

BE SOFTWARE ARCHITECTURE WISE · CAPÍTULO 3

Principios y fundamentos de diseño

Toda arquitectura se construye a partir de las mismas pocas ideas sobre la estructura: cómo dividir un sistema en partes, cuánto deben saber esas partes unas de otras, en qué dirección deben apuntar sus dependencias y cómo mantener el conjunto tan simple como permita el problema. Este capítulo enseña esas ideas a todas las escalas, desde una clase de Python hasta un componente y una flota de servicios: modularidad, acoplamiento y cohesión, ocultación de información, gestión de dependencias, SOLID, connascence, la ley de Conway, la simplicidad y las métricas que miden la estructura. Está pensado para desarrolladores que quieren razonar sobre el diseño con precisión y para arquitectos que necesitan un vocabulario común para las revisiones.

bewiseapps.com/es/software-architecture-wise/design-principles/ · Versión 2026-11-03. Solo información educativa, no asesoramiento profesional.

DE UN VISTAZO

Aprenderás a

- Explicar el criterio de Parnas para descomponer un sistema en módulos y aplicarlo para comparar dos descomposiciones candidatas.
- Clasificar las dependencias con los niveles clásicos de acoplamiento y cohesión, e identificar el acoplamiento temporal, de despliegue y semántico entre servicios.
- Distinguir la ocultación de información del encapsulamiento, especificar una interfaz como un contrato y diseñar API estables que tengan en cuenta las abstracciones con fugas y la ley de Hyrum.
- Calcular la inestabilidad, el grado de abstracción y la distancia a la secuencia principal, y usarlos junto con los principios de dependencias acíclicas, de dependencias estables y de abstracciones estables para corregir estructuras de dependencias.
- Aplicar los principios SOLID y los principios de cohesión de componentes a escala de clase, de componente y de servicio, y reconocer cuándo se están aplicando de forma dogmática.
- Usar la connascence para describir la fuerza, la localidad y el grado del acoplamiento, y elegir refactorizaciones que lo debiliten.
- Explicar la ley de Conway, la evidencia que la respalda y la maniobra inversa de Conway, y calcular las vías de comunicación de un equipo.
- Separar la complejidad esencial de la accidental, calcular la complejidad ciclomática y convertir reglas estructurales en funciones de aptitud sin caer en la ley de Goodhart.

Antes de empezar

Capítulos 0 a 2, en especial la Sección 0.3 (organizar el código), la Sección 1.7 (deuda técnica) y la Sección 2.6 (modificabilidad). Ayuda saber leer ejemplos breves de Python.

Tiempo

Unas 3 horas de lectura; 45 minutos para la prueba.

Términos clave

modularidad · ocultación de información · acoplamiento · cohesión · acoplamiento aferente y eferente · abstracción · diseño por contrato · Ley de Hyrum · Principio de dependencias acíclicas · inestabilidad y grado de abstracción · SOLID · connascence · Ley de Conway · complejidad accidental

Por qué principios, y por qué a todas las escalas

El Capítulo 2 describió las cualidades que debe ofrecer una arquitectura. La mayoría de ellas, y la modificabilidad por encima de todas, dependen menos de una tecnología concreta que de la estructura: cómo se divide un sistema en partes, qué sabe cada parte de las demás y en qué dirección van las dependencias entre ellas. Un equipo puede elegir el framework más de moda y la mejor plataforma en la nube y aun así acabar con un sistema en el que cada cambio toca

diez sitios, porque las partes se cortaron por los lugares equivocados. Los principios de este capítulo son el conocimiento acumulado durante más de cincuenta años sobre cómo cortarlas bien.

Son ideas antiguas, y eso es una fortaleza. La ocultación de información data de 1972; el acoplamiento y la cohesión, del trabajo sobre diseño estructurado de finales de la década de 1960 y de la de 1970; la ley de Conway, de 1968. Han sobrevivido porque describen algo que no cambia: el costo de un cambio depende de lo lejos que tenga que llegar ese cambio. Lo que ha cambiado es la escala a la que los aplican los arquitectos. El mismo principio que te dice que mantengas privados los campos de una clase te dice que mantengas privada la base de datos de un servicio; el mismo razonamiento que prohíbe un ciclo de dependencias entre dos paquetes prohíbe dos servicios que solo se pueden lanzar juntos. Por eso cada sección muestra cada idea dos veces: en un pequeño fragmento de código y a la escala de los componentes y los servicios.

Cómo está organizado el capítulo

Las cuatro primeras secciones sientan las bases. La Sección 3.1 se pregunta qué es un módulo y dónde deben ir sus límites, usando el ejemplo KWIC de David Parnas para mostrar por qué falla descomponer por pasos de procesamiento y por qué funciona descomponer por cambios probables, y separa los módulos lógicos de las unidades físicas que se compilan y se despliegan. La Sección 3.2 ofrece el vocabulario para juzgar los límites: los niveles clásicos de acoplamiento y cohesión, el acoplamiento aferente y eferente, y los tipos de acoplamiento que solo aparecen entre servicios. La Sección 3.3 examina la propia interfaz: ocultación de información frente a encapsulamiento, contratos, abstracciones con fugas, módulos profundos y la ley de Hyrum. La Sección 3.4 pasa a la dirección: los principios de Robert C. Martin para las dependencias entre componentes, las métricas de inestabilidad y de grado de abstracción, cómo romper ciclos, la organización en capas y cómo comprobar todo esto automáticamente.

Las cuatro secciones siguientes se apoyan en ellas. La Sección 3.5 trata los cinco principios SOLID y los principios de cohesión de componentes, cada uno mostrado tanto en código como entre servicios, junto con las críticas que reciben cuando se aplican de forma mecánica. La Sección 3.6 presenta la connascence, una medida más fina del acoplamiento con un conjunto claro de reglas para hacerlo más débil y más local. La Sección 3.7 amplía la mirada a la organización: la ley de Conway, la evidencia que la respalda, la aritmética de las vías de comunicación y la maniobra inversa de Conway. La Sección 3.8 cierra con la simplicidad y la medición: la complejidad esencial y la complejidad accidental, KISS, YAGNI y DRY bien entendidos, el precio de mantener opciones abiertas, métricas estructurales como la complejidad ciclomática y los límites de todas las métricas.

Una idea por debajo de todos ellos

Leídas juntas, las ocho secciones forman un solo argumento. El cambio es el costo dominante del software, así que una buena estructura es aquella en la que los cambios probables se mantienen locales. Todos los principios del capítulo son formas de conseguirlo. Parnas dice que hay que poner cada cambio probable dentro de un módulo. El acoplamiento y la cohesión dicen que hay que mantener juntas las cosas que cambian juntas y separadas las que cambian por separado. La ocultación de información dice que hay que asegurarse de que lo que puede cambiar sea invisible desde fuera. Los principios de dependencias dicen que hay que dirigir las dependencias hacia lo que menos cambia. La connascence dice que hay que mantener cualquier acuerdo fuerte entre dos fragmentos de código lo más cerca posible. La ley de Conway añade que las personas forman parte de la estructura: un cambio que necesita que dos equipos se pongan de acuerdo no es local, se organice como se organice el código.

Ese argumento también explica por qué los principios a veces entran en conflicto y por qué ninguno es absoluto. Eliminar la duplicación puede aumentar el acoplamiento; añadir interfaces para invertir una dependencia puede hacer que un módulo sea menos profundo; dividir un componente para mejorar la cohesión puede crear un salto de red. La habilidad profesional no consiste en aplicar todos los principios en todas partes, sino en preguntarse, para un cambio concreto que es probable en un sistema concreto, hasta dónde tendría que llegar ese cambio y qué principio acorta esa distancia al menor costo.

Cómo usar este capítulo

Quienes son nuevos en el diseño pueden leer las secciones en orden; cada una empieza con una breve explicación cotidiana y remite a la Sección 0.3, donde se presentaron por primera vez los conceptos básicos. Los ingenieros con experiencia quizá prefieran empezar por los ejemplos prácticos: la descomposición de la tienda en línea de la Sección 3.1, la refactorización de pedidos y facturación de la Sección 3.2, la interfaz de pagos de la Sección 3.3 y el cálculo de métricas de la Sección 3.4 están pensados para debatirlos. Los principios de este capítulo se usan en el resto del curso: los estilos arquitectónicos de los Capítulos 5 y 6 son, en gran medida, respuestas distintas a las preguntas de dónde poner los límites y hacia dónde apuntan las dependencias, y el Capítulo 11 convierte las comprobaciones automáticas presentadas aquí en una práctica de gobernanza mediante funciones de aptitud.

SECCIÓN 3.1 DE 3.8 · INTERMEDIO · 28 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Modularidad: dividir un sistema en partes

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.3 Organizar el código
- 2.6 Modificabilidad y desplegabilidad

Términos anteriores que utiliza: Dependencia, Biblioteca, Monolito, Erosión de la arquitectura, Función de aptitud (fitness function)

LA IDEA EN BREVE

Piensa en un hospital grande. No es una sala enorme en la que todo el mundo hace de todo; está dividido en departamentos: radiología, farmacia, maternidad. Cada uno tiene su propio personal y su propia forma de trabajar, y los demás se relacionan con él a través de un mostrador con formularios de solicitud claros. Si la farmacia cambia su sistema de existencias, las plantas no lo notan, porque siguen rellenando el mismo formulario. El software se divide de la misma manera, en módulos. El arte está en elegir dónde van las paredes, para que los cambios que esperas se queden dentro de un solo departamento.

LO QUE NECESITAS SABER

- Un módulo es una unidad de descomposición con una responsabilidad y una interfaz: una función, una clase, un paquete, un componente o un servicio, según la escala a la que estés trabajando.
- La modularidad rinde frutos gracias a la comprensión, el trabajo en paralelo de los equipos, el aislamiento del cambio, la reutilización y las pruebas, pero solo si los límites se trazan en los lugares adecuados.
- David Parnas (1972) demostró con el ejemplo KWIC que descomponer por pasos de procesamiento reparte cada decisión de diseño entre muchos módulos, y propuso en su lugar que cada módulo oculte una decisión que sea difícil o que probablemente vaya a cambiar.
- Los módulos lógicos (unidades de responsabilidad en el diseño) son distintos de los módulos físicos (unidades de compilación y de despliegue); el mismo diseño lógico se puede desplegar como una sola unidad o como muchos servicios.

- La granularidad es un compromiso: los módulos demasiado grandes mezclan cambios que no tienen relación entre sí, mientras que los módulos demasiado pequeños multiplican las interfaces, la coordinación y, a través de una red, la latencia y los modos de fallo.
- Los límites que no se hacen cumplir se erosionan, así que los profesionales usan la visibilidad del lenguaje, sistemas de módulos como el Java Platform Module System, los módulos de las herramientas de compilación y comprobaciones automáticas de dependencias.

Qué es un módulo

Un módulo es una parte de un sistema que tiene una responsabilidad clara y una forma definida en que las demás partes pueden usarla. Esa definición es deliberadamente independiente del tamaño. En la Sección 0.3 un módulo era un archivo de Python; en este capítulo la palabra abarca cualquier unidad de descomposición, el acto de dividir un sistema en partes: una función, una clase, un paquete, una biblioteca, un componente dentro de una aplicación o un servicio desplegado por separado. Las mismas preguntas valen a todas las escalas: ¿de qué es responsable esta parte, qué ofrece a las demás, qué se guarda para sí y de qué depende?

Todo módulo tiene dos caras. Su interfaz es lo que otros módulos deben saber para usarlo: nombres y parámetros de funciones, una API HTTP, los eventos que publica y también las promesas informales, como lo rápido que responde y qué errores puede devolver. Su implementación es todo lo demás: el código, las estructuras de datos, las tablas de la base de datos, las bibliotecas de terceros que use. Una buena modularidad mantiene la interfaz pequeña y estable y deja que la implementación cambie libremente.

David Parnas, cuyo artículo de 1972 es la base de esta sección, definió un módulo como una *asignación de responsabilidad* y no como un fragmento de texto de programa. Esa es la forma más útil de pensar en él como arquitecto. Un módulo es la promesa de que una parte del sistema, y normalmente un equipo, es responsable de un conjunto de decisiones. Que el código acabe en un archivo o en cuarenta, en un proceso o en diez, es otra cuestión, que se trata más adelante en el apartado sobre módulos lógicos y físicos.

SI ESTO ES NUEVO PARA TI

Si esto es nuevo para ti

Imagina una app de reparto de comida. En algún lugar de su código hay lógica para mostrar cartas, para cobrar los pagos, para seguir a los repartidores y para enviar notificaciones. La modularidad significa simplemente que cada una de esas tareas vive en su propia parte claramente separada, con un conjunto pequeño y acordado de puntos de entrada. El resto de esta sección trata de dónde poner las paredes y de cómo impedir que la gente se las salte.

Por qué modularizar

Un sistema de unos pocos cientos de líneas se puede entender en su conjunto. Un sistema de unos pocos millones de líneas no, y tampoco la organización que lo construye puede trabajar como un único grupo indiviso. La modularidad es la propiedad de estar dividido en módulos que se pueden entender, cambiar y analizar en gran medida por separado. Es una de las cinco subcaracterísticas de la mantenibilidad en ISO/IEC 25010 (Sección 2.6) y aporta al menos cinco beneficios distintos.

Comprensión. Un desarrollador puede estudiar un módulo y sus interfaces sin leer el resto del sistema. **Trabajo en paralelo.** Los equipos pueden trabajar en módulos distintos al mismo tiempo sin apenas necesidad de coordinarse, porque han acordado las interfaces de antemano. **Aislamiento del cambio.** Un cambio en una decisión de diseño, como qué normas fiscales se aplican, se hace en un módulo y no se propaga hacia fuera. **Reutilización.** Un módulo bien delimitado, como un componente de validación de direcciones, puede usarse en varios productos. **Pruebas.** Un módulo con una interfaz clara puede probarse por sí solo, sustituyendo sus dependencias por dobles de prueba.

Parnas enumeró tres de ellos en 1972: el de gestión (el tiempo de desarrollo se acorta porque grupos separados trabajan en módulos separados con poca comunicación), la flexibilidad (cambios drásticos en un módulo sin cambiar los demás) y la comprensibilidad (estudiar el sistema módulo a módulo). Medio siglo después, siguen siendo las razones. Pero fíjate en que ninguno de ellos se deriva automáticamente de cortar un sistema en trozos. Un sistema puede estar dividido en cincuenta módulos y no ofrecer ninguno de estos beneficios, si cada cambio toca diez de ellos. Los beneficios dependen por completo de *dónde* se tracen los límites.

Separación de responsabilidades

Edsger Dijkstra le dio nombre a esta idea en un breve ensayo de 1974, "On the role of scientific thought" (EWD 447). Describía una forma de pensar antes que una forma de construir: estudias a fondo un aspecto de un problema, como si un programa es correcto, sabiendo que existen otros, como si es eficiente o si alguien lo quiere. Lo llamó tener "una mente de una y de múltiples vías a la vez", y admitió que la separación nunca es perfecta. Una *inquietud* es simplemente algo en lo que necesitas pensar.

En el diseño de software, la **separación de responsabilidades** significa organizar un sistema de modo que cada parte se ocupe de una inquietud y pueda entenderse y cambiarse sin tener presentes las demás. En la plataforma de reparto de comida, calcular el precio de un pedido, estimar su hora de entrega, cobrar el pago y enviar notificaciones son inquietudes separadas. Si están enredadas en una sola función larga, un cambio en el texto de una notificación obliga a leer las reglas de precios, y una prueba de precios tiene que preparar un pago. Separadas, cada una puede analizarse, probarse y sustituirse por sí sola. El mismo razonamiento está detrás de muchas estructuras conocidas: HTML para el contenido y CSS para la presentación, modelo y vista, una regla de negocio separada de la base de datos que almacena su resultado.

Es el más amplio de los principios de este capítulo, y los demás lo afinan. La modularidad aporta las unidades. La ocultación de información (Sección 3.3) dice dónde trazar la línea: alrededor de las decisiones que probablemente cambien. El principio de responsabilidad única (Sección 3.5) dice a qué necesidades debe servir una parte.

Hay dos advertencias. Primera: algunas inquietudes se niegan a estar en un solo sitio. El logging, la autorización y los reintentos afectan a casi todo, por lo que se llaman **inquietudes transversales (cross-cutting concerns)**; suelen resolverse con middleware compartido, un gateway o un service mesh (malla de servicios) (Sección 8.5), para que el código de negocio no las repita. Segunda: una separación equivocada puede empeorar las cosas. Dividir estrictamente en capas técnicas separa la interfaz de usuario de los datos, pero dispersa una inquietud de negocio, como dar propina a un repartidor, por todas las capas, de modo que una sola funcionalidad las toca todas. Por eso la Sección 5.3 organiza los módulos en torno a capacidades de negocio. Pregúntate qué inquietudes cambian de forma independiente y separa esas.

Parnas y el ejemplo KWIC

Parnas lo demostró con un pequeño programa llamado KWIC (key word in context, palabra clave en contexto), que genera un índice de títulos. Lee líneas de texto, genera todas las rotaciones circulares de cada línea (pasando la primera palabra al final, una y otra vez), ordena

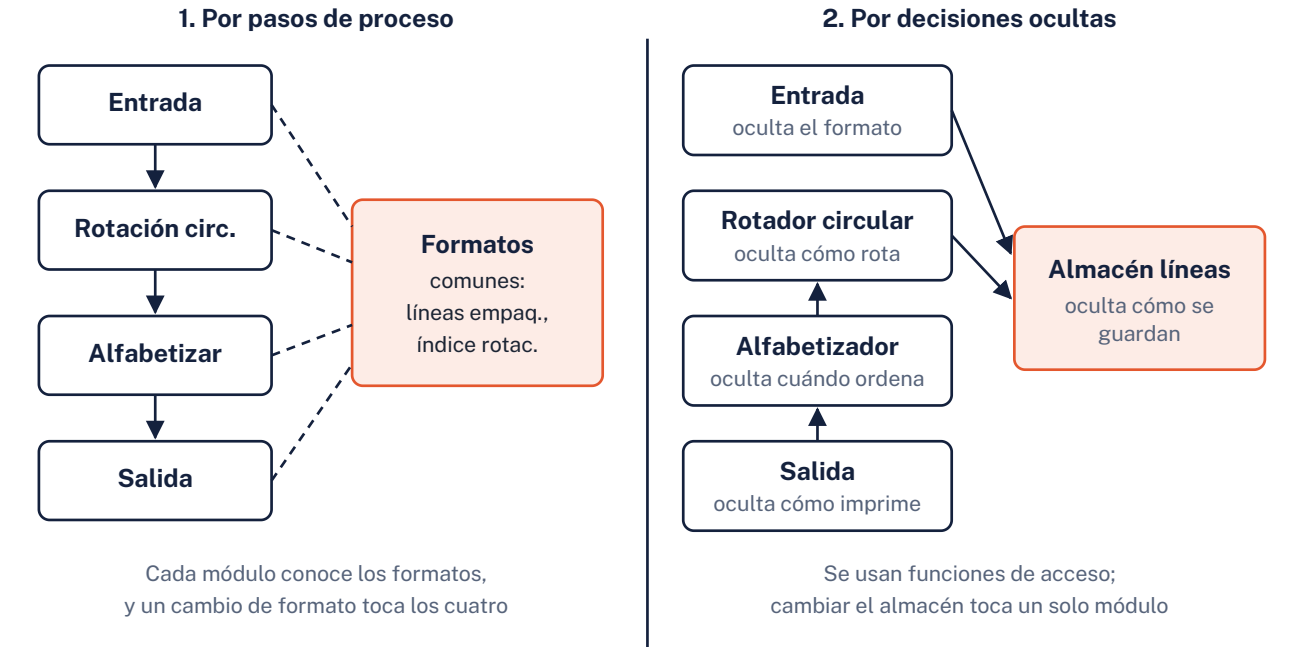
alfabéticamente todas las rotaciones y las imprime. La línea `Design of Software` produce `Design of Software`, `of Software Design` y `Software Design of`, de modo que un lector puede encontrar el título por cualquiera de sus palabras.

La primera descomposición que describe es la que produciría la mayoría de los programadores dibujando un diagrama de flujo: un módulo de entrada (Input) que lee las líneas y las guarda en memoria en un formato compactado, un módulo de rotación circular (Circular Shift) que construye un índice de dónde empieza cada rotación, un módulo de alfabetización (Alphabetising) que ordena ese índice, un módulo de salida (Output) que imprime el resultado y un módulo de control maestro que los llama en orden. Es un diseño perfectamente razonable. Cada módulo es pequeño y las interfaces entre ellos son los formatos de datos en memoria.

La segunda descomposición parece más extraña. Tiene un almacén de líneas (Line Storage) que ofrece funciones como *dame el carácter de la línea r, palabra w, posición c* y oculta cómo se guardan realmente las líneas; un módulo de entrada que lee líneas y se las entrega al almacén de líneas; un rotador circular (Circular Shifter) que presenta las rotaciones *como si* estuvieran guardadas, mediante funciones de acceso similares; un alfabetizador (Alphabetiser) que ofrece *dame la i-ésima rotación en orden* sin decir cuándo ni cómo ordena; un módulo de salida; y un control maestro. Los módulos ya no corresponden a los pasos del procesamiento.

Después, Parnas enumeró decisiones que eran discutibles y que probablemente cambiarían: el formato de entrada; mantener todas las líneas en memoria, lo que no es práctico con entradas muy grandes; compactar varios caracteres en cada palabra de máquina; guardar un índice de las rotaciones en lugar de las propias rotaciones; y ordenarlo todo de una vez en lugar de buscar bajo demanda. En la primera descomposición, la mayoría de estos cambios afectan a varios módulos, porque todos los módulos conocen los formatos de datos compartidos. En la segunda, cada cambio queda confinado al único módulo que oculta esa decisión. El segundo diseño también era más fácil de desarrollar en paralelo, porque los equipos solo tenían que acordar nombres y parámetros de funciones, no estructuras de tablas detalladas, y más fácil de entender módulo a módulo.

Las dos descomposiciones de KWIC de Parnas



El control maestro (no mostrado) llama a los módulos en orden en ambos diseños

En la descomposición del diagrama de flujo todos los módulos comparten los formatos de datos; en la descomposición por ocultación de información cada módulo oculta una decisión tras funciones de acceso.

El criterio: descomponer por decisiones que probablemente cambien

La conclusión de Parnas fue que es “casi siempre incorrecto” empezar a descomponer un sistema a partir de un diagrama de flujo. En su lugar, propuso partir de una lista de decisiones de diseño difíciles, o de decisiones de diseño que probablemente vayan a cambiar, y diseñar cada módulo para que oculte una de esas decisiones a los demás. Llamó a este principio ocultación de información; la Sección 3.3 lo trata en profundidad.

La idea parece sencilla, pero invierte el instinto que tenemos la mayoría. Un diagrama de flujo describe el *tiempo*: primero ocurre esto, después aquello. Las decisiones de diseño atraviesan el tiempo. La decisión sobre cómo se representa un pedido se usa cuando el pedido se recibe, se tarifica, se paga, se guarda y se confirma. Si los módulos siguen los pasos, esa única decisión se reparte entre todos ellos, y cambiarla significa cambiarlos todos. Si un módulo es propietario de la decisión, todos los pasos le preguntan a ese módulo, y la decisión puede cambiar en un solo lugar.

En la práctica importan dos matices. Primero, el criterio se refiere al cambio *probable*, lo que significa que un arquitecto tiene que hacer una suposición informada sobre el futuro: qué ha cambiado en sistemas similares, qué planea el negocio, qué proveedores y qué normativas son inestables. Las cuatro preguntas de modificabilidad de la Sección 2.6 (qué cambia, con qué

probabilidad, quién lo cambia y a qué costo) son exactamente la información que esto necesita. Segundo, Parnas distinguió entre una buena descomposición y una jerarquía. Un sistema puede estar limpiamente organizado en capas y aun así mal descompuesto, y al revés; la Sección 3.4 trata por separado la dirección de las dependencias.

DE DÓNDE VIENE

De dónde viene la idea

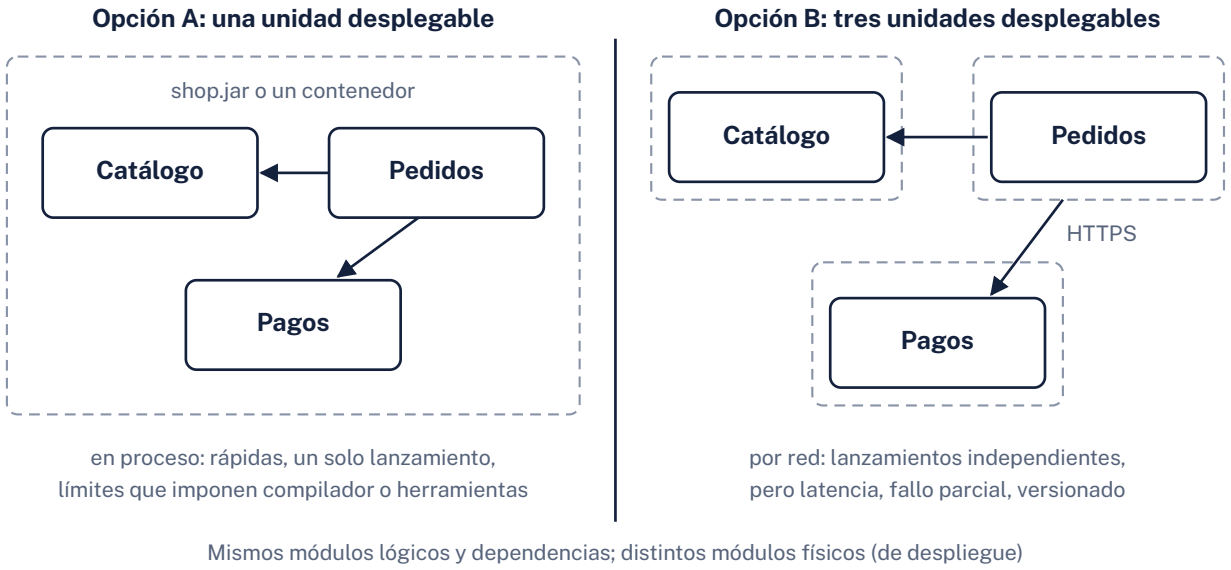
El artículo de Parnas, *On the Criteria To Be Used in Decomposing Systems into Modules*, apareció en *Communications of the ACM* en diciembre de 1972 (volumen 15, número 12). En aquella época se recomendaba ampliamente la programación modular, pero había pocas indicaciones sobre cómo elegir los módulos. El artículo tiene solo seis páginas, se lee muy bien y sigue siendo uno de los más citados de la ingeniería de software. Parnas incluso señaló un defecto en su propio segundo diseño: la interfaz del rotador circular fijaba el orden de las rotaciones, con lo que revelaba más de lo necesario.

Módulos lógicos y módulos físicos

Un módulo lógico es una unidad de responsabilidad en el diseño: Catálogo, Pedidos, Pagos. Un módulo físico es una unidad que se compila, se versiona o se despliega: un archivo JAR de Java, un wheel de Python, un paquete npm, una imagen de contenedor, una función serverless. La Sección 1.1 las llamó estructuras de módulos y estructuras de asignación. Las dos están relacionadas, pero no son lo mismo, y confundirlas provoca algunos de los errores más caros de la arquitectura moderna.

El mismo diseño lógico se puede corresponder con módulos físicos de maneras muy diferentes. Todos los módulos lógicos pueden vivir en una sola unidad desplegable, llamándose entre sí dentro del mismo proceso: un monolito modular. O cada uno puede desplegarse como su propio servicio, llamando a los demás a través de la red: microservicios. Los límites lógicos, y las decisiones que oculta cada módulo, pueden ser idénticos en ambos casos. Lo que cambia es el costo de cruzar un límite (una llamada a una función tarda nanosegundos; una llamada de red, milisegundos, y puede fallar), la capacidad de lanzar y escalar las partes de forma independiente y el trabajo de operación.

Un diseño lógico, dos diseños físicos



Mismos módulos lógicos y dependencias; distintos módulos físicos (de despliegue)

Los módulos lógicos describen responsabilidades; los módulos físicos describen lo que se compila y se despliega, y el mismo diseño lógico se puede empaquetar de cualquiera de las dos formas.

Un error común y costoso es tratar la separación física como si creara modularidad lógica. Dividir en servicios un sistema mal descompuesto no oculta ninguna decisión; convierte las llamadas dentro del proceso entre módulos enredados en llamadas de red entre servicios enredados, lo que a veces se llama monolito distribuido. También se da el error contrario: un equipo mantiene una sola unidad desplegable, pero deja que sus módulos internos crezcan unos dentro de otros, de modo que después ya nunca podrá separarlos. El consejo práctico de muchos profesionales es acertar primero con los límites lógicos, hacerlos cumplir y solo entonces decidir cuáles de ellos merecen convertirse en límites físicos.

La granularidad y sus costos

¿Qué tamaño debe tener un módulo? La granularidad es el tamaño de los módulos en relación con el sistema, y no hay una respuesta correcta universal, porque los costos tiran en direcciones opuestas. Los módulos más grandes tienen menos interfaces que diseñar y mantener, pero cada uno contiene más motivos de cambio sin relación entre sí, más personas editándolo a la vez y más cosas que entender antes de hacer un cambio. Los módulos más pequeños son, cada uno, más fáciles de entender, pero el sistema tiene más interfaces, más dependencias que gestionar y más lugares en los que hay que ensamblar una misma funcionalidad.

El costo de una interfaz también depende del tipo de límite. Entre dos clases de un mismo programa, una interfaz adicional cuesta un poco de tiempo de lectura. Entre dos servicios cuesta un salto de red, serialización, una política de versionado, monitorización, un modo de

fallo y, posiblemente, el acuerdo de un segundo equipo para cada cambio. Por eso la granularidad adecuada para los servicios suele ser mucho más gruesa que la granularidad adecuada para las clases.

Síntomas de una granularidad equivocada

Demasiado gruesa	Demasiado fina
Funcionalidades sin relación entre sí cambian el mismo módulo; muchas personas editan los mismos archivos y entran en conflicto en el control de versiones.	Una sola funcionalidad pequeña necesita cambios coordinados en muchos módulos o servicios.
Tiempos largos de compilación y de pruebas para cualquier cambio; nadie entiende el módulo entero.	Muchas interfaces triviales de paso que no añaden ninguna abstracción; cadenas de llamadas que atraviesan varios módulos.
Los equipos hacen cola para lanzar porque todo sale a la vez.	A través de una red: latencia alta por llamadas excesivamente frecuentes, fallos parciales y transacciones distribuidas.
El nombre del módulo es vago, como `core`, `common` o `manager`.	Cada módulo es superficial: su interfaz es casi tan compleja como su implementación (consulta la Sección 3.3).

Hacer cumplir los límites de los módulos en el código

Un límite que solo existe en un diagrama no sobrevivirá al choque con un plazo de entrega. Tarde o temprano alguien importará una clase interna porque es más rápido, y en cuanto existe un atajo le siguen otros; la Sección 1.7 llamó a esto erosión. Los lenguajes y las herramientas de compilación ofrecen varias formas de hacer que el límite previsto sea el camino de menor resistencia, o de hacer que cruzarlo haga fallar la compilación.

Python se basa en convenciones: los nombres que empiezan con un guion bajo se consideran internos (PEP 8), y el `__init__.py` de un paquete puede volver a exportar solo sus funciones públicas. Nada impide un import de `shop.payments._stripe_client`, así que los equipos a los que les importa usan un comprobador como import-linter para hacer fallar la compilación cuando aparece un import prohibido. Java ofrece la visibilidad de paquete (package-private, la predeterminada cuando no se indica ningún modificador) y, desde Java 9 en 2017, el Java Platform Module System: un archivo `module-info.java` declara qué módulos `requires` este y cuáles de sus paquetes `exports`; los paquetes que no se exportan no se pueden usar desde

fuera, ni en tiempo de compilación ni en tiempo de ejecución. Go trata cualquier paquete situado bajo un directorio llamado `internal` como importable solo desde el código que cuelga del directorio padre de ese directorio. C# tiene `internal`, visible solo dentro de un ensamblado; Rust tiene `pub(crate)`.

```
# shop/payments/__init__.py  (the public face of the package)
from shop.payments.api import charge, refund, PaymentResult

__all__ = ["charge", "refund", "PaymentResult"]

# shop/payments/_provider_client.py   internal: leading underscore
# shop/payments/_retry.py             internal
# shop/payments/api.py                the only module callers should use

# Elsewhere in the shop:
from shop.payments import charge          # fine
from shop.payments._retry import backoff  # works, but breaks the boundary
```

Un paquete de Python que publica una interfaz pequeña y mantiene internos, por convención, sus módulos de implementación.

```
// module-info.java for the payments module
module shop.payments {
    requires shop.orders.api;        // what this module depends on
    exports shop.payments.api;       // the only package others can use
    // shop.payments.provider and shop.payments.retry are not exported,
    // so code in other modules cannot compile against them
}
```

El mismo límite en Java, impuesto por el compilador y por el entorno de ejecución mediante un descriptor de módulo.

