BE SOFTWARE ARCHITECTURE WISE · CAPÍTULO 5

Estilos arquitectónicos I: monolitos y diseños modulares

La mayor parte del software, incluidos algunos de los productos más grandes y exitosos, se ejecuta como una única aplicación desplegable. Este capítulo explica cómo deben estructurarse por dentro esas aplicaciones. Presenta el vocabulario de los estilos arquitectónicos, los patrones, las tácticas y el cuanto de arquitectura; después trata la arquitectura en capas, el monolito modular, las tuberías y filtros, la arquitectura de microkernel y de plug-ins, las arquitecturas hexagonal, de cebolla y limpia, y el diseño estratégico guiado por el dominio, y termina con cuándo un monolito es la respuesta correcta. Está dirigido a desarrolladores y arquitectos que diseñan o mantienen aplicaciones de negocio y quieren elegir y hacer cumplir una estructura de forma deliberada.

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

DE UN VISTAZO

Aprenderás a

- Distinguir estilos arquitectónicos, patrones, tácticas y arquitecturas de referencia, y explicar por qué las restricciones de un estilo producen sus cualidades.
- Comparar el particionamiento técnico y el particionamiento por dominio, y predecir cómo afecta cada uno al costo de un cambio típico y a la organización de los equipos.
- Identificar los cuantos de arquitectura de un diseño y explicar por qué las bases de datos compartidas unen los servicios en un solo cuanto.
- Describir los estilos en capas, monolito modular, tuberías y filtros y microkernel, valorar sus características y reconocer sus antipatrones.
- Aplicar la arquitectura hexagonal, de cebolla o limpia para mantener la lógica de negocio independiente de los frameworks y de la infraestructura.
- Usar el diseño estratégico guiado por el dominio para encontrar contextos delimitados y describir sus relaciones en un mapa de contextos.
- Hacer cumplir reglas estructurales en el código con pruebas de arquitectura y herramientas de módulos como import-linter, ArchUnit y Spring Modulith.
- Decidir cuándo un monolito es la opción correcta, cómo escalarlo y qué señales indican que ha llegado el momento de extraer servicios.

Antes de empezar

Capítulo 3, en especial la modularidad, el acoplamiento y la cohesión, y las dependencias y la organización en capas; Capítulo 4, en especial el modelo C4. Los atributos de calidad del Capítulo 2 se usan a lo largo de todo el capítulo.

Tiempo

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

Términos clave

estilo arquitectónico · patrón arquitectónico · táctica · particionamiento técnico · particionamiento por dominio · cuanto de arquitectura · capas cerradas y abiertas · sumidero de arquitectura (architecture sinkhole) · monolito modular · tuberías y filtros · back pressure (contrapresión) · microkernel · puertos y adaptadores · contexto delimitado (bounded context)

Por qué importan los estilos

Los Capítulos 3 y 4 te dieron los principios de una buena estructura (modularidad, acoplamiento y cohesión, dirección de las dependencias) y las herramientas para describirla. Este capítulo y el siguiente convierten esos principios en las formas recurrentes que adoptan los sistemas reales. Un estilo arquitectónico es un conjunto con nombre de restricciones sobre los tipos de partes que tiene un sistema y sobre cómo pueden conectarse. Las restricciones son lo importante: prohíbe que las partes de un pipeline compartan estado, o que un módulo de negocio toque las tablas de otro, y obtendrás beneficios predecibles, a un precio predecible.

Conocer los estilos principales permite a un arquitecto partir de una forma que se sabe que funciona para un conjunto dado de cualidades, explicarla en una frase y reconocer pronto cuándo un sistema se está alejando de ella.

Este capítulo trata los estilos en los que toda la aplicación es una sola unidad de despliegue: una compilación, un lanzamiento, normalmente una base de datos, con partes que se llaman entre sí en memoria. Una arquitectura monolítica así evita los fallos de red, las caídas parciales y los problemas de consistencia de los sistemas distribuidos, de los que se ocupa el Capítulo 6. Eso no hace que los monolitos sean primitivos. Que trabajar en un monolito sea un placer o un suplicio depende casi por completo de cómo esté dividido por dentro, y de eso trata este capítulo.

Cómo encaja el capítulo

La Sección 5.1 establece el vocabulario que se usa para cada estilo de los Capítulos 5 y 6: estilos, patrones, tácticas y arquitecturas de referencia; la diferencia entre el particionamiento técnico (por capa) y el particionamiento por dominio (por capacidad de negocio); el despliegue monolítico frente al distribuido; y el cuanto de arquitectura, la unidad desplegable de forma independiente a la que se aplica un mismo conjunto de cualidades. Termina con una tabla resumen de todos los estilos de los dos capítulos.

Las Secciones 5.2 a 5.5 describen después cuatro estilos con un formato uniforme: definición y origen, un diagrama, cómo funciona, un ejemplo, una tabla de características, cuándo usarlo y cuándo no, antipatrones y cómo hacerlo cumplir en el código. La arquitectura en capas (5.2) es la conocida pila de presentación, negocio y persistencia, con capas cerradas y abiertas y el antipatrón del sumidero. El monolito modular (5.3) divide una sola aplicación desplegable por dominio de negocio, con API de módulo explícitas y propiedad de los datos, ilustrado con la experiencia de Shopify y un mapa de módulos de una plataforma de reparto de comida. Tuberías y filtros (5.4) organiza el procesamiento como una cadena de pasos independientes, desde las tuberías de Unix hasta los pipelines de datos. El microkernel (5.5) construye un núcleo mínimo que los plug-ins amplían, como en los IDE, los navegadores y los motores de reglas.

Las Secciones 5.6 y 5.7 tratan de la estructura dentro de los módulos y entre ellos, más que del despliegue. Las arquitecturas hexagonal, de cebolla y limpia (5.6) ponen la lógica de negocio en el centro y hacen que las bases de datos, las interfaces de usuario y los servicios externos dependan de ella, nunca al revés, lo que permite probar el núcleo y reemplazar su infraestructura. El diseño estratégico guiado por el dominio (5.7) aporta el método para encontrar buenos límites: subdominios, contextos delimitados y las relaciones entre ellos en un

mapa de contextos. Por último, la Sección 5.8 se pregunta cuándo un monolito es la respuesta correcta, hasta dónde puede escalar y qué señales indican que ha llegado el momento de dividirlo, y ofrece una guía de decisión y una tabla comparativa de los estilos del capítulo.

Un ejemplo recurrente

Como en capítulos anteriores, a lo largo de todo el capítulo aparece una plataforma de reparto de comida: los clientes hacen pedidos a restaurantes, los repartidores los entregan y los pagos pasan por un proveedor externo. En la Sección 5.1 se clasifica según cuatro diseños candidatos; en la 5.2 es una aplicación web clásica en capas; en la 5.3 se convierte en ocho módulos con una regla de dependencia comprobada en la compilación; en la 5.4 sus eventos de pedidos fluyen por un pipeline hasta un almacén de datos, donde encontrar el cuello de botella es un cálculo resuelto paso a paso. Las secciones posteriores vuelven a la misma plataforma, con más detalle en el mapa de contextos de la Sección 5.7, para que los estilos se puedan comparar sobre un único sistema conocido y no sobre ejemplos de juguete sin relación entre sí.

La idea que debes llevarte

Si recuerdas una sola cosa de este capítulo, que sea esta: **los límites dentro de un sistema importan más que los límites entre unidades de despliegue**. Un monolito dividido por dominio, con API de módulo explícitas, un único propietario para cada tabla y reglas de dependencia comprobadas automáticamente, puede ser modificado de forma segura por varios equipos durante muchos años, y cualquier módulo puede convertirse más adelante en un servicio si de verdad lo necesita. Un conjunto de servicios que comparten una base de datos y llaman a las partes internas de los demás sigue siendo un único sistema enmarañado, ahora con fallos de red añadidos. Los estilos son herramientas para crear y conservar buenos límites; ninguno de ellos es un fin en sí mismo.

De ahí se derivan dos hábitos. Primero, elige el estilo más sencillo cuyas características satisfagan los requisitos arquitectónicamente significativos del sistema, y registra la elección y sus compromisos en un registro de decisiones. Segundo, haz cumplir las reglas del estilo con pruebas de arquitectura en la compilación, porque todo estilo se erosiona cuando sus restricciones solo viven en la cabeza de las personas.

SECCIÓN 5.1 DE 5.8 · INTERMEDIO · 24 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Estilos, patrones y tácticas arquitectónicos

ANTES DE LEER ESTO

Esta sección se basa en:

- 2.1 Requisitos funcionales y de calidad
- 3.1 Modularidad
- 3.7 Ley de Conway

Términos anteriores que utiliza: Atributo de calidad, Arquitectónicamente significativo, Conector, Cohesión, Ley de Conway

LA IDEA EN BREVE

Piensa en los tipos de vivienda: una casa adosada, una casa de una sola planta, una torre de viviendas. Cada nombre le dice mucho a un constructor antes de que exista ningún plano: cuántas plantas tiene, cómo se conectan las habitaciones, cuánto cuesta, dónde funciona bien. Una torre de viviendas encaja en una ciudad abarrotada y es un despilfarro en un pueblo. El software tiene sus propias formas con nombre, llamadas estilos arquitectónicos: una aplicación en capas, un conjunto de módulos en un solo programa, una cadena de pasos de procesamiento, un grupo de servicios separados. Conocer los estilos, y en qué es bueno y en qué es malo cada uno, permite a un equipo elegir rápidamente una forma sensata y hablar de ella en pocas palabras.

LO QUE NECESITAS SABER

- Un estilo arquitectónico es un conjunto con nombre de tipos de elementos, tipos de conectores y restricciones sobre cómo se combinan (Garlan y Shaw); las restricciones son lo que da a un estilo sus cualidades predecibles.
- Un patrón arquitectónico, en la tradición de *Pattern-Oriented Software Architecture* (Buschmann et al., 1996), es una solución probada a un problema estructural recurrente en un contexto dado; una táctica es una única decisión de diseño orientada a un atributo de calidad; una arquitectura de referencia es una plantilla para un dominio.
- El particionamiento técnico divide un sistema por función técnica (presentación, lógica, datos); el particionamiento por dominio lo divide por capacidad de negocio (pedidos, pagos, entregas). La elección decide hasta dónde se extiende un cambio típico de funcionalidad y cómo se pueden organizar los equipos.

- Un estilo monolítico se despliega como una sola unidad; un estilo distribuido se despliega como varias unidades que cooperan a través de una red y hereda todos los costos de la red.
- Un cuanto de arquitectura es una parte desplegable de forma independiente con alta cohesión funcional, datos incluidos; a cada cuanto se aplica un único conjunto de características de arquitectura, así que las partes con necesidades muy distintas deben ser cuantos separados.
- Cada estilo es un compromiso con fortalezas y debilidades características; los sistemas reales combinan estilos, y las valoraciones de este curso son juicios relativos para implementaciones típicas, no leyes.

Por qué los arquitectos ponen nombre a las formas

La mayoría de los sistemas no se diseñan desde cero. Los arquitectos con experiencia reconocen que un problema nuevo se parece a otro que ya han visto y recurren a una forma que se sabe que funciona: unas pocas capas para una aplicación de negocio sencilla, una cadena de pasos de procesamiento para un flujo de datos, un núcleo con plug-ins para un producto que los clientes amplían. Poner nombre a esas formas da a un equipo un vocabulario común. Decir “es un monolito modular con comunicación basada en eventos entre módulos” transmite en una decena de palabras lo que de otro modo llevaría una página de diagramas, y trae consigo un conjunto de conocimientos sobre lo que suele salir bien y mal.

David Garlan y Mary Shaw le dieron una base sólida en *An Introduction to Software Architecture* (informe técnico de Carnegie Mellon, 1994) y en su libro *Software Architecture: Perspectives on an Emerging Discipline* (1996). Describieron un estilo arquitectónico como un vocabulario de tipos de componentes y de conectores junto con un conjunto de restricciones sobre cómo pueden combinarse. Tuberías y filtros, por ejemplo, tiene un tipo de componente (un filtro que transforma datos) y un tipo de conector (una tubería que los transporta en un solo sentido), y obliga a que los filtros no compartan estado ni sepan quiénes son sus vecinos. Roy Fielding, cuya tesis doctoral (2000) definió REST como un estilo, lo expresó de forma aún más tajante: un estilo es un conjunto coordinado de restricciones, y son las restricciones las que producen las propiedades. Prohíbe que los filtros compartan estado y obtendrás filtros que se pueden probar, reutilizar y reorganizar de forma independiente; no es suerte, es una consecuencia de la regla.

Esta es la forma más útil de leer cada estilo de este capítulo y del siguiente. Un estilo no es un dibujo que copiar. Es un conjunto de reglas que te comprometes a seguir, elegidas porque seguirlas te da unas características de arquitectura concretas, los atributos de calidad del Capítulo 2, a cambio de un precio. Un equipo que adopta el vocabulario de un estilo pero no sus reglas paga el precio sin obtener los beneficios.

Estilos, patrones, tácticas y arquitecturas de referencia

Se usan cuatro términos relacionados, a menudo sin mucha precisión, para referirse al conocimiento de diseño reutilizable. Se diferencian sobre todo en su alcance y en cuánto prescriben.

Un patrón arquitectónico describe una solución probada a un problema estructural recurrente en un contexto dado. La idea de los patrones procede del arquitecto de edificios Christopher Alexander y llegó al software a través de los patrones de diseño de Gamma, Helm, Johnson y Vlissides (1994), que trabajan a la escala de unas pocas clases. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, de Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad y Michael Stal (1996), normalmente llamado POSA, catalogó patrones a la escala de sistemas completos: Capas (Layers), Tuberías y filtros (Pipes and Filters) y Pizarra (Blackboard), agrupados como el paso del barro a la estructura; Broker para sistemas distribuidos; Modelo-Vista-Controlador y Presentación-Abstracción-Control para sistemas interactivos; y Microkernel y Reflexión (Reflection) para sistemas adaptables. Cada patrón de POSA se describe con su contexto, problema, fuerzas y solución, junto con sus consecuencias, lo que hace explícitos los compromisos.

Una táctica, en el sentido que le dan Len Bass, Paul Clements y Rick Kazman en *Software Architecture in Practice*, es algo más pequeño: una decisión de diseño que influye en la respuesta de un atributo de calidad, como añadir un heartbeat para detectar un servidor caído o una caché para reducir la latencia. La Sección 2.6 usó tácticas de modificabilidad como el encapsulamiento y el uso de un intermediario. Los patrones agrupan varias tácticas: el patrón Capas combina el encapsulamiento, la restricción de dependencias y el uso de un intermediario.

Una arquitectura de referencia es una plantilla para todo un dominio o una plataforma: nombra los bloques de construcción típicos y cómo se conectan, para que los proyectos la adapten en lugar de partir de una página en blanco. AUTOSAR, la arquitectura de software en capas que se usa para las unidades de control electrónico de los automóviles, es un ejemplo conocido; los centros de arquitectura que publican los proveedores de nube ofrecen muchos otros más pequeños. Una arquitectura de referencia suele construirse a partir de varios estilos y patrones a la vez.

Los profesionales no se ponen de acuerdo sobre dónde está la frontera entre estilo y patrón. POSA considera Capas (Layers) un patrón; Garlan y Shaw llaman estilo a los sistemas en capas; Mark Richards y Neal Ford, en *Fundamentals of Software Architecture*, reservan “estilo” para la topología general de un sistema y “patrón” para estructuras reutilizables de nivel más bajo que pueden aparecer dentro de muchos estilos. Este curso sigue el uso habitual: **estilo** para la

forma general de un sistema o de una gran parte de él, **patrón** para una solución reutilizable que puede situarse en cualquier nivel. No gastes energía en la frontera; gástala en las restricciones y sus consecuencias.

Cuatro tipos de conocimiento de diseño reutilizable

Término	Alcance	Ejemplo	Pregunta que responde
Estilo arquitectónico	Todo el sistema o una gran parte	En capas, tuberías y filtros, microservicios	¿Qué tipos de partes y conexiones se permiten, y con qué reglas?
Patrón arquitectónico	Cualquier nivel estructural	Broker, Modelo-Vista-Controlador, Microkernel	¿Cuál es una solución probada a este problema estructural recurrente, y cuánto cuesta?
Táctica	Una decisión de diseño	Heartbeat, caché, encapsulamiento, reintento con backoff	¿Qué decisión concreta mueve este atributo de calidad en la dirección que necesitamos?
Arquitectura de referencia	Un dominio o una plataforma	AUTOSAR para la electrónica del automóvil; las arquitecturas publicadas por los proveedores de nube	¿Qué suelen contener los sistemas de este tipo, y cómo suelen organizarse?

Dos formas de dividir un sistema: particionamiento técnico o particionamiento por dominio

La primera decisión estructural en cualquier estilo es cómo dividir el sistema en el nivel superior. Hay dos respuestas básicas. El particionamiento técnico agrupa el código por función técnica: todo lo relativo a la interfaz de usuario en un lugar, todas las reglas de negocio en otro, todo el acceso a la base de datos en un tercero. El particionamiento por dominio agrupa el código por capacidad de negocio: todo lo relativo a los pedidos en un lugar, los pagos en otro, las entregas en un tercero, cada uno con la interfaz de usuario, la lógica y el acceso a datos que necesite.

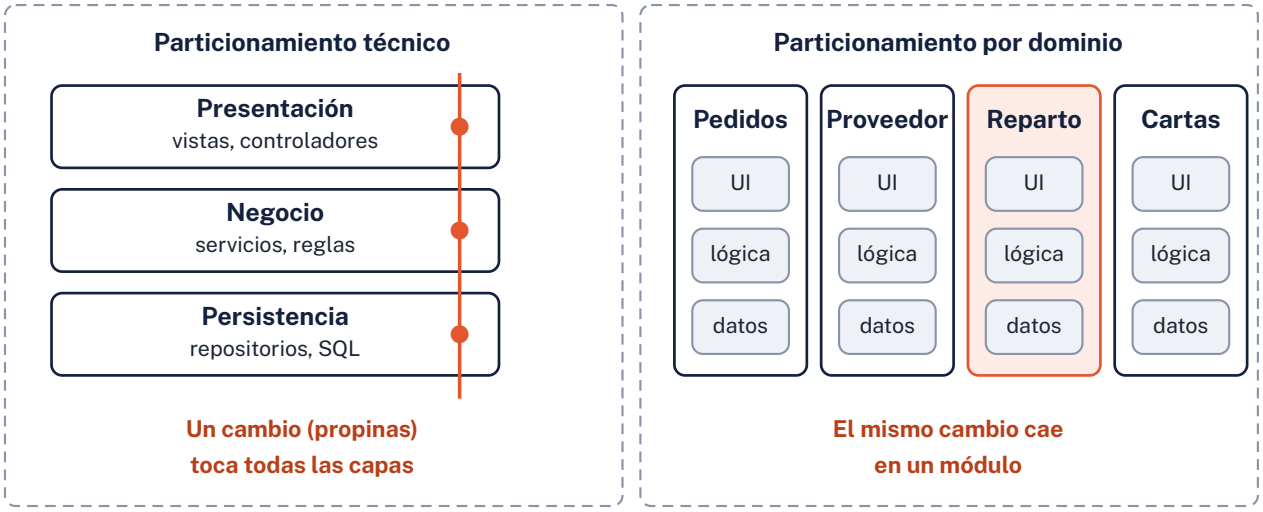
La diferencia se nota en cuanto cambia una funcionalidad. Supón que una empresa de reparto de comida permite a los clientes añadir una propina para el repartidor. En un sistema con particionamiento técnico, ese cambio necesita un campo nuevo en una pantalla (presentación), una regla nueva para los límites y el abono de las propinas (lógica de negocio) y una columna y una consulta nuevas (persistencia): tres partes, a menudo a cargo de tres personas o equipos distintos. En un sistema con particionamiento por dominio, el mismo cambio cae casi por completo en el módulo de reparto. La mayoría de las solicitudes de cambio del negocio se

formulan en términos del dominio, así que el particionamiento por dominio tiende a mantener un cambio en un solo lugar. El particionamiento técnico mantiene en un solo lugar un cambio *técnico*, como sustituir la biblioteca de acceso a la base de datos, pero esos cambios son menos frecuentes.

La elección también da forma a la organización. Según la ley de Conway, un sistema con particionamiento técnico tiende a construirlo un equipo de frontend, un equipo de backend y un equipo de base de datos, y cada funcionalidad necesita un traspaso entre ellos. Un sistema con particionamiento por dominio puede construirse con equipos que sean dueños, cada uno, de una capacidad de negocio de principio a fin. Martin Fowler hizo la misma recomendación en 2015: las capas de presentación, dominio y datos son una buena manera de organizar una pieza pequeña de software, pero el nivel superior de un sistema más grande debería dividirse en módulos orientados al dominio, cada uno organizado en capas por dentro si eso ayuda.

Ninguna de las dos es incorrecta. El particionamiento técnico es sencillo, conocido y está respaldado por la estructura de carpetas predeterminada de casi cualquier framework web, y encaja con sistemas pequeños en los que un solo equipo lo hace todo. El particionamiento por dominio exige más reflexión al principio, porque alguien tiene que encontrar buenos límites de dominio (el tema de la Sección 5.7), pero escala mejor con el tamaño del sistema y el número de equipos.

La misma funcionalidad con dos particionamientos



Con el particionamiento técnico, una funcionalidad típica cruza todas las capas; con el particionamiento por dominio, cae en un módulo, que por dentro puede seguir organizado en capas.

Estilos monolíticos y distribuidos

La segunda distinción de nivel superior tiene que ver con el despliegue. Una unidad de despliegue es lo que se compila, se lanza y se arranca como una sola pieza: un único ejecutable, una imagen de contenedor, un archivo web. En una arquitectura monolítica, toda la aplicación es una sola unidad de despliegue, normalmente con una sola base de datos. Puede ejecutarse como muchas copias idénticas detrás de un balanceador de carga, pero cada copia contiene todo el código, y lanzar cualquier cambio significa lanzarlo todo. En una arquitectura distribuida, el sistema está formado por varias unidades de despliegue que cooperan enviándose mensajes a través de una red, y cada una puede lanzarse y escalarse por separado.

Dos aclaraciones evitan mucha confusión. Primera: "monolito" describe el despliegue, no la calidad. Un monolito puede ser maravillosamente modular o una gran bola de barro (big ball of mud, el nombre que Brian Foote y Joseph Yoder dieron a un sistema sin una estructura discernible), y lo mismo vale para un sistema distribuido: muchos equipos han aprendido que dividir una bola de barro en servicios da una bola de barro distribuida, que es peor. Segunda: ser distribuido no es una mejora gratuita. Cualquier llamada que cruza una red puede ser lenta, fallar a medias o llegar dos veces, los datos ya no pueden modificarse en una única transacción local, y el sistema necesita monitorización, automatización del despliegue y capacidades de guardia de las que un monolito puede prescindir. El Capítulo 6 trata esos costos en detalle. Este capítulo trata los estilos que los evitan: todo se ejecuta en una sola unidad de despliegue, y la arquitectura reside en cómo se organiza el código dentro de ella.

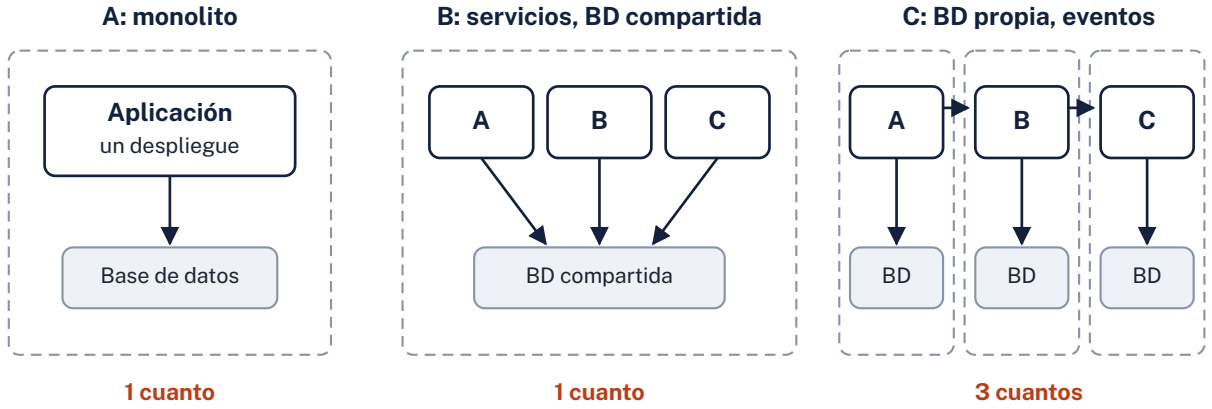
El cuanto de arquitectura

Monolítico frente a distribuido es una primera división útil, pero demasiado gruesa para las decisiones reales. Neal Ford, Rebecca Parsons y Patrick Kua introdujeron una idea más precisa en *Building Evolutionary Architectures* (2017), que Ford, Richards, Sadalage y Dehghani refinaron en *Software Architecture: The Hard Parts* (2021): el cuanto de arquitectura (architecture quantum). Un cuanto es una parte del sistema que puede desplegarse de forma independiente, tiene alta cohesión funcional (hace un trabajo coherente para el negocio) e incluye todo lo que necesita para funcionar, en particular su base de datos. La definición posterior añade que las partes de un cuanto están acopladas estáticamente (comparten código, esquemas o infraestructura que deben desplegarse juntos) y que las llamadas síncronas unen los cuantos operativamente mientras dura la llamada.

¿Por qué importa esto? Porque las características de arquitectura se aplican por cuanto. Un cuanto se escala, se despliega, se protege y se recupera como un todo, así que tiene un único nivel de escalabilidad, un único nivel de disponibilidad y una única cadencia de lanzamientos. Si dos partes de un sistema necesitan características muy distintas, por ejemplo un flujo de posiciones de repartidores que debe absorber miles de actualizaciones por segundo y un

proceso mensual de facturación a restaurantes, solo pueden obtenerlas estando en cuantos distintos. A la inversa, dividir el código en varios servicios no crea varios cuantos si siguen compartiendo una base de datos: ninguno puede cambiar el esquema ni restaurarse por sí solo, así que se mantienen o caen juntos. Un monolito es, por definición, un solo cuanto.

Contar cuantos



Los límites discontinuos marcan cada cuanto: lo que se despliega y ejecuta por sí solo, datos incluidos

Una base de datos compartida une los servicios en un solo cuanto; los cuantos separados necesitan despliegues separados y datos separados.

EJEMPLO PRÁCTICO

Ejemplo práctico: clasificar cuatro diseños

Una empresa de reparto de comida está revisando cuatro diseños candidatos para su plataforma. Para cada uno, el arquitecto se hace tres preguntas: ¿cómo se particiona el nivel superior?, ¿cuántas unidades de despliegue hay? y ¿cuántos cuantos?

Diseño 1: una aplicación Django con las carpetas `views`, `services` y `models`, una base de datos PostgreSQL	Particionamiento técnico; una unidad de despliegue; un cuanto. Sencillo, pero cada funcionalidad cruza las tres carpetas.
Diseño 2: una aplicación con los paquetes `orders`, `payments`, `delivery` y `menus`, cada uno con sus propias tablas, un servidor de base de datos	Particionamiento por dominio; una unidad de despliegue; un cuanto. Los cambios se quedan locales, y los módulos podrían separarse más adelante.
Diseño 3: cuatro servicios (pedidos, pagos, entrega, cartas) que leen y escriben todos en las mismas tablas de la base de datos	Particionamiento por dominio solo de nombre; cuatro unidades de despliegue pero un solo cuanto, porque un cambio de esquema o una caída de la base de datos afecta a los cuatro. Todos los costos de la distribución y pocos de sus beneficios.
Diseño 4: los mismos cuatro servicios, cada uno con su propia base de datos, que intercambian eventos a través de un intermediario de mensajes (message broker)	Particionamiento por dominio; cuatro unidades de despliegue; cuatro cuantos. Cada uno puede escalarse y lanzarse de forma independiente, a costa de los fallos de red y de la consistencia eventual entre ellos.

Los diseños 1 y 2 cuestan lo mismo de operar, pero el diseño 2 abarata los cambios típicos y mantiene abiertas las opciones futuras.

El diseño 3 es el caso de advertencia: el número de unidades de despliegue dice poco; el número de cuantos, que depende de los datos y del acoplamiento, dice qué puede evolucionar realmente de forma independiente.

SI ESTO ES NUEVO PARA TI

Paso a paso: un cuanto, un conjunto de cualidades

El punto en el que más gente tropieza aquí es que las características de arquitectura pertenecen a todo un cuanto de arquitectura, no a cada funcionalidad que contiene. Un cuanto se despliega, se escala y se recupera como un todo, así que todo lo que contiene comparte un único nivel de escalabilidad, un único nivel de disponibilidad y una única cadencia de lanzamientos. Sigue las dos funcionalidades contrapuestas de la propia sección a través de un monolito.

- La plataforma de reparto de comida se ejecuta como una sola unidad de despliegue con una sola base de datos. Dos de sus funcionalidades necesitan cosas muy distintas: un flujo de posiciones de repartidores que debe absorber miles de actualizaciones por segundo y un proceso mensual de facturación a restaurantes.
- A la hora de la cena, el flujo de posiciones está sobrecargado. La única forma de añadir capacidad es ejecutar más copias de toda la aplicación, y cada copia lleva también el código de facturación, que no necesita ninguna capacidad adicional.
- Si el proceso de facturación tiene un bug que agota la memoria del proceso, puede detener también el flujo de posiciones, porque ambos se ejecutan en el mismo proceso.
- Cuando el código de facturación necesita una pequeña corrección, no puede lanzarse por sí solo: se compila, se prueba y se vuelve a desplegar toda la aplicación, flujo de posiciones incluido.
- Para que el flujo tenga su propio escalado, sus propios fallos y sus propios lanzamientos, debe convertirse en un cuanto separado, desplegado por sí solo y con sus propios datos. Mover su código a un servicio separado no basta si sigue compartiendo la base de datos, porque entonces ninguna de las dos partes puede cambiar el esquema ni restaurarse por sí sola.
- **La lección:** no preguntes cuántos servicios tiene un diseño; pregunta qué partes necesitan características muy distintas. Esas partes necesitan cuantos separados, y todo lo demás puede quedarse junto en uno solo.

Cómo leer las secciones de estilos que siguen

Las Secciones 5.2 a 5.5 y la mayor parte del Capítulo 6 describen un estilo cada una, siempre en el mismo orden, para que los estilos se puedan comparar: una definición y de dónde viene el estilo; un diagrama; cómo funciona; un sistema de ejemplo concreto; una tabla que valora sus características principales; cuándo usarlo y cuándo no; los antipatrones habituales; y cómo hacerlo cumplir en el código, porque un estilo que no se comprueba se erosiona, como describió la Sección 1.7.

Las valoraciones de las características (Baja, Media, Alta, cada una con un motivo) siguen el enfoque de las comparaciones de estilos de Richards y Ford, pero son juicios propios de este curso. Léelas como relativas y típicas: comparan los estilos tal como se construyen normalmente, por un equipo competente, para un sistema de negocio típico. Un equipo disciplinado puede hacer un monolito en capas que se despliegue muchas veces al día, y uno descuidado puede construir microservicios que no se pueden desplegar en absoluto. Las valoraciones te dicen dónde ayuda el estilo y dónde tendrás que trabajar en su contra.

Dos secciones de este capítulo no tratan de estilos de despliegue. Las arquitecturas hexagonal, de cebolla y limpia (Sección 5.6) describen cómo organizar el interior de una aplicación o de un servicio para que la lógica de negocio no dependa de la tecnología. El diseño guiado por el dominio (Sección 5.7) es una forma de encontrar los límites de dominio en los que se apoyan los monolitos modulares y los microservicios. Ambos se aplican igual dentro de un monolito que de un sistema distribuido.

Los estilos de los Capítulos 5 y 6 de un vistazo

Estilo	Particionamiento principal	Despliegue	Encaje típico
En capas (5.2)	Técnico	Monolítico	Aplicaciones de negocio pequeñas y medianas construidas por un solo equipo
Monolito modular (5.3)	Por dominio	Monolítico	Productos en crecimiento con varios equipos que quieren un único despliegue
Tuberías y filtros (5.4)	Técnico (pasos de procesamiento)	Monolítico o distribuido	Procesamiento de datos: ETL, multimedia, logs, compiladores
Microkernel (5.5)	Núcleo técnico, plug-ins a menudo por dominio	Normalmente monolítico	Productos que amplían los clientes o terceros; motores de reglas
Hexagonal, de cebolla, limpia (5.6)	Estructura interna: el dominio en el centro	Cualquiera de los dos	Cualquier aplicación cuya lógica deba sobrevivir a sus frameworks
Basada en servicios (6.2)	Por dominio	Distribuido, pocos servicios de grano grueso	Un paso pragmático más allá del monolito, que a menudo comparte una base de datos
Arquitectura orientada a servicios (6.2)	Técnico (orquestación, servicios empresariales)	Distribuido	Integración empresarial, hoy en gran parte histórica
Microservicios (6.3)	Por dominio	Distribuido, muchos servicios de grano fino	Muchos equipos que deben lanzar de forma independiente a gran escala
Dirigida por eventos (6.4)	Técnico (procesamiento de eventos)	Distribuido	Reacciones muy rápidas y desacopladas a los eventos de negocio
Serverless y basada en espacios (6.8)	Variable	Distribuido	Carga con picos o extrema; cargas de trabajo de pago por uso

Los estilos se combinan, y no son objetivos

Los sistemas reales son híbridos. Un monolito modular puede estar organizado en capas dentro de cada módulo; uno de sus módulos puede ejecutar un proceso nocturno de tuberías y filtros; un microservicio puede construirse con estilo hexagonal; un sistema dirigido por eventos puede usar un microkernel para que los clientes añadan sus propias reglas. Nombrar el estilo de cada parte sigue siendo útil, porque así cada parte hereda un conjunto conocido de reglas y compromisos.

