }
```

```
ISR(TIMER1_vect) { ... }*
ISR(INT2_vect) { ... }
void uC_Sleep(void) { ... }

Functions that react to the on off button press
void btn_Handler(void) { ... }
void Dev_Control(char Activation) { ... }

A Function that can get A/D input readings from the
hardware

static char Read_RawData(void) { ... }
```

```
char Load_FLASH_Setup(void) { ... }
void Save_FLASH_Setup(void) { ... }
void Store_DataSet(void) { ... }
float bytes2float(char bytes[4]) { ... }
void Recall_DataSet(void) { ... }
```

```
void Sensor_init(void) { . . . }
```

```
#ifndef __ACME_STD_TYPES  
#define __ACME_STD_TYPES
```

```
#if defined(_ACME_X42)
    typedef unsigned int          Uint_32;
    typedef unsigned short        Uint_16;
    typedef unsigned char         Uint_8;

    typedef int                  Int_32;
    typedef short                Int_16;
    typedef char                 Int_8;

#elif defined(_ACME_A42)
    typedef unsigned long         Uint_32;
    typedef unsigned int          Uint_16;
    typedef unsigned char         Uint_8;

    typedef long                 Int_32;
    typedef int                  Int_16;
    typedef char                 Int_8;

#else
    #error <acmetypes.h> is not supported for this environment
#endif

#endif
```



```
#ifndef _STDINT_H_
#define _STDINT_H_

#include <acmetypes.h>

typedef Uint_32 uint32_t;
typedef Uint_16 uint16_t;
typedef Uint_8  uint8_t;

typedef Int_32   int32_t;
typedef Int_16   int16_t;
typedef Int_8    int8_t;

#endif
```



```
void say_hi()
{
    IE = 0b11000000;
    SBUFO = (0x68);
    while(TI_O == 0);
    TI_O = 0;
    SBUFO = (0x69);
    while(TI_O == 0);
    TI_O = 0;
    SBUFO = (0x0a);
    while(TI_O == 0);
    TI_O = 0;
    SBUFO = (0x0d);
    while(TI_O == 0);
    TI_O = 0;
    IE = 0b11010000;
}
```

Fragmentos de código

/pno 8475551212 /noise /dropped-calls