Formas de hacer cumplir un límite

Mecanismo	Ejemplos	Fuerza
Convención de nombres	Guion bajo inicial en Python; `internal` en el nombre de un espacio de nombres	Débil: depende de la disciplina y de la revisión de código
Visibilidad del lenguaje	Package-private de Java, `internal` de C#, `pub(crate)` de Rust, directorios `internal` de Go	Fuerte dentro de una compilación; la comprueba el compilador
Sistema de módulos del lenguaje	Java Platform Module System (`requires` , `exports`); el campo `exports` de un `package.json` de Node.js	Fuerte: un acceso no declarado no compila o no se carga
Módulos de la herramienta de compilación	Compilaciones multimódulo de Maven o Gradle; reglas de visibilidad de Bazel; paquetes separados en un monorrepositorio	Fuerte: un módulo solo puede usar lo que declara como dependencia
Comprobaciones automáticas de arquitectura	import-linter, ArchUnit, dependency-cruiser (Sección 3.4)	Fuerte si se ejecuta en el pipeline de compilación como función de aptitud
Límite de proceso y de red	Servicios separados con sus propios almacenes de datos, accesibles solo a través de una API	El más fuerte, y el más caro de cruzar y de cambiar

EJEMPLO PRÁCTICO

Ejemplo práctico: descomponer una tienda en línea por diagrama de flujo y por cambios probables

El proceso de pago de una tienda en línea se diseña primero como un diagrama de flujo: Recibir pedido, Validar carrito, Calcular precio (promociones e IVA), Cobrar el pago, Guardar pedido, Enviar confirmación. Los seis pasos leen y escriben un único registro de pedido compartido, un diccionario cuyas claves conocen todos los pasos. El diseño alternativo tiene seis módulos, cada uno de los cuales oculta una decisión: Pedido (cómo se representa y se guarda un pedido), Precios (reglas de promociones), Impuestos (reglas del IVA), Pasarela de pago (qué proveedor y cómo se le llama), Notificaciones (canales y plantillas) y un Flujo de compra que solo ordena las llamadas, como el control maestro de Parnas. Se cuentan cinco cambios que el negocio espera para el año próximo frente a ambos diseños.

Cambio 1: añadir un segundo proveedor de pagos	Diagrama de flujo: Cobrar el pago, más Guardar pedido, que almacena el formato de la referencia de transacción del proveedor = 2 módulos. Por cambio: Pasarela de pago = 1.
Cambio 2: cambian las reglas del IVA de una categoría de productos	Diagrama de flujo: Calcular precio, Guardar pedido (columnas del IVA) y Enviar confirmación (desglose del IVA) = 3. Por cambio: Impuestos = 1.
Cambio 3: dividir la dirección de entrega en campos estructurados	Diagrama de flujo: todos los pasos leen el registro de pedido compartido = 6. Por cambio: Pedido = 1 (los demás llaman a sus métodos).
Cambio 4: añadir confirmaciones por SMS	Diagrama de flujo: Enviar confirmación = 1. Por cambio: Notificaciones = 1.
Cambio 5: una funcionalidad nueva, un mensaje de regalo en los pedidos	Diagrama de flujo: Recibir pedido, Guardar pedido, Enviar confirmación = 3. Por cambio: Flujo de compra, Pedido, Notificaciones = 3.
Total de módulos afectados	Diagrama de flujo: 2 + 3 + 6 + 1 + 3 = 15. Por cambio: 1 + 1 + 1 + 1 + 3 = 7.

Los cambios en decisiones existentes (proveedor, reglas fiscales, representación) se quedan dentro de un módulo en el diseño basado en el cambio; eso es exactamente lo que predijo Parnas.

Algunos cambios cuestan lo mismo en ambos diseños (SMS), y una funcionalidad realmente nueva (el mensaje de regalo) cruza módulos en los dos. La ocultación de información protege las decisiones que anticipaste, no cualquier funcionalidad posible.

El recuento es una forma rápida y repetible de comparar descomposiciones candidatas en una revisión de diseño: primero enumera los cambios probables y después cuenta los módulos que toca cada uno.

Errores comunes

Descomponer solo por capa técnica. Las carpetas llamadas `controllers`, `services` y `repositories` son módulos en cierto modo, pero un cambio en cómo funciona el IVA tocará las tres. Las capas son útiles (Sección 3.4), pero una descomposición que solo tiene capas es un diagrama de flujo disfrazado. La mayoría de los sistemas maduros combinan una división por capacidad de negocio con capas dentro de cada parte.

Todo compartido. Un módulo llamado `common` o `utils` que importan todos los demás módulos se convierte en el lugar donde se acumulan decisiones sin relación entre sí, y cualquier cambio en él se propaga por todas partes. **Límites que no se hacen cumplir.** Los diagramas muestran módulos limpios mientras el código tiene cientos de imports cruzados. **Módulos definidos por el organigrama de una reorganización pasada** en lugar de por las decisiones que el sistema tiene que ocultar; la Sección 3.7 explica por qué las organizaciones y los límites de los módulos se influyen mutuamente. **Distribución prematura.** Convertir cada módulo lógico en un servicio antes de que los límites hayan demostrado ser estables encarece las correcciones posteriores de esos límites.

Lista de comprobación: revisar una descomposición propuesta

- [ ] **Nombra la decisión que oculta cada módulo**
  Si no puedes decir en una frase qué mantiene en secreto un módulo, probablemente sea un paso de procesamiento o un cajón de sastre.
- [ ] **Enumera los cambios probables**
  Anota entre cinco y diez cambios que probablemente traigan el negocio o la tecnología, y cuenta los módulos que tocaría cada uno.
- [ ] **Comprueba el tamaño de la interfaz**
  La interfaz debe ser mucho más pequeña que la implementación. Las interfaces grandes con poco comportamiento detrás sugieren que el corte es equivocado.
- [ ] **Separa lo lógico de lo físico**
  Acuerda primero los módulos lógicos; decide por separado cuáles merecen su propia unidad desplegable y por qué.
- [ ] **Decide cómo se hará cumplir el límite**
  Elige un mecanismo de visibilidad o una comprobación automática, y añádelo a la compilación antes de que aparezca el primer atajo.
- [ ] **Haz coincidir los límites con la propiedad**
  Cada módulo debe tener un equipo propietario; los módulos que comparten varios equipos para la mayor parte de su trabajo son una señal de alarma.

Cómo aplican la modularidad los profesionales

Los arquitectos con experiencia tratan la estructura de módulos como el artefacto de diseño más importante para la mantenibilidad, y la revisan una y otra vez. Al principio de la vida de un producto, cuando el dominio se entiende mal, prefieren menos módulos, más grandes y con límites internos fuertes, porque fusionar y dividir módulos dentro de una misma base de código es barato. A medida que el dominio se estabiliza y los cambios probables se aclaran, afinan los límites y solo entonces estudian cuáles deberían desplegarse por separado. Las técnicas del diseño guiado por el dominio, como los contextos delimitados, ayudan a encontrar límites que siguen el lenguaje del negocio.

También miden. Contar los módulos que han tocado los cambios recientes, algo que el historial del control de versiones facilita, muestra si los límites funcionan; la Sección 3.8 presenta el acoplamiento de cambio y los hotspots. La siguiente sección, la Sección 3.2, ofrece el vocabulario para describir cómo son los límites buenos y los malos: el acoplamiento entre módulos y la cohesión dentro de ellos.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Por qué Parnas llamó a un módulo una "asignación de responsabilidad" y no un subprograma?**

Porque lo que importa es de qué decisiones de diseño es responsable un módulo y cuáles oculta, no cómo se empaqueta su código; una responsabilidad puede abarcar varias rutinas, y las rutinas no tienen por qué coincidir con los pasos del procesamiento.

▼ **En el ejemplo KWIC, ¿por qué cambiar la forma de guardar las líneas afectaba a varios módulos en la primera descomposición?**

Todos los módulos compartían directamente los formatos de datos en memoria, así que todos los módulos conocían la decisión de almacenamiento en lugar de estar oculta en uno solo.

▼ **¿Cuál es la diferencia entre un módulo lógico y un módulo físico?**

Un módulo lógico es una unidad de responsabilidad en el diseño; un módulo físico es una unidad que se compila o se despliega, como un JAR, un paquete, un contenedor o un servicio. Un mismo diseño lógico se puede empaquetar físicamente de muchas maneras.

▼ **Nombra dos señales de que los servicios tienen una granularidad demasiado fina.**

Las funcionalidades pequeñas necesitan cambios coordinados en varios servicios, y las solicitudes implican cadenas de llamadas de red excesivamente frecuentes, con latencia alta y fallos parciales.

▼ **Inténtalo: Un proceso de pago de entradas se divide en pasos, Seleccionar, Retener, Fijar precio, Pagar y Emitir, que comparten un único registro de reserva cuyos campos conoce cada paso. Se acercan promociones y una nueva forma de guardar las retenciones. Critica esto y propón una división mejor.**

Esta es una descomposición por diagrama de flujo: cada decisión de diseño está repartida por todos los pasos, de modo que cambiar el registro de reserva o el almacenamiento de retenciones afecta a los cinco. Siguiendo a Parnas, haz que cada módulo oculte una decisión que probablemente cambie: Booking oculta la forma del registro, Holds oculta cómo se almacenan las retenciones, Pricing oculta las reglas de promoción y Payments oculta al proveedor. Cada uno ofrece una interfaz estrecha, de modo que las promociones cambian un solo módulo. Haz cumplir los límites, o se erosionarán.

FUENTES

- D. L. Parnas, On the Criteria To Be Used in Decomposing Systems into Modules (Communications of the ACM, 1972): https://doi.org/10.1145/361598.361623
- JEP 261: Module System (OpenJDK): https://openjdk.org/jeps/261
- Go 1.4 release notes: internal packages: https://go.dev/doc/go1.4
- Import Linter documentation: https://import-linter.readthedocs.io/en/stable/
- Dijkstra: On the role of scientific thought (EWD 447): https://www.cs.utexas.edu/~EWD/transcriptions/EWD04xx/EWD447.html

SECCIÓN 3.2 DE 3.8 · INTERMEDIO · 29 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Acoplamiento y cohesión

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.3 Organizar el código
- 2.6 Modificabilidad y desplegabilidad
- 3.1 Modularidad

Términos anteriores que utiliza: Módulo, Interfaz, Llamada síncrona, Mensajería asíncrona, Refactorización

LA IDEA EN BREVE

Piensa en dos vecinos que comparten un jardín. Si solo se prestan alguna herramienta de vez en cuando, cuando se la piden, cualquiera de los dos puede rediseñar su propio jardín sin problemas: están débilmente acoplados. Si comparten un cobertizo, una manguera y una llave debajo de una maceta, cada cambio exige una conversación: están fuertemente acoplados. Ahora piensa en un cajón de la cocina. Si solo contiene cubiertos, sabes dónde buscar: es cohesivo. Si contiene cubiertos, pilas, cordel y recibos viejos, no lo es. Un buen software tiene partes cohesivas que están débilmente acopladas entre sí.

LO QUE NECESITAS SABER

- El acoplamiento es el grado en que un módulo depende de otro; la cohesión es el grado en que los elementos de un módulo forman un todo. Busca un acoplamiento débil y una cohesión alta, que normalmente van juntos.
- Las escalas del diseño estructurado ordenan el acoplamiento de contenido, común, externo, de control y de marca hasta el acoplamiento de datos (del más fuerte al más débil), y la cohesión coincidental, lógica, temporal, procedimental, comunicacional y secuencial hasta la funcional (de la más débil a la más fuerte).
- El acoplamiento aferente (Ca) cuenta las dependencias entrantes, y el acoplamiento eferente (Ce), las salientes; un módulo con un Ca alto es arriesgado de cambiar, y uno con un Ce alto está expuesto a los cambios de los demás.
- Los sistemas distribuidos añaden el acoplamiento temporal (ambas partes deben estar disponibles a la vez), el acoplamiento de despliegue (deben lanzarse juntas) y el acoplamiento semántico (deben compartir un significado oculto), que una API limpia por sí sola no elimina.

- Entre las tácticas que reducen el acoplamiento están las interfaces estrechas, los intermediarios como los adaptadores y los brokers, los eventos en lugar de los comandos y una propiedad estricta de los datos, cada una con su propio costo.
- El acoplamiento no se puede eliminar, solo elegir: parte del acoplamiento es la razón de ser del sistema, así que el objetivo es hacerlo explícito, débil y local donde el cambio es probable.

Dos medidas de la estructura

El acoplamiento es el grado en que un módulo depende de otro, de modo que un cambio en uno puede obligar a cambiar el otro. La cohesión es el grado en que los elementos de un módulo forman un todo porque sirven a un único propósito. Ambas ideas las introdujo Larry Constantine a finales de la década de 1960 como parte del movimiento del diseño estructurado, y se publicaron en el artículo *Structured Design* de Wayne Stevens, Glenford Myers y Constantine en el *IBM Systems Journal* de 1974, y después con más extensión en el libro de Edward Yourdon y Constantine *Structured Design* (1979). La Sección 0.3 las presentó de manera informal; esta sección las precisa.

Las dos están relacionadas. Si las cosas que cambian juntas viven en el mismo módulo (cohesión alta), menos cambios cruzan los límites entre módulos, y los módulos necesitan saber menos unos de otros (acoplamiento bajo). Yourdon y Constantine llegaron a decir que, en la práctica, la cohesión era la más importante de las dos. Las tácticas de modificabilidad de la Sección 2.6 (aumentar la cohesión, reducir el acoplamiento) se basan exactamente en esta pareja.

Una aclaración desde el principio: el acoplamiento no es un defecto. Un sistema cuyos módulos nunca dependieran unos de otros no haría nada útil. Martin Fowler, que escribió sobre el tema en 2001, advirtió que intentar eliminar por completo el acoplamiento solo empuja todo a un módulo enorme en el que el acoplamiento está oculto, no desaparecido. El objetivo es elegir el acoplamiento de forma deliberada: pocas dependencias, a través de interfaces estrechas y estables, que apunten en direcciones sensatas y tan débiles como permita la tarea.

Las escalas clásicas



Las escalas del diseño estructurado: el acoplamiento, del de contenido (el más fuerte) al de datos (el más débil), y la cohesión, de la coincidental (la más débil) a la funcional (la más fuerte).

Los niveles de acoplamiento

El diseño estructurado ordenó las formas en que pueden conectarse dos módulos, de la más perjudicial a la más inofensiva. La forma de seis niveles que aparece a continuación suele atribuirse a los libros de Glenford Myers sobre diseño compuesto de mediados de la década de 1970, y más tarde la popularizó Meilir Page-Jones; los nombres y los límites varían ligeramente entre fuentes, pero esta es la forma que recogen la mayoría de los libros de texto. Se definieron para procedimientos dentro de un programa, pero cada uno tiene un equivalente directo entre componentes y servicios, que es donde los encuentran hoy los arquitectos.

Niveles de acoplamiento, del más fuerte al más débil

Nivel	Qué significa	En el código	Entre servicios
Contenido	Un módulo se mete en las partes internas de otro: cambia sus datos privados o salta dentro de su código.	Asignar `cart._items` desde fuera; aplicar un monkey-patch a la función de otro módulo.	Leer o escribir directamente en las tablas de la base de datos de otro servicio.
Común	Varios módulos leen y escriben los mismos datos globales.	Un diccionario global `config` que cualquier función puede cambiar.	Varios servicios que comparten un esquema de base de datos o un espacio de claves de caché compartido.
Externo	Los módulos comparten un formato, un protocolo o un dispositivo impuestos desde fuera.	Dos módulos analizan el mismo formato de archivo de un proveedor.	Varios servicios que dependen de la estructura de archivo de un socio o de una versión concreta de un protocolo de mensajes.
Control	Un módulo pasa un indicador que le dice al otro qué hacer, de modo que quien llama conoce la lógica interna de quien es llamado.	`process(order, mode="refund")`	Una API con un campo `action` que selecciona comportamientos completamente distintos.
Marca	Un módulo recibe una estructura de datos entera, pero solo usa una parte.	Pasar el objeto `Customer` completo a una función que solo necesita la dirección de correo electrónico.	Enviar el registro completo del cliente en cada evento cuando los consumidores necesitan dos campos.
Datos	Los módulos se comunican solo mediante parámetros de datos simples, todos ellos necesarios.	`send_receipt(email, order_id, total)`	Una API o un evento estrecho y documentado, con solo los campos que necesitan los consumidores.

Dos advertencias. Primero, la escala es una guía del riesgo, no una ley. El acoplamiento de marca suele ser un intercambio sensato: pasar un objeto entero es más sencillo que desempaquetar seis campos, y el único costo es que el receptor puede romperse ahora cuando cambian campos que ni siquiera usa. Segundo, la escala no dice nada sobre cuántas dependencias hay ni sobre la frecuencia con que cambia la otra parte. Una sola llamada con acoplamiento de control a un módulo que nunca cambia puede ser inofensiva; el acoplamiento

de datos con un módulo que cambia cada semana puede hacer daño. Más adelante en este capítulo, la connascence ofrece un vocabulario más fino, y *Balancing Coupling in Software Design* (2024), de Vlad Khononov, sostiene que la fuerza del acoplamiento debe juzgarse siempre junto con la distancia entre los módulos y su volatilidad.

```
# Control coupling: the caller chooses the callee's internal branch
def handle_payment(order, mode):
    if mode == "charge":
        ...  # call the provider, store the transaction
    elif mode == "refund":
        ...  # look up the transaction, reverse it
    elif mode == "dry_run":
        ...  # validate only

handle_payment(order, "refund")

# Data coupling: separate operations, each taking only what it needs
def charge(order_id: str, amount_pence: int) -> str: ...
def refund(transaction_id: str, amount_pence: int) -> None: ...

refund("tx_81f2", 1999)
```

Acoplamiento de control (arriba) frente a acoplamiento de datos (abajo): el indicador obliga a quien llama a saber cómo está organizado por dentro quien es llamado.

SI ESTO ES NUEVO PARA TI

Paso a paso: un recibo, cuatro tipos de acoplamiento

La idea más difícil de esta sección es por qué algunos tipos de acoplamiento son peores que otros. A grandes rasgos, cuanto más depende un módulo de cosas que otros módulos controlan o pueden cambiar, más cambios de esos módulos pueden romperlo. Sigue una pequeña tarea, enviar un recibo, escrita de cuatro maneras (cuatro de los seis niveles de la tabla, del más fuerte al más débil).

- El código del recibo necesita la dirección de correo electrónico del cliente, el número de pedido y el total. En la primera versión, se mete en los datos privados del módulo de clientes y lee directamente el campo del correo electrónico: acoplamiento de contenido. Si se cambia el nombre de ese campo, el recibo se rompe, aunque nadie prometió que se mantendría.
- En la segunda versión, el proceso de pago pone la dirección de correo electrónico en un diccionario global que cualquier función puede leer o cambiar, y el código del recibo la lee de ahí: acoplamiento común. Ahora cualquier código que escriba en ese diccionario puede enviar el recibo a quien no corresponde, y nada en el código del recibo muestra cuál.
- En la tercera versión, al código del recibo se le pasa el objeto Customer entero y solo usa la dirección de correo electrónico: acoplamiento de marca. Ya no se mete dentro, pero todavía puede romperse cuando cambian campos que no usa.
- En la cuarta versión, recibe solo los tres valores simples que necesita: acoplamiento de datos, el nivel más débil. No sabe nada de cómo se guardan los clientes ni los pedidos, así que el módulo de clientes puede reorganizarse libremente.
- **La lección:** cada paso hacia el extremo de los datos de la escala significa que el código del recibo depende de menos cosas que controlan otros módulos, así que menos cambios de esos módulos pueden romperlo. La escala es una guía del riesgo, no una ley: pasar el objeto entero suele ser un intercambio sensato.

Los niveles de cohesión

La cohesión pregunta por qué están juntos los elementos de un módulo. Yourdon y Constantine describieron siete tipos de asociación, también de peor a mejor. Los niveles inferiores agrupan cosas por razones que no tienen que ver con el problema (se escribieron por casualidad al mismo tiempo, o se ejecutan en el mismo momento); los niveles superiores agrupan cosas porque trabajan sobre los mismos datos o hacia un mismo propósito único. Los resúmenes posteriores de los libros de texto coinciden en el orden general: la cohesión coincidental y la lógica son pobres, la comunicacional y la secuencial son buenas y la funcional es la mejor, aunque los saltos entre niveles no están espaciados de manera uniforme.

Niveles de cohesión, del más débil al más fuerte

Nivel	Los elementos están agrupados porque…	Ejemplo
Coincidental	no hay ninguna relación significativa entre ellos.	Un módulo `utils` que contiene el formateo de fechas, un decorador de reintentos y un tipo de IVA.
Lógica	hacen el mismo tipo de cosa, seleccionada mediante un indicador.	Un módulo `exports` con una función que escribe CSV, PDF o XML según un parámetro.
Temporal	se ejecutan al mismo tiempo.	Una rutina `startup()` que abre la base de datos, precarga las cachés y envía un ping de monitorización.
Procedim.	se suceden en una secuencia de control, aunque sobre datos distintos.	Una función que valida un formulario, después registra la dirección IP del usuario y después actualiza un menú.
Comunicacional	operan sobre los mismos datos.	Un módulo que calcula varios informes a partir del mismo historial de pedidos.
Secuencial	la salida de uno es la entrada del siguiente.	Analizar un extracto bancario, después clasificar sus líneas y después totalizar cada categoría.
Funcional	juntos realizan una tarea bien definida y nada más.	Un módulo `Tax` que calcula el IVA de un carrito, y solo eso.

A escala de arquitectura, la cohesión es la cuestión de si un componente o un servicio tiene una sola razón para existir. Un servicio llamado `CustomerService` que gestiona el inicio de sesión, los puntos de fidelidad y las preferencias de marketing tiene, como mucho, cohesión lógica: tres equipos lo cambiarán por tres motivos sin relación entre sí. Una buena heurística es describir el propósito de un módulo en una sola frase sin usar la palabra *y*; la Sección 3.5 desarrolla esta idea como el principio de responsabilidad única, definido en función de las personas que solicitan los cambios.

Acoplamiento aferente y eferente

Los niveles anteriores describen el *tipo* de una dependencia. Los arquitectos también necesitan contarlas. El acoplamiento aferente (Ca) es el número de módulos externos a un componente que dependen de él: sus flechas entrantes. El acoplamiento eferente (Ce) es el número de módulos externos de los que depende: sus flechas salientes. Los términos vienen de la anatomía, donde los nervios aferentes llevan las señales hacia dentro y los eferentes, hacia

fuera. Robert C. Martin los definió para paquetes en 1994 contando clases; las herramientas actuales cuentan clases, módulos o paquetes, así que comprueba siempre qué cuenta una herramienta antes de comparar cifras.

Las dos cifras cuentan historias distintas. Un componente con un Ca alto, como una biblioteca compartida `money` que usan cuarenta módulos, es *responsable*: cualquier cambio en él puede romper muchos otros, así que debe cambiar con cuidado y rara vez. Un componente con un Ce alto, como un flujo de compra que llama a precios, existencias, pagos, entregas y notificaciones, está *expuesto*: puede necesitar cambios cada vez que cambie cualquiera de ellos. Ninguna de las dos situaciones es mala en sí misma; el problema surge cuando de un componente dependen muchos otros y, a la vez, él mismo depende de muchas cosas volátiles. La Sección 3.4 convierte Ca y Ce en la métrica de inestabilidad de Martin y en las reglas que se basan en ella.

EJEMPLO PRÁCTICO

Ejemplo práctico: contar Ca y Ce en una app de reparto de comida

Siete módulos (Caja, Carta, Precios, Pagos, Notificaciones, Seguimiento de repartidores y Administración) tienen estas dependencias ("A depende de B" se escribe "A a B"): Caja a Carta, Caja a Precios, Caja a Pagos, Caja a Notificaciones; Seguimiento de repartidores a Notificaciones; Precios a Carta; Administración a Carta, Administración a Precios.

Carta	Ca = 3 (Caja, Precios, Administración); Ce = 0
Precios	Ca = 2 (Caja, Administración); Ce = 1 (Carta)
Notificaciones	Ca = 2 (Caja, Seguimiento de repartidores); Ce = 0
Pago	Ca = 0; Ce = 4 (Carta, Precios, Pagos, Notificaciones)
Lectura	Carta es el módulo del que más se depende, así que su interfaz necesita el máximo cuidado y los cambios más lentos. Caja es el que más depende de otros, así que debería ser el más fácil de cambiar y el que absorba los cambios de los demás.

Contar lleva unos minutos con una herramienta de dependencias y convierte una impresión vaga ("Carta es central") en una cifra que puedes seguir a lo largo del tiempo.

El acoplamiento en los sistemas distribuidos

Cuando los módulos se convierten en servicios que se comunican a través de una red, los niveles clásicos siguen siendo válidos, pero cobran importancia otros tres tipos de acoplamiento. Ninguno de ellos es visible en la firma de una función, por eso los equipos que dividen un sistema en servicios suelen sorprenderse de lo acoplados que siguen estando los servicios.

El acoplamiento temporal significa que una parte solo funciona si otra está disponible y responde en ese mismo momento. Una llamada HTTP síncrona de Caja a Pagos acopla su disponibilidad: si Pagos cae, también cae el proceso de pago, y la disponibilidad combinada es el producto de las dos (Sección 2.5). Los mensajes asíncronos a través de una cola eliminan este acoplamiento, a cambio de resultados eventuales en lugar de inmediatos. (En el diseño a nivel de código, el mismo término significa a veces que las llamadas deben hacerse en un orden fijo, como `open()` antes de `read()`; normalmente el contexto deja claro el significado).

El acoplamiento de despliegue significa que dos partes deben lanzarse juntas, o en un orden fijo, porque un cambio en una rompe la otra. Si cada lanzamiento de Pedidos necesita un lanzamiento simultáneo de Facturación, en la práctica los dos son una sola unidad desplegable, diga lo que diga el diagrama. Los cambios compatibles con versiones anteriores, las API versionadas y el patrón expandir y contraer son los remedios habituales.

El acoplamiento semántico significa que dos partes deben estar de acuerdo en un significado que ninguna interfaz expresa: que un pedido con `status = 4` está enviado, que los precios están en peniques e incluyen el IVA, que un "cliente activo" ha hecho un pedido en los últimos 90 días. Estos acuerdos se rompen en silencio cuando una de las partes cambia su interpretación. El remedio es hacer explícito el significado en el contrato (estados con nombre, unidades en los nombres de los campos, esquemas publicados) o mantener la regla en un único lugar al que los demás consultan.

Tácticas para reducir el acoplamiento

Interfaces estrechas. Expón el menor número de operaciones y campos que necesiten los clientes. Cada campo adicional en una respuesta de una API es algo de lo que un cliente puede llegar a depender (Sección 3.3, sobre la ley de Hyrum). **Intermediarios.** Pon algo entre las dos partes para que ninguna conozca los detalles de la otra: un adaptador o una capa anticorrupción que traduzca el modelo de un proveedor, un API gateway o un intermediario de mensajes (message broker) que desacople al emisor del receptor en el tiempo y en la identidad. **Eventos en lugar de comandos.** Un comando (*Facturación, crea una factura*) obliga al emisor a conocer al receptor y su tarea; un evento (*se ha hecho un pedido*) permite al emisor declarar un hecho y a cualquier número de receptores reaccionar, así que se pueden añadir

consumidores nuevos sin cambiar al productor. **Propiedad de los datos.** Cada dato tiene un único propietario que lo escribe; los demás módulos se lo piden al propietario a través de su interfaz o mantienen su propia copia alimentada por eventos, sin leer nunca las tablas del propietario (Sección 7.1).

Toda táctica tiene un precio. Los intermediarios añaden un salto y otra cosa más que operar. Los eventos renuncian a la consistencia inmediata y hacen más difícil ver en un solo lugar el flujo de un proceso de negocio. Las copias de datos pueden estar desactualizadas. La propiedad de los datos obliga a construir y versionar API que una tabla compartida te daba gratis. El compromiso merece la pena donde el cambio es frecuente o los equipos están separados, y a menudo no la merece entre dos clases que mantiene una sola persona.

La ley de Deméter: habla solo con tus amigos inmediatos

La **ley de Deméter** surgió del proyecto Demeter de la Northeastern University, donde Ian Holland identificó la regla en 1987; Karl Lieberherr, Holland y A. Riel la publicaron en la conferencia OOPSLA en 1988. Su forma general es que cada unidad solo debe tener un conocimiento limitado de otras unidades, aquellas estrechamente relacionadas con ella: "habla solo con tus amigos inmediatos". En términos de orientación a objetos, un método puede enviar mensajes a su propio objeto, a sus argumentos, a los objetos que crea y a las partes inmediatas del objeto, pero no a desconocidos a los que se llega a través de ellos.

Piensa en una rutina de asignación de repartidores de la plataforma de reparto de comida que contiene `order.getCustomer().getAddress().getPostcode()`. Necesita una sola cosa, un código postal, pero ahora sabe que un pedido tiene un cliente, que un cliente tiene una dirección y que una dirección tiene un código postal. Cada uno de esos hechos es una dependencia. Si los clientes pasan a tener varias direcciones guardadas, se rompe código que no tiene nada que ver con los clientes. Escrita como `order.deliveryPostcode()`, la misma llamada depende de una sola operación, y la estructura que hay detrás puede cambiar libremente. Un principio relacionado es "ordena, no preguntes" (tell, don't ask): en lugar de sacar datos de un objeto para actuar sobre ellos, pide al objeto que haga el trabajo. En los términos de la Sección 3.6, la cadena larga difunde el conocimiento de la estructura muy lejos de donde le corresponde estar.

La idea se puede aplicar a mayor escala. Un cliente que sigue una cadena de llamadas (obtener el pedido, luego su restaurante y luego la cuenta de liquidación de ese restaurante) depende de cómo se relacionan tres servicios. Darle una sola operación para lo que necesita, quizá mediante un backend for frontend, elimina ese conocimiento, como recomiendan las interfaces estrechas de más arriba.

El nombre la exagera. Martin Fowler ha dicho que estaría más contento si se llamara la *sugerencia* de Deméter, y advierte contra tomar "ordena, no preguntes" como motivo para eliminar todos los métodos de consulta. Contar puntos es una mala prueba: una cadena sobre datos simples, o un builder fluido que sigue devolviendo el mismo objeto, no revela nada sobre colaboradores ocultos. Y aplicada a ciegas, la regla llena las clases de métodos de reenvío triviales, lo que ensancha sus interfaces. Úsala como una señal para preguntarte por qué este código necesita saber tanto.

Composición en lugar de herencia

En 1994, Gamma, Helm, Johnson y Vlissides lo convirtieron en el segundo principio de *Design Patterns*: "Prefiere la composición de objetos a la herencia de clases". Su razonamiento se basaba en la visibilidad. La herencia es reutilización de *caja blanca*: la subclase ve los detalles de su clase padre y depende de ellos. La composición es reutilización de *caja negra*: un objeto usa otro solo a través de su interfaz. En los términos de esta sección, la herencia es uno de los acoplamientos más fuertes que ofrece un lenguaje, y queda fijada al escribir el código, así que no puede cambiarse mientras el programa se ejecuta. Un cambio en una clase padre puede romper subclases que nadie ha tocado, un problema que suele llamarse **clase base frágil**.

La plataforma de reparto de comida muestra cómo sale mal. Un equipo modela `ExpressOrder` y `ScheduledOrder` como subclases de `Order`. Luego llegan los pedidos en grupo, y alguien necesita un pedido en grupo programado con entrega exprés. La herencia simple obliga a elegir una clase padre, y añadir clases para cada combinación crece rápidamente. Con la composición, un `Order` *tiene* una `FulfilmentPolicy` y una `PricingRule`. Cada una es una interfaz pequeña con varias implementaciones, las combinaciones no necesitan ninguna clase nueva y una política puede intercambiarse en tiempo de ejecución o sustituirse por una falsa en una prueba. Esa es la forma de la interfaz de pagos de la Sección 3.3.

El principio se ha extendido al diseño de lenguajes. Go no tiene ninguna jerarquía de tipos: los tipos satisfacen las interfaces automáticamente, y la incrustación ofrece algo análogo, pero no idéntico, a la herencia de clases. A escala arquitectónica, la misma preferencia se ve en los plug-ins detrás de una interfaz estable (Sección 5.5), en lugar de un framework que todos los equipos deban extender. Una biblioteca base compartida de la que todos los servicios deban heredar puede recrear la clase base frágil en toda una organización.

La palabra es *preferir*, no *siempre*. La herencia es razonable para una verdadera relación de subtipo que cumpla la sustitución de Liskov (Sección 3.5), sobre todo dentro de un módulo cuyos propietarios controlen ambos lados. La composición tiene sus propios costos: código de reenvío, más objetos pequeños y un comportamiento más difícil de seguir leyendo una sola clase. Pregúntate si el nuevo tipo *es* realmente el antiguo en todos los aspectos o si solo quiere reutilizar parte de su código. Si es lo segundo, compón.

Ejemplo práctico: refactorizar una pareja de pedidos y facturación fuertemente acoplada

El módulo Pedidos de una tienda en línea llama directamente a Facturación cada vez que se hace un pedido. Después, Facturación lee la tabla de pedidos para obtener los detalles, interpreta los códigos numéricos de la columna de estado y vuelve a calcular el IVA a partir de los precios de las líneas. El código siguiente es una versión simplificada; muestra tres de los problemas a la vez.

```
# orders.py
def place_order(basket, customer_id):
    order_id = db.insert("orders", customer_id=customer_id, status=1,
                         lines=basket.lines)
    billing.process(order_id, mode="invoice")   # Orders waits for Billing
    return order_id

# billing.py
def process(order_id, mode):
    row = db.query_one("SELECT * FROM orders WHERE id = %s", order_id)
    if row["status"] ≠ 1:              # 1 means 'placed', by convention
        return
    net = sum(l["price"] * l["qty"] for l in row["lines"])
    vat = round(net * 0.2)               # VAT rule duplicated here
    if mode == "invoice":
        db.insert("invoices", order_id=order_id, net=net, vat=vat)
    elif mode == "credit_note":
        db.insert("credit_notes", order_id=order_id, net=-net, vat=-vat)
```

Antes: acoplamiento de control (el indicador de modo), acoplamiento común (ambos módulos usan la tabla de pedidos) y acoplamiento semántico (los dos conocen los códigos de estado y las reglas del IVA).

