BE SOFTWARE ARCHITECTURE WISE · CAPÍTULO 1

Qué es la arquitectura de software

Este capítulo explica qué es la arquitectura de software y por qué importa. Compara las definiciones estándar, separa la arquitectura del diseño, muestra mediante incidentes reales cómo las decisiones estructurales determinan el costo, la calidad y el riesgo, y describe lo que hacen los arquitectos. Después trata las partes interesadas y sus inquietudes, las decisiones y los compromisos, la deuda técnica y la erosión, y el lugar de la arquitectura en la entrega iterativa moderna. Está escrito para desarrolladores que avanzan hacia responsabilidades de arquitectura y para cualquier persona que trabaje con arquitectos, y solo presupone los bloques de construcción del Capítulo 0.

bewiseapps.com/es/software-architecture-wise/what-is-architecture/ · Versión 2026-10-26. Solo información educativa, no asesoramiento profesional.

DE UN VISTAZO

Aprenderás a

- Explicar las definiciones de arquitectura de software de ISO/IEC/IEEE 42010:2022 y del SEI, y las ideas que comparten.
- Distinguir las estructuras de módulos, de componentes y conectores y de asignación, y una arquitectura de su descripción.
- Clasificar decisiones como arquitectónicas o de diseño según su reversibilidad, su alcance y su efecto sobre los atributos de calidad.
- Analizar cómo contribuyeron las decisiones estructurales a incidentes reales como los de Knight Capital y HealthCare.gov.
- Describir las responsabilidades, las habilidades y los tipos de arquitecto, y cómo se mantienen los arquitectos en contacto con la práctica.
- Identificar a las partes interesadas y sus inquietudes, y hacer explícitas las inquietudes en conflicto.
- Evaluar las decisiones arquitectónicas como compromisos, teniendo en cuenta la reversibilidad y el momento oportuno.
- Explicar la deuda técnica, la erosión y la deriva, y cómo detectarlas y gestionarlas a lo largo del ciclo de vida.

Antes de empezar

El Capítulo 0, o un conocimiento práctico de los componentes, las bases de datos, las API, las redes y la nube.

Tiempo

Unas 2½ horas de lectura; 45 minutos para la prueba.

Términos clave

Arquitectura de software · Descripción de arquitectura · Estructura · Arquitectónicamente significativo · Atributo de calidad · Costo del cambio · Parte interesada · Inquietud · Compromiso · Reversibilidad · Deuda técnica · Erosión de la arquitectura · Función de aptitud (fitness function) · Ley de Conway

Por qué este capítulo va primero

Todo sistema de software tiene una arquitectura, la haya diseñado alguien o no. Es el conjunto de estructuras fundamentales que forman el sistema (sus partes principales, cómo se conectan y qué propiedades tienen) y de las decisiones significativas que las produjeron. Esas estructuras determinan mucho más de lo que la mayoría cree: lo rápido y fiable que es el sistema, cuánto cuesta ejecutarlo, cuántos equipos pueden trabajar en él a la vez y, sobre todo, lo caro que será cambiarlo el año que viene. Las funcionalidades son lo que hace un sistema; la arquitectura determina en gran medida lo bien que las hace y lo que cuesta hacer algo diferente.

El Capítulo 0 presentó los bloques de construcción: programas, almacenes de datos, redes, API, servidores y la nube. Este capítulo da un paso atrás y se pregunta qué significa organizar esos bloques de forma deliberada, quién lo hace y por qué merece la pena el esfuerzo. Te da el vocabulario del que depende el resto del curso y una forma de pensar que puedes aplicar a cualquier sistema, desde una pequeña app de reservas hasta una plataforma nacional de pagos.

Qué es la arquitectura y qué no es

Las dos primeras secciones tratan de definiciones, porque las definiciones confusas conducen a un trabajo confuso. La Sección 1.1 compara las definiciones que importan: la norma internacional ISO/IEC/IEEE 42010:2022, que habla de los conceptos o propiedades fundamentales de un sistema en su entorno y de los principios que rigen su evolución; la definición del Software Engineering Institute, que habla de las estructuras necesarias para razonar sobre un sistema, formadas por elementos, relaciones y sus propiedades; y las definiciones de los profesionales, que destacan el entendimiento compartido y las decisiones que son caras de cambiar. Muestra lo que tienen en común y presenta tres tipos de estructura que tiene todo sistema: cómo se divide su código, qué se ejecuta y qué se comunica con qué, y cómo se corresponde el software con la infraestructura y los equipos.

La Sección 1.2 aborda después la confusión más habitual: dónde termina la arquitectura y dónde empieza el diseño. La respuesta es que no hay un muro, solo un continuo ordenado por significancia. Una decisión es arquitectónicamente significativa cuando es costosa de revertir, tiene un alcance amplio o es decisiva para una cualidad clave, y por eso una convención modesta, como la forma de almacenar los importes de dinero, puede ser más arquitectónica que una caja grande en un diagrama. La sección también separa los niveles en los que se usa la palabra: arquitectura empresarial, de soluciones y de software, con la arquitectura de datos, de seguridad y de infraestructura atravesándolas.

Por qué importa y quién la hace

La Sección 1.3 expone por qué debe importarte. Muestra cómo los atributos de calidad, como la disponibilidad y el rendimiento, surgen de la estructura y no de líneas de código concretas, cómo crece el costo del cambio a medida que se construye más sobre una decisión, y cómo la arquitectura condiciona el riesgo, la organización de los equipos (mediante la ley de Conway), la comunicación y la estimación. Tres casos reales, contados a partir de fuentes primarias, sustentan el argumento: la pérdida bursátil de Knight Capital en agosto de 2012, el accidentado lanzamiento de HealthCare.gov en octubre de 2013 y la decisión de Amazon Prime

Video en 2023 de pasar un servicio de monitorización de pasos serverless distribuidos a un único proceso, lo que, según se informó, redujo su costo en más de un 90% e ilustra que la arquitectura trata de la adecuación, no de la moda.

La Sección 1.4 se centra en las personas. Describe lo que hacen realmente los arquitectos, a partir de las ocho expectativas que plantean Mark Richards y Neal Ford y de las tareas del plan de estudios iSAQB Foundation; los distintos tipos de arquitecto; por qué la arquitectura es un rol antes que un cargo; por qué los arquitectos valoran la amplitud de conocimientos; y el largo debate sobre si los arquitectos deberían seguir escribiendo código.

La Sección 1.5 amplía la mirada hacia las partes interesadas (todas las personas con un interés en el sistema, desde usuarios y desarrolladores hasta el personal de operaciones, los delegados de protección de datos y los reguladores) y sus inquietudes, y muestra cómo las inquietudes en conflicto se hacen explícitas en lugar de resolverse a favor de quien grita más fuerte. La Sección 1.6 pasa a las decisiones y los compromisos: la anatomía de una buena decisión, la diferencia entre las decisiones que se pueden revertir fácilmente y las que no, la idea de decidir en el último momento responsable y las trampas cognitivas que llevan por mal camino a personas capaces.

La arquitectura a lo largo del tiempo

Las dos últimas secciones tratan del tiempo. La Sección 1.7 explica la deuda técnica tal como la entendía originalmente Ward Cunningham en 1992, en qué se diferencia del simple desorden y cómo las arquitecturas se erosionan y se alejan de la forma prevista; muestra cómo detectar el daño y cómo explicarlo en términos de negocio. La Sección 1.8 sitúa la arquitectura en el ciclo de vida de la entrega: la vieja discusión entre el gran diseño inicial (big design up front) y el diseño puramente emergente, y el término medio práctico de la arquitectura dirigida por riesgos, continua y evolutiva, en la que un equipo hace la arquitectura justa, en el momento adecuado, y sigue comprobando que todavía encaja.

La idea que debes llevarte

Si recuerdas una sola cosa de este capítulo, que sea esta: la arquitectura es el conjunto de decisiones que es caro equivocar, y el trabajo del arquitecto es identificar esas decisiones, tomarlas de forma deliberada con las personas adecuadas, registrar por qué y mantenerlas en revisión, a la vez que hace que el mayor número posible de las demás decisiones sean baratas de cambiar. Todo lo demás en este curso, desde los atributos de calidad del Capítulo 2 hasta el diseño de sistemas con IA del Capítulo 12, es una aplicación de esa idea.

SECCIÓN 1.1 DE 1.8 · PRINCIPIANTE · 18 MIN DE LECTURA

Qué es la arquitectura de software: las definiciones estándar y lo que tienen en común

LA IDEA EN BREVE

Piensa en el edificio de un hospital. Su arquitectura no es el plano clavado en la pared; es cómo está organizado realmente el edificio: dónde están las salas, qué pasillos las conectan, dónde van los ascensores, qué puertas deben permanecer cerradas con llave y las reglas que todos siguen cuando lo amplían. Algunas decisiones, como dónde están los ascensores, costaría una fortuna cambiarlas; el color de las cortinas, no. La arquitectura de software es la misma idea aplicada a un sistema de software: sus partes principales, cómo se conectan, qué propiedades tienen y las decisiones caras de cambiar que hay detrás.

LO QUE NECESITAS SABER

- ISO/IEC/IEEE 42010:2022 define la arquitectura como los conceptos o propiedades fundamentales de una entidad en su entorno, más los principios que rigen su realización y su evolución.
- Bass, Clements y Kazman definen la arquitectura de software como el conjunto de estructuras necesarias para razonar sobre un sistema: los elementos de software, las relaciones entre ellos y las propiedades de ambos.
- Las definiciones de los profesionales añaden que la arquitectura es el entendimiento compartido de los desarrolladores expertos (Ralph Johnson) y las decisiones significativas, donde la significancia se mide por el costo del cambio (Grady Booch).
- Las definiciones comparten siete ideas: estructuras, elementos, relaciones, propiedades, decisiones, significancia y entorno.
- Los sistemas tienen tres tipos de estructura: estructuras de módulos (código), estructuras de componentes y conectores (ejecución) y estructuras de asignación (correspondencia con la infraestructura, los equipos y los archivos).
- Todo sistema tiene una arquitectura, se haya diseñado deliberadamente o no y esté documentada o no; una descripción de arquitectura es un producto de trabajo independiente que la expresa.

Por qué hace falta definir la palabra

Pregunta a diez ingenieros con experiencia qué es la arquitectura de software y oirás diez respuestas distintas: el diagrama de cajas y flechas de la pared, la elección del proveedor de nube, el esquema de la base de datos, la división en servicios, las reglas sobre qué código

puede llamar a cuál o, simplemente, "las decisiones difíciles". Cada respuesta capta algo real y, por sí sola, a cada una le falta algo. Por eso, un curso que va a enseñar arquitectura tiene que empezar por precisar de qué está hablando.

Esto importa por razones prácticas. Si un equipo cree que la arquitectura es "el diagrama", dejará de prestarle atención una vez dibujado el diagrama, aunque el sistema siga cambiando por debajo. Si cree que la arquitectura es "el stack tecnológico", discutirá sobre frameworks mientras ignora cómo se reparten las responsabilidades y cómo fluyen los datos. Una buena definición te dice qué mirar, qué proteger y qué hablar con los demás.

Las definiciones que siguen proceden de una norma internacional, del Software Engineering Institute (SEI) de la Universidad Carnegie Mellon y de profesionales del sector. Se escribieron con fines distintos, pero coinciden más de lo que discrepan, y los puntos en los que coinciden son el núcleo de la disciplina.

La definición estándar: ISO/IEC/IEEE 42010:2022

ISO/IEC/IEEE 42010 es la norma internacional para describir arquitecturas. Su edición actual, publicada en noviembre de 2022, sustituyó a la edición de 2011. Define la arquitectura, en una paráfrasis fiel, como los conceptos o propiedades fundamentales de una entidad en su entorno, junto con los principios que rigen la realización y la evolución de esa entidad y sus procesos del ciclo de vida relacionados.

Tres rasgos de esta definición merecen atención. Primero, habla de una *entidad*, no solo de un sistema de software. La edición de 2022 amplió deliberadamente el alcance para que las mismas ideas se apliquen al software, a los sistemas de sistemas, a los productos, a las líneas de productos y a empresas enteras. Segundo, sitúa la entidad *en su entorno*: no puedes entender una arquitectura sin los usuarios, los otros sistemas, la organización, las leyes y los presupuestos que la rodean. Tercero, incluye *principios que rigen* cómo se construye la entidad y cómo cambia. La arquitectura no es solo una instantánea de la estructura; incluye las reglas que mantienen la estructura coherente a medida que el sistema evoluciona, como "cada servicio es dueño de sus propios datos" o "el código del dominio nunca depende del framework web".

La norma también traza una línea clara entre una arquitectura y una descripción de arquitectura. La arquitectura es el conjunto de propiedades fundamentales que el sistema tiene realmente. La descripción es un producto de trabajo (documentos, modelos, diagramas, registros de decisiones) que la expresa. ISO/IEC/IEEE 42010 es sobre todo una norma sobre descripciones: dice qué debe contener una buena descripción, como las partes interesadas

identificadas, sus inquietudes y las vistas que abordan esas inquietudes. No te dice cómo diseñar una arquitectura. La Sección 1.5 vuelve a las partes interesadas y las inquietudes, y el Capítulo 4 trata las vistas.

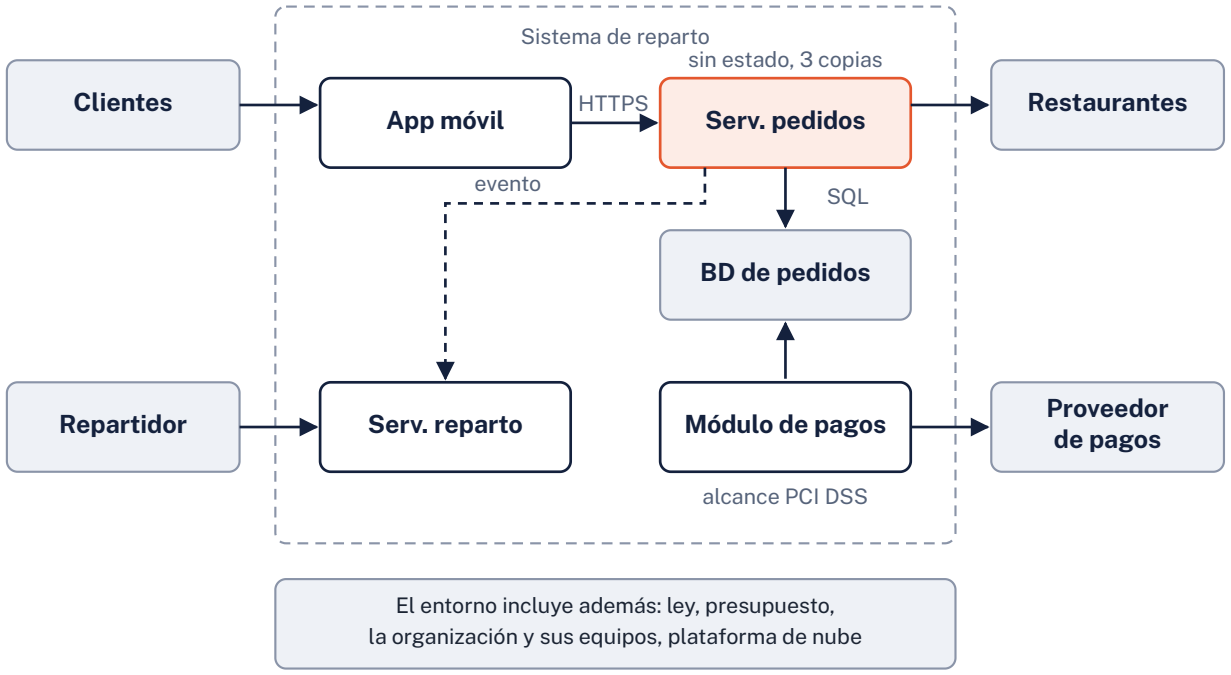
La definición del SEI: estructuras, elementos, relaciones y propiedades

La definición más enseñada procede de *Software Architecture in Practice*, de Len Bass, Paul Clements y Rick Kazman (4.ª edición, Addison-Wesley, 2021), cuyos autores trabajaron en el SEI. En sus palabras, parafraseadas: la arquitectura de software de un sistema es el conjunto de estructuras necesarias para razonar sobre el sistema, y esas estructuras constan de elementos de software, las relaciones entre ellos y las propiedades de ambos.

Cada parte de esa frase cumple una función. **Estructuras**, en plural: un sistema tiene muchas estructuras a la vez (sus módulos de código, sus procesos en ejecución, su despliegue en máquinas) y ninguna de ellas es por sí sola "la" arquitectura. **Necesarias para razonar sobre el sistema**: la prueba de si algo es arquitectónico es si lo necesitas para responder preguntas importantes, como "¿Aguantará esto la carga máxima?" o "¿Pueden dos equipos trabajar en esto de forma independiente?". Los **elementos** son las partes: módulos, servicios, bases de datos, colas. Las **relaciones** son cómo se conectan: *usa*, *llama a*, *publica en*, *se despliega en*. Las **propiedades** son lo que hace que los elementos y las relaciones se comporten como lo hacen: un servicio no tiene estado, una llamada es síncrona con un timeout de dos segundos, una base de datos contiene datos personales y debe permanecer en la UE.

La expresión sobre el razonamiento también explica lo que *no* es arquitectura. El nombre de una variable local, el algoritmo interno de una rutina de ordenación o la disposición de una pantalla de configuración normalmente no te ayudan a razonar sobre el sistema en su conjunto, así que pertenecen al diseño detallado y no a la arquitectura. La Sección 1.2 examina dónde cae esa línea.

Elementos, relaciones, propiedades y entorno



Elementos (cajas) conectados por relaciones (flechas), con sus propiedades escritas al lado, dentro de un límite del sistema situado en un entorno más amplio.

Definiciones de los profesionales: entendimiento compartido, lo importante y decisiones

Los profesionales han propuesto definiciones más breves que ponen el acento en las personas y las decisiones más que en las estructuras. Martin Fowler ha popularizado dos de Ralph Johnson (uno de los cuatro autores del libro *Design Patterns* de 1994). La primera es que la arquitectura es el entendimiento compartido que tienen los desarrolladores expertos de cómo está diseñado el sistema. La segunda, de la misma discusión, es deliberadamente abierta: “La arquitectura trata de lo importante. Sea lo que sea.” Johnson también sugirió que la arquitectura es el conjunto de decisiones que te gustaría acertar al principio de un proyecto, no porque sea más probable acertarlas al principio, sino porque equivocarse en ellas sale muy caro.

Grady Booch, uno de los creadores de UML, lo planteó en términos económicos: la arquitectura está formada por las decisiones de diseño significativas que dan forma a un sistema, donde la significancia se mide por el costo del cambio. Eso convierte una idea abstracta en una prueba utilizable. Elegir el color de un botón es barato de revertir. Elegir entre una base de datos

relacional y un almacén de documentos, o entre una unidad desplegable y cuarenta, es caro de revertir una vez que los datos, el código y los equipos han crecido a su alrededor. Booch también observó que toda arquitectura es diseño, pero no todo diseño es arquitectura.

Estas definiciones añaden algo que las estructurales dejan implícito. La arquitectura vive en parte en la cabeza de las personas; un sistema cuyos ingenieros sénior comparten una imagen clara de cómo funciona tiene una arquitectura más coherente que uno en el que cada cual tiene una imagen distinta, aunque ninguno de los dos tenga un diagrama. Y la arquitectura se forma a partir de decisiones, muchas de las cuales implican un compromiso: elegir una opción significa renunciar a las ventajas de otra. La Sección 1.6 estudia en detalle las decisiones y los compromisos.

Comparación de las definiciones

Fuente	Idea central (parafraseada)	Qué destaca	Más útil para
ISO/IEC/IEEE 42010:2022	Conceptos o propiedades fundamentales de una entidad en su entorno, más los principios que rigen su realización y su evolución	Entorno, principios, evolución; la arquitectura es distinta de su descripción	Contratos, trabajo de normalización, describir arquitecturas de forma coherente en toda una organización
Bass, Clements y Kazman (2021)	El conjunto de estructuras necesarias para razonar sobre el sistema: elementos, relaciones y propiedades de ambos	Múltiples estructuras; razonar sobre las cualidades	Análisis y diseño: preguntarse si una estructura cumplirá sus objetivos
Ralph Johnson, a través de Martin Fowler	El entendimiento compartido de los desarrolladores expertos; lo importante, sea lo que sea	Las personas, la comunicación, el criterio sobre lo que es importante	La práctica del equipo: mantener vivo un modelo mental común
Grady Booch	Las decisiones de diseño significativas que dan forma a un sistema, con la significancia medida por el costo del cambio	Las decisiones y su economía	Decidir dónde invertir el esfuerzo de análisis

Lo que comparten las definiciones

Leídas en paralelo, las definiciones convergen en siete ideas, y juntas forman una definición de trabajo para el resto de este curso.

Siete ideas compartidas

1. **Estructuras**
   Un sistema tiene varias estructuras a la vez, y cada una lo muestra desde un ángulo distinto. Ningún diagrama por sí solo es toda la arquitectura.

2. **Elementos**
   Las partes que forman esas estructuras: módulos, componentes, servicios, almacenes de datos, colas, servidores.

3. **Relaciones**
   Cómo se conectan los elementos y cómo dependen unos de otros: llamadas, mensajes, datos compartidos, despliegue, propiedad.

4. **Propiedades**
   Las características visibles desde fuera de los elementos y las relaciones, como no tener estado, estar cifrado, ser asíncrono o estar replicado, que juntas determinan los atributos de calidad del sistema.

5. **Decisiones**
   Las elecciones que produjeron las estructuras, con las razones y las alternativas descartadas que hay detrás. Sin las razones, más adelante la gente deshace buenas decisiones por accidente.

6. **Significancia**
   Solo algunas cosas son arquitectónicas: las que son costosas de cambiar, afectan a muchas partes o determinan una cualidad clave. La arquitectura es selectiva por definición.

7. **Entorno**
   El sistema se sitúa entre usuarios, otros sistemas, una organización, leyes y presupuestos. Todo ello da forma a la arquitectura y, a su vez, la arquitectura le da forma a ello.