El error habitual es tratar un estilo como un fin en sí mismo: "nos estamos pasando a microservicios" como estrategia, en lugar de como medio para lograr una cualidad declarada, como el despliegue independiente para doce equipos. Parte de los requisitos arquitectónicamente significativos del Capítulo 2, elige el estilo más sencillo cuyas características los satisfagan, deja constancia de la elección en un registro de decisiones y haz cumplir sus reglas. La Sección 5.8 ofrece una guía de decisión para los estilos de este capítulo.

ATENCIÓN

Malentendidos habituales

- **Monolito significa sistema heredado.** Significa una sola unidad de despliegue. Muchos productos de éxito se ejecutan como monolitos bien estructurados.
- **Más servicios significa más modular.** La modularidad tiene que ver con los límites y las dependencias; los servicios que comparten una base de datos son un solo cuanto, por muchos que haya.
- **Un estilo se elige una sola vez.** Los estilos evolucionan: una aplicación en capas puede convertirse en un monolito modular y después perder un módulo, que pasa a ser un servicio separado cuando sus necesidades divergen.
- **Usar las carpetas del framework es elegir un estilo.** Una estructura de carpetas predeterminada es un particionamiento técnico que los autores del framework eligieron para aplicaciones pequeñas, no una decisión sobre tu sistema.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **Según Garlan y Shaw, ¿qué dos cosas definen un estilo arquitectónico?**

Un vocabulario de tipos de componentes y de conectores, y un conjunto de restricciones sobre cómo pueden combinarse. Las restricciones son las que producen las cualidades características del estilo.

▼ **¿Qué diferencia hay entre una táctica y un patrón?**

Una táctica es una decisión de diseño orientada a la respuesta de un atributo de calidad, como un heartbeat o una caché. Un patrón es una solución documentada más amplia que normalmente combina varias tácticas.

▼ **¿Por qué cuatro servicios que comparten una base de datos pueden contar como un único cuanto de arquitectura?**

Porque ninguno de ellos puede cambiar el esquema compartido, restaurarse o escalarse en el nivel de los datos sin afectar a los demás; en la práctica se despliegan y fallan juntos, así que comparten un único conjunto de características operativas.

▼ **¿Qué tipo de cambio mantiene en un solo lugar el particionamiento técnico?**

Un cambio técnico, como sustituir la biblioteca de acceso a la base de datos o el framework de interfaz de usuario. Los cambios de negocio típicos, en cambio, atraviesan todas las capas técnicas.

▼ **Inténtalo: Una plataforma de venta de entradas tiene tres servicios desplegados por separado, Booking, Payments e Reports, que usan una base de datos compartida. ¿Cómo está particionada, cuántas unidades de despliegue y cuántos cuantos tiene? ¿Por qué podría importar la carga pesada de Reports?**

Está particionado por dominio, con tres unidades de despliegue, pero un solo cuanto, porque la base de datos compartida los une: un cuanto incluye sus datos. Así que el sistema no es tan independiente como parece. Las consultas pesadas de Reports pueden ralentizar Booking el día de la salida a la venta, y un único conjunto de características de calidad se aplica a los tres. Para dar a Reports necesidades distintas, conviértelo en un cuanto separado, por ejemplo con su propio almacén de datos de informes.

FUENTES

- Garlan and Shaw: An Introduction to Software Architecture (SEI, 1994): https://www.sei.cmu.edu/library/an-introduction-to-software-architecture/
- Roy Fielding: Architectural Styles and the Design of Network-based Software Architectures, chapter 1: https://ics.uci.edu/~fielding/pubs/dissertation/software_arch.htm
- Martin Fowler: Presentation Domain Data Layering: https://martinfowler.com/bliki/PresentationDomainDataLayering.html
- Microsoft Azure Architecture Center: Architecture styles: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/

SECCIÓN 5.2 DE 5.8 · INTERMEDIO · 24 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Arquitectura en capas

ANTES DE LEER ESTO

Esta sección se basa en:

- 3.4 Dependencias
- 5.1 Estilos y patrones

Términos anteriores que utiliza: Particionamiento técnico, Unidad de despliegue, Cuanto de arquitectura (architecture quantum), Ley de Conway

LA IDEA EN BREVE

Piensa en un restaurante. Los clientes solo hablan con los camareros. Los camareros pasan los pedidos a la cocina y nunca entran ellos mismos en el almacén. Los cocineros piden los ingredientes al personal del almacén, y solo el personal del almacén trata con los proveedores. Cada grupo tiene una tarea y solo habla con el grupo de al lado, así que puedes volver a formar a los camareros sin tocar la cocina. Una aplicación en capas funciona igual: las pantallas hablan con las reglas de negocio, las reglas de negocio hablan con el código de acceso a datos, y solo ese código habla con la base de datos.

LO QUE NECESITAS SABER

- Una arquitectura en capas organiza el código en capas horizontales por función técnica, normalmente presentación, negocio, persistencia y base de datos, y permite que cada capa dependa solo de las capas que tiene debajo.
- Las capas cerradas no deben saltarse, lo que proporciona capas de aislamiento: un cambio dentro de una capa no se propaga a las demás mientras su interfaz se mantenga. Una capa abierta, como una capa de servicios compartidos, puede saltarse.
- La organización en capas es particionamiento técnico, así que una funcionalidad de negocio típica toca todas las capas; esto ralentiza el cambio y favorece estructuras de equipo divididas por capa.
- El antipatrón del sumidero de arquitectura consiste en solicitudes que pasan por capas que solo las reenvían; como regla general, que una minoría de solicitudes sea así es normal, y una gran mayoría sugiere que el estilo no es el adecuado.
- El estilo puntúa alto en sencillez y bajo costo, y bajo en desplegabilidad, escalabilidad, elasticidad y tolerancia a fallos, porque todo el sistema es una sola unidad de despliegue.

- Las reglas de las capas se erosionan si no se hacen cumplir automáticamente, por ejemplo con import-linter en Python o ArchUnit en Java, como pruebas que hacen fallar la compilación.

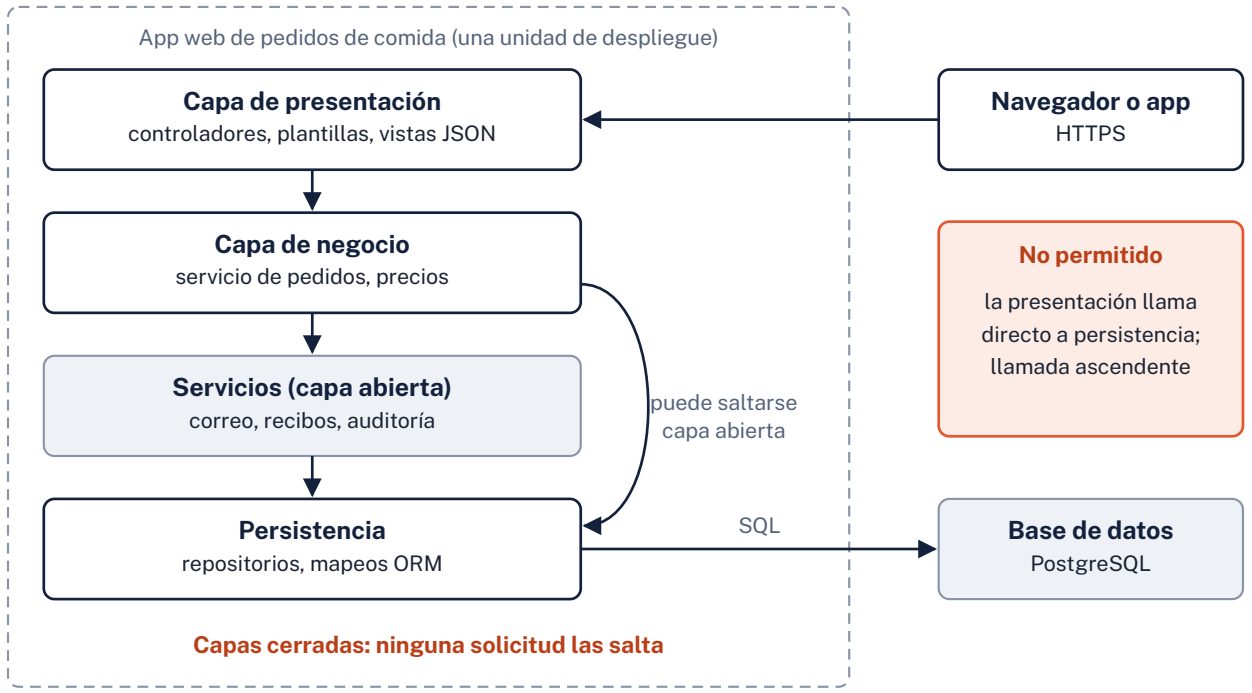
Definición y origen

Una arquitectura en capas organiza una aplicación en capas horizontales, cada una de las cuales tiene una responsabilidad técnica, ofrece servicios a la capa superior y usa los servicios de la capa inferior. Es el estilo más usado para aplicaciones de negocio, a menudo por defecto más que por decisión, porque la mayoría de los frameworks web generan un proyecto con carpetas para las vistas, la lógica de negocio y los modelos.

La idea es antigua y procede del software de sistemas. Edsger Dijkstra describió el sistema operativo multiprogramado THE (Communications of the ACM, 1968) como una jerarquía de niveles, cada uno construido solo sobre los que tiene debajo, de modo que cada nivel pudiera analizarse y probarse por sí solo. Las pilas de protocolos de red, como el modelo de referencia OSI, usan el mismo principio. En el software de negocio, la organización en capas llegó con los sistemas cliente-servidor y de tres niveles en la década de 1990, y POSA (1996) la catalogó como el patrón Capas (Layers). Garlan y Shaw incluyeron los sistemas en capas entre sus estilos básicos, con la restricción de que las interacciones se limitan a capas adyacentes.

La aplicación de negocio típica tiene cuatro capas. La **capa de presentación** se ocupa de la interfaz de usuario y del protocolo: solicitudes HTTP, pantallas, JSON. La **capa de negocio** contiene las reglas y los casos de uso: cómo se calcula el precio de un pedido, cuándo puede cancelarse. La **capa de persistencia** convierte los objetos de negocio en datos almacenados y viceversa, normalmente mediante repositorios y un mapeador objeto-relacional. La **capa de base de datos** es la propia base de datos. Las aplicaciones pequeñas a veces fusionan negocio y persistencia; las más grandes añaden capas, como una capa de aplicación que coordina los casos de uso por encima de una capa de dominio que contiene el modelo central. El estilo no fija el número; fija la dirección de las dependencias.

Una aplicación de pedidos de comida en capas



Las llamadas solo fluyen hacia abajo; las capas cerradas no pueden saltarse, mientras que la capa abierta de servicios sí.

Cómo funciona: capas cerradas, capas abiertas y aislamiento

Cada capa puede ser cerrada o abierta. Una solicitud debe pasar por una capa cerrada: la capa de presentación no puede saltarse la capa de negocio para llegar a la capa de persistencia, aunque así fuera más rápido de escribir. Una capa abierta se puede saltar. El ejemplo habitual es una capa de servicios técnicos compartidos, como enviar correos electrónicos, generar recibos en PDF o escribir registros de auditoría. Solo algunas operaciones de negocio los necesitan, así que obligar a cada solicitud a pasar por esa capa añadiría código de reenvío inútil. En el diagrama, la capa de negocio puede llamar a la capa de servicios o pasar de largo directamente a la persistencia.

Las capas cerradas proporcionan capas de aislamiento. Como la capa de presentación solo conoce la interfaz de la capa de negocio, el equipo puede sustituir el mapeador objeto-relacional, reestructurar las tablas o pasar de un producto de base de datos a otro sin tocar ni un solo controlador o plantilla, siempre que la interfaz de la capa de negocio siga igual. Es la ocultación de información (David Parnas, 1972) aplicada a todo un estrato del código, y es el principal beneficio que ofrece el estilo. Cada capa abierta, y cada excepción que permite a una capa saltarse otra, lo debilita: si los controladores consultan directamente la base de datos, un cambio de esquema se propaga ahora también a la capa de presentación.

La dirección de las dependencias es la otra mitad de la regla. Las capas inferiores no deben saber nada de las superiores: la capa de persistencia nunca debe importar un controlador, y la capa de negocio nunca debe construir HTML. Cuando una capa inferior necesita avisar a una superior, lo hace a través de una abstracción que pertenece a la capa inferior, como una interfaz de callback o un evento, de modo que en el código la dependencia siga apuntando hacia abajo. La Sección 3.4 trata en profundidad la dirección de las dependencias, los ciclos y la organización en capas estricta frente a la relajada.

Conviene no confundir capas y niveles. Una capa es una agrupación lógica en el código. Un nivel (tier) es un límite físico en tiempo de ejecución, como un servidor o un proceso separado. Un sistema clásico de tres niveles ejecuta el navegador, un servidor de aplicaciones y un servidor de base de datos como tres niveles, mientras que el servidor de aplicaciones por sí solo puede contener tres o cuatro capas. Convertir una capa en un nivel pone una red entre dos capas que antes se llamaban en memoria, lo que multiplica el costo de cada llamada; rara vez merece la pena hacerlo por sí mismo.

Ejemplo: una aplicación web clásica

Piensa en el sitio web de pedidos de una empresa regional de reparto de comida: los clientes consultan las cartas, hacen pedidos y pagan; el personal de los restaurantes ve los pedidos entrantes. Un equipo de cinco personas lo construye con un framework web convencional y una base de datos PostgreSQL. Una estructura en capas es el punto de partida obvio, y la organización del código hace visibles las capas.

```
foodshop/
  presentation/          # HTTP only: parse requests, render responses
    order_views.py
    menu_views.py
    templates/
  business/              # rules and use cases; no HTTP, no SQL
    order_service.py
    pricing.py
  services/              # open layer: shared technical services
    email.py
    receipts.py
  persistence/           # all database access
    order_repository.py
    menu_repository.py
    models.py
  settings.py
```

Estructura de paquetes de la aplicación de pedidos de comida en capas: un paquete de nivel superior por capa.

EJEMPLO PRÁCTICO

Ejemplo práctico: seguir una solicitud capa a capa hacia abajo

Un cliente hace clic en **Hacer pedido**. La solicitud llega como `POST /orders` con el carrito y la dirección de entrega.

Presentación	`order_views.place_order` analiza el JSON, comprueba que está bien formado y que el usuario ha iniciado sesión, y llama a `order_service.place(customer_id, items, address)`. No sabe nada de tablas ni de precios.
Negocio	`order_service.place` comprueba que el restaurante está abierto, calcula el precio del carrito con `pricing.py` (descuentos, gastos de envío, impuestos) y pide a `order_repository` que guarde el pedido. Llama a `receipts.send` en la capa abierta de servicios para enviar un recibo por correo electrónico.
Persistencia	`order_repository.save` convierte el pedido en filas de `orders` y `order_lines` y las escribe en una sola transacción de base de datos.
Base de datos	PostgreSQL confirma la transacción y responde. El resultado sube de vuelta: el servicio devuelve el nuevo pedido y la vista genera `201 Created` con el número de pedido.
Solicitud de cambio	Producto pide una propina para el repartidor. El desarrollador añade un campo de propina a la solicitud y a la respuesta (presentación), una regla de propina máxima y la propina en el total (negocio), y una columna y un mapeo (persistencia), además de una migración (base de datos). Cuatro capas para una funcionalidad pequeña.

Cada capa hizo un solo tipo de trabajo, y cada una pudo probarse por separado sustituyendo la capa inferior por un doble de prueba.

La propina muestra el costo del particionamiento técnico: incluso un pequeño cambio de negocio atraviesa todas las capas, así que toca más archivos, más revisores y, si los equipos están divididos por capa, más equipos.

Las consecuencias del particionamiento técnico

Como la organización en capas es particionamiento técnico, el dominio de negocio queda repartido entre las capas. No hay un único lugar llamado “entrega”: hay controladores de entrega, servicios de entrega y repositorios de entrega, intercalados con todo lo demás en cada capa. Un desarrollador que quiera entender cómo funcionan las propinas tiene que leerlos todos. A medida que el sistema crece, cada capa se convierte en una colección grande y de baja cohesión de código sin relación: la capa de negocio contiene juntas las reglas de pedidos,

pagos, cartas y repartidores, y nada impide que el código de pagos llame a las profundidades del código de cartas. Con el tiempo, las capas siguen limpias, pero el interior de cada una se convierte en una gran bola de barro.

También hay un costo organizativo. Organizar los equipos por capa (un equipo de frontend, un equipo de backend, un equipo de base de datos) significa que casi cada funcionalidad necesita un traspaso y una cola entre equipos, algo que la ley de Conway acaba consolidando en la arquitectura. Martin Fowler considera por este motivo que los equipos organizados por capa son un antipatrón: los desarrolladores no tienen por qué ser full-stack, pero los equipos sí. Una arquitectura en capas no obliga a dividir los equipos por capa, pero hace que sea el camino de menor resistencia.

Por último, un monolito en capas es un solo cuanto de arquitectura: una unidad de despliegue y normalmente una base de datos. Cada cambio, por pequeño que sea, significa compilar, probar y volver a desplegar toda la aplicación, y toda la aplicación escala y falla como una sola. A menudo eso es perfectamente aceptable, pero pone un techo.

Un corte alternativo: la arquitectura de porciones verticales

La **arquitectura de porciones verticales (vertical slice architecture)** de Jimmy Bogard (2018) organiza el código en torno a solicitudes individuales en lugar de capas. Cada funcionalidad, como *hacer pedido* o *cancelar pedido*, contiene su propia gestión, validación y acceso a datos, desde el inicio de la solicitud hasta la base de datos. Su objetivo es "minimizar el acoplamiento entre porciones y maximizar el acoplamiento dentro de una porción", de modo que un cambio toque una funcionalidad y no todas las capas. El precio es la estructura compartida que aportaban las capas: la duplicación puede acumularse y los manejadores pueden hincharse, y Bogard dice que el enfoque presupone un equipo que reconozca los code smells y refactorice. Las porciones también se combinan bien con los módulos: los módulos dividen los dominios y las porciones dividen las funcionalidades dentro de un módulo.

El sumidero de arquitectura y otros antipatrones

El antipatrón del sumidero de arquitectura (architecture sinkhole), un nombre popularizado por Mark Richards, describe solicitudes que fluyen por capas que no hacen nada más que pasarlas a la siguiente. Ver una carta es el caso típico: el controlador llama a un método de servicio que llama a un método de repositorio que ejecuta una consulta, y el controlador y el servicio no aportan nada. Cada capa de simple paso añade código que escribir, probar y leer, y un poco de latencia. Algo de paso directo es el precio del aislamiento y no supone un problema. Richards propone una regla general: si aproximadamente una quinta parte de las solicitudes son de simple paso, es normal; si lo son la mayoría, probablemente la aplicación es sobre todo acceso

a datos con poca lógica de negocio, y la organización en capas es el estilo equivocado o habría que abrir las capas. Abrir capas, sin embargo, supone renunciar al aislamiento, así que decídelo de forma deliberada y deja constancia del motivo.

Vale la pena reconocer pronto otros antipatrones habituales. **Capas con fugas**: SQL en los controladores, objetos de solicitud HTTP pasados a los servicios o entidades del ORM convertidas directamente en JSON, de modo que un cambio en una capa se propaga a otras. **Dependencias hacia arriba**: un repositorio que llama a un servicio, o un objeto de dominio que da formato a HTML. **La capa de negocio anémica**: servicios que solo mueven datos de un lado a otro mientras las reglas reales viven en los controladores o en procedimientos almacenados; Fowler bautizó el antipatrón relacionado Anemic Domain Model (modelo de dominio anémico) en 2003. **El servicio dios**: un único `OrderService` de varios miles de líneas al que llaman todos los demás servicios. **La capa de utilidades de la que todo depende**: un paquete `common` que poco a poco acumula lógica de negocio y acopla todo con todo.

Sumidero frente a organización en capas útil



Las solicitudes de simple paso cuestan código y latencia sin añadir comportamiento; unas pocas son normales, una mayoría es una señal de alerta.

Características del estilo en capas

Característica	Valoración	Motivo
Sencillez	Alto	Conocido por casi todos los desarrolladores y estructura predeterminada de la mayoría de los frameworks; fácil de explicar y de empezar.
Costo total	Alta (barato)	Una base de código, un despliegue, una base de datos; infraestructura y operación mínimas.
Capacidad de prueba (testabilidad)	Media	Cada capa puede probarse sustituyendo la capa inferior por un doble de prueba, pero las pruebas realistas a menudo necesitan toda la pila y una base de datos.
Modificabilidad ante cambios de negocio	De baja a media	Los cambios técnicos se quedan dentro de una capa, pero la mayoría de los cambios de funcionalidad cruzan todas las capas.
Desplegabilidad	Bajo	Cualquier cambio obliga a volver a desplegar toda la aplicación; el riesgo de cada lanzamiento crece con el tamaño.
Escalabilidad y elasticidad	Bajo	Toda la aplicación escala como una sola unidad; no puedes escalar solo la parte con más carga, y el arranque de una aplicación grande ralentiza el escalado elástico.
Tolerancia a fallos	Bajo	Una sola fuga de memoria o una consulta descontrolada pueden tumbar todo el proceso; no hay aislamiento entre funcionalidades.
Rendimiento	Media	Las llamadas en proceso son rápidas, pero las capas adicionales y el mapeo de objetos añaden sobrecarga, y los sumideros añaden más.
Capacidad de evolución a gran escala	Bajo	La baja cohesión dentro de cada capa y el acoplamiento entre funcionalidades hacen que los sistemas grandes en capas sean difíciles de reestructurar.

SI ESTO ES NUEVO PARA TI

Paso a paso: una capa que parece no hacer nada

La idea más difícil de esta sección es el compromiso que hay detrás del sumidero de arquitectura. Una capa que solo pasa una solicitud a la siguiente parece código desperdiciado, pero la regla que obliga a las solicitudes a pasar por ella es lo que mantiene aisladas las capas. Sigue dos solicitudes a través de la aplicación de pedidos de comida.

- Un cliente consulta una carta. El controlador llama a un método de la capa de negocio, que simplemente llama al repositorio de cartas, que ejecuta la consulta. El controlador y el método de negocio solo pasan la solicitud a la siguiente capa.
- Un desarrollador propone dejar que el controlador llame directamente al repositorio. Ahorraría código, pero convertiría la capa de negocio en una capa abierta: una que las solicitudes pueden saltarse.
- Meses después, el equipo reestructura las tablas de cartas. Pasando por la capa de negocio, el controlador no tendría que cambiar, siempre que la interfaz de esa capa siguiera igual. Con el atajo, el cambio llega también al controlador.
- Hacer un pedido es distinto: el controlador comprueba la entrada, la capa de negocio calcula el precio del carrito y comprueba las reglas, y el repositorio guarda el pedido en una sola transacción. Cada capa hace trabajo real.
- Así que la pregunta útil es cuántas solicitudes son como la de la carta. La regla general de Richards: que aproximadamente una quinta parte sean de simple paso es normal. Si lo son la mayoría, probablemente la aplicación es sobre todo acceso a datos con pocas reglas. Entonces puede que otro estilo le encaje mejor, o que el equipo abra algunas capas a propósito y deje constancia del motivo, aceptando la pérdida de aislamiento.
- **La lección:** algo de paso directo es el precio de las capas de aislamiento. Cuéntalo en lugar de eliminarlo en cuanto lo veas, y trata una gran mayoría de solicitudes de simple paso como señal de que el estilo no encaja.

Cuándo encaja bien y cuándo no

La arquitectura en capas es una buena opción para aplicaciones pequeñas y medianas construidas por un solo equipo, para productos cuyo dominio todavía no se entiende bien (el particionamiento por dominio necesita límites que aún no conoces), para presupuestos y plazos ajustados y para aplicaciones que son sobre todo formularios sobre datos, con reglas modestas. También es una estructura interna razonable para un único módulo o servicio que forma parte, a su vez, de un sistema con particionamiento por dominio: un módulo `delivery` en capas dentro de un monolito modular obtiene lo mejor de ambos.

Es una mala opción como estructura de nivel superior de un sistema grande con varios equipos, porque todos los equipos trabajan en todas las capas y las funcionalidades chocan. Es mala cuando partes del sistema necesitan una escalabilidad o una disponibilidad muy distintas, porque solo hay un cuanto. Y es mala cuando la lógica de negocio es rica y central, porque las reglas quedan dispersas entre controladores, servicios y procedimientos almacenados; en ese caso, una estructura interna centrada en el dominio, como la arquitectura hexagonal o limpia, mantiene el núcleo independiente de la base de datos y del framework web.

La evolución más habitual es pasar de un monolito en capas a un monolito modular: mantener un único despliegue, pero reorganizar el nivel superior por dominio y conservar las capas dentro de cada módulo.

Hacer cumplir las reglas de las capas en el código

Las reglas de las capas se deterioran rápido. Con la presión de una fecha de entrega, un desarrollador importa el repositorio directamente en un controlador porque le ahorra veinte minutos, nadie lo nota en la revisión y el siguiente desarrollador copia el patrón. La defensa es una prueba de arquitectura: una comprobación automatizada de las dependencias que se ejecuta en la compilación y la hace fallar cuando se incumple una regla, lo que convierte la regla en una función de aptitud.

En Python, import-linter lee contratos de un archivo de configuración. Su contrato `layers` enumera las capas de la más alta a la más baja y prohíbe que una capa inferior importe una superior. Por sí solo no hace que las capas sean cerradas (una capa superior puede importar cualquier capa inferior), así que un contrato `forbidden` añade la regla de que la presentación no debe importar directamente la persistencia; necesita `allow_indirect_imports` activado, porque de lo contrario la cadena normal de la presentación a la persistencia pasando por el negocio contaría como una infracción. En Java, ArchUnit expresa las mismas reglas como una prueba unitaria: `layeredArchitecture()` declara las capas por paquete y establece, por ejemplo, que solo la capa de servicios puede acceder a la capa de persistencia. Existen herramientas similares para otros ecosistemas, como dependency-cruiser para JavaScript y TypeScript.

Dos prácticas hacen que las comprobaciones perduren. Introdúcelas registrando las infracciones existentes como excepciones conocidas, para que la compilación pase a verde de inmediato y solo las infracciones nuevas la hagan fallar; después, reduce las excepciones con el tiempo. Y mantén la configuración junto al código y revisa sus cambios como si fueran código, porque relajar una regla es una decisión arquitectónica que merece un registro de decisiones.

```
[importlinter]
root_package = foodshop

[importlinter:contract:layers]
name = Layers only depend downwards
type = layers
layers =
    foodshop.presentation
    foodshop.business
    foodshop.services
    foodshop.persistence

[importlinter:contract:closed-business]
name = Presentation must go through business
type = forbidden
source_modules = foodshop.presentation
forbidden_modules = foodshop.persistence
# presentation → business → persistence is the normal path
allow_indirect_imports = True
```

Una configuración de import-linter (.importlinter) para la aplicación de pedidos de comida: las capas solo pueden depender hacia abajo, y la presentación no puede saltarse la capa de negocio.

PARA PROFUNDIZAR

Capas dentro de otros estilos

La organización en capas sobrevive dentro de la mayoría de los demás estilos. Cada módulo de un monolito modular, cada microservicio y cada plug-in pueden estar organizados en capas por dentro. Las arquitecturas hexagonal y limpia pueden leerse como una organización en capas con un cambio crucial: la dependencia entre la lógica de negocio y la persistencia se invierte, de modo que el código de base de datos depende de interfaces definidas por la capa de negocio y no al revés. Esa inversión es lo que permite probar el núcleo de negocio sin una base de datos y que sobreviva a sus frameworks.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Qué diferencia hay entre una capa cerrada y una capa abierta?**

Una solicitud debe pasar por una capa cerrada y no puede saltársela; la capa superior puede saltarse una capa abierta, algo típico de una capa de servicios compartidos que solo necesitan algunas solicitudes.

▼ ¿Por qué una aplicación en capas tiende a necesitar cambios en todas las capas para una nueva funcionalidad?

Porque tiene particionamiento técnico: el código de cada capacidad de negocio está repartido entre las capas de presentación, negocio y persistencia, y la mayoría de las funcionalidades se definen en términos de negocio.

▼ ¿Qué diferencia hay entre una capa y un nivel (tier)?

Una capa es una agrupación lógica del código por responsabilidad; un nivel es un límite físico en tiempo de ejecución, como un servidor separado. Normalmente, varias capas se ejecutan dentro de un mismo nivel.

▼ ¿Por qué un contrato layers de import-linter, por sí solo, no hace que las capas sean cerradas?

Solo prohíbe que las capas inferiores importen las superiores; una capa superior puede seguir importando cualquier capa inferior, así que hace falta un contrato forbidden aparte para impedir que la presentación importe la persistencia.

▼ Inténtalo: Trazas 500 solicitudes en una app de venta de entradas en capas, y 400 pasan directamente por las capas de negocio y persistencia sin que se añada lógica. ¿Cómo se llama esto, qué sugiere y qué comprobarías antes de actuar?

Es el antipatrón del sumidero de arquitectura. Algo de paso directo es el precio normal del aislamiento, alrededor de una quinta parte según la regla práctica habitual, pero aquí es el 80%, lo que sugiere que las capas pueden ser el estilo equivocado. Comprueba primero si esas solicitudes son en su mayoría lecturas simples y si el resto del sistema gana de verdad con el aislamiento. Las opciones incluyen abrir una capa o pasar a la partición por dominios. No reescribas basándote en una sola muestra.

FUENTES

- Martin Fowler: Presentation Domain Data Layering: https://martinfowler.com/bliki/PresentationDomainDataLayering.html
- Microsoft Azure Architecture Center: N-tier architecture style: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/n-tier
- import-linter documentation: Layers contract: https://import-linter.readthedocs.io/en/stable/contract_types/layers/
- ArchUnit user guide (layer checks): https://www.archunit.org/userguide/html/000_Index.html
- Jimmy Bogard: Vertical Slice Architecture (2018): https://www.jimmybogard.com/vertical-slice-architecture/

SECCIÓN 5.3 DE 5.8 · INTERMEDIO · 26 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

El monolito modular

ANTES DE LEER ESTO

Esta sección se basa en:

- 3.3 Ocultación de información
- 5.1 Estilos y patrones
- 5.2 En capas

Términos anteriores que utiliza: Unidad de despliegue, Particionamiento por dominio, Gran bola de barro (big ball of mud), Prueba de arquitectura, Transacción

LA IDEA EN BREVE

Piensa en una gran tienda por departamentos. Es un solo edificio con una sola puerta de entrada, un solo horario de apertura y un solo equipo de seguridad, pero por dentro está dividido en departamentos: alimentación, ropa, electrónica. Cada departamento tiene su propio personal, su propio almacén y su propia caja, y si ropa necesita algo de alimentación, lo pide en el mostrador de alimentación en lugar de meterse en el almacén de alimentación. Un monolito modular es software construido como esa tienda: un solo programa, desplegado como una sola pieza, pero dividido por dentro en departamentos de negocio que guardan sus propios datos y solo se relacionan entre sí a través de mostradores acordados.

LO QUE NECESITAS SABER

- Un monolito modular es una única unidad de despliegue cuyo nivel superior está dividido por dominio de negocio en módulos, cada uno con una API pública pequeña y explícita, partes internas privadas y sus propios datos.
- Conserva la simplicidad operativa de un monolito (una compilación, un despliegue, llamadas en proceso, transacciones locales) y a la vez ofrece muchas de las ventajas de modularidad que se buscan en los microservicios.
- Los límites se erosionan si no se hacen cumplir: visibilidad del lenguaje, pruebas de arquitectura (ArchUnit, import-linter), la verificación de Spring Modulith para aplicaciones Spring, Packwerk para Ruby y esquemas de base de datos separados por módulo.
- Cada tabla pertenece exactamente a un módulo; los demás módulos usan su API o reaccionan a sus eventos, nunca a sus tablas. Los joins entre módulos se sustituyen por llamadas a la API o por modelos de lectura.

- Los módulos pueden comunicarse mediante llamadas en proceso (simples, inmediatas, transaccionales) o eventos en proceso (menor acoplamiento, pero el trabajo del listener se hace por separado y puede fallar por separado).
- Shopify reorganizó su gran monolito de Ruby on Rails en componentes de dominio a partir de 2017 y lo describió en 2019; sus publicaciones posteriores muestran tanto el valor del enfoque como lo difícil que es el aislamiento completo.
- Un módulo con límites bien definidos puede extraerse más adelante a un servicio con un esfuerzo moderado, porque su API, sus datos y sus dependencias ya son explícitos.