Cualquier cambio en la estructura de la tabla de pedidos, en los códigos de estado o en las reglas del IVA tiene que hacerse ahora en los dos módulos a la vez. Pedidos no puede aceptar un pedido si Facturación va lenta o está fallando. Y los dos comportamientos de Facturación se seleccionan con un indicador que Pedidos tiene que conocer. La refactorización se hace en cuatro pasos: Pedidos deja de llamar a Facturación y, en su lugar, publica un evento `OrderPlaced` con exactamente los datos que necesita Facturación, incluidos los totales que Pedidos ya ha calculado; Facturación se suscribe y registra las facturas en su propio almacén; los dos comportamientos se convierten en dos operaciones; y la tabla de pedidos pasa a ser privada de Pedidos.

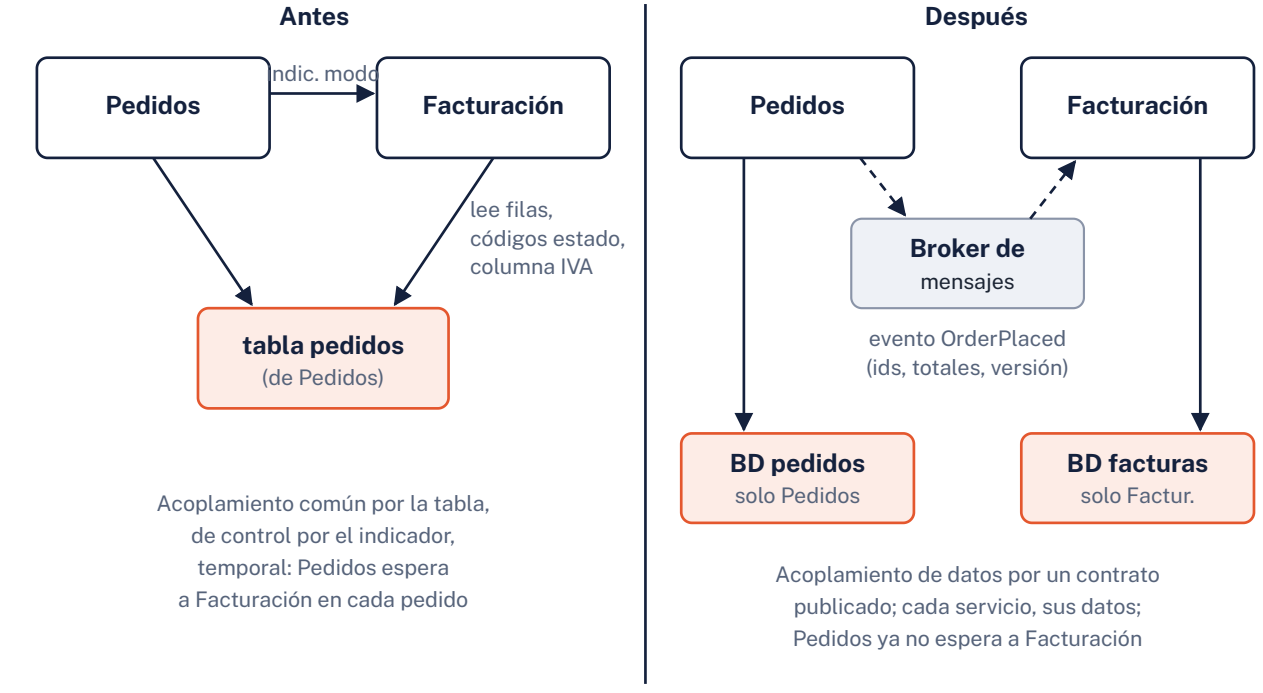
```
# orders.py
def place_order(basket, customer_id):
    order = Order.create(customer_id, basket)      # computes net and VAT once
    repository.save(order)
    events.publish("OrderPlaced", {
        "schema_version": 1, "order_id": order.id,
        "customer_id": customer_id,
        "net_pence": order.net_pence, "vat_pence": order.vat_pence})
    return order.id

# billing.py
@events.subscribe("OrderPlaced")
def on_order_placed(event):
    create_invoice(event["order_id"], event["net_pence"], event["vat_pence"])

def create_invoice(order_id, net_pence, vat_pence): ...
def create_credit_note(order_id, net_pence, vat_pence): ...
```

Después: Pedidos declara un hecho mediante un evento versionado; Facturación es propietaria de sus propios datos y ya no conoce las tablas, los códigos ni las reglas fiscales de Pedidos.

La pareja de pedidos y facturación, antes y después



La refactorización sustituye una llamada directa con un indicador y una tabla compartida por un evento publicado y almacenes de datos separados.

EJEMPLO PRÁCTICO

Qué cambió, medido

Comparación de las dos versiones con el vocabulario de acoplamiento de esta sección.

Tipo de acoplamiento	Antes: casi de contenido y común (tabla compartida), de control (indicador de modo), semántico (estado 1, regla del IVA). Después: acoplamiento de datos mediante un evento versionado de cinco campos.
Acoplamiento eferente de Pedidos	Antes: Ce incluye Facturación. Después: Pedidos depende solo del contrato del evento y del broker; no sabe que Facturación existe.
Acoplamiento temporal	Antes: un pedido falla si Facturación está caída. Después: el evento espera en el broker hasta que Facturación se recupera.
Acoplamiento de despliegue	Antes: un cambio en la tabla de pedidos necesita un lanzamiento conjunto. Después: Pedidos puede cambiar su tabla libremente; solo los cambios en el evento requieren cuidado, y el versionado se encarga de ellos.
Costos nuevos	Un broker que operar; las facturas aparecen unos segundos después en lugar de al instante; el esquema del evento es ahora un contrato público que hay que versionar; el proceso es más difícil de rastrear sin una buena observabilidad.

El evento lleva los totales que calculó Pedidos, así que el IVA se calcula en un solo lugar. Facturación ya no necesita conocer las reglas fiscales ni la estructura de las líneas de pedido.

La refactorización eliminó una elección, no todo el acoplamiento: Facturación sigue dependiendo del significado de `OrderPlaced`. Esa dependencia es ahora explícita, estrecha y versionada, que es lo que significa en la práctica el acoplamiento débil.

ATENCIÓN

Ideas erróneas habituales

- Los microservicios no están débilmente acoplados de forma automática. Los servicios que comparten una base de datos o que deben lanzarse juntos están fuertemente acoplados, se despliegue como se despliegue.
- Eliminar la duplicación puede aumentar el acoplamiento. Dos servicios que comparten una biblioteca para una regla de negocio pasan a cambiar juntos; a veces un poco de duplicación es el diseño menos acoplado (consulta la Sección 3.8, sobre DRY).
- La mensajería asíncrona elimina el acoplamiento temporal, no el semántico. Un consumidor que interpreta mal un campo de un evento está igual de roto, solo que más tarde.
- Un acoplamiento bajo no es lo mismo que pocas líneas entre recuadros en un diagrama. Una sola dependencia de una interfaz grande, inestable y mal especificada puede ser peor que cinco de interfaces pequeñas y estables.

Cómo usan los profesionales estas ideas

Los diseñadores con experiencia rara vez citan los nombres de los niveles en las revisiones, pero los aplican constantemente: cuestionan cada parámetro que actúa como indicador, buscan tablas compartidas y estado global, preguntan cuál es el único propósito de cada módulo y observan qué módulos cambian juntos en el control de versiones. En los límites entre equipos y servicios son estrictos, porque allí el acoplamiento es caro de deshacer; dentro de un módulo pequeño del que es propietaria una sola persona son flexibles, porque allí es barato.

También hacen visible el acoplamiento. Un grafo de dependencias generado a partir del código, un recuento de Ca y Ce por componente y una lista de las tablas de base de datos a las que accede más de un servicio están entre las vistas más informativas de una arquitectura, y todas pueden generarse automáticamente y seguirse en la compilación. La siguiente sección pasa al principio que mantiene débil el acoplamiento desde el principio: ocultar las decisiones tras interfaces.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué una función con un parámetro de modo que selecciona comportamientos completamente distintos es un ejemplo de acoplamiento de control?

Porque quien llama tiene que saber cómo está organizado por dentro quien es llamado y elegir su rama interna; cambiar la lógica de quien es llamado puede obligar a cambiar a quienes lo llaman.

▼ Un módulo tiene Ca = 12 y Ce = 1. ¿Qué sugiere esto?

Muchos módulos dependen de él y él depende de poco, así que cambiarlo es arriesgado y los cambios deben ser escasos y cuidadosos; su interfaz merece una estabilidad especial.

▼ Dos servicios se comunican solo mediante eventos, pero siempre deben lanzarse juntos. ¿Qué tipo de acoplamiento queda?

El acoplamiento de despliegue, normalmente causado por cambios no compatibles con versiones anteriores en el evento compartido o por acuerdos semánticos ocultos.

▼ Indica un costo de sustituir una llamada directa por un evento.

Los resultados pasan a tener consistencia eventual en lugar de inmediata; además, ganas un broker que operar, un esquema de eventos que versionar y un flujo más difícil de rastrear.

▼ Inténtalo: Booking depende de Seats, Payments y Email. Refunds depende de Payments y Email. Reports depende de Seats y Payments. Calcula Ca y Ce de Payments y Booking. ¿Cuál es más arriesgado cambiar y cuál está más expuesto?

Payments: Ca = 3 (Booking, Refunds, Reports), Ce = 0. Booking: Ca = 0, Ce = 3 (Seats, Payments, Email). Payments es más arriesgado de cambiar, porque tres módulos notan cualquier cambio en su interfaz, así que mantén esa interfaz estrecha y estable. Booking está más expuesto, ya que un cambio en cualquiera de sus tres dependencias puede romperlo, pero cambiar el propio Booking es seguro, porque nada depende de él. Ese reparto del riesgo es saludable.

FUENTES

- Martin Fowler, Reducing Coupling (IEEE Software, 2001): https://martinfowler.com/ieeeSoftware/coupling.pdf
- W. P. Stevens, G. J. Myers and L. L. Constantine, Structured Design (IBM Systems Journal, 1974): https://doi.org/10.1147/sj.132.0115
- Microservices: database per service pattern (microservices.io): https://microservices.io/patterns/data/database-per-service.html

- Martin Fowler, What do you mean by Event-Driven?: https://martinfowler.com/articles/201701-event-driven.html
- Law of Demeter: general formulation (Northeastern University): https://www2.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/general-formulation.html
- Law of Demeter: object formulation (Northeastern University): https://www2.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/object-formulation.html
- Object-oriented programming: an objective sense of style (record): https://scilit.com/publications/8b7cd3e2eaafe26d184e99c8c9e37165
- Law of Demeter (Wikipedia, for authors and project): https://en.wikipedia.org/wiki/Law_of_Demeter
- Fowler: Mocks Aren't Stubs: https://martinfowler.com/articles/mocksArentStubs.html
- Fowler: TellDontAsk: https://martinfowler.com/bliki/TellDontAsk.html
- When did people favor composition over inheritance?: https://sicpers.info/2025/11/when-did-people-favor-composition-over-inheritance
- The Go FAQ: https://go.dev/doc/faq

SECCIÓN 3.3 DE 3.8 · INTERMEDIO · 26 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Ocultación de información, abstracción e interfaces

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.3 Organizar el código
- 0.6 API e integración
- 3.1 Modularidad

Términos anteriores que utiliza: Módulo, API (interfaz de programación de aplicaciones), Clase y objeto, Cambio incompatible (breaking change)

LA IDEA EN BREVE

Cuando envías un paquete por correo, lo entregas en el mostrador y recibes un número de seguimiento. No sabes, ni te importa, si viaja en furgoneta, tren o avión, ni por qué centro de distribución pasa. El servicio postal puede cambiar todo eso mañana sin avisarte, porque el trato en el mostrador sigue siendo el mismo. Los módulos de software funcionan mejor como ese mostrador: ofrecen una promesa pequeña y clara, y ocultan cómo la cumplen. Los problemas empiezan cuando los clientes llegan a depender de detalles que nadie prometió, como que el paquete siempre llegue antes de las diez.

LO QUE NECESITAS SABER

- La ocultación de información (Parnas, 1972) significa que cada módulo oculta una decisión de diseño difícil o que probablemente cambie detrás de una interfaz que revela lo menos posible.
- El encapsulamiento es un mecanismo del lenguaje (campos privados, módulos, control de acceso); la ocultación de información es un principio de diseño sobre qué decisiones ocultar. Se puede encapsular sin ocultar nada útil.
- Una interfaz es un contrato; el diseño por contrato (Bertrand Meyer) lo hace preciso con precondiciones, postcondiciones e invariantes, y atribuye la culpa de cada fallo al llamante o al proveedor.
- Todas las abstracciones no triviales tienen fugas en algún grado (Joel Spolsky, 2002), así que los buenos diseñadores exponen con honestidad las características de fallo, costo y rendimiento en lugar de fingir que no existen.
- Los módulos profundos de Ousterhout ofrecen mucha funcionalidad detrás de una interfaz pequeña; los módulos superficiales añaden el costo de una interfaz sin ocultar gran cosa.

> - La ley de Hyrum dice que de todo comportamiento observable de una API muy utilizada acabará dependiendo alguien, por lo que las API estables necesitan superficies mínimas, versionado explícito, cambios aditivos y pruebas de contrato.

El principio: ocultar lo que probablemente cambie

En la Sección 3.1, el ejemplo KWIC de David Parnas mostró que descomponer un programa por pasos de procesamiento repartía cada decisión de diseño entre varios módulos. Su alternativa, la ocultación de información, da a cada módulo una decisión que mantener en secreto: cómo se almacenan las líneas, cómo se calculan los desplazamientos, cuándo se ordena. La interfaz del módulo se elige para revelar lo menos posible sobre esa decisión. Todo lo demás del sistema depende solo de la interfaz, así que la decisión puede cambiar sin que el resto lo note.

La lista de Parnas de lo que merece la pena ocultar sigue siendo una buena lista de comprobación medio siglo después: las estructuras de datos y cómo se almacenan; el orden en que se realizan las operaciones; los detalles del hardware, los dispositivos y los sistemas operativos; los formatos de entrada y salida; y los algoritmos que podrían sustituirse por otros mejores. Los añadidos modernos siguen la misma lógica: qué proveedor externo se usa, qué base de datos, qué intermediario de mensajes (message broker), qué versión de una API externa y qué reglas de negocio están en revisión.

El principio se aplica de forma idéntica a cualquier escala. Una clase oculta sus campos; un paquete oculta sus módulos internos; un servicio oculta su base de datos, su lenguaje y su despliegue; la API pública de una organización oculta qué servicios existen detrás de ella. En todos los casos, lo que está oculto puede cambiar a bajo costo, y lo que está expuesto se vuelve costoso de cambiar porque otros dependen de ello.

El encapsulamiento no es ocultación de información

Los dos términos se usan a menudo como sinónimos, pero significan cosas distintas. El encapsulamiento es un mecanismo: agrupar los datos con las operaciones que actúan sobre ellos y restringir el acceso directo, mediante campos privados, límites de módulo o modificadores de acceso (Sección 0.3). La ocultación de información es un principio de diseño: decidir *qué* decisiones mantener en secreto y diseñar la interfaz para que no las revele. El encapsulamiento es una de las herramientas que usas para ocultar información; no decide por ti qué ocultar.

Es fácil encapsular a la perfección y aun así no ocultar nada. Una clase con campos privados y un getter y un setter públicos para cada uno de ellos está totalmente encapsulada, pero todos los que la llaman saben exactamente qué campos existen y de qué tipo son. Si cambias la

representación interna, todos los llamantes se rompen. También ocurre lo contrario: Python no impone ninguna privacidad, pero un módulo de Python bien diseñado puede ocultar sus decisiones con mucha eficacia mediante un conjunto de funciones elegido con cuidado.

```
# Encapsulated, but the representation leaks through the interface
class Price:
    def __init__(self, pounds: float):
        self._pounds = pounds
    def get_pounds(self) → float:          # callers now depend on 'float pounds'
        return self._pounds
    def set_pounds(self, value: float):
        self._pounds = value

# Information hiding: callers ask for what they need, not for the field
class Money:
    def __init__(self, minor_units: int, currency: str):
        self._minor, self._currency = minor_units, currency
    def add(self, other: "Money") → "Money": ...
    def allocate(self, ratios: list[int]) → list["Money"]: ...
    def format(self, locale: str) → str: ...
# Switching from pence to a decimal type later changes only this class.
```

Ambas clases están encapsuladas; solo la segunda oculta la decisión sobre cómo se almacena un precio.

SI ESTO ES NUEVO PARA TI

Paso a paso: campos privados que no ocultan nada

La idea que requiere más atención aquí es que una clase puede mantener todos sus datos privados y aun así no ocultar nada útil. El encapsulamiento guarda los datos bajo llave; la ocultación de información decide qué decisión de diseño no debe revelar la interfaz. Sigue las dos clases anteriores a través de un cambio.

- El primer diseño de la tienda es la clase Price. Su importe está en un campo privado, así que está correctamente encapsulada, pero get_pounds entrega el importe como un número de coma flotante en libras, y set_pounds recibe uno.
- El proceso de pago, las facturas y los reembolsos llaman a get_pounds y hacen sus propias cuentas con el resultado: sumar precios o repartir un reembolso entre varios artículos.
- El Capítulo 0 explicó que los números de coma flotante son solo aproximados. Algunos totales empiezan a salir con un penique de error tras el redondeo, así que el equipo decide guardar el dinero en peniques enteros.
- Cambiar el campo privado es fácil, pero no arregla nada: get_pounds sigue entregando números de coma flotante, y la aritmética que falla está en cada llamante. Para arreglarlo, el equipo tiene que cambiar la interfaz y todos los llamantes que hacen cuentas. El campo privado ocultaba dónde se guardaba el número, pero no la decisión que importaba: que el dinero es un número de coma flotante en libras.
- En el segundo diseño, Money, los llamantes nunca reciben el número en bruto. Piden a Money que sume dos importes, que reparta uno según unas proporciones (allocate) o que se formatee para mostrarse. La aritmética de los importes está dentro de Money, así que un cambio de representación solo afecta a esa clase.
- **La lección:** los campos privados son la cerradura; la ocultación de información es elegir qué se guarda detrás. Primero identifica la decisión con más probabilidades de cambiar y luego comprueba que ninguna operación de la interfaz la revele.

La abstracción y las interfaces como contratos

Una abstracción es una vista simplificada de algo que conserva lo que importa para un propósito y deja fuera el resto. Un archivo es una abstracción sobre bloques de disco, cachés y controladores; un *pago* es una abstracción sobre una red de tarjetas, la API de un proveedor y el proceso de liquidación de un banco. La abstracción y la ocultación de información son dos caras del mismo acto: elegir una abstracción decide lo que muestra la interfaz; ocultar el resto hace que sea seguro confiar en la abstracción.

Por tanto, una interfaz es un contrato entre un módulo y sus usuarios. Su parte formal es lo que puede ver un comprobador de tipos o un esquema de API: nombres de operaciones, tipos de parámetros y de retorno, tipos de error, nombres de campos en un cuerpo JSON. Su parte informal es igual de vinculante: qué hace la operación, sus efectos secundarios, si puede reintentarse con seguridad, cuánto tarda, qué errores son posibles y qué significan. John Ousterhout insiste en que la parte informal solo puede recogerse en la documentación, y en que, si los usuarios tienen que leer el código de un módulo para aprender a usarlo, el módulo no ofrece una abstracción real.

Diseño por contrato

El diseño por contrato, desarrollado por Bertrand Meyer en la década de 1980 con el lenguaje Eiffel y descrito en su libro *Object-Oriented Software Construction* y en su artículo de 1992 *Applying 'Design by Contract'*, hace preciso el contrato. Cada operación tiene una precondición, que quien llama debe hacer verdadera antes de llamar; una postcondición, que la operación garantiza al terminar si se cumplía la precondición; y cada clase o módulo tiene invariantes, condiciones que se cumplen entre cada par de operaciones. El contrato asigna la responsabilidad: si se incumple una precondición, el error está en quien llama; si se incumple una postcondición, está en el proveedor.

Meyer argumentó en contra del hábito opuesto, la programación defensiva, en la que cada rutina lo comprueba todo por si alguien la llama mal. Las comprobaciones redundantes añaden código, complejidad y nuevas vías de fallo. En cambio, cada condición debe ser responsabilidad clara de una de las partes. Una rutina con una precondición exigente es más sencilla; una rutina con una precondición tolerante es más fácil de llamar; la elección es una decisión de diseño que se toma de forma explícita. En los límites de un sistema, donde la entrada procede de usuarios no fiables o de otras organizaciones, la validación no es opcional: ahí el contrato dice que el propio módulo del límite lo comprueba todo (Capítulo 8).

```
class Account:
    """Invariant: balance_pence ≥ -overdraft_limit_pence."""

    def withdraw(self, amount_pence: int) → None:
        # Preconditions (the caller's obligations)
        if amount_pence ⩽ 0:
            raise ValueError("amount must be positive")
        if self.balance_pence - amount_pence < -self.overdraft_limit_pence:
            raise InsufficientFunds(self.id)
        old = self.balance_pence
        self.balance_pence -= amount_pence
        # Postcondition (the supplier's guarantee)
        assert self.balance_pence == old - amount_pence
        assert self.balance_pence ≥ -self.overdraft_limit_pence  # invariant
```

Un contrato por escrito: precondiciones comprobadas a la entrada, y la postcondición y el invariante enunciados y verificados con aserciones.

Las mismas tres ideas se aplican a escala de servicios. Una especificación de API, como un documento OpenAPI, enuncia las precondiciones (campos obligatorios, formatos, autenticación), las postcondiciones (la respuesta y el cambio de estado, como *existe un pedido con estado PLACED*) y los invariantes en los que puede confiar un cliente (*el total de un pedido nunca cambia después del pago*). Las pruebas de contrato dirigidas por el consumidor, en las que cada cliente publica las expectativas de las que depende y el proveedor las ejecuta en su compilación, comprueban esos contratos automáticamente.

Abstracciones con fugas

En noviembre de 2002, Joel Spolsky publicó *The Law of Leaky Abstractions*, que afirma que "Todas las abstracciones no triviales tienen, en algún grado, fugas." Una abstracción con fugas (leaky abstraction) es aquella cuyos detalles ocultos siguen afectando a sus usuarios. TCP presenta un flujo fiable sobre una red no fiable, hasta que se corta un cable y ya no puede. SQL te permite decir lo que quieres, no cómo conseguirlo, hasta que dos consultas lógicamente equivalentes difieren en velocidad por un factor de mil y tienes que leer el plan de consulta. Un mapeador objeto-relacional oculta SQL hasta que un bucle lanza sin hacer ruido una consulta por fila.

La consecuencia para los arquitectos no es que las abstracciones sean malas, sino que una interfaz debe ser honesta sobre los detalles que se van a filtrar. El ejemplo clásico es la llamada remota que parece exactamente una llamada a una función local. Las llamadas de red son órdenes de magnitud más lentas, pueden agotar el tiempo de espera, pueden tener éxito en el servidor mientras se pierde la respuesta y pueden fallar con independencia de quien llama; la Sección 6.1 enumera estas cuestiones como las falacias de la computación distribuida. Una buena interfaz remota lo deja ver: acepta un timeout, devuelve errores que

distinguen *rechazado* de *resultado desconocido* y acepta una clave de idempotencia para que los reintentos sean seguros. Ocultar la red por completo es la fuga que está esperando a ocurrir.

Módulos profundos y superficiales

A Philosophy of Software Design, de John Ousterhout (publicado por primera vez en 2018; segunda edición en 2021), ofrece una forma útil de juzgar una interfaz. Imagina cada módulo como un rectángulo: el borde superior es su interfaz, que cada usuario tiene que aprender, y el área es la funcionalidad que proporciona. Un módulo profundo tiene una interfaz pequeña sobre mucha funcionalidad, así que compensa muchas veces el costo de aprenderla. Un módulo superficial tiene una interfaz casi tan compleja como lo que hace, así que añade costo sin ocultar gran cosa.

Su ejemplo favorito de profundidad es la entrada y salida de archivos de Unix: unas pocas llamadas (open, read, write, seek, close) ocultan los sistemas de archivos, la disposición en disco, la caché, los permisos, los dispositivos y el acceso concurrente. Su ejemplo de superficialidad es una clase cuyos métodos se limitan a reenviar cada uno una llamada a otra clase, o una clase de lista enlazada cuya interfaz expone casi todo lo que hace. Ousterhout sostiene que el consejo habitual de mantener las clases y los métodos muy pequeños puede producir lo que él llama clasitis: muchas clases superficiales cuyas interfaces suman en conjunto más complejidad de la que tendría un módulo más profundo.

Módulos profundos y superficiales



El esquema de Ousterhout: la anchura del borde superior es la interfaz que deben aprender los usuarios; el área inferior es la funcionalidad que oculta.

No todo el mundo está de acuerdo con lo lejos que lleva Ousterhout esta idea. Los defensores de las clases y funciones pequeñas, como Robert C. Martin en *Clean Code*, sostienen que las unidades diminutas son más fáciles de nombrar, probar y reutilizar; Ousterhout y Martin debatieron la cuestión en un intercambio por escrito publicado en 2025. Ambas partes coinciden en el objetivo de fondo, que es reducir la complejidad que un lector tiene que tener en mente. A escala de arquitectura, la prueba de la profundidad es especialmente valiosa: un servicio cuya API refleja una a una las tablas de su base de datos, con crear, leer, actualizar y eliminar para cada una, es superficial y traslada toda la lógica real a sus clientes.

La ley de Hyrum: todo lo observable se convierte en el contrato

La ley de Hyrum, que lleva el nombre de Hyrum Wright, entonces ingeniero de software en Google, por iniciativa de su colega Titus Winters, y que se popularizó en *Software Engineering at Google* (2020), observa que, con un número suficiente de usuarios de una API, no importa lo que prometas en el contrato: alguien dependerá de todos los comportamientos observables de tu sistema. El orden en que una función devuelve los elementos de un conjunto no ordenado, la redacción exacta de un mensaje de error, un campo que siempre está presente aunque esté documentado como opcional, el hecho de que una llamada tarde 50 milisegundos: si los usuarios pueden verlo, algunos dependerán de ello.

La ley explica por qué la ocultación de información tiene que diseñarse, no esperarse. Cada comportamiento que expones por accidente pasa a formar parte del contrato de facto, y cambiarlo romperá algo a alguien, por muy claro que la documentación dijera que no estaba especificado. Algunos equipos aleatorizan a propósito el comportamiento no especificado, como el orden de iteración, para que nadie llegue a depender de él; Go aleatoriza el orden de iteración de los mapas por este motivo. Otros mantienen ligeras las cargas útiles de las respuestas, para que haya menos de lo que depender.

La ley de Postel y su crítica

Mientras que la ley de Hyrum describe lo que les ocurre a las interfaces, la ley de Postel te dice cómo comportarte en una de ellas. Jon Postel editó las primeras especificaciones de internet, y la idea aparece en el RFC 760, sobre el protocolo IP, en 1980: "una implementación debe ser conservadora en su comportamiento de envío y liberal en su comportamiento de recepción". El RFC 761, sobre TCP, dio la forma que suele citarse: "sé conservador en lo que haces, sé liberal en lo que aceptas de los demás". También se conoce como **principio de robustez**. Es una máxima de diseño, no una ley de la naturaleza. Encajaba con su época: muchas implementaciones independientes de especificaciones imperfectas tenían que funcionar juntas.

Sigue siendo útil en una forma concreta. El **lector tolerante (tolerant reader)** de Martin Fowler lo aplica a la integración: un consumidor debe leer solo lo que necesita e ignorar el resto. Si el servicio de Notificaciones lee dos campos de un evento `OrderPlaced` e ignora los desconocidos, Pedidos puede añadir campos sin romperlo, que es lo que hace que funcionen los cambios compatibles de la Sección 7.7.

La crítica se refiere a lo que ocurre con el tiempo. Combina un receptor tolerante con la ley de Hyrum y cada error aceptado se convierte en algo de lo que otros dependen. Supón que la integración de un restaurante envía los precios como `"12,50"` y que la plataforma lo acepta sin decir nada. El remitente nunca se entera de que está mal, otros socios lo copian y la rareza se convierte en una parte no escrita del contrato que cualquier reescritura debe reproducir.

El Internet Architecture Board lo expuso en el RFC 9413, *Maintaining Robust Protocols* (junio de 2023), un documento informativo, no un estándar, cuya aprobación fue amplia pero no unánime. Sostiene que tolerar entradas inesperadas conduce a la degradación del protocolo: los defectos permanecen ocultos, los errores se convierten en estándares de facto y las nuevas implementaciones deben copiar los bugs de las antiguas para interoperar. Su ejemplo son los servidores TLS que fallaban con extensiones vacías, lo que obligó a los clientes a sortearlos. Recomienda el mantenimiento activo de los protocolos, para que las ambigüedades se resuelvan en la especificación y no se absorban, y la *intolerancia virtuosa*: generar un error fatal ante una infracción de la especificación, para que los bugs salgan a la luz pronto.

Una regla práctica concilia ambas posturas. Sé tolerante solo con lo que el contrato dice que puedes ignorar, como los campos adicionales desconocidos. Sé estricto con todo lo que el contrato especifica: un campo obligatorio ausente, un tipo incorrecto o un valor no válido reciben un error claro, no una suposición. Esto importa sobre todo con el dinero, la identidad y la seguridad, donde un receptor que "hace lo que puede" puede interpretar un mensaje de forma distinta que el sistema que tiene al lado.

Diseñar API estables entre componentes

Entre componentes y servicios, las interfaces son lo más costoso de cambiar, porque cambiarlas exige coordinarse con cada consumidor. Las prácticas siguientes se derivan de todo lo anterior; la Sección 0.6 presentó las API y la Sección 7.7 trata en profundidad la compatibilidad de esquemas.

Prácticas para API estables de componentes y servicios

Práctica	Qué significa	Por qué funciona
Superficie mínima	Expón solo las operaciones y los campos que necesita un consumidor conocido; añade más cuando te lo pidan.	Menos de lo que depender, así que menos que romper (ley de Hyrum).
Modelar el dominio, no el almacenamiento	Ofrece operaciones como `place_order` y `cancel_order` en lugar de crear, actualizar y eliminar con forma de tabla.	El almacenamiento puede cambiar sin cambiar el contrato; el módulo sigue siendo profundo.
Cambio aditivo	Añade campos opcionales y operaciones nuevas; nunca renombres, elimines ni cambies el significado de los existentes dentro de una versión.	Los clientes antiguos siguen funcionando; los nuevos usan las partes nuevas.
Lector tolerante (tolerant reader)	Los clientes ignoran los campos que no entienden y no dependen del orden de los campos.	Los proveedores pueden añadir campos sin romper nada a nadie.
Versionado explícito	Introduce una nueva versión principal (en la URL, en una cabecera o en el esquema del evento) para los cambios incompatibles, y ejecuta la antigua y la nueva en paralelo durante un periodo anunciado.	Los consumidores migran a su propio ritmo; se evita el acoplamiento de despliegue.
Expandir y contraer (expand and contract)	Añade la forma nueva, migra a los consumidores, confirma mediante la monitorización que nadie sigue usando la forma antigua y luego elimínala, en lanzamientos separados.	Los cambios incompatibles se hacen sin un lanzamiento conjunto.
Pruebas de contrato	Los consumidores publican sus expectativas; la compilación del proveedor las ejecuta.	Los cambios incompatibles accidentales se detectan antes del despliegue.
Semántica de fallos honesta	Documenta los timeouts, la posibilidad de reintento, la idempotencia y el significado de los errores como parte del contrato.	Evita las peores fugas de la abstracción de red.

Ejemplo práctico: una interfaz de pago que oculta al proveedor

Una empresa de reparto de comida cobra los pagos a través del Proveedor A. Finanzas quiere añadir el Proveedor B, con comisiones más bajas en un país, y poder cambiar de proveedor en un día si uno sufre una caída. Hoy, el código del proceso de pago llama directamente al SDK del

Proveedor A en once lugares, usa sus códigos de error para decidir qué mostrar al cliente y guarda sus objetos de transacción en la tabla de pedidos. Cada una de esas cosas es una decisión que se ha filtrado.