Tres tipos de estructura

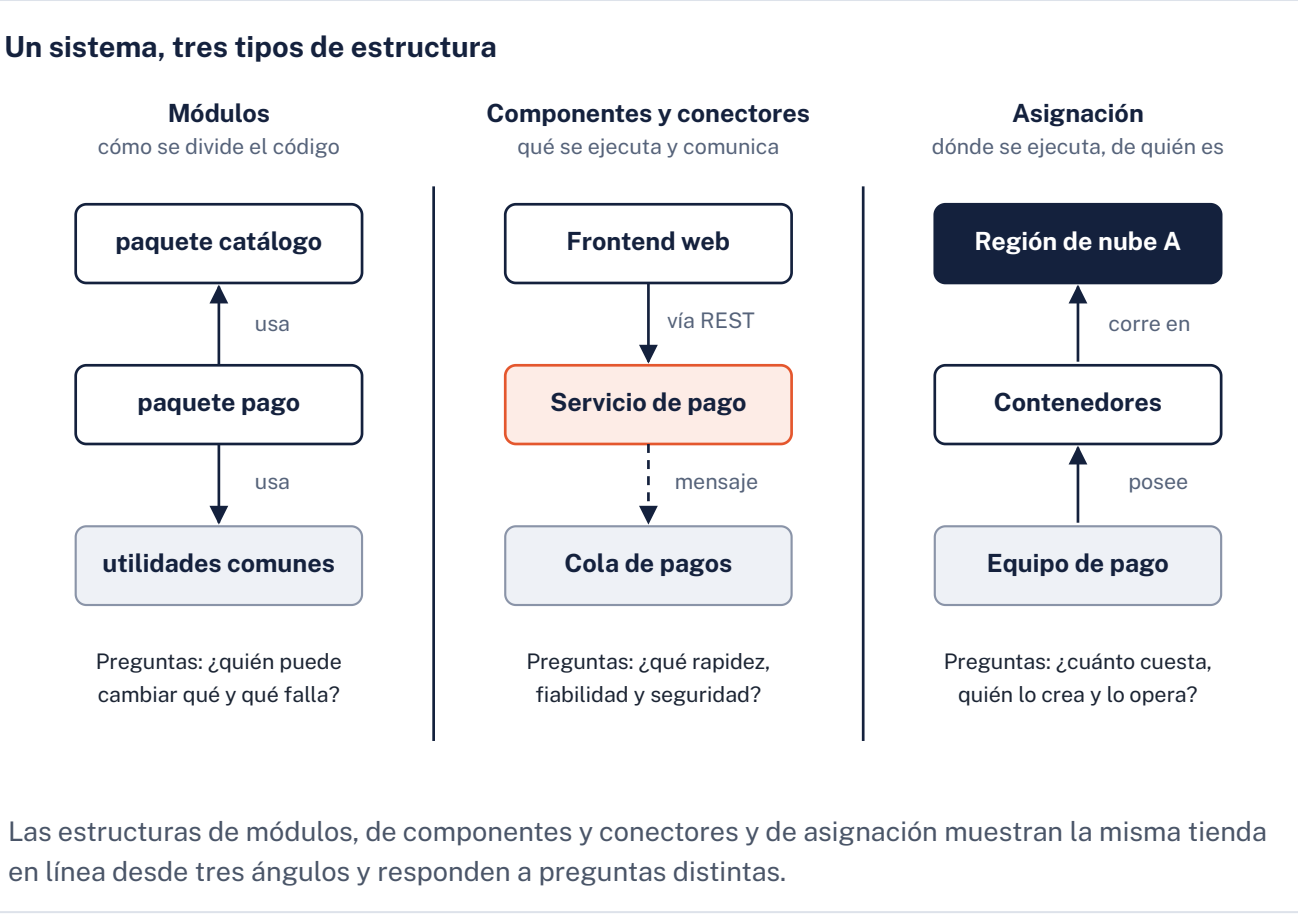
El trabajo del SEI sobre documentación, publicado como *Documenting Software Architectures: Views and Beyond* (Clements y colaboradores, 2.ª edición, 2010), agrupa las estructuras en tres tipos, y *Software Architecture in Practice* usa la misma agrupación. Conocer los tres tipos te protege del error más común en las discusiones de arquitectura: hablar de un tipo de estructura mientras la otra persona está pensando en otro.

Las **estructuras de módulos** muestran cómo se divide el código en unidades de implementación, como paquetes, bibliotecas y espacios de nombres, y cómo dependen esos módulos unos de otros. Existen en tiempo de desarrollo. Responden a preguntas como: ¿qué equipo puede cambiar qué sin romper el trabajo de los demás, y qué código se puede reutilizar?

Las **estructuras de componentes y conectores** muestran lo que existe cuando el sistema se ejecuta: procesos, servicios, clientes, bases de datos y los conectores entre ellos, como las llamadas HTTP, las colas de mensajes y los flujos de eventos. Responden a preguntas de tiempo de ejecución: ¿qué rapidez tiene una solicitud, qué ocurre cuando falla una parte, dónde se guardan los datos?

Las **estructuras de asignación** hacen corresponder el software con elementos ajenos al software: con el hardware y la infraestructura en la nube (despliegue), con los equipos (asignación del trabajo) y con los archivos y repositorios (implementación). Responden a preguntas sobre costos, operaciones y organización: ¿dónde se ejecuta, quién lo construye, a quién despiertan de madrugada cuando falla?

Un mismo elemento puede aparecer en las tres. Una funcionalidad de pago es un *paquete* en la estructura de módulos, un *servicio* en ejecución en la estructura de componentes y conectores, y un conjunto de *contenedores en una región de nube cuyo responsable es el equipo de pago* en las estructuras de asignación. La correspondencia entre estructuras no siempre es de uno a uno: un solo módulo puede desplegarse como varios servicios, o varios módulos pueden ejecutarse en un mismo proceso. Es justo en ese desajuste donde se esconden muchas sorpresas.



Las estructuras de módulos, de componentes y conectores y de asignación muestran la misma tienda en línea desde tres ángulos y responden a preguntas distintas.

Todo sistema tiene una arquitectura

Como la arquitectura consiste en estructuras y en las propiedades que estas tienen realmente, todo sistema de software tiene una arquitectura, la haya diseñado alguien a propósito o no, y esté documentada o no. Una macro de hoja de cálculo que acabó convirtiéndose en una herramienta crítica para el negocio tiene una arquitectura. También la tiene la base de código de una startup escrita a toda prisa por tres personas. La pregunta nunca es si un sistema tiene arquitectura, sino si esa arquitectura es adecuada para su propósito, la entienden las personas que trabajan en ella y se ha moldeado de forma deliberada.

Esto lleva a una distinción útil. La arquitectura *prevista* (o prescriptiva) es lo que el equipo cree o ha decidido que debe ser la estructura: el diagrama de la wiki, las reglas acordadas. La arquitectura *real* (o descriptiva) es lo que hacen de verdad el código y la infraestructura. Las dos se van separando con el tiempo, a menos que alguien las mantenga alineadas, y ese desajuste es una de las principales fuentes de deuda técnica, que se trata en la Sección 1.7. Una arquitectura sin documentar no está ausente; simplemente es más difícil de ver, de razonar sobre ella y de cambiar con seguridad.

DE DÓNDE VIENE

De dónde vienen las definiciones

La arquitectura de software surgió como disciplina con nombre propio a principios de la década de 1990. El artículo de Dewayne Perry y Alexander Wolf de 1992, *Foundations for the Study of Software Architecture*, propuso que una arquitectura consta de elementos, su forma y la justificación que hay detrás, y el trabajo de David Garlan y Mary Shaw, que culminó en su libro de 1996 *Software Architecture: Perspectives on an Emerging Discipline*, catalogó estilos arquitectónicos recurrentes. La primera edición de *Software Architecture in Practice* apareció en 1998, y IEEE 1471, el antecesor de ISO/IEC/IEEE 42010, se publicó en 2000.

Las ideas son aún más antiguas. El sistema operativo THE de Edsger Dijkstra, organizado en capas (1968), y el artículo de David Parnas de 1972 sobre cómo dividir los sistemas en módulos ocultando decisiones de diseño son ambos pensamiento arquitectónico anterior al uso de la palabra.

EJEMPLO PRÁCTICO

Ejemplo práctico: aplicar las definiciones a una app de reparto de comida

Una empresa de reparto de comida pide a una arquitecta recién incorporada que "nos diga cuál es nuestra arquitectura". Hay un único diagrama desactualizado. Usando las ideas compartidas como lista de comprobación, la arquitecta elabora una primera descripción en una semana.

Elementos	App móvil, aplicación web para restaurantes, servicio de pedidos, servicio de asignación de repartidores, módulo de pagos, una base de datos PostgreSQL compartida, un intermediario de mensajes (message broker) y un proveedor externo de pagos con tarjeta.
Relaciones	Las apps llaman al servicio de pedidos mediante HTTPS; el servicio de pedidos escribe en la base de datos compartida y publica eventos de "pedido realizado"; la asignación de repartidores consume esos eventos; el módulo de pagos llama a la API del proveedor.
Propiedades	El servicio de pedidos no tiene estado y se ejecuta como tres copias; la asignación de repartidores guarda el estado en memoria y se ejecuta como una sola copia; la base de datos contiene datos personales y tokens de tarjetas.
Entorno	Dos equipos de desarrollo, un regulador de seguridad alimentaria, la legislación de protección de datos, un presupuesto fijo de nube y restaurantes con tabletas poco fiables.
Decisiones significativas encontradas	Una sola base de datos compartida por todos los servicios; una sola copia de la asignación de repartidores; eventos para el envío de repartidores, pero llamadas directas para los pagos.
Prevista frente a real	El diagrama antiguo muestra bases de datos separadas para cada servicio. En realidad, el servicio de repartidores lee directamente las tablas de pedidos, así que los dos servicios no se pueden cambiar de forma independiente.

El ejercicio saca a la luz dos riesgos que nadie había señalado: la única copia de la asignación de repartidores es un punto único de fallo, y la base de datos compartida acopla a dos equipos.

Ninguno de los dos riesgos era visible en el código de ningún servicio concreto. Ambos son propiedades de estructuras y relaciones, y por eso cuentan como arquitectura.

La descripción llevó una semana porque nadie había registrado las decisiones ni sus razones. Registrarlas es el tema del Capítulo 4.

ATENCIÓN

Ideas erróneas habituales

- **"La arquitectura es el diagrama."** Un diagrama es una descripción parcial de una sola estructura. La arquitectura es el conjunto de propiedades que el sistema tiene realmente.
- **"La arquitectura es el stack tecnológico."** Las elecciones tecnológicas pueden ser arquitectónicas, pero cómo se reparten las responsabilidades, los datos y las dependencias suele importar más que el framework que se use.
- **"La arquitectura es solo para sistemas grandes."** Los sistemas pequeños también tienen arquitectura; simplemente tienen menos decisiones significativas, así que hace falta menos esfuerzo para acertar con ellas.
- **"La arquitectura se decide una vez, al principio."** Todas las definiciones incluyen la evolución. La arquitectura se moldea a lo largo de toda la vida de un sistema, como explica la Sección 1.8.

Cómo usan las definiciones los profesionales

En la práctica, los arquitectos con experiencia rara vez recitan una definición, pero usan cada una como una lente. La visión de ISO les recuerda que deben mirar hacia fuera, al entorno, y separar lo que el sistema *es* de lo que *dicen* los documentos. La visión del SEI les da un vocabulario de estructuras, elementos, relaciones y propiedades para el análisis, y una prueba ("¿necesitamos esto para razonar sobre el sistema?"). La visión de Johnson les recuerda que un modelo mental compartido en todo el equipo es en sí mismo un activo que hay que mantener. La visión de Booch les dice dónde dedicar su tiempo limitado: a las decisiones que serían caras de revertir.

De ahí se deriva una regla práctica. Cuando no estés seguro de si algo es arquitectónico, hazte tres preguntas. ¿Sería caro o arriesgado cambiarlo más adelante? ¿Afecta a varias partes del sistema o a varios equipos? ¿Determina si el sistema cumple un objetivo de calidad importante, como la velocidad, la disponibilidad o la seguridad? Si la respuesta a cualquiera de ellas es sí, trátalo como arquitectura: decídelo de forma deliberada, registra las razones y asegúrate de que las personas afectadas lo entienden.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Cuál es la diferencia entre una arquitectura y una descripción de arquitectura?

La arquitectura es el conjunto de propiedades y estructuras fundamentales que el sistema tiene realmente. La descripción es un producto de trabajo, como diagramas y documentos, que la expresa, y puede estar incompleta o desactualizada.

▼ ¿Qué tipo de estructura examinarías para averiguar a qué equipo despiertan cuando falla un servicio?

Una estructura de asignación, en concreto la correspondencia de asignación del trabajo o de propiedad entre el software y los equipos.

▼ Según la prueba de Booch, ¿es arquitectónica la elección de la biblioteca de logging?

Normalmente no, si está oculta tras una interfaz pequeña y es barato sustituirla. Se vuelve arquitectónica si todos los módulos la llaman directamente, lo que hace que sea caro cambiarla.

▼ ¿Por qué menciona el entorno la definición de ISO?

Porque una arquitectura solo se puede juzgar en relación con los usuarios, los sistemas, la organización, las leyes y los presupuestos que la rodean; la misma estructura puede ser buena en un entorno y mala en otro.

FUENTES

- ISO/IEC/IEEE 42010:2022 Software, systems and enterprise: Architecture description (ISO): https://www.iso.org/standard/74393.html
- Software Architecture in Practice, Fourth Edition (SEI, Carnegie Mellon University): https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/
- Software Architecture Guide (Martin Fowler): https://martinfowler.com/architecture/
- Documenting Software Architectures: Views and Beyond, Second Edition (SEI): https://www.sei.cmu.edu/library/documenting-software-architectures-views-and-beyond-second-edition/

SECCIÓN 1.2 DE 1.8 · PRINCIPIANTE · 17 MIN DE LECTURA

Arquitectura, diseño y código: significancia, alcance y niveles

LA IDEA EN BREVE

Cuando se proyecta una casa, algunas decisiones fijan todo lo demás: cuántas plantas tendrá, dónde va la escalera, dónde se levantan los muros de carga. Otras se pueden cambiar en un fin de semana: la pintura, las estanterías, las lámparas. Ambas son diseño, pero solo las primeras son arquitectura, porque cambiarlas después significa hacer obras de envergadura en la casa. El software es parecido. Elegir cómo se divide un sistema en partes, o cómo se almacenan los importes de dinero en todas partes, es arquitectura, porque es costoso de deshacer. Elegir cómo se escribe una función dentro de una parte es diseño.

LO QUE NECESITAS SABER

- La arquitectura y el diseño forman un continuo: toda arquitectura es diseño, pero solo las decisiones de diseño significativas son arquitectura.
- Una decisión es arquitectónicamente significativa cuando es difícil o costosa de revertir, tiene un alcance amplio, determina un atributo de calidad clave, restringe otras decisiones o implica compromisos importantes.
- El nivel de abstracción por sí solo es una mala prueba: convenciones de bajo nivel, como los formatos de los identificadores o de los importes de dinero, pueden ser arquitectónicas, y algunas cajas de alto nivel son triviales.
- Las decisiones estratégicas marcan la dirección a largo plazo para muchas personas; las decisiones tácticas resuelven problemas acotados e inmediatos, pero las elecciones tácticas repetidas pueden volverse arquitectónicas.
- La arquitectura empresarial abarca toda una organización (TOGAF es el marco más conocido), la arquitectura de soluciones un problema de negocio que atraviesa varios sistemas, y la arquitectura de software un sistema; la arquitectura de datos, de seguridad y de infraestructura atraviesan todos los niveles.
- Las decisiones deben tomarse en el nivel más bajo que tenga suficiente contexto y responsabilidad, con una consulta más amplia para las decisiones que afectan a muchos equipos.

Un continuo, no un muro

Si la arquitectura es el conjunto de decisiones y estructuras significativas (Sección 1.1), entonces el diseño es todo lo demás: las numerosas decisiones más pequeñas sobre cómo funciona cada parte por dentro. La dificultad es que no hay una línea nítida entre ambos.

Decidir dividir una plataforma de reparto de comida en un sistema de pedidos y un sistema de repartidores es claramente arquitectónico. Ponerle nombre a una variable claramente no lo es. Entre esos extremos hay un amplio terreno intermedio de decisiones (el formato de error de una API, una regla de caché, cómo se divide un módulo en paquetes) cuya naturaleza depende del contexto.

Grady Booch lo expresó con claridad: toda arquitectura es diseño, pero no todo diseño es arquitectura. La arquitectura y el diseño detallado son puntos de un mismo continuo de decisiones de diseño, ordenados según lo significativos que son. Por eso la pregunta más útil no es "¿esto es arquitectura o diseño?" en abstracto, sino "¿qué significancia tiene esta decisión aquí y cuánto cuidado merece?".

Una segunda forma popular de trazar la línea es por nivel de abstracción: la arquitectura es la vista de alto nivel ("cajas y flechas") y el diseño es la vista de bajo nivel ("clases y métodos"). Es una primera aproximación razonable, pero falla en ambas direcciones. Un detalle de bajo nivel puede ser arquitectónico: el formato de un identificador compartido por veinte servicios, o la elección de una convención de zona horaria para las fechas almacenadas, puede costar meses cambiarlo. Y una caja de alto nivel puede ser trivial: que una herramienta interna de administración sea un servicio aparte puede no tener ninguna importancia.

Qué hace que una decisión sea arquitectónicamente significativa

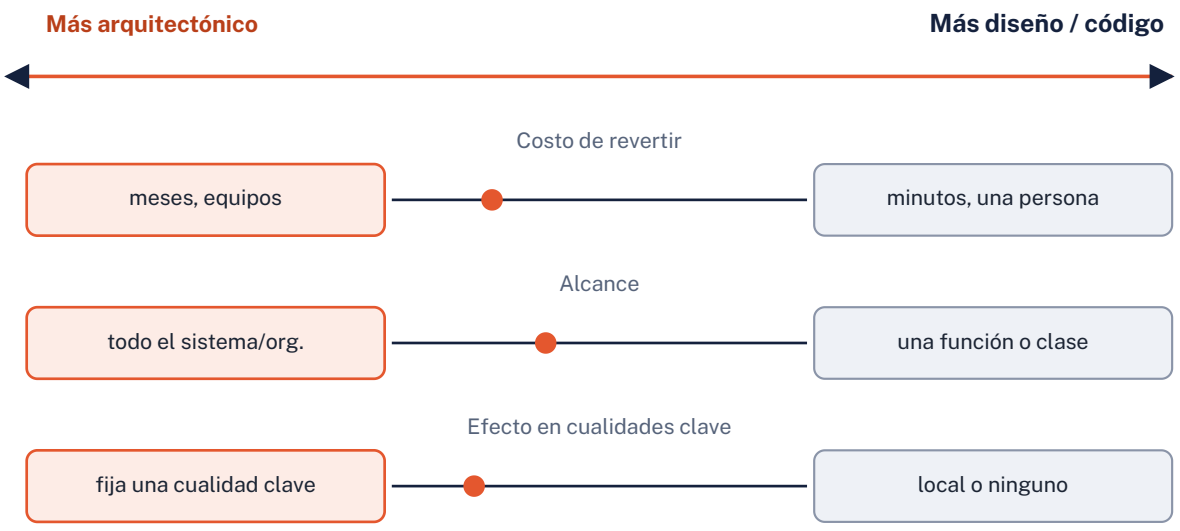
Una decisión es arquitectónicamente significativa cuando tiene una o varias de las características siguientes. No son reglas formales de una sola norma; resumen lo que dicen con distintas palabras el SEI, Martin Fowler, Mark Richards y Neal Ford, entre otros.

Señales de que una decisión es arquitectónicamente significativa

Característica	Pregunta que hay que hacerse	Ejemplo
Difícil o costosa de revertir	Si dentro de un año resulta errónea, ¿cuánto costará cambiarla?	Elegir un modelo de datos basado en event sourcing en lugar de almacenar el estado actual
Alcance amplio	¿A cuántos componentes, equipos o sistemas afecta?	Definir el formato de los identificadores de cliente que se usan en todos los servicios
Determina una cualidad clave	¿Decide si cumplimos un objetivo de rendimiento, disponibilidad, seguridad o costo?	Poner una caché delante del catálogo de productos para cumplir un objetivo de respuesta de 200 ms
Impone una restricción a otros	¿Limita lo que pueden elegir otros equipos?	Exigir que todos los servicios publiquen eventos en un único intermediario de mensajes compartido
Implica compromisos significativos	¿Renunciamos a algo importante para obtener otra cosa?	Aceptar la consistencia eventual entre pedidos y existencias para mantener disponible el proceso de pago
Mucha novedad o mucho riesgo	¿Alguien aquí lo ha hecho antes? ¿Y si la tecnología nos falla?	Adoptar una base de datos que la organización nunca ha operado en producción

Mark Richards y Neal Ford, en *Fundamentals of Software Architecture*, describen el espectro entre arquitectura y diseño en términos parecidos: una decisión se sitúa más hacia la arquitectura cuanto más estratégica es (a largo plazo, con muchas personas implicadas), cuanto más esfuerzo costaría cambiarla y cuanto más significativos son sus compromisos. Martin Fowler ofrece una versión compacta: la arquitectura es lo que es difícil de cambiar, lo que también significa que uno de los objetivos de un arquitecto es hacer que más cosas sean fáciles de cambiar. Hacer que una decisión sea barata de revertir, por ejemplo poniéndola detrás de una interfaz, la desplaza en la práctica de la arquitectura hacia el diseño. Esta idea, la reversibilidad, se estudia con más detalle en la Sección 1.6.

Situar una decisión en el continuo



Puntos: una decisión (formato de mensajes) situada en cada dimensión

Tres dimensiones sitúan una decisión entre la arquitectura y el diseño; una decisión puede puntuar de forma distinta en cada una.

Decisiones estratégicas y tácticas

Una distinción relacionada es la que separa las decisiones estratégicas de las tácticas. Las decisiones **estratégicas** marcan la dirección durante mucho tiempo y para muchas personas: adoptar un proveedor de nube, elegir entre un monolito y servicios distribuidos, decidir que los datos de los clientes se gestionarán como maestros en un único sistema. Normalmente necesitan la opinión de varias partes interesadas y vale la pena ponerlas por escrito con sus razones. Las decisiones **tácticas** resuelven un problema inmediato dentro de un área acotada: qué biblioteca usar para analizar fechas en un servicio, cómo estructurar una consulta concreta a la base de datos, cómo nombrar los endpoints de una API interna.

Estratégico no significa lento. Muchas decisiones estratégicas pueden y deben tomarse con rapidez una vez que las opciones están claras, y algunas son reversibles si se estructuran bien. Táctico no significa poco importante: una mala decisión táctica repetida en cincuenta lugares (por ejemplo, que cada servicio escriba su propia lógica de reintento sin espera progresiva, o back-off) puede acabar sumando un problema arquitectónico. Los arquitectos vigilan las decisiones tácticas que están a punto de volverse estratégicas por repetición y las convierten en un principio compartido o en una capacidad de la plataforma.