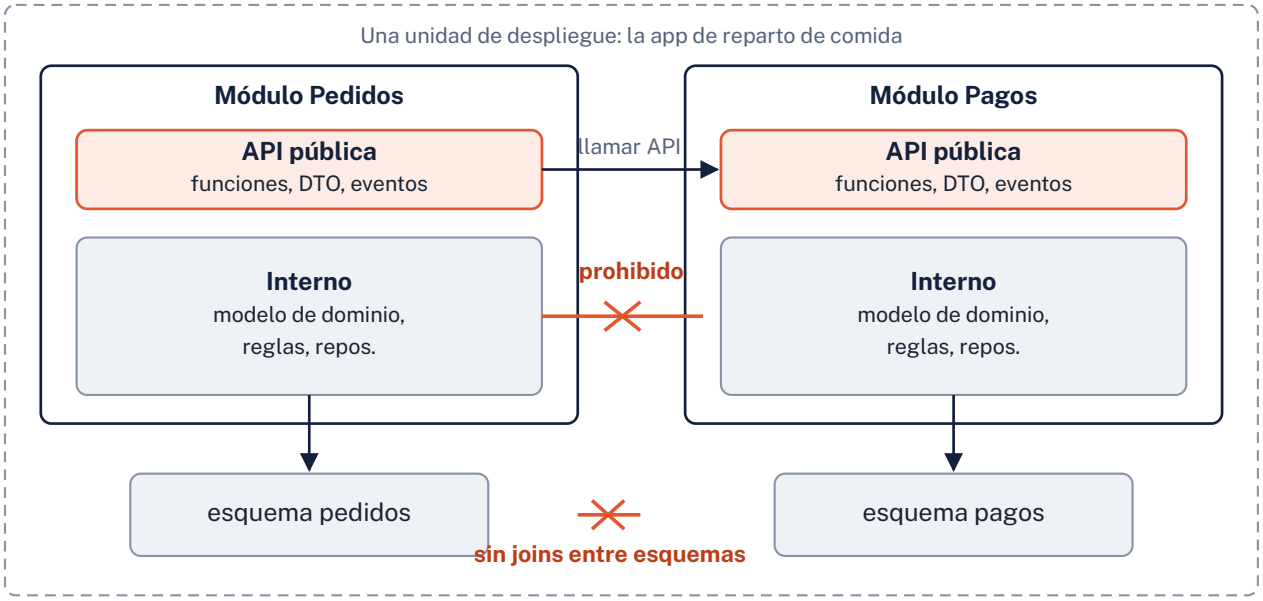
Definición: un despliegue, muchos dominios

Un monolito modular es una aplicación que se construye y se despliega como una sola unidad, pero cuyo código está dividido, en el nivel superior, en módulos que siguen límites de negocio: pedidos, pagos, reparto, cartas. Cada módulo tiene una API pública pequeña y deliberada que otros módulos pueden usar, una parte interna más grande que nadie de fuera puede tocar, y sus propios datos. Combina el particionamiento por dominio de los microservicios con la unidad de despliegue única de un monolito clásico.

La idea no es nueva. Los módulos con partes internas ocultas se remontan a la ocultación de información de David Parnas (1972), y los monolitos grandes y bien estructurados siempre han existido. Lo más reciente es el nombre y la atención que recibe. Tras una década en la que muchas organizaciones dividieron sus sistemas en microservicios y descubrieron el costo de la distribución, profesionales como Simon Brown y Sam Newman sostuvieron que el monolito modular es una opción infravalorada y que a muchos equipos les iría mejor con uno que con un conjunto de servicios. Richards y Ford le dedican un capítulo propio en la segunda edición de *Fundamentals of Software Architecture* (2025).

La diferencia con el estilo en capas está en el corte del nivel superior. Una aplicación en capas se divide en presentación, negocio y persistencia, con cada dominio repartido entre las tres. Un monolito modular se divide en dominios, y cada módulo puede organizarse en capas por dentro. La diferencia con los microservicios es el despliegue: los módulos comparten un proceso, un lanzamiento y, normalmente, un servidor de base de datos, y se llaman entre sí en memoria.

Anatomía de un monolito modular



Cada módulo expone una API pública pequeña y es dueño de su propio esquema; meterse en las partes internas o en las tablas de otro módulo está prohibido.

API de los módulos y código interno

La regla más importante de todas es que un módulo solo se usa a través de su API. La API es el conjunto de funciones, objetos de transferencia de datos (DTO) y eventos que los responsables del módulo se comprometen a mantener estables; todo lo demás, incluidos el modelo de dominio, los repositorios y las tablas de la base de datos, es interno y puede cambiar en cualquier momento. En la práctica, la API es pequeña: un módulo de pagos podría exponer `authorise_payment`, `refund`, un tipo `PaymentResult` y un evento `PaymentCaptured`, mientras oculta decenas de clases para proveedores, reintentos, libros contables y conciliación.

Las buenas API de módulo comparten varios rasgos. Hablan el lenguaje del dominio (`authorise_payment`, no `insert_payment_row`). Pasan datos simples, como identificadores y pequeños registros inmutables, en lugar de objetos internos, para que quien llama no pueda atravesar un objeto devuelto y llegar a las partes internas del módulo. Son de grano lo bastante grueso como para que un caso de uso típico necesite una o dos llamadas, no veinte. Y se diseñan como si algún día pudiera haber una red entre quien llama y quien es llamado: sin objetos mutables compartidos, sin depender de la transacción de base de datos de quien llama, con errores explícitos. Este último rasgo es el que conserva la opción de extraer el módulo más adelante.

Dentro del límite vale todo lo que al equipo del módulo le resulte útil: una estructura en capas, una hexagonal, un modelo de dominio rico o simples scripts de transacción. Distintos módulos pueden tomar decisiones distintas, porque nada de fuera depende de ellas.

```
# delivery_platform/payments/__init__.py  (the module's public API)
from .api import authorise_payment, refund, PaymentResult
from .events import PaymentCaptured

__all__ = ["authorise_payment", "refund", "PaymentResult", "PaymentCaptured"]

# delivery_platform/ordering/internal/checkout.py
from delivery_platform import payments             # allowed: public API
# from delivery_platform.payments.internal import ledger   # forbidden

def place_order(order, card_token):
    result = payments.authorise_payment(order.id, order.total, card_token)
    if not result.ok:
        raise PaymentDeclined(result.reason)
    order.confirm(result.payment_id)
    events.publish(OrderPlaced(order.id, order.restaurant_id))
```

Un paquete de Python como módulo: el paquete payments exporta una API pequeña, y ordering importa solo esa API.

Hacer cumplir los límites

En un monolito, técnicamente cualquier clase puede llamar a cualquier otra, así que los límites solo se mantienen si algo los hace cumplir. Sin ese control, un atajo hacia las partes internas de otro módulo, tomado por la presión de un plazo, pasa inadvertido en la revisión, y un monolito modular degenera en una gran bola de barro (big ball of mud) en uno o dos años. Hay varias capas de defensa, y lo mejor es usarlas juntas.

Visibilidad del lenguaje. Usa la ocultación más fuerte que ofrezca el lenguaje: clases privadas de paquete en Java, `internal` en Kotlin y C# (que impone límites solo entre módulos de compilación o ensamblados, no entre paquetes de un mismo proyecto), los `exports` del Java Platform Module System, módulos de compilación separados (subproyectos de Maven o Gradle, proyectos .NET) cuyas dependencias se declaran explícitamente. Python y Ruby solo ofrecen convenciones (un guion bajo inicial, un subpaquete `internal`), así que las herramientas tienen que hacer el trabajo.

Pruebas de arquitectura. Las pruebas de arquitectura comprueban las dependencias en la compilación. ArchUnit (Java) puede afirmar que ninguna clase de fuera de un módulo toca sus paquetes internos y que las dependencias entre módulos no tienen ciclos. import-linter (Python) ofrece un contrato `forbidden`, que puede impedir que otros módulos importen el paquete interno de un módulo sin dejar de permitirles usar su API pública; un contrato `layers` cuyas capas pueden contener varios hermanos mutuamente independientes separados por `|`,

lo que permite comprobar un orden de dependencia en un solo sentido entre módulos; y un contrato `independence` para paquetes que no deben depender unos de otros en absoluto, ni siquiera de forma indirecta.

Soporte del framework. Spring Modulith, para aplicaciones Spring Boot, trata cada subpaquete directo del paquete principal de la aplicación como un módulo, trata el paquete base del módulo como su API y sus subpaquetes como internos, y ofrece una verificación de una sola línea (`ApplicationModules.of(Application.class).verify()`) que falla cuando un módulo hace referencia a las partes internas de otro, cuando las dependencias entre módulos forman un ciclo o, si un módulo enumera sus dependencias permitidas, cuando usa una que no está en la lista. También puede generar diagramas de componentes de los módulos al estilo C4. En el mundo Ruby, Packwerk, la herramienta de código abierto de Shopify, analiza estáticamente las referencias entre paquetes, cada uno declarado mediante un archivo `package.yml`, e informa de las dependencias no declaradas.

Límites de datos. Dale a cada módulo su propio esquema de base de datos (o al menos sus propias tablas con un prefijo claro) e, idealmente, su propio usuario de base de datos que solo pueda acceder a ese esquema. La revisión de código por sí sola no impedirá un `JOIN` cómodo entre módulos; los permisos de la base de datos sí.

```
[importlinter:contract:payments-internals]
name = Only payments may use payments internals
type = forbidden
source_modules =
    delivery_platform.ordering
    delivery_platform.delivery
    delivery_platform.reporting
forbidden_modules =
    delivery_platform.payments.internal
# going through the public API (which itself uses internals) is fine
allow_indirect_imports = True

[importlinter:contract:module-order]
name = Module dependencies point one way
type = layers
layers =
    delivery_platform.reporting | delivery_platform.notifications
    delivery_platform.delivery
    delivery_platform.ordering | delivery_platform.restaurants
    delivery_platform.payments | delivery_platform.menus | delivery_platform.customers
```

Contratos de import-linter (en la práctica, uno por módulo): otros módulos no pueden importar las partes internas de payments, y las dependencias entre módulos deben seguir una sola dirección, con los módulos de una misma línea independientes entre sí.

Propiedad de los datos dentro de un monolito

La propiedad de los datos es donde los monolitos modulares fallan con más frecuencia. Compartir un servidor de base de datos está bien; compartir tablas, no. Si el módulo de reparto lee directamente la tabla `orders`, el equipo de pedidos ya no puede cambiar esa tabla sin coordinarse con reparto, y los dos módulos quedan tan acoplados como si fueran uno. La regla es fácil de enunciar: cada tabla tiene exactamente un módulo propietario, solo el propietario escribe en ella, y los demás módulos obtienen los datos a través de la API del propietario o de los eventos que este publica.

La regla tiene consecuencias que los equipos deben aceptar de forma deliberada. Un informe que antes era una sola consulta SQL que unía pedidos, pagos y repartidores ahora necesita o bien llamadas a las API de tres módulos, o bien un modelo de lectura: un módulo de informes que se suscribe a los eventos de los demás y mantiene su propia copia desnormalizada, optimizada para consultas. Las claves foráneas entre esquemas de módulos suelen eliminarse, porque harían que el esquema de un módulo dependiera del de otro; en su lugar, la integridad entre módulos la mantiene la aplicación. Estos costos son reales, pero son los mismos que impondría un paso posterior a servicios, pagados antes y en un entorno más barato.

Queda una ventaja que los sistemas distribuidos pierden: los módulos de un mismo proceso pueden, cuando de verdad es necesario, compartir una única transacción local de base de datos. Úsala con moderación. Cada uso acopla operativamente los datos de los módulos y habrá que deshacerlo si algún día se extrae un módulo.

SI ESTO ES NUEVO PARA TI

Paso a paso: un propietario para cada tabla

La idea que requiere más atención aquí es la propiedad de los datos dentro de un solo programa. Los módulos comparten un servidor de base de datos, pero cada tabla tiene un único propietario: solo ese módulo puede leerla o escribirla directamente. Sigue un atajo y un informe a través de la plataforma de reparto de comida.

- El módulo Reparto necesita los datos de cada pedido nuevo. La tabla de pedidos está en la misma base de datos, así que el código más rápido de escribir es una consulta que la lee directamente.
- Meses después, Pedidos reorganiza esa tabla y sustituye su columna de estado por dos horas separadas, la de realización del pedido y la del pago. La consulta de Reparto deja de funcionar, y a partir de entonces los dos equipos deben planificar juntos cada cambio de la tabla, tan acoplados como si fueran un solo módulo.
- La regla elimina el atajo. Solo Pedidos toca sus tablas; los demás módulos usan la API pública de Pedidos o los eventos en proceso que publica, mensajes dentro del programa que anuncian que algo ha ocurrido. En el diseño que aparece más adelante en esta sección, Reparto escucha OrderPlaced en lugar de hacer consultas.
- Los informes son más difíciles. Un informe de cancelaciones que antes era una sola consulta SQL que unía pedidos, reembolsos y clientes se metería en las tablas de tres módulos.
- Hay dos salidas: llamar a las API de los tres módulos y combinar las respuestas, o dejar que Informes mantenga un modelo de lectura: escucha los eventos de los módulos, como OrderCancelled, y guarda su propia copia solo de los datos que necesita el informe.
- Las claves foráneas entre tablas de distintos módulos también suelen eliminarse; es la aplicación, no la base de datos, la que mantiene correctos esos vínculos.
- **La lección:** compartir un servidor de base de datos está bien; compartir tablas, no. Los eventos y los modelos de lectura suponen trabajo extra, pero es el mismo trabajo que necesitaría un paso posterior a servicios, hecho antes y de forma más barata.

Comunicación entre módulos: llamadas o eventos

Los módulos se comunican de dos maneras, ambas en memoria. Una **llamada en proceso** a la API de otro módulo es simple, rápida e inmediata: pedidos llama a `payments.authorise_payment` y obtiene el resultado al instante, y una excepción se propaga con normalidad. Crea una dependencia directa, de quien llama a quien es llamado, y quien llama no puede terminar hasta que termine el otro. Usa llamadas cuando quien llama necesita una respuesta para continuar, como saber si se autorizó el pago.

Un evento en proceso invierte la dependencia. Pedidos publica `OrderPlaced` y no sabe ni le importa quién escucha; reparto, notificaciones e informes se suscriben. Añadir una nueva reacción significa añadir un listener, sin tocar pedidos. Usa eventos cuando quien publica no necesita respuesta y cuando varios módulos reaccionan al mismo hecho.

Los eventos introducen sutilezas que los equipos que no los conocen suelen pasar por alto. Si un listener se ejecuta dentro de la transacción de quien publica, un fallo en el listener puede revertir el pedido; si se ejecuta después del commit, el pedido queda guardado, pero el trabajo del listener puede perderse si el proceso se cae entre medias. Spring Modulith lo resuelve con un registro de publicación de eventos que anota cada par evento-listener en la misma transacción que el cambio de negocio y solo lo marca como completado cuando el listener termina con éxito, de modo que los incompletos se puedan reintentar; un equivalente hecho a mano es la bandeja de salida transaccional (transactional outbox) descrita en la Sección 6.7. Los listeners también deben ser idempotentes, para que un evento reintentado no cause daños. Ningún estilo es mejor; la mayoría de los monolitos modulares usan llamadas para las consultas y los comandos que necesitan respuesta, y eventos para notificar cosas que ya han ocurrido.

Migraciones y pruebas por módulo

La propiedad solo perdura si las herramientas la respetan. Da a cada módulo su propio esquema de base de datos y su propia carpeta de migraciones, para que los cambios de Pedidos vivan junto a su código y no puedan alterar discretamente las tablas de Reparto; Flyway, por ejemplo, puede ejecutarse una vez por esquema con ubicaciones de scripts y tablas de historial separadas. Un inicio de sesión de base de datos por módulo, con acceso solo a su propio esquema, convierte una consulta descarriada en un error. Las pruebas siguen las mismas líneas: prueba cada módulo a través de su API pública, sustituyendo a sus vecinos por dobles. `@ApplicationModuleTest` de Spring Modulith, por ejemplo, arranca por defecto un solo módulo. Conserva unas pocas pruebas de toda la aplicación para las conexiones entre módulos. Como todo se sigue desplegando a la vez, expandir y contraer sigue siendo aplicable durante un lanzamiento gradual.

El monolito modular de Shopify

La aplicación principal de Shopify es una de las mayores bases de código de Ruby on Rails que existen, en la que han trabajado más de mil desarrolladores durante más de una década. En “Deconstructing the Monolith” (Shopify Engineering, febrero de 2019), Kirsten Westeinde explicó por qué la empresa no la dividió en microservicios y, en su lugar, a partir de 2017, la convirtió en un monolito modular. El problema era conocido: el código de áreas de negocio sin relación entre sí estaba enredado, los cambios tenían efectos inesperados en otras partes y la

incorporación de personas nuevas era lenta. Se consideraron los microservicios y se concluyó que traían sus propios costos (pipelines e infraestructura separados, latencia y fallos de red, refactorizaciones más difíciles entre servicios), así que el equipo optó por conseguir modularidad sin múltiples unidades de despliegue.

El primer paso fue reorganizar el código, de la estructura estándar de Rails (modelos, vistas y controladores) a componentes con nombres de dominios de negocio, como pedidos, envíos, inventario y facturación, en un único gran cambio automatizado. El siguiente fue el aislamiento: cada componente debía tener una API pública y la propiedad exclusiva de sus datos, y una herramienta interna, Wedge, seguía el progreso de cada componente registrando las llamadas que cruzaban límites durante la batería de pruebas. Una actualización de septiembre de 2020 informaba de 37 componentes en un monolito central de unos 2.8 millones de líneas de Ruby, y ese mismo mes Shopify liberó como código abierto Packwerk, una herramienta de análisis estático para declarar y comprobar dependencias entre paquetes.

La historia es instructiva porque no es un éxito sencillo. La actualización de 2020 decía que las ideas originales seguían en pie, pero que casi todos los detalles habían cambiado, y que casi ningún componente tenía todavía un límite completo y sólido. Una retrospectiva de 2024 sobre Packwerk informaba de que se habían eliminado sus comprobaciones de privacidad, de que algunos paquetes organizados por nombre y no por comportamiento en tiempo de ejecución confundían a los desarrolladores, y de que aislar por completo incluso un paquete base llevaba muchos meses; Packwerk se había convertido en una protección de límites clave más que en el centro de la estrategia. La lección para los arquitectos es que un monolito modular es una dirección de viaje que necesita una inversión sostenida, y que incorporar límites a posteriori en una base de código grande ya existente es mucho más difícil que mantenerlos desde el principio.

EJEMPLO PRÁCTICO

Ejemplo práctico: la plataforma de reparto de comida como monolito modular

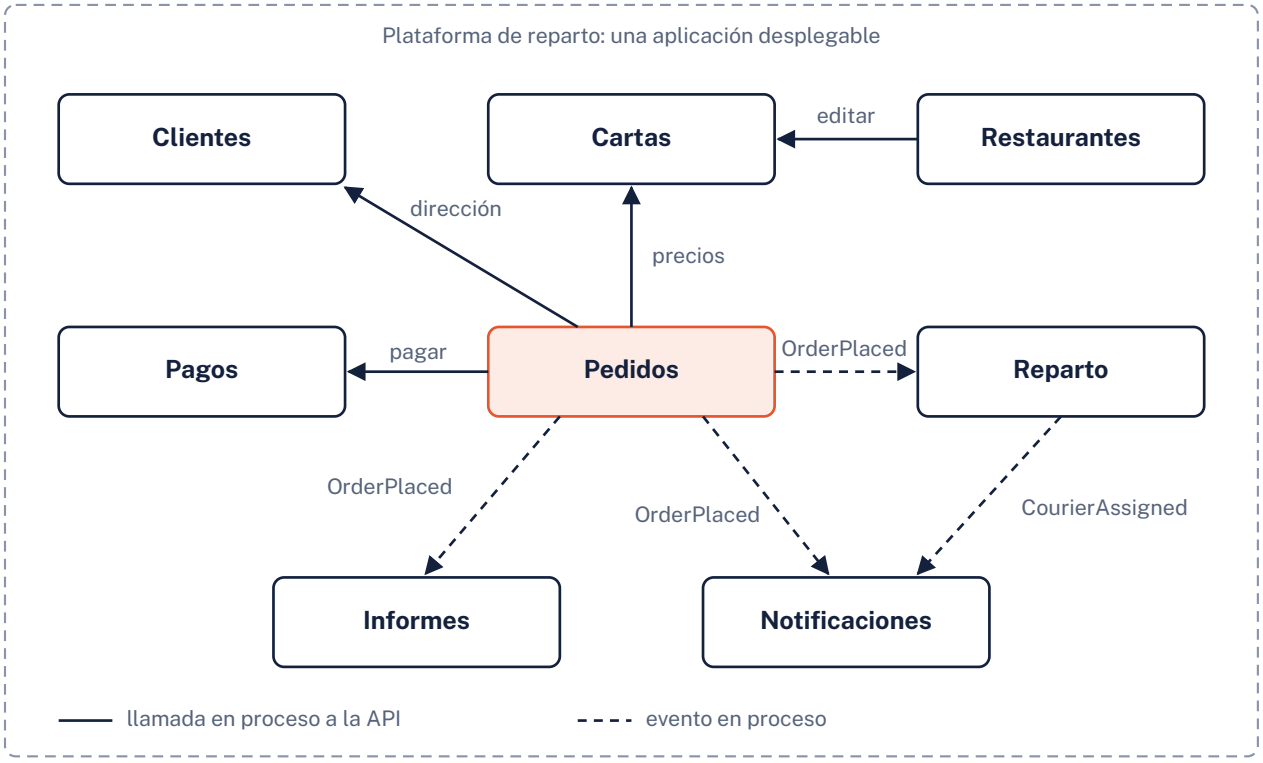
Una startup de reparto de comida con cuatro equipos (unos 25 ingenieros) quiere que cada equipo sea dueño de su parte de forma independiente, sin el costo de operar microservicios. El arquitecto propone ocho módulos en una única aplicación Python desplegable, con un servidor PostgreSQL y un esquema por módulo.

Clientes	Cuentas, direcciones, preferencias. API: `get_customer`, `get_delivery_address`. Equipo A.
Restaurantes; Cartas (dos módulos)	Restaurantes guarda las cuentas de los restaurantes y los horarios de apertura, y actualiza las cartas a través de la API de Cartas; Cartas ofrece precios y disponibilidad a todos los demás (`price_items`, `is_open`). Equipo B.
Pedidos	Carrito, proceso de pago, ciclo de vida del pedido. Llama a Clientes, Cartas y Pagos; publica `OrderPlaced` y `OrderCancelled`. Equipo C.
Pagos	Autorización, captura y reembolsos a través de un proveedor externo. API: `authorise_payment`, `refund`; publica `PaymentCaptured`. Equipo C.
Entrega	Asignación de repartidores y seguimiento. Escucha `OrderPlaced`; publica `CourierAssigned` y `Delivered`. Equipo D.
Notificaciones e Informes	Listeners puros: envían mensajes y mantienen modelos de lectura. No llaman a ningún otro módulo. Equipo A.
Regla de dependencia	Pagos, Cartas y Clientes no dependen de nadie; Pedidos y Restaurantes los llaman; Reparto reacciona a los eventos de Pedidos; Notificaciones e Informes solo escuchan. Lo comprueba import-linter en la compilación.
Despliegue	Una sola imagen de contenedor, ejecutada como seis réplicas detrás de un balanceador de carga, más un proceso worker (con el mismo código) que gestiona los listeners de eventos y los trabajos en segundo plano.

Cada equipo es dueño de módulos completos, así que la mayoría de los cambios caen en un solo módulo y un solo equipo.

La regla de dependencias hace imposibles los ciclos: si algún día Pagos necesita saber algo de los pedidos, recibe un identificador de pedido, no una importación de Pedidos.

Mapa de módulos de la plataforma de reparto de comida



Las flechas continuas son llamadas a la API de otro módulo; las discontinuas son eventos en proceso. Pedidos llama a tres módulos, y sus listeners solo dependen de los eventos que publica.

Características del monolito modular

Característica	Valoración	Motivo
Sencillez	De media a alta	Un solo despliegue y llamadas en proceso, pero encontrar buenos límites de módulo exige reflexión.
Costo total	Alta (barato)	Infraestructura y operación de un monolito; algo de esfuerzo extra para las herramientas que vigilan los límites.
Modificabilidad ante cambios de negocio	Alto	El particionamiento por dominio mantiene la mayoría de los cambios dentro de un módulo del que es dueño un solo equipo.
Capacidad de prueba (testabilidad)	De media a alta	Los módulos se pueden probar a través de sus API sustituyendo a sus vecinos; las pruebas del sistema completo se ejecutan en un solo proceso.
Desplegabilidad	De baja a media	Sigue siendo una unidad de despliegue: cada lanzamiento incluye todos los módulos, aunque ayuda que cada cambio tenga un radio de impacto menor.
Escalabilidad y elasticidad	De baja a media	Las réplicas escalan toda la aplicación; un módulo no se puede escalar por separado, aunque los procesos worker pueden separar la carga en segundo plano.
Tolerancia a fallos	Bajo	Un solo proceso: una caída o una fuga de memoria en cualquier módulo los afecta a todos.
Rendimiento	Alto	Las llamadas entre módulos son en memoria, sin serialización ni latencia de red.
Capacidad de evolución	Alto	Las API explícitas y la propiedad de los datos hacen que los módulos sean fáciles de reestructurar o extraer más adelante.

Cuándo usarlo, cuándo no y antipatrones

Un monolito modular es una opción por defecto sólida para productos nuevos cuyo dominio está razonablemente claro, para organizaciones de hasta unos pocos equipos, para sistemas que necesitan consistencia fuerte entre áreas de negocio y para equipos con capacidad de operaciones limitada. También es el destino natural al modernizar un monolito enredado: introducir límites dentro de él suele ser más barato y menos arriesgado que dividirlo en servicios de una sola vez. Es menos adecuado cuando partes del sistema necesitan atributos de calidad realmente distintos (un flujo de posiciones de repartidores que debe absorber enormes volúmenes de escritura y un módulo de back-office con una carga modesta), cuando decenas de equipos deben lanzar de forma independiente muchas veces al día, o cuando algunas partes deben escribirse en lenguajes distintos o ejecutarse en lugares distintos por motivos de residencia de datos. Esas necesidades exigen cuantos separados.

Antipatrones habituales: **módulos solo de nombre**, donde las carpetas siguen los dominios pero cualquier cosa importa cualquier cosa; **la base de datos compartida sin reglas**, donde los módulos mantienen su propio código pero hacen joins con las tablas de los demás; **el límite charlatán**, una API de grano tan fino que un caso de uso necesita decenas de llamadas, lo que resultará doloroso si algún día se extrae el módulo; **el núcleo compartido que crece**, un módulo `common` que poco a poco absorbe lógica de negocio y acopla a él todos los módulos; y **los módulos por entidad**, que cortan por entidad de datos (un módulo Cliente, un módulo Pedido, un módulo Producto que todos los casos de uso necesitan a la vez) en lugar de por capacidad de negocio, lo que produce módulos que no pueden hacer nada por sí solos.

Mantener la opción de extraer servicios

Un monolito modular se presenta a veces como un paso intermedio hacia los microservicios y a veces como un destino. Puede ser cualquiera de los dos, y lo importante es que la decisión puede aplazarse hasta el último momento responsable. Si más adelante un módulo necesita escalar por separado o lanzarse de forma independiente, la extracción es mucho más fácil cuando el módulo ya tiene una API explícita, es dueño de sus datos y depende de los demás solo a través de API y eventos. El trabajo pasa a ser en gran parte mecánico: poner una interfaz de red delante de la API, mover el esquema a su propia base de datos, sustituir los eventos en proceso por un intermediario de mensajes (message broker) y sustituir cualquier transacción compartida por una saga explícita o una compensación.

Sam Newman defiende lo mismo en *Monolith to Microservices* (2019): un monolito bien modularizado es el mejor punto de partida para la descomposición, porque la parte difícil, encontrar los límites y hacerlos cumplir, ya está hecha. La Sección 5.8 analiza las señales de que ha llegado el momento de extraer, y la Sección 11.6 trata técnicas de migración incremental como el strangler fig (higuera estranguladora).

Una división más gruesa: los sistemas autocontenidos

Los **sistemas autocontenidos** (SCS) dividen un sistema grande en unas pocas aplicaciones web autónomas, cada una con sus propios datos, lógica e interfaz de usuario y cada una a cargo de un equipo. Suelen ser más grandes y menos numerosos que los microservicios: el sitio de SCS sugiere quizá de 5 a 25 para una tienda en línea, donde los microservicios podrían ser cientos. Cada sistema debe hacer su trabajo principal por sí solo, así que las llamadas a otros quedan fuera del ciclo de solicitud y respuesta, y las interfaces se conectan mediante enlaces en lugar de compartir código. En la plataforma de reparto de comida, los pedidos, el portal de restaurantes y las herramientas de repartidores podrían ser uno cada uno. El estilo encaja con dominios claros con interfaces web, y cada sistema puede ser a su vez un monolito modular. Las pantallas que necesitan datos de muchos dominios son más difíciles.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Qué distingue a un monolito modular de un monolito en capas?**

Su nivel superior está dividido por dominio de negocio en lugar de por capa técnica; cada módulo tiene una API explícita, partes internas privadas y sus propios datos, y puede estar organizado en capas por dentro.

▼ **¿Por qué las API de los módulos deben pasar identificadores y registros simples en lugar de objetos internos del dominio?**

Los objetos internos devueltos permiten a quienes llaman meterse en las partes internas del módulo y los acoplan a su modelo; los datos simples dejan las partes internas libres para cambiar y la API lista para cruzar una red más adelante.

▼ **Un informe necesita datos de los esquemas de pedidos y de pagos. ¿Cuáles son dos opciones que respetan la propiedad de los datos?**

Llamar a las API de ambos módulos y combinar los resultados, o crear un modelo de lectura en un módulo de informes que escuche los eventos de ambos módulos y mantenga su propia copia desnormalizada.

▼ **¿Qué riesgo introduce un listener de eventos en proceso que se ejecuta después de que se confirme la transacción de quien publica?**

Si el proceso se cae después del commit pero antes de que se ejecute el listener, el trabajo del listener se pierde, a menos que el evento se registre de forma duradera, por ejemplo en un registro de publicación de eventos o en una bandeja de salida (outbox), y se reintente.

▼ **Inténtalo: En un monolito modular de reparto de comida, el módulo Reports hace un join directo con las tablas del módulo Payments para un resumen nocturno. Funciona y es rápido. ¿Por qué objetar, con qué lo sustituirías y cómo evitarías que se repita?**

Cada tabla pertenece exactamente a un módulo. El join acopla Reports al esquema privado de Payments, de modo que un cambio en Payments puede romperlo sin aviso, y la extracción posterior se hace más difícil. Sustitúyelo por una llamada a la API pública de Payments o por un modelo de lectura alimentado por los eventos de Payments, aceptando algo de trabajo extra y datos desactualizados. Da a cada módulo su propio esquema y añade un test de arquitectura, con import-linter o con permisos de la base de datos, para que la compilación falle ante cualquier repetición.

FUENTES

- Shopify Engineering: Deconstructing the Monolith (2019): https://shopify.engineering/deconstructing-monolith-designing-software-maximizes-developer-productivity
- Shopify Engineering: Under Deconstruction: The State of Shopify's Monolith (2020): https://shopify.engineering/shopify-monolith
- Shopify Engineering: A Packwerk Retrospective (2024): https://shopify.engineering/a-packwerk-retrospective
- Spring Modulith reference documentation: Fundamentals: https://docs.spring.io/spring-modulith/reference/fundamentals.html
- import-linter documentation: Forbidden contract: https://import-linter.readthedocs.io/en/stable/contract_types/forbidden/
- Redgate: Handling multiple schemas in the same database with Flyway: https://www.red-gate.com/blog/handling-multiple-schemas-in-the-same-database-with-flyway/
- Spring Modulith reference: integration testing application modules: https://docs.spring.io/spring-modulith/reference/testing.html
- Self-Contained Systems: https://scs-architecture.org/
- SCS compared with microservices: https://scs-architecture.org/vs-ms.html

SECCIÓN 5.4 DE 5.8 · INTERMEDIO · 23 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Tuberías y filtros y arquitecturas de pipeline

ANTES DE LEER ESTO

Esta sección se basa en:

- 2.4 Rendimiento y escalabilidad
- 5.1 Estilos y patrones

Términos anteriores que utiliza: Rendimiento (throughput), Idempotente, Prueba de arquitectura, Esquema

LA IDEA EN BREVE

Piensa en un túnel de lavado de vehículos. Cada vehículo pasa por una serie de estaciones: remojo, frotado, enjuague, secado. Cada estación hace un solo trabajo, no sabe nada de las demás y simplemente pasa el vehículo a la siguiente por una cinta transportadora. Puedes añadir una estación de encerado sin cambiar las demás, y si el secado es lento, los vehículos empiezan a hacer cola delante de él. El estilo de tuberías y filtros funciona igual con los datos: cada registro pasa por una cadena de pasos sencillos (los filtros), y una cinta transportadora (la tubería) lo lleva de uno al siguiente.

LO QUE NECESITAS SABER

- El estilo de tuberías y filtros organiza el procesamiento como una cadena de filtros independientes conectados por tuberías de un solo sentido; los filtros no comparten estado y no conocen a sus vecinos, solo el formato de los datos que leen y escriben.
- Hay cuatro tipos de filtro: productores (fuentes), transformadores, testers (que dejan pasar o descartan registros) y consumidores (sumideros).
- Doug McIlroy propuso acoplar programas como mangueras de jardín en un memorándum de 1964; Ken Thompson añadió las tuberías a Unix en 1973, y la filosofía Unix de herramientas pequeñas unidas por tuberías hizo famoso el estilo.
- Los usos típicos son los procesos ETL y los pipelines de datos, las fases de un compilador, el procesamiento de medios y el procesamiento de logs; la misma estructura funciona por lotes (entrada acotada) o en streaming (entrada no acotada).
- El rendimiento (throughput) de un pipeline lo fija su filtro más lento, y un productor rápido desbordará a un consumidor lento a menos que los búferes estén acotados y la back pressure (contrapresión) frene al productor.