El diseño introduce una interfaz `PaymentGateway` que pertenece a la tienda, no a ningún proveedor. Se expresa en los términos propios de la tienda: importes como enteros en unidades menores con una divisa, el identificador de pedido de la tienda, una clave de idempotencia y tres resultados. Cada proveedor recibe un adaptador que implementa la interfaz y traduce hacia y desde la API, los códigos de error y las peculiaridades de ese proveedor. La configuración elige el adaptador por país. Es el mismo movimiento que el intermediario de la Sección 2.6, visto ahora como ocultación de información: la decisión oculta es *qué proveedor, y cómo se le llama*.

```
from typing import Protocol, Literal
from dataclasses import dataclass

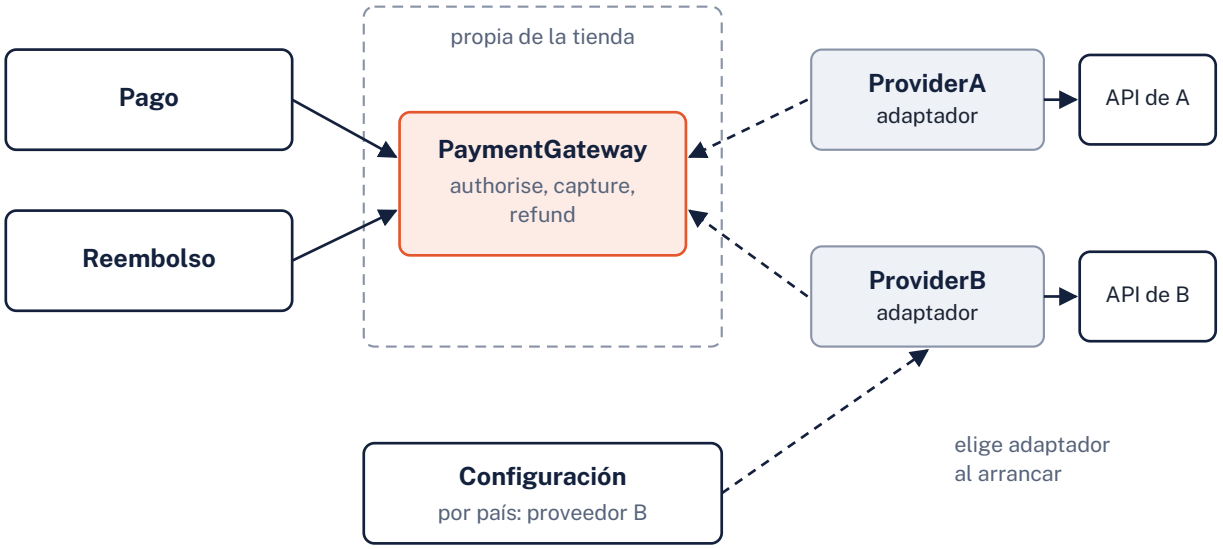
@dataclass(frozen=True)
class PaymentResult:
    outcome: Literal["approved", "declined", "unknown"]  # unknown: safe to retry
    reference: str | None        # our reference, not the provider's object

class PaymentGateway(Protocol):
    def authorise(self, order_id: str, amount_minor: int, currency: str,
                  idempotency_key: str) → PaymentResult: ...
    def capture(self, reference: str) → PaymentResult: ...
    def refund(self, reference: str, amount_minor: int) → PaymentResult: ...

class ProviderBAdapter:            # implements PaymentGateway
    def authorise(self, order_id, amount_minor, currency, idempotency_key):
        resp = self._client.create_charge(value=amount_minor / 100,   # B wants decimals
                                          currency=currency, key=idempotency_key)
        return PaymentResult(_map_status(resp.code), _store_ref(resp.id))
```

La interfaz pertenece a la tienda y habla su idioma; solo los adaptadores saben que existe un proveedor.

La interfaz de pago y sus adaptadores



Quien llama solo ve la interfaz; proveedores, códigos de error, reintentos y reglas de divisa quedan en los adaptadores
Flechas continuas: depende o llama. Discontinuas: implementa o selecciona.

Los llamantes dependen de una interfaz que pertenece a la tienda; los adaptadores la implementan, y la configuración elige cuál se ejecuta.

EJEMPLO PRÁCTICO

Lo que oculta la interfaz, y lo que deliberadamente no oculta

Contrastar el diseño con las ideas de esta sección.

Decisión oculta	Qué proveedor, su SDK y su versión de API, sus códigos de error, sus reglas de reintento, sus importes decimales o enteros, sus identificadores de transacción.
Contrato (precondiciones)	Importe positivo en unidades menores, una divisa ISO 4217 admitida y una clave de idempotencia única por cada intento de pago.
Contrato (postcondiciones)	`approved` significa que los fondos están reservados y se devuelve una referencia; `declined` significa que no se ha movido dinero; `unknown` significa que no se pudo determinar el resultado y que la misma llamada puede reintentarse con la misma clave.
No ocultación deliberada	La interfaz expone `unknown` en lugar de fingir que los pagos siempre tienen éxito o fallan: la fuga de la red forma parte del contrato, no es una sorpresa.
Costo de un nuevo proveedor	Antes: cambios en 11 puntos de llamada más la tabla de pedidos. Después: una nueva clase adaptadora, sus pruebas y una entrada de configuración.
Profundidad	Tres operaciones ocultan una cantidad considerable de lógica específica de cada proveedor: profundo, no superficial.

Cuidado con la trampa del mínimo común denominador: si el Proveedor B ofrece una función útil de la que A carece, como el pago a plazos, o la interfaz incorpora una capacidad opcional o la función no estará disponible. Las abstracciones sobre proveedores siempre sacrifican parte de los puntos fuertes de cada proveedor a cambio de la libertad de cambiar.

La ley de Hyrum sigue aplicándose: si el proceso de pago empieza a depender de la latencia habitual del Proveedor A, un Proveedor B más lento lo pondrá en evidencia. Incluye los timeouts en el contrato.

ATENCIÓN

Errores comunes

- Getters y setters para cada campo: encapsulado, pero con la representación totalmente expuesta.
- Interfaces con el nombre de la implementación actual, como `StripePaymentService` o `S3FileStore`, que anulan el sentido de la interfaz.
- Capas de paso que añaden una interfaz pero no ocultan nada, lo que hace el sistema más superficial y más largo de leer.
- Ocultar los fallos: llamadas remotas disfrazadas de locales, sin timeout y sin forma de distinguir un resultado desconocido de un rechazo.
- Devolver objetos internos, como filas de la base de datos o la respuesta de un proveedor, a través de una API pública, lo que sin hacer ruido los convierte en parte del contrato.

Cómo lo aplican los profesionales

Los diseñadores con experiencia empiezan una interfaz a partir de las necesidades de quienes la llaman, escrita en el lenguaje de quien llama, y solo después diseñan la implementación. Mantienen una lista de las decisiones que oculta cada módulo importante y revisan las nuevas interfaces contrastándolas con ella. Tratan cada API pública como un producto con un ciclo de vida (propuesta, revisión, versionado, retirada), porque, según la ley de Hyrum, todo lo que se publica acabará usándose. Y aceptan que las abstracciones tienen fugas, así que hacen explícitas esas fugas: los presupuestos de latencia, la semántica de los errores y las reglas de reintento figuran en el contrato.

La ocultación de información te dice qué debe guardarse para sí cada módulo. No te dice en qué dirección deben apuntar las dependencias entre módulos, ni qué ocurre cuando forman bucles. Ese es el tema de la Sección 3.4.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué una clase con campos privados y getters y setters públicos para todos ellos puede fallar igualmente en la ocultación de información?

Porque la interfaz revela la representación interna exacta, así que quienes la llaman dependen de ella y cualquier cambio en la representación los rompe; hay encapsulamiento, pero no se oculta ninguna decisión.

▼ **En el diseño por contrato, ¿de quién es la culpa si se viola una precondición?**

De quien llama, porque cumplir la precondición es su obligación; una postcondición violada indica un error en el proveedor.

▼ **¿Qué hace que un módulo sea profundo en el sentido de Ousterhout?**

Ofrece mucha funcionalidad detrás de una interfaz pequeña y sencilla, así que el costo de aprender la interfaz es bajo en comparación con lo que oculta.

▼ **¿Por qué la ley de Hyrum aboga por respuestas de API pequeñas?**

Porque algún cliente acabará dependiendo de cada campo o comportamiento observable, así que todo lo que se expone se vuelve difícil de cambiar; exponer menos deja más libertad.

▼ **Inténtalo: Critica esta interfaz de notificaciones: sendSms(phone, text, vendorRetryCount) devuelve un objeto VendorXResponse que quienes llaman inspeccionan en busca de códigos de error del proveedor. ¿Qué filtra y qué cambiarías, sin dejar de ser honesto sobre los fallos?**

Filtra la configuración de reintentos, el tipo de respuesta y los códigos de error del proveedor, de modo que cambiar de proveedor rompería a todos los llamadores. Define una interfaz propiedad de tu empresa, como send_message(recipient, text), que devuelva sent, rejected o try-later. Oculta los reintentos, el SDK y los códigos en un adaptador. No finjas que no puede fallar: el contrato debe indicar que la entrega puede retrasarse o fallar y con qué rapidez responde. Las abstracciones tienen fugas, así que expón esos hechos con honestidad.

FUENTES

- Joel Spolsky, The Law of Leaky Abstractions (2002): https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/
- Hyrum's Law: https://www.hyrumslaw.com/
- Bertrand Meyer, Applying 'Design by Contract' (IEEE Computer, 1992): https://se.inf.ethz.ch/~meyer/publications/computer/contract.pdf
- John Ousterhout, A Philosophy of Software Design (book page): https://web.stanford.edu/~ouster/cgi-bin/book.php
- Martin Fowler, Tolerant Reader: https://martinfowler.com/bliki/TolerantReader.html
- RFC 9413: Maintaining Robust Protocols: https://www.rfc-editor.org/rfc/rfc9413.html
- RFC 761: Transmission Control Protocol: https://www.rfc-editor.org/rfc/rfc761.html

SECCIÓN 3.4 DE 3.8 · INTERMEDIO · 25 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Gestionar las dependencias: dirección, ciclos y estabilidad

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.3 Organizar el código
- 3.1 Modularidad
- 3.2 Acoplamiento y cohesión

Términos anteriores que utiliza: Acoplamiento aferente (Ca), Acoplamiento eferente (Ce), Inversión de control, Función de aptitud (fitness function), Interfaz

LA IDEA EN BREVE

Piensa en un edificio. Los pisos superiores descansan sobre los inferiores, y los cimientos no descansan sobre nada de lo que tienen encima. Puedes redecorar el último piso sin tocar los cimientos, pero si excavas bajo los cimientos, todo lo que hay encima corre peligro. Ahora imagina dos pisos que se sostienen el uno al otro: nunca podrías reconstruir ninguno de los dos. Los módulos de software se apoyan unos en otros de la misma manera. Las buenas estructuras colocan arriba las partes que cambian a menudo, apoyadas en partes sólidas que rara vez cambian, y nunca dejan que dos partes se sostengan mutuamente en un bucle.

LO QUE NECESITAS SABER

- Una dependencia apunta del módulo que necesita algo al módulo que necesita; los cambios fluyen en sentido contrario a las flechas, por lo que la dirección de las dependencias decide a quién afectan los cambios de quién.
- El Principio de dependencias acíclicas prohíbe los ciclos entre componentes, porque todos los componentes de un ciclo deben compilarse, probarse y lanzarse juntos.
- El Principio de dependencias estables dice que hay que depender en la dirección de la estabilidad; la inestabilidad es I = Ce / (Ca + Ce), de 0 (solo dependencias entrantes) a 1 (solo salientes).
- El Principio de abstracciones estables dice que los componentes estables deben ser abstractos; el grado de abstracción A es el número de tipos abstractos dividido entre el total de tipos, y la distancia a la secuencia principal es D = |A + I - 1|.
- Los ciclos se rompen mediante la inversión de dependencias (el módulo del que se depende es dueño de una interfaz que el otro implementa) o extrayendo la parte compartida a un componente nuevo y más estable.

- La organización en capas estricta solo permite llamadas a la capa inmediatamente inferior, y la relajada, a cualquier capa inferior; ambas, y cualquier otra regla de dependencia, deben comprobarse automáticamente con herramientas como ArchUnit, dependency-cruiser o import-linter.

Por qué importa la dirección

Existe una dependencia cuando un módulo necesita otro para compilarse, ejecutarse o comportarse correctamente. Dibújala como una flecha del módulo que depende al módulo del que depende: de Pago a Precios. La flecha te dice algo importante sobre el cambio. Si Precios cambia su interfaz o su comportamiento, puede que Pago tenga que cambiar; si Pago cambia, a Precios no le afecta. Los cambios viajan en sentido contrario a las flechas. Así que la dirección de cada dependencia es una decisión sobre quién se verá afectado por los cambios de quién.

A los arquitectos les importan dos tipos de dirección que es fácil confundir. El *flujo de control* es quién llama a quién en tiempo de ejecución. La *dependencia de código fuente* es quién importa, referencia o se compila contra quién. Normalmente apuntan en el mismo sentido, pero no tienen por qué. Un framework llama a tu código sin depender de él (la Sección 0.3 lo llamó inversión de control), y, con una interfaz, un módulo de alto nivel puede llamar a uno de bajo nivel mientras la dependencia de código fuente apunta en sentido contrario. Esa libertad para elegir la dirección de la dependencia de código fuente con independencia del flujo de control es la herramienta más potente de esta sección.

Entre servicios, la dirección se manifiesta como conocimiento. Si Pedidos llama a la API de Facturación, Pedidos depende de Facturación y tiene que cambiar cuando cambia la API de Facturación. Si Pedidos publica un evento que Facturación consume, la dependencia se invierte: Facturación depende del contrato de eventos de Pedidos, y Pedidos no sabe que Facturación existe (Sección 3.2). Por tanto, elegir entre una llamada y un evento es en parte una elección sobre la dirección de la dependencia.

El Principio de dependencias acíclicas

Robert C. Martin formuló tres principios para las dependencias entre componentes, primero en artículos y ponencias a mediados de la década de 1990 y después en *Agile Software Development: Principles, Patterns, and Practices* (2002) y *Clean Architecture* (2017). El primero, el Principio de dependencias acíclicas (ADP), dice que el grafo de dependencias entre componentes no debe contener ciclos: debe ser un grafo acíclico dirigido.

Un ciclo de dependencias, como que Pedidos dependa de Notificaciones, Notificaciones dependa de Clientes y Clientes dependa de Pedidos, convierte tres componentes en uno. Ninguno puede compilarse ni probarse sin los demás; un cambio en cualquiera puede

propagarse por el bucle hasta volver a él mismo; ninguno puede lanzarse por separado; y nadie puede decir cuál compilar primero. Martin presenta el principio con lo que él llama el *síndrome de la mañana siguiente*: te vas a casa con código que funciona y vuelves al día siguiente para encontrarlo roto porque otra persona cambió algo de lo que dependes. Su remedio es dividir el sistema en componentes que se lanzan por separado, y eso solo funciona si las dependencias entre ellos no contienen ciclos. Los ciclos también hacen imposible extraer más adelante un componente como biblioteca o servicio, porque arrastra consigo todo el bucle.

El Java Platform Module System impone el ADP entre módulos: la Java Language Specification establece que es un error de compilación que un módulo dependa, directa o indirectamente, de sí mismo. La mayoría de los demás lenguajes permiten ciclos entre paquetes, así que los equipos los detectan con herramientas. Los ciclos entre clases dentro de un mismo componente pequeño son menos dañinos, y Martin Fowler, entre otros, los tolera ahí; el principio importa sobre todo entre componentes que construyen y lanzan personas distintas.

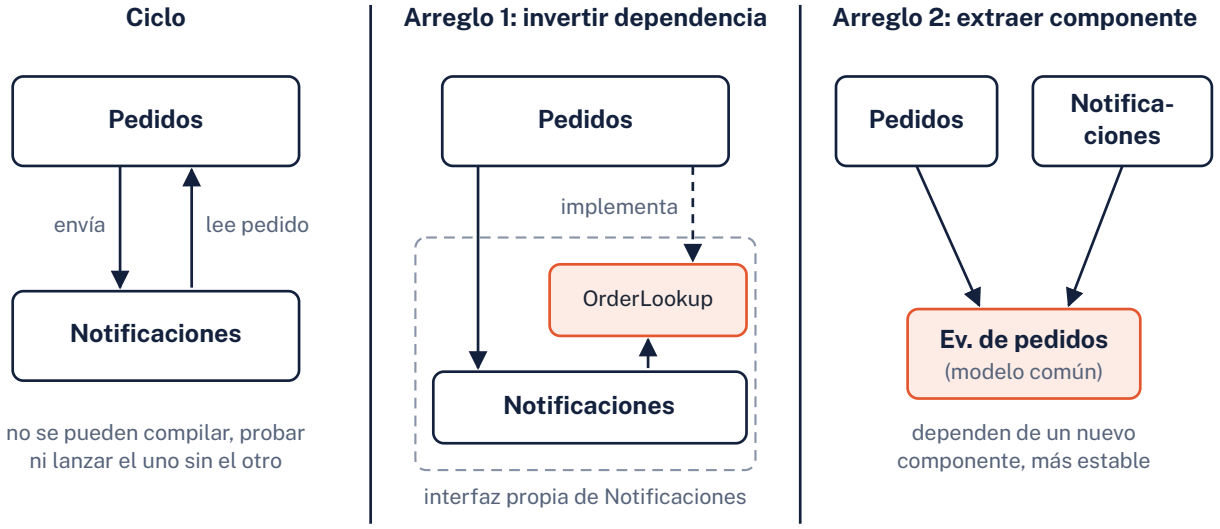
Romper ciclos

Todo ciclo puede romperse, y hay dos técnicas habituales. Ambas se muestran en el diagrama siguiente para un ciclo en el que Pedidos llama a Notificaciones para enviar una confirmación y Notificaciones vuelve a llamar a Pedidos para leer los detalles del pedido.

Inversión de dependencias. Notificaciones declara una interfaz que describe exactamente lo que necesita, por ejemplo `OrderLookup` con un solo método, y Pedidos la implementa. La flecha de Notificaciones a Pedidos apunta ahora en sentido contrario: Pedidos depende de Notificaciones (para llamarlo y para implementar su interfaz), y Notificaciones no depende de nada de Pedidos. Esto es la inversión de dependencias, que la Sección 3.5 trata como la D de SOLID. El detalle clave es la propiedad: la interfaz pertenece al módulo que la *usa*, no al que la implementa.

Extraer un componente nuevo. Averigua qué necesitan realmente las dos partes la una de la otra, a menudo un pequeño modelo o contrato compartido, y trasládalo a un componente nuevo del que dependan ambas. Aquí, Pedidos publica un tipo de evento definido en un componente de *eventos de pedidos*, y Notificaciones lee los detalles del pedido del evento en lugar de volver a llamar. Ambos dependen del nuevo componente y este no depende de ninguno, así que es más estable que cualquiera de los dos: exactamente lo que pide el siguiente principio.

Dos formas de romper un ciclo



Arreglo 1: Pedidos llama a Notificaciones, que expresa lo que necesita como interfaz; Pedidos la implementa.
Arreglo 2: lo que ambos necesitaban pasa a su propio componente; Notificaciones ya no lee de Pedidos.

Las flechas van del módulo que depende al módulo del que depende. Discontinua: implementa.

Un ciclo puede romperse invirtiendo una dependencia mediante una interfaz que pertenece a quien la usa, o extrayendo lo que comparten ambas partes a un componente nuevo.

SI ESTO ES NUEVO PARA TI

Paso a paso: la misma llamada, la flecha invertida

El punto en el que más gente tropieza aquí es que quién llama a quién cuando el programa se ejecuta y qué código menciona a qué otro son dos cosas distintas, y que una interfaz te permite elegir la segunda. "Romper ciclos" acaba de mostrar el arreglo para Pedidos y Notificaciones; sigue un correo electrónico de confirmación a través de él dos veces, una mientras se ejecuta y otra tal como está escrito.

- Antes del arreglo, Pedidos llamaba a Notificaciones para enviar la confirmación, y Notificaciones volvía a llamar a Pedidos para obtener los detalles del pedido. El código de cada módulo nombraba al otro: un ciclo de dependencias.
- Después del arreglo, obsérvalo en ejecución. Pedidos llama a Notificaciones como antes. Al arrancar, Notificaciones recibió un OrderLookup que en realidad es la propia implementación de Pedidos. Así que pedirle los detalles del pedido ejecuta el código de Pedidos, como antes: el flujo de control no ha cambiado.
- Ahora lee el código. Notificaciones solo nombra OrderLookup, que declara ella misma, y nunca nombra Pedidos. Así que, en tiempo de ejecución, la llamada va de Notificaciones al código de Pedidos, mientras que la flecha de código fuente apunta de Pedidos a Notificaciones.
- Esa diferencia es la ganancia. Ahora Notificaciones puede compilarse y probarse sin nada de Pedidos en la compilación, usando un pequeño OrderLookup falso que devuelve detalles de pedidos inventados.
- La propiedad es lo que hace que funcione. Si OrderLookup se hubiera declarado dentro de Pedidos, Notificaciones seguiría teniendo que nombrar el código de Pedidos para usarlo, y el ciclo se mantendría. La interfaz tiene que estar con Notificaciones, la parte que la necesita.
- **La lección:** la llamada en tiempo de ejecución y la dependencia de código fuente pueden apuntar en sentidos opuestos. Una interfaz que pertenece a quien la usa te permite orientar la dependencia como necesites, y esa es la inversión de dependencias a la que esta sección vuelve una y otra vez.

El Principio de dependencias estables y la inestabilidad

Algunos componentes cambian a menudo (interfaces de usuario, integraciones con socios, funciones nuevas); otros deberían cambiar rara vez (conceptos centrales del dominio, contratos compartidos, gestión del dinero). El Principio de dependencias estables (SDP) dice: depende en

la dirección de la estabilidad. Un componente pensado para ser fácil de cambiar no debería tener como dependiente a un componente difícil de cambiar, o también se volverá difícil de cambiar.

Martin mide la estabilidad a través de las dependencias, no de la frecuencia con que se edita el código. Un componente del que dependen muchos otros es *estable* en el sentido de que es costoso de cambiar: cada cambio corre el riesgo de romper a sus dependientes. Un componente que depende de muchos otros y no tiene dependientes es *inestable*: nada le impide cambiar, y sus dependencias pueden obligarle a hacerlo. Usando el acoplamiento aferente (Ca, entrante) y el acoplamiento eferente (Ce, saliente) de la Sección 3.2, la inestabilidad es:

I = Ce / (Ca + Ce)

I va de 0 a 1. I = 0 significa solo dependencias entrantes: máximamente estable, responsable ante muchos, sin depender de nada. I = 1 significa solo dependencias salientes: máximamente inestable, nada depende de él. El SDP, expresado como métrica, dice que I debería disminuir a lo largo de cada flecha de dependencia: cada componente debería depender de componentes con una I menor que la suya. Una dependencia de un componente estable (I = 0.1) a uno volátil (I = 0.8) es una señal de alerta; el arreglo habitual vuelve a ser la inversión de dependencias, colocando una interfaz en un lugar estable.

El Principio de abstracciones estables y la secuencia principal

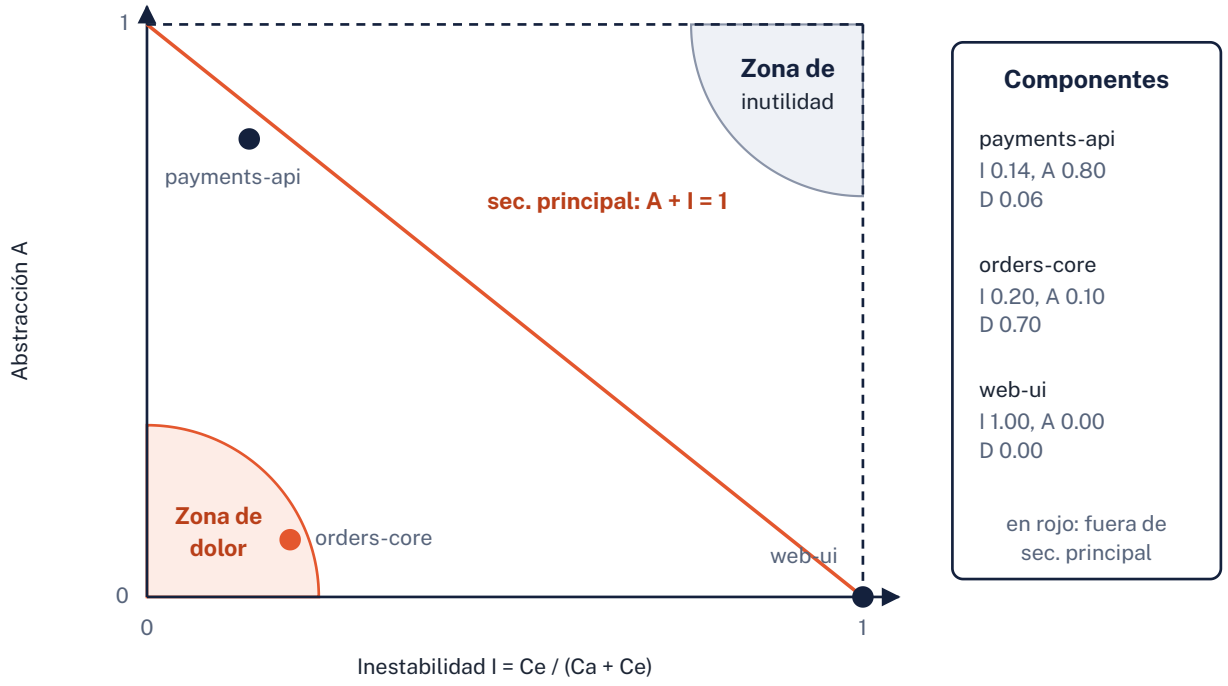
La estabilidad tiene un costo: un componente con una I cercana a 0 es difícil de cambiar. Si está lleno de código concreto, el sistema se vuelve rígido. El Principio de abstracciones estables (SAP) lo resuelve: un componente debe ser tan abstracto como estable. Los componentes estables deben consistir sobre todo en interfaces y tipos abstractos, para que su comportamiento pueda ampliarse con nuevas implementaciones en otro lugar sin modificarlos; los componentes inestables pueden ser concretos, porque de todos modos son fáciles de cambiar. El SDP y el SAP juntos equivalen al principio de inversión de dependencias aplicado a componentes enteros: las dependencias apuntan hacia la estabilidad, y la estabilidad va unida a la abstracción.

El grado de abstracción se mide como A = Na / Nc, donde Na es el número de clases abstractas e interfaces del componente y Nc el número total de clases. A = 0 significa totalmente concreto; A = 1 significa nada más que abstracciones. Representar A frente a I da un cuadrado. La línea que va de (I = 0, A = 1) a (I = 1, A = 0) es la secuencia principal: estable y abstracto en un extremo, inestable y concreto en el otro, con mezclas equilibradas entre ambos. La distancia a ella es:

D = |A + I - 1|

D va de 0 (sobre la línea) a 1 (en una esquina lejana). La esquina cercana a (0, 0) es la *zona de dolor*: estable, porque mucho depende de ello, y concreto, así que cualquier cambio es a la vez necesario y doloroso. Un esquema de base de datos que todos los módulos leen directamente vive aquí. Martin señala que algunos componentes están ahí sin causar daño, como una biblioteca de utilidades concreta y madura que nunca necesita cambiar; la zona solo duele en los componentes que son volátiles. La esquina cercana a (1, 1) es la *zona de inutilidad*: abstracciones de las que nada depende, normalmente interfaces que quedaron tras una refactorización o que se diseñaron para una extensión que nunca llegó.

La secuencia principal



Los componentes cercanos a la secuencia principal equilibran el grado de abstracción con la estabilidad; la zona de dolor contiene código estable y concreto, y la zona de inutilidad, abstracciones que nadie usa.

EJEMPLO PRÁCTICO

Ejemplo práctico: calcular I, A y D para tres componentes

Una tienda tiene tres componentes. Siguiendo la definición de Martin, Ca cuenta las clases de fuera del componente que dependen de clases de dentro, y Ce cuenta las clases de dentro que dependen de clases de fuera. Los recuentos proceden de una herramienta de dependencias.

payments-api (5 clases, 4 de ellas interfaces)	Ca = 6, Ce = 1. I = 1 / (6 + 1) = 0.14. A = 4 / 5 = 0.80. D = |0.80 + 0.14 - 1| = 0.06.
orders-core (20 clases, 2 abstractas)	Ca = 8, Ce = 2. I = 2 / (8 + 2) = 0.20. A = 2 / 20 = 0.10. D = |0.10 + 0.20 - 1| = 0.70.
web-ui (12 clases, ninguna abstracta)	Ca = 0, Ce = 9. I = 9 / (0 + 9) = 1.00. A = 0 / 12 = 0.00. D = |0 + 1 - 1| = 0.00.
Comprobar el SDP a lo largo de las flechas	web-ui (1.00) depende de orders-core (0.20) y de payments-api (0.14); orders-core (0.20) depende de payments-api (0.14). I disminuye a lo largo de cada flecha, así que se cumple el SDP.
Diagnóstico	orders-core tiene D = 0.70: estable (muchos dependientes) pero concreto. Si sus reglas cambian a menudo, cada cambio pone en riesgo ocho clases dependientes. Está en la zona de dolor.
Remedio y nuevo cálculo	Extrae los cuatro tipos que usan sus dependientes a orders-api (3 interfaces y 1 tipo de valor: A = 0.75; Ca es 8 o más, ya que ahora las clases que lo implementan también dependen de él, y Ce = 0, así que I = 0; D = |0.75 + 0 - 1| = 0.25). Las 16 clases concretas restantes pasan a ser orders-impl, del que solo depende el código de conexión del arranque (Ca = 1) y que depende de orders-api y de otros dos (Ce = 3): I = 3 / 4 = 0.75, A = 0, D = 0.25.

El remedio acerca mucho ambas partes a la secuencia principal (de 0.70 a 0.25 cada una). Los dependientes se apoyan ahora en un componente pequeño, estable y mayoritariamente abstracto, y el código concreto volátil está donde es barato cambiarlo.

Los números son heurísticas, no objetivos. Una D de 0.70 es un motivo para mirar, no la prueba de un problema; si orders-core nunca cambiara, no haría ningún daño donde está.

ATENCIÓN

Usar las métricas con sensatez

- Las herramientas difieren en lo que cuentan (clases, archivos, paquetes) y en si incluyen bibliotecas de terceros; compara solo números de la misma herramienta y la misma configuración.
- En lenguajes sin clases abstractas, como Python con tipado pato (duck typing), A es aproximado; cuenta los Protocols y las clases base abstractas, y trata la métrica como punto de partida para el debate.
- Las tendencias dicen más que las instantáneas: un componente cuya D crece versión tras versión se está metiendo en problemas.
- Las métricas pueden manipularse, por ejemplo añadiendo interfaces sin usar para subir A. La Sección 3.8 vuelve sobre la ley de Goodhart.

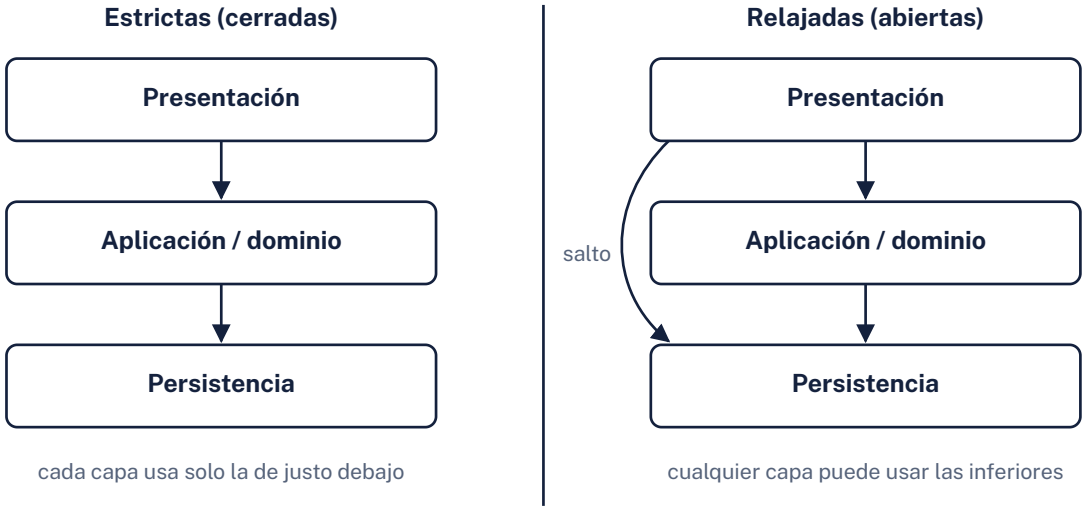
Organización en capas estricta y relajada

La organización en capas (layering) es la regla de dependencia más antigua y común: organizar los módulos en capas, como presentación, aplicación o dominio, y persistencia, y permitir que cada capa use solo las capas inferiores. Las capas inferiores no saben nada de las superiores, así que puede añadirse una nueva interfaz de usuario sin cambiar el dominio. La organización en capas es un estilo arquitectónico por derecho propio (Sección 5.2); aquí lo que interesa es la regla de dependencia que impone.

En la organización en capas **estricta** (cerrada), cada capa solo puede usar la capa inmediatamente inferior. Así, cada capa oculta todo lo que tiene debajo, y sustituir una capa inferior solo afecta a otra capa; el precio es el código de paso, en el que la capa intermedia se limita a reenviar llamadas a las que no tiene nada que añadir. En la organización en capas **relajada** (abierta), una capa puede usar cualquier capa inferior. Eso elimina el código repetitivo, pero significa que un cambio en la capa inferior puede afectar a todas las capas superiores. Muchos sistemas reales mezclan ambas y marcan como abiertas capas concretas, como una capa compartida de logging o de utilidades que todos pueden usar.

La organización en capas clásica tiene una debilidad conocida: la capa de dominio depende de la capa de persistencia, así que las reglas de negocio más importantes dependen de detalles de la base de datos. Las arquitecturas hexagonal, de cebolla y limpia (Sección 5.6) aplican la inversión de dependencias para corregirlo: el dominio es dueño de interfaces como `OrderRepository` y el código de persistencia las implementa, de modo que todas las dependencias de código fuente apuntan hacia dentro, hacia el dominio.

Organización en capas estricta y relajada



La organización en capas estricta solo permite llamadas a la capa inmediatamente inferior; la relajada también permite que una capa se salte las capas que tiene debajo.