¿Arquitectura o diseño? Decisiones de ejemplo, razonadas

Decisión (para una tienda en línea)	Veredicto	Razonamiento
Dividir el sistema en servicios desplegables por separado para el catálogo, el proceso de pago y la entrega	Arquitectónica	Afecta a todos los equipos, al despliegue, a la propiedad de los datos y al comportamiento ante fallos; muy costosa de revertir.
Usar PostgreSQL en lugar de una base de datos documental como almacén principal de los pedidos	Arquitectónica	Condiciona el modelo de datos, las transacciones y las habilidades de operación; migrar más adelante es caro y arriesgado.
Usar una biblioteca JSON concreta dentro del servicio de pago	Diseño	Es local a un servicio y se puede sustituir en un día si está oculta tras el propio código del servicio.
Exigir que todos los servicios emitan logs estructurados con un ID de correlación compartido	Arquitectónica	Una inquietud transversal (cross-cutting concern): solo funciona si todos la siguen, y determina la rapidez con la que se pueden diagnosticar los incidentes.
Poner nombre al método que calcula las tarifas de entrega	Diseño (código)	Puramente local; las herramientas automatizan el cambio de nombre.
Elegir el formato de los números de pedido que se muestran a los clientes y se comparten con el socio de almacén	Arquitectónica	Es de bajo nivel, pero cruza los límites del sistema y de la empresa y se imprime en los paquetes; cambiarlo requiere coordinarse con un tercero.
Usar el patrón Strategy para admitir varias reglas de tarifas de entrega	Diseño	Condiciona el funcionamiento interno de un componente; los demás componentes no se ven afectados.
Almacenar en caché los precios de los productos durante 60 segundos	Depende	Diseño si los precios cambian rara vez y que estén desactualizados no causa daño; arquitectónica si un precio desactualizado crea un problema legal o financiero, porque entonces determina una cualidad clave (la corrección).

Niveles de arquitectura

La palabra arquitectura también se usa en varios niveles de alcance, y los cargos siguen esos niveles. Confundirlos provoca muchos malentendidos: un arquitecto empresarial y un arquitecto de software pueden ser ambos "el arquitecto" y, sin embargo, su trabajo apenas se solapa.

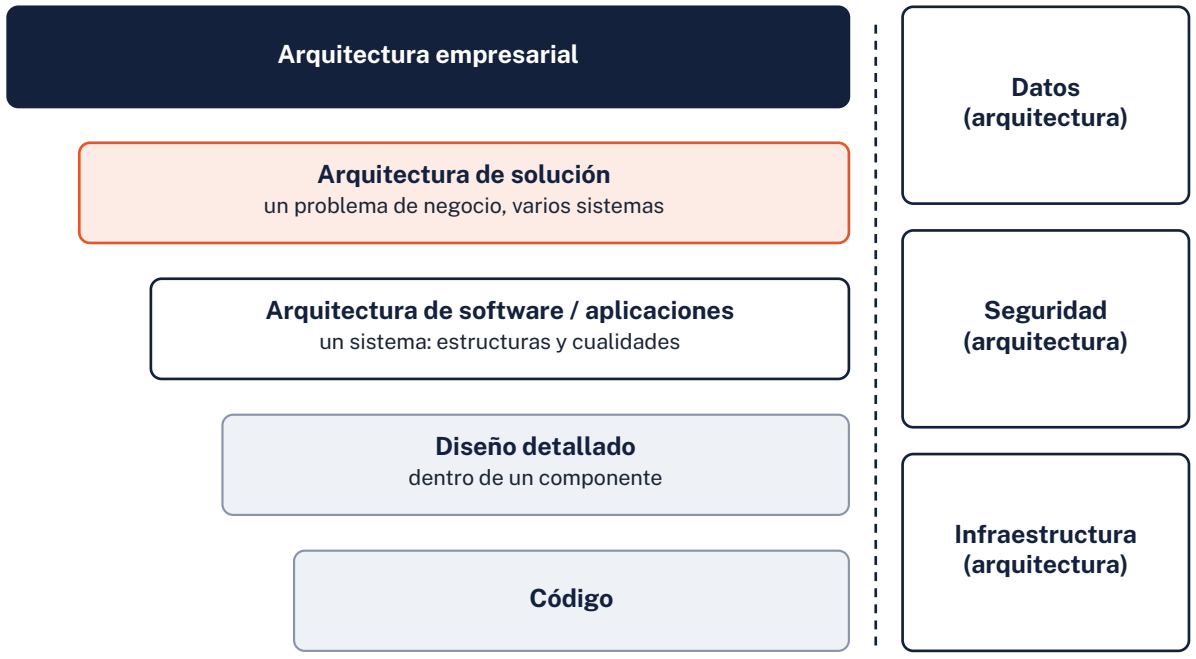
La **arquitectura empresarial** abarca toda una organización: sus capacidades de negocio (lo que debe ser capaz de hacer, como "evaluar siniestros" o "dar de alta a clientes"), la información que posee, su cartera de aplicaciones y sus plataformas tecnológicas, y cómo deben cambiar todas ellas para respaldar la estrategia. Los resultados típicos son mapas de capacidades, hojas de ruta hacia el estado objetivo, estándares tecnológicos y decisiones sobre qué sistemas retirar. El marco más usado en este ámbito es TOGAF, publicado por The Open Group, cuya versión actual es el TOGAF Standard, 10th Edition (2022). Ofrece un método (el Architecture Development Method) y un cuerpo de contenidos para planificar y gobernar el cambio en la empresa. Las organizaciones lo usan en distinta medida; muchas adoptan partes de él y a algunas les resulta pesado, y este curso solo lo describe para que puedas reconocerlo.

La **arquitectura de soluciones** aborda un problema o programa de negocio, que normalmente abarca varias aplicaciones, equipos y proveedores: por ejemplo, un hospital que introduce la reserva de citas en línea y debe conectar un nuevo frontend web, el sistema de administración de pacientes existente, un proveedor de SMS y el servicio de identidad. El arquitecto de soluciones hace que las piezas encajen, a menudo con restricciones comerciales y contractuales.

La **arquitectura de software o de aplicaciones**, tema principal de este curso, se ocupa de las estructuras y las cualidades de un sistema: su descomposición, sus interacciones en tiempo de ejecución, su despliegue, sus datos y su evolución.

Tres especialidades atraviesan todos estos niveles. La **arquitectura de datos** abarca cómo se modelan, poseen, almacenan, mueven, gobiernan y usan para analítica los datos (Capítulo 7). La **arquitectura de seguridad** abarca la identidad, el acceso, la protección de los datos, los modelos de amenazas y dónde se sitúan los controles de seguridad (Capítulo 8). La **arquitectura de infraestructura** (hoy a menudo arquitectura de nube o de plataforma) abarca el cómputo, las redes, el almacenamiento, los entornos y las plataformas de despliegue (Capítulo 10).

Niveles y especialidades transversales



Cada nivel restringe al inferior; las especialidades atraviesan todos los niveles

El alcance se estrecha de la empresa al código; la arquitectura de datos, de seguridad y de infraestructura se aplican en todos los niveles.

Quién toma cada decisión

En una organización sana, las decisiones se toman en el nivel más bajo que tiene suficiente contexto y responde de las consecuencias. La arquitectura no es propiedad exclusiva de las personas que llevan "arquitecto" en su cargo. Un reparto razonable por defecto es el siguiente, aunque cada organización lo ajusta.

Tipo de decisión	Normalmente la toma	Normalmente se consulta a
Estándares y plataformas tecnológicas de la organización (p. ej., proveedor de nube aprobado, plataforma de identidad)	Arquitectos empresariales, CTO, junta de arquitectura	Arquitectos de soluciones y de software, seguridad, finanzas
Cómo se combinan varios sistemas y proveedores para resolver un problema de negocio	Arquitecto de soluciones	Product owners, arquitectos empresariales, arquitectos de los proveedores
La estructura principal de un sistema: descomposición, estilo de comunicación, propiedad de los datos	El arquitecto de software o el líder técnico del equipo, junto con el equipo	Operaciones, seguridad, otros equipos afectados
El diseño dentro de un componente: clases, patrones, algoritmos	El equipo de desarrollo	El arquitecto, solo si está en juego un principio o una cualidad
Elecciones a nivel de código: nombres, refactorización local	Cada desarrollador, mediante la revisión de código	Nadie más, normalmente

Hay dos señales de alerta de que el reparto es incorrecto. Si los arquitectos deciden las estructuras internas de las clases, se convierten en un cuello de botella y el equipo deja de pensar. Si los equipos toman por su cuenta decisiones con efectos en todo el sistema (por ejemplo, que cada uno elija su propio intermediario de mensajes), la organización paga más tarde por la incoherencia. Poner por escrito qué decisiones requieren una consulta más amplia, y facilitar su registro, se trata en decisiones y compromisos y, en profundidad, en el tratamiento de los registros de decisiones de arquitectura (ADR) del Capítulo 4.

EJEMPLO PRÁCTICO

Ejemplo práctico: clasificar decisiones para una app de transporte con conductor

Una startup de transporte con conductor tiene cinco decisiones pendientes. El líder técnico usa las tres señales (costo de revertir, alcance, efecto en las cualidades clave) para decidir cuáles necesitan un registro escrito y una revisión más amplia.

1. Guardar las ubicaciones de los viajes en una base de datos de series temporales	Revertir: alto (migración de datos). Alcance: viajes, facturación, analítica. Calidad: rendimiento del seguimiento en vivo. Veredicto: arquitectónica; hay que documentarla y revisarla con el personal de datos y de operaciones.
2. Usar una biblioteca cliente HTTP concreta en el servicio de precios	Revertir: bajo. Alcance: un servicio. Calidad: nada significativo. Veredicto: diseño; decide el equipo de precios.
3. Calcular las tarifas en el servidor, nunca en el teléfono	Revertir: medio. Alcance: apps y backend. Calidad: seguridad e integridad de los precios. Veredicto: arquitectónica; es un principio rector.
4. Representar el dinero como enteros en unidades menores (peniques, centavos) en todas partes	Revertir: muy alto una vez que los datos y las API de los socios lo usan. Alcance: todos los servicios. Calidad: corrección. Veredicto: arquitectónica, aunque sea de bajo nivel.
5. Usar colores de modo oscuro en la app de los conductores	Revertir: trivial. Alcance: un conjunto de pantallas. Veredicto: diseño.

Dos de las tres decisiones arquitectónicas (3 y 4) no tienen nada que ver con cajas y flechas; son principios y convenciones. El nivel de abstracción por sí solo las habría clasificado mal.

La decisión 4 es del tipo que más a menudo pasa desapercibido: parece un detalle de programación, pero se vuelve muy cara de cambiar en cuanto otros sistemas dependen de ella.

Preguntas frecuentes

¿Es una decisión arquitectónica elegir un lenguaje de programación?
A menudo, sí. Afecta a la contratación, a las bibliotecas disponibles, a las características de rendimiento y a la facilidad con que se puede compartir código entre componentes. En un sistema de muchos servicios independientes, el lenguaje de un servicio pequeño puede ser una elección local; el lenguaje por defecto de toda la organización es arquitectónico.

¿Un esquema de base de datos es arquitectura o diseño?
Depende de quién lo use. Una tabla que solo se usa dentro de un servicio, detrás de su propia API, es diseño. Un esquema que leen directamente varios servicios, informes y exportaciones a socios es arquitectónico, porque cada cambio debe coordinarse entre todos ellos. Muchos equipos hacen que los esquemas sean privados de un solo servicio precisamente para devolverlos al terreno del diseño.

¿El desarrollo ágil significa que no hay arquitectura?
No. Todo sistema tiene una arquitectura; los enfoques ágiles cambian cuándo y cómo se decide, favoreciendo pasos pequeños y reversibles y aplazando las decisiones irreversibles hasta saber lo suficiente. La Sección 1.8 trata este tema.

¿Quién decide si una decisión es significativa?
Normalmente el equipo y su arquitecto o líder técnico juntos, con criterios acordados como las señales de esta sección. Poner esos criterios por escrito evita dos modos de fallo: debatir cada elección y descubrir a posteriori que un equipo tomó por su cuenta una decisión que afecta a todo el sistema.

Cómo aplican la distinción los profesionales

Los equipos con experiencia convierten el continuo en una rutina ligera. Acuerdan una lista breve de los tipos de decisión que necesitan un registro escrito y una consulta más amplia, por ejemplo cualquier cosa que cambie una API pública, introduzca un nuevo almacén de datos o una nueva tecnología, cruce los límites entre equipos o afecte a un objetivo de calidad declarado. Todo lo demás lo deciden las personas que hacen el trabajo, mediante la revisión de código habitual. Revisan la lista cuando algo se les escapa o cuando las revisiones empiezan a parecer burocracia.

También buscan activamente formas de reducir el conjunto de decisiones arquitectónicas, porque cada decisión que pasa a ser barata de revertir es una menos que hay que acertar a la primera. Ocultar un proveedor detrás de una interfaz, hacer que una base de datos sea privada

de un solo servicio, usar feature flags para los comportamientos arriesgados y desplegar en incrementos pequeños desplazan decisiones del extremo arquitectónico del continuo hacia el extremo del diseño.

PARA PROFUNDIZAR

La idea de intensión y localidad

Amnon Eden y Rick Kazman propusieron en 2003 una distinción más formal, basada en dos criterios. La *intensión* pregunta si un enunciado es general (describe toda una clase de implementaciones posibles) o específico (describe una sola implementación); la *localidad* pregunta si se puede comprobar en una parte del sistema o si restringe el conjunto. Los enunciados arquitectónicos son generales y *no locales*: restringen todo el sistema ("todos los componentes se comunican a través del bus de mensajes"), así que pueden infringirse en cualquier parte. Los enunciados de diseño son generales pero locales ("usar el patrón Observer en este módulo"). Los enunciados de implementación son específicos y locales ("esta clase tiene estos métodos"). La prueba es útil al redactar reglas: un enunciado que debe cumplirse en todas partes es candidato a principio arquitectónico y a comprobación automatizada, una función de aptitud (fitness function).

ATENCIÓN

Errores que debes evitar

- Tratarlo todo como arquitectura, lo que produce juntas de revisión que ralentizan cada cambio y equipos que dejan de asumir la responsabilidad.
- No tratar nada como arquitectura, lo que deja que las propiedades de todo el sistema surjan por accidente de cientos de elecciones locales.
- Dar por hecho que una decisión es de diseño porque es de bajo nivel. Los formatos de datos, los identificadores, la representación del tiempo y del dinero y los contratos de API son con frecuencia arquitectónicos.
- Dar por hecho que una decisión es arquitectónica porque implica un componente grande. Si se revierte fácilmente y su efecto es local, no necesita ceremonia.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué puede ser arquitectónica una elección sobre el formato de un identificador?

Porque, una vez que muchos servicios, bases de datos y socios externos dependen de él, cambiarlo exige un trabajo coordinado entre todos ellos, así que su costo del cambio y su alcance son altos.

▼ ¿Cuál es la diferencia entre la arquitectura empresarial y la arquitectura de soluciones?

La arquitectura empresarial abarca las capacidades, la información, las aplicaciones y la tecnología de toda la organización; la arquitectura de soluciones aborda un problema de negocio que normalmente abarca varios sistemas y equipos.

▼ ¿Cómo puede un arquitecto desplazar una decisión de la arquitectura hacia el diseño?

Haciendo que sea más barata de revertir, por ejemplo ocultando una tecnología detrás de una interfaz para que se pueda sustituir localmente.

FUENTES

- Who Needs an Architect? Martin Fowler, IEEE Software, 2003 (PDF): https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf
- Software Architecture Guide (Martin Fowler): https://martinfowler.com/architecture/
- The TOGAF Standard (The Open Group): https://www.opengroup.org/togaf
- iSAQB CPSA-Foundation curriculum: https://public.isaqb.org/curriculum-foundation/curriculum-foundation-en.html

SECCIÓN 1.3 DE 1.8 · PRINCIPIANTE · 19 MIN DE LECTURA

Por qué importa la arquitectura: costo del cambio, calidad, riesgo y fallos reales

LA IDEA EN BREVE

Imagina dos cocinas que preparan la misma carta. En una, cada puesto está dispuesto para que los cocineros apenas se crucen; en la otra, el único refrigerador está detrás del lavavajillas. Las dos sirven los mismos platos en una noche tranquila. Un sábado de mucho trabajo, o cuando cambia la carta, la segunda cocina se atasca. Con el software ocurre algo parecido: las funcionalidades pueden parecer idénticas, pero la disposición de las partes decide lo rápido, lo fiable y lo barato que es el sistema, y lo doloroso que resulta cambiarlo. Algunos de los fallos de software más costosos de la historia fueron problemas de disposición, no de líneas de código individuales.

LO QUE NECESITAS SABER

- La arquitectura determina en gran medida atributos de calidad como el rendimiento, la disponibilidad, la seguridad y el costo, porque estos surgen de cómo se organizan e interactúan las partes.
- El costo de cambiar una decisión crece a medida que se construyen sobre ella código, datos, integraciones y equipos, por lo que las primeras decisiones estructurales condicionan el costo y la velocidad de todo el trabajo posterior.
- La arquitectura gobierna el riesgo, cómo puede repartirse el trabajo entre equipos (ley de Conway), la comunicación y la capacidad de estimar los cambios.
- Knight Capital perdió más de 460 millones de USD el 1 de agosto de 2012 después de que un despliegue manual dejara uno de sus ocho servidores ejecutando código antiguo que un flag reutilizado reactivó.
- La GAO concluyó que HealthCare.gov se lanzó en octubre de 2013 sin verificar que cumpliera los requisitos de rendimiento, tras unos requisitos sin cerrar, una supervisión débil y unas pruebas incompletas.
- El traslado en 2023 de un servicio de monitorización de Prime Video, de pasos serverless distribuidos a un único proceso, redujo según se informó su costo de infraestructura en más del 90%, lo que muestra que la arquitectura es cuestión de adecuación, no de moda.

Las decisiones que salen caras si se toman mal

La Sección 1.1 definió la arquitectura como el conjunto de estructuras y decisiones significativas que dan forma a un sistema, con la significancia medida por el costo del cambio. Esta sección plantea la continuación práctica: ¿y qué? ¿Por qué debería importarle la arquitectura a un equipo, a un propietario del producto o a un director financiero, y no simplemente las funcionalidades?

La respuesta breve es que la arquitectura decide, más que cualquier otro factor técnico, *cómo* será el sistema: lo rápido, lo fiable, lo seguro y lo barato de operar que será, y sobre todo lo fácil que será cambiarlo. Las funcionalidades son lo que hace un sistema. La arquitectura determina lo bien que lo hace y lo que costará hacer algo distinto el año que viene. Muchos grandes fallos de software y muchos éxitos costosos se remontan a un puñado de decisiones estructurales.

La arquitectura determina los atributos de calidad

Un atributo de calidad es una propiedad medible de lo bien que un sistema hace su trabajo: rendimiento, disponibilidad, seguridad, modificabilidad, escalabilidad, usabilidad, eficiencia de costos y otras. El Capítulo 2 las define con precisión. Lo fundamental aquí es que la mayoría de los atributos de calidad son *emergentes*: surgen de cómo se organizan las partes y de cómo interactúan, no de ningún fragmento de código concreto.

Piensa en la disponibilidad. Si el proceso de pago de una tienda en línea debe hacer cinco llamadas síncronas seguidas a servicios que están disponibles cada uno el 99.9% del tiempo, la cadena en su conjunto no puede estar disponible más de aproximadamente el 99.5%, por bien escrito que esté cada servicio, porque las probabilidades se multiplican. Añadir una cola para que algunos de esos pasos ocurran más tarde, o guardar resultados en caché para que algunas llamadas no sean necesarias, cambia la aritmética. Ninguno de los dos cambios es visible dentro de una sola función; ambos son arquitectónicos. El rendimiento, la seguridad y el costo se comportan de la misma manera. Una página de inicio de sesión segura no hace seguro un sistema si un servicio interno sin autenticación es accesible desde internet.

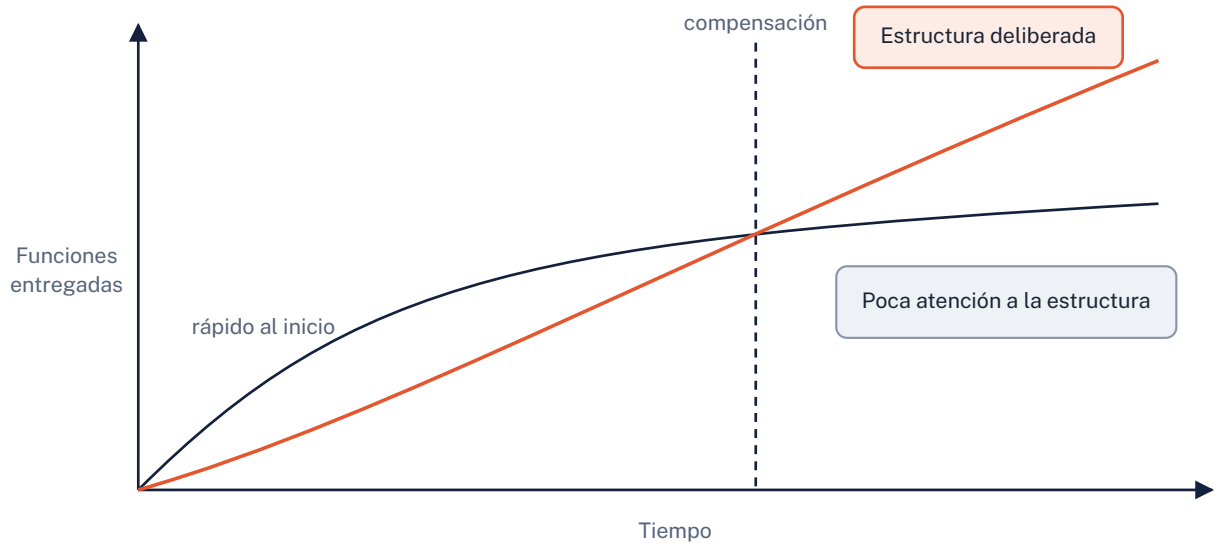
La funcionalidad, en cambio, suele ser en gran medida independiente de la estructura. Puedes implementar las mismas funcionalidades de una app bancaria como una sola unidad desplegable o como cincuenta servicios; los usuarios ven los mismos botones. Lo que cambia es todo lo demás: lo rápido que se puede lanzar una corrección, qué ocurre cuando falla una parte, cuántos ingenieros puede absorber el sistema y cómo es la factura de la nube.

Costo y velocidad del cambio

La mayor parte del costo de un sistema de éxito se produce después de su primer lanzamiento, a medida que se modifica para atender nuevas necesidades. Por eso la estructura importa sobre todo porque determina lo costoso que será cada cambio futuro. En un sistema bien estructurado, un cambio en las normas fiscales afecta a un módulo. En uno enmarañado, afecta a veinte, y cada uno necesita pruebas y coordinación.