> - El estilo puntúa alto en simplicidad, modularidad y reutilización de filtros, y bajo en interactividad y en el manejo de estado compartido o de una recuperación de errores compleja entre pasos.

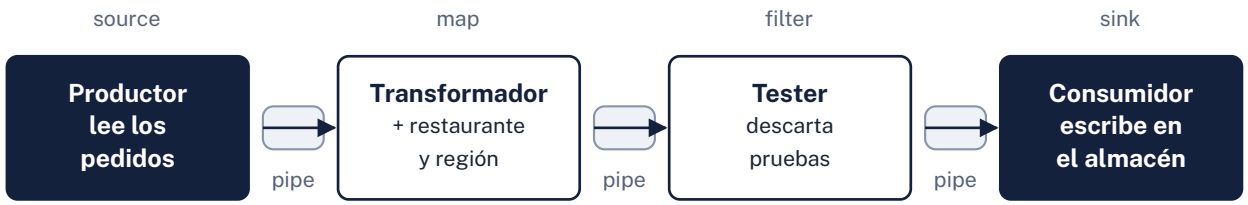
Definición y origen

En el estilo de tuberías y filtros, el procesamiento es una cadena de pasos independientes. Cada filtro lee datos de su entrada, hace un solo trabajo con ellos y escribe el resultado en su salida. Cada tubería lleva los datos en un solo sentido de un filtro al siguiente, normalmente con un búfer entre ambos. La descripción de Garlan y Shaw (1994) enuncia las restricciones que dan al estilo sus propiedades: los filtros deben ser independientes, no deben compartir estado con otros filtros y no deben conocer la identidad de los filtros situados a ambos extremos de sus tuberías. Un filtro solo conoce el formato de lo que lee y de lo que escribe.

El hogar más conocido del estilo es Unix. En un memorándum fechado el 11 de octubre de 1964, Doug McIlroy escribió que debería haber formas de acoplar programas como una manguera de jardín, enroscando otro tramo cuando hay que tratar los datos de otra manera. Ken Thompson implementó la llamada al sistema `pipe` y las tuberías en el shell en la Version 3 de Unix, en 1973, y la notación `|` llegó en la Version 4. La filosofía Unix que surgió de ahí, escribir programas que hagan una sola cosa bien y trabajen juntos a través de una interfaz universal de texto, es el estilo de tuberías y filtros aplicado al conjunto de herramientas de todo un sistema operativo. POSA (1996) catalogó Pipes and Filters como patrón arquitectónico, y *Enterprise Integration Patterns* (2003), de Hohpe y Woolf, lo aplicó a la mensajería entre sistemas.

Richards y Ford llaman a este mismo estilo arquitectura de pipeline y distinguen cuatro tipos de filtro. Un **productor** (o fuente, source) inicia el flujo, leyendo de un archivo, una base de datos o una cola. Un **transformador** cambia cada registro: lo analiza, lo convierte, lo enriquece. Un **tester** comprueba una condición y deja pasar el registro o lo descarta, a veces enviándolo a una de varias salidas. Un **consumidor** (o sumidero, sink) termina el flujo, escribiendo en una base de datos, un archivo o una pantalla.

Filtros y tuberías



Cada filtro solo conoce su formato de entrada y de salida, nunca a sus vecinos

Tuberías: búfer de registros, en un solo sentido

Cuatro tipos de filtro conectados por tuberías; cada filtro se puede desarrollar, probar y sustituir por separado.

Cómo funciona

Las restricciones producen directamente las ventajas. Como un filtro no comparte estado y no conoce a sus vecinos, se puede escribir y probar por separado, con una lista de registros de entrada y una lista esperada de salidas; se puede reutilizar en otros pipelines; y se pueden añadir, quitar o reordenar filtros cambiando solo la forma en que se monta el pipeline. Como los registros fluyen de forma incremental, un filtro puede empezar a trabajar con el primer registro antes de que el filtro anterior haya terminado el último, así que las etapas se ejecutan de forma concurrente y todo el pipeline solo necesita memoria para unos pocos registros a la vez, no para todo el conjunto de datos.

La tubería oculta cómo se mueven los datos. En un shell de Unix es un búfer del sistema operativo entre dos procesos; en Linux admite 64 KiB por defecto, y un escritor que lo llena simplemente se bloquea hasta que el lector se pone al día. En un programa Python puede ser un generador, de modo que cada filtro pide el siguiente registro al anterior. En una plataforma de datos puede ser una cola de mensajes o un topic de un log como Apache Kafka, lo que permite que los filtros se ejecuten en máquinas distintas y sobrevivan a los reinicios. El código de los filtros apenas cambia entre estas opciones; las propiedades operativas cambian muchísimo.

Hay variaciones de la forma estricta. Un pipeline puede abrirse en abanico (fan-out: un tester envía registros a dos consumidores distintos) o converger (fan-in: dos productores alimentan a un transformador). Puede ser lineal, lo más fácil de razonar, o un grafo acíclico dirigido de pasos, como en herramientas de flujos de trabajo como Apache Airflow. En sentido estricto, un filtro con estado interno, como uno que cuenta u ordena, rompe la propiedad incremental,

porque debe ver toda la entrada antes de producir salida; `sort` en un pipeline de Unix es un filtro de ese tipo. Estos filtros están permitidos, pero se convierten en los puntos donde hay que pensar en la memoria, la latencia y los reinicios.

Ejemplos: dónde aparecen los pipelines

ETL y pipelines de datos. Los trabajos de extracción, transformación y carga (ETL) mueven datos de los sistemas operacionales a un almacén de datos: extraer los pedidos de ayer, depurarlos y convertirlos, unirlos con los datos de los restaurantes y cargarlos en tablas de informes. Cada paso es un filtro, y las plataformas de datos modernas (Apache Beam, Spark, modelos de dbt, servicios de flujo de datos en la nube) son, en gran medida, herramientas para construir y ejecutar esos pipelines a escala.

Compiladores. Un compilador clásico es un pipeline de fases: el análisis léxico convierte caracteres en tokens, el análisis sintáctico convierte tokens en un árbol sintáctico, el análisis semántico comprueba tipos y nombres, la optimización reescribe una representación intermedia y la generación de código emite código máquina. Cada fase consume una representación y produce la siguiente, así que las fases se pueden desarrollar y sustituir por separado; el proyecto LLVM, por ejemplo, permite que muchos frontends de lenguajes compartan un optimizador y un conjunto de backends. Los compiladores reales relajan el estilo estricto, porque las fases comparten una tabla de símbolos.

Procesamiento de medios. La transcodificación de video y el procesamiento de audio encadenan decodificadores, escaladores, filtros y codificadores. GStreamer construye aplicaciones multimedia explícitamente como pipelines de elementos enlazados mediante pads, y los grafos de filtros de FFmpeg funcionan de la misma manera.

Procesamiento de logs y eventos. Los enviadores de logs como Logstash y Fluent Bit se configuran como etapas de entrada, filtro y salida: leen líneas de log, las descomponen en campos, descartan el ruido de las comprobaciones de estado (health checks), enmascaran los datos personales y las envían a un índice de búsqueda. Los datos de observabilidad de la plataforma de reparto de comida fluyen de esta manera.

Pipelines de características y de inferencia de aprendizaje automático. La preparación de los datos de entrenamiento, y el preprocesamiento y el posprocesamiento alrededor de un modelo, se construyen habitualmente como pipelines, a los que vuelve el Capítulo 12.

```
# Top ten URLs that returned HTTP 500 in today's access log
grep '" 500 ' access.log \
  | awk '{print $7}' \
  | sort \
  | uniq -c \
  | sort -rn \
  | head -10
```

Un pipeline de Unix: seis pequeñas herramientas, cada una un filtro, encuentran las URL que más a menudo fallaron con HTTP 500 (en el formato habitual de los logs de acceso, el código de estado va después de la línea de solicitud entre comillas).

```
import json

def read_lines(path):                # producer
    with open(path) as f:
        for line in f:
            yield line

def parse(lines):                    # transformer
    for line in lines:
        yield json.loads(line)

def only_failed(events):             # tester
    for e in events:
        if e["status"] == "payment_failed":
            yield e

def count_by_provider(events):       # consumer
    counts = {}
    for e in events:
        counts[e["provider"]] = counts.get(e["provider"], 0) + 1
    return counts

events = only_failed(parse(read_lines("orders.jsonl")))
print(count_by_provider(events))
```

La misma idea con generadores de Python: cada función es un filtro y los registros fluyen de uno en uno, así que el archivo nunca se carga entero en memoria.

Por lotes y en streaming

La misma estructura funciona en dos modos. El procesamiento por lotes trata una entrada acotada, como todos los pedidos de ayer, se ejecuta hasta completarse y se detiene. Es fácil de razonar: si una ejecución falla, corriges el problema y la vuelves a ejecutar, siempre que cada ejecución sustituya su salida en lugar de añadirse a ella. Los pagos nocturnos a los restaurantes, las facturas mensuales y las cargas del almacén de datos son pipelines por lotes típicos. Su punto débil es el retraso: los resultados tienen la antigüedad de la última ejecución.

El procesamiento de flujos (stream processing) trata una entrada no acotada y procesa cada registro poco después de que llegue: posiciones de repartidores, eventos de pago, clics. Los resultados están frescos, pero aparecen nuevas preguntas. Un flujo nunca termina, así que los agregados deben calcularse sobre ventanas (pedidos por restaurante cada cinco minutos); los registros pueden llegar tarde o desordenados; y el pipeline debe funcionar de forma continua, recuperarse de las caídas sin perder registros ni contarlos dos veces, y aguantar cuando la tasa de entrada se dispara. Frameworks como Kafka Streams, Apache Flink y Apache Beam ofrecen ventanas, checkpoints y almacenes de estado para estos problemas. El modelo de Beam trata el procesamiento por lotes como un caso especial del streaming, con una entrada acotada, así que una misma definición de pipeline puede ejecutarse de las dos formas.

Muchas organizaciones usan ambos: un flujo para paneles operativos frescos y aproximados, y un trabajo por lotes que recalcula las cifras exactas durante la noche. La Sección 7.8 analiza el lado de arquitectura de datos de esta elección.

EJEMPLO PRÁCTICO

Ejemplo práctico: encontrar el cuello de botella

La plataforma de reparto de comida carga los eventos de pedido en su almacén de datos a través de cuatro filtros. Medido en un solo worker, cada filtro puede procesar por sí solo el siguiente número de registros por segundo.

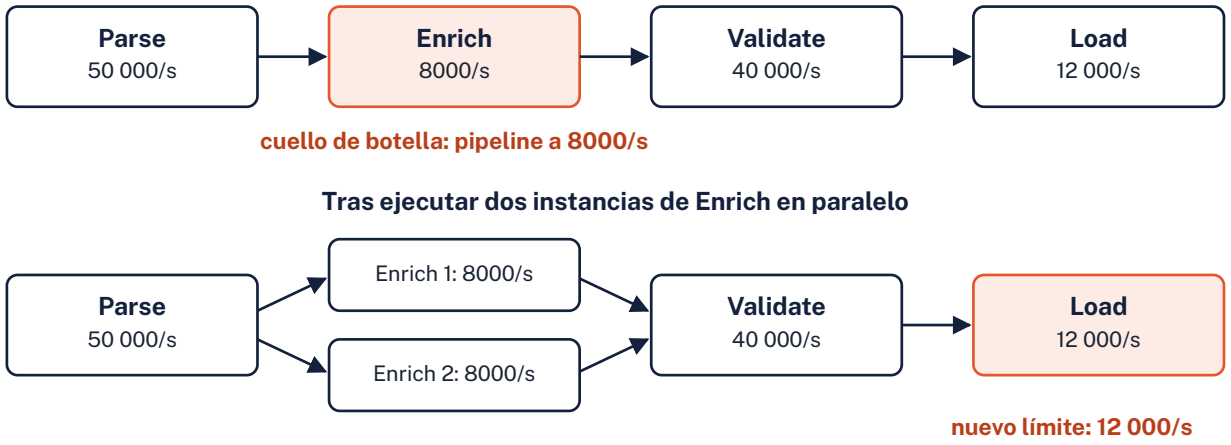
Capacidades	Parse 50 000/s; Enrich (busca el restaurante y la región) 8000/s; Validate 40 000/s; Load 12 000/s.
Rendimiento (throughput) del pipeline	Un pipeline no puede ir más rápido que su filtro más lento: min(50 000, 8000, 40 000, 12 000) = 8000 registros por segundo.
Tiempo para los datos de un día	En un día de mucha actividad, la plataforma genera 1 000 000 de eventos de pedido. 1 000 000 ÷ 8000 = 125 segundos.
Escalar el cuello de botella	Enrich no tiene estado, así que pueden ejecutarse dos instancias en paralelo, cada una con la mitad de los registros: 2 × 8000 = 16 000/s en esa etapa.
Nuevo throughput	min(50 000, 16 000, 40 000, 12 000) = 12 000/s. Ahora el cuello de botella es Load. 1 000 000 ÷ 12 000 ≈ 83 segundos, un ahorro del 33% y no del 50% que sugeriría una lectura ingenua.
Qué hacen los búferes	Sin back pressure, Parse metería registros en la tubería previa a Enrich a 50 000/s mientras Enrich vacía 8000/s, así que el búfer crecería en 42 000 registros por segundo hasta agotar la memoria.

Acelerar cualquier filtro que no sea el cuello de botella no cambia nada; mide primero.

Solo los filtros sin estado se pueden paralelizar de forma tan sencilla. Un filtro que mantiene estado, como los totales acumulados por restaurante, debe particionar su entrada por clave para que cada instancia vea todos los registros de sus claves.

Es el mismo razonamiento que la ley de Amdahl y la utilización de la Sección 2.4: la etapa en serie más lenta fija el límite.

Escalar un pipeline desplaza el cuello de botella



El throughput lo fija el filtro más lento; paralelizarlo ayuda hasta que otro filtro pasa a ser el más lento.

Back pressure y gestión de errores

Cuando un productor es más rápido que un consumidor, algo tiene que ceder. Las opciones son usar un búfer (lo que solo aplaza el problema, a menos que el búfer esté acotado), descartar registros o tomar muestras (aceptable para algunas métricas, nunca para pagos), o aplicar back pressure (contrapresión): hacer que la etapa rápida baje a la velocidad de la lenta. Las tuberías de Unix lo hacen automáticamente, porque una tubería llena bloquea al escritor. Los diseños basados en extracción (pull), como los generadores de Python, lo hacen por construcción, porque no se produce nada hasta que la siguiente etapa lo pide. Los pipelines distribuidos necesitan que se diseñe expresamente: colas acotadas, consumidores que extraen de un log a su propio ritmo, o protocolos como la especificación Reactive Streams de la JVM, en la que un suscriptor indica cuántos registros está listo para recibir. La Sección 9.3 trata a fondo las colas y la back pressure.

Los errores necesitan una política para cada filtro, decidida de antemano. Un **registro mal formado** (un mensaje envenenado, poison message) no debe detener el pipeline: envíalo a una cola de mensajes fallidos (dead-letter queue) o a un archivo de errores con el motivo, cuéntalo y lanza una alerta si la tasa aumenta. Un **fallo transitorio**, como que un servicio de consulta agote su tiempo de espera, se reintenta con backoff. Una **caída** del pipeline necesita un punto de reinicio: los trabajos por lotes se vuelven a ejecutar desde el principio o desde un checkpoint, y los frameworks de streaming guardan en checkpoints su posición y su estado. Como los reinicios implican que algunos registros se procesen dos veces, los consumidores que escriben resultados deben ser idempotentes, por ejemplo haciendo un upsert sobre una clave natural en lugar de insertar a ciegas. Los errores son el punto más débil del estilo: los

filtros no se conocen entre sí a propósito, así que un fallo descubierto en el paso cuatro no puede deshacer fácilmente lo que ya hizo el paso dos, y un pipeline que necesite ese tipo de reversión coordinada probablemente usa el estilo equivocado.

Características de tuberías y filtros (despliegue único)

Característica	Valoración	Motivo
Sencillez	Alto	Cada filtro hace una sola cosa; el flujo es fácil de dibujar, leer y explicar.
Modularidad y reutilización	Alto	Los filtros no comparten estado y solo conocen formatos de datos, así que se pueden reutilizar, reordenar y sustituir.
Capacidad de prueba (testabilidad)	Alto	Un filtro se prueba con una entrada de ejemplo y una salida esperada, sin sus vecinos.
Costo total	Alta (barato)	Un solo programa o trabajo; poca infraestructura, salvo si es distribuido.
Rendimiento (throughput)	De media a alta	Las etapas se ejecutan de forma concurrente y las que no tienen estado se paralelizan, pero el filtro más lento lo limita todo.
Latencia e interactividad	Bajo	Pensado para flujos de datos, no para conversaciones de solicitud y respuesta con usuarios; cada etapa añade retraso.
Desplegabilidad	De baja a media	Como un solo programa o trabajo, cualquier cambio vuelve a desplegar todo el pipeline; los filtros distribuidos sobre colas se pueden lanzar por separado.
Tolerancia a fallos	De baja a media	En un solo proceso, una caída detiene todo el pipeline; hay que añadir checkpoints, reintentos y colas de mensajes fallidos.
Estado compartido y transacciones	Bajo	Los filtros son deliberadamente independientes y sin estado; la consistencia o la reversión entre pasos es complicada.
Escalabilidad	De baja a media en un solo proceso	Mayor cuando los filtros se distribuyen sobre colas, a costa de la operación y de las garantías de entrega.

SI ESTO ES NUEVO PARA TI

Paso a paso: cuando un pipeline falla a medio camino

El punto en el que más gente tropieza aquí es qué pasa cuando un pipeline falla a mitad del proceso. Los filtros no saben nada unos de otros, así que no hay una forma fácil de deshacer el trabajo ya hecho, y un reinicio procesa algunos registros dos veces. Sigue la carga de un día en el almacén de datos a través de tres tipos de fallo.

- Una vez al día, la plataforma de reparto de comida carga en el almacén de datos los registros de entregas completadas de sus repartidores a través de cuatro filtros: Parse, Enrich (que busca la zona de cada repartidor), Validate y Load.
- Un registro está mal formado y Parse no puede leerlo. En lugar de detener todo el trabajo por un registro defectuoso, Parse lo aparta en una cola de mensajes fallidos (dead-letter queue), anota el motivo, lo cuenta y sigue adelante.
- Después, el servicio de consulta del que depende Enrich deja de responder durante unos segundos. Es un fallo transitorio, así que Enrich reintenta con backoff, esperando un poco más cada vez.
- Entonces el trabajo se cae después de que Load haya escrito 300 000 de los 450 000 registros del día. Esas filas se quedan: los filtros no se coordinan, así que nada en el pipeline puede revertirlas fácilmente.
- La recuperación más sencilla es volver a ejecutar el trabajo desde el principio. Pero si Load simplemente inserta filas, las primeras 300 000 entregas aparecen ahora dos veces, y cada informe construido sobre ellas las cuenta dos veces.
- Así que Load se hace idempotente: hace un upsert sobre el identificador propio de cada entrega, añadiendo una fila si la entrega es nueva y sustituyéndola si ya existe. Volver a ejecutar el trabajo deja entonces exactamente una fila por entrega.
- **La lección:** decide la política de errores de cada filtro antes del primer fallo. Aparta los registros defectuosos, reintenta los fallos breves y haz que el paso final se pueda repetir sin riesgo, porque los reinicios procesarán algunos registros dos veces.

Cuándo usarlo, cuándo no y antipatrones

Usa tuberías y filtros cuando el problema sea por naturaleza un flujo de datos en un solo sentido a través de transformaciones distintas: ingesta, conversión, enriquecimiento, validación, agregación, exportación. Es la opción por defecto en ingeniería de datos, en el procesamiento de archivos y medios, y en cualquier trabajo en el que los pasos cambien más a menudo que el flujo general. También aparece dentro de otros estilos: un trabajo nocturno de informes dentro de un monolito modular, o un servicio de procesamiento de flujos dentro de un sistema dirigido por eventos.

Evítalo como estructura principal de aplicaciones interactivas, en las que los usuarios esperan respuestas inmediatas y conversacionales y el trabajo no es un flujo; para lógica en la que muchos pasos necesitan compartir y actualizar el mismo estado; y para procesos que necesitan una reversión coordinada entre pasos, que requieren una transacción o una saga.

Antipatrones habituales: **el filtro dios**, un paso que analiza, enriquece, valida y escribe, con lo que se pierden las ventajas del estilo; **el estado compartido oculto**, filtros que se comunican a través de una tabla de base de datos compartida o una variable global en lugar de la tubería, lo que reintroduce el acoplamiento que el estilo elimina; **la deriva de formato**, un productor que cambia sin avisar su formato de salida y rompe todos los filtros posteriores, algo que se evita con esquemas explícitos y comprobaciones de esquema (consulta la Sección 7.7); **los búferes no acotados**, que convierten a un consumidor lento en una caída por falta de memoria; y **los sumideros no idempotentes**, que cuentan dos veces después de cada reintento.

Hacer cumplir el estilo en el código

Las restricciones del estilo se pueden hacer explícitas. Da a cada filtro la misma firma sencilla (entra un iterable de registros, sale un iterable), para que la composición sea mecánica. Define el formato de los registros en cada tubería como un esquema (una dataclass, un JSON Schema, una definición de Avro o de Protocol Buffers) y valídalo en el límite. Mantén los filtros libres de estado mutable compartido: una prueba de arquitectura puede prohibir que los módulos de filtros importen la capa de base de datos o se importen entre sí, de modo que la única forma de que los datos pasen entre ellos sea la tubería. Prueba cada filtro con pequeños datos de prueba (fixtures) de entrada y salida, y prueba el pipeline montado de extremo a extremo con una muestra representativa, que incluya registros mal formados, para demostrar que la política de errores funciona.

En los pipelines distribuidos, monitoriza el throughput por etapa, la profundidad de las colas y el retraso de los consumidores (consumer lag): una cola que crece delante de una etapa es la señal visible de un cuello de botella o de un filtro que falla, y es la primera métrica sobre la que alertar.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Qué tres restricciones impusieron Garlan y Shaw a los filtros?**

Los filtros deben ser entidades independientes, no deben compartir estado con otros filtros y no deben conocer la identidad de los filtros situados antes o después de ellos.

▼ **¿Por qué el comando sort de Unix rompe la propiedad incremental de un pipeline?**

Debe leer toda su entrada antes de poder producir la primera línea, así que retiene todo el conjunto de datos y retrasa todo lo que viene después hasta que termina su entrada.

▼ **Un pipeline tiene etapas que pueden procesar 20 000, 5000 y 15 000 registros por segundo. ¿Cuál es su throughput y qué ocurre si se duplica la etapa intermedia?**

5000 por segundo, fijado por la etapa más lenta. Duplicarla hasta 10 000 hace que el pipeline funcione a 10 000 por segundo, todavía por debajo de las otras dos etapas.

▼ **¿Por qué el consumidor final de un pipeline que se puede reiniciar debe ser idempotente?**

Porque un reinicio o un reintento puede entregar algunos registros dos veces; una escritura idempotente, como un upsert sobre una clave natural, da el mismo resultado por muchas veces que llegue un registro.

▼ **Inténtalo: Un pipeline de eventos de pedidos tiene los filtros Parse 40 000/s, Enrich 6000/s, Validate 30 000/s y Load 10 000/s, y en el pico llegan 9000 registros/s. ¿Qué ocurre? ¿Cuántos workers de Enrich lo solucionan y qué limita entonces el rendimiento (throughput)?**

El rendimiento (throughput) lo fija el filtro más lento, Enrich a 6000/s, de modo que en el pico crece un backlog de 3000 registros por segundo. Dos workers de Enrich dan 12 000/s, suponiendo que escale limpiamente. Load pasa entonces a ser el límite con 10 000/s, lo que sigue superando los 9000/s, pero deja alrededor de un 10% de margen. Mantén los búferes acotados, con back pressure (contrapresión), para que una etapa lenta no pueda agotar la memoria.

FUENTES

- Doug McIlroy's 1964 memo on coupling programs (cat-v.org): https://doc.cat-v.org/unix/pipes/
- Enterprise Integration Patterns: Pipes and Filters: https://www.enterpriseintegrationpatterns.com/patterns/messaging/PipesAndFilters.html
- Microsoft Azure Architecture Center: Pipes and Filters pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters
- Python documentation: Functional Programming HOWTO (generators): https://docs.python.org/3/howto/functional.html

SECCIÓN 5.5 DE 5.8 · INTERMEDIO · 26 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Arquitecturas de microkernel y de plug-ins

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.6 API e integración
- 3.3 Ocultación de información
- 5.1 Estilos y patrones

Términos anteriores que utiliza: Unidad de despliegue, Cambio incompatible (breaking change), Interfaz

LA IDEA EN BREVE

Piensa en una consola de videojuegos. La consola en sí hace unas pocas cosas generales: arranca, muestra un menú, lee los mandos y dibuja imágenes. Cada juego es un cartucho que se inserta mediante un conector estándar. Puedes añadir juegos nuevos durante años sin abrir la consola, y un juego mal hecho no puede volver a cablearla. Una arquitectura de microkernel construye el software de la misma manera: un núcleo pequeño y estable hace el trabajo general, y los plug-ins añaden las partes que varían, mediante un conector acordado llamado contrato.

LO QUE NECESITAS SABER

- Una arquitectura de microkernel (de plug-ins) tiene un sistema central mínimo más plug-ins independientes que se conectan solo mediante contratos publicados y que se localizan a través de un registro.
- Se adapta a productos con un flujo de trabajo estable y muchas variantes de pasos concretos, como las normas fiscales de cada país, los métodos de pago, los importadores de archivos o los lenguajes de un editor.
- Los plug-ins deben depender solo del contrato, nunca unos de otros ni de las partes internas del núcleo, y el núcleo nunca debe contener casos especiales para plug-ins concretos.
- El aislamiento es un compromiso: los plug-ins en proceso son rápidos pero corren la misma suerte que el anfitrión; los procesos separados, los sandboxes o las llamadas remotas protegen el núcleo a cambio de un costo.

- Versiona el contrato por separado, comprueba la compatibilidad cuando se cargan los plug-ins y prefiere los cambios aditivos; la ley de Hyrum implica que todo lo que los plug-ins pueden observar pasa a formar parte del contrato.
- Los microkernels de aplicación buscan la extensibilidad; los microkernels de sistema operativo, como L4 o QNX, buscan el aislamiento y un núcleo de confianza pequeño.

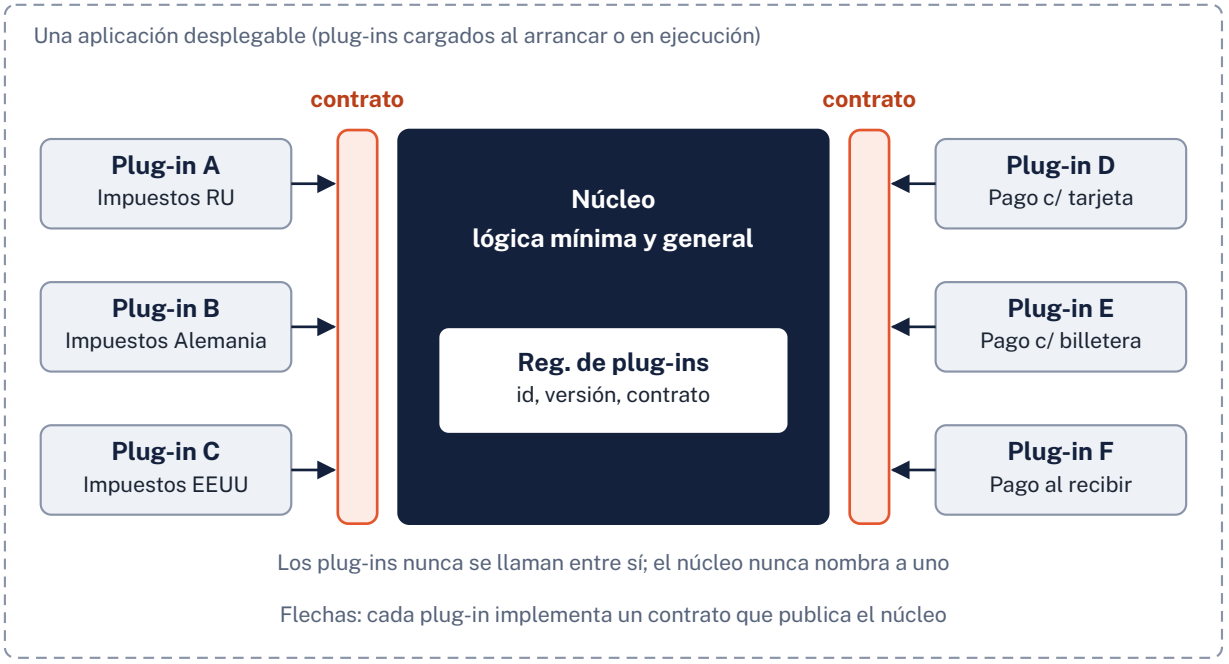
Definición y origen

Una arquitectura de microkernel, también llamada arquitectura de plug-ins, divide una aplicación en dos tipos de parte: un sistema central (core system) que contiene la funcionalidad mínima y general necesaria para funcionar, y plug-ins que añaden o cambian comportamiento sin que se modifique el núcleo. Los plug-ins se conectan al núcleo solo mediante contratos publicados, y el núcleo los localiza a través de un registro. Mark Richards y Neal Ford la catalogan como uno de los estilos arquitectónicos clásicos en *Fundamentals of Software Architecture*, y encaja con la descripción de estilo arquitectónico dada en la Sección 5.1: un conjunto con nombre de restricciones sobre los elementos (núcleo, plug-ins) y los conectores (contratos) de un sistema.

La idea nació en el software de producto: herramientas que instalan muchos clientes, cada uno de los cuales quiere algo ligeramente distinto. En lugar de mantener una versión separada para cada cliente, el proveedor distribuye un solo núcleo y deja que los clientes, los socios o los propios equipos del proveedor añadan plug-ins. Los entornos de desarrollo integrados, los navegadores web, los editores gráficos y los paquetes de contabilidad funcionan así. El nombre se toma de los sistemas operativos, donde un microkernel es un núcleo muy pequeño con todo lo demás ejecutándose fuera de él; las dos ideas están relacionadas, pero no deben confundirse, como se explica más adelante.

Una aplicación de microkernel suele ser una única unidad de despliegue, así que pertenece a los estilos monolíticos de este capítulo: una unidad de despliegue, un proceso, un conjunto de inquietudes operativas. Los plug-ins pueden instalarse por separado, pero en tiempo de ejecución viven dentro del núcleo o justo a su lado.

La forma de un sistema de microkernel



El sistema central es dueño del flujo de trabajo general y de un registro; cada plug-in implementa un contrato que publica el núcleo, y los plug-ins nunca dependen unos de otros.

Cómo funciona: núcleo, plug-ins, registro y contratos

El sistema central contiene lo que necesita toda instalación: el flujo de trabajo principal, servicios compartidos como el logging o la configuración, y los puntos de extensión donde se acoplan los plug-ins. Richards y Ford lo describen como el camino feliz a través de la aplicación, con poco o ningún procesamiento personalizado; en el proceso de pago de una plataforma de reparto de comida, eso es poner precio a un carrito, cobrar el pago y confirmar el pedido. Un buen núcleo es deliberadamente aburrido. Cada caso especial que varía según el cliente, el país o la línea de producto es candidato a salir de él.

Los plug-ins son módulos independientes y autocontenidos que contienen procesamiento especializado. Deben depender solo del contrato, no unos de otros ni de las clases internas del núcleo. Cuando un plug-in necesita algo que proporciona otro, esa necesidad debe cubrirla el núcleo (como un servicio que expone a través del contrato) o bien hay que fusionar los dos plug-ins; las dependencias de plug-in a plug-in convierten sin hacer ruido una estrella limpia en una maraña.

El registro anota qué plug-ins están disponibles, cómo se llama cada uno, qué versión es, qué contrato implementa y cómo llegar a él. Puede ser tan sencillo como un diccionario rellenado al arrancar, un directorio en el que el núcleo busca manifiestos, entradas en los metadatos de un paquete (como con los puntos de entrada de Python), una tabla de base de datos o un servicio

remoto que lista plug-ins desplegados en otro lugar. El registro es lo que permite al núcleo decir "dame las normas fiscales de Alemania" sin saber que existe una clase llamada `GermanTax`.

Los contratos definen qué debe proporcionar un plug-in y de qué puede depender: una interfaz con firmas de métodos, las estructuras de datos que la cruzan, el comportamiento ante errores y una versión. En Eclipse, un contrato es un punto de extensión declarado en el manifiesto de un plug-in con un esquema; en Visual Studio Code es la API de extensiones más los puntos de contribución enumerados en el `package.json` de una extensión; en tu propio código suele ser una interfaz (o un `Protocol` de Python) en un pequeño paquete propio. Cuando el plug-in procede de otra organización y no puede seguir tu contrato, un adaptador traduce entre ambos, exactamente como en la arquitectura hexagonal.