Comprobar automáticamente las reglas de dependencia

Las reglas de dependencia escritas solo en una wiki se incumplen en cuestión de semanas. La forma fiable de mantenerlas es expresarlas como comprobaciones ejecutables que se ejecutan en la compilación y la hacen fallar cuando se viola una regla. Una comprobación así es una función de aptitud (fitness function) arquitectónica, una idea presentada en la Sección 2.8 y desarrollada en la Sección 11.3. Tres herramientas de código abierto muy usadas cubren los principales ecosistemas.

ArchUnit (Java y otros lenguajes de la JVM) te permite escribir reglas de arquitectura como pruebas unitarias normales: qué paquetes pueden acceder a cuáles, definiciones de capas y ausencia de ciclos entre porciones del código base. **dependency-cruiser** (JavaScript y TypeScript) analiza las importaciones y las valida contra reglas de un archivo de configuración, y su configuración inicial ya prohíbe las dependencias circulares. **import-linter** (Python) comprueba *contratos* declarados en un archivo de configuración; entre sus tipos de contrato están las importaciones prohibidas, las capas, la independencia entre módulos hermanos, los módulos protegidos y los hermanos acíclicos.

```
@ArchTest
static final ArchRule layers = layeredArchitecture()
    .consideringAllDependencies()
    .layer("Web").definedBy("..web..")
    .layer("Domain").definedBy("..domain..")
    .layer("Persistence").definedBy("..persistence..")
    .whereLayer("Web").mayNotBeAccessedByAnyLayer()
    .whereLayer("Domain").mayOnlyBeAccessedByLayers("Web", "Persistence")
    .whereLayer("Persistence").mayNotBeAccessedByAnyLayer();

@ArchTest
static final ArchRule noCycles =
    slices().matching("com.shop.(*)..").should().beFreeOfCycles();
```

ArchUnit (Java): una regla de capas y una comprobación de ciclos escritas como pruebas.

```
// .dependency-cruiser.js
module.exports = {
  forbidden: [
    { name: "no-circular", severity: "error",
      from: {}, to: { circular: true } },
    { name: "domain-not-to-db", severity: "error",
      comment: "Domain code must use repository interfaces, not the database",
      from: { path: "^src/domain" },
      to: { path: "^src/db" } },
  ],
};
```

dependency-cruiser (JavaScript y TypeScript): prohibir los ciclos e impedir que el dominio importe la capa de base de datos.

```
# .importlinter
[importlinter]
root_package = shop

[importlinter:contract:layers]
name = Web above domain above adapters
type = layers
layers =
    shop.web
    shop.domain
    shop.adapters

[importlinter:contract:features-independent]
name = Feature packages do not import each other
type = independence
modules =
    shop.catalogue
    shop.ordering
    shop.payments
```

import-linter (Python): un contrato de capas y un contrato de independencia, comprobados ejecutando lint-imports en el pipeline.

Observa que la regla de ArchUnit anterior permite que Persistence acceda a Domain pero no al revés: es la organización centrada en el dominio que acabamos de describir, no la organización en capas clásica. Reglas como estas son baratas de añadir, se ejecutan en segundos y convierten un diagrama de arquitectura en algo que la compilación defiende. Cuando un código base existente ya incumple una regla, la mayoría de las herramientas pueden registrar las infracciones actuales como línea base y fallar solo con las nuevas, de modo que la regla impide que las cosas empeoren desde el día en que se introduce.

GUÍA DE DECISIÓN

Guía de decisión: qué hacer con una dependencia no deseada

Úsala cuando falle una comprobación de dependencias o una revisión encuentre una dependencia que apunta en el sentido equivocado.

Q1 **¿Forma parte la dependencia de un ciclo entre componentes?**

- **Sí** Rompe primero el ciclo (ADP): sigue abajo para elegir cómo.
- **No** Pasa a la pregunta sobre la estabilidad.

Q2 **¿Necesita una de las partes solo un servicio pequeño y bien definido de la otra?**

- **Sí** Invierte la dependencia: haz que la parte que usa el servicio sea dueña de una interfaz, y que la otra parte la implemente.
- **No** Pasa a la siguiente pregunta.

Q3 **¿Dependen ambas partes de un modelo, un contrato o un fragmento de lógica compartidos?**

- **Sí** Extráelo a un componente nuevo del que dependan ambas, y haz que ese componente sea pequeño, estable y mayoritariamente abstracto.
- **No** Plantéate si los dos componentes son en realidad uno: fusiónalos y vuelve a dividirlos por un límite mejor (Sección 3.1).

Q4 **¿Depende un componente estable (I baja) de uno volátil (I alta)?**

- **Sí** Esto incumple el SDP. Coloca una interfaz en el componente estable, o en uno nuevo que sea estable, y traslada detrás de ella el código volátil.
- **No** La dirección es aceptable. Registra la regla en una función de aptitud para que siga así.

Cómo gestionan las dependencias los profesionales

Los arquitectos con experiencia tienen una imagen del grafo de dependencias entre componentes y la consultan con regularidad, generada a partir del código en lugar de dibujada a mano. Deciden qué componentes deben ser estables (conceptos del dominio, contratos compartidos) y cuáles volátiles (interfaces de usuario, integraciones), y hacen que las flechas apunten de lo volátil a lo estable. Tratan los ciclos nuevos como fallos de compilación, no como cuestiones de estilo. Y aplican el mismo razonamiento entre servicios: las bibliotecas compartidas y los esquemas de eventos compartidos son los componentes estables de un sistema distribuido, que merecen superficies mínimas y un versionado cuidadoso, mientras que las partes internas de cada servicio siguen siendo libres de cambiar.

Nada de esto exige calcular las métricas con precisión. Su valor está en las preguntas que obligan a hacerse: ¿en qué sentido debería apuntar esta flecha?, ¿qué les pasa a todos los que están aguas arriba si esto cambia?, ¿es este componente tan abstracto como exige el número de cosas que dependen de él? La Sección 3.5 continúa con los principios SOLID, que aplican las mismas ideas dentro de los componentes y entre ellos.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Por qué todos los componentes de un ciclo de dependencias deben lanzarse juntos?**

Cada uno depende, directa o indirectamente, de todos los demás, así que un cambio en cualquiera puede exigir cambios en el resto; ninguno puede compilarse ni probarse sin los demás.

▼ **Un componente tiene Ca = 3 y Ce = 9. ¿Cuál es su inestabilidad y qué significa?**

I = 9 / 12 = 0.75. Depende de mucho más de lo que depende de él, así que es relativamente inestable: fácil de cambiar, pero expuesto a los cambios de sus dependencias.

▼ **¿Qué hay en la zona de dolor y cuándo duele de verdad?**

Componentes que son a la vez estables (muchos dependientes) y concretos (pocas abstracciones). Duele cuando un componente así necesita cambiar a menudo; un componente concreto que nunca cambia hace poco daño ahí.

▼ **¿Cuál es la diferencia entre la organización en capas estricta y la relajada?**

En la organización en capas estricta, una capa solo puede usar la capa inmediatamente inferior; en la relajada, puede usar cualquier capa inferior.

▼ **Inténtalo: El componente pricing-rules tiene Ca = 8, Ce = 0 y 12 clases, ninguna abstracta. El componente reports tiene Ca = 2, Ce = 6 y 10 clases, 5 abstractas. Calcula I, A y D de cada uno y di cuál te preocupa más.**

pricing-rules: I = 0 / 8 = 0, A = 0, D = |0 + 0 - 1| = 1, de lleno en la zona de dolor: mucho depende de él y, sin embargo, es concreto, así que cada cambio resulta doloroso. reports: I = 6 / 8 = 0.75, A = 0.5, D = 0.25, razonablemente cerca de la secuencia principal. pricing-rules es lo preocupante. Extrae interfaces, para que los dependientes se apoyen en abstracciones estables. Las reglas que cambian a menudo hacen que esto sea urgente.

FUENTES

- Robert C. Martin, OO Design Quality Metrics: An Analysis of Dependencies (1994): https://bccdev.ime.usp.br/tccs/2010/lucianna-joao/arquivos/referencias/oodmetrics.pdf
- ArchUnit User Guide: https://www.archunit.org/userguide/html/000_Index.html
- Import Linter: contract types: https://import-linter.readthedocs.io/en/stable/contract_types/
- dependency-cruiser (GitHub): https://github.com/sverweij/dependency-cruiser
- The Java Language Specification, Chapter 7: Packages and Modules: https://docs.oracle.com/javase/specs/jls/se21/html/jls-7.html

SECCIÓN 3.5 DE 3.8 · INTERMEDIO · 26 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Los principios SOLID aplicados a los componentes

ANTES DE LEER ESTO

Esta sección se basa en:

- 3.2 Acoplamiento y cohesión
- 3.3 Ocultación de información
- 3.4 Dependencias

Términos anteriores que utiliza: Diseño por contrato, Precondición, Invariante, Ley de Hyrum, Principio de dependencias estables

LA IDEA EN BREVE

Piensa en un grupo de restaurantes bien gestionado. Cada cocina responde ante un solo jefe de cocina, de modo que nunca hay dos jefes dándole órdenes contradictorias. Se añaden platos nuevos a la carta sin reconstruir la cocina. Un chef suplente debe cocinar cada plato exactamente como lo haría el chef habitual, o los comensales lo notan. Cada proveedor recibe solo el formulario de pedido que necesita, no el manual completo de la empresa. Y la sede central fija los estándares, mientras que los proveedores se adaptan a ellos, y no al revés. SOLID son estos cinco hábitos, escritos para el software.

LO QUE NECESITAS SABER

- SOLID nombra cinco principios reunidos por Robert C. Martin: responsabilidad única, abierto-cerrado, sustitución de Liskov, segregación de interfaces e inversión de dependencias; cada uno se aplica a clases, componentes y servicios.
- Responsabilidad única significa un actor por módulo: el código que se cambia a solicitud de distintos grupos de personas debe separarse, y no significa que un módulo solo pueda hacer una cosa.
- La sustitución de Liskov exige compatibilidad de comportamiento: precondiciones no más fuertes, postcondiciones no más débiles e invariantes preservados, que es exactamente la prueba de una versión de API compatible hacia atrás.
- La segregación de interfaces aboga por interfaces de rol pequeñas en el código y, entre servicios, por API específicas para cada cliente, eventos acotados y bibliotecas pequeñas.
- La inversión de dependencias hace que los detalles dependan de abstracciones que pertenecen al lado de la política; importa más quién es el dueño de la interfaz que el hecho de que exista.

- REP, CCP y CRP deciden qué entra en un componente y tiran en direcciones opuestas; el equilibrio adecuado se desplaza de CCP hacia REP y CRP a medida que un sistema madura.
- Aplicado de forma dogmática, SOLID produce abstracciones superficiales y especulativas; funciona mejor como vocabulario para diagnosticar diseños que están causando problemas reales.

Cinco principios con historia

SOLID es un acrónimo mnemotécnico de cinco principios de diseño que Robert C. Martin reunió y perfeccionó a finales de la década de 1990, de forma más completa en su artículo de 2000 *Design Principles and Design Patterns*. Michael Feathers se dio cuenta de que sus iniciales podían reordenarse para formar la palabra SOLID, y el acrónimo se quedó. Los principios se escribieron para las clases de la orientación a objetos, pero el libro posterior de Martin, *Clean Architecture* (2017), sostiene que cada uno se aplica también a los componentes, y los profesionales los aplican habitualmente también a servicios y API.

Por eso tienen cabida en un curso de arquitectura. Cada principio es en realidad una afirmación sobre **dónde debe recaer el cambio** y **hacia dónde deben apuntar las dependencias**, las dos cuestiones de las secciones anteriores sobre acoplamiento y cohesión y sobre gestión de dependencias. Esta sección trata cada principio por turno, lo muestra en unas pocas líneas de Python y después muestra lo que significa cuando la "clase" es un componente desplegable o un servicio que pertenece a otro equipo. Termina con los principios de cohesión a nivel de componente y con las críticas que recibe SOLID, que son sustanciales y vale la pena conocer.

SI ESTO ES NUEVO PARA TI

Si esto es nuevo para ti

Necesitas tres ideas de secciones anteriores. Un **módulo** es cualquier unidad de código con un límite: una función, una clase, un paquete, un componente o un servicio. Una **interfaz** es el conjunto de operaciones que un módulo ofrece a otros, un contrato que oculta cómo se hace el trabajo. El **acoplamiento** es cuánto debe saber un módulo sobre otro; cuanto más sabe, más probable es que un cambio en un lado rompa el otro. El `Protocol` de Python, que se usa a continuación, es simplemente una forma de dar nombre a una interfaz: cualquier clase con los métodos correspondientes la cumple.

S: responsabilidad única significa un actor

El principio de responsabilidad única es el más citado y el peor entendido de los cinco. *No* dice que un módulo deba hacer una sola cosa; esa es una buena regla para las funciones, pero es una regla distinta. La formulación del propio Martin evolucionó de decir que una clase debe tener un solo motivo para cambiar a decir, en *Clean Architecture*, que un módulo debe ser responsable ante un actor, y solo uno. Un actor es un grupo de personas (usuarios o partes interesadas) que quieren que el sistema cambie de la misma manera. El principio trata de las personas y de la dirección del cambio futuro.

La ilustración clásica es una clase `Employee` con tres métodos: `calculate_pay`, especificado por finanzas; `report_hours`, especificado por operaciones; y `save`, especificado por los administradores de bases de datos. Si el cálculo de la nómina y el informe de horas comparten una función auxiliar para calcular las horas ordinarias, un cambio que pide finanzas altera en silencio el informe del que depende operaciones. Nadie fue descuidado; el código simplemente mezcló las inquietudes de dos actores, de modo que un cambio solicitado por uno podía perjudicar al otro.

```
from dataclasses import dataclass

@dataclass
class EmployeeRecord:          # shared data, no behaviour
    id: str
    hourly_rate: float
    hours: list[float]

class PayCalculator:           # actor: finance
    def pay(self, e: EmployeeRecord) → float:
        regular = sum(min(h, 8) for h in e.hours)
        overtime = sum(max(h - 8, 0) for h in e.hours)
        return (regular + 1.5 * overtime) * e.hourly_rate

class HoursReporter:           # actor: operations
    def weekly_report(self, e: EmployeeRecord) → str:
        return f"{e.id}: {sum(e.hours):.1f} hours"

class EmployeeRepository:      # actor: database administrators
    def save(self, e: EmployeeRecord) → None: ...
```

Separar por actor: cada clase cambia solo cuando lo piden sus propias partes interesadas, y los datos compartidos siguen siendo un simple registro.

A escala de componentes y de servicios, el principio se convierte en una prueba para los límites: **¿de quién son las solicitudes que cambian este componente?** En una plataforma de reparto de comida, un único servicio "Restaurante" que contiene las cartas (que cambian a solicitud de los dueños de los restaurantes), las reglas de comisiones (que cambia el equipo de

finanzas) y la clasificación en las búsquedas (que cambia el equipo de crecimiento) se modificará por tres motivos no relacionados, por tres grupos de personas, con tres ritmos de lanzamiento. Cada lanzamiento pone en riesgo los otros dos asuntos, y el equipo propietario se convierte en una cola de espera. Separar por actores, por ejemplo en un servicio de cartas, un componente de comisiones dentro del dominio de finanzas y un servicio de clasificación, alinea cada límite con una sola fuente de cambio. Esta es también la idea que hay detrás de alinear los servicios con las capacidades de negocio y, en el diseño guiado por el dominio, con los contextos delimitados (que se tratan en el Capítulo 5).

Observa que el principio no dice "haz que los servicios sean pequeños". Un servicio grande que sirve bien a un único actor está bien; un servicio diminuto que dos departamentos no dejan de cambiar en direcciones contradictorias no lo está.

O: abierto a la extensión, cerrado a la modificación

El principio abierto-cerrado lo formuló por primera vez Bertrand Meyer en *Object-Oriented Software Construction* (1988): un módulo debe estar abierto a la extensión (puedes añadir un comportamiento nuevo) pero cerrado a la modificación (su código fuente existente y probado no necesita editarse para ello). Meyer pensaba en la herencia; Martin lo reinterpretó en términos de interfaces y polimorfismo. La forma práctica es: **cuando se espera un tipo de variación, dale un punto de extensión**, para que la siguiente variante sea código nuevo y no otra rama más en el código antiguo.

```
from typing import Protocol

class Carrier(Protocol):
    def quote(self, weight_kg: float) → int: ...   # price in pence

class RoyalMail:
    def quote(self, weight_kg: float) → int:
        return 350 + int(weight_kg * 120)

class Courier:
    def quote(self, weight_kg: float) → int:
        return 600 + int(weight_kg * 80)

CARRIERS: dict[str, Carrier] = {"royal_mail": RoyalMail(), "courier": Courier()}

def cheapest(weight_kg: float) → tuple[str, int]:
    return min(((name, c.quote(weight_kg)) for name, c in CARRIERS.items()),
               key=lambda pair: pair[1])
```

Añadir un transportista significa registrar una clase nueva; el código que calcula la cotización nunca cambia.

A escala de arquitectura, la misma idea aparece como **puntos de extensión que otros equipos pueden usar sin pedirte que cambies tu código**. Un servicio de pedidos que publica un evento `OrderPlaced` está abierto a la extensión: el equipo de fidelización puede añadir un consumidor nuevo que otorgue puntos, y el servicio de pedidos no se modifica, ni se vuelve a probar, ni se vuelve a desplegar. Una arquitectura de plug-ins (el estilo microkernel) hace la misma promesa para las funcionalidades. En las API, el principio se traduce en una evolución aditiva: campos opcionales nuevos, endpoints nuevos y tipos de evento nuevos amplían un contrato sin modificar aquello de lo que dependen los clientes existentes.

La trampa es la predicción. Un punto de extensión solo tiene valor para los tipos de cambio que realmente llegan. Un registro de transportistas vale la pena si la tienda añade transportistas cada pocos meses; un "motor de reglas de precios" genérico construido porque los precios *podrían* variar algún día es maquinaria especulativa. Los profesionales añaden el punto de extensión la segunda o la tercera vez que aparece una variación, no antes.

L: sustitución de Liskov y contratos de comportamiento

El principio de sustitución de Liskov procede de la conferencia magistral de Barbara Liskov *Data Abstraction and Hierarchy* en la conferencia OOPSLA de 1987, y Liskov y Jeannette Wing lo precisaron en *A Behavioral Notion of Subtyping* (ACM Transactions on Programming Languages and Systems, 1994). Dicho de manera informal: si un código funciona con objetos del tipo T, debe seguir funcionando cuando recibe un objeto de cualquier subtipo S, sin saber cuál ha recibido. Tener los nombres de método correctos no basta; el *comportamiento* debe ser compatible.

Las reglas del subtipado conductual conectan directamente con el diseño por contrato (Sección 3.3). Un subtipo **no puede reforzar las precondiciones** (exigir más al llamante), **no puede debilitar las postcondiciones** (prometer menos a cambio) y debe **preservar los invariantes** (los hechos que siempre se cumplen para el tipo). Liskov y Wing añadieron una restricción de historia: un subtipo no debe permitir cambios de estado que el supertipo prohíbe, y por eso un punto mutable no puede hacerse pasar por uno inmutable.

```
class Discount:
    """Postcondition: 0 ⩽ result ⩽ total."""
    def apply(self, total: int) → int:
        return total

class PercentOff(Discount):
    def __init__(self, pct: int):
        self.pct = pct
    def apply(self, total: int) → int:
        return total - total * self.pct // 100      # keeps the contract

class FixedVoucher(Discount):
    def __init__(self, pence: int):
        self.pence = pence
    def apply(self, total: int) → int:
        return total - self.pence                    # can go below zero!

# FixedVoucher(500).apply(300) == -200: the checkout now issues a refund.
# Fix: return max(total - self.pence, 0), honouring the base postcondition.
```

El contrato base dice que un descuento nunca hace que el total sea negativo. Un subtipo que lo incumple hace fallar a los llamantes que confiaban en el contrato.

El síntoma de una violación del LSP es la comprobación de tipos en los llamantes: `if isinstance(d, FixedVoucher): ...`. Cada una de esas comprobaciones significa que una abstracción ya no es fiable, y el conocimiento de sus excepciones se extiende por todo el código.

A escala de servicios, la sustituibilidad es la pregunta que hay detrás de todo **cambio compatible de una API**. Si la versión 2.3 de una API de pagos debe sustituir a la 2.2 sin que los clientes lo noten, es un subtipo conductual de la 2.2 y se aplican las mismas reglas. No debe reforzar las precondiciones: ningún campo obligatorio nuevo en la solicitud, ninguna validación más estricta de los campos existentes, ningún límite de tasa más bajo. No debe debilitar las postcondiciones: ningún campo eliminado de la respuesta, ningún tiempo de respuesta garantizado más lento, ninguna pérdida de idempotencia. Debe preservar los invariantes: un importe sigue expresado en unidades menores, un identificador de pedido sigue sin cambiar nunca. Añadir un valor nuevo a una enumeración en una *respuesta* es un caso sutil: amplía lo que el cliente puede recibir, lo que rompe a los clientes que daban por hecho que el conjunto anterior estaba completo. El mismo razonamiento se aplica cuando una implementación de un servicio sustituye a otra detrás de un balanceador de carga, o cuando cambias el adaptador de un proveedor de pagos por otro. Las pruebas de contrato dirigidas por el consumidor son, en la práctica, comprobaciones de Liskov automatizadas, y la ley de Hyrum (Sección 3.3) recuerda que los clientes pueden depender de un comportamiento que nunca prometiste.

I: segregación de interfaces y API específicas para cada cliente

El principio de segregación de interfaces dice que ningún cliente debe verse obligado a depender de operaciones que no usa. Una interfaz inflada acopla a cada cliente con cada operación: cuando cambia una operación, todos los clientes se ven afectados, como mínimo porque tienen que recompilar, volver a probar o volver a desplegar, incluso los que nunca la llaman. En los lenguajes compilados ese acoplamiento es literal (un cambio en un archivo de cabecera o en un JAR compartido obliga a recompilar todo lo que depende de él); en Python se manifiesta en dobles de prueba que deben implementar veinte métodos para probar código que llama a dos.

```
from typing import Protocol

class OrderReader(Protocol):
    def get(self, order_id: str) → dict: ...
    def list_for_customer(self, customer_id: str) → list[dict]: ...

class OrderWriter(Protocol):
    def save(self, order: dict) → None: ...
    def cancel(self, order_id: str) → None: ...

def monthly_statement(orders: OrderReader, customer_id: str) → int:
    return sum(o["total"] for o in orders.list_for_customer(customer_id))

# The reporting job cannot cancel an order, and its tests
# need a fake with two read methods, not the whole repository.
```

Cada cliente depende solo del rol que necesita; una misma clase puede seguir implementando ambos roles.

A escala de servicios, la segregación de interfaces es el argumento a favor de las **API específicas para cada cliente**. Una única API de propósito general que da servicio a la aplicación móvil, a la tienda web, al sistema del almacén y a las integraciones de socios tiende a convertirse en un diseño de mínimo común denominador: cada cliente recibe campos que ignora, y cada cambio tiene que negociarse con todos ellos. El patrón backend for frontend (bautizado en SoundCloud y descrito por Phil Calçado y Sam Newman) da a cada tipo de cliente su propia capa de API delgada, propiedad del equipo que construye ese cliente, que compone llamadas a los servicios subyacentes. GraphQL aborda el mismo problema de otro modo, permitiendo que cada cliente seleccione exactamente los campos que necesita. En el caso de los eventos, sugiere publicar varios tipos de evento acotados (`OrderPlaced`, `OrderShipped`) en lugar de un único evento `OrderChanged` con todos los campos, que obliga a cada consumidor a analizarlo y filtrarlo todo.

La versión arquitectónica también se aplica a las **bibliotecas compartidas**. Una biblioteca `common` de toda la empresa que contiene logging, manejo de importes monetarios, generación de PDF y clientes HTTP acopla a cada servicio con cada una de sus partes: una corrección de seguridad en el código de PDF obliga a lanzar una versión nueva de servicios que nunca generan PDF. Dividirla en bibliotecas pequeñas que los clientes adoptan de forma independiente es segregación de interfaces a nivel de paquetes, y anticipa el principio de reutilización común que se ve más abajo.

D: inversión de dependencias y quién es el dueño de la interfaz

El principio de inversión de dependencias tiene dos partes: los módulos de alto nivel (la política de negocio) no deben depender de los módulos de bajo nivel (detalles como bases de datos, intermediarios de mensajes y frameworks web); ambos deben depender de abstracciones. Y las abstracciones no deben depender de los detalles. La palabra "inversión" se refiere a la dirección de la dependencia del código fuente: en un diseño ingenuo, la política llama al detalle y, por tanto, lo importa; tras la inversión, el detalle importa una interfaz que define la política, de modo que la flecha del código fuente apunta en sentido contrario al flujo de control.

El punto crucial, y que a menudo se pasa por alto, es la **propiedad**. La interfaz pertenece al componente de alto nivel y se expresa en sus términos (`OrderRepository.save(order)` , no `PostgresClient.execute(sql)`). Si la interfaz vive en el paquete de persistencia, has añadido una abstracción pero no has invertido nada: la política sigue dependiendo del paquete de persistencia.

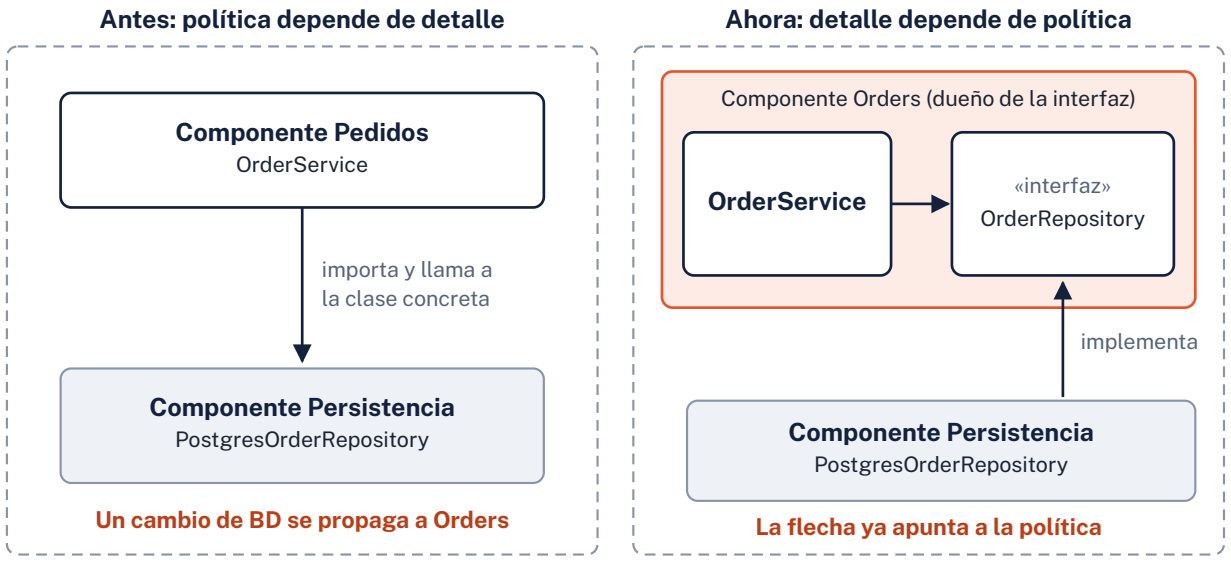
```
# orders/ports.py  (high-level policy owns this)
from typing import Protocol
class PaymentPort(Protocol):
    def charge(self, order_id: str, pence: int) → str: ...  # returns payment id

# orders/service.py
class Checkout:
    def __init__(self, payments: PaymentPort):
        self.payments = payments
    def pay(self, order_id: str, pence: int) → str:
        return self.payments.charge(order_id, pence)

# infrastructure/stripe_adapter.py  (detail depends on policy)
from orders.ports import PaymentPort
class StripeAdapter(PaymentPort):
    def charge(self, order_id: str, pence: int) → str:
        ...  # call the provider's API, map its errors to ours
```

El paquete de pedidos define el puerto que necesita; la infraestructura lo implementa y se conecta al arrancar.

Invertir una dependencia a través del límite de un componente



Llevar la interfaz al componente Orders invierte la dependencia del código fuente, de modo que la persistencia puede cambiar sin tocar la política de negocio.

La inversión de dependencias no es lo mismo que la inyección de dependencias, aunque suelen ir juntas. La inyección es una técnica para proporcionar a un módulo sus colaboradores desde fuera (el argumento del constructor `payments` de más arriba); la inversión es una decisión sobre hacia dónde apuntan las dependencias del código fuente. Puedes inyectar una clase concreta sin invertir nada.

A escala de arquitectura, la inversión de dependencias es la regla que organiza las arquitecturas hexagonal, de cebolla y limpia: el dominio está en el centro y define puertos; los adaptadores para bases de datos, colas e interfaces de usuario están fuera y dependen hacia dentro. Entre servicios, la misma pregunta se formula como "¿quién es el dueño del contrato?". Si un servicio central de facturación define un esquema `BillingEvent` y cada equipo de producto debe adaptarse a él, la facturación es la política estable y los productos dependen de ella, lo que a menudo es correcto. Si, en cambio, la facturación lee las tablas internas de cada producto, la dependencia apunta de lo estable a lo volátil, y cada cambio en el esquema de un producto rompe la facturación. La Sección 3.4 mostró cómo cuantificar esto con el principio de dependencias estables.

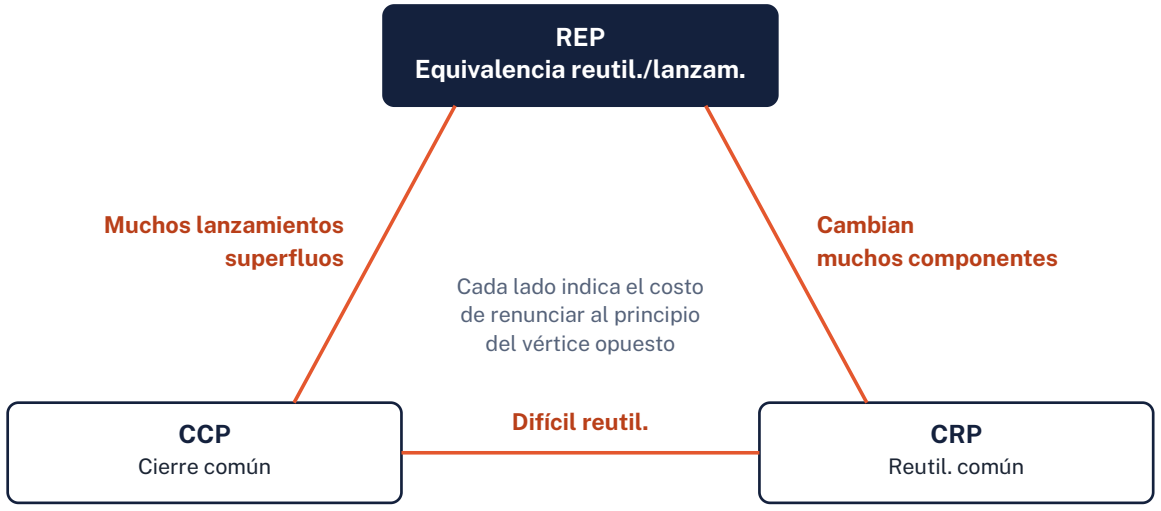
Los principios de cohesión de componentes: REP, CCP y CRP

SOLID te dice cómo dar forma a las clases y a sus dependencias. Martin lo combina con tres principios para decidir **qué clases van juntas en un mismo componente**, donde un componente es una unidad de lanzamiento y despliegue, como un JAR, un paquete de Python o un ensamblado de .NET.

El principio de equivalencia entre reutilización y lanzamiento (REP) dice que la unidad de reutilización es la unidad de lanzamiento: si otros van a reutilizar un componente, este necesita números de versión, notas de la versión y un tema coherente, así que las clases que contiene deben tener sentido lanzadas juntas. El principio de cierre común (CCP) es el principio de responsabilidad única para los componentes: reúne en un componente las clases que cambian por los mismos motivos y en los mismos momentos, para que un cambio típico toque un solo componente y solo haya que lanzar ese. El principio de reutilización común (CRP) es el principio de segregación de interfaces para los componentes: las clases que se usan juntas van juntas, y las que no, no deben empaquetarse juntas, porque depender de un componente significa depender de todo él.

Los tres tiran en direcciones opuestas. REP y CCP son *inclusivos* (hacen los componentes más grandes); CRP es *exclusivo* (los hace más pequeños). Martin los dibuja como un triángulo en el que cada lado nombra el precio de abandonar el principio del vértice opuesto.

El triángulo de tensiones



Concentrarse en dos principios de cohesión cualesquiera conlleva el costo escrito en el lado que los une; los arquitectos eligen una posición, y la posición se desplaza a medida que el proyecto madura.

La observación de Martin es que la posición de un proyecto en el triángulo debe desplazarse con el tiempo. Al principio, lo que más importa es la facilidad de desarrollo, así que los equipos favorecen el CCP: agrupar por cambio y aceptar que los componentes son incómodos de reutilizar. A medida que un sistema madura y otros equipos empiezan a depender de sus partes, el equilibrio se desplaza hacia el REP y el CRP. La misma tensión aparece entre servicios. Fusionar dos servicios que siempre cambian juntos (CCP) reduce los lanzamientos

coordinados, pero hace que el resultado sea menos reutilizable; dividir un servicio para que los consumidores dependan solo de lo que usan (CRP) aumenta el número de desplegables a los que afecta un único cambio de negocio.