La *hipótesis de la resistencia del diseño* (design stamina hypothesis) de Martin Fowler (2007) recoge este efecto. Un equipo que presta poca atención a la estructura interna suele entregar más rápido al principio. Con el tiempo, la maraña acumulada frena cada cambio, y un equipo que invirtió en estructura lo adelanta. Fowler lo llama hipótesis porque es difícil de medir con rigor, y sugiere que el punto de cruce suele llegar en semanas más que en meses. La ralentización es el interés de la deuda técnica, que trata la Sección 1.7.

La hipótesis de la resistencia del diseño



Un esbozo conceptual basado en Martin Fowler: prestar menos atención a la estructura es más rápido al principio, pero una estructura deliberada mantiene el ritmo de entrega.

EJEMPLO PRÁCTICO

Ejemplo práctico: el costo de una decisión a lo largo del tiempo

Una tienda en línea almacena los importes de dinero como números de coma flotante. La aritmética de coma flotante no puede representar exactamente muchos valores decimales (0.1 + 0.2 no es exactamente 0.3), así que de vez en cuando los totales difieren en un penique de lo que esperan los contables. Las cifras siguientes son supuestos ilustrativos, no datos del sector, elegidos para mostrar cómo crece el costo del cambio.

Mes 1: decidir de otro modo	Dos desarrolladores acuerdan almacenar en su lugar unidades menores enteras (peniques). Esfuerzo: aproximadamente medio día. Costo, suponiendo £600 por día-persona: unas £300.
Año 3: lo que se ha construido sobre la decisión	35 servicios manejan dinero; 120 columnas de base de datos lo almacenan; 200 millones de filas históricas de pedidos; 4 integraciones con socios (proveedor de pagos, almacén, contabilidad, marketplace) lo intercambian.
Año 3: cambios en el código	35 servicios a unos 3 días-persona cada uno para cambiar, probar y lanzar = 105 días-persona.
Año 3: migración de datos	Convertir columnas, migrar el histórico, conciliar los totales: unos 20 días-persona, más una ventana de mantenimiento planificada.
Año 3: coordinación con los socios	4 socios a unos 10 días-persona cada uno, incluidas sus pruebas y los cambios de contrato = 40 días-persona.
Año 3: pruebas de extremo a extremo y conciliación con finanzas	Unos 30 días-persona.
Año 3: total	Unos 195 días-persona, o aproximadamente £117 000, más el riesgo de errores durante la migración y meses de tiempo transcurrido. Eso es varios cientos de veces el costo del mes 1.

La decisión en sí nunca cambió de tamaño. Lo que creció fue todo lo construido sobre ella: código, datos, socios y hábitos.

Por eso Ralph Johnson describió la arquitectura como las decisiones que desearías poder acertar pronto, y por eso los arquitectos buscan convenciones de bajo nivel con un alcance amplio.

Una estrategia alternativa es hacer que esas decisiones sean más baratas de revertir, por ejemplo poniendo todo el manejo del dinero detrás de un único tipo o biblioteca compartidos desde el principio, para que un cambio posterior sea local.

Riesgo, equipos, comunicación y estimación

Riesgo. La arquitectura es donde la mayoría de los riesgos técnicos quedan contenidos o se amplifican. Un diseño con un punto único de fallo, una integración sin probar entre partes construidas de forma independiente o una tecnología no probada en su núcleo conlleva un riesgo que ninguna cantidad de programación cuidadosa elimina. Gran parte del trabajo arquitectónico es gestión de riesgos: identificar qué podría salir muy mal y dar forma a la estructura para que no pueda ocurrir o, si ocurre, cause un daño limitado.

Repartir el trabajo entre equipos. Melvin Conway observó en 1968 que las organizaciones que diseñan sistemas tienden a producir diseños que reflejan sus propias estructuras de comunicación. Hoy se llama ley de Conway, y funciona en ambas direcciones. Un sistema cuyas partes tienen límites claros e interfaces estrechas permite que varios equipos trabajen en paralelo con poca coordinación. Un sistema en el que todo depende de todo obliga a cada equipo a estar en cada reunión. Por eso la arquitectura y el diseño de la organización están entrelazados, como explica la sección del Capítulo 3 sobre la ley de Conway.

Comunicación. Una arquitectura da a las personas un vocabulario compartido y un mapa compartido. Los nuevos ingenieros aprenden el sistema más rápido, los gerentes de producto entienden por qué algunas solicitudes son baratas y otras caras, y el personal de operaciones sabe qué depende de qué durante un incidente.

Estimación y planificación. La descomposición en partes conocidas hace posible la estimación: un cambio puede rastrearse hasta los componentes que toca, y cada uno puede estimarlo el equipo que lo conoce. Sin una estructura clara, las estimaciones son conjeturas, porque nadie puede decir con seguridad hasta dónde se propagará un cambio.

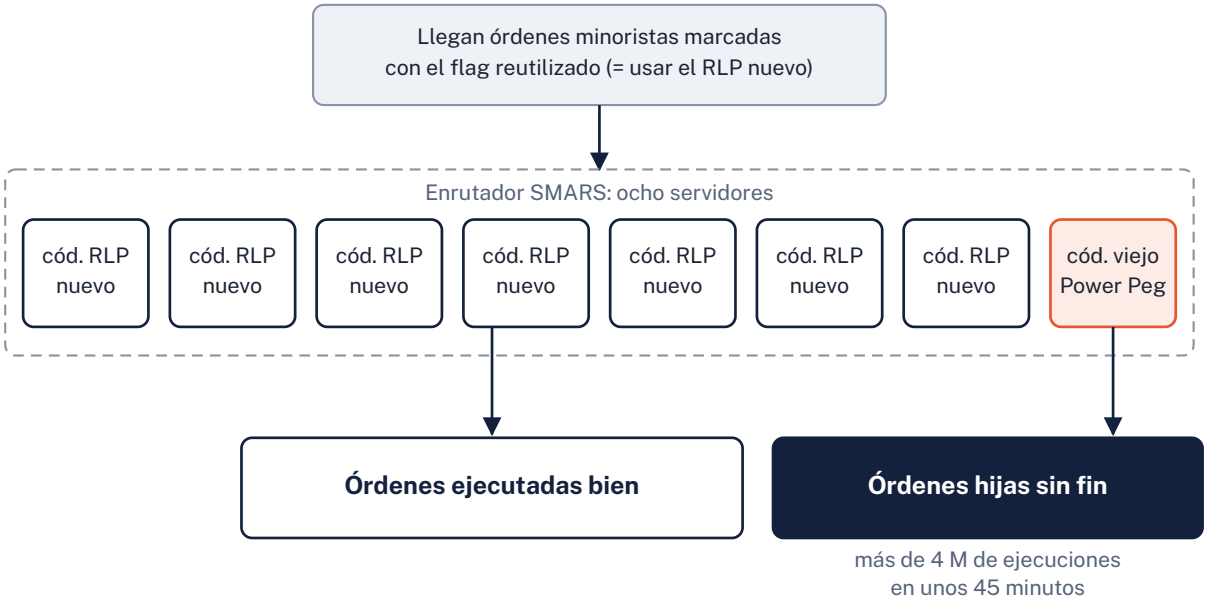
Caso práctico: Knight Capital, 1 de agosto de 2012

Knight Capital era uno de los mayores creadores de mercado de la renta variable estadounidense. Según la orden de la Comisión de Bolsa y Valores de Estados Unidos (SEC) del 16 de octubre de 2013, en los primeros 45 minutos de negociación del 1 de agosto de 2012 el enrutador automatizado de órdenes de Knight (llamado SMARS) envió al mercado millones de órdenes no deseadas mientras procesaba 212 pequeñas órdenes minoristas. Recibió más de 4 millones de ejecuciones en 154 valores, por más de 397 millones de acciones, y se quedó con grandes posiciones no deseadas. Knight perdió más de 460 millones de USD, necesitó una inversión externa de emergencia en cuestión de días, y más tarde la SEC le impuso una multa de 12 millones de USD por infringir las normas de acceso al mercado.

La orden de la SEC describe el mecanismo. Años antes, Knight había dejado de usar una antigua función llamada Power Peg, pero su código seguía en el enrutador y aún podía activarse: era código muerto que no estaba muerto. En 2012, Knight escribió código nuevo para

un nuevo programa de la NYSE, el Retail Liquidity Program, y ese código nuevo reutilizó un flag que antes activaba Power Peg. El código nuevo se desplegó manualmente en los ocho servidores del enrutador. Un técnico no lo copió en uno de ellos, ningún segundo técnico revisó el despliegue, y Knight no tenía ningún procedimiento escrito que exigiera esa revisión. Cuando se activó el flag, siete servidores ejecutaron el código nuevo; el octavo interpretó el flag como "ejecutar Power Peg" y envió órdenes sin reconocer que ya se habían ejecutado. Antes de la apertura del mercado se enviaron correos electrónicos automáticos que avisaban de un problema, pero nadie actuó en consecuencia.

Cómo un servidor desactualizado causó la pérdida de Knight Capital



Ningún segundo técnico revisó el despliegue ni se verificó automáticamente que los servidores coincidieran

Un flag reutilizado, un servidor desactualizado y ninguna comprobación del despliegue: el fallo estaba en la estructura y el proceso, no en una sola línea de código.

Sería injusto llamarlo simplemente un bug de programación, e igual de injusto llamarlo un error puramente de operaciones. Las lecciones arquitectónicas tienen que ver con la estructura y la desplegabilidad: funcionalidad obsoleta que seguía siendo accesible en un sistema en producción; un mismo flag con dos significados en distintas versiones del código; un proceso de despliegue sin verificación automática de que todos los servidores ejecutaban la misma versión; y, como subrayó la SEC, controles inadecuados para limitar el daño una vez que algo salía mal. Cada una es una decisión sobre cómo se construye y se lanza el sistema, no sobre una sola línea de código.

Caso práctico: el lanzamiento de HealthCare.gov, octubre de 2013

HealthCare.gov, el mercado federal de seguros médicos de Estados Unidos, abrió las inscripciones el 1 de octubre de 2013. Muchas personas no pudieron crear cuentas ni completar solicitudes, y el sitio sufrió caídas durante semanas. La Oficina de Rendición de Cuentas del Gobierno de Estados Unidos (GAO) y la Oficina del Inspector General (OIG) del Departamento de Salud y Servicios Humanos investigaron después, y sus conclusiones son la mejor fuente sobre lo que salió mal.

El informe de la GAO de julio de 2014 (GAO-14-694) concluyó que los Centros de Servicios de Medicare y Medicaid (CMS) adjudicaron contratos antes de que se fijaran requisitos clave, que los requisitos no dejaron de cambiar, que la supervisión fue débil y que los CMS lanzaron el sistema sin verificar que cumpliera sus requisitos de rendimiento. Las obligaciones del contrato principal del mercado pasaron de 56 millones de USD a más de 209 millones de USD entre 2011 y principios de 2014. Un segundo informe de la GAO, de marzo de 2015 (GAO-15-238), concluyó que los sistemas no se habían probado por completo antes del lanzamiento, y que entre los problemas del lanzamiento figuraban una planificación de capacidad inadecuada, errores de programación del software y funciones previstas que no se habían implementado. El estudio de caso de la OIG de 2016 señaló como causas centrales la ausencia de un liderazgo claro y la mala coordinación entre el trabajo normativo y el técnico. Tras un intenso trabajo correctivo, el sitio pudo atender una alta demanda de los consumidores en unos dos meses.

La historia de HealthCare.gov trata sobre todo de la gestión del programa, pero su lección arquitectónica es directa. El mercado lo construyeron varios contratistas como partes separadas que tenían que funcionar juntas, entre ellas un nodo central de datos (data hub) que se conectaba con otras agencias federales, y nadie verificó de extremo a extremo, con una carga realista, que el conjunto ensamblado cumpliera sus objetivos de capacidad y rendimiento. Los atributos de calidad como la capacidad son propiedades del sistema completo, así que solo puede garantizarlos alguien que sea responsable de la estructura de todo el sistema y lo pruebe como un todo.

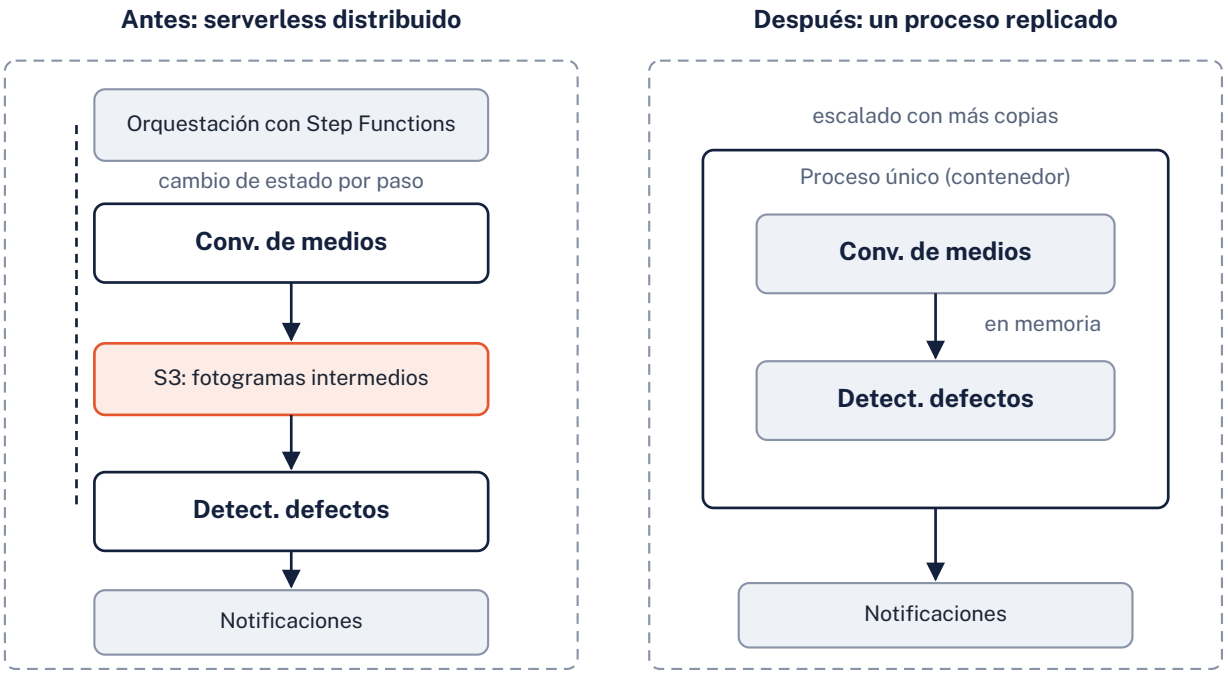
Caso práctico: Prime Video y la arquitectura como adecuación, no como moda

En marzo de 2023, un ingeniero del equipo de análisis de calidad de video de Amazon Prime Video publicó una entrada en el blog Prime Video Tech sobre un servicio que monitoriza las transmisiones de video en busca de defectos de calidad. La primera versión se había construido con componentes serverless distribuidos: AWS Step Functions orquestaba los pasos y las funciones de AWS Lambda hacían el trabajo, y los fotogramas de video intermedios pasaban de un paso a otro a través del almacenamiento de Amazon S3. El equipo informó de que este

diseño chocó con un límite de escalado rígido en torno al 5% de la carga esperada y habría sido demasiado caro a plena escala, en gran parte por los cargos de orquestación por paso y las numerosas solicitudes al almacenamiento.

El equipo trasladó los pasos de conversión de medios y de detección de defectos a un único proceso que se ejecutaba en contenedores en Amazon ECS, pasando los fotogramas en memoria en lugar de a través del almacenamiento, y escaló ejecutando más copias de ese proceso. La entrada informó de que los costos de infraestructura cayeron más del 90% y de que el servicio podía escalar aún más. La entrada se compartió ampliamente como "Amazon abandona los microservicios", algo que no decía: el cambio afectaba a un servicio, y el autor señaló que los microservicios y los componentes serverless sí funcionan a gran escala, pero que la elección entre ellos y un monolito debe hacerse caso por caso.

El servicio de monitorización de Prime Video, antes y después



Los mismos pasos, reagrupados: mover los datos en memoria dentro de un proceso eliminó los costos de orquestación por paso y de almacenamiento.

La lección no es que los monolitos superen a serverless. Es que la arquitectura es cuestión de adecuación. El diseño serverless fue rápido de construir y adecuado para una carga de trabajo temprana y más pequeña. Cuando la característica definitoria de la carga de trabajo (volúmenes muy altos de datos pasados entre pasos por cada segundo de cada transmisión) pasó a ser dominante, una estructura que movía esos datos por la red muchas veces dejó de encajar, y otra estructura encajaba mejor. Un buen arquitecto se pregunta qué necesitan la carga de trabajo, el equipo y el presupuesto, no qué está de moda.

ATENCIÓN

Leer los casos prácticos con atención

- Los relatos públicos simplifican. Usa fuentes primarias (órdenes de los reguladores, informes de auditoría, las entradas de los propios ingenieros) en lugar de versiones de segunda mano, que a menudo exageran o añaden detalles.
- La mayoría de los fallos reales tienen varias causas: estructura técnica, proceso, organización e incentivos. Culpar a un ingeniero o a una línea de código casi siempre es un error.
- Una decisión que fracasó en un contexto puede ser acertada en otro. Pregúntate cuál era el contexto antes de sacar una regla general.

PARA PROFUNDIZAR

Por qué cuesta justificar la arquitectura y cómo hacerlo

Los beneficios de una buena arquitectura son sobre todo cosas que *no* ocurren: la caída evitada, la reescritura que no hace falta, el equipo que no se queda bloqueado. Eso hace que sea fácil restarles importancia. Los arquitectos eficaces traducen las decisiones estructurales a términos de negocio: tiempo de entrega de los cambios, costo por transacción, horas de inactividad al año, número de equipos que pueden trabajar en paralelo, tamaño del riesgo que se asume. El Capítulo 11 vuelve sobre cómo construir casos de negocio para el trabajo arquitectónico.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Por qué un servicio bien escrito puede formar parte de un sistema poco fiable?**

Porque la disponibilidad surge de cómo dependen los servicios unos de otros. Una cadena de llamadas síncronas multiplica las probabilidades de fallo, así que el conjunto puede estar menos disponible que cualquiera de sus partes.

▼ **¿Qué hizo que el fallo de Knight Capital fuera arquitectónico y no un simple bug?**

Código obsoleto accesible, un flag con dos significados según la versión y un despliegue manual sin ninguna comprobación automática de que todos los servidores coincidían: todas ellas decisiones estructurales y de lanzamiento, no una línea defectuosa.

▼ **¿Cuál es la lección principal del caso de Prime Video?**

Que una arquitectura debe adecuarse a su carga de trabajo y a su contexto. Un diseño adecuado para una escala y un patrón de datos puede dejar de encajar, y cambiar la estructura puede transformar el costo.

▼ **¿Por qué crece con el tiempo el costo de cambiar una decisión?**

Porque pasan a depender de ella más código, más datos almacenados, más integraciones, socios y hábitos, así que cambiarla significa cambiarlos todos.

FUENTES

- SEC press release 2013-222: SEC charges Knight Capital with violations of market access rule: https://www.sec.gov/news/press-release/2013-222
- GAO-14-694: Healthcare.gov: Ineffective Planning and Oversight Practices Underscore the Need for Improved Contract Management: https://www.gao.gov/products/gao-14-694
- GAO-15-238: Healthcare.gov: CMS Has Taken Steps to Address Problems, but Needs to Further Implement Systems Development Best Practices: https://www.gao.gov/products/GAO-15-238
- Return of the Monolith: Amazon Dumps Microservices for Video Monitoring (The New Stack): https://thenewstack.io/return-of-the-monolith-amazon-dumps-microservices-for-video-monitoring/
- Design Stamina Hypothesis (Martin Fowler): https://martinfowler.com/bliki/DesignStaminaHypothesis.html

SECCIÓN 1.4 DE 1.8 · PRINCIPIANTE · 18 MIN DE LECTURA

Qué hace un arquitecto: responsabilidades, habilidades y tipos de arquitecto

LA IDEA EN BREVE

En una obra, el arquitecto no coloca cada ladrillo, pero decide las cosas que serían ruinosas si salieran mal, las explica a los constructores y a los clientes, comprueba en la obra que se siguen los planos y hace ajustes cuando el terreno resulta más blando de lo esperado. Un arquitecto de software hace lo mismo con un sistema de software: toma o guía las grandes decisiones, las explica, comprueba continuamente que se siguen y que siguen teniendo sentido, y necesita saber un poco de muchas tecnologías y bastante de personas y de negocio.

LO QUE NECESITAS SABER

- Los arquitectos asumen la responsabilidad de que las decisiones significativas se tomen bien, se entiendan y se mantengan vigentes a medida que cambia un sistema; la arquitectura es un rol antes que un cargo.
- Richards y Ford describen ocho expectativas: tomar decisiones, analizar continuamente, mantenerse al día, asegurar el cumplimiento, tener una exposición diversa, conocer el dominio de negocio, tener habilidades interpersonales y saber moverse en la política.
- El plan de estudios de iSAQB enumera las tareas de los arquitectos: aclarar los requisitos, diseñar estructuras y conceptos transversales, comunicar, acompañar la implementación y evaluar, de forma iterativa.
- Entre los tipos de arquitecto están el empresarial, el de soluciones, el de software o de aplicaciones, el de datos, el de seguridad y el de nube o infraestructura; los staff engineers y los principal engineers suelen asumir las mismas responsabilidades.
- Los arquitectos valoran la amplitud técnica (conocer muchas opciones y sus compromisos) por encima de la experiencia profunda en unas pocas, como muestra la pirámide del conocimiento.
- Por lo general, los arquitectos deberían seguir implicados en el código sin hacerse cargo del código de la ruta crítica: pruebas de concepto, funciones de aptitud, corrección de bugs y revisiones de código.

Para qué sirven los arquitectos

Si la arquitectura es el conjunto de decisiones y estructuras significativas de un sistema (Sección 1.1), el arquitecto es quien asume la responsabilidad de que esas decisiones se tomen bien, los demás las entiendan y se mantengan vigentes a medida que cambia el sistema. Parece sencillo, pero el rol es uno de los que se definen de forma menos coherente en el software. En una empresa, un arquitecto es un ingeniero sénior que sigue escribiendo código a diario; en otra, un miembro de un equipo central que revisa diseños y mantiene estándares; en una tercera, una persona que sobre todo habla con los líderes de negocio y los proveedores.