Los plug-ins pueden conectarse de tres maneras. Las llamadas **punto a punto, en proceso** son lo habitual: el núcleo llama a un método del objeto plug-in. Los plug-ins **en un proceso separado** se ejecutan en su propio proceso y hablan con el núcleo mediante mensajes entre procesos, lo que protege al núcleo de los bloqueos. Los plug-ins **remotos** están detrás de una interfaz HTTP o de mensajería; esto permite plug-ins en otros lenguajes o de otros equipos, pero añade latencia de red y modos de fallo, y el sistema empieza a parecerse a los estilos distribuidos del Capítulo 6.

SI ESTO ES NUEVO PARA TI

Paso a paso: un núcleo que nunca nombra a sus plug-ins

La idea más difícil de esta sección es cómo un núcleo puede usar plug-ins sin saber cuáles existen. El núcleo depende solo del contrato, y un registro le dice en tiempo de ejecución qué plug-ins están instalados. Sigue el proceso de pago de la plataforma de reparto de comida mientras añade un país.

- Hoy el código del proceso de pago calcula él mismo los impuestos, en ramas como: si el país es GB, aplica las normas del Reino Unido.
- Lanzarse en Alemania implica abrir el código del proceso de pago, añadir una rama para Alemania y luego volver a probar y lanzar todo el proceso de pago. El núcleo conoce cada país por su nombre y crece con cada lanzamiento.
- En el diseño de microkernel, el núcleo publica en cambio un contrato: una pequeña interfaz que dice lo que deben proporcionar todas las normas fiscales, una forma de calcular el impuesto sobre un carrito con precio, y exactamente qué datos la cruzan.
- El equipo del país alemán escribe un plug-in que implementa el contrato e indica a qué país sirve, DE. Las normas del Reino Unido salen del proceso de pago a un plug-in GB de la misma manera, así que el proceso de pago ya no nombra ningún país.
- Al arrancar, el registro encuentra los plug-ins instalados y anota a qué país sirve cada uno. Cuando un cliente alemán finaliza la compra, el núcleo pide al registro las normas fiscales de DE y las llama a través del contrato, sin saber qué fragmento de código responde.
- Añadir más adelante Estados Unidos funciona igual: un plug-in nuevo con sus propias pruebas, que el registro encuentra. El código del proceso de pago en sí no cambia.
- **La lección:** el núcleo conoce el contrato y el registro, nunca los plug-ins. Cualquier línea del núcleo que nombre un plug-in concreto es señal de que al contrato le falta algo.

Ejemplos reales

Eclipse. El IDE Eclipse es el caso de manual. Su núcleo es poco más que un banco de trabajo y un entorno de ejecución de plug-ins (construido sobre Equinox, una implementación de la especificación del framework OSGi). Casi todo lo que ve un desarrollador, desde las herramientas de Java hasta la integración con el control de versiones, es un plug-in. Cada plug-in tiene un manifiesto que declara lo que necesita, qué puntos de extensión ofrece a otros y a qué extensiones se conecta. Los plug-ins se activan de forma diferida, solo cuando algo los necesita de verdad, para que cientos de plug-ins instalados no ralenticen el arranque.

Visual Studio Code. Las extensiones declaran eventos de activación (abrir un archivo Markdown, ejecutar un comando) para cargarse solo cuando hace falta, y aportan comandos, lenguajes, temas y vistas a través de su manifiesto. Las extensiones se ejecutan en un proceso anfitrión de extensiones separado, de modo que una extensión que se comporta mal no puede congelar la interfaz de usuario del editor. Es un buen ejemplo de elegir el aislamiento por procesos para un ecosistema de plug-ins que incluye miles de autores externos.

Navegadores web. Las extensiones de navegador usan un conjunto estándar de API de extensiones y deben declarar en un manifiesto los permisos que necesitan (qué sitios pueden leer, si pueden ver las pestañas o las descargas). El navegador hace cumplir esos permisos, porque por defecto no se puede confiar en los autores de extensiones.

Motores de reglas de negocio. El sistema de siniestros de una aseguradora puede mantener el flujo de trabajo de los siniestros en el núcleo y poner las normas de cada jurisdicción (qué está cubierto, qué documentos se necesitan, los plazos legales) en plug-ins. Los motores fiscales, los sistemas de nóminas y las plataformas de comercio electrónico con plug-ins de métodos de pago siguen el mismo patrón: un proceso estable, con normas o proveedores que varían según el país, el producto o el cliente.

Herramientas para desarrolladores. El framework de pruebas pytest está construido en torno a la biblioteca de plug-ins `pluggy`; muchas de sus propias funcionalidades están implementadas como plug-ins que usan los mismos hooks que usan terceros. Las herramientas de compilación, los linters y los generadores de sitios estáticos suelen hacer lo mismo.

Microkernel de aplicación frente a microkernel de sistema operativo

Aspecto	Microkernel de sistema operativo	Arquitectura de microkernel de aplicación (de plug-ins)
Qué es mínimo	El kernel: espacios de direcciones, hilos, planificación y comunicación entre procesos	El sistema central: el flujo de trabajo general de una aplicación
Qué queda fuera del núcleo	Controladores de dispositivos, sistemas de archivos y pilas de red, ejecutados como servidores separados en espacio de usuario	Comportamiento de negocio o funcionalidades variables, como normas fiscales, métodos de pago o lenguajes de un editor
Objetivos principales	Fiabilidad, seguridad y una base de cómputo de confianza pequeña	Extensibilidad, personalización y un ecosistema de terceros
Cómo se comunican las partes	Paso de mensajes entre procesos aislados	Normalmente, llamadas en proceso a través de una interfaz; a veces, un proceso separado o una llamada remota
Ejemplos	Mach, la familia L4 (incluido seL4, verificado formalmente), QNX, MINIX 3	Eclipse, Visual Studio Code, navegadores web, pytest, motores de reglas de seguros y fiscales

NOTA

No confundas los dos microkernels

Los microkernels de sistema operativo, como Mach y la familia L4, sacan los controladores de dispositivos y los sistemas de archivos del kernel a procesos separados en espacio de usuario, para que un controlador que falla no pueda tumbar toda la máquina. Linux, en cambio, es un kernel monolítico con módulos cargables. Andrew Tanenbaum y Linus Torvalds discutieron los dos diseños en un célebre debate en Usenet en 1992. Un microkernel de aplicación toma prestada la forma (núcleo pequeño, todo lo demás fuera), pero su objetivo habitual es la extensibilidad, no el aislamiento, y sus plug-ins normalmente se ejecutan en el mismo proceso que el núcleo.

Aislamiento, versionado y compatibilidad de los plug-ins

Tres preguntas deciden si una arquitectura de plug-ins se mantiene sana con los años: cuán aislados están los plug-ins, cómo evoluciona el contrato y cómo se detecta la incompatibilidad.

Aislamiento. Los plug-ins en proceso comparten la memoria, los hilos y los permisos del núcleo. Eso es rápido y sencillo, pero un plug-in con un bucle infinito, una fuga de memoria o un agujero de seguridad afecta a todo. Las opciones, de menor a mayor aislamiento y costo, son: en proceso con llamadas defensivas (timeouts, captura de excepciones por plug-in, límites de recursos); un proceso o worker separado por plug-in o por grupo de plug-ins; un sandbox, como un entorno de ejecución de WebAssembly o un intérprete restringido, que limita lo que puede tocar el código del plug-in; y un servicio remoto. Elige preguntándote quién escribe los plug-ins. Las normas fiscales de tu propio equipo pueden ejecutarse en proceso; el código arbitrario de un marketplace público no debería.

Versionado. El contrato es una API, y se aplican las reglas de la Sección 0.6 sobre los cambios incompatibles. Versiona el contrato por separado del núcleo, normalmente con versionado semántico: una nueva versión principal para los cambios incompatibles y una versión menor para las incorporaciones compatibles hacia atrás. Deja que cada plug-in declare para qué versión del contrato se construyó, y que el núcleo declare qué versiones admite. El cambio aditivo (campos opcionales nuevos, hooks opcionales nuevos) mantiene funcionando los plug-ins antiguos; cuando un cambio incompatible es inevitable, admite la versión principal antigua y la nueva en paralelo durante un periodo de obsolescencia anunciado.

Comprobaciones de compatibilidad. Comprueba la compatibilidad cuando se cargan los plug-ins, no cuando un cliente los activa por primera vez. Un registro que rechaza un plug-in incompatible al arrancar, con un mensaje de log claro, convierte un error en tiempo de ejecución en un fallo de despliegue. Merece la pena recordar aquí la observación de Hyrum Wright, conocida como ley de Hyrum: con suficientes usuarios, alguien dependerá de cada comportamiento observable de una interfaz, diga lo que diga la documentación. Mantén el contrato pequeño, deja las partes internas del núcleo fuera de alcance (Eclipse marca como `internal` los paquetes que no son API por esta razón) y trata todo lo que ven los plug-ins como parte del contrato.

Características del estilo microkernel

Característica	Valoración	Motivo
Extensibilidad y modificabilidad (en las direcciones previstas)	Alto	El comportamiento nuevo llega como un plug-in nuevo; ni el núcleo ni los demás plug-ins cambian.
Sencillez	Media	La idea es sencilla, pero los contratos, un registro, la carga y las comprobaciones de versión son maquinaria real.
Capacidad de prueba (testabilidad)	De media a alta	Cada plug-in puede probarse por separado contra el contrato; el núcleo puede probarse con plug-ins falsos. Las combinaciones siguen necesitando pruebas.
Desplegabilidad	Media	A menudo se pueden añadir plug-ins sin volver a desplegar el núcleo; un cambio de contrato obliga a lanzamientos coordinados.
Rendimiento	De media a alta	Los plug-ins en proceso son llamadas a funciones normales; el aislamiento por proceso o sandbox añade sobrecarga.
Escalabilidad y elasticidad	Bajo	Suele ser una sola unidad de despliegue, así que escala en bloque, como cualquier monolito.
Tolerancia a fallos	De baja a media	Un plug-in en proceso puede hacer caer o ralentizar al anfitrión; el aislamiento por procesos lo mejora a cambio de un costo.
Costo total	Alta (barato)	Una sola base de código y un solo entorno de ejecución que operar; el costo principal es diseñar y hacer evolucionar los contratos.

Cuándo usarlo y cuándo no

Buenos casos. Software de producto que muchos clientes instalan y personalizan. Sistemas con un flujo de trabajo estable pero muchas variantes de un paso: impuestos y nóminas por país, normas de seguros por jurisdicción, métodos de pago, importadores de formatos de archivo, empresas de transporte. Plataformas que quieren un ecosistema de terceros. Sistemas en los que un regulador o un cliente cambia las normas a menudo y quieres que esos cambios sean lanzamientos pequeños y probados por separado. El estilo también funciona dentro de un sistema mayor: un servicio de una arquitectura distribuida puede ser en sí mismo un microkernel.

Malos casos. Sistemas en los que aún no sabes qué varía. Los puntos de extensión diseñados de antemano para necesidades imaginadas son generalidad especulativa: añaden maquinaria, frenan al equipo y suelen estar en el lugar equivocado cuando llega la necesidad real (consulta el análisis de la reversibilidad en la Sección 1.6). Sistemas que necesitan una escalabilidad o un

aislamiento de fallos muy altos, ya que todo escala y falla como una sola unidad. Sistemas en los que los plug-ins tendrían que coordinarse estrechamente entre sí, lo que anula el principal beneficio del estilo.

Una regla práctica: espera a haber construido el mismo tipo de variación dos o tres veces y, entonces, extrae el contrato de los casos reales. Un contrato derivado de tres implementaciones fiscales que funcionan será mucho mejor que uno adivinado antes de la primera.

ATENCIÓN

Antipatrones comunes

- **El núcleo que lo sabe todo.** El núcleo contiene `if plugin_name == "stripe"` o casos especiales para plug-ins concretos. Cada uno es señal de que al contrato le falta algo.
- **Núcleo obeso.** Las normas variables vuelven a colarse en el núcleo porque es más rápido que ampliar el contrato, hasta que el núcleo vuelve a ser el viejo monolito.
- **Marañas de plug-ins.** Los plug-ins se importan entre sí o comparten estado oculto, así que ya no pueden instalarse, probarse ni eliminarse de forma independiente.
- **Contrato con fugas.** El contrato pasa a los plug-ins objetos de dominio del núcleo o sesiones de base de datos, así que cada cambio interno se convierte en un cambio incompatible para todos los plug-ins.
- **Contrato sin versión.** Nadie sabe qué plug-ins se romperán cuando cambie el núcleo, así que las actualizaciones se convierten en acontecimientos temidos y de todo a la vez.
- **Un plug-in por cliente.** Los plug-ins se convierten en una forma de mantener cientos de bifurcaciones por cliente. Las pequeñas diferencias entre clientes deben expresarse con configuración, no con código.

EJEMPLO PRÁCTICO

Ejemplo práctico: diseñar un contrato de plug-in para normas fiscales de cada país

La plataforma de reparto de comida que se usa a lo largo de este curso se lanza en Alemania y Estados Unidos después del Reino Unido. Cada país grava de forma distinta la comida, el alcohol y las tarifas de envío, y el impuesto sobre las ventas de EE. UU. además varía según el estado y la ciudad. El código del proceso de pago ya contiene ramas `if country == "GB"`. El arquitecto propone un microkernel: el proceso de pago es el núcleo, y las normas de cada país son un plug-in. Estas son las decisiones, en el orden en que se tomaron.

1. Nombrar el punto de variación	Solo el cálculo del impuesto sobre un carrito con precio varía según el país. Los precios, los descuentos, el pago y los recibos se quedan en el núcleo. Un punto de variación estrecho mantiene pequeño el contrato.
2. Decidir qué cruza el contrato	Entradas: líneas del carrito con una categoría neutral (`hot-food`, `alcohol`, `delivery-fee`), importes netos como `Decimal` y una región opcional. Salida: el impuesto total y un desglose etiquetado para el recibo. No se pasan objetos del núcleo (pedidos, clientes, sesiones de base de datos), así que los plug-ins no pueden depender de las partes internas del núcleo.
3. Hacer del contrato un paquete separado	`tax_contract` contiene solo clases de datos y el protocolo `TaxRules`, versionado con versionado semántico. Tanto el núcleo como los plug-ins dependen de él; ninguno depende del otro.
4. Definir la compatibilidad	Un plug-in declara la versión del contrato que necesita, por ejemplo (2, 1). El núcleo lo acepta si la versión principal coincide y la versión menor del núcleo es al menos igual de alta. Añadir un campo opcional es un cambio menor; renombrar o eliminar uno es un cambio principal.
5. Elegir el descubrimiento	Los plug-ins son paquetes normales de Python que se registran en el grupo de puntos de entrada `fooddelivery.tax_rules`. Basta con instalar un paquete; ningún código del núcleo enumera los países.
6. Decidir el comportamiento ante fallos	Si no hay ningún plug-in registrado para un país, el proceso de pago rechaza el pedido con un error claro en lugar de cobrar un impuesto cero. Un plug-in que lanza una excepción hace fallar solo ese proceso de pago, y el error se notifica con el nombre y la versión del plug-in.
7. Decidir el aislamiento	Los plug-ins los escriben los propios equipos de país de la empresa y son de confianza, así que se ejecutan en proceso por velocidad. Si se permitiera que proveedores fiscales externos suministraran plug-ins, el diseño los pasaría detrás de una llamada remota con un timeout.

8. Especificar las pruebas	Un conjunto compartido de pruebas de contrato (redondeo, carritos vacíos, líneas de reembolso negativas, categorías desconocidas) se ejecuta contra cada plug-in en su propio pipeline. Cada equipo de país añade sus propios casos a partir de las orientaciones fiscales publicadas.

El núcleo ya no cambia cuando se añade un país, así que el lanzamiento de un cuarto país es un paquete nuevo más sus pruebas, no un cambio en el proceso de pago.

La decisión más difícil es la del paso 2. Una lista de categorías demasiado gruesa obliga a los plug-ins a adivinar; una demasiado detallada acopla el contrato a la ley de un solo país. Cuenta con revisarla, y planifica la migración de versión principal cuando lo hagas.

El registro rechaza los plug-ins incompatibles al arrancar, así que una discrepancia de versión aparece en una comprobación del despliegue, no como un impuesto erróneo en el recibo de un cliente.

```
# package: tax_contract  (version 2.1.0)
from dataclasses import dataclass
from decimal import Decimal
from typing import Protocol

CONTRACT_VERSION = (2, 1)

@dataclass(frozen=True)
class TaxLine:
    sku: str
    category: str      # "hot-food", "cold-food", "alcohol", "delivery-fee"
    net: Decimal       # never float for money

@dataclass(frozen=True)
class TaxResult:
    total_tax: Decimal
    breakdown: dict[str, Decimal]   # label shown on the receipt

class TaxRules(Protocol):
    country: str                    # ISO 3166-1 alpha-2, e.g. "DE"
    requires_contract: tuple[int, int]
    def calculate(self, lines: list[TaxLine], region: str | None) → TaxResult: ...
```

El paquete del contrato, versionado y publicado por separado tanto del núcleo como de los plug-ins.

```
from importlib.metadata import entry_points
from tax_contract import CONTRACT_VERSION

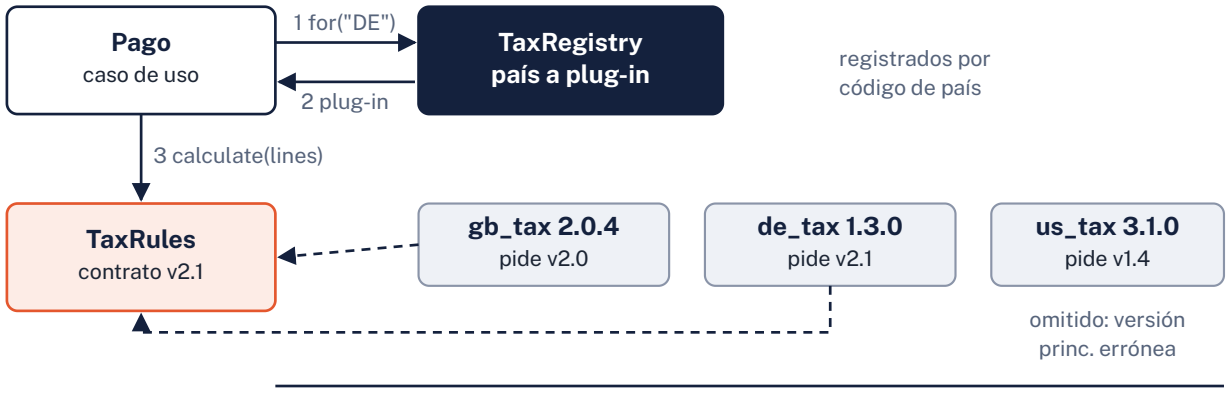
class UnsupportedCountry(Exception): pass

class TaxRegistry:
    def __init__(self, log):
        self._by_country = {}
        for ep in entry_points(group="fooddelivery.tax_rules"):
            plugin = ep.load()()
            major, minor = plugin.requires_contract
            if major ≠ CONTRACT_VERSION[0] or minor > CONTRACT_VERSION[1]:
                log.warning("skipping %s: needs contract %s", ep.name,
plugin.requires_contract)
                continue
            self._by_country[plugin.country] = plugin

    def for_country(self, country):
        if country not in self._by_country:
            raise UnsupportedCountry(country)   # fail loudly, never guess a rate
        return self._by_country[country]
```

El registro en el núcleo: descubre los plug-ins instalados, comprueba la compatibilidad del contrato y los busca por país.

Los plug-ins fiscales en tiempo de ejecución



Carga: al arrancar, el registro lee los paquetes instalados (puntos de entrada), comprueba la versión de contrato que requiere cada plug-in y lo registra por país; los incompatibles se anotan en el log y se omiten sin hacer caer el pago.

El proceso de pago pide al registro el plug-in registrado para un país y lo llama a través del contrato; un plug-in construido para una versión de contrato incompatible se omite al arrancar.

Hacer cumplir el estilo en el código

Una arquitectura de plug-ins solo sigue siéndolo si sus reglas se comprueban automáticamente. La estructura que funciona en la mayoría de los lenguajes tiene tres tipos de paquete: un pequeño paquete de **contrato**, solo con interfaces y tipos de datos; el **núcleo**, que depende del contrato; y un paquete por cada **plug-in**, que depende del contrato y de nada más de lo tuyo.

Después, haz cumplir las reglas de dependencia con herramientas como las que se tratan en la Sección 3.4: en Python, import-linter puede prohibir que el núcleo importe cualquier paquete de plug-in y que los plug-ins se importen entre sí o importen el núcleo; en Java, las pruebas de ArchUnit o los sistemas de módulos (JPMS, OSGi) pueden hacer lo mismo. Haz fallar la compilación cuando se incumpla una regla.

Añade un **conjunto de pruebas de contrato**: un conjunto de pruebas escritas una sola vez contra el contrato y ejecutadas contra cada plug-in en el pipeline propio de ese plug-in. Comprueba el comportamiento que promete el contrato (redondeo, entrada vacía, tipos de error), no la ley fiscal de cada país, que cubren las pruebas propias del plug-in. Por último, añade una comprobación al arrancar o una prueba de humo del despliegue que cargue cada plug-in instalado y verifique su versión de contrato, para detectar las incompatibilidades antes de que las encuentren los usuarios.

Preguntas frecuentes

¿Un microkernel no es más que el patrón estrategia?
A nivel de código, el patrón estrategia es el mismo mecanismo: elegir en tiempo de ejecución una implementación de una interfaz. El microkernel es la versión arquitectónica: los plug-ins se construyen y versionan por separado, y a menudo se instalan por separado, un registro los descubre, y el contrato es una API publicada con reglas de compatibilidad.

¿Pueden los plug-ins tener sus propios datos?
Sí, y a menudo deberían, pero trátalos como los datos propios de cada módulo en un monolito modular: las tablas o los archivos de un plug-in le pertenecen, y el núcleo llega a ellos solo a través del contrato.

¿Debería la configuración sustituir a los plug-ins?
Para diferencias pequeñas y de tipo dato (una tasa, un umbral, una etiqueta), sí: la configuración es más barata y más segura. Los plug-ins son para diferencias que necesitan lógica. Muchos sistemas usan ambos: un plug-in por país más configuración para las tasas vigentes de cada país.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Qué tres cosas debe definir una arquitectura de plug-ins además del propio núcleo?**

Los plug-ins, los contratos que implementan y un registro que permite al núcleo descubrir y buscar plug-ins sin nombrarlos en el código.

▼ **¿Por qué los plug-ins no deben depender unos de otros?**

Porque entonces ya no pueden instalarse, eliminarse, probarse ni versionarse de forma independiente, que es el principal beneficio del estilo; las necesidades compartidas van en el núcleo o en el contrato.

▼ **¿Por qué Visual Studio Code ejecuta las extensiones en un proceso separado?**

Para que una extensión lenta o que se bloquea no pueda congelar ni hacer caer la interfaz de usuario del editor: cambia algo de rendimiento y de complejidad por aislamiento frente a miles de autores externos.

▼ **¿Cuándo es mejor la configuración que un plug-in?**

Cuando la diferencia entre variantes son datos, como una tasa, un límite o una etiqueta, y no lógica. La configuración es más sencilla y más segura de cambiar.

▼ **Inténtalo: Un proceso de pago de entradas usa plugins de medios de pago. El núcleo contiene if plugin == "voucher", y el plugin de vales lee la tabla bookings del núcleo y llama directamente al plugin de tarjeta. Critica esto y di qué debería ofrecer el contrato.**

Esto rompe las reglas del estilo. El núcleo no debe contener casos especiales para plug-ins concretos, y los plug-ins deben depender solo del contrato, nunca de los detalles internos del núcleo ni unos de otros. Define un contrato versionado, con operaciones como can_pay y take_payment, que pase los datos de la reserva que necesita el plug-in. Mueve la lógica de los vales detrás de él, y deja que el núcleo encadene métodos para los pagos divididos. Todo lo que los plug-ins pueden observar pasa a formar parte del contrato, así que mantenlo pequeño.

FUENTES

- Azad Bolour: Notes on the Eclipse Plug-in Architecture (eclipse.org): https://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html
- Visual Studio Code: Extension Host: https://code.visualstudio.com/api/advanced-topics/extension-host
- Python Packaging User Guide: Entry points specification: https://packaging.python.org/en/latest/specifications/entry-points/

- pluggy: the plug-in system behind pytest: https://pluggy.readthedocs.io/en/latest/
- Semantic Versioning 2.0.0: https://semver.org/

SECCIÓN 5.6 DE 5.8 · INTERMEDIO · 25 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Arquitectura hexagonal, de cebolla y limpia

ANTES DE LEER ESTO

Esta sección se basa en:

- 2.6 Modificabilidad y desplegabilidad
- 3.5 SOLID a escala
- 5.2 En capas

Términos anteriores que utiliza: Inversión de dependencias, Doble de prueba, Capa, Interfaz, Framework

LA IDEA EN BREVE

Piensa en un adaptador de viaje. El cargador de tu portátil tiene una forma de enchufe; cada país tiene su propia toma de corriente. No reconstruyes el portátil para cada país: llevas pequeños adaptadores que encajan en la toma por un lado y en tu enchufe por el otro. La arquitectura hexagonal y la arquitectura limpia construyen el software de la misma manera. La parte importante, las reglas de negocio, tiene sus propias tomas estándar (llamadas puertos). Las bases de datos, los sitios web, las empresas de pagos y las herramientas de prueba se conectan mediante adaptadores, así que cualquiera de ellos puede sustituirse sin tocar las reglas.

LO QUE NECESITAS SABER

- La arquitectura hexagonal (puertos y adaptadores, Cockburn, 2005), la de cebolla (Palermo, 2008) y la limpia (Martin, blog de 2012, libro de 2017) comparten una idea: la lógica de negocio en el centro, independiente de frameworks e infraestructura, con las dependencias apuntando hacia dentro.
- Un puerto es una interfaz que pertenece a la aplicación; un adaptador traduce entre un puerto y una tecnología. Los adaptadores conductores llaman a la aplicación; los adaptadores conducidos implementan interfaces a las que llama la aplicación.
- La regla de dependencia dice que el código interior nunca debe nombrar nada de un anillo exterior; la inversión de dependencias permite que el flujo de control vaya hacia fuera mientras las dependencias del código fuente apuntan hacia dentro.
- Los principales beneficios son pruebas rápidas y aisladas de las reglas de negocio y la capacidad de sustituir infraestructura, como los proveedores de pago, escribiendo un adaptador nuevo.

- Los costos son interfaces adicionales, código de mapeo e indirección, que no compensan en aplicaciones CRUD sencillas ni en herramientas de vida corta.
- Haz cumplir la regla automáticamente, por ejemplo con import-linter, con la regla onionArchitecture de ArchUnit o con módulos de compilación separados.

Una idea, tres nombres

La arquitectura hexagonal, la arquitectura de cebolla y la arquitectura limpia son tres formulaciones de una misma idea: poner la lógica de negocio en el centro, hacerla independiente de las interfaces de usuario, las bases de datos, los frameworks y los servicios externos, y hacer que todas las dependencias del código fuente apunten hacia ella. Describen la estructura interna de una aplicación, así que se combinan con cualquier estilo de despliegue: un único monolito, un monolito modular cuyos módulos se construyen cada uno de esta manera, o un microservicio.

Surgieron de la frustración con la clásica arquitectura en capas. En el estilo en capas de la Sección 5.2, la capa de negocio depende de la capa de persistencia, que depende de la base de datos, así que las reglas de negocio acaban moldeadas por las tablas y el mapeo objeto-relacional, y probar una regla implica arrancar una base de datos. La familia de estilos de esta sección invierte esa dependencia con el principio de inversión de dependencias: el código de negocio declara las interfaces que necesita, y el código de infraestructura las implementa. El principio en sí se trata en la Sección 3.5; esta sección muestra las arquitecturas construidas sobre él.

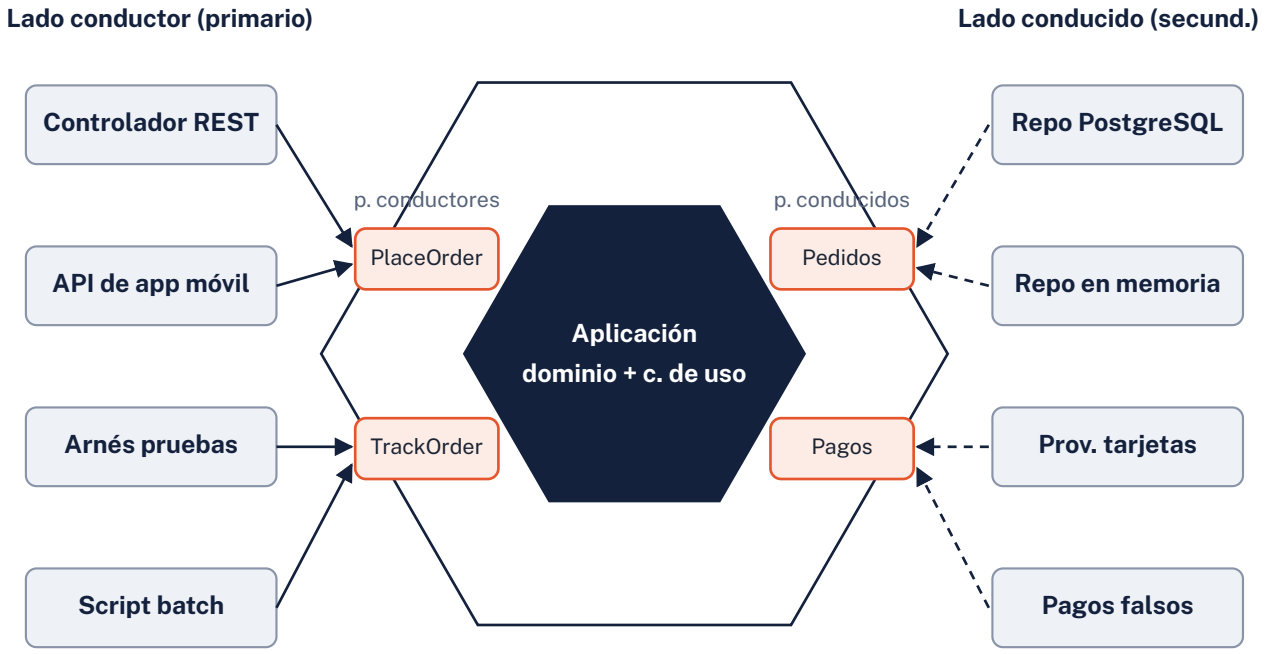
Arquitectura hexagonal: puertos y adaptadores

Alistair Cockburn describió la idea por primera vez en la wiki del Portland Pattern Repository y la desarrolló en un artículo en 2005 con el nombre, más descriptivo, de puertos y adaptadores. Su intención declarada era “permitir que una aplicación sea dirigida por igual por usuarios, programas, pruebas automatizadas o scripts por lotes”, y desarrollarla y probarla aislada de sus futuras bases de datos y dispositivos. En 2024, él y Juan Manuel Garrido de Paz publicaron un libro breve, *Hexagonal Architecture Explained*, que reformula el patrón.

Un puerto es una conversación con un propósito entre la aplicación y el mundo exterior, expresada como una interfaz: "hacer un pedido", "guardar pedidos", "cobrar una tarjeta". Un adaptador es el código que traduce entre un puerto y una tecnología concreta: un controlador HTTP, un repositorio SQL, el SDK de un proveedor de pagos, un script de línea de comandos, un fake en memoria. Un puerto puede tener muchos adaptadores. La aplicación dentro del hexágono solo conoce sus puertos.

El hexágono no tiene ningún significado como forma; Cockburn lo eligió para escapar de la imagen de capas de arriba abajo y para dejar espacio para dibujar varios puertos en cada lado. Lo que importa es el límite entre dentro y fuera y la asimetría entre dos tipos de puerto.

Puertos conductores y conducidos



Flechas continuas: el adaptador llama a un puerto. Discontinuas: lo implementa. Todas las dependencias apuntan hacia dentro.

Los adaptadores conductores llaman a la aplicación a través de los puertos conductores; la aplicación llama a los puertos conducidos, que implementan los adaptadores conducidos, de modo que la base de datos y el proveedor de pagos son plug-ins sustituibles.

Lado conductor y lado conducido

Los puertos y adaptadores conductores (primarios) se sitúan a la izquierda de la imagen habitual. Un adaptador conductor recibe una solicitud de un actor primario (el navegador de un usuario, una app para dispositivos móviles, otro sistema, una prueba, un trabajo programado) y llama a un puerto conductor, que es la API propia de la aplicación: en la práctica, un caso de uso como `PlaceOrder`. El adaptador convierte HTTP, un mensaje o un argumento de línea de comandos en una llamada con valores del dominio, y convierte el resultado de nuevo en una respuesta.

Los puertos y adaptadores conducidos (secundarios) se sitúan a la derecha. La aplicación los llama cuando necesita algo del mundo exterior: guardar un pedido, cobrar una tarjeta, enviar una notificación, leer la hora. El puerto es una interfaz que pertenece a la aplicación y está escrita en sus términos (`OrderRepository.save(order)`), y un adaptador conducido la implementa para una tecnología (PostgreSQL, la API de un proveedor de tarjetas, una pasarela

de SMS). Aquí es donde se produce la inversión de dependencias: en tiempo de ejecución la aplicación llama al adaptador, pero en el código fuente el adaptador depende de la interfaz de la aplicación, y no al revés.