EJEMPLO PRÁCTICO

Ejemplo práctico: revisar un componente compartido en una tienda en línea

Una tienda en línea tiene un paquete `shop-core` que usan cinco servicios. Contiene reglas de precios, cálculo de impuestos, validación de direcciones, generación de facturas en PDF y una función auxiliar de base de datos. En el último trimestre tuvo 40 lanzamientos. Aplica los principios.

Actores (SRP, CCP)	Los precios los cambia el equipo comercial; los impuestos, finanzas cuando cambian los tipos; el diseño del PDF, marketing; la función auxiliar de base de datos, el equipo de plataforma. Cuatro actores significan cuatro motivos independientes para cambiar, así que el paquete no tiene cierre común.
Usuarios (ISP, CRP)	El servicio del almacén solo usa la validación de direcciones y, sin embargo, cada uno de los 40 lanzamientos lo obligó a actualizarse y a volver a probarse. Según el CRP, aproximadamente 40 lanzamientos menos los pocos que afectaron a las direcciones fueron innecesarios para él.
Lanzamientos (REP)	Las notas de la versión no tienen un tema coherente, así que los consumidores no pueden juzgar si un lanzamiento les afecta.
Dependencias (DIP, SDP)	La función auxiliar de base de datos importa los modelos ORM de los servicios, así que un paquete estable y muy usado depende de código volátil de los servicios: la flecha apunta en la dirección equivocada.
División propuesta	`pricing` (propiedad del dominio comercial), `tax` (finanzas), `address`, `invoice-pdf`, y la función auxiliar de base de datos trasladada a cada servicio o a una biblioteca de plataforma sin importaciones del dominio.
Efecto esperado	Cada servicio depende solo de los paquetes que usa. Un cambio en un tipo impositivo lanza un solo paquete y solo afecta a los servicios que calculan impuestos.

Los principios no dictaron la respuesta; dieron un vocabulario para explicar por qué el paquete actual causa problemas y qué cuesta cada opción.

La división aumenta el número de paquetes que hay que versionar y publicar: el lado "cambian muchos componentes" del triángulo. Es aceptable aquí porque los cambios son, en su mayoría, independientes.

Críticas y límites

SOLID se enseña mucho y se critica mucho, y ambas reacciones tienen su parte de razón. Las críticas más serias son estas.

Los principios son lo bastante vagos como para justificar casi cualquier cosa. Dan North ha defendido esta idea en charlas de conferencias y en su ensayo de 2022 que presenta CUPID (composable, Unix philosophy, predictable, idiomatic, domain-based). Sostiene que "un solo motivo para cambiar" es arbitrario, ya que de casi cualquier código puede decirse que cambia por muchos motivos, y que aplicarlo con entusiasmo crea separaciones artificiales entre código que en realidad cambia junto, como el contenido de un informe y su formato. Propone propiedades a las que aspirar en lugar de reglas que obedecer. Martin respondió a esta línea de críticas en una publicación de 2020 sobre la vigencia de SOLID, en la que sostiene que los principios siguen siendo válidos porque los problemas de fondo de la gestión de dependencias no han cambiado, y que decir a la gente que "simplemente lo mantenga simple" sin ninguna orientación produce un desastre.

La aplicación dogmática produce un exceso de abstracción. Una interfaz para cada clase, una fábrica para cada interfaz y un punto de extensión para cada variación concebible conducen a un código en el que seguir un solo comportamiento obliga a abrir diez archivos. La distinción de John Ousterhout entre módulos profundos y superficiales (Sección 3.3) es un contrapeso útil: una abstracción debe ocultar mucho detrás de una interfaz pequeña, y muchas clases inspiradas en SOLID son superficiales. Los puntos de extensión especulativos violan YAGNI (Sección 3.8) y tienen un costo real.

Los principios se escribieron para un solo paradigma. En los lenguajes con funciones de primera clase y tipado estructural, muchos patrones SOLID se reducen a pasar una función o a confiar en el duck typing. Las ideas de fondo (separar por actor, depender de contratos estables, mantener pequeñas las interfaces) siguen siendo válidas; la ceremonia, a menudo, no.

La visión profesional equilibrada es tratar SOLID como un **vocabulario de diagnóstico**: cuando un diseño causa problemas, los principios ayudan a explicar por qué y sugieren una dirección. No son una lista de comprobación que toda clase deba cumplir, y "viola el OCP" no es por sí solo un argumento; "cada transportista nuevo ha obligado a editar esta función y ha causado dos incidentes" sí lo es.

GUÍA DE DECISIÓN

Guía de decisión: ¿debo añadir una abstracción aquí?

Úsala cuando un principio parezca pedir una interfaz nueva, un punto de extensión o una división.

Q1 **¿Este tipo de variación ha ocurrido realmente al menos dos veces, o es segura (por ejemplo, se ha firmado un contrato con un segundo proveedor)?**

- **Sí** Probablemente esté justificado un punto de extensión; continúa.
- **No** Mantén el código concreto, mantenlo fácil de refactorizar y espera a tener pruebas reales (YAGNI).

Q2 **¿Las partes que quieres separar las cambian actores distintos o con calendarios distintos?**

- **Sí** Sepáralas (SRP, CCP); el límite reducirá la coordinación.
- **No** Mantenlas juntas; separarlas añadiría costuras que cada cambio tendría que cruzar.

Q3 **¿Una política de alto nivel importa actualmente un detalle volátil, como una base de datos, el SDK de un proveedor o un framework?**

- **Sí** Define una interfaz que pertenezca al lado de la política e impleméntala en el detalle (DIP).
- **No** No hace falta ninguna inversión; una dependencia directa de un módulo estable está bien.

Q4 **¿La interfaz nueva ocultará una complejidad significativa detrás de unas pocas operaciones?**

- **Sí** Es un módulo profundo; vale la pena tenerlo.
- **No** Es un simple intermediario superficial; replantéatelo o fusiónalo con quien lo llama.

SOLID en tres escalas

Principio	Escala de clase	Escala de componente o de servicio
Responsabilidad única	Un actor por clase	Servicio o componente alineado con una capacidad de negocio y un equipo propietario
Abierto-cerrado	Comportamiento nuevo mediante nuevas implementaciones de una interfaz	Eventos, plug-ins y cambios aditivos en la API que permiten a otros extender sin modificarte
Sustitución de Liskov	Las subclases respetan el contrato de la clase base	Las nuevas versiones de la API y las implementaciones de sustitución respetan el contrato anterior: ninguna entrada obligatoria nueva, ninguna salida eliminada
Segregación de interfaces	Interfaces de rol pequeñas	API específicas para cada cliente (backend for frontend), tipos de evento acotados, bibliotecas compartidas pequeñas
Inversión de dependencias	Depender de abstracciones que te pertenecen	El dominio define puertos; los adaptadores y la infraestructura dependen hacia dentro; los propietarios estables definen los contratos

COMPRUEBA LO QUE HAS ENTENDIDO

▼ Una clase da formato a una factura y además calcula sus impuestos. ¿Viola eso automáticamente el principio de responsabilidad única?

No automáticamente. Lo viola si el formato y las reglas fiscales los cambian actores distintos (por ejemplo, marketing y finanzas), porque entonces los cambios solicitados por uno pueden romper lo del otro; si un mismo grupo es responsable de ambos, la clase puede estar bien.

▼ ¿Por qué añadir un campo obligatorio nuevo a una solicitud de una API rompe la sustituibilidad?

Refuerza la precondición: los llamantes que cumplían el contrato anterior ahora fallan, así que la versión nueva ya no puede ocupar el lugar de la anterior.

▼ ¿Cuál es la diferencia entre la inversión de dependencias y la inyección de dependencias?

La inversión trata de la dirección de las dependencias del código fuente, con el lado de la política como dueño de la abstracción; la inyección es una técnica para proporcionar colaboradores desde fuera. Puedes inyectar una clase concreta sin invertir nada.

▼ **Un equipo agrupa las clases estrictamente según lo que cambia junto, y otros equipos se quejan de que es imposible reutilizar los componentes. ¿En qué lado del triángulo de tensiones están?**

En el lado CCP-CRP, con la etiqueta difícil de reutilizar: han descuidado el principio de equivalencia entre reutilización y lanzamiento.

▼ **Inténtalo: La versión 2 de una API interna de reembolsos rechaza ahora importes superiores a £100, que la versión 1 aceptaba, y ya no devuelve el campo reason. El equipo la llama compatible porque la URL no ha cambiado. Usa la sustitución de Liskov para juzgarlo.**

No es compatible. La sustitución de Liskov pregunta si la nueva versión puede reemplazar a la antigua sin que lo noten los llamadores. Rechazar importes superiores a £100 refuerza una precondición, y eliminar el campo de motivo debilita una poscondición; cada cosa puede romper a los llamadores existentes. Que la URL no cambie no demuestra nada. O bien mantén el comportamiento antiguo, haciendo cambios solo por adición, o bien publica una nueva versión y migra a los llamadores, aplicando el límite de £100 solo a quienes opten por él.

FUENTES

- Robert C. Martin: The Single Responsibility Principle (Clean Coder blog): https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html
- Robert C. Martin: Solid Relevance (Clean Coder blog, 2020): https://blog.cleancoder.com/uncle-bob/2020/10/18/Solid-Relevance.html
- Barbara Liskov and Jeannette Wing: A Behavioral Notion of Subtyping (1994): https://www.cs.columbia.edu/~wing/publications/LiskovWing94.pdf
- Dan North: CUPID, for joyful coding (2022): https://dannorth.net/blog/cupid-for-joyful-coding/
- Sam Newman: Backends for Frontends pattern: https://samnewman.io/patterns/architectural/bff/

SECCIÓN 3.6 DE 3.8 · INTERMEDIO · 22 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Connascence: una medida más fina del acoplamiento

ANTES DE LEER ESTO

Esta sección se basa en:

- 3.2 Acoplamiento y cohesión
- 3.3 Ocultación de información

Términos anteriores que utiliza: Acoplamiento, Encapsulamiento, Acoplamiento semántico, Acoplamiento temporal, Invariante

LA IDEA EN BREVE

Cuando dos amigos acuerdan verse "en el sitio de siempre a la hora de siempre", quedan ligados por un acuerdo que nadie más puede ver; si uno de ellos cambia de plan, el otro tiene que saberlo. El software está lleno de acuerdos así: nombres, el orden de los valores, lo que significa en secreto un número, qué paso debe ocurrir primero. La connascence es una forma de enumerar estos acuerdos, juzgar lo arriesgado que es cada uno y hacer que los arriesgados sean más débiles, más visibles o estén más cerca entre sí.

LO QUE NECESITAS SABER

- Dos elementos tienen connascence si un cambio en uno exigiría un cambio en el otro para que el sistema siga siendo correcto; Meilir Page-Jones introdujo la idea en 1992.
- Las formas estáticas (nombre, tipo, significado, posición, algoritmo) son visibles en el código fuente; las formas dinámicas (ejecución, temporización, valor, identidad) solo se manifiestan en tiempo de ejecución y suelen ser más fuertes.
- La fuerza es lo difícil que es encontrar y cambiar una forma, la localidad es lo alejados que están los elementos y el grado es cuántos elementos comparten el acuerdo.
- Convierte las formas fuertes en otras más débiles, mantén local cualquier connascence fuerte y reduce el grado canalizando una convención a través de una sola definición o un solo propietario.
- Sustituir los parámetros posicionales por parámetros con nombre y los valores mágicos por enumeraciones con nombre convierte acuerdos fuertes e invisibles en acuerdos débiles que las herramientas pueden comprobar.

- Entre servicios, solo son sanas las formas débiles, como el nombre y el tipo en un esquema publicado; la connascence de valor o de identidad entre servicios indica un límite mal situado.

Por qué el acoplamiento necesita un vocabulario más fino

La Sección 3.2 describió el acoplamiento con los niveles del diseño estructurado (de contenido, común, externo, de control, de estampado, de datos) y con las formas distribuidas que importan entre servicios. Esos niveles clasifican *cómo* están conectados los módulos. Dicen menos sobre dos preguntas a las que se enfrentan los ingenieros cada día: **¿cuánto dolerá un cambio?** y **¿qué refactorización reduciría ese dolor?** La connascence responde a ambas.

La connascence (del latín, "nacidos juntos") la introdujo Meilir Page-Jones en un artículo de 1992 en *Communications of the ACM*, "Comparing techniques by means of encapsulation and connascence", y la desarrolló en su libro *What Every Programmer Should Know About Object-Oriented Design* (Dorset House, 1995), reelaborado más tarde como *Fundamentals of Object-Oriented Design in UML* (1999). Su definición es sencilla: dos elementos tienen connascence si algún cambio en uno exigiría un cambio en el otro para que el sistema siga siendo correcto. Los elementos pueden ser cualquier cosa: dos funciones, una función y quienes la llaman, un productor y los consumidores de sus mensajes, una columna de una base de datos y cada informe que la lee.

Dos características hacen que la idea sea útil para los arquitectos. En primer lugar, la connascence se presenta en **formas** con nombre que pueden ordenarse por fuerza, de modo que puedes decir no solo "estos están acoplados", sino "estos están acoplados por posición, que es peor que el acoplamiento por nombre, y así es como se convierte uno en el otro". En segundo lugar, tiene dos dimensiones más, la **localidad** y el **grado**, que te indican si una forma dada es aceptable allí donde está.

SI ESTO ES NUEVO PARA TI

Si esto es nuevo para ti

Siempre que dos fragmentos de código deben ponerse de acuerdo en algo, tienen connascence. Si una función llama a otra por su nombre, ambas deben coincidir en ese nombre: si renombras una, debes renombrar la otra. Eso es inevitable e inofensivo. El peligro está en los acuerdos difíciles de ver (un número que en secreto significa "enviado"), difíciles de comprobar (dos servicios que deben ejecutarse en un orden concreto) o repartidos por muchos lugares. La connascence es una forma de poner nombre a estos acuerdos ocultos para que puedas hacerlos visibles, más débiles o más cercanos entre sí.

Formas estáticas: visibles en el código

La connascence estática puede encontrarse leyendo el código fuente, a menudo con herramientas, sin ejecutar el programa. De la más débil a la más fuerte, en el orden que se da con más frecuencia:

Connascence de nombre: los elementos deben coincidir en un nombre. Un llamante usa `calculate_tax`; un consumidor de JSON lee `customer_id`. Es la forma más débil, porque renombrar es mecánico y las herramientas encuentran cada uso.

Connascence de tipo: los elementos deben coincidir en un tipo. Una función espera un número entero de peniques; pasarle un número de coma flotante compila en Python, pero corrompe el resultado. Los verificadores de tipos y los esquemas la hacen visible.

Connascence de significado (o de convención): los elementos deben coincidir en lo que significa un valor. El estado `3` significa "enviado"; `-1` significa "no encontrado"; una cadena vacía significa "usa el valor predeterminado". Esos valores mágicos son invisibles para las herramientas: nada vincula el `3` del código del almacén con el `3` del servicio de correo electrónico.

Connascence de posición: los elementos deben coincidir en el orden de los valores. `create_user("Ann", "Smith", "ann@example.com")` depende del orden de los parámetros; un archivo CSV depende del orden de las columnas; una tupla `(lat, lon)` se rompe en silencio si alguien supone `(lon, lat)`, como hace GeoJSON.

Connascence de algoritmo: los elementos deben coincidir en un algoritmo. El emisor y el receptor de un webhook deben calcular la misma firma HMAC; un cliente y un servidor deben aplicar el mismo esquema de hash de contraseñas o la misma suma de comprobación; cada servicio que enruta por cliente debe usar la misma función hash para elegir un shard.

Formas dinámicas: visibles solo en tiempo de ejecución

La connascence dinámica depende de cómo se comporta el sistema cuando se ejecuta, así que es más difícil de encontrar y suele ser más fuerte.

Connascence de ejecución: importa el orden de ejecución. Hay que abrir una conexión antes de enviar datos; `begin()` debe preceder a `commit()`; hay que crear un pedido antes de capturar su pago.

Connascence de temporización: importa el momento de la ejecución. El timeout de un cliente debe ser más largo que el tiempo de procesamiento normal del servidor; una prueba que espera dos segundos confía en que un trabajo asíncrono haya terminado; dos hilos compiten por actualizar un saldo.

Connascence de valor: varios valores deben cambiar juntos, normalmente debido a un invariante. La lista de códigos de país válidos del frontend web y la del backend deben coincidir; el total de un pedido debe ser igual a la suma de sus líneas; los cargos y los abonos de un asiento contable deben cuadrar.

Connascence de identidad: los elementos deben referirse a la misma entidad, no solo a una igual. Dos repositorios deben compartir la misma transacción de base de datos para que sus escrituras sean atómicas; dos componentes deben publicar en la *misma* instancia de cola, no en dos colas con la misma configuración.

Page-Jones también dio nombre a la relación opuesta, la **contranascence**: elementos que deben *diferir*, como dos clases de un mismo espacio de nombres que no deben compartir nombre, o dos servicios que no deben usar el mismo puerto. Obedece las mismas reglas.

Las nueve formas, con un ejemplo de cada una

Forma	Clase	Ejemplo en una app de reparto de comida	Forma habitual de debilitarla
Nombre	Estática	El servicio de pedidos llama a `restaurant.get_menu()`	Ya es la forma más débil; mantener nombres significativos
Tipo	Estática	La tarifa de envío se pasa como un entero de peniques, no como un número de coma flotante en libras	Anotaciones de tipo, esquemas, un tipo `Money`
Significado	Estática	`status == 4` significa "repartidor asignado"	Constantes con nombre o una enumeración, compartidas mediante una sola definición
Posición	Estática	`quote(2.5, 51.5, -0.12, True)`	Argumentos con nombre, un objeto de parámetros, campos JSON con nombre
Algoritmo	Estática	La app y el servidor calculan la misma suma de comprobación del carrito	Una sola biblioteca compartida, o dejar que un lado calcule y envíe el resultado
Ejecución	Dinámica	El pago debe autorizarse antes de avisar al restaurante	Encapsular la secuencia en un solo lugar (un flujo de trabajo o una saga)
Temporización	Dinámica	La app del repartidor desiste a los 5 s, mientras que el despacho tarda hasta 8 s	Hacer asíncrona la interacción; derivar los timeouts de los SLO
Valor	Dinámica	El frontend y el backend tienen cada uno la lista de zonas de reparto	Un solo propietario de los datos; los demás los consultan o se suscriben a ellos
Identidad	Dinámica	Dos módulos deben escribir dentro de la misma transacción de base de datos	Dar a un solo módulo la propiedad de la unidad de trabajo

Fuerza, localidad y grado

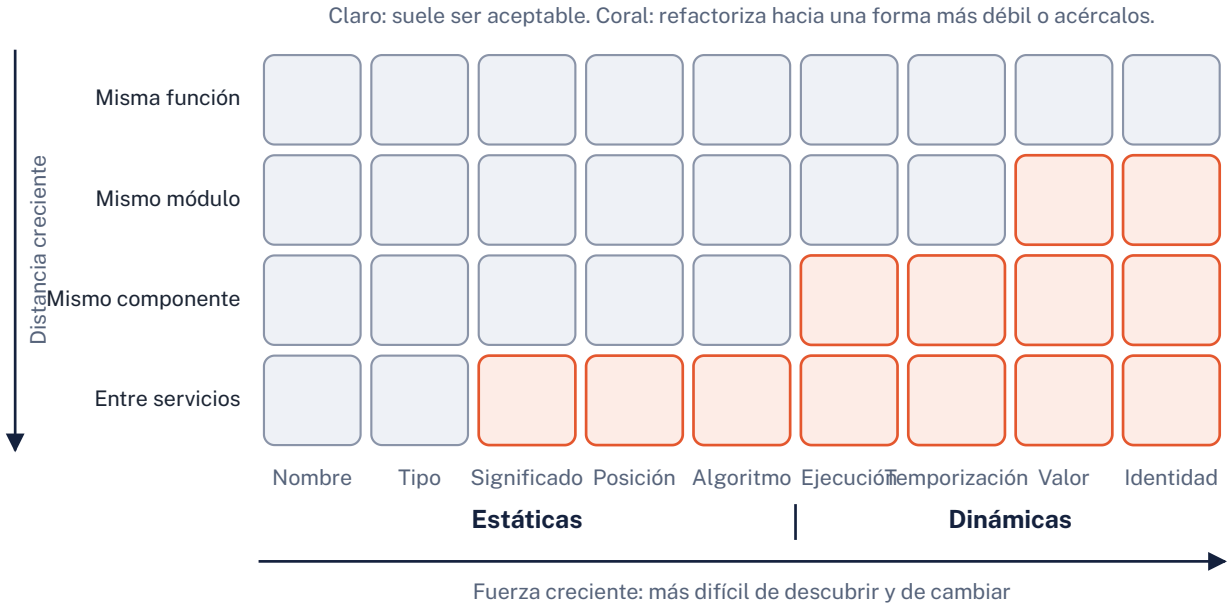
Cada caso de connascence puede juzgarse según tres propiedades. La fuerza es lo difícil que es descubrir y cambiar la forma: el nombre es débil porque las herramientas lo encuentran y lo renombran; el significado es más fuerte porque nada señala los valores mágicos; la

temporización y la identidad son las más fuertes porque puede que solo aparezcan bajo carga o en producción. El orden de la tabla es la clasificación convencional, aunque los autores difieren ligeramente en las formas estáticas (algunos sitúan la posición por debajo del significado).

La localidad es lo cerca que están los elementos con connascence. Los parámetros posicionales entre dos funciones auxiliares privadas del mismo archivo son inofensivos: quien cambia una ve la otra. Los campos posicionales en un mensaje entre servicios que pertenecen a equipos distintos son peligrosos, porque las personas que deben cambiar a la vez no se sientan juntas, no lanzan juntas y puede que ni siquiera sepan unas de otras.

El grado es cuántos elementos intervienen. Un número mágico compartido por dos funciones es una molestia; el mismo número usado por cuarenta archivos en seis servicios es un proyecto de migración. El grado puede contarse: si el código de estado `3` aparece en 14 lugares del servicio del almacén, en 9 del servicio de notificaciones y en 5 de las vistas de la base de datos de informes, el grado es 28, y cambiar su significado implica coordinar 28 ediciones entre tres equipos y al menos dos despliegues, con una migración de datos para las vistas.

La fuerza debe disminuir a medida que aumenta la distancia



Las formas fuertes de connascence son tolerables dentro de una función o una clase; a través de los límites de componentes y servicios, busca las formas más débiles.

Las pautas

El consejo de Page-Jones puede resumirse en tres afirmaciones: minimizar la connascence global dividiendo un sistema en elementos encapsulados; minimizar la connascence que cruza los límites de encapsulamiento; y maximizar la connascence dentro de ellos. La última

sorprende a la gente, pero es simplemente la cohesión expresada de otro modo: lo que debe cambiar a la vez debe estar junto.

Jim Weirich, el desarrollador de Ruby conocido sobre todo como autor de la herramienta de compilación Rake, recuperó la idea para una nueva generación en charlas a partir de 2009, entre ellas "The Building Blocks of Modularity" (impartida en varias conferencias de Ruby en 2009, entre ellas MountainWest RubyConf) (que él llamaba en broma una gran teoría unificada del desarrollo de software) y "Connascence Examined". Condensó la práctica en dos reglas. La **regla de la localidad**: cuanto más alejados estén dos elementos, más débil debe ser la connascence entre ellos. La **regla del grado**: convierte las formas fuertes de connascence en otras más débiles, sobre todo cuando intervienen muchos elementos.

En la práctica, esto da tres movimientos, aproximadamente en orden de preferencia:

Debilitar la forma. Sustituir el significado por el nombre (una constante con nombre), la posición por el nombre (argumentos con nombre, campos con nombre), el algoritmo por el nombre (una única función compartida a la que llaman ambos lados) y la ejecución por el encapsulamiento (un único método que realiza los pasos en el orden correcto).

Aumentar la localidad. Si dos elementos deben compartir una forma fuerte, acércalos: a la misma clase, módulo, componente o servicio, propiedad del mismo equipo.

Reducir el grado. Si muchos elementos comparten una convención, canalízalos a través de una sola definición, un solo propietario o un solo adaptador, para que solo unos pocos lugares conozcan el detalle.

EJEMPLO PRÁCTICO

Ejemplo práctico: sustituir parámetros posicionales y valores mágicos

Una app de transporte con conductor tiene una función que usan la API de reservas, el planificador y una herramienta de administración. Su firma ha crecido durante dos años: `book(rider_id, 51.501, -0.142, 51.515, -0.072, 2, True, 0)`. Lee la connascence y refactorízala.

Posición	Ocho argumentos posicionales. Intercambiar las coordenadas de recogida y de destino sigue pasando la comprobación de tipos y envía en silencio al conductor al lugar equivocado.
Significado	`2` es la clase de vehículo (1 estándar, 2 grande, 3 accesible); `True` significa "pagar con tarjeta"; `0` significa "reservar ahora", mientras que cualquier otro número es una hora programada en segundos desde la época Unix.
Grado	Una búsqueda encuentra 31 puntos de llamada en tres componentes, así que cada parámetro nuevo implica 31 ediciones en una posición fija.
Paso 1: nombre en lugar de posición	Introducir un tipo `Location` y argumentos solo por nombre, para que los llamantes escriban `pickup=Location(lat=51.501, lon=-0.142)`.
Paso 2: nombre en lugar de significado	Sustituir `2` por `VehicleClass.LARGE`, `True` por `Payment.CARD` y `0` por `when=None`, que significa ahora.
Paso 3: reducir el grado	Agrupar los argumentos en un objeto `BookingRequest`; añadir más adelante un campo opcional solo afecta al código que lo necesita.
Resultado	La connascence de posición y de significado (fuerte, de grado 31) se convierte en connascence de nombre y de tipo (débil), que el verificador de tipos y el IDE pueden comprobar y renombrar automáticamente.

El comportamiento no cambió en nada; la refactorización solo cambió qué acuerdos deben respetar los llamantes y lo visibles que son esos acuerdos.

Haz el cambio en pasos que mantengan funcionando a los llamantes antiguos, por ejemplo aceptando ambas formas durante un lanzamiento y registrando el uso de la antigua.

```
from dataclasses import dataclass
from datetime import datetime
from enum import Enum

class VehicleClass(Enum):
    STANDARD = "standard"
    LARGE = "large"
    ACCESSIBLE = "accessible"

class Payment(Enum):
    CARD = "card"
    CASH = "cash"

@dataclass(frozen=True, kw_only=True)
class Location:
    lat: float
    lon: float

@dataclass(frozen=True, kw_only=True)
class BookingRequest:
    rider_id: str
    pickup: Location
    dropoff: Location
    vehicle: VehicleClass = VehicleClass.STANDARD
    payment: Payment = Payment.CARD
    when: datetime | None = None          # None means now

def book(request: BookingRequest) → str: ...
```

Después de la refactorización: cada acuerdo entre llamante y llamado es un nombre o un tipo (kw_only hace que las llamadas posicionales sean un error).

La connascence a través de los límites de los servicios

Mark Richards y Neal Ford, en *Fundamentals of Software Architecture* (2020; segunda edición, 2025), presentan la connascence junto con la cohesión y el acoplamiento como una de las medidas fundamentales de la modularidad, y sostienen que es tan útil entre servicios como dentro del código. Lee las formas pensando en un sistema distribuido y cada una se corresponde con un problema conocido:

El **nombre y el tipo** entre servicios son los acuerdos que un esquema publicado hace explícitos: nombres y tipos de campos en un documento OpenAPI, un esquema Avro o Protobuf, o un catálogo de eventos. Son las formas que quieres, y las pruebas de contrato pueden comprobarlas.

El **significado** entre servicios es el código de estado compartido o el indicador sin explicar: lo que la Sección 3.2 llamó acoplamiento semántico. Prefiere los valores autodescriptivos (`"status": "rider_assigned"`) a los números, y las unidades en los nombres de los campos (`fee_pence`, `timeout_ms`) a las suposiciones. Las unidades merecen especial cuidado: la Mars Climate Orbiter de la NASA se perdió en 1999 porque el software de tierra de un equipo producía cifras de impulso en libras-fuerza-segundo, mientras que el software de navegación que las consumía esperaba newtons-segundo, una connascence de significado entre organizaciones que nadie había hecho explícita.

La **posición** entre servicios aparece en los intercambios de archivos de ancho fijo o CSV y en los formatos de mensaje posicionales; los campos con nombre la eliminan. El **algoritmo** aparece cuando dos servicios deben calcular por separado el mismo precio, descuento o hash; el remedio suele ser que un servicio lo calcule y envíe el resultado.

La **ejecución y la temporización** entre servicios son los problemas difíciles de los sistemas distribuidos, y la temporización es la connascence que hay detrás del acoplamiento temporal de la Sección 3.2: un consumidor que da por hecho que la escritura del productor en la base de datos es visible antes de que llegue el evento, una cadena de llamadas síncronas cuyos timeouts no están bien anidados, un flujo de trabajo que solo funciona si los mensajes llegan en orden. Hacer que las interacciones sean asíncronas, idempotentes y tolerantes al orden debilita estas formas; patrones como las sagas y la bandeja de salida (outbox) (Capítulo 6) existen en gran medida para gestionarlas.

El **valor y la identidad** entre servicios son señales de un límite en el lugar equivocado. Si dos servicios deben mantener un valor consistente (el total del pedido en el servicio de pedidos y el total de la factura en facturación), o deben operar sobre la misma fila en la misma transacción, entonces o bien un servicio debe ser el dueño de esos datos y el otro debe pedírselos, o bien los dos servicios probablemente deberían ser uno solo. Una connascence fuerte a través de una red es la definición de un monolito distribuido.

ATENCIÓN

Errores comunes

- Tratar todo acoplamiento como malo. La connascence de nombre es el precio de cualquier colaboración; el objetivo es una connascence débil, local y de grado bajo, no la ausencia de connascence.
- Debilitar la forma pero aumentar el grado, por ejemplo metiendo todas las constantes de todos los servicios en una única biblioteca compartida de "constantes" que todos los servicios deben actualizar a la vez.
- Corregir las formas estáticas e ignorar las dinámicas. Una API con tipos impecables cuyos llamantes deben invocar tres endpoints en un orden concreto sigue teniendo una connascence de ejecución fuerte.
- Olvidar los elementos que no son código: los archivos de configuración, las vistas de base de datos, los paneles y las reglas de alerta tienen connascence con el código que describen.

Acoplamiento equilibrado: fuerza, distancia y volatilidad

La connascence describe el tipo de conocimiento que comparten dos partes. *Balancing Coupling in Software Design* (2024), de Vlad Khononov, mencionado en la Sección 3.2, se apoya en el diseño estructurado y en la connascence y plantea otra pregunta: ¿es este acoplamiento un problema *aquí*? Sostiene que el acoplamiento no es un mal que haya que minimizar, sino el pegamento que hace que un sistema sea un sistema, y lo juzga en tres dimensiones a la vez.

La **fuerza de integración** es cuánto conocimiento comparten las partes. En orden descendente, sus cuatro niveles son *intrusiva* (basarse en los detalles internos privados de otro, como sus tablas de base de datos), *funcional* (compartir conocimiento de los requisitos de negocio, como con una regla duplicada), *de modelo* (compartir un modelo de dominio) y *de contrato* (compartir solo una interfaz explícita y estable que oculta el resto). La **distancia** es cuán separadas están las partes: el mismo archivo, el mismo módulo, servicios distintos. Aumenta el costo de cambiarlas juntas, y es en parte organizativa, porque dos servicios del mismo equipo están en la práctica más cerca que dos de equipos distintos, como predeciría la ley de Conway. La **volatilidad** es la probabilidad de que una parte cambie. El diseño guiado por el dominio ayuda a estimarla: los subdominios centrales son los que más cambian, y los de soporte y los genéricos, mucho menos.

El modelo suele resumirse en una fórmula simplificada:

```
BALANCE = (STRENGTH XOR DISTANCE) OR NOT VOLATILITY
```

Léela en dos pasos. La fuerza y la distancia deben compensarse: las partes que comparten mucho deben estar juntas (eso es cohesión), y las partes alejadas deben compartir poco. Cuando ambas son altas, los cambios son costosos y frecuentes, lo que da el monolito distribuido. Cuando ambas son bajas, hay cosas sin relación juntas sin motivo, lo que también es complejidad. El segundo paso es la cláusula de escape: si ninguna de las partes va a cambiar, un desequilibrio puede no importar.

En la plataforma de reparto de comida, supón que Reparto lee directamente las tablas de Pedidos. Eso es fuerza intrusiva a través de un límite de servicio y de equipo, y los pedidos son centrales para el negocio, así que está desequilibrado. Hay dos soluciones honestas: reducir la fuerza publicando un contrato o reducir la distancia fusionando ambos. En cambio, una integración fina con un sistema financiero heredado que rara vez cambia puede ser intrusiva y distante, pero tolerable mientras ese sistema siga quieto.

Trata el modelo como una herramienta para pensar, no como una medición. La fórmula es una simplificación binaria del tratamiento más completo del libro. La volatilidad es una previsión, y el historial de commits puede engañar: los commits frecuentes pueden reflejar un mal diseño, y el código que rara vez se toca puede dar simplemente demasiado miedo para cambiarlo. Por eso, un juicio de baja volatilidad debe registrarse, por ejemplo en un registro de decisiones, y revisarse.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **Una función recibe `(width, height)` y un llamante pasa `(height, width)`. ¿Qué forma de connascence se ha roto y cuál es la manera más sencilla de debilitarla?**

La connascence de posición. Usa argumentos con nombre o un objeto de parámetros para que el acuerdo pase a ser de nombre.