Detrás de esta variedad hay un núcleo común. Los arquitectos toman o guían decisiones costosas de revertir, cuidan las cualidades del sistema completo y no de una sola parte, y conectan las decisiones técnicas con los objetivos de negocio. El resto de esta sección describe lo que eso implica, qué habilidades requiere y cómo se organiza el rol.

Ocho expectativas de un arquitecto

Mark Richards y Neal Ford, en *Fundamentals of Software Architecture* (O'Reilly, 1.ª edición 2020, 2.ª edición 2025), describen ocho expectativas fundamentales de un arquitecto, sea cual sea el cargo o la empresa. A continuación se parafrasean, con un ejemplo de cada una tomado de una empresa de reparto de comida.

Las ocho expectativas (según Richards y Ford)

Expectativa	Qué significa	Ejemplo
Tomar decisiones de arquitectura	Definir decisiones y principios que orienten las elecciones tecnológicas, normalmente guiando al equipo en lugar de imponer productos concretos.	Decidir que las integraciones con restaurantes pasen por una única capa de adaptadores, en lugar de elegir la biblioteca de cada restaurante.
Analizar continuamente la arquitectura	Reevaluar si la arquitectura sigue encajando a medida que cambian la tecnología y el negocio, y buscar signos de deterioro.	Darse cuenta de que el servicio de asignación de repartidores acumula ahora el 40% de todos los fallos de despliegue y preguntar por qué.
Mantenerse al día de las tendencias	Seguir la evolución de la tecnología y de la práctica, porque las decisiones arquitectónicas son duraderas.	Entender las nuevas ofertas de bases de datos gestionadas antes de la próxima revisión de capacidad.
Asegurar el cumplimiento de las decisiones	Comprobar que las decisiones y los principios acordados se siguen realmente, idealmente con comprobaciones automáticas.	Añadir una comprobación en la compilación que falle cuando la app llame directamente al proveedor de pagos.
Tener una exposición y una experiencia diversas	Conocer una amplia variedad de tecnologías, frameworks y plataformas lo bastante bien como para compararlas.	Comparar intermediarios de mensajes (message brokers) según el orden, la durabilidad y el costo de operación.
Conocer el dominio de negocio	Entender el negocio lo bastante bien como para hablar con las partes interesadas en sus propios términos.	Saber cómo funcionan los pagos a los restaurantes y los reembolsos antes de rediseñar los pagos.
Tener habilidades interpersonales	Liderar, colaborar, hacer de mentor y resolver desacuerdos; la mayor parte del trabajo de arquitectura se hace a través de otras personas.	Facilitar una sesión en la que dos equipos acuerdan un contrato de eventos.
Entender la política y saber moverse en ella	Conseguir apoyo para las decisiones en una organización donde los presupuestos, las prioridades y la propiedad están en disputa.	Lograr el acuerdo de finanzas y de operaciones antes de retirar un antiguo sistema de informes.

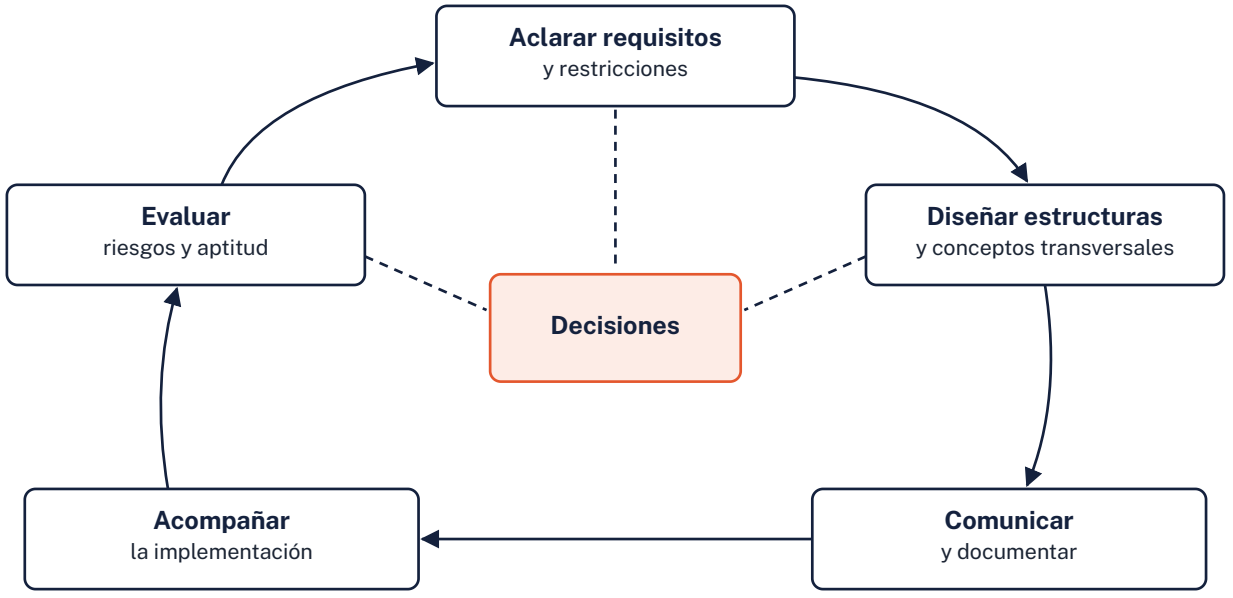
Fíjate en qué pocas de las ocho son puramente técnicas. Richards y Ford insisten en que casi todas las decisiones que toma un arquitecto serán cuestionadas por alguien (desarrolladores, otros arquitectos, propietarios del producto, jefes de proyecto), así que la negociación y la persuasión son habilidades centrales, no extras. Una arquitectura correcta sobre el papel, pero que no se entiende ni se acepta, no se construirá como se pretendía.

La visión de iSAQB sobre las tareas de los arquitectos

El International Software Architecture Qualification Board (iSAQB), cuyo plan de estudios de nivel Foundation (CPSA-F) se usa ampliamente en Europa, describe las tareas y responsabilidades de los arquitectos de forma parecida. En resumen, los arquitectos: aclaran y examinan a fondo los requisitos y las restricciones; deciden cómo descomponer el sistema en bloques de construcción con sus dependencias e interfaces; deciden sobre inquietudes transversales (cross-cutting concerns) como la persistencia, la comunicación y la interfaz de usuario; comunican y documentan la arquitectura; acompañan su realización e implementación; la analizan y evalúan, sobre todo en cuanto a los riesgos para cumplir los requisitos; y explican las consecuencias de las decisiones arquitectónicas a otras partes interesadas.

El plan de estudios también insiste en que estas tareas son iterativas: los arquitectos trabajan en ciclos y revisan los requisitos y las decisiones a medida que llega la retroalimentación, en lugar de diseñar una vez y pasar el trabajo a otros. La Sección 1.8 examina cómo encaja esto en la entrega iterativa y continua.

El trabajo del arquitecto como ciclo



Cada tarea alimenta y revisa las decisiones; el ciclo se repite durante toda la vida del sistema

Según el plan de estudios de iSAQB: aclarar, diseñar, comunicar, acompañar y evaluar se repiten, y todo gira en torno a las decisiones.

Tipos de arquitecto

Los cargos siguen los niveles de arquitectura descritos en la Sección 1.2. Las responsabilidades exactas varían mucho entre organizaciones, así que considera la tabla siguiente como típica más que definitiva.

Tipo	Enfoque	Resultados típicos
Arquitecto empresarial	Toda la organización: capacidades, cartera de aplicaciones, estándares tecnológicos, alineación con la estrategia	Mapas de capacidades, hojas de ruta del estado objetivo, estándares, asesoramiento sobre inversiones
Arquitecto de soluciones	Un problema de negocio o un programa que abarca varios sistemas, proveedores y equipos	Diseños de soluciones, diseños de integración, evaluaciones de proveedores
Arquitecto de software o de aplicaciones	La estructura y las cualidades de un sistema o de un grupo de sistemas estrechamente relacionados	Estructuras de componentes, registros de decisiones, escenarios de calidad, orientación técnica a los equipos
Arquitecto de datos	Cómo se modelan los datos, a quién pertenecen y cómo se almacenan, mueven, gobiernan y analizan	Modelos de datos, mapas de propiedad de los datos, diseños de pipelines y de plataformas
Arquitecto de seguridad	Identidad, acceso, protección de los datos y ubicación de los controles de seguridad	Modelos de amenazas, patrones de seguridad, requisitos de control
Arquitecto de nube o de infraestructura	Cómputo, red, almacenamiento, entornos y plataformas	Landing zones (zonas de aterrizaje), diseños de red, estándares de plataforma, modelos de costos

¿Un rol o un cargo?

La arquitectura es un conjunto de responsabilidades antes que un cargo. En muchas organizaciones, sobre todo en las más pequeñas y en las organizadas en torno a equipos de producto autónomos, nadie tiene el cargo de "arquitecto". El trabajo arquitectónico se sigue haciendo, a cargo de los líderes técnicos, de los ingenieros sénior o del equipo en su conjunto. Lo que importa es que alguien sea claramente responsable de las decisiones significativas y de las cualidades de todo el sistema. Cuando nadie lo es, se sigue produciendo arquitectura, pero por accidente.

Muchas empresas tecnológicas ofrecen una carrera técnica para los contribuidores individuales sénior, con niveles como staff engineer y principal engineer. *Staff Engineer* (2021), de Will Larson, describe cuatro arquetipos habituales de estos roles, uno de los cuales se llama explícitamente Architect (arquitecto): una persona responsable de la dirección y la calidad de un área técnica crítica. Los demás (Tech Lead, Solver y Right Hand) también asumen una responsabilidad arquitectónica considerable. En estas organizaciones, el trabajo del arquitecto lo hacen personas que también son ingenieros.

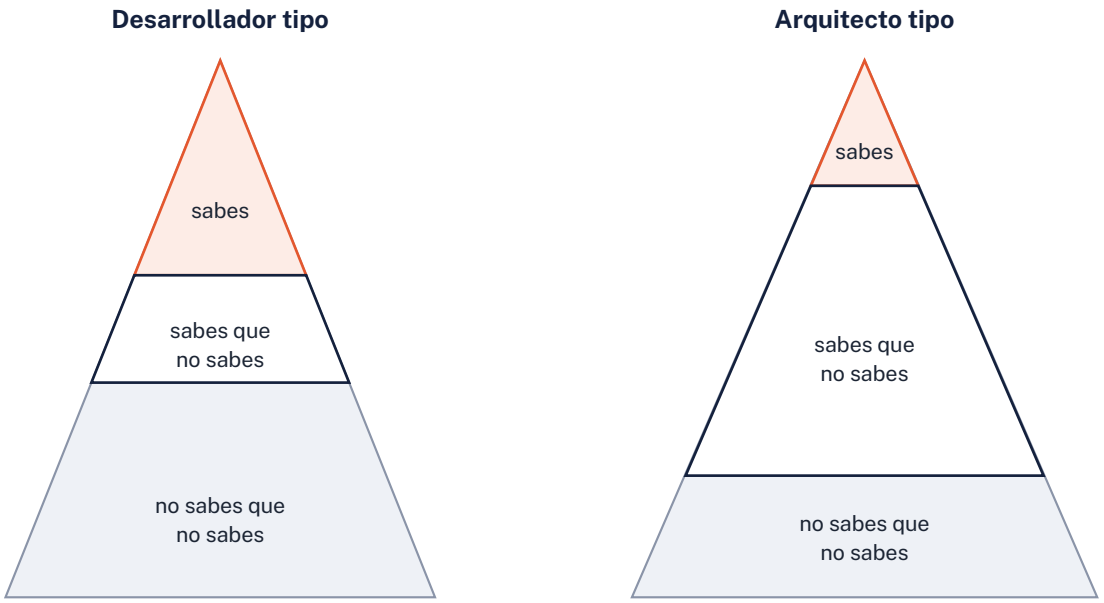
El artículo de Martin Fowler de 2003 *Who Needs an Architect?* contrapone dos caricaturas: un arquitecto que toma en solitario todas las decisiones importantes y otro que trabaja codo con codo con el equipo, hace de mentor de los desarrolladores y les ayuda a tomar buenas decisiones por sí mismos. Fowler defendió con firmeza el segundo. Hoy la mayoría de los profesionales coincide en que los arquitectos que dictan desde lejos suelen producir diseños que los equipos esquivan discretamente.

Amplitud técnica frente a profundidad

A los desarrolladores se les suele recompensar por la profundidad: conocer extremadamente bien un lenguaje, un framework o una plataforma. Los arquitectos necesitan un perfil distinto. Richards y Ford describen una pirámide del conocimiento con tres capas: lo que sabes (donde tienes verdadera experiencia), lo que sabes que no sabes (sabes que existe y, a grandes rasgos, para qué sirve, pero hoy no podrías construir nada con ello) y lo que no sabes que no sabes. La experiencia de la capa superior hay que mantenerla, y eso lleva tiempo. La capa intermedia es barata de ampliar y es la que te permite reconocer opciones.

El valor de un arquitecto reside en gran medida en la amplitud técnica: saber que existen cinco formas de resolver un problema, en qué es buena y en qué es mala cada una y cuál merece un análisis más detenido. Un arquitecto que solo conoce una base de datos la recomendará siempre. La amplitud tiene un precio, porque el tiempo dedicado a ensanchar la capa intermedia no se dedica a profundizar en la superior; por eso los arquitectos dejan deliberadamente que se desvanezca parte de su antigua experiencia e invierten en saber de más cosas. Richards y Ford advierten del fallo contrario: el arquitecto cuyas opiniones se formaron a partir de una mala experiencia de hace años y que rechaza toda opción que se le parezca.

La pirámide del conocimiento



La profundidad (arriba) sigue importando, pero la amplitud (en medio) permite comparar opciones

Según Richards y Ford: un arquitecto ensancha deliberadamente la capa intermedia, la de las cosas que sabe que existen pero no domina.

¿Deberían escribir código los arquitectos?

Aquí los profesionales discrepan de verdad, y el desacuerdo es útil porque los dos lados tienen razón en algo.

Argumentos a favor de programar. Los arquitectos que programan entienden el costo real de sus decisiones, notan cuándo un principio resulta penoso en la práctica, mantienen al día su criterio técnico y conservan el respeto de los desarrolladores que deben implementar sus diseños. Los arquitectos que dejan de programar durante años corren el riesgo de diseñar sistemas elegantes en una pizarra y torpes en un editor.

Argumentos en contra. El tiempo de un arquitecto se reparte entre muchos equipos, reuniones y decisiones. Un arquitecto que se hace cargo de funcionalidades de la ruta crítica se convierte en un cuello de botella: el equipo espera un código que se retrasa por las otras obligaciones del arquitecto. Su atención también es necesaria en inquietudes que afectan a varios equipos y de las que ningún equipo se ocupará por sí solo.

La solución habitual. La mayoría de las voces experimentadas, entre ellas Richards y Ford, recomiendan que los arquitectos sigan implicados en el código de maneras que no los pongan en la ruta crítica. Por ejemplo: construir una prueba de concepto para probar una opción arriesgada antes de que el equipo se comprometa con ella; escribir funciones de aptitud (fitness functions), comprobaciones automáticas que verifican reglas arquitectónicas como "el

módulo de dominio no debe depender del framework web"; corregir bugs y hacer pequeñas tareas de deuda técnica; crear herramientas y automatizaciones para el equipo; y revisar código con regularidad, lo que muestra cómo se están implementando realmente las decisiones.

```
import pathlib, re

FORBIDDEN = re.compile(r"^\s*(from|import)\s+flask\b", re.M)

def test_domain_does_not_depend_on_web_framework():
    offenders = []
    for path in pathlib.Path("shop/domain").rglob("*.py"):
        if FORBIDDEN.search(path.read_text(encoding="utf-8")):
            offenders.append(str(path))
    assert not offenders, f"Domain code imports Flask: {offenders}"
```

Una función de aptitud sencilla que podría escribir un arquitecto: hacer fallar la compilación si algún módulo de dominio importa el framework web.

Una semana en la vida de un arquitecto de software (ilustrativa)

Día	Actividades principales	Expectativa puesta en práctica
Lunes	Planificación con dos equipos de producto; detectar que una nueva funcionalidad de fidelización necesita datos que el servicio de pedidos no expone; redactar opciones	Tomar decisiones; conocer el dominio de negocio
Martes	Taller con el delegado de protección de datos y el equipo de seguridad sobre el almacenamiento de las direcciones de los clientes; redactar un registro de decisiones	Habilidades interpersonales; asegurar el cumplimiento
Miércoles	Prueba de concepto de dos días de un servicio gestionado de streaming de eventos para probar las garantías de orden bajo carga	Exposición diversa; mantenerse al día
Jueves	Revisiones de código en el equipo de pagos; añadir una función de aptitud tras detectar una dependencia prohibida; mentoría individual	Asegurar el cumplimiento; habilidades interpersonales
Viernes	Revisar informes de incidentes y métricas de tiempo de entrega en busca de signos de deterioro arquitectónico; reunión con finanzas sobre los costos de la nube; lectura	Analizar continuamente; moverse en la política; mantenerse al día

EJEMPLO PRÁCTICO

Ejemplo práctico: una decisión, cinco tareas

El sistema de reserva de citas de un hospital debe enviar recordatorios por SMS. El arquitecto de software recorre las tareas de iSAQB para esta única necesidad.

Aclarar requisitos	Pregunta cuántos recordatorios se envían al día (unos 20 000), con qué rapidez deben salir, qué ocurre si el proveedor de SMS no está disponible y si los mensajes pueden contener datos clínicos (no pueden, por privacidad).
Diseño	Propone que el servicio de reservas publique eventos de "cita reservada" y que un componente de notificaciones independiente envíe los mensajes desde una cola, para que una caída del SMS nunca bloquee las reservas.
Comunicar	Redacta un registro de decisiones de una página con las opciones consideradas (llamada directa frente a cola) y los motivos; lo presenta al equipo y al responsable de gobernanza de la información.
Acompañar la implementación	Trabaja en pareja con un desarrollador en la primera versión, revisa la lógica de reintentos y añade una función de aptitud que impide incluir campos clínicos en la plantilla del mensaje.
Evaluar	Tres meses después comprueba los tiempos de entrega y las tasas de fallo; descubre que los recordatorios se quedan 30 minutos en cola durante una caída del proveedor y confirma con el personal clínico que es aceptable.

La mayor parte del esfuerzo del arquitecto se dedicó a aclarar, comunicar y comprobar, no a dibujar el diseño.

La decisión se registró con sus motivos, para que un equipo futuro pueda revisarla si cambia el contexto.

Preguntas frecuentes

¿Cómo me convierto en arquitecto?
Normalmente, haciendo trabajo arquitectónico antes de tener el cargo: asumir las decisiones transversales de tu equipo, redactar registros de decisiones, ampliar deliberadamente tu amplitud técnica, aprender el dominio de negocio y practicar cómo explicar compromisos a personas no técnicas. Certificaciones como la CPSA-F de iSAQB pueden estructurar ese aprendizaje, pero no sustituyen a la experiencia.

¿Todo equipo necesita un arquitecto?
Todo equipo necesita que la responsabilidad arquitectónica tenga un dueño claro, pero no necesariamente una persona dedicada. Los equipos pequeños suelen asignarla a un líder técnico; las organizaciones más grandes pueden tener arquitectos que dan apoyo cada uno a varios equipos.

¿Cuál es la diferencia entre un líder técnico y un arquitecto?
Un líder técnico suele ser responsable de la entrega técnica de un equipo y dedica gran parte de su tiempo a programar; un arquitecto suele ser responsable de decisiones y cualidades que abarcan varios equipos o sistemas. Los roles se solapan mucho, y en muchas organizaciones el líder técnico es el arquitecto del sistema de ese equipo.

¿Cuánto tiempo debería pasar un arquitecto en reuniones?
No hay una cifra correcta, pero una prueba útil es si las reuniones producen decisiones, entendimiento compartido u obstáculos eliminados. La imagen de Gregor Hohpe del arquitecto que viaja en ascensor entre la sala de juntas y la sala de máquinas recuerda que hay que visitar ambos extremos con regularidad.

ATENCIÓN

Antipatrones en el rol de arquitecto

- **La torre de marfil:** diseños impuestos desde arriba sin implicarse en la implementación, que los equipos ignoran o esquivan.
- **El cuello de botella:** toda decisión, por pequeña que sea, tiene que pasar por el arquitecto, así que la entrega se ralentiza y los equipos dejan de pensar.
- **El experto congelado:** un criterio anclado en tecnologías o en malas experiencias de hace años.
- **Arquitectura dirigida por el currículum:** elegir tecnologías porque le resultan interesantes al arquitecto y no porque sean las adecuadas para el problema.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Por qué incluyen Richards y Ford la política entre las expectativas de un arquitecto?**

Porque casi todas las decisiones arquitectónicas serán cuestionadas por alguien, y una decisión que no consiga apoyo entre presupuestos, prioridades y equipos no se implementará como se pretendía.

▼ **¿Qué capa de la pirámide del conocimiento debería ampliar deliberadamente un arquitecto, y por qué?**

La capa intermedia, lo que sabes que no sabes, porque saber que existen opciones y para qué sirven permite a un arquitecto comparar soluciones.

▼ **Nombra dos formas en que un arquitecto puede seguir implicado en el código sin convertirse en un cuello de botella.**

Dos cualesquiera de las siguientes: construir pruebas de concepto, escribir funciones de aptitud, corregir bugs o pequeñas tareas de deuda, crear herramientas y automatizaciones, y revisar código.

FUENTES

- iSAQB CPSA-Foundation curriculum (LG 01-04: tasks and responsibilities of software architects): https://public.isaqb.org/curriculum-foundation/curriculum-foundation-en.html
- Who Needs an Architect? Martin Fowler, IEEE Software, 2003 (PDF): https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf
- Staff archetypes (Will Larson, StaffEng): https://staffeng.com/guides/staff-archetypes/
- The Architect Elevator (Gregor Hohpe): https://architectelevator.com/

SECCIÓN 1.5 DE 1.8 · PRINCIPIANTE · 18 MIN DE LECTURA

Las partes interesadas, sus inquietudes y cómo las atiende la arquitectura

LA IDEA EN BREVE

Piensa en la construcción de una casa. La familia que vivirá en ella quiere espacio y luz. El banco que presta el dinero quiere que la casa valga lo que el préstamo. La oficina de urbanismo quiere que encaje en la calle. El constructor quiere planos que realmente se puedan construir, y los vecinos no quieren ruido por la noche. Todos ellos son partes interesadas, y lo que preocupa a cada uno es una inquietud. Un sistema de software también tiene sus partes interesadas: usuarios, quienes pagan, desarrolladores, las personas que lo mantienen en funcionamiento, reguladores. La arquitectura es el arte de escucharlos a todos y de decidir abiertamente dónde chocan sus deseos.