El texto original de Cockburn habla de actores primarios y secundarios; los términos conductor y conducido son hoy más comunes, y él mismo los usa. La distinción importa para las pruebas: los puertos conductores se ejercitan con arneses de pruebas que actúan como usuarios, y los puertos conducidos se sustituyen por dobles de prueba. También importa para el cambio: los canales nuevos (un asistente de voz, una API para socios) llegan como adaptadores conductores nuevos, y la infraestructura nueva llega como adaptadores conducidos nuevos.

Alguien tiene que crear los adaptadores y conectarlos a la aplicación. Eso ocurre en una única raíz de composición, normalmente el código de arranque del programa, que es la única parte del sistema que conoce todos los adaptadores. Un framework de inyección de dependencias puede hacer este trabajo, pero unas simples llamadas a constructores en `main` suelen ser más claras.

SI ESTO ES NUEVO PARA TI

Paso a paso: llamar hacia dentro y llamar hacia fuera

La idea que requiere más atención aquí es la diferencia entre los dos lados del hexágono. En el lado conductor, un adaptador llama a la aplicación; en el lado conducido, la aplicación llama a un adaptador. Aun así, en el código fuente todas las dependencias siguen apuntando hacia dentro. La Sección 3.5 presentó la inversión de dependencias; sigue un pedido para ver dónde se produce.

- Un cliente hace un pedido. El controlador REST, un adaptador conductor, recibe la solicitud HTTP, la convierte en valores simples como el ID del pedido y el total, y llama al caso de uso PlaceOrder. La llamada va hacia dentro.
- PlaceOrder guarda primero el pedido y luego necesita autorizar la tarjeta. Llama a authorise sobre PaymentGateway, una interfaz que declara la propia aplicación, con sus propias palabras. No tiene ni idea de qué empresa de pagos hay detrás.
- El adaptador del proveedor de tarjetas, fuera del hexágono, es un adaptador conducido: implementa PaymentGateway usando el kit de herramientas propio del proveedor. En tiempo de ejecución la llamada va hacia fuera, del caso de uso al adaptador, pero en el código fuente el adaptador hace referencia a la interfaz de la aplicación, nunca al revés.
- Guardar funciona igual. PlaceOrder llama a save sobre OrderRepository; PostgresOrderRepository lo implementa con SQL, y una versión en memoria implementa el mismo puerto para las pruebas.
- Algo tiene que elegir los adaptadores. Eso ocurre una sola vez, en la raíz de composición al arrancar, que crea los adaptadores de PostgreSQL y del proveedor de tarjetas y se los entrega a PlaceOrder. Una prueba hace lo mismo con adaptadores en memoria y falsos.
- **La lección:** en el lado conductor, el exterior llama hacia dentro; en el lado conducido, el interior llama hacia fuera, a través de una interfaz que le pertenece. En ambos casos la aplicación nunca nombra una tecnología, y eso es lo que permite sustituir y probar los adaptadores.

Arquitectura de cebolla

Jeffrey Palermo describió la arquitectura de cebolla en una serie de entradas de blog a partir de julio de 2008, pensada para aplicaciones de negocio de larga vida con un comportamiento considerable (la desaconsejaba para sitios web pequeños). Dibujó anillos concéntricos: el modelo de dominio en el centro, los servicios de dominio a su alrededor, los servicios de aplicación alrededor de estos, y un anillo exterior que contiene la interfaz de usuario, la infraestructura (incluida la base de datos) y las pruebas. Sus principios clave son que la aplicación se construye en torno a un modelo de objetos independiente, que las capas

interiores definen interfaces que implementan las capas exteriores, que todo el acoplamiento va hacia el centro y que el núcleo de la aplicación puede compilarse y ejecutarse por separado de la infraestructura.

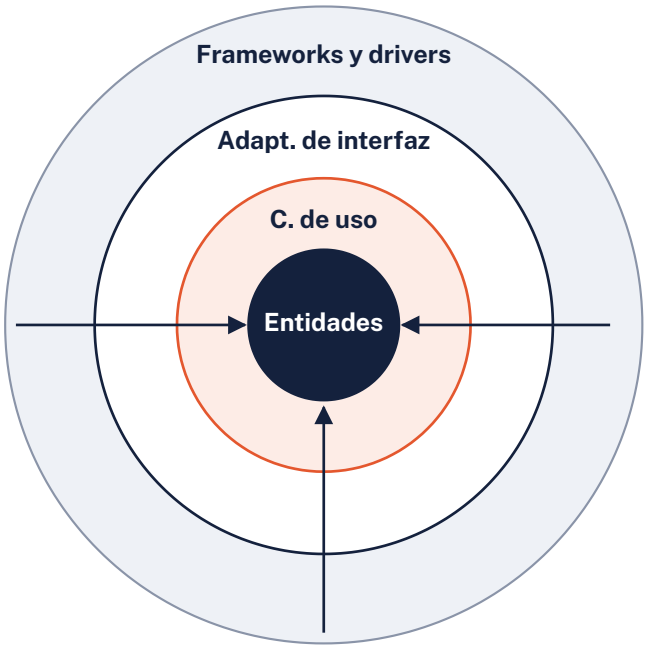
Frente a la arquitectura hexagonal, la de cebolla añade anillos con nombre dentro de la aplicación y hace explícito un punto que la imagen de Cockburn deja implícito: la base de datos no es el cimiento del sistema, sino un servicio externo en el borde. Las interfaces de repositorio viven en el núcleo; sus implementaciones viven en el anillo de infraestructura.

La arquitectura limpia y la regla de dependencia

Robert C. Martin publicó "The Clean Architecture" en su blog en agosto de 2012 como síntesis de la arquitectura hexagonal, la arquitectura de cebolla y otras propuestas similares, y la amplió en el libro *Clean Architecture* en 2017. Su imagen tiene cuatro anillos. **Las entidades** contienen las reglas de negocio de toda la empresa. **Los casos de uso** contienen las reglas específicas de la aplicación que orquestan las entidades. **Los adaptadores de interfaz** convierten los datos entre la forma que necesitan los casos de uso y la que necesitan los agentes externos: controladores, presentadores y gateways. **Los frameworks y drivers** forman el anillo exterior: el framework web, la base de datos, la interfaz de usuario.

La regla de dependencia lo une todo: las dependencias del código fuente apuntan solo hacia dentro, y nada de un anillo interior puede nombrar nada declarado en un anillo exterior, ya sea una clase, una función o un formato de datos. Cuando el control debe fluir hacia fuera (un caso de uso que guarda un pedido), el anillo interior declara una interfaz y el anillo exterior la implementa. Los datos que cruzan un límite deben ser estructuras simples en los términos del anillo interior, nunca una fila de base de datos ni un objeto de solicitud HTTP. Martin sostiene también que la estructura de nivel superior debe revelar el propósito del sistema (pedidos, pagos, entregas) y no su framework, una idea que llama arquitectura que grita (screaming architecture).

Los anillos de la arquitectura limpia



Exterior: framework web, ORM, BD, broker de mensajes, UI

Adapt. de interfaz: controladores, presentadores, implement. de repositorios

Casos de uso: reglas de aplicación, p. ej. PlaceOrder

Entidades: reglas de la empresa, p. ej. Order, Money

La regla de dependencia: el código fuente solo depende hacia dentro

Cada anillo solo puede depender de los anillos que tiene dentro; las entidades no dependen de nada, y los frameworks y drivers se sitúan en el borde, donde pueden sustituirse.

Arquitectura hexagonal, de cebolla y limpia comparadas

Aspecto	Hexagonal (puertos y adaptadores)	De cebolla	Limpia
Creador y fecha	Alistair Cockburn; desarrollada en un artículo en 2005	Jeffrey Palermo; serie de entradas de blog desde julio de 2008	Robert C. Martin; entrada de blog en agosto de 2012, libro en 2017
Imagen	Un hexágono con un interior (la aplicación) y un exterior (los adaptadores), el lado izquierdo conductor y el derecho conducido	Anillos concéntricos: modelo de dominio, servicios de dominio, servicios de aplicación y, después, interfaz de usuario, infraestructura y pruebas	Anillos concéntricos: entidades, casos de uso, adaptadores de interfaz, frameworks y drivers
Regla central	La aplicación habla con el mundo solo a través de puertos; los adaptadores traducen	Todo el acoplamiento va hacia el centro; el núcleo define interfaces que implementa el anillo exterior	La regla de dependencia: las dependencias del código fuente apuntan solo hacia dentro
Estructura interna prescrita	Ninguna: solo dentro frente a fuera	Sí: anillos interiores con nombre para el dominio	Sí: entidades y casos de uso, más orientaciones sobre los presentadores y los datos que cruzan los límites
Énfasis principal	Simetría: usuarios, pruebas, scripts y otros programas pueden dirigir la aplicación por igual	Poner el modelo de dominio, y no la base de datos, en el centro de una aplicación de negocio	Independencia de los frameworks, la interfaz de usuario, la base de datos y los agentes externos; reglas de negocio que se pueden probar
Vocabulario típico	Puerto, adaptador, conductor (primario), conducido (secundario), actor	Modelo de dominio, servicios de dominio, servicios de aplicación, infraestructura	Entidad, caso de uso (interactor), gateway, presentador, controlador

Un caso de uso con un puerto y dos adaptadores

El siguiente ejemplo en Python implementa al estilo hexagonal el caso de uso `PlaceOrder` de la plataforma de reparto de comida. El primer fragmento es el interior: una pequeña clase de dominio, dos puertos conducidos escritos como clases `typing.Protocol` y el caso de uso, que actúa como puerto conductor. El segundo fragmento es el exterior: un adaptador de

PostgreSQL y un doble de prueba en memoria que implementan el mismo puerto. El tercero muestra por qué importa: una prueba unitaria rápida y la raíz de composición donde se conectan los adaptadores de producción.

```
from dataclasses import dataclass
from decimal import Decimal
from typing import Protocol

@dataclass
class Order:
    id: str
    total: Decimal
    status: str = "pending-payment"

class OrderRepository(Protocol):        # driven port
    def save(self, order: Order) → None: ...

class PaymentGateway(Protocol):         # driven port
    def authorise(self, order_id: str, amount: Decimal,
                  idempotency_key: str) → bool: ...

class PlaceOrder:                       # driving port + use case
    def __init__(self, orders: OrderRepository, payments: PaymentGateway):
        self.orders, self.payments = orders, payments

    def __call__(self, order_id: str, total: Decimal) → Order:
        if total ⩽ 0:
            raise ValueError("empty basket")
        order = Order(order_id, total)
        self.orders.save(order)                 # save first: a crash cannot lose a
charged order
        approved = self.payments.authorise(order_id, total, idempotency_key=order_id)
        order.status = "placed" if approved else "payment-failed"
        self.orders.save(order)
        return order
```

Dentro del hexágono: una regla de dominio, dos puertos conducidos y el caso de uso PlaceOrder. Nada de esto importa un framework web, un driver de base de datos ni un SDK de pagos. El pedido se guarda antes de autorizar la tarjeta, y el ID del pedido sirve también como clave de idempotencia, de modo que una caída o un reintento no pueden cobrar a un cliente sin pedido ni cobrarle dos veces.

```
class PostgresOrderRepository:          # real driven adapter
    def __init__(self, conn):
        self.conn = conn
    def save(self, order):
        with self.conn.cursor() as cur:
            cur.execute(
                "INSERT INTO orders (id, total, status) VALUES (%s, %s, %s) "
                "ON CONFLICT (id) DO UPDATE SET status = EXCLUDED.status",
                (order.id, order.total, order.status))
        self.conn.commit()

class InMemoryOrderRepository:          # test double, same port
    def __init__(self):
        self.saved = {}
    def save(self, order):
        self.saved[order.id] = order
```

Fuera del hexágono: un adaptador real y un doble de prueba para el mismo puerto. El caso de uso no puede distinguirlos.

```
class DecliningPayments:
    def authorise(self, order_id, amount, idempotency_key): return False

def test_failed_payment_is_recorded():
    orders = InMemoryOrderRepository()
    place = PlaceOrder(orders, DecliningPayments())
    place("A-17", Decimal("23.50"))
    assert orders.saved["A-17"].status == "payment-failed"

# composition root (main.py): the only place that knows every adapter
# place_order = PlaceOrder(PostgresOrderRepository(conn), CardProviderGateway(key))
# app.add_route("/orders", make_rest_controller(place_order))   # driving adapter
```

Una prueba unitaria dirige el caso de uso directamente, sin servidor web, base de datos ni red; la conexión de producción se hace en un solo lugar al arrancar.

Fíjate en lo que el caso de uso no contiene: nada de SQL, ni códigos de estado HTTP, ni tipos de excepción específicos del proveedor, ni decoradores del framework. La prueba se ejecuta en milisegundos y cubre un caso (un pago rechazado) que sería incómodo de preparar contra el sandbox de un proveedor real. El adaptador de PostgreSQL necesita sus propias pruebas, pero son pruebas de integración centradas en una sola clase contra una base de datos real, no pruebas de reglas de negocio. Este es el beneficio para la capacidad de prueba descrito en la Sección 2.6: controlabilidad y observabilidad mediante dependencias inyectadas.

EJEMPLO PRÁCTICO

Ejemplo práctico: cambiar de proveedor de pagos

La plataforma de reparto de comida debe pasar de un proveedor de tarjetas a otro en un plazo de seis semanas por un cambio de precios. Compara el trabajo en dos versiones del sistema: una en la que el código del proceso de pago llama directamente al SDK del proveedor antiguo en 23 lugares, y otra construida con un puerto conducido `PaymentGateway` y un adaptador.

Llamadas directas: encontrar el impacto	La búsqueda encuentra el SDK en el proceso de pago, los reembolsos, las suscripciones y tres pantallas de administración. Los tipos de excepción propios del SDK se capturan en 11 lugares, y su objeto `Charge` se guarda en una columna de la base de datos.
Llamadas directas: el cambio	Cambian todos los puntos de llamada, más una migración de datos para los objetos guardados. Todas las pruebas que tocaban los pagos se escribieron contra el sandbox del SDK antiguo y hay que reescribirlas. Estimación: de 5 a 6 semanas, con un cambio arriesgado de todo a la vez.
Puertos y adaptadores: encontrar el impacto	Una clase adaptadora (unas 200 líneas) implementa `PaymentGateway`. El núcleo solo conoce `charge`, `refund` y el tipo `PaymentResult` propio del núcleo.
Puertos y adaptadores: el cambio	Escribir un segundo adaptador para el proveedor nuevo; ejecutar contra él las pruebas de contrato existentes del puerto; elegir el adaptador por pedido con un feature flag para que el 1%, luego el 10% y luego el 100% de los pagos usen el nuevo. Estimación: 2 semanas, reversible en cada paso.
Lo que costó el puerto	Diseñar el puerto y el adaptador al principio llevó unos 3 días adicionales, y cada funcionalidad de pago nueva necesita un pequeño cambio en el puerto además de en el adaptador.

El valor de un puerto es proporcional a la probabilidad y al costo del cambio que aísla. Los proveedores de pago, los intermediarios de mensajes (message brokers) y las API de terceros cambian a menudo; el motor de tu propia base de datos principal puede no cambiar nunca, y envolverlo en puertos elaborados es más difícil de justificar.

El puerto debe hablar el lenguaje del dominio (`charge`, `PaymentResult`), no el del proveedor antiguo. Un puerto que refleja la API de un solo proveedor no protege nada.

Características de la familia de puertos y adaptadores

Característica	Valoración	Motivo
Capacidad de prueba (testabilidad)	Alto	Las reglas de negocio pueden probarse en memoria, de forma rápida y determinista, conectando dobles de prueba a los puertos conducidos.
Modificabilidad de la infraestructura	Alto	Sustituir una base de datos, un proveedor de pagos o un framework web implica escribir un adaptador nuevo; el núcleo no cambia.
Modificabilidad de las reglas de negocio	De media a alta	Las reglas viven en un solo lugar, libres de código del framework, pero una funcionalidad sigue atravesando puerto, caso de uso y adaptador.
Sencillez	De baja a media	Más tipos, interfaces y código de mapeo que una aplicación en capas sencilla; al principio cuesta más a los recién llegados.
Velocidad de desarrollo (al inicio)	Media	La estructura adicional ralentiza las primeras funcionalidades de una aplicación CRUD sencilla; se amortiza a medida que crece el dominio.
Costo total	De barato a medio	Sin infraestructura adicional; el costo es el tiempo de los desarrolladores dedicado a escribir y mantener puertos, adaptadores y código de mapeo.
Rendimiento	De media a alta	Las llamadas a interfaces no cuestan casi nada; las verdaderas sobrecargas son el mapeo de objetos entre capas y la pérdida de algunas comodidades del ORM.
Escalabilidad, desplegabilidad, tolerancia a fallos	No los decide este estilo	Dependen del estilo de despliegue (monolito o servicios) en el que se sitúa el hexágono.

Costos, críticas y cuándo usarla

Los beneficios son reales, y también los costos. Cada puerto es una interfaz adicional; cada límite puede necesitar código de mapeo para convertir entre, por ejemplo, una entidad del ORM, un objeto de dominio y una respuesta de la API. Para una aplicación que sobre todo traslada campos de formulario a tablas con poca lógica entre medias, esta estructura añade archivos sin añadir protección, y las convenciones de un framework (un modelo active record, una interfaz de administración generada) pueden dar un resultado mejor y más rápido. Algunos profesionales con experiencia sostienen que las versiones estrictas de la arquitectura limpia producen demasiada indirección y que la mayoría de los equipos nunca cambian de base de

datos; otros responden que el beneficio principal no es cambiar de tecnología, sino pruebas rápidas y fiables y un código de negocio que se puede leer sin conocer el framework. Ambos tienen razón en contextos distintos.

Úsala cuando la lógica de negocio es considerable y va a sobrevivir a los frameworks actuales; cuando el sistema se integra con servicios externos que probablemente cambien; cuando importan las pruebas rápidas y aisladas de las reglas de negocio; y como estructura interna de los módulos de un monolito modular o de servicios construidos con diseño guiado por el dominio (Sección 5.7). **Ten cuidado** con las aplicaciones CRUD sencillas, los prototipos y las herramientas de vida corta, donde la estructura predeterminada del framework suele bastar. Es común un término medio pragmático: puertos para los servicios externos y para todo lo que probablemente cambie, y las comodidades propias del framework en lo demás.

ATENCIÓN

Antipatrones comunes

- **Puertos con forma de tecnología.** Un puerto llamado `SqlQueryExecutor`, o uno que refleja el SDK de un proveedor, mantiene el acoplamiento que debía eliminar. Los puertos deben hablar el lenguaje del dominio.
- **Filtración del framework.** Los casos de uso que aceptan objetos de solicitud HTTP o devuelven entidades del ORM tienen dependencias hacia fuera disfrazadas.
- **Lógica en los adaptadores.** Las reglas de negocio en controladores o en clases de repositorio son invisibles para las pruebas de los casos de uso y se duplican entre adaptadores.
- **Una interfaz para cada clase.** Los puertos van en el límite con el mundo exterior, no entre cada par de clases internas.
- **Casos de uso anémicos que solo reenvían.** Los casos de uso que se limitan a reenviar llamadas a un repositorio son el sumidero de arquitectura (architecture sinkhole) de la Sección 5.2 con otra forma; acéptalos para lecturas sencillas, pero date cuenta si son la mayoría.
- **Una raíz de composición en todas partes.** Los adaptadores creados en lo más profundo del código de negocio (`PostgresOrderRepository()` dentro de un caso de uso) anulan la sustitución de la que depende el estilo.

Hacer cumplir la regla de dependencia

Como ocurre con las capas, la regla se erosiona a menos que la compruebe una máquina. En Python, import-linter puede declarar los anillos como un contrato de tipo *layers* (adaptadores por encima de la aplicación, y esta por encima del dominio) y añadir contratos de tipo *forbidden*

que impidan que los paquetes de dominio y de aplicación importen frameworks web, drivers de base de datos o SDK. En Java, ArchUnit tiene una regla ya preparada, `onionArchitecture()`, en la que nombras los paquetes del modelo de dominio, de los servicios de dominio, de los servicios de aplicación y de los adaptadores, y hace fallar una prueba si alguna dependencia apunta hacia fuera. Dividir el código en módulos de compilación separados (un módulo de dominio sin ninguna dependencia de frameworks) hace que las dependencias hacia fuera sean imposibles de compilar, que es la comprobación más fuerte de todas. Estas comprobaciones son ejemplos de las funciones de aptitud que se tratan en el Capítulo 11.

Una estructura de paquetes que hace visible la regla ayuda a quienes revisan: por ejemplo `orders/domain`, `orders/application` (casos de uso y puertos) y `orders/adapters/{rest,postgres,payments}`. Organizar primero por área de negocio y después por anillo mantiene la estructura alineada con los límites del dominio de las secciones 5.3 y 5.7.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Un controlador REST es un adaptador conductor o conducido? ¿Y un repositorio SQL?**

Un controlador REST es un adaptador conductor: recibe solicitudes y llama a la aplicación. Un repositorio SQL es un adaptador conducido: implementa una interfaz a la que llama la aplicación.

▼ **En la arquitectura limpia, ¿puede un caso de uso devolver una entidad del ORM a un controlador?**

No. El ORM pertenece al anillo exterior, así que un caso de uso que lo nombra tiene una dependencia hacia fuera. Debe devolver una estructura simple o un objeto de dominio definido en un anillo interior.

▼ **¿Dónde se crean los adaptadores y se conectan a la aplicación?**

En una raíz de composición, normalmente el código de arranque, que es el único lugar que conoce todos los adaptadores.

▼ **¿Qué añade la arquitectura de cebolla a la imagen hexagonal?**

Anillos con nombre dentro de la aplicación (modelo de dominio, servicios de dominio, servicios de aplicación) y la afirmación explícita de que la base de datos es una inquietud externa situada en el borde, no el cimiento.

▼ **Inténtalo: Un producto de venta de entradas tiene reglas complejas de asignación de asientos y una pantalla de administración sencilla para editar los datos de contacto de los recintos. ¿Usarías arquitectura hexagonal para ambas? Explica qué ganarías o perderías en cada caso.**

Úsalo solo para la asignación de asientos. Esas reglas ganan tests rápidos y aislados, y la base de datos o el broker se pueden reemplazar cambiando un adaptador, lo que conviene a una lógica que importa y cambia. La pantalla de administración es un CRUD simple, de modo que los puertos y el código de mapeo añadirían indirección con poco que proteger. Usa ahí un enfoque sencillo, manteniéndolo separado del núcleo. Aplica el patrón donde la complejidad justifique su costo, y haz cumplir la regla de dependencia automáticamente.

FUENTES

- Alistair Cockburn: Hexagonal architecture: https://alistair.cockburn.us/hexagonal-architecture/
- Jeffrey Palermo: The Onion Architecture, part 1 (2008): https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/
- Robert C. Martin: The Clean Architecture (2012): https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
- ArchUnit user guide (layered and onion architecture rules): https://www.archunit.org/userguide/html/000_Index.html
- import-linter documentation: https://import-linter.readthedocs.io/en/stable/

SECCIÓN 5.7 DE 5.8 · INTERMEDIO · 26 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Diseño estratégico guiado por el dominio: contextos delimitados y mapas de contextos

ANTES DE LEER ESTO

Esta sección se basa en:

- 3.1 Modularidad
- 3.7 Ley de Conway
- 5.6 Hexagonal y limpia

Términos anteriores que utiliza: Adaptador, Monolito modular, Gran bola de barro (big ball of mud), Ley de Conway

LA IDEA EN BREVE

En un hospital, la palabra "paciente" significa algo distinto para la recepción (nombre, dirección, cita), la planta (cama, medicación, constantes) y la oficina de facturación (aseguradora, factura). Si un único formulario gigante intentara servir a todos, sería enorme y todos se pelearían por él. Funciona mejor cuando cada departamento mantiene su propio registro sencillo y todos acuerdan cómo pasarse la información. El diseño guiado por el dominio aplica esa idea al software: encuentra los departamentos naturales de un negocio, da a cada uno su propio modelo y vocabulario, y hace explícitas las conexiones entre ellos.

LO QUE NECESITAS SABER

- El diseño guiado por el dominio (Eric Evans, 2003) construye el software en torno a un modelo del dominio de negocio desarrollado con expertos del dominio y expresado en un lenguaje ubicuo que se usa igual al hablar que en el código.
- Los subdominios son centrales (donde compite el negocio), de soporte (específicos, pero no diferenciadores) o genéricos (problemas ya resueltos); la clasificación decide dónde invertir el esfuerzo de diseño y qué comprar.
- Un contexto delimitado (bounded context) es un límite dentro del cual un modelo y un lenguaje se aplican de forma coherente; la misma palabra puede significar sin peligro cosas distintas en contextos distintos.
- Un mapa de contextos muestra las relaciones entre los contextos: asociación (partnership), núcleo compartido (shared kernel), cliente-proveedor, conformista, capa anticorrupción, servicio de host abierto (open host service), lenguaje publicado y caminos separados (separate ways).

- El event storming (Alberto Brandolini) descubre los contextos colocando eventos de dominio en una línea de tiempo junto con las personas que conocen el negocio.
- Los agregados son límites de consistencia que se modifican a través de una raíz que hace cumplir las invariantes; los contextos delimitados son la mejor guía para los límites de módulos y servicios.

Por qué el diseño guiado por el dominio tiene su lugar en un capítulo sobre estructura

Todos los estilos de este capítulo necesitan límites. Un monolito modular necesita límites entre módulos; los microservicios del Capítulo 6 necesitan límites entre servicios; incluso una aplicación hexagonal necesita saber qué pertenece al interior del hexágono. El diseño guiado por el dominio (DDD) es el método más usado para encontrar esos límites en el propio dominio de negocio, y no en capas técnicas.

Eric Evans lo presentó en *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley, 2003). Su argumento es que, en la mayor parte del software de negocio, los problemas más difíciles no son técnicos, sino de comprensión del dominio: qué es realmente una póliza, un siniestro, una entrega o un reembolso, y qué reglas lo rigen. Por tanto, el software debería construirse en torno a un modelo del dominio diseñado de forma deliberada, desarrollado conjuntamente por desarrolladores y expertos del dominio, y ese modelo debería protegerse para que no se confunda con otros modelos. En 2015 Evans publicó una breve *DDD Reference* que resume los patrones y añade algunos, como la asociación (partnership) y la gran bola de barro (big ball of mud) como patrones del mapa de contextos, y los eventos de dominio como pieza básica.

El DDD tiene dos mitades. El **diseño estratégico** divide un dominio grande en subdominios y contextos delimitados y traza las relaciones entre ellos; es arquitectura, y es el tema principal aquí. El **diseño táctico** ofrece piezas básicas (entidades, objetos de valor, agregados y otras) para escribir el modelo dentro de un contexto; es diseño, y se trata brevemente.

Lenguaje ubicuo

Un lenguaje ubicuo es un vocabulario compartido, construido entre desarrolladores y expertos del dominio, que se usa de forma coherente en las conversaciones, los documentos, las pruebas y el código. Si los despachadores dicen que un repartidor va "apilado" cuando lleva dos pedidos del mismo restaurante, entonces el código dice `stack`, no `multiOrderAssignment`. Cuando un término resulta ambiguo o incómodo, el equipo cambia el lenguaje y el código a la vez.

Parece una cuestión de nombres, pero tiene consecuencias arquitectónicas. La traducción entre las palabras del negocio y las palabras del código es donde nacen los malentendidos y, por tanto, los defectos. Y en el momento en que dos grupos usan la misma palabra con significados distintos, has encontrado indicios de dos modelos diferentes, que es el punto de partida de los contextos delimitados.

Subdominios: centrales, de soporte y genéricos

El dominio de negocio en su conjunto se divide en subdominios: áreas de actividad como las cartas, los pedidos, los pagos y el despacho. Los profesionales los clasifican en tres tipos, a partir de la distinción de Evans entre el dominio central y los subdominios genéricos; la división en tres con los subdominios de soporte la popularizaron autores posteriores, en especial Vaughn Vernon en *Implementing Domain-Driven Design* (2013).

El dominio central (core domain) es lo que diferencia al negocio de sus competidores y la razón por la que merece la pena construir el software. Para una empresa de reparto de comida, probablemente sea el despacho: asignar pedidos a repartidores y predecir las horas de llegada. Un subdominio de soporte es específico del negocio y necesario, pero no es una fuente de ventaja: la gestión de cartas, el alta de repartidores. Un subdominio genérico es un problema que comparten muchas empresas y para el que existen buenas soluciones: pagos, identidad, correo electrónico y SMS.

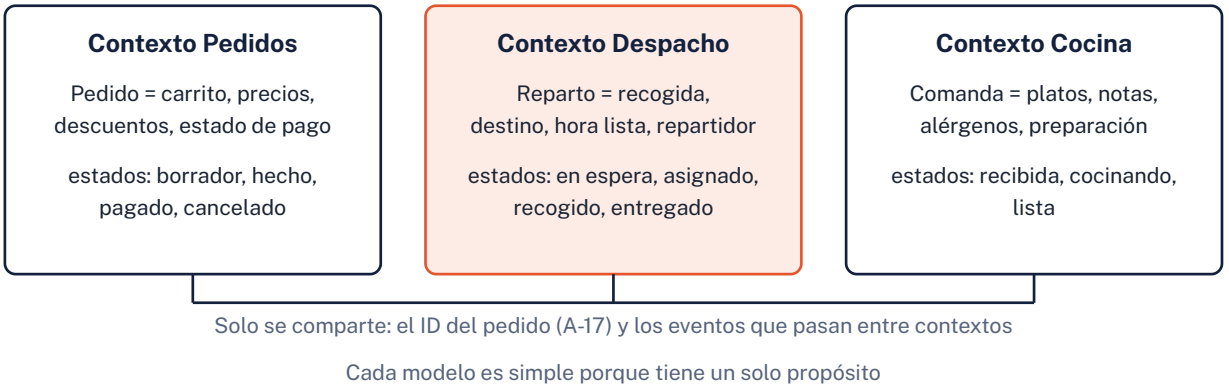
La clasificación es una decisión de inversión. El dominio central merece a las personas con más experiencia, el modelado más rico y un código que sea totalmente propio, porque ahí es donde un software mejor se convierte directamente en ventaja para el negocio. Los subdominios de soporte merecen diseños sólidos pero más sencillos, a menudo un módulo en capas sin complicaciones. Los subdominios genéricos normalmente deberían comprarse, usarse como servicio o construirse como adaptadores finos sobre componentes de código abierto: construir tu propio proveedor de identidad es esfuerzo invertido donde no puedes ganar. La clasificación también cambia con el tiempo; una capacidad de soporte puede pasar a ser central cuando cambia la estrategia del negocio.

Contextos delimitados

Un contexto delimitado (bounded context) es un límite explícito dentro del cual un modelo, y un lenguaje ubicuo, se aplica de forma coherente. Dentro de él, "pedido" tiene exactamente un significado. Fuera de él, la misma palabra puede significar otra cosa, y eso es aceptable, porque cada modelo está diseñado para un propósito. Evans observó que intentar unificar en uno solo el modelo de un sistema grande no es ni viable ni rentable; el intento produce enormes clases compartidas que sirven mal a todos los propósitos y que todos los equipos tienen que modificar.

Los subdominios y los contextos delimitados son cosas distintas, aunque a menudo coinciden. Un subdominio es parte del problema: existe en el negocio haya o no software. Un contexto delimitado es parte de la solución: un límite que eliges para un modelo y, normalmente, para un código y unos datos que pertenecen a un equipo. Lo ideal es un contexto delimitado por subdominio, pero los sistemas heredados suelen tener un contexto que abarca varios subdominios, y un subdominio grande puede necesitar varios contextos.

Una palabra, tres modelos



"Pedido" significa un carrito con precios para Pedidos, un trabajo de recogida y entrega para Despacho y una lista de platos que cocinar en el contexto Cocina; cada modelo se mantiene pequeño porque tiene un solo propósito.

SI ESTO ES NUEVO PARA TI

Paso a paso: las partes del negocio y los límites del software

El punto en el que más gente tropieza aquí es la diferencia entre un subdominio y un contexto delimitado. A menudo coinciden, así que es fácil confundirlos, pero uno es parte del problema de negocio y el otro es un límite que el equipo elige en el software.

- El despacho, que consiste en asignar pedidos a repartidores y predecir las horas de llegada, es un subdominio del negocio de reparto de comida. Existe haya o no software: un despachador con un teléfono y un mapa seguiría haciéndolo.
- Un contexto delimitado es una decisión sobre el software: un límite dentro del cual se aplican un modelo y un lenguaje. El equipo decide que, dentro del contexto Despacho, lo que se gestiona es un 'trabajo de reparto' con una recogida, un destino, una hora en que la comida estará lista y un repartidor.
- Lo ideal es que cada subdominio tenga exactamente un contexto delimitado, normalmente propiedad de un equipo, y el despacho y su contexto coinciden así.
- El encaje no siempre es tan limpio. En un sistema antiguo, una única clase Order compartida puede servir a la vez a los pedidos, al despacho y a la vista de cocina de los restaurantes: un modelo, y por tanto un contexto, estirado sobre varios subdominios, que todos los equipos tienen que modificar y que nadie consigue mantener simple.
- También ocurre lo contrario: un subdominio grande puede necesitar varios contextos delimitados, cada uno con un modelo más pequeño.
- Los subdominios pueden quedarse como están mientras el equipo redibuja sus contextos, como hace el ejemplo práctico que aparece más adelante en esta sección cuando divide esa clase Order compartida en tres modelos.
- **La lección:** encuentra los subdominios estudiando el negocio; elige los contextos delimitados decidiendo dónde deben mantenerse un modelo y un lenguaje. Aspira a un contexto por subdominio y fíjate en dónde no coinciden.