▼ **¿Por qué la connascence de significado es más fuerte que la connascence de nombre?**

Las herramientas y los lectores pueden encontrar cada uso de un nombre, pero nada vincula los usos separados de un valor mágico como 3, así que es fácil pasar por alto algún cambio.

▼ **Dos servicios deben actualizar siempre el límite de crédito de un cliente en la misma transacción. ¿Qué sugiere eso?**

Una connascence fuerte de identidad o de valor a través de un límite de red: probablemente los datos necesitan un único dueño, o los dos servicios deberían estar juntos.

▼ **¿Por qué podría ser perfectamente aceptable una connascence fuerte dentro de una sola función pequeña?**

Porque la localidad es alta: quien cambia un elemento ve el otro de inmediato, así que el costo y el riesgo de mantenerlos consistentes son mínimos.

▼ **Inténtalo: La llamada hold(customer, "A", 14, 1, True) la usan el sitio web, la taquilla y una herramienta de administración, donde 1 significa espacio para silla de ruedas. Nombra los tipos de connascence implicados y di cómo los debilitarías.**

Connascence de posición: cada llamador debe mantener los argumentos en el orden correcto, y row y seat podrían intercambiarse sin que nadie lo note. Connascence de significado: 1 y True son valores mágicos que los tres llamadores deben interpretar igual. El grado es tres y los elementos están lejos entre sí, lo que lo empeora. Usa parámetros con nombre o un pequeño tipo de solicitud, y una enumeración con nombre como SeatType.WHEELCHAIR. Ambas pasan a ser connascence de nombre, que las herramientas pueden comprobar.

FUENTES

- connascence.io: reference to the forms and properties of connascence: https://connascence.io/
- connascence.io: connascence of position: https://connascence.io/position.html
- Practicing Ruby: Connascence as a software design metric (on Jim Weirich's talk): https://practicingruby.com/articles/connascence
- Balanced Coupling (coupling.dev): https://coupling.dev/
- Balanced Coupling: Balance: https://coupling.dev/posts/core-concepts/balance/
- Balanced Coupling: Integration strength: https://coupling.dev/posts/dimensions-of-coupling/integration-strength/
- Balanced Coupling: Distance: https://coupling.dev/posts/dimensions-of-coupling/distance/
- Balanced Coupling: Volatility: https://coupling.dev/posts/dimensions-of-coupling/volatility/
- Pearson: Balancing Coupling in Software Design: https://www.pearson.com/en-nz/subject-catalog/p/balancing-coupling-in-software-design-successful-software-architecture-in-general-and-distributed-systems/P200000000372/9780137353576

SECCIÓN 3.7 DE 3.8 · INTERMEDIO · 22 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

La ley de Conway y la forma de las organizaciones

ANTES DE LEER ESTO

Esta sección se basa en:

- 1.3 Por qué importa
- 3.1 Modularidad
- 3.2 Acoplamiento y cohesión

Términos anteriores que utiliza: Interfaz, Acoplamiento, Dependencia

LA IDEA EN BREVE

Si cuatro grupos de constructores que nunca hablan entre sí construyen cada uno una parte de una casa, obtendrás cuatro partes que encajan mal, con las uniones justo donde se encontraron los grupos. El software funciona igual: las partes de un sistema y las conexiones entre ellas tienden a copiar qué equipos hablan entre sí. Así que, cuando diseñas un sistema, también estás diseñando, quieras o no, quién necesita hablar con quién, y lo contrario es igual de cierto.

LO QUE NECESITAS SABER

- El artículo de Melvin Conway de 1968 en Datamation sostenía que las organizaciones que diseñan sistemas están obligadas a producir diseños que copian sus propias estructuras de comunicación.
- La razón es mecánica: una interfaz entre dos subsistemas solo existe donde sus diseñadores la negociaron, por lo que el grafo del sistema refleja el grafo de comunicación de la organización.
- MacCormack, Baldwin y Rusnak (2012) encontraron que los productos de organizaciones débilmente acopladas eran sistemáticamente más modulares que productos equivalentes de organizaciones fuertemente acopladas, aunque una revisión de 2016 halló que este reflejo es común, pero no universal.
- Un grupo de n personas tiene n(n-1)/2 vías de comunicación, así que el costo de coordinación crece aproximadamente con el cuadrado del tamaño del equipo, y los equipos pequeños que son dueños de partes equivalentes del sistema lo mantienen manejable.

- La maniobra inversa de Conway da forma a los equipos y a sus interacciones en función de la arquitectura que quieres, pero solo funciona lentamente frente a un código existente muy acoplado.
- La carga cognitiva del equipo limita cuánto de un sistema puede poseer un equipo; los límites deben dar a cada equipo una parte que pueda entender, construir y operar por completo.

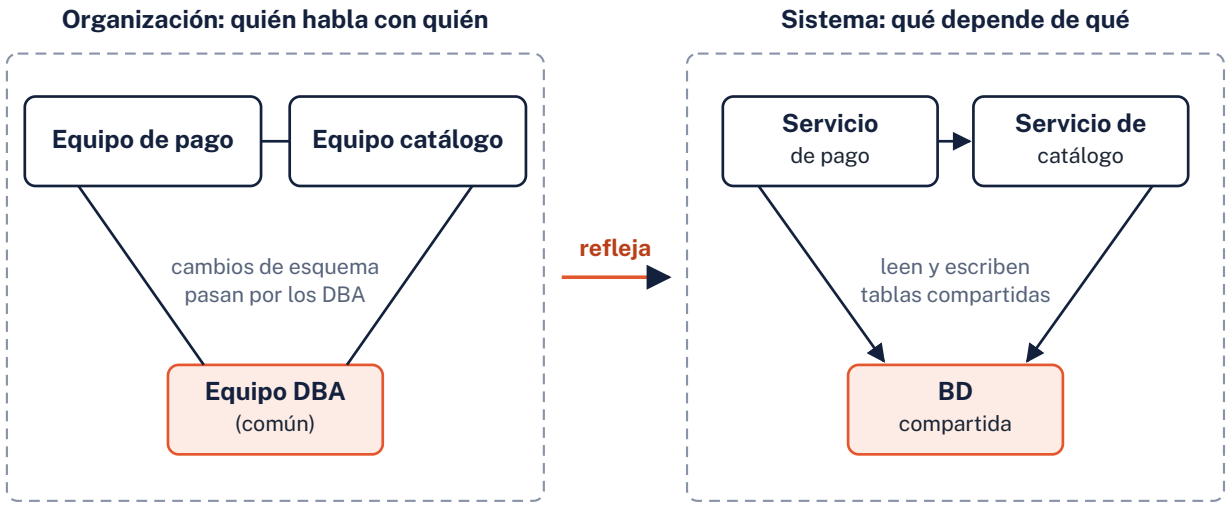
Lo que Conway dijo realmente

En abril de 1968 la revista *Datamation* publicó "How Do Committees Invent?" de Melvin Conway, programador y consultor informático. Su conclusión es la frase conocida hoy como ley de Conway: las organizaciones que diseñan sistemas están “obligadas a producir diseños que son copias de las estructuras de comunicación de esas organizaciones”. El nombre llegó después: Fred Brooks citó el artículo en *The Mythical Man-Month* (1975) y llamó a la idea ley de Conway.

El argumento de Conway es más preciso que el eslogan. Un sistema está formado por subsistemas conectados mediante interfaces. Para que dos subsistemas se conecten, las personas que los diseñan tienen que haber negociado y acordado esa interfaz, así que tienen que haberse comunicado. Si los diseñadores de dos subsistemas no se comunican, no hay ninguna interfaz que acordar. Existe, por tanto, una correspondencia que preserva la estructura (Conway usó el término matemático homomorfismo) entre el grafo del sistema y el grafo de la organización que lo diseñó. Su ejemplo: una organización de investigación por contrato tenía ocho personas para escribir un compilador de COBOL y otro de ALGOL. Cinco se asignaron a COBOL y tres a ALGOL. El compilador de COBOL funcionaba en cinco fases; el de ALGOL, en tres.

Dos puntos más del artículo suelen olvidarse. Primero, Conway dejó claro que "sistema" se entiende en sentido amplio: se aplica a cualquier cosa diseñada, no solo al software. Segundo, su conclusión práctica se refería a la **flexibilidad**. Como el primer diseño rara vez es el mejor, y como una organización tiende a fijar su estructura pronto (una vez formados los equipos, el diseño ya está decidido en parte), sostuvo que las organizaciones de diseño deben mantenerse ligeras y flexibles, y organizarse según la comunicación que el diseño realmente necesita.

El espejo



La estructura del sistema copia la estructura de la comunicación

Cuando dos equipos solo pueden coordinarse a través de un equipo de base de datos compartido, el sistema que construyen tiende a acoplarse a través de una base de datos compartida.

SI ESTO ES NUEVO PARA TI

Si esto es nuevo para ti

Imagina que tres departamentos de una administración local que nunca hablan entre sí construyen cada uno una parte de un sitio web. Probablemente obtendrás tres secciones con menús distintos, inicios de sesión distintos y formularios que te piden la dirección tres veces. Nadie lo planificó; es lo que ocurre cuando las únicas uniones de un producto están donde las personas que lo construyen hablaron de verdad. La ley de Conway dice que esto no es un accidente, sino una tendencia que puedes predecir y, por tanto, aprovechar.

La evidencia: la hipótesis del espejo

Durante décadas, la ley de Conway se apoyó sobre todo en anécdotas. Desde la década de 2000 los investigadores la han puesto a prueba, normalmente con el nombre de hipótesis del espejo (mirroring hypothesis): las dependencias técnicas de un producto coincidirán con los vínculos organizativos (comunicación, ubicación compartida, relación laboral) entre las personas que lo construyen.

El estudio más conocido sobre software es el de Alan MacCormack, Carliss Baldwin y John Rusnak, publicado en *Research Policy* en 2012 como "Exploring the duality between product and organizational architectures: a test of the mirroring hypothesis". Compararon pares

equivalentes de productos con funciones similares, uno construido por una organización fuertemente acoplada (una empresa comercial con un equipo en un mismo lugar) y otro por una organización débilmente acoplada (una comunidad de código abierto distribuida): por ejemplo, dos paquetes de gestión financiera, dos procesadores de texto, dos bases de datos, y Linux frente a núcleos de sistemas operativos comerciales. Midieron la modularidad mediante el *costo de propagación*, la proporción media de los archivos de un sistema a los que un cambio en un archivo puede afectar directa o indirectamente. En todos los pares, el producto de la organización débilmente acoplada era más modular, y en el par más extremo la diferencia era de varias veces. Los autores lo interpretan como evidencia de que los equipos distribuidos, que no pueden coordinarse con facilidad, se ven empujados hacia diseños que necesitan menos coordinación.

Un estudio de Microsoft Research sobre Windows Vista, de Nachiappan Nagappan, Brendan Murphy y Victor Basili (2008), abordó la cuestión desde el lado de la calidad. Las métricas que describían la organización que construyó cada binario (cuántos ingenieros lo tocaron, cuántos se habían ido, a qué distancia en la jerarquía de gestión estaban los contribuyentes) predecían qué binarios eran propensos a fallos mejor que las métricas de código convencionales, como la rotación de código (churn), la complejidad, la cobertura y las dependencias.

La evidencia no es unánime, y el resumen honesto importa. Una revisión de 2016 de 142 estudios empíricos, de Lyra Colfer y Carliss Baldwin, encontró que el reflejo era común en los estudios de industrias y empresas, pero no universal, y que los estudios de proyectos colaborativos abiertos a menudo *no* lo respaldaban, probablemente porque los repositorios de código compartidos, los gestores de incidencias y el debate abierto permiten a los contribuyentes distribuidos coordinarse sin los vínculos organizativos habituales. El reflejo es una tendencia fuerte, no una ley de la física, y unas buenas herramientas y prácticas pueden romperlo en parte.

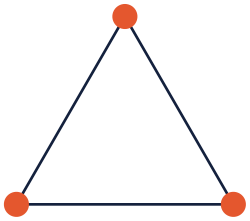
Las vías de comunicación y por qué importa el tamaño del equipo

Un grupo de n personas tiene $n(n - 1) / 2$ posibles vías de comunicación de persona a persona. El número crece aproximadamente con el cuadrado del grupo: duplicar el equipo cuadruplica las vías. Brooks usó la misma aritmética para explicar por qué añadir personas a un proyecto retrasado lo retrasa aún más: cada recién llegado debe recibir formación y añade nuevas vías que todos deben mantener.

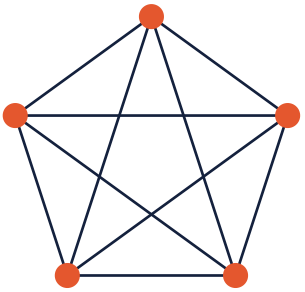
Desde el punto de vista arquitectónico, la aritmética explica dos cosas. Primero, un solo equipo no puede crecer indefinidamente y seguir manteniendo una comprensión compartida de su código; a partir de cierto tamaño se fragmenta en subgrupos informales y, por la ley de Conway, el código se fragmenta por las mismas líneas, tanto si alguien diseñó esas costuras como si no. Segundo, el número de vías entre *equipos* importa tanto como dentro de ellos: si

cada equipo tiene que hablar con todos los demás para entregar un cambio, la estructura del sistema es, en la práctica, una única unidad grande y acoplada, por muchos servicios que tenga.

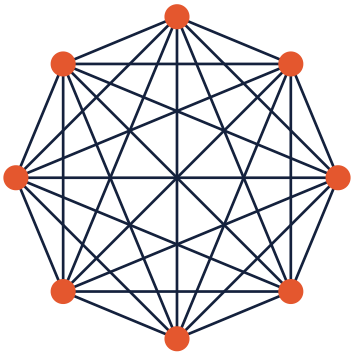
Las vías crecen mucho más deprisa que las personas



3 personas
3 vías



5 personas
10 vías



8 personas
28 vías

Cada par de personas es una posible vía de comunicación: 3 personas tienen 3, 5 tienen 10 y 8 tienen 28.

EJEMPLO PRÁCTICO

Ejemplo práctico: vías de comunicación en un grupo de producto que crece

El producto de pedidos de una empresa de reparto de comida lo construye un equipo de 8 ingenieros. Tiene previsto crecer hasta 24 ingenieros. Compara un equipo grande con tres equipos de 8 con una propiedad clara.

Un equipo de 8	8 × 7 / 2 = 28 vías.
Un equipo de 24	24 × 23 / 2 = 276 vías: unas diez veces más para el triple de personas.
Tres equipos de 8, vías internas	3 × 28 = 84 vías dentro de los equipos.
Tres equipos, vías entre equipos	Si cada equipo tiene un único punto de contacto con cada uno de los otros equipos (por ejemplo, el líder técnico), eso añade 3 × 2 / 2 = 3 vías entre equipos. Total: unas 87.
Tres equipos, todos se coordinan con todos	Si la propiedad no está clara y cada cambio necesita a todos los ingenieros, el grafo vuelve a ser, en la práctica, el grafo de 276 vías.
Lo que muestra	Dividir a las personas en equipos solo ayuda si el sistema se divide de forma equivalente, de modo que la mayoría de los cambios solo necesiten comunicación dentro del equipo. La arquitectura es lo que hace posible el grafo más pequeño.

La fórmula da vías posibles, no conversaciones reales; lo importante es lo rápido que puede crecer el costo de coordinación, no una predicción precisa.

Team Topologies (Skelton y Pais) propone equipos estables de aproximadamente 5 a 9 personas, algo coherente con la aritmética: lo bastante pequeños para que las vías sigan siendo manejables.

La maniobra inversa de Conway

Si las organizaciones tienden a producir sistemas que reflejan su estructura, puedes usar esa tendencia de forma deliberada: **diseña la estructura de equipos y las vías de comunicación para la arquitectura que quieres**, y deja que el sistema crezca con esa forma. Jonny LeRoy y Matt Simons llamaron a esto maniobra inversa de Conway en un artículo del *Cutter IT Journal* de diciembre de 2010, y desde entonces se ha convertido en una técnica habitual en los programas de microservicios y de plataforma.

En la práctica, significa decidir primero los límites previstos (por ejemplo, pedidos, pagos, seguimiento de entregas y alta de restaurantes como servicios separados), después formar un equipo estable en torno a cada uno, darle la propiedad del código, los datos y el pipeline de despliegue, y hacer que los equipos interactúen principalmente mediante API y eventos publicados, en lugar de código compartido o bases de datos compartidas. Donde el contacto entre equipos sea realmente necesario, hazlo explícito: un equipo de plataforma que ofrece un producto de autoservicio, un equipo facilitador que ayuda durante un tiempo limitado.

La maniobra tiene límites. Solo funciona lentamente a favor de la veta de un sistema existente: reorganizar los equipos no reestructura un código fuertemente acoplado y, durante un tiempo, los nuevos equipos estarán luchando contra la arquitectura antigua. Martin Fowler señala que no puede arreglar rápidamente un sistema existente rígido. También supone que conoces los límites correctos; si los trazas mal y luego los consolidas en estructuras de equipo, la ley de Conway reproducirá fielmente el error y hará que sea caro cambiarlo. Por eso los equipos suelen empezar con un monolito modular y mover los límites antes de convertirlos en organizativos.

Carga cognitiva del equipo

La ley de Conway te dice que los límites de los equipos se convierten en límites del sistema. Una segunda restricción te dice qué tamaño puede tener la parte de cada equipo. La carga cognitiva del equipo, un concepto que Matthew Skelton y Manuel Pais llevaron al diseño de organizaciones de software en *Team Topologies* (2019; segunda edición en 2025), es la cantidad total que un equipo debe tener en la cabeza para construir, operar y cambiar lo que posee: el dominio de negocio, el código, la infraestructura, las herramientas, las guardias.

Toman de la psicología la distinción entre la carga **intrínseca** (la dificultad inherente de la tarea, como entender el dominio), la carga **extrínseca** (el esfuerzo que impone el entorno, como un procedimiento de despliegue manual o una compilación oscura) y la carga **pertinente** (el esfuerzo de aprendizaje que mejora el trabajo futuro). El objetivo de diseño es minimizar la carga extrínseca, por ejemplo mediante plataformas y automatización, para que la capacidad se dedique a las de tipo intrínseco y pertinente.

Para los arquitectos, la consecuencia es directa. **El límite de un servicio o componente debe dar al equipo que lo posee una parte del sistema que pueda entender por completo.** Si un equipo posee doce servicios repartidos en cuatro dominios, el equipo está sobrecargado y se resentirán la calidad, la velocidad o la salud de las guardias, por limpio que sea el código de cada servicio. A la inversa, un dominio tan complicado que ningún equipo alineado al flujo puede absorberlo (un motor de precios con modelos actuariales, un códec de video) puede justificar un equipo especializado y un límite claro en torno a su subsistema.

Dejar que la estructura de equipos dé forma a los límites

En conjunto, estas ideas ofrecen a los arquitectos un conjunto de reglas de trabajo:

Alinea los límites con la propiedad de los equipos. Cada módulo, servicio, almacén de datos y pipeline debe tener un único equipo propietario. La propiedad compartida, en la práctica, significa ausencia de propiedad, y el código reflejará lo que necesitara el último que contribuyó.

Minimiza la comunicación que necesita cada cambio. Un límite está en el lugar correcto si la mayoría de los cambios en una capacidad de negocio puede hacerlos, probarlos y desplegarlos un solo equipo. Vigila los cambios que habitualmente necesitan que se coordinen tres equipos: muestran que la estructura del sistema y la estructura de equipos no coinciden.

Haz que las interfaces entre equipos sean explícitas y estables. Las interfaces entre equipos merecen versionado, documentación y pruebas de contrato; las interfaces dentro de un equipo pueden seguir siendo informales y cambiar libremente.

Revisa ambas cosas a la vez. Las reorganizaciones son decisiones de arquitectura; las decisiones de arquitectura son decisiones organizativas. Un arquitecto al que no se consulta sobre los cambios de equipos, o un responsable que reorganiza sin tener en cuenta el código, está ignorando una fuerza que actuará de todos modos.

Los patrones de tipos de equipo y modos de interacción, y cómo hacerlos evolucionar, se tratan en Team Topologies y diseño sociotécnico.

Cuando el organigrama y la arquitectura no coinciden

Situación	Lo que ves en el sistema	Remedio habitual
Equipos organizados por capa técnica (frontend, backend, base de datos)	Cada funcionalidad necesita tres equipos y tres traspasos; las capas desarrollan interfaces gruesas y genéricas; los lanzamientos se coordinan	Equipos multifuncionales alineados con las capacidades de negocio, cada uno propietario de una porción vertical
Microservicios con un equipo propietario por servicio, pero un equipo central de datos es dueño de todos los esquemas	Un sistema distribuido que se acopla a través de una base de datos compartida; los cambios de esquema esperan en cola durante semanas	Dar a cada equipo de servicio la propiedad de sus datos; el equipo de datos proporciona una plataforma y unos estándares
Dos equipos comparten la propiedad de un servicio	Convenciones contradictorias dentro de un mismo código, conflictos de fusión, nadie de guardia para las partes difíciles	Dividir el servicio según las inquietudes de cada equipo, o dárselo a un solo equipo
Un equipo es dueño de muchos servicios sin relación entre sí	Servicios descuidados, dependencias desactualizadas, respuesta lenta a incidentes: sobrecarga cognitiva	Consolidar servicios, pasar algunos a otros equipos o invertir en herramientas de plataforma
Un proveedor externo construye un subsistema con un contrato cerrado	Una interfaz rígida y muy especificada que es muy cara de cambiar	Tratar el límite como una puerta de un solo sentido; situarlo donde el cambio sea menos probable
Una reorganización traslada una capacidad a un nuevo equipo, pero el código se queda en el repositorio del equipo anterior	El nuevo equipo sortea código que no es suyo; erosión del límite previsto	Trasladar el código, los pipelines y las guardias junto con la responsabilidad

ATENCIÓN

Malentendidos habituales

- La ley de Conway no es una recomendación para copiar el organigrama actual; es la descripción de una fuerza. La decisión es si dejar que la organización existente dé forma al sistema o dar forma a la organización para el sistema que necesitas.
- Los microservicios no escapan a la ley. Cien servicios construidos por equipos que tienen que coordinarse todos entre sí se comportan como un solo sistema, con el costo añadido de una red.
- La ley trata de la estructura de comunicación, no de las líneas jerárquicas formales. Dos equipos con responsables distintos que se sientan juntos y hablan a diario pueden producir partes muy integradas.
- El trabajo remoto y distribuido no anula la ley; cambia qué comunicación resulta barata. Las interfaces asíncronas, escritas y bien documentadas se convierten en los límites naturales.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ En el argumento de Conway, ¿por qué la estructura del sistema tiene que parecerse a la estructura de la organización?

Porque cada interfaz entre subsistemas tiene que ser negociada por las personas que los diseñan; donde las personas no se comunican no se diseña ninguna interfaz, así que las conexiones del sistema siguen a las conexiones entre personas.

▼ ¿Cuántas vías de comunicación tiene un equipo de 12 personas?

12 × 11 / 2 = 66.

▼ Una empresa se reorganiza en un equipo por microservicio, pero mantiene una única base de datos compartida que todos los equipos modifican. ¿Qué predice la ley de Conway?

Los servicios seguirán acoplados a través de la base de datos y los cambios necesitarán coordinación entre todos los equipos, porque la estructura de comunicación (todos tocan el mismo esquema) no ha cambiado.

▼ ¿Qué es la carga cognitiva extrínseca y cómo puede reducirla la arquitectura?

El esfuerzo impuesto por el entorno y no por el problema, como los despliegues manuales o unas herramientas oscuras; las plataformas, la automatización y unos límites claros la reducen para que los equipos puedan centrarse en su dominio.

▼ **Inténtalo: Un grupo de venta de entradas de 10 ingenieros crecerá hasta 30. Compara las vías de comunicación de un solo equipo de 30 con las de tres equipos de 10 que tienen un enlace entre cada par de equipos. ¿Qué más debe cambiar en el software?**

Un equipo de 10 tiene 10 x 9 / 2 = 45 vías de comunicación. Un equipo de 30 tiene 30 x 29 / 2 = 435. Tres equipos de 10 tienen 3 x 45 = 135 dentro de los equipos, más 3 enlaces, es decir, 138. Pero esto solo funciona si cada equipo es dueño de una parte correspondiente del sistema con interfaces claras. De lo contrario, los equipos deben coordinarse constantemente, así que planifica juntos los límites de los módulos y la división de los equipos.

FUENTES

- Melvin Conway: How Do Committees Invent? (Datamation, April 1968): https://www.melconway.com/Home/Committees_Paper.html
- Martin Fowler: Conway's Law: https://martinfowler.com/bliki/ConwaysLaw.html
- MacCormack, Baldwin and Rusnak: Exploring the duality between product and organizational architectures (Research Policy, 2012): https://doi.org/10.1016/j.respol.2012.04.011
- Team Topologies: key concepts: https://teamtopologies.com/key-concepts
- Thoughtworks Technology Radar: Inverse Conway Maneuver: https://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver

SECCIÓN 3.8 DE 3.8 · INTERMEDIO · 25 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Simplicidad, reversibilidad y medición de la estructura

ANTES DE LEER ESTO

Esta sección se basa en:

- 1.6 Decisiones y compromisos
- 1.7 Deuda técnica
- 3.4 Dependencias

Términos anteriores que utiliza: Acoplamiento aferente (Ca), Inestabilidad (I), Ciclo de dependencias, Acoplamiento, Refactorización

LA IDEA EN BREVE

Algunas cosas son difíciles porque el trabajo es difícil: una declaración de impuestos es complicada porque la legislación fiscal lo es. Otras cosas son difíciles por cómo decidimos hacerlas: un sistema de archivo con siete carpetas donde bastarían dos. Los buenos diseñadores aceptan el primer tipo y combaten el segundo. Construyen lo que se necesita ahora, dejan cada dato escrito en un solo lugar, evitan las decisiones que no pueden deshacer y comprueban su trabajo con unas pocas mediciones honestas, sin olvidar que un número puede manipularse si se convierte en el objetivo.

LO QUE NECESITAS SABER

- Fred Brooks distinguió la complejidad esencial, que pertenece al problema, de la complejidad accidental, que procede de cómo se construye la solución; los arquitectos ubican la primera y eliminan la segunda.
- YAGNI advierte contra construir para necesidades supuestas por los costos de construir, del retraso, de mantenerlo y de repararlo, pero no justifica descuidar las pruebas, la refactorización ni las decisiones de puerta de un solo sentido.
- DRY, tal como lo definen Hunt y Thomas, trata de que cada pieza de conocimiento tenga una única representación autorizada; fusionar código que solo se parece crea un acoplamiento perjudicial.
- La flexibilidad es una opción con un precio: págala cuando el cambio sea probable o muy caro sin ella y la opción sea barata, y confía en una buena modularidad para lo demás.
- La complejidad ciclomática, M = E − N + 2P, cuenta los caminos independientes a través de una función y equivale al número de decisiones más uno en el código estructurado; McCabe sugirió un límite de alrededor de 10.

- LCOM, las métricas de acoplamiento y estabilidad, el acoplamiento de cambio y los hotspots revelan cada uno problemas estructurales distintos, y todos se vuelven engañosos cuando se convierten en objetivos (ley de Goodhart).
- Las funciones de aptitud convierten decisiones estructurales, como las dependencias prohibidas, en comprobaciones automatizadas que hacen fallar la compilación cuando se incumplen.

Dos tipos de complejidad

En "No Silver Bullet: Essence and Accidents of Software Engineering" (presentado en el Congreso Mundial de Computación de la IFIP en 1986 y publicado en *IEEE Computer* en 1987), Fred Brooks separó las dificultades de construir software en dos tipos. La complejidad esencial pertenece al problema: las reglas reales del IVA, la variedad real de productos de seguros, las formas reales en que puede fallar una entrega. Ningún diseño puede eliminarla sin cambiar lo que hace el sistema. La complejidad accidental procede de cómo construimos la solución: lenguajes, herramientas, frameworks, maquinaria de despliegue, soluciones provisionales y una estructura más complicada de lo que el problema requiere. Brooks sostuvo que los lenguajes de alto nivel y otros avances ya habían eliminado gran parte de la dificultad accidental, por lo que ninguna técnica nueva por sí sola lograría una mejora de diez veces en una década.

Para los arquitectos, la distinción es una herramienta diaria. La complejidad esencial debe *ubicarse*: ponla donde puedan trabajar en ella las personas que la entienden, detrás de un límite que impida que se filtre a todo lo demás (un módulo de precios, un servicio de impuestos). La complejidad accidental debe *eliminarse o prevenirse*: cada servicio, cola, caché, capa de abstracción y opción de configuración adicional añade algo, y cada uno debe ganarse su lugar resolviendo un problema real. El ensayo de Ben Moseley y Peter Marks "Out of the Tar Pit" (2006) llevó la idea más lejos, al identificar el estado mutable y el flujo de control como fuentes principales de complejidad accidental, lo que es una de las razones por las que son populares los diseños funcionales y dirigidos por eventos que limitan el estado compartido.

El resto de esta sección trata los principios que mantienen baja la complejidad accidental (KISS, YAGNI, DRY bien entendido, reversibilidad) y las mediciones que ayudan a ver la estructura de forma objetiva, junto con sus límites.

KISS y YAGNI

KISS, que suele desarrollarse como "keep it simple, stupid" ("hazlo simple, estúpido"), es una máxima de diseño registrada en la Marina de los Estados Unidos en 1960 y asociada a Kelly Johnson, ingeniero jefe de Skunk Works de Lockheed, de quien se dice que exigía que los aviones a reacción pudiera repararlos sobre el terreno un mecánico medio con unas pocas

herramientas. La lectura para el software es la misma: un diseño es bueno si las personas que deben operarlo y repararlo, a menudo a las 3 de la madrugada y bajo presión, pueden entenderlo. Simple no significa fácil ni ingenuo. Significa pocos conceptos, pocas piezas móviles y pocos casos especiales para el problema que se resuelve.

YAGNI ("you aren't gonna need it", "no lo vas a necesitar") procede de Extreme Programming a finales de la década de 1990: no construyas capacidades para necesidades que solo supones. El análisis de Martin Fowler es útil porque nombra los costos de ignorarlo. Está el **costo de construir** la funcionalidad supuesta; el **costo del retraso**, porque no se construyó en su lugar algo valioso; el **costo de mantenerla**, porque cada cambio posterior debe sortear el código adicional; y, si la suposición era errónea, el **costo de repararla**. Incluso cuando la suposición resulta acertada, la funcionalidad se construyó con menos conocimiento del que habrías tenido más adelante.

Dos matices impiden que YAGNI se convierta en una excusa. Primero, se aplica a *las funcionalidades y a la flexibilidad especulativa*, no a las prácticas que abaratan el cambio. Las pruebas automatizadas, la refactorización, la entrega continua y unos límites de módulo claros son lo que hace seguro aplazar funcionalidades; abandonarlos en nombre de YAGNI convierte el aplazamiento en deuda. Segundo, algunas decisiones son puertas de un solo sentido (Sección 1.6). Elegir un modelo de datos, la forma de una API pública, un modelo de tenencia o un límite de seguridad no es una funcionalidad que puedas añadir después al mismo costo. Para esas decisiones, pensar con antelación no es especulación; es el trabajo del arquitecto.

DRY significa conocimiento, no texto

Andy Hunt y Dave Thomas introdujeron DRY en *The Pragmatic Programmer* (1999; edición del 20.º aniversario en 2019) con esta definición: “Cada pieza de conocimiento debe tener una representación única, inequívoca y autorizada dentro de un sistema.” La palabra clave es *conocimiento*. Si el tipo de IVA, la regla de un código postal válido o el significado de un estado de pedido están escritos en varios lugares, cambiarlos exige encontrar cada copia, y las copias irán divergiendo. La edición de 2019 subraya que DRY trata de la duplicación de conocimiento y de intención, no de líneas de código que casualmente se parecen.

El mal uso habitual va en sentido contrario. Dos funciones que se parecen se fusionan porque se parecen, aunque representan conocimientos distintos que cambiarán por razones distintas: la regla de la tarifa de envío y la regla de la comisión del restaurante multiplican hoy ambas un total por un porcentaje, pero pertenecen a actores distintos (finanzas y alianzas) y divergirán. Fusionarlas acopla dos conceptos que el dominio mantiene separados: ambas dependen ahora de una única función compartida, así que un cambio solicitado para una cambia la otra sin que nadie lo note. En los términos de la Sección 3.6, crea connascence donde el problema no la tiene (compartir una función es el remedio correcto solo cuando ambos lados deben calcular

realmente lo mismo); también es una violación de la responsabilidad única (Sección 3.5). El conocido consejo de Sandi Metz es preferir la duplicación a la abstracción equivocada, porque una abstracción compartida equivocada acumula parámetros y condicionales hasta que nadie se atreve a cambiarla. Muchos equipos usan la regla de tres: tolerar una segunda copia y extraer una abstracción cuando aparece la tercera y el conocimiento compartido está claro.

A escala de arquitectura, el compromiso se vuelve explícito. Una biblioteca compartida que todos los servicios deben usar elimina el código duplicado, pero acopla los ciclos de lanzamiento de los servicios. Muchas organizaciones de microservicios aceptan deliberadamente el *código* duplicado (pequeñas utilidades copiadas en cada servicio) mientras exigen un único propietario para el *conocimiento* duplicado (el registro canónico del cliente, las reglas fiscales). Pregúntate ante cualquier duplicación: si esta regla cambia, ¿deben cambiar todas las copias a la vez? Si la respuesta es sí, es conocimiento duplicado y debe tener un único propietario. Si es no, es una similitud casual y puede quedarse.