LO QUE NECESITAS SABER

- Una parte interesada es cualquier rol, persona, organización o clase de ellos con un interés, derecho, participación o reclamación en un sistema, incluidas las personas que nunca lo usan directamente, como los reguladores y los operadores.
- Una inquietud, en ISO/IEC/IEEE 42010, es cualquier asunto de relevancia o importancia para una parte interesada: cuentan las cualidades, los costos, los riesgos, las restricciones, la viabilidad y las cuestiones organizativas.
- Las inquietudes todavía no son requisitos: recógelas con las palabras de la parte interesada y luego conviértelas en escenarios de atributos de calidad medibles, restricciones o requisitos funcionales.
- La matriz poder/interés ayuda a planificar cómo implicar a cada parte interesada, pero no dice nada sobre qué inquietudes son las más importantes; los grupos con poco poder pueden ser los que más se juegan.
- Las inquietudes en conflicto son normales; el arquitecto las hace explícitas, cuantifica las opciones, nombra a quién decide y registra la justificación.
- Las inquietudes organizan la documentación de la arquitectura: cada punto de vista enmarca ciertas inquietudes, y cada vista muestra cómo las atiende el sistema.

Quiénes son las partes interesadas

Una parte interesada es cualquiera que tenga un interés legítimo en un sistema: las personas que lo usan, lo pagan, lo construyen, lo operan, lo regulan o se ven afectadas por él. ISO/IEC/IEEE 42010:2022, la norma internacional para la descripción de arquitecturas, define

el término de forma amplia como un rol, cargo, individuo, organización o clase de ellos que tiene un interés, derecho, participación o reclamación en aquello que se describe. Dos rasgos de esa definición importan en la práctica. Primero, una parte interesada suele ser una *clase* y no una persona: "los pacientes" o "los ingenieros de guardia" en lugar de un individuo con nombre. Segundo, basta con una reclamación o un derecho: un regulador que nunca inicia sesión en tu sistema sigue siendo una parte interesada, porque puede impedirte operar.

La lista es más larga de lo que espera la mayoría de los equipos. Para un sistema comercial típico incluye:

Los **usuarios** (las personas que manejan el software en el día a día, a menudo varios grupos distintos); los **clientes** (quien paga, que puede no ser usuario en absoluto); el **product owner** o responsable de producto que decide qué se construye; el **patrocinador** que financia el trabajo y responde de su éxito; los **desarrolladores**, que deben entender y modificar el código; los **testers**, que deben poder preparar datos y observar el comportamiento; el **personal de operaciones y los ingenieros de fiabilidad del sitio (SRE)**, que lo despliegan, lo monitorizan y lo recuperan a las tres de la mañana; el **equipo de seguridad**; el **delegado de protección de datos** cuando se tratan datos personales (conforme al RGPD (GDPR) del Reino Unido y de la UE, algunas organizaciones deben designar uno); el **área jurídica y de compras**, a la que le importan las licencias, los contratos y la responsabilidad; el **área de finanzas**, a la que le importan los costos de funcionamiento tanto como los de construcción; los **reguladores y auditores**; y los **integradores externos**, los otros equipos y empresas cuyo software llama al tuyo o es llamado por él.

Algunas partes interesadas faltan a todas las reuniones: los desarrolladores que mantendrán el sistema dentro de cinco años, el servicio de soporte que recibe las quejas, las personas con discapacidad que usarán la interfaz y los ciudadanos afectados por sus decisiones. Parte del trabajo de un arquitecto es hablar por quienes no están en la sala.

Inquietudes: lo que importa a las partes interesadas

En 42010, una inquietud es cualquier asunto de relevancia o importancia para una parte interesada. La palabra es deliberadamente amplia. Una inquietud puede ser una cualidad ("¿seguirá siendo rápida la página de reservas los lunes por la mañana?"), un costo ("¿cuánto costará operarlo al mes?"), un riesgo ("¿qué pasa si se cae el proveedor de SMS?"), una restricción ("¿deben quedarse en el Reino Unido los datos de los pacientes?"), una cuestión de viabilidad ("¿puede nuestro equipo construir esto con las habilidades que tiene?") o un asunto organizativo ("¿qué equipo es responsable de la integración con el sistema de los médicos de cabecera?").

Las inquietudes todavía no son requisitos. Suelen expresarse de forma vaga, con el vocabulario propio de la parte interesada, y a menudo son preocupaciones más que especificaciones. "No quiero otra caída como la del invierno pasado" es una inquietud; "el servicio de reservas está disponible el 99.9% de cada mes natural, medido en el balanceador de carga" es un requisito derivado de ella. Recoger primero la inquietud con las palabras de la parte interesada, y solo después precisarla, te obliga a ser honesto sobre de dónde viene un requisito y por qué importa.

Una misma inquietud suele ser compartida por varias partes interesadas, y una parte interesada suele tener muchas inquietudes. La seguridad es una inquietud del equipo de seguridad, del delegado de protección de datos, del patrocinador (que teme los titulares) y de los pacientes (que temen que se filtren sus historiales). Esa relación de muchos a muchos es exactamente la razón por la que 42010 trata las inquietudes como la bisagra entre las personas y la arquitectura: las decisiones responden a inquietudes, y las vistas se organizan en torno a las inquietudes que atienden.

Partes interesadas típicas y sus inquietudes

Parte interesada	Inquietudes típicas	Lo que la arquitectura debe darles
Público	Velocidad, facilidad de uso, accesibilidad, que el sistema esté disponible cuando se necesita	Objetivos de tiempo de respuesta y de disponibilidad; una capa de interfaz capaz de cumplir las normas de accesibilidad
Clientes y patrocinadores	Valor de negocio, tiempo de comercialización, costo total, riesgo para la reputación	Un plan de entrega creíble, un modelo de costos y una lista de los principales riesgos
Product owner	Con qué rapidez se pueden añadir nuevas funcionalidades, cómo se mantienen abiertas las opciones	Estructura modular alineada con el dominio de negocio; pocas decisiones difíciles de revertir
Desarrolladores	Comprensibilidad, velocidad de compilación y de pruebas, propiedad clara	Límites de módulo limpios, una compilación local rápida, convenciones documentadas
Testers	Capacidad de prueba, datos de prueba controlables, comportamiento observable	Puntos de unión (seams) donde se pueden sustituir las dependencias; entornos de prueba parecidos a producción
Operaciones y SRE	Desplegabilidad, monitorización, recuperación, capacidad, carga de guardias	Comprobaciones de estado, logs, métricas y trazas; rollback seguro; runbooks
Equipo de seguridad	Superficie de ataque, identidad, secretos, respuesta a vulnerabilidades	Límites de confianza, diseño de la autenticación, estrategia de parches y de dependencias
Delegado de protección de datos	Base jurídica, minimización de datos, conservación, derechos de los interesados, transferencias	Un inventario de datos y una vista de flujo de datos; mecanismos de borrado y exportación
Área jurídica y de compras	Licencias, contratos, responsabilidad, dependencia del proveedor (lock-in)	Una lista de componentes y servicios de terceros con sus condiciones
Finanzas	Costo de construcción, costo de funcionamiento, previsibilidad de costos	Un modelo de costos por entorno y por unidad de uso
Reguladores y auditores	Cumplimiento de las normas del sector, evidencia de control	Registros de auditoría, trazabilidad desde el requisito hasta la prueba, decisiones documentadas

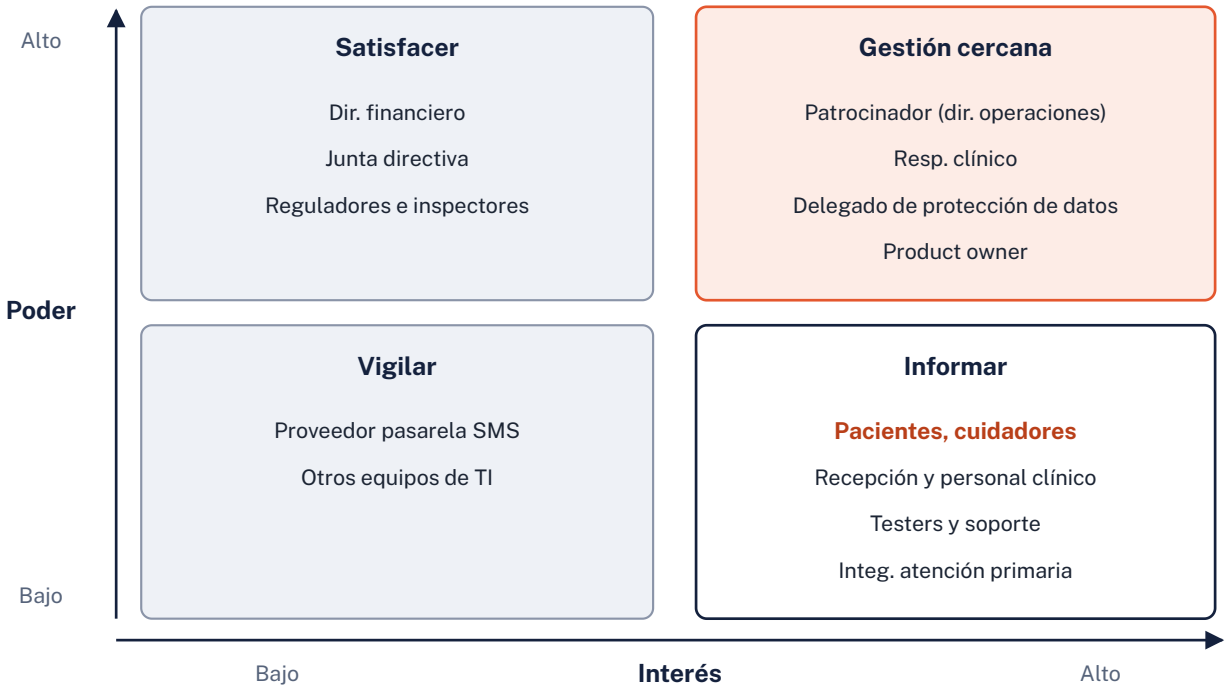
Parte interesada	Inquietudes típicas	Lo que la arquitectura debe darles
Integradores externos	Interfaces estables, versionado, documentación, entornos sandbox de prueba	Contratos de API publicados y una política de versionado

Mapa de las partes interesadas: la matriz poder/interés

Con veinte o más grupos de partes interesadas no puedes dar a cada uno la misma atención. Una herramienta muy usada es la matriz poder/interés, que suele atribuirse a Aubrey Mendelow, cuyo artículo sobre las partes interesadas y el análisis del entorno data de 1981 (a menudo se cita como de 1991). Cada parte interesada se sitúa según el poder que tiene sobre el sistema (para financiarlo, aprobarlo, bloquearlo o cambiarle la forma) y el interés que muestra por él. Los cuadrantes sugieren una estrategia de implicación: **gestionar de cerca** (alto poder, alto interés), **mantener satisfechos** (alto poder, bajo interés), **mantener informados** (bajo poder, alto interés) y **vigilar** (bajo poder, bajo interés).

La matriz trata de la implicación, no de qué inquietudes cuentan. Los pacientes de un hospital tienen poco poder formal sobre el diseño de un sistema de reservas, pero son los que más se juegan, y algunas obligaciones legales (accesibilidad, protección de datos) existen precisamente para proteger a las personas con poco poder. Usa la matriz como recordatorio de con quién hablar y con qué frecuencia, y mantén un juicio aparte sobre qué inquietudes son críticas. Las posiciones también cambian: un regulador con poco interés se convierte en la parte interesada más interesada de todas la semana siguiente a una filtración de datos.

Una matriz poder/interés para un sistema de reservas de un hospital



Los pacientes están en el cuadrante de poco poder y, sin embargo, son los que más se juegan; por eso la matriz orienta la implicación y no la prioridad.

Un análisis ligero de las partes interesadas

1. **Identificar**
   Haz una lluvia de ideas sobre las clases de partes interesadas con el patrocinador y el equipo, y luego contrasta la lista con un conjunto estándar (usuarios, quienes pagan, quienes construyen, quienes operan, quienes protegen, reguladores, integradores). Pregunta a cada grupo que encuentres: ¿con quién más deberíamos hablar?

2. **Recoger las inquietudes**
   Entrevista a cada grupo importante u organiza un taller con él. Anota las inquietudes con sus propias palabras y luego apunta la cualidad, la restricción o el riesgo que hay detrás de cada una.

3. **Mapear y priorizar**
   Sitúa a las partes interesadas en la matriz poder/interés para planificar la implicación. Por separado, ordena las inquietudes según su impacto en el negocio y según cuánto restringen la arquitectura.

4. **Hacer explícitos los conflictos**
   Cuando dos inquietudes tiren en direcciones opuestas, pon el conflicto por escrito, nombra a las personas que deben ponerse de acuerdo y acuerda quién decide.

5 **Trazar y revisar**
Vincula cada decisión significativa con las inquietudes que atiende, y repite el análisis en los hitos importantes, porque las partes interesadas y sus inquietudes cambian.

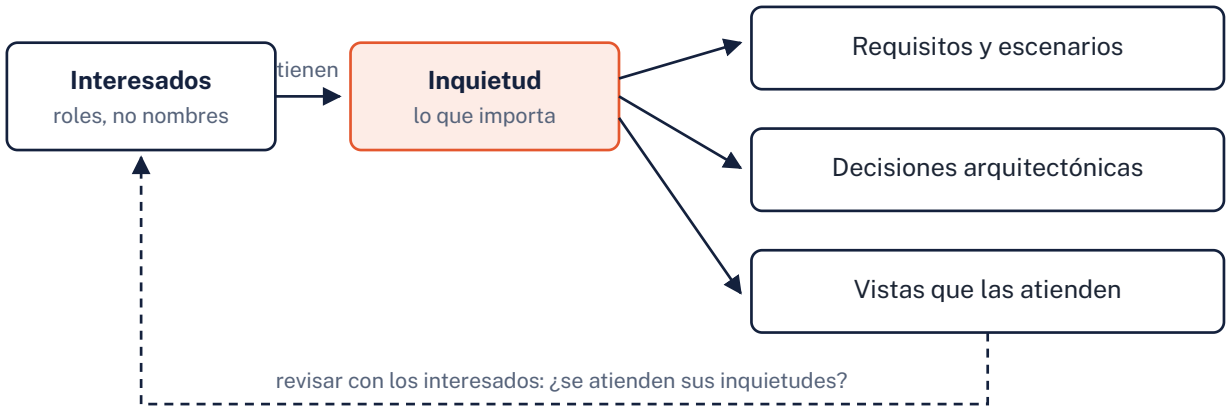
Las inquietudes en conflicto y cómo hacerlas explícitas

Las inquietudes entran en conflicto todo el tiempo, y una arquitectura que contenta a todos suele significar que un conflicto se ha ocultado en lugar de resolverse. El equipo de seguridad quiere que todo inicio de sesión esté protegido con autenticación multifactor; la representante de los pacientes señala que muchos pacientes mayores no tienen teléfono inteligente. Finanzas quiere un único servidor de base de datos pequeño; operaciones quiere una réplica en espera en un segundo centro de datos. El product owner quiere una nueva funcionalidad cada semana; los testers quieren tiempo para hacer pruebas de regresión de la integración con el sistema de administración de pacientes del hospital.

La aportación de un arquitecto no es elegir un ganador en silencio, sino hacer que el conflicto sea visible, cuantificado y tenga un responsable. En la práctica, eso significa cuatro cosas. **Formula ambas inquietudes con precisión**, idealmente como escenarios comprobables, para que la discusión trate de números y no de adjetivos. **Muestra las opciones y sus consecuencias** una al lado de otra: por ejemplo, MFA para el personal y para los pacientes que la elijan, con un código de un solo uso por mensaje de texto o por llamada a un teléfono fijo como alternativa. **Nombra al responsable de la decisión**: algunos compromisos corresponden al patrocinador o a un comité de riesgos, no al arquitecto. **Registra el resultado y su justificación**, para que, cuando alguien pregunte dentro de dos años por qué los pacientes pueden iniciar sesión sin MFA, la respuesta exista. La sección siguiente, Decisiones y compromisos, trata la mecánica de sopesar esas opciones.

Algunos conflictos se disuelven al examinarlos. Puede que el equipo de seguridad no necesite MFA para todas las acciones del paciente, solo para ver los resultados de las pruebas; puede que al grupo de pacientes le parezca bien en ese caso. Preguntar a cada parte *por qué* tiene esa inquietud, y qué resultado teme realmente, a menudo revela un diseño que satisface a ambas.

Adónde van las inquietudes



Las inquietudes son el vínculo entre las personas y la arquitectura: se convierten en requisitos, impulsan las decisiones y determinan qué vistas dibujas.

Cómo alimentan las inquietudes los requisitos y la documentación

Las inquietudes son la materia prima de dos actividades posteriores. La primera es la ingeniería de requisitos. Una inquietud sobre la velocidad se convierte en un requisito de atributo de calidad, que el Capítulo 2 te enseña a escribir como un escenario medible (fuente, estímulo, artefacto, entorno, respuesta y medida de la respuesta): consulta Escribir escenarios de atributos de calidad que se puedan probar. Una inquietud como "los datos de los pacientes deben quedarse en el Reino Unido" se convierte en una *restricción*, una decisión que ya se ha tomado por ti. Las inquietudes que resultan no afectar a la estructura del sistema se quedan como requisitos funcionales o de proyecto ordinarios.

La segunda es la documentación. 42010 organiza la descripción de una arquitectura en torno a las inquietudes: un punto de vista nombra las inquietudes que enmarca y las partes interesadas a las que sirve, y una vista aplica ese punto de vista a tu sistema. Las inquietudes del equipo de operaciones piden una vista de despliegue que muestre los servidores, las regiones y el failover; las del delegado de protección de datos piden una vista de flujo de datos que muestre por dónde viajan los datos personales y dónde se almacenan; los desarrolladores necesitan una vista de módulos que muestre qué depende de qué. Un diagrama que no responde a la inquietud de ninguna parte interesada es decoración. El Capítulo 4 trata esto a fondo en Vistas y puntos de vista.

La trazabilidad une las dos. Si puedes seguir una línea desde una parte interesada, pasando por una inquietud, hasta un requisito, una decisión y una vista, puedes responder a las dos preguntas que todo revisor acaba haciendo: ¿por qué está construido así y nos hemos olvidado de alguien?

EJEMPLO PRÁCTICO

Ejemplo práctico: las partes interesadas de un sistema de reserva de citas de un hospital

Un consorcio hospitalario regional quiere un sistema en línea para que los pacientes reserven, cambien y cancelen sus citas de consultas externas, en sustitución de la reserva por teléfono. Debe integrarse con el sistema de administración de pacientes (PAS) existente y enviar recordatorios por mensaje de texto. Este es el primer análisis de partes interesadas que podría elaborar un arquitecto en las dos primeras semanas.

Pacientes, cuidadores	Inquietudes: reservar con facilidad desde un teléfono, accesibilidad (lectores de pantalla, texto grande), recordatorios, privacidad. Impacto en la arquitectura: frontend web adaptable que cumple WCAG 2.2 AA; sin necesidad de instalar una app; datos mínimos en pantalla.
Recepción y personal clínico	Inquietudes: ninguna reserva duplicada, listas de consulta que se correspondan con la realidad, menos llamadas telefónicas. Impacto: el PAS sigue siendo la única fuente de verdad de las franjas horarias; el sistema de reservas no debe tener su propia copia, que podría desviarse.
Resp. clínico	Inquietudes: las derivaciones urgentes no deben quedar desplazadas por las reservas rutinarias; seguridad clínica. Impacto: reglas de franjas horarias que son responsabilidad de los clínicos y configurables sin un lanzamiento; un caso de seguridad clínica (en Inglaterra, conforme a la norma DCB0129) que abarque el sistema.
Delegado de protección de datos	Inquietudes: los datos de salud son una categoría especial de datos según el RGPD (GDPR) del Reino Unido; minimización; conservación; auditoría. Impacto: una vista de flujo de datos, cifrado en tránsito y en reposo, registros de auditoría de quién vio qué, una política de conservación.
Equipo de operaciones	Inquietudes: los picos de los lunes por la mañana, la capacidad limitada del PAS, el soporte fuera de horario. Impacto: una cola o un limitador de tasa delante del PAS; caché de la disponibilidad de franjas con una caducidad corta; monitorización de los tiempos de respuesta del PAS.
Dir. financiero	Inquietudes: presupuesto fijo, costo de funcionamiento previsible, ahorro en personal del centro de llamadas. Impacto: servicios gestionados en la nube dimensionados para el pico; un modelo de costos por cada mil reservas.
Proveedor del PAS (integrador)	Inquietudes: ninguna carga no soportada sobre su sistema; uso exclusivo de su interfaz publicada. Impacto: integración a través de la interfaz del proveedor con límites de tasa acordados; un conjunto de pruebas de contrato.

Principal conflicto encontrado	Operaciones quiere guardar en caché la disponibilidad de franjas para proteger el PAS; el personal de recepción teme reservas duplicadas por datos desactualizados. Resolución propuesta: caché solo para mostrar (30 segundos), pero confirmar cada reserva contra el PAS en tiempo real, con un mensaje claro si la franja acaba de ocuparse.

Siete grupos de partes interesadas produjeron al menos cinco decisiones con consecuencias estructurales (única fuente de verdad, limitación de tasa, registros de auditoría, reglas configurables, frontend accesible) antes de que se hubiera diseñado en detalle ninguna funcionalidad.

El conflicto de mayores consecuencias se dio entre dos grupos de perfil bajo, operaciones y el personal de recepción, ninguno de los cuales habría dominado una reunión del comité de dirección.

Cada fila se puede trazar hacia delante hasta un escenario de atributo de calidad, un registro de decisión y una vista, y hacia atrás hasta una clase de personas con nombre.

ATENCIÓN

Errores comunes

Estas son las formas en que el trabajo con las partes interesadas sale mal con más frecuencia.

- Tratar "el negocio" como una sola parte interesada. Ventas, finanzas, cumplimiento normativo y atención al cliente quieren cosas distintas, a menudo en conflicto.
- Olvidarse de los operadores. Los sistemas diseñados solo en torno a los usuarios y los desarrolladores suelen ser difíciles de desplegar, monitorizar y recuperar.
- Confundir el volumen con la importancia. La persona que asiste a todas las reuniones no es necesariamente aquella cuya inquietud hundirá el proyecto.
- Fiarse de intermediarios. Un product owner puede representar a los usuarios, pero no siempre puede decirte lo que vive realmente un usuario que depende de la accesibilidad o una enfermera del turno de noche. Habla con algunos reales.
- Hacer el análisis una sola vez. Las partes interesadas llegan y se van, la normativa cambia y un nuevo socio de integración trae nuevas inquietudes. Revisa la lista en cada hito importante.