Mapas de contextos

Un mapa de contextos muestra los contextos delimitados de un sistema y las relaciones entre ellos. Las relaciones describen tanto la integración técnica como el poder y la comunicación entre los equipos, por eso el mapeo de contextos trata tanto de organizaciones como de código. Es una aplicación directa de la ley de Conway: el mapa muestra dónde deben coincidir los caminos de comunicación de la organización y los caminos de integración del sistema.

Muchas relaciones son **upstream/downstream** (aguas arriba/aguas abajo): las decisiones del contexto upstream afectan al downstream, pero no al revés. El equipo downstream tiene entonces que elegir cómo arreglárselas, y esa elección es lo que nombra la mayoría de los

patrones.

Patrones de relación del mapa de contextos

Patrón	Qué significa	Quién se adapta a quién	Ejemplo de reparto de comida
Asociación (partnership)	Dos equipos triunfan o fracasan juntos, así que planifican conjuntamente y evolucionan su interfaz a la vez.	Ambos, de mutuo acuerdo	Pedidos y Despacho lanzan juntos el traspaso del pedido a la entrega.
Núcleo compartido (shared kernel)	Dos contextos comparten una parte pequeña y acordada explícitamente del modelo y de su código; los cambios necesitan el consentimiento de ambos equipos.	Ambos, con cuidado	Despacho y Alta de repartidores comparten una biblioteca diminuta de perfil de repartidor (tipo de vehículo, ID del repartidor).
Cliente-proveedor	El equipo upstream (proveedor) se compromete a atender las necesidades del equipo downstream (cliente), que forman parte de su planificación.	Upstream sirve a downstream	Catálogo de cartas suministra cartas y precios a Pedidos, y las necesidades de Pedidos entran en el backlog de Cartas.
Conformista	El equipo downstream adopta el modelo upstream tal como es, porque no puede influir en él y traducirlo no compensa el costo.	Downstream se adapta	Despacho usa directamente el modelo de rutas y ubicaciones del proveedor de mapas.
Capa anticorrupción	El equipo downstream construye una capa de traducción para que el modelo upstream no pueda filtrarse en el suyo.	Downstream traduce	Cartas traduce decenas de formatos de punto de venta de restaurantes a su propio modelo de carta; Pagos aísla la API del proveedor.
Servicio de host abierto (open host service)	El contexto upstream ofrece un protocolo o una API bien definidos para todos los consumidores, en lugar de una integración a medida para cada uno.	Upstream publica una vez	Pagos ofrece una única API de pagos documentada a Pedidos, a reembolsos y a suscripciones.

Patrón	Qué significa	Quién se adapta a quién	Ejemplo de reparto de comida
Lenguaje publicado	Un formato de intercambio documentado y compartido, que a menudo se usa con un servicio de host abierto.	Ambos usan el lenguaje	Los eventos de pedido (OrderPlaced, OrderCancelled) tienen un esquema versionado y documentado.
Caminos separados (separate ways)	Los contextos no se integran en absoluto, porque el beneficio es menor que el costo.	Ninguno	Los informes de nómina de los repartidores se hacen en una herramienta de finanzas aparte, alimentada por una exportación mensual, sin integración en tiempo real.

Dos de estos patrones merecen especial atención. Una capa anticorrupción (ACL) es un conjunto de adaptadores y traductores que construye un contexto downstream para que un modelo ajeno, a menudo un sistema heredado o un proveedor externo, no pueda filtrarse en su propio modelo. Es la misma idea que el adaptador conducido de la Sección 5.6, aplicada entre contextos, y es la opción por defecto para integrarse con cualquier cosa que no controlas. Un servicio de host abierto (open host service) junto con un lenguaje publicado es la forma que tiene el contexto upstream de servir bien a muchos consumidores: un único protocolo documentado y versionado en lugar de una integración a medida para cada uno. Un núcleo compartido (shared kernel) debe seguir siendo pequeño; uno grande vuelve a convertir dos contextos en uno con dos equipos discutiendo por él, y la referencia de Evans de 2015 añade la gran bola de barro (big ball of mud) como etiqueta honesta para una zona sin ningún límite claro, que hay que contener en lugar de ampliar.

Event storming: descubrir los contextos

Los límites se descubren con los expertos del dominio, no se diseñan en solitario. El event storming, creado por Alberto Brandolini y descrito por primera vez en su blog en 2013, es el formato de taller más popular para ello. Las personas que conocen el negocio y las que construyen el software se colocan ante una larga pared de papel y escriben eventos de dominio (cosas que han pasado, en pasado: *Pedido realizado*, *Repartidor asignado*, *Comida recogida*) en notas adhesivas naranjas, que colocan en una línea de tiempo.

Después, el grupo añade los comandos que causan los eventos, las personas y los sistemas externos implicados, las políticas que reaccionan a los eventos ("cada vez que se realiza un pedido, buscar un repartidor") y, lo que es crucial, los puntos calientes (hot spots) donde la gente no está de acuerdo o nadie lo sabe. Los grupos de eventos que comparten un lenguaje y

un propósito sugieren contextos delimitados; los lugares donde la misma palabra se usa de forma distinta sugieren límites; los lugares donde se produce un traspaso sugieren puntos de integración. Una sesión de visión global (big picture) para todo un negocio dura uno o dos días y casi no necesita más preparación que las personas adecuadas y mucho espacio en la pared.

Los patrones tácticos, en breve

Dentro de un contexto delimitado, el DDD ofrece piezas básicas para el modelo. Una entidad tiene una identidad que persiste mientras cambian sus atributos: un pedido es el mismo pedido después de que cambie su estado. Un objeto de valor no tiene identidad y se define solo por sus valores: dos objetos `Money(12.50, "GBP")` son intercambiables, y los objetos de valor suelen ser inmutables. Un evento de dominio registra algo significativo que ha pasado en el dominio.

Un agregado es un grupo de entidades y objetos de valor que se trata como una sola unidad para los cambios, con una entidad como raíz. El código externo solo puede guardar referencias a la raíz y solo debe modificar el agregado a través de ella, de modo que la raíz pueda hacer cumplir las invariantes del agregado (reglas que deben cumplirse siempre, como "no se pueden añadir artículos después de realizar un pedido"). Una transacción debería modificar un solo agregado; a los demás agregados se les hace referencia por su ID y se actualizan mediante eventos de dominio, lo que convierte a los agregados en la unidad natural de consistencia y, más adelante, de distribución. Un repositorio carga y guarda agregados completos y, para el dominio, parece una colección; su interfaz es un puerto conducido en términos hexagonales.

```
from dataclasses import dataclass, field
from decimal import Decimal

@dataclass(frozen=True)
class Money:                                # value object: equal if values equal
    amount: Decimal
    currency: str
    def __add__(self, other):
        if other.currency ≠ self.currency:
            raise ValueError("currency mismatch")
        return Money(self.amount + other.amount, self.currency)
    def times(self, n: int):
        return Money(self.amount * n, self.currency)

@dataclass
class OrderLine:                            # entity inside the aggregate
    dish_id: str
    qty: int
    price: Money

@dataclass
class Order:                                # aggregate root
    id: str
    restaurant_id: str                      # other aggregates referenced by ID only
    lines: list[OrderLine] = field(default_factory=list)
    status: str = "draft"
    events: list = field(default_factory=list)

    def add(self, dish_id, qty, price):
        if self.status ≠ "draft":
            raise ValueError("order already placed")    # invariant
        self.lines.append(OrderLine(dish_id, qty, price))

    def place(self, minimum: Money):
        total = Money(Decimal(0), minimum.currency)
        for line in self.lines:
            total = total + line.price.times(line.qty)
        if total.amount < minimum.amount:
            raise ValueError("below restaurant minimum")  # invariant
        self.status = "placed"
        self.events.append({"type": "OrderPlaced", "order_id": self.id, "total":
str(total.amount)})
```

Un agregado Order: un objeto de valor (Money), una entidad (OrderLine), la raíz del agregado que hace cumplir las invariantes y un evento de dominio que se registra cuando se realiza el pedido.

El ejemplo mantiene pequeño el agregado, hace referencia al restaurante por su ID en lugar de contener un objeto restaurante y hace cumplir sus reglas en métodos en lugar de dejárselas a quien lo llama. Un error habitual es lo contrario: agregados tan grandes que el cambio de

cualquier usuario bloquea las mismas filas, u objetos de dominio que son solo bolsas de getters y setters mientras las reglas viven en clases de servicio (Martin Fowler lo llama modelo de dominio anémico).

EJEMPLO PRÁCTICO

Ejemplo práctico: un mapa de contextos para la plataforma de reparto de comida

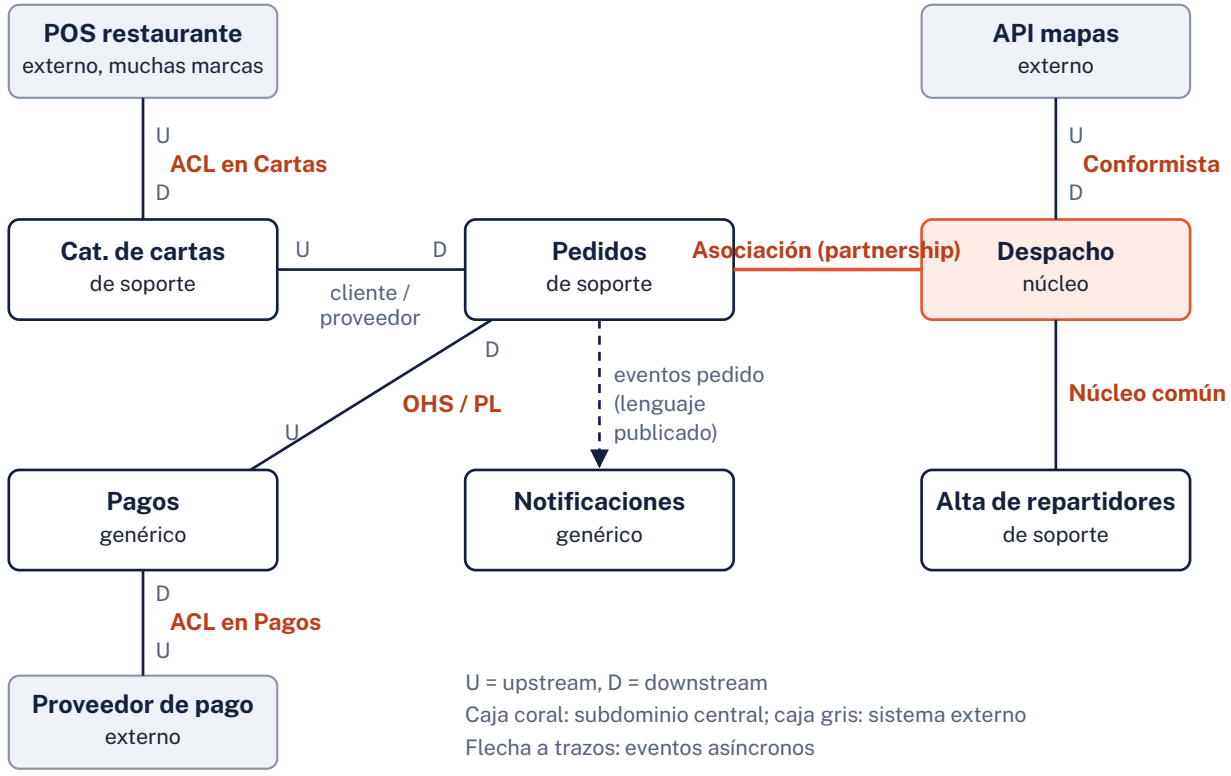
La plataforma ha crecido hasta unos 60 ingenieros en ocho equipos, y parece que cada cambio toca la clase compartida `Order`, que ya tiene 94 campos. El arquitecto organiza un taller de event storming de dos días con gestores de producto, personal de operaciones, soporte a restaurantes e ingenieros, y después dibuja el mapa de contextos que se muestra abajo.

1. Encontrar los subdominios	A partir de la línea de tiempo de eventos: gestión de cartas, pedidos y proceso de pago, pagos, despacho (asignar repartidores a pedidos y predecir las horas de llegada), alta de repartidores, notificaciones a clientes, integración con restaurantes.
2. Clasificarlos	El despacho es el **dominio central**: las entregas más rápidas y baratas son lo que gana clientes y restaurantes. Las cartas, los pedidos y el alta de repartidores son **de soporte**: necesarios y específicos de este negocio, pero no es ahí donde compite. Los pagos y las notificaciones son **genéricos**: problemas resueltos, que es mejor comprar o construir de forma mínima sobre un proveedor.
3. Dibujar los contextos delimitados	Cada subdominio tiene su propio modelo. El "pedido" de Pedidos (carrito, precios, estado de pago) se separa del "trabajo de reparto" de Despacho (recogida, destino, hora en que la comida estará lista) y de la "comanda" de la vista de cocina del restaurante. La clase de 94 campos se convierte en tres modelos enfocados unidos por el ID del pedido.
4. Nombrar las relaciones	Pedidos y Despacho: asociación. De Cartas a Pedidos: cliente-proveedor. Pagos: servicio de host abierto con un lenguaje publicado, y una capa anticorrupción hacia el proveedor externo. Cartas: capa anticorrupción hacia los sistemas de punto de venta de los restaurantes. Despacho: conformista con la API de mapas, una elección deliberada, porque traducir su rico modelo costaría más de lo que ahorra.
5. Decidir la inversión	Los mejores ingenieros y el modelado más cuidadoso van a Despacho. Pagos usa un proveedor a través de un adaptador fino. Notificaciones usa un servicio de mensajería comprado.
6. Decidir el despliegue	Los contextos se convierten primero en módulos de la aplicación existente, que pasa a ser un monolito modular (Sección 5.3), cada uno con su propio esquema e interfaz pública. Despacho, que tiene necesidades de escalado y de lanzamiento muy distintas, es el primer candidato a convertirse más adelante en un servicio separado.

Clasificar Pedidos como de soporte sorprendió a algunas partes interesadas. El criterio no es la importancia (la empresa no puede funcionar sin pedidos), sino la diferenciación: ¿un mejor modelo de pedidos le ganaría al negocio más clientes que un mejor modelo de despacho?

La relación conformista con el proveedor de mapas es un compromiso consciente, recogido en un registro de decisiones, no un accidente. Si el proveedor cambia o se sustituye, Despacho lo notará.

El mapa de contextos



Cada línea indica el patrón de relación; U y D marcan upstream y downstream, y las capas anticorrupción están en el lado downstream, hacia los sistemas que la plataforma no controla.

Los contextos delimitados como límites de módulos y servicios

Los contextos delimitados son la mejor guía disponible para decidir dónde trazar los límites de los módulos en un monolito y los límites de los servicios en un sistema distribuido. Un contexto tiene su propio modelo, su propio lenguaje, sus propios datos e, idealmente, un único equipo propietario, que son exactamente las propiedades que hacen que un módulo sea cohesivo y

esté poco acoplado. En un monolito modular, cada contexto se convierte en un módulo con su propio esquema; dentro de él, la estructura hexagonal de la Sección 5.6 mantiene el modelo independiente de la infraestructura.

Si más adelante hay que desplegar un contexto por separado, su límite ya tiene la forma adecuada para un servicio, y sus relaciones en el mapa de contextos te dicen cómo debe ser la integración. Por eso muchos profesionales recomiendan encontrar primero los contextos y decidir después el despliegue. Lo contrario también se cumple: los servicios que atraviesan los contextos delimitados (un "servicio de pedidos" en el que escriben todos los contextos) producen el monolito distribuido fuertemente acoplado que se trata en la Sección 6.3. Ten en cuenta también que un contexto delimitado no es automáticamente un microservicio; un contexto puede desplegarse como varios servicios, y varios contextos pequeños pueden compartir un mismo despliegue.

Lo que aporta y lo que cuesta el DDD estratégico

Aspecto	Efecto	Motivo
Modificabilidad dentro de un contexto	Alto	Un modelo, un lenguaje y un propietario: los cambios se quedan en un sitio y es fácil razonar sobre ellos.
Autonomía de los equipos	Alto	Los límites de los contextos coinciden con los límites de los equipos, así que los equipos pueden cambiar sus modelos sin negociar cada cambio.
Encaje entre el software y el negocio	Alto	El lenguaje ubicuo y el modelado conjunto reducen los errores de traducción entre los expertos y el código.
Esfuerzo inicial y continuo	De media a alta	Las sesiones de modelado, el tiempo de los expertos del dominio, las capas de traducción y el mantenimiento de los contratos cuestan esfuerzo.
Duplicación e integración de datos	Costo medio	Cada contexto guarda su propia copia de lo que necesita, que debe mantenerse sincronizada mediante eventos o API.
Valor para dominios sencillos	Bajo	En sistemas basados sobre todo en CRUD y con poca lógica, el DDD táctico en particular añade ceremonia con poco retorno.

ATENCIÓN

Errores comunes

- **DDD táctico sin estrategia.** Los equipos adoptan entidades, repositorios y agregados en todas partes, pero nunca trazan contextos delimitados, así que obtienen la ceremonia sin los límites.
- **Un contexto por tabla de la base de datos o por sustantivo.** Los contextos siguen el lenguaje y el propósito, no las entidades de datos; un "servicio de clientes" y un "servicio de pedidos" para cada sustantivo producen partes muy comunicativas y acopladas.
- **Modelar sin expertos del dominio.** Un lenguaje ubicuo inventado solo por los desarrolladores no es más que una convención de nombres.
- **Tratar el mapa como permanente.** Los contextos y las relaciones cambian a medida que cambian el negocio y la organización; revisa el mapa al menos una vez al año.
- **Sobredimensionar los subdominios genéricos.** A los ingenieros los pagos o la identidad les suelen parecer más interesantes que el dominio central; la clasificación existe para resistirse a eso.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Cuál es la diferencia entre un subdominio y un contexto delimitado?**

Un subdominio es parte del problema de negocio y existe haya o no software; un contexto delimitado es un límite elegido en la solución, dentro del cual se aplica un modelo. Lo ideal es que coincidan, pero no tienen por qué.

▼ **Tu equipo se integra con un sistema de facturación heredado cuyo modelo es confuso y que no puedes cambiar. ¿Qué patrón del mapa de contextos protege tu modelo?**

Una capa anticorrupción: traductores de tu lado que convierten el modelo heredado a tus propios términos, para que no pueda filtrarse en tu contexto.

▼ **¿Por qué una transacción debería modificar normalmente un solo agregado?**

Porque el agregado es la unidad cuyas invariantes deben cumplirse en conjunto; modificar varios a la vez crea bloqueos grandes y acoplamiento, mientras que los demás agregados pueden actualizarse mediante eventos de dominio, lo que además permite distribuirlos más adelante.

▼ **¿Por qué los pagos suelen ser un subdominio genérico para una empresa de reparto de comida?**

Porque cobrar pagos es un problema resuelto que comparten muchas empresas y no es donde compite la empresa, así que es mejor comprarlo o construirlo de forma mínima sobre un proveedor que modelarlo en profundidad.

▼ **Inténtalo: En una empresa de venta de entradas, "ticket" significa un asiento vendido para Ventas y una queja de un cliente para Soporte, y una clase Ticket compartida ha crecido hasta 80 campos. Pagos viene de un proveedor externo. ¿Qué contextos y relaciones propondrías?**

Divide la clase en contextos delimitados, Ventas y Soporte, cada uno con su propio modelo y lenguaje, de modo que "ticket" pueda diferir sin riesgo y Soporte pueda decir "caso". La asignación de asientos es probablemente el subdominio principal, Soporte es de soporte y el correo electrónico es genérico, así que invierte y compra en consecuencia. Protege Ventas frente al proveedor externo de pagos con una capa anticorrupción. Entre Ventas y Soporte puede funcionar una relación cliente-proveedor que comparta solo una referencia de reserva. Compruébalo con los expertos del dominio mediante event storming.

FUENTES

- Eric Evans: Domain-Driven Design Reference (2015, PDF): https://www.domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf
- Martin Fowler: Bounded Context: https://martinfowler.com/bliki/BoundedContext.html
- Martin Fowler: DDD Aggregate: https://martinfowler.com/bliki/DDD_Aggregate.html
- EventStorming (Alberto Brandolini): https://www.eventstorming.com/

SECCIÓN 5.8 DE 5.8 · INTERMEDIO · 25 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Cuándo un monolito es la respuesta correcta

ANTES DE LEER ESTO

Esta sección se basa en:

- 5.1 Estilos y patrones
- 5.3 Monolito modular
- 5.7 Diseño guiado por el dominio

Términos anteriores que utiliza: Monolito modular, Contexto delimitado (bounded context), Unidad de despliegue, Gran bola de barro (big ball of mud), Balanceador de carga

LA IDEA EN BREVE

Un restaurante nuevo suele empezar con una sola cocina en la que cada cocinero puede ver y ayudar a todos los demás. Abrir cinco cocinas separadas por toda la ciudad el primer día, cada una con su propio personal, sus entregas y su encargado, costaría mucho más y convertiría cada cambio en la carta en una negociación. Solo cuando esa única cocina está realmente desbordada, o un plato necesita un equipo especial, tiene sentido abrir otra. Con el software pasa algo parecido: un único programa bien organizado suele ser la forma más barata y rápida de empezar, y dividirlo solo merece la pena cuando hay pruebas claras de que esa única unidad se ha convertido en el cuello de botella.

LO QUE NECESITAS SABER

- Un monolito es una decisión de despliegue (una unidad que se compila y se lanza en conjunto), no un veredicto sobre su calidad interna; un monolito bien modularizado es una arquitectura respetable y a menudo superior.
- La prima de los microservicios (microservice premium) y MonolithFirst (2015) de Martin Fowler sostienen que los microservicios solo compensan por encima de cierto nivel de complejidad, y que la mayoría de los sistemas de microservicios exitosos empezaron como monolitos.
- El monolito majestuoso (majestic monolith) de David Heinemeier Hansson (2016) lo defiende para equipos pequeños; Stefan Tilkov (2015) advierte de que los monolitos no modularizados son difíciles de dividir, así que haz cumplir los límites de los módulos desde el principio.

- Un monolito suele ser lo adecuado para equipos pequeños, dominios inciertos, productos en fase temprana, necesidades de consistencia fuerte y una capacidad de operaciones limitada.
- Los monolitos escalan mediante réplicas sin estado detrás de un balanceador de carga, réplicas de lectura, caché, procesos en segundo plano y, si hace falta, particionamiento por cliente (tenant).
- Divide cuando haya pruebas: contención entre equipos, colas de despliegue, necesidades de escalado o de cumplimiento muy distintas; divide de forma incremental siguiendo los contextos delimitados.

La pregunta que hay detrás del capítulo

Todos los estilos de este capítulo se despliegan como una sola unidad. El Capítulo 6 lleva las mismas preguntas a través de una red, donde cada llamada puede fallar, ser lenta o llegar dos veces. Antes de cruzar esa línea, merece la pena preguntarse directamente: ¿cuándo es un monolito la respuesta correcta? Durante buena parte de la década de 2010, la respuesta por defecto del sector fue "rara vez", y muchos equipos dividieron sistemas jóvenes en microservicios porque grandes empresas famosas lo habían hecho. Una década de experiencia ha producido una visión más equilibrada, y esta sección la expone.

La palabra monolito requiere cuidado. Como explicó la Sección 5.1, describe el despliegue (una unidad que se compila, se prueba y se lanza en conjunto), no la calidad interna. Un monolito puede ser un sistema modular bien estructurado o una gran bola de barro, y lo mismo vale para un conjunto de servicios. Muchos de los problemas que se achacan a los monolitos son en realidad problemas de mala modularidad, y dividir en servicios un sistema mal modularizado produce una gran bola de barro distribuida, que es peor.

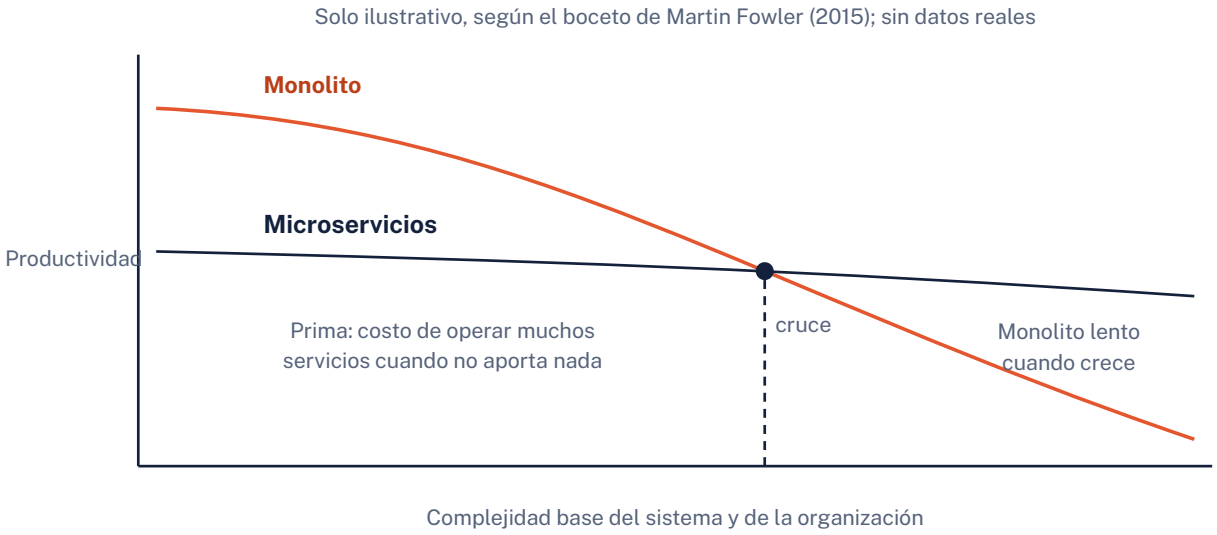
MonolithFirst y la prima de los microservicios

En mayo de 2015, Martin Fowler describió la prima de los microservicios (microservice premium): los microservicios traen costos reales propios (despliegue automatizado de muchas unidades, monitorización, gestión de fallos parciales, consistencia eventual, API versionadas entre equipos), y esos costos frenan a un equipo a menos que el sistema sea lo bastante complejo como para que los microservicios los compensen. Su consejo era partir por defecto de un único monolito bien modularizado y pasar a microservicios solo cuando la complejidad lo exija.

Un mes después, en un artículo titulado MonolithFirst (junio de 2015), añadió dos observaciones de los proyectos que había visto: casi todos los sistemas de microservicios exitosos habían empezado como un monolito que creció demasiado y luego se dividió, y casi

todos los sistemas construidos como microservicios desde cero habían acabado en serios problemas. Una razón es que es difícil encontrar buenos límites entre servicios antes de entender el dominio, y un límite equivocado entre servicios es mucho más caro de mover que un límite equivocado entre módulos. Describió varias vías: construir un monolito e ir desgajando servicios de sus bordes más adelante; construir un monolito deliberadamente desechable; o empezar con unos pocos servicios de grano grueso y dividirlos más cuando los límites se asienten.

La prima de los microservicios



Un boceto conceptual: los microservicios solo compensan a partir de cierto nivel de complejidad; por debajo de él, su costo operativo es una prima que no aporta nada.

El monolito majestuoso y la postura contraria

En febrero de 2016, David Heinemeier Hansson, creador de Ruby on Rails, publicó "The Majestic Monolith" en el blog Signal v. Noise. Describió Basecamp, construido entonces por un equipo de una docena de programadores, como una aplicación integrada que servía a clientes web, móviles, de escritorio y de correo electrónico, y sostuvo que los equipos pequeños deberían apostar por un monolito bien hecho en lugar de copiar la arquitectura de organizaciones con miles de ingenieros. Su frase clave fue “¡No distribuyas tu computación!”, salvo que de verdad tengas que hacerlo.

No todos los profesionales están de acuerdo. También en junio de 2015, Stefan Tilkov sostuvo en el sitio de Fowler, en "Don't start with a monolith", que dividir un monolito más adelante suele ser más difícil de lo esperado, porque sus partes acaban fuertemente acopladas a través de bibliotecas compartidas, objetos de dominio y la base de datos. Si el sistema es claramente lo bastante grande como para justificar partes separadas y el dominio se entiende bien,

recomendaba diseñar subsistemas separados desde el principio. Las dos posturas están menos alejadas de lo que parece: ambas dicen que lo difícil son los límites, y un monolito cuyos módulos se hacen cumplir desde el primer día (Sección 5.3) responde a la mayor parte de la inquietud de Tilkov.

Los casos reales apuntan en ambas direcciones, pero comparten una lección. Segment describió en 2018 cómo había consolidado más de 140 servicios pequeños de nuevo en un único servicio después de que la sobrecarga operativa y unas bibliotecas compartidas cada vez más divergentes frenaran la entrega; Amazon Prime Video pasó en 2023 una carga de trabajo de monitorización de componentes serverless distribuidos a un único proceso (consulta la Sección 1.3). Del mismo modo, las organizaciones con cientos de equipos, como los casos a gran escala de la Sección 6.3, obtienen una independencia real de los servicios. La lección es el encaje: el estilo de despliegue debe corresponder a la organización, a la madurez del dominio y a las cualidades que se exigen.

Cuándo un monolito es la respuesta correcta

Un equipo pequeño. Con uno o unos pocos equipos, el costo de comunicación que eliminan los microservicios es bajo, mientras que su costo operativo es el mismo que para una organización grande. Un monolito permite que todos entiendan y cambien el sistema completo.

Un dominio incierto o un producto en fase temprana. Cuando aún estás descubriendo qué es el producto, los límites se moverán. Mover un límite entre módulos es una refactorización; moverlo entre servicios es una migración con datos, API y despliegues.

Necesidades de consistencia fuerte. Una única transacción de base de datos que abarca pedidos, pagos y existencias es sencilla y correcta en un monolito. Entre servicios necesita sagas, bandejas de salida y compensaciones (Sección 6.7).

Capacidad de operaciones limitada. Cada servicio necesita su propio pipeline, su monitorización, sus alertas, sus reglas de escalado y sus conocimientos para las guardias. Un equipo sin una plataforma para todo esto dedicará su tiempo a la fontanería.

Interacciones sensibles al rendimiento. Las llamadas dentro del proceso tardan de nanosegundos a microsegundos; las llamadas de red tardan milisegundos y pueden fallar. Las interacciones muy frecuentes entre partes del dominio son mucho más baratas dentro de un mismo proceso.

En todos estos casos, la forma recomendada es un monolito modular, no uno sin estructura. Mantiene abierta la opción de extraer servicios, como explica la Sección 5.3.

SI ESTO ES NUEVO PARA TI

Paso a paso: mover un límite que se trazó demasiado pronto

La idea más difícil de esta sección es por qué un límite equivocado cuesta mucho más entre servicios que entre módulos, una de las principales razones detrás de MonolithFirst. Al principio, un equipo todavía está aprendiendo el dominio, así que algunos límites estarán en el lugar equivocado. Sigue un dato que tiene que moverse.

- Un producto joven de reparto de comida tiene una parte de Pedidos y una parte de Despacho. Al principio, el equipo guarda en Pedidos la hora en que estará listo cada pedido, es decir, cuándo espera la cocina que la comida esté lista.
- Meses después descubren que Despacho usa esa hora mucho más, para planificar a los repartidores. Su sitio está en Despacho, como en el modelo de la Sección 5.7.
- En un monolito modular, moverla es una refactorización: mover el código y la columna al otro módulo, actualizar a quienes la llaman, dejar que las pruebas encuentren lo que se haya roto y entregarlo en un solo lanzamiento.
- Si Pedidos y Despacho fueran servicios separados, el mismo cambio sería una migración: cambiar dos API, copiar datos en vivo entre dos bases de datos mientras siguen llegando pedidos, lanzar ambos servicios en un orden seguro y mantener funcionando el camino antiguo hasta que todos los que llaman hayan cambiado.
- Al principio de un producto, cambios como este se repiten una y otra vez, porque el equipo todavía está aprendiendo cómo funciona el negocio. Cada uno es barato dentro de un único despliegue y caro a través de una red, por eso Fowler aconseja empezar con un monolito.
- **La lección:** mientras todavía estás descubriendo dónde deben ir los límites, mantenlos dentro de un único despliegue, donde moverlos es barato. Hazlos cumplir como módulos desde el principio, para que cualquier división posterior, hecha cuando haya pruebas que la justifiquen, siga límites que ya han demostrado su valor.

Escalar un monolito

Una creencia extendida es que los monolitos no pueden escalar. Escalan más de lo que la mayoría de los sistemas llegan a necesitar, usando el eje X del cubo de escalado descrito por Martin Abbott y Michael Fisher (ejecutar muchas copias idénticas) junto con unas pocas tácticas estándar. El Capítulo 9 trata cada una en profundidad (Sección 9.1).

Réplicas horizontales detrás de un balanceador de carga. Ejecuta muchas copias idénticas de la aplicación y reparte las solicitudes entre ellas. Esto solo funciona si la aplicación es sin estado (stateless): las sesiones, los archivos subidos y las cachés deben estar en almacenes

compartidos, para que cualquier réplica pueda atender cualquier solicitud (Sección 9.2). También aporta redundancia: que falle una réplica ya no detiene el servicio.