SI ESTO ES NUEVO PARA TI

Paso a paso: cuando fusionar código parecido sale mal

La idea más difícil de esta sección es que DRY trata del conocimiento, no del código que se parece. El Capítulo 0 mostró por qué una regla debe escribirse una sola vez; la dificultad está en distinguir una regla escrita dos veces de dos reglas que solo se parecen. Sigue el propio ejemplo de la sección.

- En la plataforma de reparto de comida, la regla de la tarifa de envío y la regla de la comisión del restaurante multiplican hoy ambas el total de un pedido por un porcentaje. La misma aritmética aparece dos veces.
- Recordando que una regla debe vivir en un solo lugar, un desarrollador las fusiona en una única función compartida. El código es más corto.
- Meses después, se solicita un cambio solo para la regla de la comisión: ahora la comisión tiene un importe máximo por pedido. El desarrollador añade el tope a la función compartida y las tarifas de envío también quedan limitadas sin que nadie lo note. Nadie pidió eso.
- Para arreglarlo, se añade un indicador para que la función sepa qué regla está calculando: el acoplamiento de control de la Sección 3.2. Cada solicitud posterior añade otro parámetro u otra condición, hasta que nadie se atreve a cambiar la función.
- Las dos reglas nunca fueron el mismo conocimiento: pertenecen a equipos distintos (finanzas y alianzas) y cambian por razones distintas. Dos funciones cortas y separadas eran el mejor diseño.
- Compáralo con el tipo de IVA escrito en el proceso de pago, la factura y la pantalla de reembolsos: cada copia debe cambiar siempre que cambie el tipo. Eso es una pieza de conocimiento escrita tres veces, y necesita un único propietario.
- **La lección:** la prueba es si todas las copias deben cambiar a la vez cuando cambia la regla. Si deben hacerlo, da a la regla un único hogar. Si no, el parecido es una coincidencia, y un poco de duplicación suele ser más barato que la abstracción compartida equivocada.

La reversibilidad y el costo de las opciones

La simplicidad tiene una dimensión temporal. Un diseño que hoy es simple pero imposible de cambiar no es barato; un diseño lleno de flexibilidad que nadie usa tampoco lo es. El concepto útil es la reversibilidad: ¿cuánto costaría deshacer o cambiar esta decisión más adelante? La Sección 1.6 presentó las puertas de un solo sentido y de doble sentido. Aquí lo importante es

que la flexibilidad se comporta como una opción financiera: tiene un precio ahora (código adicional, abstracción adicional, costo operativo adicional) y un valor que depende de lo probable y lo caro que sea el cambio contra el que protege.

Esto da una regla práctica que concilia YAGNI con la previsión. **Paga por una opción solo cuando el cambio sea razonablemente probable, o cuando sería muy caro sin la opción, y cuando la propia opción sea barata.** Envolver un proveedor de pagos tras una interfaz propia cuesta un día y protege frente a un cambio que es a la vez plausible y caro: una buena opción. Construir una capa de abstracción multinube para una startup con un solo cliente es una opción cara frente a un cambio improbable. A menudo la opción más barata es simplemente una buena modularidad: unos límites bien situados mantienen locales muchos cambios futuros sin que nadie tenga que predecirlos, que es el argumento original de Parnas a favor de la ocultación de información (Sección 3.1).

Medir la estructura: la complejidad ciclomática

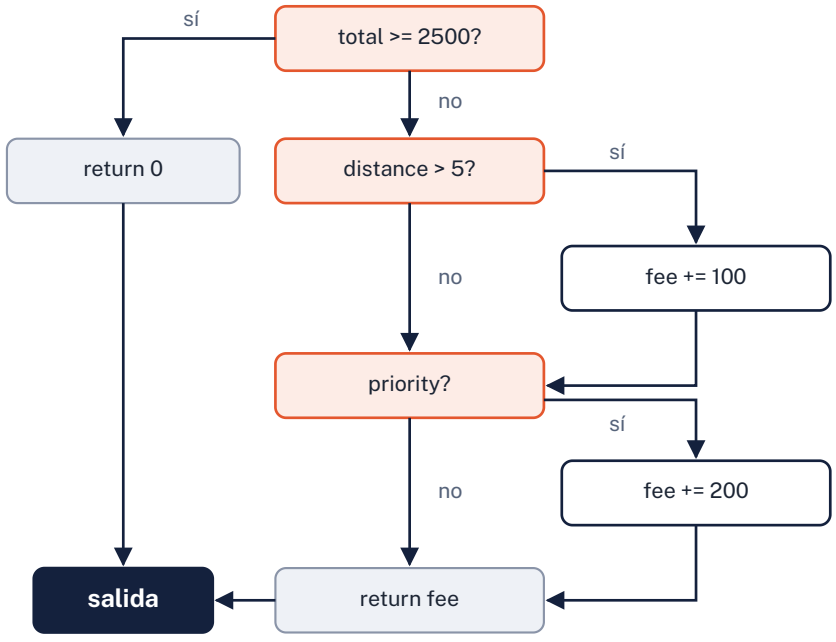
El juicio sobre la simplicidad mejora cuando se apoya en la medición. La métrica de código más antigua de uso extendido es la complejidad ciclomática, presentada por Thomas McCabe en "A Complexity Measure" (*IEEE Transactions on Software Engineering*, 1976). Trata una función como un grafo de flujo de control, con nodos para los bloques de código y aristas para las posibles transferencias de control, y cuenta los caminos linealmente independientes que la atraviesan: $M = E - N + 2P$, donde E es el número de aristas, N el número de nodos y P el número de componentes conexas (1 para una sola función). En el código estructurado con un único punto de entrada, esto equivale al número de puntos de decisión más uno.

McCabe sugirió 10 como límite superior razonable para un módulo. El posterior informe del NIST sobre pruebas estructuradas de Arthur Watson y McCabe (1996) señala que equipos experimentados han usado con éxito límites de hasta 15, y recomienda una explicación por escrito siempre que un módulo supere el límite elegido. Herramientas de análisis estático como `radon` o el plugin `mccabe` para flake8 en Python, la regla `complexity` de ESLint en JavaScript y SonarQube la calculan automáticamente. Las herramientas difieren en detalles, por ejemplo en si cada `and` u `or` de una condición cuenta como una decisión adicional, así que compara números de la misma herramienta.

```
def delivery_fee(total: int, distance_km: float, priority: bool) → int:
    if total ⩾ 2500:          # decision 1: free delivery over 25 pounds
        return 0
    fee = 299
    if distance_km > 5:        # decision 2
        fee += 100
    if priority:               # decision 3
        fee += 200
    return fee
```

Tres decisiones, así que la complejidad ciclomática es 4.

El grafo de flujo de control y el recuento



Recuento

Nodos N = 8

Aristas E = 10

M = E − N + 2

M = 10 − 8 + 2 = 4

Verif.: 3 decisiones + 1 = 4

4 caminos indep.

al menos 4 casos de prueba

(prueba de camino básico)

Ocho nodos y diez aristas dan M = 10 − 8 + 2 = 4, que coincide con tres decisiones más uno.

EJEMPLO PRÁCTICO

Ejemplo práctico: cuando la función crece

Un año después, `delivery_fee` ha incorporado un bucle sobre el carrito y una condición compuesta. Calcula su complejidad y decide qué hacer.

Código nuevo	`for item in items:` con `if item.bulky: fee += 50` dentro, y `if priority and rider_available: fee += 200` en lugar de `if priority`.
Decisiones	Las 3 originales, más el bucle `for` (1), más el `if item.bulky` (1), más el `and` (1, si la herramienta cuenta los operadores booleanos): 6.
Complejidad ciclomática	6 + 1 = 7. Por debajo del límite de 10 de McCabe, pero casi se ha duplicado.
Implicación para las pruebas	La prueba de camino básico necesita al menos 7 casos de prueba para recorrer una vez cada camino independiente; el antiguo conjunto de 4 ya no cubre todos los caminos.
Lectura de diseño	La función mezcla ahora tres inquietudes de precios: el umbral según el importe del carrito, los recargos por distancia y por artículo, y la prioridad. Cambian por solicitudes de áreas distintas (marketing, operaciones, producto).
Refactorización	Extrae `distance_surcharge`, `bulky_surcharge` y `priority_surcharge`, cada una con complejidad 2 o 3, y deja que `delivery_fee` las sume. La complejidad total es parecida, pero cada pieza es pequeña, tiene nombre y se puede probar por separado.

La refactorización rara vez reduce la complejidad total; la redistribuye en piezas que son más fáciles de entender, probar y cambiar de forma independiente. La complejidad esencial de las reglas de precios se mantiene.

La métrica señaló una tendencia que merecía una conversación; la decisión de dividir se tomó leyendo el código y preguntando quién lo cambia.

Medir la estructura más allá de las funciones

La complejidad ciclomática ve una función cada vez. Los arquitectos necesitan medidas de cómo se relacionan los módulos.

Métricas de acoplamiento y estabilidad. La Sección 3.4 (gestionar las dependencias) presentó el acoplamiento aferente y eferente, la inestabilidad $I = Ce / (Ca + Ce)$, el grado de abstracción y la distancia a la secuencia principal. Son baratas de calcular a partir de los grafos de

importaciones y sirven bien para revelar componentes de los que dependen muchos y que, a su vez, dependen de código volátil, y para detectar ciclos de dependencias.

Métricas de cohesión. La familia LCOM (falta de cohesión de los métodos) estima si una clase tiene una responsabilidad o varias. La versión original de Shyam Chidamber y Chris Kemerer (1994) compara pares de métodos según si usan alguno de los mismos campos. Una variante posterior muy usada, LCOM4, de Martin Hitz y Behzad Montazeri, construye un grafo en el que los métodos están conectados si comparten un campo o si uno llama al otro, y cuenta los grupos conectados. Una clase con LCOM4 = 1 es coherente; una clase con LCOM4 = 3 es, en realidad, tres clases que comparten un archivo. Por ejemplo, una clase `Customer` cuyos `update_address` y `format_address` usan los campos de dirección, cuyos `add_points` y `points_balance` usan los campos de fidelización, y cuyo `send_newsletter` usa solo el campo de correo electrónico tiene LCOM4 = 3, lo que sugiere responsabilidades de dirección, fidelización y marketing.

Métricas basadas en el historial. El control de versiones registra cómo cambia realmente el sistema, algo que el análisis estático no puede ver. El acoplamiento de cambio encuentra archivos o servicios que se confirman juntos una y otra vez: si `pricing.py` e `invoice_template.html` cambian juntos en el 70% de los commits, están acoplados tanto si uno importa al otro como si no, y el acoplamiento puede cruzar un límite de servicio que debía separarlos. Los hotspots (puntos calientes) combinan la frecuencia de cambio con la complejidad para encontrar código que es a la vez difícil de trabajar y que se trabaja con frecuencia; *Your Code as a Crime Scene* (2015; segunda edición en 2024), de Adam Tornhill, popularizó la técnica. La Sección 1.7, sobre deuda técnica, mostró cómo los hotspots dirigen el pago hacia la deuda que genera más intereses.

Métricas estructurales de un vistazo

Métrica	Alcance	Lo que revela	Principal punto ciego
Complejidad ciclomática	Función	Lógica de ramificación difícil de probar y de razonar	No dice nada de los nombres, los datos ni cómo se relacionan las funciones
Ca, Ce, inestabilidad, distancia	Paquete o componente	Dirección de las dependencias, estabilidad, ciclos	Cuenta las dependencias, no su fuerza ni su significado
LCOM (por ejemplo, LCOM4)	Clase	Clases que tienen varias responsabilidades sin relación entre sí	Sensible a los métodos de acceso y a las características del lenguaje; ruidosa en clases pequeñas
Acoplamiento de cambio	Archivo, módulo o servicio	Acoplamiento oculto que solo se ve en cómo cambia el sistema	Necesita suficiente historial; los commits masivos generan ruido
Hotspots	Archivo o módulo	Dónde la mejora se amortiza más rápido	Prioriza; no dice qué está mal
Connascence (evaluada mediante revisión)	Cualquiera	La forma y la fuerza de cada dependencia	En gran parte manual; pocas herramientas miden las formas dinámicas

La ley de Goodhart y los límites de las métricas

Cada una de las métricas anteriores es un indicador indirecto de algo que nos importa pero que no podemos medir directamente: lo fácil que es entender y cambiar el sistema. La ley de Goodhart, que lleva el nombre del economista Charles Goodhart (1975) y es más conocida en la formulación de 1997 de la antropóloga Marilyn Strathern, advierte que cuando una medida se convierte en un objetivo, deja de ser una buena medida. Las personas optimizan el número, no la propiedad.

Los equipos de software lo ven constantemente. Un límite estricto de complejidad ciclomática de 10 produce funciones divididas en puntos arbitrarios, cada una de las cuales supera la comprobación mientras la lógica se vuelve más difícil de seguir. Un objetivo de cobertura del 80% produce pruebas que ejecutan código sin comprobar nada. Una regla según la cual ninguna clase puede superar las 200 líneas produce clases divididas en `OrderHelper` y `OrderHelper2`. Un objetivo de "cero ciclos" puede cumplirse moviendo código a un paquete `common` convertido en cajón de sastre del que todo depende.

Los profesionales usan las métricas de tres formas que resisten la manipulación. Miran las **tendencias** en lugar de umbrales absolutos (la complejidad del módulo de pagos se ha duplicado en seis meses: ¿por qué?). Usan las métricas para **iniciar conversaciones y priorizar revisiones**, no para calificar a las personas. Y **combinan** métricas que tiran en direcciones distintas, como la complejidad con la frecuencia de cambio, o el acoplamiento con la propiedad de los equipos, de modo que ningún número por sí solo pueda optimizarse a costa de los demás.

Funciones de aptitud: hacer que la estructura se pueda comprobar

Algunas reglas estructurales son lo bastante claras para automatizarlas, y entonces dejan de depender de la vigilancia. *Building Evolutionary Architectures* (2017; segunda edición en 2022 con Pramod Sadalage), de Neal Ford, Rebecca Parsons y Patrick Kua, llama a estas comprobaciones funciones de aptitud (fitness functions): pruebas objetivas y automatizadas de una característica arquitectónica que se ejecutan en el pipeline de compilación o contra el sistema en ejecución. Algunos ejemplos para la estructura: ningún ciclo de dependencias entre paquetes de nivel superior; el paquete del dominio no importa nada de los paquetes web ni de base de datos; ninguna función por encima de una complejidad de 15 sin una excepción explícita; ningún servicio lee las tablas de la base de datos de otro servicio.

Herramientas como ArchUnit (Java), import-linter (Python) y dependency-cruiser (JavaScript y TypeScript) expresan estas reglas como código; la Sección 3.4 dio ejemplos. La diferencia con una métrica de un panel es que una función de aptitud codifica una *decisión* que el equipo ya ha tomado, así que incumplirla hace fallar la compilación y obliga a una elección consciente: arreglar el código o cambiar la regla y dejar constancia del motivo. El tratamiento completo, incluidas las funciones de aptitud para el rendimiento, la seguridad y otras cualidades, está en Funciones de aptitud y arquitectura evolutiva.

```
[importlinter]
root_package = shop

[importlinter:contract:domain-is-pure]
name = Domain must not depend on infrastructure
type = forbidden
source_modules =
    shop.domain
forbidden_modules =
    shop.infrastructure
    shop.web
```

Un contrato de import-linter en setup.cfg o .importlinter que hace fallar la compilación si la capa de dominio importa infraestructura.

Lista de comprobación: revisar el diseño de un módulo

Úsala al revisar un nuevo módulo, componente o servicio, o un cambio importante en uno de ellos. Se basa en todas las secciones de este capítulo.

- [ ] **¿Puedes enunciar su responsabilidad en una frase y nombrar al actor que solicita sus cambios?**
  Si la frase necesita una "y", o varios actores lo cambian, plantéate dividirlo (modularidad, SRP, CCP).
- [ ] **¿Oculta una decisión de diseño que probablemente cambiará?**
  Un módulo debe ser dueño de un secreto, como un formato de datos, un proveedor o un algoritmo (ocultación de información).
- [ ] **¿Es la interfaz pequeña en relación con lo que oculta?**
  Los módulos profundos ocultan mucho detrás de poco; las capas de paso añaden costo sin ocultar nada.
- [ ] **¿Apuntan sus dependencias hacia módulos más estables y más abstractos?**
  Comprueba que no haya ciclos ni módulos estables que dependan de otros volátiles (ADP, SDP, DIP).
- [ ] **¿Qué necesita saber un cliente para usarlo correctamente?**
  Busca parámetros posicionales, valores mágicos, órdenes de llamada obligatorios y supuestos de temporización; debilítalos o mantenlos locales (connascence).
- [ ] **¿Quién es su propietario, y puede ese equipo cambiarlo y desplegarlo sin coordinarse con otros?**
  Un único equipo propietario por módulo; las interfaces entre equipos deben ser explícitas y estar versionadas (ley de Conway).
- [ ] **¿Hay alguna parte de su complejidad que sea accidental?**
  Cuestiona cada capa, opción y mecanismo genérico adicional: ¿a qué requisito real sirve (KISS, YAGNI)?
- [ ] **¿Hay algún conocimiento duplicado, o alguna similitud casual compartida por error?**
  El conocimiento duplicado necesita un único propietario; el código de aspecto similar con propietarios distintos puede seguir separado (DRY).
- [ ] **¿Qué decisiones de él son difíciles de revertir, y son deliberadas?**
  La forma de las API públicas, los modelos de datos y los límites merecen más escrutinio que los detalles internos.

- [ ] **¿Qué dicen las métricas y el historial?**
  Comprueba las tendencias de complejidad, las métricas de acoplamiento, el acoplamiento de cambio y los hotspots, y añade una función de aptitud para cualquier regla que merezca la pena mantener.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ Una función tiene dos sentencias `if` y un bucle `while`, y ningún operador booleano. ¿Cuál es su complejidad ciclomática?

Tres decisiones más uno, así que 4.

▼ Dos servicios contienen cada uno una función idéntica de 20 líneas para dar formato a fechas. ¿Es una violación de DRY?

Probablemente no. Es texto duplicado, no conocimiento de negocio duplicado, y compartirlo acoplaría los lanzamientos de los servicios; las reglas de negocio duplicadas, como el cálculo de impuestos, son otra cuestión.

▼ ¿Por qué YAGNI no se aplica a escribir pruebas automatizadas?

Las pruebas forman parte de lo que hace que el cambio futuro sea barato y seguro; YAGNI trata de no construir funcionalidades ni flexibilidad especulativas, y depende de esas prácticas para funcionar.

▼ A un equipo se le dice que ninguna función puede superar una complejidad de 10, y la complejidad baja mientras los defectos aumentan. ¿Qué ha ocurrido probablemente?

La ley de Goodhart: las funciones se dividieron mecánicamente para cumplir la métrica, lo que hizo la lógica más difícil de seguir sin hacerla más simple.

▼ Inténtalo: Dos servicios tienen cada uno una comprobación de direcciones de 20 líneas que parece casi idéntica. Una comprueba las direcciones de los restaurantes para el registro legal; la otra comprueba si un repartidor puede llegar a una puerta. Un colega quiere una biblioteca compartida. ¿Estás de acuerdo?

Probablemente no. DRY se refiere a que cada pieza de conocimiento tenga un solo lugar, y aquí el código se parece, pero expresa dos reglas distintas, que cambian por razones distintas. Unirlas las acoplaría, y el código compartido pronto acumularía indicadores para cada caso. Conserva las dos copias, quizá compartiendo solo un analizador neutro de códigos postales si de verdad es el mismo conocimiento. Revísalo si aparece una tercera copia y el conocimiento compartido queda claro.

FUENTES

- Fred Brooks: No Silver Bullet (UNC technical report version, 1986): https://techreports.cs.unc.edu/papers/86-020.pdf
- Martin Fowler: Yagni: https://www.martinfowler.com/bliki/Yagni.html
- Sandi Metz: The Wrong Abstraction: https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction
- Watson and McCabe: Structured Testing (NIST Special Publication 500-235, 1996): https://www.mccabe.com/pdf/mccabe-nist235r.pdf
- Martin Fowler: Beck Design Rules: https://martinfowler.com/bliki/BeckDesignRules.html

Lecturas recomendadas

- *On the Criteria To Be Used in Decomposing Systems into Modules*, David L. Parnas (1972), Communications of the ACM, volume 15, issue 12. Seis páginas breves que presentaron la ocultación de información a través del ejemplo KWIC; sigue siendo el mejor argumento para descomponer por cambios probables.
  https://doi.org/10.1145/361598.361623
- *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Edward Yourdon and Larry L. Constantine (1979), Yourdon Press / Prentice-Hall. El libro que convirtió el acoplamiento y la cohesión en vocabulario estándar, con el análisis original de sus niveles.
- *How Do Committees Invent?*, Melvin E. Conway (1968), Datamation (April 1968). La formulación original de la ley de Conway, disponible gratuitamente en el sitio web del autor y con más matices que el resumen de una línea.
  https://www.melconway.com/Home/Committees_Paper.html
- *A Philosophy of Software Design (2nd edition)*, John Ousterhout (2021), Yaknyam Press. Un libro breve y con opiniones firmes sobre cómo reducir la complejidad, origen de los módulos profundos y superficiales.
  https://web.stanford.edu/~ouster/cgi-bin/book.php
- *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Robert C. Martin (2017), Prentice Hall. Reúne en un solo lugar los principios SOLID, los principios de cohesión y de acoplamiento de componentes y las métricas de inestabilidad y de grado de abstracción.
- *Balancing Coupling in Software Design*, Vlad Khononov (2024), Addison-Wesley. Una síntesis moderna del diseño estructurado, la connascence y el diseño guiado por el dominio que juzga el acoplamiento por su fuerza, su distancia y su volatilidad a la vez.
- *Software Engineering at Google: Lessons Learned from Programming Over Time*, Titus Winters, Tom Manshreck and Hyrum Wright (eds) (2020), O'Reilly Media. Se puede leer gratis en línea; trata la ley de Hyrum y ofrece una explicación excelente de cómo mantener sanas las interfaces y las dependencias a gran escala.
  https://abseil.io/resources/swe-book
- *Building Evolutionary Architectures (2nd edition)*, Neal Ford, Rebecca Parsons, Patrick Kua and Pramod Sadalage (2022), O'Reilly Media. Muestra cómo proteger las decisiones estructurales, como las reglas de dependencia, con funciones de aptitud automatizadas.

Glosario

Abstracción
Una visión simplificada de algo que conserva lo que importa para un propósito y omite el resto, como un "archivo" que oculta discos, sectores y caché.

Abstracción con fugas (leaky abstraction)
Abstracción cuyos detalles ocultos siguen afectando a quienes la usan; por ejemplo, una llamada remota que parece local hasta que la red es lenta o falla. Bautizada por Joel Spolsky en 2002.

Acoplamiento aferente (Ca)
El número de elementos externos a un componente que dependen de él: sus dependencias entrantes. Un Ca alto significa que muchos otros se ven afectados si cambia.

Acoplamiento de despliegue
Una situación en la que dos partes deben lanzarse juntas, o en un orden fijo, porque un cambio en una rompe la otra.

Acoplamiento eferente (Ce)
El número de elementos externos a un componente de los que este depende: sus dependencias salientes. Un Ce alto significa que está expuesto a los cambios de muchos otros.

Acoplamiento semántico
Dependencia de un significado o de reglas de negocio compartidos que no es visible en ninguna interfaz, como dos servicios que deben coincidir en qué significa 'cliente activo'.

Acoplamiento temporal
En los sistemas distribuidos, la dependencia de que otra parte esté disponible y responda en el mismo momento, como ocurre con una llamada síncrona. En otros ámbitos el término puede significar un orden obligatorio de llamadas o (en el análisis del control de versiones) archivos que cambian juntos, lo que este capítulo llama acoplamiento de cambio.

Backend for frontend (BFF)
Una API del lado del servidor construida para un tipo de cliente, como la app móvil, de modo que cada cliente obtenga operaciones y datos adaptados a sus propias necesidades.

Carga cognitiva del equipo
La cantidad total de conocimiento y esfuerzo mental que necesita un equipo para construir, operar y cambiar las partes de un sistema que le pertenecen. Cuando supera la capacidad del equipo, la calidad y la velocidad caen.

Complejidad accidental
Complejidad introducida por la forma en que se construye una solución (herramientas, estructura, soluciones provisionales) y no por el problema. En principio, puede eliminarse.

Complejidad ciclomática
La medida de Thomas McCabe de 1976 del número de caminos independientes a través de un fragmento de código, calculada a partir de su grafo de flujo de control; en el código sencillo equivale al número de decisiones más uno.

Complejidad esencial
Complejidad que pertenece al propio problema, como las reglas reales de los impuestos o los seguros, y que no puede eliminarse sin cambiar lo que hace el sistema.

Connascence
Dos elementos son connascentes si un cambio en uno exigiría un cambio en el otro para que el sistema siga siendo correcto. Es una forma más detallada de describir el acoplamiento, introducida por Meilir Page-Jones.

Connascence dinámica
Connascence que solo se manifiesta cuando el programa se ejecuta, como la dependencia del orden o los tiempos de ejecución, de valores relacionados o de que dos partes compartan la misma instancia de un objeto.

Connascence estática
Connascence visible en el código fuente sin ejecutarlo, como el acuerdo sobre un nombre, un tipo, el significado de un valor, el orden de los parámetros o un algoritmo.

Descomposición
Dividir un sistema en partes (módulos) y decidir de qué se encarga cada parte. El criterio usado para dividir importa más que el número de partes.

Diseño por contrato
El método de Bertrand Meyer para especificar cada operación con precondiciones que debe cumplir quien la llama, postcondiciones que la operación garantiza e invariantes que siempre se mantienen.

DRY
Don't repeat yourself (no te repitas): el principio de The Pragmatic Programmer según el cual cada pieza de conocimiento debe tener una única representación autorizada en un sistema. Se refiere al conocimiento, no a textos que parecen idénticos.

Fuerza (de la connascence)
Lo difícil que es descubrir y refactorizar una forma de connascence. La de nombre es la forma más débil; las formas dinámicas, como las de temporización, valor e identidad, están entre las más fuertes.

Grado (de la connascence)
Cuántos elementos comparten una connascence concreta. Un valor acordado por dos funciones es mucho más fácil de cambiar que uno acordado por cuarenta servicios.

Grado de abstracción (A)

La métrica de Martin: el número de clases abstractas e interfaces de un componente dividido por su número total de clases, de 0 (totalmente concreto) a 1 (totalmente abstracto).

Granularidad

El tamaño de los módulos o servicios en relación con todo el sistema. De grano grueso significa pocas partes grandes; de grano fino, muchas partes pequeñas. Cada opción tiene sus costos.

Hipótesis del espejo (mirroring hypothesis)

La versión académica de la ley de Conway: las dependencias técnicas de un producto tienden a coincidir con los vínculos de comunicación y organizativos entre las personas que lo construyen.

Inestabilidad (I)

La métrica de Martin I = Ce / (Ca + Ce), de 0 (máximamente estable: solo dependencias entrantes) a 1 (máximamente inestable: solo dependencias salientes).

Invariante

Una condición que siempre es verdadera para un módulo u objeto entre operaciones, como "el saldo de una cuenta nunca baja de su límite de descubierto".

Inversión de dependencias

Hacer que un módulo de alto nivel y uno de bajo nivel dependan ambos de una abstracción, normalmente propiedad del lado de alto nivel, de modo que la dependencia del código fuente apunte en sentido contrario al flujo de control.

KISS

Keep it simple (mantenlo simple): una regla práctica de diseño, procedente de la ingeniería de los años sesenta, según la cual debe evitarse la complejidad innecesaria porque hace que un sistema sea más difícil de entender, operar y reparar.

LCOM (falta de cohesión de los métodos)

Familia de métricas a nivel de clase que estiman la cohesión a partir de cómo comparten los métodos de una clase sus campos; un valor alto sugiere que la clase reúne varias responsabilidades sin relación entre sí.

Ley de Goodhart

La observación, formulada a menudo como "cuando una medida se convierte en objetivo, deja de ser una buena medida", de que las personas optimizan la cifra en lugar de aquello que debía indicar.

Ley de Hyrum

La observación, que lleva el nombre de Hyrum Wright, de que con suficientes usuarios de una API, alguien dependerá de cada comportamiento observable, prometa lo que prometa el contrato.

Localidad (de la connascence)

Lo cerca que están entre sí los elementos con connascence: dentro de una función, un módulo, un componente o a través de una red. Una connascence fuerte solo es tolerable cuando es local.

Maniobra inversa de Conway
Dar forma deliberadamente a los equipos y a sus vías de comunicación para que coincidan con la arquitectura que quieres, de modo que la ley de Conway juegue a tu favor y no en tu contra.

Modularidad
El grado en que un sistema está dividido en partes que pueden entenderse, cambiarse y probarse en gran medida por separado. Es una subcaracterística de la mantenibilidad en ISO/IEC 25010.

Módulo físico
Unidad que se compila, se versiona o se despliega, como un archivo JAR, un paquete de Python, una imagen de contenedor o un servicio. Un mismo diseño lógico puede empaquetarse en módulos físicos de muchas formas.

Módulo lógico
Unidad de responsabilidad en el diseño, como Pedidos o Pagos, con independencia de cómo se empaquete o despliegue su código.

Módulo profundo
Término de John Ousterhout para un módulo cuya interfaz sencilla oculta una gran cantidad de funcionalidad. Un módulo superficial ofrece poco comportamiento para la complejidad de su interfaz.

Ocultación de información
El principio de David Parnas (1972) según el cual cada módulo debe ocultar una decisión de diseño difícil o con probabilidad de cambiar, para que esa decisión pueda cambiar sin afectar a otros módulos.

Organización en capas (layering)
Organizar los módulos en capas donde cada capa solo puede usar las capas inferiores. La organización estricta permite usar solo la capa inmediatamente inferior; la relajada permite cualquier capa inferior.

Postcondición
Condición que una operación garantiza que será verdadera cuando termine, siempre que se haya cumplido su precondición.

Precondición
Condición que debe ser verdadera antes de llamar a una operación; cumplirla es obligación de quien llama.

Principio abierto-cerrado (OCP)
Un módulo debe estar abierto a la extensión pero cerrado a la modificación: el comportamiento nuevo debe añadirse agregando código, como una nueva implementación de una interfaz, no editando código que funciona. Formulado por primera vez por Bertrand Meyer en 1988.

Principio de abstracciones estables
Principio de Robert C. Martin según el cual un componente debe ser tan abstracto como estable, para que los componentes estables puedan seguir extendiéndose.

Principio de cierre común (CCP)
Las clases que cambian por los mismos motivos y en los mismos momentos deben estar en el mismo componente; es el principio de responsabilidad única aplicado a los componentes.

Principio de dependencias acíclicas
Principio de Robert C. Martin según el cual el grafo de dependencias entre componentes no debe contener ciclos.

Principio de dependencias estables
Principio de Robert C. Martin según el cual un componente solo debe depender de componentes más estables que él: depender en la dirección de la estabilidad.

Principio de equivalencia entre reutilización y lanzamiento (REP)
La unidad de reutilización es la unidad de lanzamiento: las clases agrupadas en un componente deben tener sentido para lanzarse, versionarse y reutilizarse juntas.

Principio de inversión de dependencias (DIP)
La política de alto nivel no debe depender de los detalles de bajo nivel; ambos deben depender de abstracciones, y la abstracción debe pertenecer al lado de la política.

Principio de responsabilidad única (SRP)
Un módulo debe responder ante un solo actor: un grupo de personas que solicita sus cambios. El código que cambia por motivos distintos, a petición de personas distintas, debe mantenerse separado.

Principio de reutilización común (CRP)
Las clases que se usan juntas deben estar juntas, y no se debe obligar a quienes usan un componente a depender de clases que no utilizan.

Principio de segregación de interfaces (ISP)
No se debe obligar a los clientes a depender de operaciones que no usan; es preferible tener varias interfaces pequeñas y específicas para cada cliente que una grande.

Principio de sustitución de Liskov (LSP)
Todo lo que afirma ser un subtipo debe poder usarse dondequiera que se espere su supertipo sin que quien llama note una diferencia de comportamiento. Lleva el nombre de Barbara Liskov.

Propiedad de los datos
La regla de que cada dato lo escribe exactamente un módulo o servicio, al que los demás deben preguntar en lugar de leer o modificar directamente su almacenamiento.

Secuencia principal

La recta A + I = 1 en un gráfico de abstracción frente a inestabilidad, donde se sitúan los componentes bien equilibrados. La distancia a ella es D = |A + I - 1|.

SOLID

Acrónimo mnemotécnico de cinco principios de diseño orientado a objetos reunidos por Robert C. Martin: responsabilidad única, abierto-cerrado, sustitución de Liskov, segregación de interfaces e inversión de dependencias.

Subtipado conductual

La regla que hay detrás de la sustitución de Liskov: un subtipo no puede exigir más a quienes lo llaman (precondiciones más fuertes), prometer menos (postcondiciones más débiles) ni romper las invariantes del supertipo.

Valor mágico

Número o cadena literal cuyo significado no se indica en el código, como un estado 3 que significa 'enviado'. Crea connascence de significado entre todo lo que lo usa.

Vías de comunicación

Los posibles vínculos uno a uno entre los miembros de un grupo. Un grupo de n personas tiene n(n-1)/2, por lo que el número crece aproximadamente con el cuadrado del tamaño del grupo.

YAGNI

You aren't gonna need it (no lo vas a necesitar): una práctica de Extreme Programming que consiste en no construir capacidades para necesidades solo supuestas, porque cuestan esfuerzo ahora y complejidad después.