Preguntas frecuentes

¿De verdad el equipo de desarrollo es una parte interesada? Somos quienes lo construimos.
Sí. Los desarrolladores conviven con la arquitectura más tiempo que casi nadie, y sus inquietudes (tiempo de compilación, capacidad de prueba, propiedad clara) se traducen directamente en velocidad de entrega. La definición de 42010 no distingue entre quienes construyen y quienes usan.

¿Cuál es la diferencia entre una inquietud de una parte interesada y un requisito?
Una inquietud es el interés de fondo, a menudo vago ("quiero que sea fiable"). Un requisito es un enunciado preciso y verificable derivado de una o más inquietudes ("99.9% de disponibilidad mensual"). Varios requisitos pueden servir a una misma inquietud, y algunas inquietudes se convierten en restricciones.

¿Quién decide cuando dos partes interesadas con mucho poder no están de acuerdo?
Normalmente no el arquitecto solo. El arquitecto plantea las opciones y las consecuencias; la decisión corresponde a quien responde del resultado, normalmente el patrocinador o un comité de gobernanza. Acordar de antemano quién decide cada tipo de compromiso ahorra semanas de discusión.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **Nombra tres partes interesadas de un sistema comercial típico que rara vez asisten a las reuniones de diseño.**

Por ejemplo: los desarrolladores que lo mantendrán en años futuros, el servicio de soporte, los usuarios con discapacidad, el personal de operaciones de guardia por la noche y los ciudadanos afectados por sus decisiones.

▼ **Un patrocinador dice "tiene que ser seguro". ¿Es una inquietud o un requisito?**

Una inquietud. Hay que refinarla en requisitos específicos y comprobables, por ejemplo sobre autenticación, cifrado o registros de auditoría, antes de que pueda guiar las decisiones.

▼ **¿Por qué una inquietud, como la seguridad, puede pertenecer a varias partes interesadas a la vez?**

Porque distintas personas temen distintas consecuencias de la misma cualidad: el equipo de seguridad teme los ataques, el delegado de protección de datos teme las filtraciones de datos personales, el patrocinador teme el daño a la reputación y los usuarios temen que se filtren sus datos.

▼ **¿Qué vincula una inquietud de una parte interesada con una vista de la arquitectura?**

Un punto de vista: indica a qué inquietudes y partes interesadas sirve y cómo modelarlas, y la vista lo aplica al sistema.

FUENTES

- ISO/IEC/IEEE 42010:2022 Software, systems and enterprise: Architecture description (ISO): https://www.iso.org/standard/74393.html
- ISO/IEC/IEEE 42010 summary (arc42 quality model): https://quality.arc42.org/standards/iso-42010
- Stakeholder analysis for UX projects (Nielsen Norman Group): https://www.nngroup.com/articles/stakeholder-analysis/
- arc42 section 3: Context and scope: https://docs.arc42.org/section-3/

SECCIÓN 1.6 DE 1.8 · INTERMEDIO · 20 MIN DE LECTURA

Decisiones arquitectónicas y compromisos: reversibilidad y momento oportuno

ANTES DE LEER ESTO

Esta sección se basa en:

- 1.2 Arquitectura frente a diseño
- 1.5 Partes interesadas e inquietudes

Términos anteriores que utiliza: Parte interesada, Inquietud, Arquitectónicamente significativo

LA IDEA EN BREVE

Elegir un vehículo es un compromiso: un motor más grande gasta más combustible, un vehículo más pequeño es más fácil de estacionar pero lleva menos. Nadie lo consigue todo. Algunas elecciones son fáciles de deshacer, como escoger un color de pintura que puedes cambiar más adelante; otras, como comprar la casa que está junto a una autopista, son muy difíciles de deshacer. La arquitectura de software está llena de elecciones como estas. Los buenos arquitectos toman rápido las que son fáciles de deshacer, se toman su tiempo con las difíciles de deshacer, deciden en el momento adecuado, ni demasiado pronto ni demasiado tarde, y anotan por qué eligieron, para que quienes vengan después lo entiendan.

LO QUE NECESITAS SABER

- Toda elección arquitectónica significativa es un compromiso: mejora algunas cualidades a costa de otras, y hay que encontrar y declarar sus desventajas.
- Una decisión bien formada registra su contexto, las opciones consideradas, los criterios, la decisión con su justificación, las consecuencias y su estado.
- La reversibilidad es la propiedad clave de una decisión: las puertas de doble sentido deben decidirse rápido y localmente; las de un solo sentido, de forma deliberada y con las personas que cargan con las consecuencias.
- El último momento responsable es el punto en el que un retraso eliminaría una opción importante; decidir entonces aprovecha el mayor conocimiento sin dejar que la decisión se tome por omisión.
- Las abstracciones en los límites, las feature flags, la migración incremental y los spikes hacen las decisiones más reversibles, con un costo que solo merece la pena pagar cuando tanto la incertidumbre como el costo de revertir son altos.

- Las matrices de decisión ponderadas y los registros de decisiones de arquitectura hacen visible el razonamiento y lo exponen a cuestionamiento; acuerda los criterios y los pesos antes de puntuar, y comprueba hasta qué punto el resultado depende de ellos.

Todo es un compromiso

Toda elección arquitectónica significativa compra algo renunciando a otra cosa. Dividir un sistema en servicios desplegables de forma independiente compra autonomía de los equipos y escalado independiente, y lo paga con latencia de red, complejidad operativa y una consistencia de datos más difícil. La caché compra velocidad y lo paga con datos desactualizados y lógica de invalidación. Una base de datos gestionada en la nube compra la liberación de parches y copias de seguridad, y lo paga con costo y dependencia de un único proveedor. Mark Richards y Neal Ford ponen esto al principio de *Fundamentals of Software Architecture* (2020; segunda edición 2025) como su primera ley: “Todo en la arquitectura de software es un compromiso.” Un corolario que extraen es que, si crees haber encontrado algo que no es un compromiso, probablemente todavía no has identificado el compromiso.

Su segunda ley es que el *porqué* es más importante que el *cómo*. Cualquiera puede mirar un sistema en funcionamiento y ver cómo está construido: tres servicios, un intermediario de mensajes (message broker), una base de datos documental. Lo que no se puede recuperar del código es el porqué: qué alternativas se consideraron, qué inquietudes ganaron, qué riesgos se aceptaron. Sin el porqué, los ingenieros posteriores o bien conservan elecciones que ya no tienen sentido o bien deshacen elecciones cuyas razones siguen vigentes. La segunda edición añade una tercera ley: la mayoría de las decisiones arquitectónicas no son binarias, sino que se sitúan en un espectro entre extremos. Rara vez eliges entre "monolito" y "microservicios"; eliges cuántas unidades desplegables, divididas según qué líneas y compartiendo qué.

Pensar en compromisos es el hábito que más claramente separa el pensamiento arquitectónico del entusiasmo. Cuando alguien propone una tecnología, las preguntas útiles no son "¿es buena?", sino "¿qué cuesta, qué hace más difícil y son aceptables esos costos dadas nuestras inquietudes?". La sección anterior mostró de dónde vienen esas inquietudes.

La anatomía de una decisión

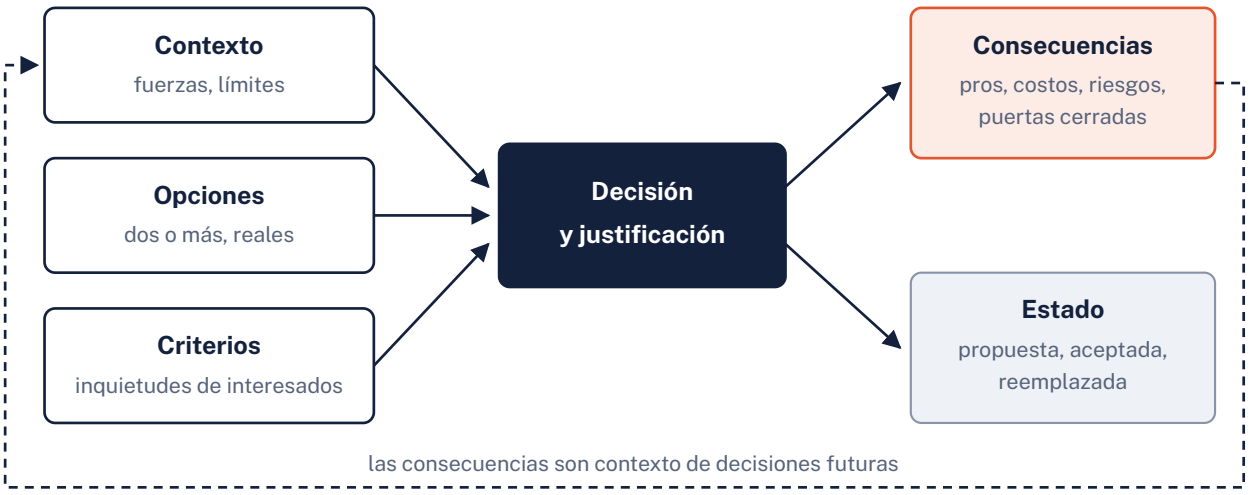
Una decisión arquitectónica es más que su resultado. Una decisión que se puede explicar, revisar y reconsiderar tiene seis partes, y nombrarlas es el primer paso para tomar bien las decisiones.

Contexto: las fuerzas en juego. El problema que se resuelve, los requisitos de atributos de calidad que influyen en él, las restricciones (presupuesto, plazos, normativa, sistemas existentes, habilidades del equipo) y los supuestos que se hacen. **Opciones**: al menos dos

alternativas genuinas, a menudo incluidas "no hacer nada" o "aplazar". Una decisión en la que solo se ha considerado una opción es un anuncio, no una decisión. **Criterios**: cómo se juzgarán las opciones, derivados de las inquietudes de las partes interesadas y ponderados según su importancia. **Decisión y justificación**: la opción elegida y, sobre todo, por qué superó a las demás según los criterios. **Consecuencias**: qué se vuelve más fácil, qué se vuelve más difícil, qué riesgos nuevos aparecen y qué opciones futuras quedan cerradas. **Estado**: propuesta, aceptada, obsoleta o reemplazada, además de quién tomó la decisión y cuándo.

Las consecuencias merecen una atención especial porque es ahí donde se pone a prueba la honestidad. Toda opción tiene desventajas; ponerlas por escrito protege al equipo de las sorpresas y da a quien tome la siguiente decisión un relato justo. Las consecuencias de una decisión también pasan a formar parte del contexto de la siguiente: una vez que has elegido la mensajería asíncrona, debes decidir cómo tratar los mensajes duplicados y desordenados.

Anatomía de una decisión arquitectónica



El contexto, las opciones y los criterios alimentan una decisión; sus consecuencias y su estado quedan registrados y se convierten en contexto para decisiones posteriores.

Puertas de un solo sentido y puertas de doble sentido

La propiedad individual más útil de una decisión es su reversibilidad: con qué poco costo y con qué seguridad se puede deshacer. En su carta de 2015 a los accionistas de Amazon, Jeff Bezos distinguía las decisiones de **Tipo 1**, que tienen grandes consecuencias y son irreversibles o casi ("puertas de un solo sentido"), de las decisiones de **Tipo 2**, que se pueden cambiar y revertir ("puertas de doble sentido"). Las decisiones de Tipo 1 merecen una deliberación lenta y cuidadosa, y consultas. Las decisiones de Tipo 2 deben tomarlas rápido personas o grupos pequeños con buen criterio, porque si resultan equivocadas puedes volver a cruzar la puerta. Su advertencia era que las grandes organizaciones tienden a aplicar a todo el pesado proceso del Tipo 1, lo que las vuelve lentas y timoratas.

La idea se traslada directamente a la arquitectura. Una puerta de un solo sentido en el software es una decisión que, para deshacerse, exigiría reescribir grandes partes del sistema, migrar grandes cantidades de datos, renegociar contratos o volver a formar a muchas personas: el lenguaje de programación principal de una base de código grande, el modelo de datos de un sistema de registro (system of record), la API pública con la que se integran cientos de socios, un compromiso de nube de varios años. Entre las puertas de doble sentido están la elección de una biblioteca de logging detrás de tu propia interfaz, la estructura interna de un módulo, el ajuste de una caché, o cuál de dos formatos de mensaje equivalentes usar dentro de un servicio.

Dos advertencias. Primero, la reversibilidad es un espectro, no algo binario: la mayoría de las decisiones se sitúan en algún punto entre "trivial de deshacer" e "imposible de deshacer", y el costo de revertir suele crecer a medida que más código, más datos y más personas pasan a depender de la elección. Una decisión que es una puerta de doble sentido en el primer mes puede convertirse en una puerta de un solo sentido hacia el duodécimo. Segundo, a menudo puedes cambiar, mediante el diseño, el tipo de puerta ante el que estás, como muestra más abajo el apartado sobre la reversibilidad.

El momento: el último momento responsable

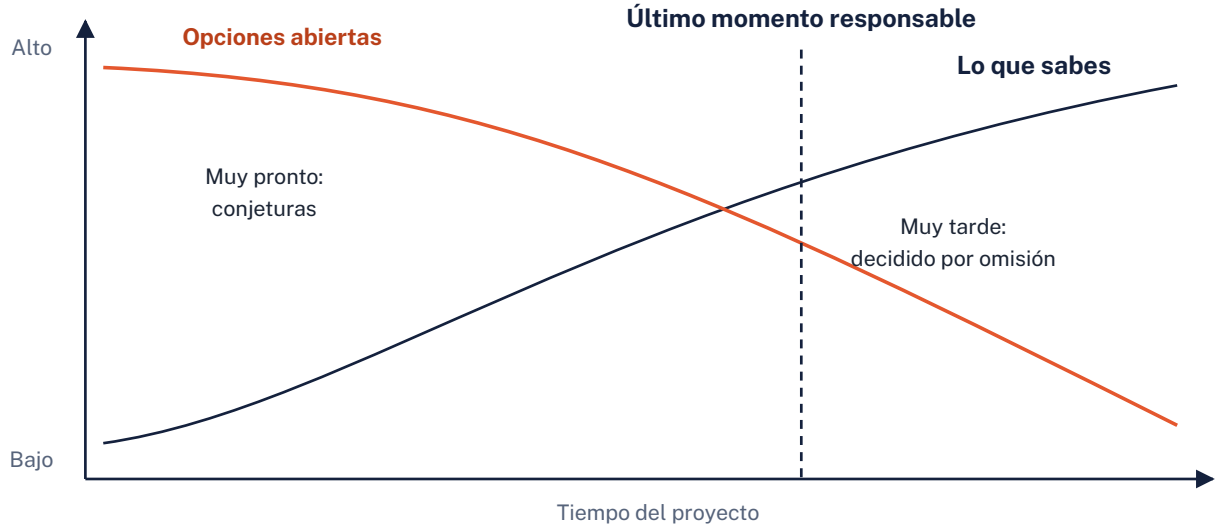
¿Cuándo debe tomarse una decisión? Mary y Tom Poppendieck, en *Lean Software Development: An Agile Toolkit* (2003), popularizaron la idea de decidir en el último momento responsable: el momento en el que no decidir eliminaría una alternativa importante. Antes de ese momento decides con menos información de la que podrías tener; después, la decisión se toma por ti por omisión, normalmente mal.

El principio funciona en los dos sentidos. Desaconseja decidir demasiado pronto: fijar la tecnología de base de datos en la primera semana, antes de entender la forma de los datos y los patrones de acceso, es adivinar. Igualmente desaconseja decidir demasiado tarde. Si el equipo ha empezado a escribir código de persistencia basado en un supuesto no expresado, la decisión ya se ha tomado, solo que no de forma consciente. Si compras necesita seis semanas para aprobar un proveedor de nube, el último momento responsable para esa elección es al menos seis semanas antes de que necesites el entorno.

Los profesionales no se ponen de acuerdo sobre hasta dónde llevar esto. Los críticos señalan que el "último momento responsable" se usa fácilmente mal como licencia para procrastinar, que algunas decisiones (la estructura de los equipos, el lenguaje de programación) dan forma a todo lo que sigue y por eso deben tomarse pronto, y que la indecisión tiene su propio costo: los equipos no pueden trabajar en paralelo sin interfaces acordadas. Los defensores responden que el principio nunca significó "lo más tarde posible", sino "lo más tarde que sea responsable",

y que identificar el momento es en sí mismo un acto de planificación. Ambas posturas coinciden en el núcleo práctico: saber qué decisiones están pendientes, saber cuándo debe tomarse cada una y usar el tiempo hasta entonces para aprender.

Decidir demasiado pronto, demasiado tarde y en el momento adecuado



El conocimiento crece a medida que avanza el proyecto mientras las opciones abiertas se cierran; el último momento responsable es el punto más tardío antes de que desaparezca una opción importante.

SI ESTO ES NUEVO PARA TI

Paso a paso: cuándo elegir la base de datos del servicio de pedidos

La idea más difícil de esta sección es el último momento responsable: una decisión no debe tomarse ni lo antes ni lo más tarde posible, sino en el último punto antes de que esperar elimine una opción importante. Sigue una decisión a lo largo del tiempo: la base de datos del nuevo servicio de pedidos de la empresa de reparto de comida, que se puntúa más adelante en el ejemplo práctico de esta sección.

- Primera semana. El equipo podría elegir una base de datos enseguida, pero todavía no sabe cómo se guardarán y se leerán los pedidos. Elegir ahora sería adivinar: decidir demasiado pronto.
- Así que el equipo espera y usa el tiempo para aprender. Descubre, por ejemplo, que cada pedido y su pago deben guardarse juntos, todo o nada, como una sola transacción: exactamente el tipo de hecho en el que debe basarse la elección de una base de datos.
- Pero esperar tiene un plazo. Supón que cualquier nuevo almacén de datos necesita una revisión de seguridad de tres semanas, y que la base de datos debe estar funcionando para las pruebas en la semana nueve. Eso hace de la semana seis, como muy tarde, el último momento responsable para la elección.
- También hay un plazo más silencioso. Si los desarrolladores empiezan a escribir el código que guarda los pedidos, y ese código da por hecha en silencio una base de datos concreta, la elección se ha tomado por omisión: nadie la decidió, pero cambiarla ahora significa reescribir ese código.
- Así que el equipo nombra la decisión, le asigna un responsable y una fecha anterior a ambos plazos, decide entonces con lo que ha aprendido y registra los supuestos en los que se basa.
- **La lección:** el último momento responsable no significa 'lo más tarde posible'. Sabe qué decisiones están pendientes y cuándo debe tomarse cada una, usa el tiempo hasta entonces para aprender y decide antes de que la elección se tome por ti.

Diseñar para la reversibilidad

La forma más barata de tomar una decisión difícil suele ser convertirla en una más fácil. Varias técnicas reducen el costo de cambiar de opinión más adelante, a cambio de algo de esfuerzo ahora.

Abstracción en el límite. Pon una interfaz estrecha propia entre tu lógica central y una tecnología que podrías sustituir, como un proveedor de pagos, un intermediario de mensajes o la API de un modelo de lenguaje grande. Así, cambiar la tecnología más adelante afecta a un adaptador en lugar de a toda la base de código. La idea se remonta a la ocultación de

información de David Parnas (1972) y reaparece en la arquitectura hexagonal (Capítulo 5). **Feature flags.** Entrega el nuevo comportamiento desactivado, actívalo para una pequeña parte de los usuarios y vuelve a desactivarlo en segundos si se comporta mal; esto convierte los lanzamientos en puertas de doble sentido. **Migración incremental.** Ejecuta lo antiguo y lo nuevo en paralelo y traslada el tráfico de forma gradual en lugar de cambiar en un solo paso. **Spikes y pruebas de concepto.** Dedica unos días a construir la parte más arriesgada con una tecnología candidata antes de comprometerte con ella. **Salidas contractuales.** Prefiere compromisos más cortos, formatos de datos estándar y la posibilidad de exportar tus datos.

Esto es el *pensamiento en opciones*, tomado de las finanzas: pagas ahora una pequeña prima (un adaptador, una flag, un spike) para comprar el derecho, pero no la obligación, de cambiar de rumbo más adelante. Como cualquier opción, solo merece la pena comprarla cuando la prima es pequeña en relación con el valor de la flexibilidad. Envolver cada biblioteca en una abstracción "por si la cambiamos" es una forma clásica de añadir costo e indirección a cambio de una flexibilidad que nadie usa. Compra opciones sobre las decisiones que son a la vez inciertas y costosas de revertir; para todo lo demás, decide y sigue adelante.

GUÍA DE DECISIÓN

Guía de decisión: ¿decidir ahora o aplazar?

Usa esto para cualquier decisión que pueda ser arquitectónicamente significativa. Recorre las preguntas en orden.

Q1 **¿Hay otro trabajo bloqueado, o desaparecerá pronto una opción importante si nadie decide (por ejemplo, un plazo de compras o un equipo que empieza a construir sobre un supuesto)?**

- **Sí** Estás en el último momento responsable. Decide ahora con la información que tienes, registra la decisión y sus supuestos, y fija una fecha para revisarla.
- **No** Pasa a la siguiente pregunta.

Q2 **¿La decisión es barata y segura de revertir (una puerta de doble sentido)?**

- **Sí** Decide rápido, en el nivel sensato más bajo (a menudo, el equipo responsable del código), regístrala brevemente y sigue adelante. No pases una semana debatiéndola.
- **No** Es una puerta de un solo sentido. Pasa a la siguiente pregunta.

Q3 **¿Puedes hacerla más reversible con poco costo, por ejemplo con una interfaz alrededor de la tecnología, una feature flag o una vía de migración incremental?**

- **Sí** Compra primero esa opción y luego trata el resto de la decisión como una puerta de doble sentido.
- **No** Pasa a la siguiente pregunta.

Q4 **¿Cambiaría la elección la información que puedes obtener de forma realista antes del plazo (un spike, una prueba de carga, una conversación con una parte interesada)?**

- **Sí** Aplaza de forma deliberada: nombra la decisión, el responsable, la fecha en la que debe tomarse y el experimento que la informará.
- **No** Esperar más no ayudará. Haz ahora una comparación estructurada (criterios, pesos, opciones), implica a las partes interesadas que cargan con las consecuencias y escribe un registro de decisiones de arquitectura.

Trampas cognitivas

Los arquitectos son humanos, y las decisiones arquitectónicas se toman bajo incertidumbre, presión de tiempo y presión social: condiciones ideales para los sesgos cognitivos. Conocer los más comunes por su nombre hace más fácil detectarlos en uno mismo y cuestionarlos con amabilidad en los demás.