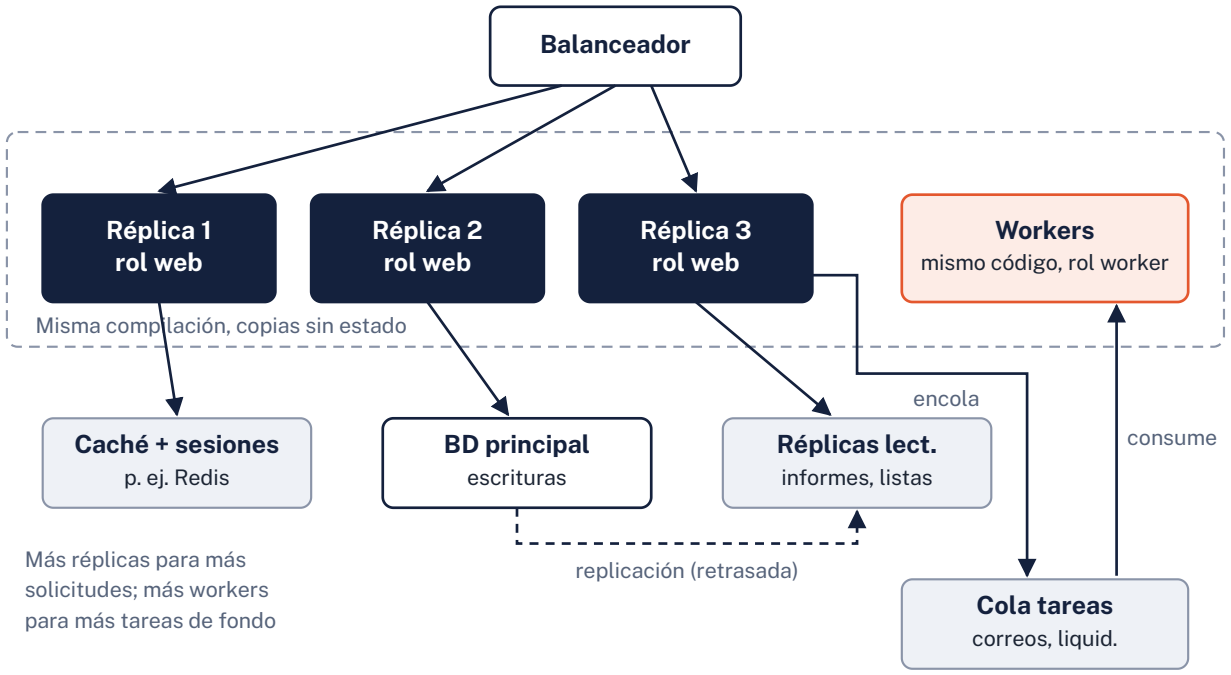
Réplicas de lectura. La mayoría de los sistemas leen mucho más de lo que escriben. Las réplicas de lectura de la base de datos copian los cambios de la principal y atienden lecturas como el historial de pedidos y los informes, a costa del retardo de replicación (replication lag): una réplica puede ir una fracción de segundo por detrás (Sección 7.4).

Caché. Las cartas, las páginas de producto y la configuración cambian rara vez y se leen constantemente; guardarlas en caché en memoria o en una caché compartida elimina la mayor parte de la carga de la base de datos (Sección 7.6). Los recursos estáticos van a una CDN.

Procesos en segundo plano. El trabajo lento o a ráfagas (correos electrónicos, facturas en PDF, importaciones, liquidaciones) va a una cola y lo procesan procesos en segundo plano (workers). Los workers suelen ejecutar el mismo código base con un comando de arranque distinto, así que el sistema sigue siendo un único código base, pero los roles web y worker escalan por separado (Sección 9.3).

Máquinas más grandes y particionamiento. El escalado vertical de la base de datos suele ser el siguiente paso más sencillo. En sistemas muy grandes, los datos pueden particionarse por cliente (tenant) o por región, con una copia completa del monolito por partición (el eje Z del cubo de escalado). Stack Overflow es un ejemplo conocido de hasta dónde puede llegar un diseño mayoritariamente monolítico: según el informe de sus propios ingenieros, un día de febrero de 2016 sus balanceadores de carga gestionaron unos 209 millones de solicitudes HTTP, con la aplicación principal de Q&A (una aplicación multicliente para todos sus sitios de Q&A) funcionando en nueve servidores web principales, y cuatro máquinas SQL Server atendiendo a toda la red.

Un monolito escalado horizontalmente



Un código base, muchas copias: réplicas web sin estado detrás de un balanceador de carga, caché y sesiones compartidas, una base de datos principal con réplicas de lectura y workers alimentados por una cola.

EJEMPLO PRÁCTICO

Ejemplo práctico: ¿hasta dónde llega un monolito?

Una nueva start-up de reparto de comida tiene 7 ingenieros y espera, para finales del segundo año, un pico de 1200 solicitudes HTTP por segundo a la hora de la cena del viernes. Una prueba de carga del monolito modular muestra que una réplica (4 vCPU) atiende 180 solicitudes por segundo dentro del objetivo de p95 de 300 ms. Alrededor del 85% de las solicitudes son lecturas (cartas, seguimiento de pedidos). Las cifras son ilustrativas.

Réplicas para el pico	1200 / 180 = 6.7, así que 7 réplicas en el pico
Margen de capacidad (headroom)	Mantén la utilización muy por debajo de la saturación (Sección 2.4) y sobrevive a la caída de una de las tres zonas de disponibilidad: planifica alrededor de un 50% de margen de capacidad (headroom; con solo dos zonas sería alrededor del 100%), así que 7 x 1.5 = unas 11 réplicas en el pico, con menos fuera de las horas punta
Escrituras en la base de datos	El 15% de 1200 = 180 solicitudes por segundo, cada una con unas 3 escrituras: unas 540 escrituras por segundo, muy dentro de lo que admite una única base de datos principal PostgreSQL gestionada en una instancia de tamaño medio
Lecturas de la base de datos	1020 solicitudes de lectura por segundo; una caché para las cartas (que cambian rara vez) absorbe la mayoría, y dos réplicas de lectura se encargan del historial de pedidos y los informes, aceptando un segundo o así de retardo de replicación
Trabajo en segundo plano	Los correos de recibo, las liquidaciones a los repartidores y los informes para restaurantes pasan a un grupo de workers alimentado por una cola, para que un proveedor de correo lento no pueda ralentizar el proceso de pago
Conclusión	Un único código base, desplegado como una sola unidad y ejecutado como unas 11 réplicas web más los workers, cumple el objetivo de dos años. El presupuesto de arquitectura se dedica a los límites de los módulos y a la ausencia de estado, no a dividir.

El límite que llega primero rara vez es la CPU. Lo más frecuente es la coordinación entre equipos a medida que crece el grupo de ingeniería, o un único módulo (como el despacho) con necesidades de escalado o de lanzamiento muy distintas.

Todas las cifras de aquí deberían sustituirse por mediciones de tus propias pruebas de carga; la planificación de capacidad se trata en la Sección 9.7.

Señales de que ha llegado el momento de dividir

Dividir debe ser una respuesta a las pruebas, no a las modas. Las señales de que una única unidad de despliegue se ha convertido en la restricción son sobre todo organizativas y operativas:

Contención entre equipos. Varios equipos modifican el mismo código y se esperan unos a otros; los conflictos de fusión, las reuniones de coordinación y las compilaciones rotas por otros equipos aumentan a medida que crece la organización. Si los módulos ya están bien delimitados y la contención persiste, quizá lo que falte sea la independencia de despliegue.

Colas de despliegue. Los lanzamientos se agrupan en trenes porque cada cambio exige volver a probar todo el sistema; una pequeña corrección espera días detrás de trabajo no relacionado; el cambio fallido de un equipo bloquea el lanzamiento de todos.

Necesidades de escalado o de disponibilidad muy distintas. Un módulo (por ejemplo, el seguimiento de repartidores en tiempo real) necesita en el pico cincuenta veces la capacidad del resto, o debe seguir funcionando cuando todo lo demás está caído. Escalar todo el monolito para las necesidades de un módulo resulta caro.

Distinto alcance de seguridad o de cumplimiento. Los datos de tarjetas o los historiales médicos pueden justificar una unidad separada y estrictamente controlada para mantener el resto del sistema fuera del alcance de las auditorías.

Distintas necesidades tecnológicas. Por ejemplo, un modelo de machine learning que es mejor servir desde Python dentro de un monolito en Java.

Algunas razones no son buenas razones por sí solas: "los monolitos son legado", el deseo de usar una tecnología nueva o un código lento (que es un problema de diseño que dividir no arreglará y que puede empeorar al añadir llamadas de red).

Cómo dividir cuando llegue el momento

Divide de forma incremental, un módulo cada vez, empezando por el módulo que tenga a la vez el límite más claro y la razón más fuerte para salir. Modulariza primero si el código todavía no es modular: extraer un servicio de una maraña arrastra la maraña consigo. Después, extrae siguiendo los contextos delimitados (Sección 5.7): pon una API de red delante de la API existente del módulo, mueve sus datos a su propia base de datos, sustituye los eventos dentro del proceso por un intermediario de mensajes (message broker) y sustituye cualquier transacción compartida por sagas explícitas. Dirige el tráfico al nuevo servicio de forma gradual y mantén el camino antiguo hasta que el nuevo haya demostrado su valor.

Este es el enfoque strangler fig (higuera estranguladora) y sus variantes, que se tratan en la Sección 11.6. El destino a menudo no es "todo microservicios", sino un monolito con unos pocos servicios extraídos allí donde las pruebas lo exigieron; la arquitectura basada en servicios de la Sección 6.2 es un estado final habitual y pragmático.

Los estilos del Capítulo 5 comparados

Característica	En capas	Monolito modular	Tuberías y filtros	Microkernel	Hexagonal / limpia
Particionamiento principal	Técnico	Por dominio	Pasos de procesamiento	Núcleo más plug-ins	Dentro frente a fuera
Sencillez	Alto	De media a alta	Alto	Media	De baja a media
Costo total	Barato	Barato	Barato	Barato	De barato a medio
Modificabilidad ante cambios de negocio	De baja a media	Alto	Alta dentro del flujo	Alta en las direcciones previstas	Alta para la infraestructura
Capacidad de prueba (testabilidad)	Media	De media a alta	Alto	De media a alta	Alto
Desplegabilidad	Bajo	De baja a media	De baja a media	Media	Depende del estilo de despliegue
Escalabilidad	Bajo	De baja a media	De baja a media	Bajo	Depende del estilo de despliegue
Tolerancia a fallos	Bajo	Bajo	De baja a media	De baja a media	Depende del estilo de despliegue
Encaje típico	Apps pequeñas y medianas, un equipo	Productos en crecimiento, varios equipos	ETL, multimedia, logs, compiladores	Productos con muchas variantes	Lógica que debe sobrevivir a los frameworks

Las valoraciones resumen los juicios hechos en cada sección y suponen implementaciones típicas. El diseño guiado por el dominio no aparece como columna porque es un método para encontrar límites, que cualquiera de estos estilos puede usar. Los sistemas reales combinan

estilos: un monolito modular cuyos módulos son hexagonales por dentro, con un proceso de importación de tuberías y filtros y un microkernel para las reglas fiscales, es totalmente normal.

GUÍA DE DECISIÓN

Guía de decisión: elegir entre los estilos del Capítulo 5

Úsala para un sistema nuevo o una reestructuración importante que vaya a ejecutarse como una sola unidad de despliegue. Recorre las preguntas de arriba abajo; varios estilos pueden aplicarse a distintos niveles.

Q1 **¿El sistema es principalmente un flujo de datos a través de una serie de transformaciones (importar, limpiar, enriquecer, exportar), y no usuarios que interactúan con un estado compartido?**

- **Sí** Usa tuberías y filtros para ese flujo (Sección 5.4). Mantén cada filtro sin estado y el formato de datos entre filtros explícito.
- **No** Pasa a la siguiente pregunta.

Q2 **¿El producto tiene un flujo de trabajo central estable, pero muchas variantes de pasos concretos (por país, cliente, proveedor o formato de archivo), o necesitan ampliarlo terceros?**

- **Sí** Usa un microkernel (Sección 5.5) para esos puntos de variación, idealmente después de construir dos o tres variantes, para que el contrato se base en casos reales.
- **No** Pasa a la siguiente pregunta.

Q3 **¿Es pequeño (un equipo, unas pocas decenas de pantallas), con reglas sencillas y con pocas probabilidades de crecer mucho?**

- **Sí** Usa una arquitectura en capas (Sección 5.2) con capas cerradas que haga cumplir una prueba de arquitectura. No añadas más estructura de la que necesita el problema.
- **No** Pasa a la siguiente pregunta.

Q4 **¿Trabajarán en él varios equipos, o el dominio abarca varias áreas con su propio lenguaje (cartas, pedidos, despacho)?**

- **Sí** Usa un monolito modular (Sección 5.3) con módulos trazados según los contextos delimitados (Sección 5.7), cada uno propietario de sus datos y su API.
- **No** Tanto una estructura en capas como una modular funcionan; si es probable que crezca, prefiere módulos por dominio, porque reestructurar más adelante es costoso.

Q5 **Dentro de cada módulo o aplicación, ¿la lógica de negocio es considerable, o depende de servicios externos que probablemente cambien?**

Sí Estructura el interior con puertos y adaptadores (Sección 5.6): el dominio en el centro y adaptadores para la web, la base de datos y los proveedores.

No Mantén dentro del módulo la estructura por defecto del framework; añade puertos solo en los límites que probablemente cambien.

Lista de comprobación: un monolito que se mantiene sano y divisible

La mayoría de estos puntos cuestan poco si se hacen pronto y muchísimo si se incorporan a posteriori.

- [ ] **Los módulos siguen los dominios de negocio**
  Los paquetes de nivel superior llevan el nombre de los contextos delimitados (pedidos, despacho, pagos), no de capas técnicas.
- [ ] **Cada módulo tiene una API pública pequeña**
  Todo lo demás es interno; una prueba de arquitectura hace fallar la compilación ante una importación prohibida.
- [ ] **Cada tabla tiene un único módulo propietario**
  Los esquemas separados o las convenciones de nombres hacen visible la propiedad; ningún módulo lee las tablas de otro.
- [ ] **La aplicación es sin estado**
  Las sesiones, los archivos subidos y las cachés están en almacenes compartidos, para que cualquier réplica pueda atender cualquier solicitud y se puedan añadir réplicas a voluntad.
- [ ] **El trabajo lento se ejecuta en procesos en segundo plano**
  Los correos electrónicos, los informes, las liquidaciones y las importaciones pasan por una cola, para que las réplicas web sigan siendo rápidas y ambas partes puedan escalarse por separado.
- [ ] **La compilación y las pruebas siguen siendo rápidas**
  Controla los tiempos de compilación y de pruebas; mantén el pipeline principal por debajo de unos 10 a 15 minutos probando los módulos por separado y en paralelo.
- [ ] **El despliegue es automatizado y frecuente**
  Lanzamientos pequeños y frecuentes con rollback automatizado; los feature flags desacoplan el despliegue del lanzamiento.
- [ ] **Observabilidad por módulo**
  Los registros, las métricas y las trazas se etiquetan con el módulo, para que puedas ver qué módulo es lento, falla o es caro.
- [ ] **Se miden las señales de tensión**
  El tiempo de entrega, la longitud de la cola de despliegue, los conflictos de fusión por equipo y el uso de recursos por módulo se revisan con regularidad, para que la decisión de dividir se base en pruebas.

ATENCIÓN

Errores comunes

- **Tratar "monolito" como una estructura.** Es una decisión de despliegue. La calidad interna es una decisión aparte, y es la que decide si podrás cambiar o dividir el sistema más adelante.
- **Dividir para arreglar la calidad del código.** Los servicios trazados alrededor de un código base enmarañado se convierten en un monolito distribuido: todo el acoplamiento, más la red.
- **Escalar lo que no es.** Añadir réplicas mientras las sesiones se guardan en memoria, o mientras la base de datos es el cuello de botella, no aporta nada.
- **No medir nunca la tensión.** Sin datos de tiempo de entrega, de despliegue y de contención, la decisión de dividir se convierte en una cuestión de opinión y de moda.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Por qué un límite equivocado entre servicios es más caro que un límite equivocado entre módulos?**

Mover el límite de un módulo es una refactorización dentro de un único código base y una única transacción de base de datos; mover el límite de un servicio cambia API de red, almacenes de datos, despliegues y a menudo la propiedad de los equipos, y debe hacerse sin tiempo de inactividad.

▼ **¿Qué debe cumplir un monolito antes de poder escalarlo añadiendo réplicas?**

Debe ser sin estado: las sesiones, los archivos y las cachés deben estar en almacenes compartidos, para que cualquier réplica pueda atender cualquier solicitud.

▼ **Nombra dos señales de que un monolito se ha convertido en la restricción, y una razón para dividirlo que no basta por sí sola.**

Señales: contención persistente entre equipos y colas de despliegue, o un módulo con necesidades de escalado o de cumplimiento muy distintas. No basta por sí sola: la moda, querer una tecnología nueva o un código lento.

▼ **¿En qué coinciden las posturas de MonolithFirst y de "Don't start with a monolith"?**

Ambas dicen que el problema difícil es encontrar y mantener buenos límites; un monolito con límites entre módulos que se hacen cumplir responde a la inquietud de Tilkov y conserva el menor costo que defiende Fowler.

▼ **Inténtalo: Seis ingenieros esperan un pico de 900 solicitudes por segundo, un 80% de lecturas. Una réplica de su monolito modular atiende 150 dentro del objetivo. ¿Cuántas réplicas se necesitan con un 30% de margen? Quieren microservicios: ¿qué les aconsejas?**

900 / 150 = 6 réplicas en el límite. Funcionar al 70% significa 900 / (150 x 0.7) = 8.6, es decir, 9 réplicas. Eso se ejecuta fácilmente detrás de un balanceador de carga y, con un 80% de lecturas, la caché y las réplicas de lectura protegerán la base de datos. No hay evidencias de contención entre equipos, colas de despliegue ni necesidades de escalado muy diferentes, y seis ingenieros no pueden asumir el costo operativo. Quédate con el monolito y divide más adelante según contextos delimitados si aparecen evidencias.

FUENTES

- Martin Fowler: MonolithFirst (2015): https://martinfowler.com/bliki/MonolithFirst.html
- Martin Fowler: Microservice Premium (2015): https://martinfowler.com/bliki/MicroservicePremium.html
- David Heinemeier Hansson: The Majestic Monolith (2016): https://signalvnoise.com/svn3/the-majestic-monolith/
- Stefan Tilkov: Don't start with a monolith (2015): https://martinfowler.com/articles/dont-start-monolith.html
- Nick Craver: Stack Overflow, the architecture (2016 edition): https://nickcraver.com/blog/2016/02/17/stack-overflow-the-architecture-2016-edition/

Lecturas recomendadas

- *Fundamentals of Software Architecture (2nd edition)*, Mark Richards and Neal Ford (2025), O'Reilly Media. Un capítulo por estilo, incluidos el de capas, el monolito modular, el pipeline y el microkernel, cada uno con una valoración de sus características; el modelo de las comparaciones de este capítulo.
- *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal (1996), John Wiley and Sons. El catálogo clásico de patrones arquitectónicos, incluidos Capas (Layers), Tuberías y filtros (Pipes and Filters) y Microkernel, cada uno con su contexto, sus fuerzas y sus consecuencias.
- *An Introduction to Software Architecture*, David Garlan and Mary Shaw (1994), Carnegie Mellon University, Software Engineering Institute (CMU/SEI-94-TR-021). El breve artículo que definió los estilos arquitectónicos como componentes, conectores y restricciones, con las tuberías y filtros y los sistemas en capas entre sus ejemplos.
  https://www.sei.cmu.edu/library/an-introduction-to-software-architecture/
- *Software Architecture: The Hard Parts*, Neal Ford, Mark Richards, Pramod Sadalage and Zhamak Dehghani (2021), O'Reilly Media. Desarrolla el cuanto de arquitectura y el análisis del acoplamiento y de la propiedad de los datos que decide por dónde se puede dividir un sistema.
- *Monolith to Microservices*, Sam Newman (2019), O'Reilly Media. Una guía equilibrada sobre cuándo, si conviene y cómo descomponer un monolito, con argumentos sólidos a favor del monolito modular.
- *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Eric Evans (2003), Addison-Wesley. El origen de los contextos delimitados, los mapas de contextos y el lenguaje ubicuo que se usan para encontrar los límites de módulos y servicios.
- *Hexagonal Architecture*, Alistair Cockburn (2005), alistair.cockburn.us. La descripción original de puertos y adaptadores, breve y todavía la exposición más clara de la idea.
  https://alistair.cockburn.us/hexagonal-architecture/
- *Deconstructing the Monolith: Designing Software that Maximizes Developer Productivity*, Kirsten Westeinde (2019), Shopify Engineering. Un relato de primera mano sobre cómo se convirtió una aplicación Rails muy grande en un monolito modular en lugar de en microservicios.
  https://shopify.engineering/deconstructing-monolith-designing-software-maximizes-developer-productivity

Glosario

Adaptador
Código que traduce entre un puerto y una tecnología concreta, como un controlador HTTP, un repositorio SQL o el SDK de un proveedor de pagos. Un puerto puede tener muchos adaptadores.

Adaptador conducido (secundario)
Un adaptador que implementa una interfaz a la que llama la aplicación cuando necesita algo del exterior, como almacenar datos, cobrar en una tarjeta o enviar un mensaje.

Adaptador conductor (primario)
Un adaptador que recibe solicitudes del exterior, como un controlador web, un arnés de pruebas o un trabajo programado, y llama a la aplicación a través de un puerto conductor.

Agregado
Un grupo de objetos de dominio que se modifica como una unidad a través de una única entidad raíz, que hace cumplir las reglas que siempre deben mantenerse para el grupo.

Antipatrón
Una solución de uso común que parece razonable pero que, de forma sistemática, causa más problemas de los que resuelve.

API del módulo
La parte pequeña y deliberadamente pública de un módulo que otros módulos pueden usar; todo lo demás es interno y puede cambiar libremente.

Aplicación sin estado (stateless)
Aplicación que no guarda datos por usuario en su propia memoria entre solicitudes, de modo que cualquier copia puede atender cualquier solicitud y pueden añadirse o retirarse copias libremente.

Arquitectura de microkernel
Estilo arquitectónico, también llamado arquitectura de plug-ins, en el que un pequeño sistema central proporciona el flujo de trabajo general y plug-ins independientes añaden o modifican comportamiento mediante contratos publicados.

Arquitectura de referencia
Arquitectura plantilla para un dominio o una plataforma que asigna las funciones típicas a un conjunto estándar de partes, y que los proyectos adaptan en lugar de diseñar desde cero.

Arquitectura distribuida
Una arquitectura formada por varias unidades desplegadas por separado que cooperan enviándose mensajes a través de una red.

Arquitectura en capas
Estilo que organiza los componentes en capas horizontales, cada una con un único papel técnico, donde cada capa usa solo las capas que tiene debajo.

Arquitectura monolítica
Arquitectura en la que toda la aplicación se construye y despliega como una sola unidad, sea cual sea su estructura interna.

Back pressure (contrapresión)
Una forma de que un consumidor lento haga que un productor rápido reduzca su ritmo, en lugar de dejar que los búferes y las colas crezcan sin límite.

Capa
Grupo de componentes con la misma responsabilidad técnica, que ofrece servicios a la capa superior y usa la capa inferior.

Capa abierta
Capa que la capa superior puede saltarse, normalmente una capa de servicios compartidos que solo necesitan algunas solicitudes.

Capa anticorrupción
Una capa de traducción que construye un contexto descendente para que el modelo de otro contexto o sistema no pueda filtrarse en el suyo.

Capa cerrada
Una capa por la que debe pasar toda solicitud: la capa superior no puede saltársela para llamar directamente a una capa inferior.

Capas de aislamiento
La propiedad de que un cambio dentro de una capa no se propaga a otras capas mientras su interfaz se mantenga igual; depende de que las capas sean cerradas.

Característica de arquitectura
Término de Richards y Ford para un atributo de calidad que la arquitectura debe soportar, como la escalabilidad, la desplegabilidad o la tolerancia a fallos.

Cola de mensajes fallidos (dead-letter queue)
Un lugar aparte donde se aparcan para su inspección los mensajes cuyo procesamiento falla repetidamente, para que no bloqueen el resto del flujo.

Contexto delimitado (bounded context)
Un límite explícito dentro del cual un modelo de dominio y su lenguaje ubicuo se aplican de forma coherente; fuera de él, las mismas palabras pueden significar otra cosa.

Contrato de plug-in
La interfaz publicada entre un sistema central y sus plug-ins: las operaciones, los tipos de datos, el comportamiento ante errores y la versión que ambas partes acuerdan.

Cuanto de arquitectura (architecture quantum)
Una parte de un sistema desplegable de forma independiente y con alta cohesión funcional, que incluye todo lo que necesita para ejecutarse, como su base de datos; el ámbito dentro del cual se aplica un mismo conjunto de características de arquitectura.

Cubo de escalado
Modelo de Abbott y Fisher con tres formas de escalar: eje X (copias idénticas detrás de un balanceador de carga), eje Y (división por función) y eje Z (división por datos, por ejemplo por cliente).

Diseño guiado por el dominio (DDD)
Un enfoque introducido por Eric Evans en 2003 que construye el software en torno a un modelo del dominio del negocio, desarrollado junto con expertos del dominio, y divide los dominios grandes en modelos delimitados explícitamente.

Dominio central (core domain)
El subdominio que da al negocio su ventaja competitiva y que, por tanto, merece a las mejores personas, el modelado más rico y código propiedad de la empresa.

Entidad
Un objeto de dominio definido por una identidad que perdura en el tiempo mientras cambian sus atributos, como un pedido o un cliente.

Estilo arquitectónico
Una familia de arquitecturas con nombre propio, definida por los tipos de elemento y conector que usa y por las restricciones sobre cómo pueden combinarse, como el estilo en capas o el de tuberías y filtros.

ETL
Extract, transform, load (extraer, transformar, cargar): un pipeline que toma datos de los sistemas de origen, los depura y les da nueva forma, y los carga en un almacén de destino como un almacén de datos.

Event storming
Un formato de taller creado por Alberto Brandolini en el que expertos del dominio y desarrolladores colocan eventos de dominio en una línea de tiempo para explorar un proceso de negocio y encontrar límites.

Evento de dominio
Un registro de algo significativo que ocurrió en el dominio del negocio, nombrado en tiempo pasado, como OrderPlaced o CourierAssigned.

Evento en proceso
Un mensaje publicado dentro de una aplicación en ejecución para que otros módulos puedan reaccionar a algo que ha ocurrido, sin que el publicador sepa quién escucha.

Filtro
Un paso de procesamiento autónomo que lee una entrada y luego la produce, la transforma, la comprueba o la consume, sin saber qué pasos van antes ni después de él.

Gran bola de barro (big ball of mud)
Un sistema sin estructura reconocible, en el que casi todo depende de todo lo demás; el nombre procede de Brian Foote y Joseph Yoder (1997).

Idempotente
Describe una operación que tiene el mismo efecto tanto si se aplica una vez como varias, de modo que puede reintentarse con seguridad.

Lenguaje publicado
Formato documentado y compartido para intercambiar información entre contextos, como un esquema de eventos versionado, que a menudo se ofrece mediante un servicio de host abierto.

Lenguaje ubicuo
Vocabulario compartido por los desarrolladores y los expertos del dominio y usado de forma coherente en las conversaciones, los documentos, las pruebas y el código dentro de un contexto delimitado.

Mapa de contextos
Una representación de los contextos delimitados de un sistema y de las relaciones entre ellos, como cliente-proveedor o capa anticorrupción, que abarca tanto la integración como las relaciones entre equipos.

MonolithFirst (primero el monolito)
Consejo de Martin Fowler de 2015 de empezar un sistema nuevo como un monolito bien modularizado y extraer servicios más adelante, cuando los límites se entiendan y hagan falta.

Monolito distribuido
Un sistema desplegado como servicios separados que están tan fuertemente acoplados que deben modificarse y lanzarse juntos, con lo que tiene los costos de la distribución sin sus beneficios.

Monolito modular
Aplicación desplegable única cuyo código está dividido en módulos según límites de negocio, cada uno con una interfaz pública explícita y sus propios datos.

Nivel (tier)
Límite físico en tiempo de ejecución, como un proceso o servidor separado. Una capa es un límite lógico en el código; varias capas pueden ejecutarse en un mismo nivel.

Núcleo compartido (shared kernel)
Pequeña parte de un modelo, y su código, que dos contextos acuerdan explícitamente compartir y que solo puede cambiarse con el consentimiento de ambos equipos.

Objeto de valor
Objeto de dominio definido solo por sus valores, sin identidad propia y normalmente inmutable, como una cantidad de dinero o una dirección.

Particionamiento por dominio
Dividir la estructura de nivel superior de un sistema por capacidad de negocio o subdominio, como pedidos, pagos y entregas.

Particionamiento técnico
Dividir la estructura de nivel superior de un sistema según el papel técnico, como interfaz de usuario, lógica de negocio y acceso a datos.

Patrón arquitectónico
Una solución documentada y reutilizable a un problema estructural recurrente en un contexto determinado, que nombra las partes, sus responsabilidades y cómo se relacionan. A menudo se usa como sinónimo de estilo.

Patrón de diseño
Una solución reutilizable a un problema recurrente a la escala de unas pocas clases u objetos, como Strategy u Observer; de alcance menor que un patrón arquitectónico.

Plug-in
Módulo construido por separado que añade comportamiento a una aplicación anfitriona mediante un contrato que publica el anfitrión, sin que cambie el código del anfitrión.

Prima de los microservicios (microservice premium)
Término de Martin Fowler (2015) para el costo y el riesgo adicionales que añaden los microservicios, que solo compensan cuando un sistema y una organización son lo bastante complejos.

Procesamiento de flujos (stream processing)
Procesar de forma continua un flujo ilimitado de registros, cada uno poco después de que llegue.

Procesamiento por lotes
Procesar un conjunto acotado de datos en una sola ejecución, como todos los pedidos de ayer, y terminar.

Proceso en segundo plano (background worker)
Un proceso que toma trabajos lentos o con picos, como correos electrónicos o informes, de una cola y los procesa fuera de la solicitud que los creó.

Prueba de arquitectura
Una prueba automatizada que hace fallar la compilación cuando el código rompe una regla estructural, como una dependencia prohibida entre capas o módulos; entre las herramientas están ArchUnit e import-linter.

Puertos y adaptadores (arquitectura hexagonal)
Estilo de Alistair Cockburn en el que una aplicación se comunica con el mundo exterior solo a través de puertos (interfaces que le pertenecen), y los adaptadores traducen entre cada puerto y una tecnología concreta.

Punto de entrada (empaquetado de Python)
Metadatos de un paquete de Python instalado que anuncian un objeto, como una clase de plugin, bajo un grupo con nombre para que otro código pueda descubrirlo en tiempo de ejecución.

Punto de extensión
Un hueco con nombre que declara una aplicación anfitriona o un plugin para que otros plugins puedan aportarle comportamiento, como en la plataforma Eclipse.

Raíz de composición
El único lugar, normalmente el código de arranque del programa, donde se crean los adaptadores concretos y se conectan a la aplicación.

Registro de plug-ins
La parte de un sistema central que registra qué plug-ins están disponibles, sus nombres, versiones y contratos, para que el núcleo pueda encontrarlos sin nombrarlos en el código.

Regla de dependencia
La regla de Robert C. Martin para la arquitectura limpia: las dependencias del código fuente apuntan solo hacia dentro, de modo que los anillos interiores nunca nombran nada declarado en los anillos exteriores.

Réplica de lectura
Copia de una base de datos que recibe los cambios del primario y atiende consultas de lectura, lo que reduce la carga del primario a costa de datos ligeramente desactualizados.

Repositorio
Objeto que carga y guarda agregados completos y que, para el dominio, parece una colección, ocultando cómo y dónde se almacenan.

Servicio de host abierto (open host service)
Patrón del mapa de contextos en el que un contexto upstream ofrece un único protocolo o API bien definido y documentado para todos sus consumidores.

Sistema central (core system)
En una arquitectura de microkernel, la parte mínima de la aplicación que necesita toda instalación: el flujo de trabajo principal, los servicios compartidos y los puntos donde se conectan los plugins.

Subdominio
Área diferenciada de la actividad de un negocio, como los menús, los pedidos, los pagos o el despacho. Los subdominios pertenecen al problema, exista o no software.

Subdominio de soporte
Subdominio específico del negocio y necesario para él, pero que no es donde compite; merece un diseño sólido pero más sencillo.

Subdominio genérico
Un subdominio cuyos problemas comparten muchas empresas y para el que existen buenas soluciones, como los pagos o la identidad; normalmente se compra o se construye de forma mínima.

Sumidero de arquitectura (architecture sinkhole)
Un antipatrón de los sistemas en capas en el que las solicitudes atraviesan capas que solo las reenvían, lo que añade código y retraso sin añadir comportamiento.

Tubería (pipe)
Canal unidireccional que transporta datos de un filtro al siguiente, normalmente con un búfer entre ellos.

Tuberías y filtros
Estilo en el que los datos fluyen por una cadena de pasos de procesamiento independientes (filtros) conectados por canales unidireccionales (tuberías).

Unidad de despliegue
Lo que se compila, se lanza y se inicia como una sola pieza: por ejemplo, un ejecutable, una imagen de contenedor o un archivo web.

Versionado semántico
Convención de numeración de versiones, MAJOR.MINOR.PATCH, en la que el número mayor cambia con los cambios incompatibles, el menor con las adiciones compatibles con versiones anteriores y el de parche con las correcciones.