Trampas que distorsionan las decisiones arquitectónicas

Trampa	Cómo se manifiesta	Contramedida
Anclaje	La primera opción que se menciona, o la primera estimación que se da, se convierte en el punto de referencia con el que se juzgan todas las demás opciones.	Genera varias opciones de forma independiente antes de discutir ninguna; pide a las personas que anoten sus estimaciones antes de compartirlas.
Costo hundido	"Llevamos ocho meses con este framework, no podemos abandonarlo ahora", aunque lo ya gastado está perdido decidas lo que decidas.	Juzga las opciones solo por sus costos y beneficios futuros; pregúntate qué elegirías si empezaras hoy.
Pensamiento de grupo	Un equipo cohesionado converge rápido, discrepar parece desleal y nadie expresa la objeción obvia.	Asigna a alguien para que defienda la postura contraria; recoge las opiniones por escrito antes de la reunión; invita a un revisor externo.
Desarrollo guiado por el currículum	Se elige una tecnología porque es emocionante aprenderla o queda bien en un currículum, no porque se ajuste al problema.	Exige que cada elección se justifique frente a los criterios acordados; presupuesta por separado el tiempo de aprendizaje.
Moda y efecto arrastre	"Todo el mundo se está pasando a X" sustituye al análisis; las charlas de conferencias de empresas muy grandes se aplican a equipos pequeños.	Pregunta qué problema resolvió X a quienes lo adoptaron y si tú tienes ese problema a esa escala.
Sesgo de confirmación	Se buscan y se recuerdan las pruebas a favor de la opción preferida; las pruebas en contra se justifican.	Anota de antemano qué resultado te haría cambiar de opinión y después ejecuta el spike o la prueba.

EJEMPLO PRÁCTICO

Ejemplo práctico: una matriz de decisión ponderada

Una empresa de reparto de comida está construyendo un nuevo servicio de pedidos. Los pedidos implican pagos y nunca deben perderse ni cobrarse dos veces. El equipo de seis personas conoce bien PostgreSQL y no tiene un administrador de bases de datos dedicado. Hay tres opciones sobre la mesa: (A) un servicio gestionado de PostgreSQL de su proveedor de nube, (B) la base de datos documental gestionada y propietaria del proveedor, y (C) PostgreSQL autoalojado en máquinas virtuales. El equipo construye una matriz de decisión ponderada: acuerda criterios y pesos a partir de las inquietudes de las partes interesadas (5 = lo más importante) y después puntúa cada opción de 1 (deficiente) a 5 (excelente).

Integridad transaccional (peso 5)	A: 5, B: 3, C: 5. Las transacciones de varias filas son centrales para los pedidos y los pagos; el almacén documental las admite con más restricciones.
Esfuerzo operativo (peso 4)	A: 4, B: 5, C: 2. Autoalojar significa que los parches, las copias de seguridad y el failover quedan en manos de un equipo sin especialista en bases de datos.
Familiaridad del equipo (peso 3)	A: 5, B: 2, C: 4.
Costo con la carga prevista (peso 3)	A: 3, B: 4, C: 4.
Margen de escalado (peso 2)	A: 3, B: 5, C: 3. El crecimiento previsto es de unas tres veces en dos años, muy por debajo del límite de una sola instancia grande de PostgreSQL.
Reversibilidad y dependencia del proveedor (lock-in) (peso 2)	A: 4, B: 2, C: 5. PostgreSQL estándar puede trasladarse entre proveedores; la API propietaria no.
Totales ponderados (máximo 95)	A: 25 + 16 + 15 + 9 + 6 + 8 = **79**. B: 15 + 20 + 6 + 12 + 10 + 4 = **67**. C: 25 + 8 + 12 + 12 + 6 + 10 = **73**.
Comprobación de sensibilidad	Sube el margen de escalado a peso 5 (como si se esperara un crecimiento de cincuenta veces): A 88, B 82, C 82. Baja el esfuerzo operativo a peso 1 (como si un equipo de plataforma les gestionara las bases de datos): A 67, B 52, C 67, un empate entre A y C.

Decisión	Opción A, PostgreSQL gestionado. Gana con los pesos acordados y se mantiene primera o empatada en primer lugar ante cambios plausibles. Consecuencias registradas: una factura mensual más alta que con el autoalojamiento; dependencia de las ventanas de mantenimiento del proveedor; un plan para mantener estándar el esquema y las consultas, de modo que la opción C siga siendo una alternativa.

Los números no toman la decisión; hacen visible el razonamiento. Quien no esté de acuerdo puede señalar un peso o una puntuación concretos en lugar de discutir sobre el conjunto.

La comprobación de sensibilidad es el paso más informativo: muestra de qué supuestos depende el resultado. Aquí el resultado depende sobre todo de la falta de capacidad del equipo para operar bases de datos.

Las matrices se pueden manipular eligiendo los criterios y los pesos después de conocer la opción preferida. Acuerda los criterios y los pesos antes de puntuar, idealmente con las partes interesadas cuyas inquietudes representan.

Registrar las decisiones: un adelanto de los ADR

La segunda ley dice que el porqué importa más que el cómo, así que el porqué debe ponerse por escrito. El formato ligero más usado es el registro de decisiones de arquitectura (ADR), propuesto por Michael Nygard en una entrada de blog de 2011. Un ADR es un archivo de texto breve, guardado con el código en el control de versiones, que registra una decisión usando la anatomía descrita arriba: título, estado, contexto, decisión y consecuencias. Los ADR nunca se editan para cambiar la historia; una decisión posterior reemplaza a una anterior y enlaza con ella, de modo que el registro cuenta la historia de cómo evolucionó la arquitectura.

El Capítulo 4 trata las plantillas, las herramientas y los hábitos que mantienen útiles los ADR en Registros de decisiones de arquitectura (ADR). El ejemplo de abajo registra la decisión del ejemplo práctico.

```
ADR 7: Use managed PostgreSQL for the order service

Status: Accepted (2026-10-12)

Context
Orders and payments need multi-row transactions and must never be lost.
The team knows PostgreSQL well and has no database administrator.
Expected growth is about 3x in two years.

Decision
We will use the cloud provider's managed PostgreSQL service.

Consequences
+ Backups, patching and failover are handled by the provider.
+ Standard SQL keeps self-hosting or another provider open as a fallback.
- Monthly cost is about 30% higher than self-hosting at current load.
- Maintenance windows are set by the provider.
Revisit if write volume exceeds what one primary instance can handle.
```

Un ADR breve para la decisión sobre la base de datos del servicio de pedidos, siguiendo la estructura de Nygard

ATENCIÓN

Errores comunes

- Debatir largamente las puertas de doble sentido mientras se cruzan de pasada puertas de un solo sentido ("por ahora usaremos la tabla de clientes compartida y ya está").
- Considerar solo una opción. Si no puedes nombrar una alternativa creíble, no has entendido la decisión.
- Ocultar las desventajas. Unas consecuencias que solo enumeran beneficios acabarán descubiertas, y la confianza en el arquitecto se irá con ellas.
- Decidir por antigüedad o por quién habla más alto, en lugar de por criterios acordados de antemano.
- No revisar nunca. Las decisiones se basan en supuestos; regístralos y revisa la decisión cuando cambie un supuesto.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Cuál es la diferencia entre la primera y la segunda ley de Richards y Ford?**

La primera dice que toda elección arquitectónica es un compromiso. La segunda dice que el razonamiento detrás de una elección (el porqué) importa más que su implementación (el cómo), porque el porqué no se puede recuperar del código.

▼ Da un ejemplo de puerta de doble sentido y otro de puerta de un solo sentido en una aplicación web típica.

De doble sentido: la biblioteca de logging usada detrás de tu propia interfaz, o el ajuste de caducidad de una caché. De un solo sentido: el modelo de datos de un sistema de registro, o una API pública de la que ya dependen muchos socios.

▼ ¿Por qué decidir demasiado pronto puede ser tan perjudicial como decidir demasiado tarde?

Demasiado pronto significa decidir con menos información de la que podrías haber tenido, a menudo adivinando. Demasiado tarde significa que la decisión se toma en la práctica por omisión, mediante supuestos ya incorporados al código o a los planes, o que una opción ha caducado.

▼ ¿Qué aporta una comprobación de sensibilidad a una matriz de decisión ponderada?

Muestra si la opción ganadora sigue ganando cuando los pesos o las puntuaciones cambian dentro de rangos plausibles, lo que revela de qué supuestos depende la decisión.

FUENTES

- Documenting Architecture Decisions, Michael Nygard (2011): https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions
- Architectural Decision Records (adr.github.io): https://adr.github.io/
- Amazon.com 2015 letter to shareholders (SEC filing): https://www.sec.gov/Archives/edgar/data/1018724/000119312516530910/d168744dex991.htm
- Feature Toggles (aka Feature Flags), Pete Hodgson on martinfowler.com: https://martinfowler.com/articles/feature-toggles.html

SECCIÓN 1.7 DE 1.8 · INTERMEDIO · 20 MIN DE LECTURA

Deuda técnica, erosión y deriva de la arquitectura

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.3 Organizar el código
- 0.8 Cómo se hace el software
- 1.3 Por qué importa

Términos anteriores que utiliza: Módulo, Dependencia, Costo del cambio, Control de versiones

LA IDEA EN BREVE

Imagina que te mudas de casa con prisa y lo metes todo en cajas sin etiquetas. Te mudaste a tiempo, que era lo importante, pero ahora, cada vez que necesitas algo, pierdes diez minutos buscándolo. Esos minutos perdidos son los intereses. Desempacar y etiquetar bien es saldar la deuda. Con el software pasa lo mismo: los atajos, o los diseños que tenían sentido antes de que entendieras mejor el problema, hacen más lento cada cambio posterior. Un poco de esto está bien y a veces es sensato. El peligro es dejar que se acumule hasta que el equipo pase la mayor parte de su tiempo buscando entre cajas.

LO QUE NECESITAS SABER

- Ward Cunningham acuñó la metáfora de la deuda en 1992 para explicar que se entrega código que refleja el entendimiento actual y luego se refactoriza a medida que ese entendimiento crece; nunca fue una licencia para escribir código descuidado.
- El principal es el costo de corregir el problema; el interés es el costo adicional que se paga en cada cambio mientras el problema persiste, por lo que la deuda en código que nunca cambia no cuesta nada.
- El cuadrante de Fowler clasifica la deuda como deliberada o inadvertida y como prudente o imprudente; la deuda prudente es un subproducto normal de entregar y aprender, y la deuda imprudente es un problema de habilidades o de disciplina.
- La deuda arquitectónica, en la estructura y no en el código, es la más costosa, porque saldarla afecta a muchos módulos, a muchos equipos y, a menudo, a los datos.
- Perry y Wolf (1992) distinguieron la erosión, causada por violaciones de la arquitectura, de la deriva, causada por la falta de atención a ella.

- Los hotspots (código complejo que cambia con frecuencia), el acoplamiento de cambio, los ciclos de dependencias y las tendencias de la entrega hacen visible la deuda; un registro, una capacidad protegida y la refactorización junto con las funcionalidades la mantienen bajo control.

Lo que quería decir realmente Ward Cunningham

El término deuda técnica procede del informe de experiencia de Ward Cunningham sobre el sistema de gestión de carteras WyCash, presentado en la conferencia OOPSLA en 1992. Escribió que entregar código por primera vez es como endeudarse: un poco de deuda acelera el desarrollo, siempre que se pague pronto con una reescritura, y el peligro llega cuando no se paga: “Cada minuto dedicado a un código que no es del todo correcto cuenta como interés de esa deuda.” Organizaciones de ingeniería enteras, advirtió, pueden quedar paralizadas bajo el peso de la deuda.

La metáfora se malinterpreta a menudo como un permiso para escribir código chapucero. Cunningham explicó más tarde (en un breve video de 2009) que nunca quiso decir eso. Su argumento trataba del *entendimiento*. Un equipo entrega código que refleja su entendimiento actual y parcial de un problema, porque esa es la forma más rápida de aprender del uso real. A medida que el entendimiento mejora, el código deja de coincidir con lo que el equipo sabe ahora, y cada cambio tiene que sortear ese desajuste. Saldar la deuda significa aplicar refactorización al código para que refleje lo aprendido. Eso solo funciona si, para empezar, el código se escribió con la limpieza suficiente para poder refactorizarlo. Según esta lectura, la deuda es un subproducto del aprendizaje, no del descuido.

Desde entonces, el uso se ha ampliado. Hoy "deuda técnica" abarca cualquier carencia de calidad interna que haga que los cambios futuros sean más lentos o más arriesgados, tanto si surgió del aprendizaje como de atajos deliberados, del desconocimiento o de la simple dejadez. El significado amplio es útil siempre que recuerdes que los distintos tipos requieren respuestas distintas.

El principal y los intereses

El vocabulario financiero es lo que hace que la metáfora sea tan eficaz en las conversaciones con personas que no son ingenieros. El principal es el esfuerzo necesario para corregir el problema: reestructurar el módulo, separar la base de datos, eliminar el ciclo. El interés es el esfuerzo adicional que se paga en cada trabajo que toca el problema mientras este persiste. El ejemplo de Martin Fowler: si una funcionalidad llevaría cuatro días en un código limpio pero lleva seis por culpa de la suciedad acumulada, los dos días adicionales son intereses.

De ello se derivan tres consecuencias. Primera, **la deuda que nunca se toca no cuesta nada**: un módulo desordenado que nadie cambia no acumula intereses, así que saldarla puede ser un esfuerzo desperdiciado. Segunda, **los intereses se capitalizan**: los parches se convierten en la base de más parches, los nuevos desarrolladores copian los patrones que ven y el costo de la reparación final aumenta. Tercera, **saldar la deuda es una decisión de inversión**: vale la pena pagar el principal cuando los intereses que te ahorrarás en un horizonte razonable lo superan. Eso convierte un argumento emocional ("el código es un desastre") en uno económico ("este módulo nos cuesta once días por trimestre").

SI ESTO ES NUEVO PARA TI

Paso a paso: cuándo el desorden cuesta dinero y cuándo no

La idea más difícil de esta sección es que la deuda técnica se juzga por los intereses que cobra, no por lo desordenado que parezca el código. El principal se paga una vez, para corregir el problema; el interés se paga en cada cambio que tiene que sortearlo. Dos módulos igual de desordenados muestran por qué eso importa.

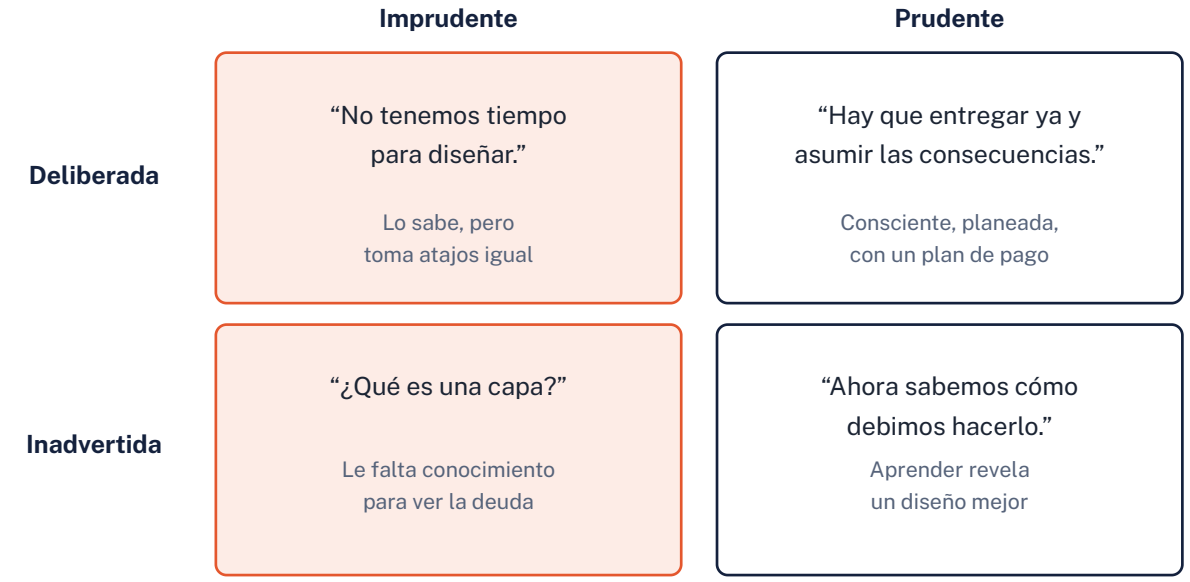
- Toma las cifras de Martin Fowler: un cambio que llevaría cuatro días en un código limpio lleva seis en un módulo desordenado, porque los desarrolladores primero tienen que entender el desorden y sortearlo. Los dos días adicionales son intereses.
- Supón que ordenar el módulo, el principal, llevaría diez días-desarrollador y devolvería cada cambio a unos cuatro días. Que valga la pena o no depende de la frecuencia con que cambie el módulo.
- Si el equipo lo cambia aproximadamente una vez al mes, paga unos dos días de intereses al mes, así que los diez días de limpieza se amortizan en unos cinco meses. Si se deja como está, los intereses tienden a crecer, porque se construyen parches nuevos sobre los antiguos.
- Un segundo módulo, igual de desordenado, solo se cambia unas dos veces al año. Cobra unos cuatro días de intereses al año, así que los mismos diez días de limpieza tardarían unos dos años y medio en amortizarse.
- Un módulo que nadie cambia, y que no causa fallos, no cuesta casi nada en intereses, por desordenado que parezca. Ordenarlo puede ser un esfuerzo desperdiciado.
- **La lección:** no preguntes '¿qué tan desordenado está este código?', sino '¿cuánto cuesta de más cada cambio y con qué frecuencia lo cambiamos?'. Salda primero donde los intereses son más altos. El ejemplo práctico del final de esta sección hace el mismo cálculo a partir de las incidencias y del historial del control de versiones.

El cuadrante de la deuda técnica de Fowler

No toda la deuda es igual. En 2009, Martin Fowler propuso clasificar la deuda según dos ejes: si se contrajo de forma **deliberada** o **inadvertida**, y si la decisión fue **prudente** o **imprudente**. Las cuatro combinaciones requieren respuestas distintas.

Prudente y deliberada: el equipo conoce el mejor diseño, juzga que entregar ahora compensa el costo y planea saldarla. Es el sentido original de Cunningham, y puede ser un buen negocio: una startup que demuestra que hay demanda antes de invertir en escala. **Imprudente y deliberada**: el equipo sabe hacerlo mejor, pero se salta el diseño porque cree que no tiene tiempo, y normalmente se frena a sí mismo en cuestión de semanas. **Imprudente e inadvertida**: el equipo no sabe lo suficiente para ver que está creando deuda; el remedio son las habilidades, la revisión y la mentoría, no solo la refactorización. **Prudente e inadvertida**: un equipo competente hace un buen trabajo y solo más tarde entiende cómo debería haberlo estructurado. Fowler sostiene que esto es inevitable en cualquier equipo que aprende, y por eso la refactorización continua forma parte del trabajo normal y no es una señal de fracaso.

El cuadrante de la deuda técnica



Según Martin Fowler (2009): los dos cuadrantes imprudentes son problemas de disciplina o de habilidad; los dos prudentes son consecuencias normales de entregar en condiciones de incertidumbre.

Deuda arquitectónica frente a deuda de código

La deuda existe en todos los niveles, pero su perfil de costo cambia con la escala. La **deuda de código** vive dentro de un módulo: lógica duplicada, una función que hace cinco cosas, pruebas que faltan, nombres poco claros. Es real, pero es local; un solo desarrollador suele poder saldarla en horas o días, a menudo mientras trabaja en una funcionalidad cercana. La **deuda**

arquitectónica vive en la estructura: servicios que comparten una base de datos, de modo que ninguno puede cambiar su esquema por sí solo; un dominio dividido por las líneas equivocadas, de modo que cada funcionalidad afecta a cuatro equipos; una cadena de llamadas síncronas que hace que un servicio lento tumbe el proceso de pago; un framework soldado a cada capa. Saldarla afecta a muchos módulos, a menudo a muchos equipos y, a veces, a los datos.

La investigación respalda la intuición de que la deuda estructural es la costosa. Una encuesta de 2015 a 1831 ingenieros y arquitectos, realizada por Neil Ernst y sus colegas del Software Engineering Institute, concluyó que las decisiones arquitectónicas eran la fuente de deuda técnica más importante de las señaladas, por delante de los problemas a nivel de código. Las herramientas que detectan code smells son buenas para encontrar deuda de código y malas para encontrar deuda arquitectónica, porque el problema está en las relaciones entre las partes y no en un archivo concreto.

La deuda arquitectónica importa por la misma razón por la que importa la arquitectura: como mostró Por qué importa la arquitectura, las decisiones estructurales son las que tienen el mayor costo de cambio.

Erosión y deriva

La deuda de arquitectura a menudo no se acumula por una mala decisión, sino por miles de pequeños cambios. Dewayne Perry y Alexander Wolf, en su artículo de 1992 "Foundations for the Study of Software Architecture", nombraron dos formas en que esto ocurre. La **erosión de la arquitectura** la causan las *violaciones* de la arquitectura: un controlador que consulta la base de datos directamente, saltándose la capa de servicios; un módulo que se mete en las tablas internas de otro. Cada violación hace el sistema más frágil, como quitar un muro de carga. La **deriva de la arquitectura** la causa la *falta de atención* a la arquitectura: cambios que no rompen ninguna regla explícita, pero ignoran los conceptos previstos, de modo que el sistema pierde poco a poco coherencia y claridad de forma. La deriva es menos peligrosa de inmediato que la erosión, pero oscurece la arquitectura, lo que a su vez facilita las violaciones.

Investigadores posteriores suelen usar "erosión" para la brecha más amplia entre la arquitectura prevista (lo que dicen los documentos y la cabeza del arquitecto) y la arquitectura implementada (lo que hace realmente el código). Sean cuales sean los términos, las causas son conocidas:

La presión de los plazos, que premia el camino más corto por encima del camino correcto. **La pérdida de conocimiento**, cuando se van las personas que entendían el diseño y el personal nuevo copia lo que encuentra. **La arquitectura invisible**: reglas que solo viven en una wiki que nadie lee, sin nada en la compilación que las haga cumplir. **Los requisitos cambiados**, que hacen que una estructura antes buena encaje mal con el problema. **El cambio organizativo**: por

la ley de Conway, una reorganización crea presión para que el sistema se remodele según las nuevas líneas de los equipos. **La tecnología que envejece**: los frameworks y las plataformas que se quedan sin soporte se convierten en deuda sin que nadie toque el código.

Detectar la deuda

La deuda que no se ve no se puede gestionar, y la intuición no es fiable: los desarrolladores tienden a quejarse más del código que más tocan, lo cual es útil, pero rara vez saben cuánto cuesta cada problema. Varias técnicas hacen visible la deuda.

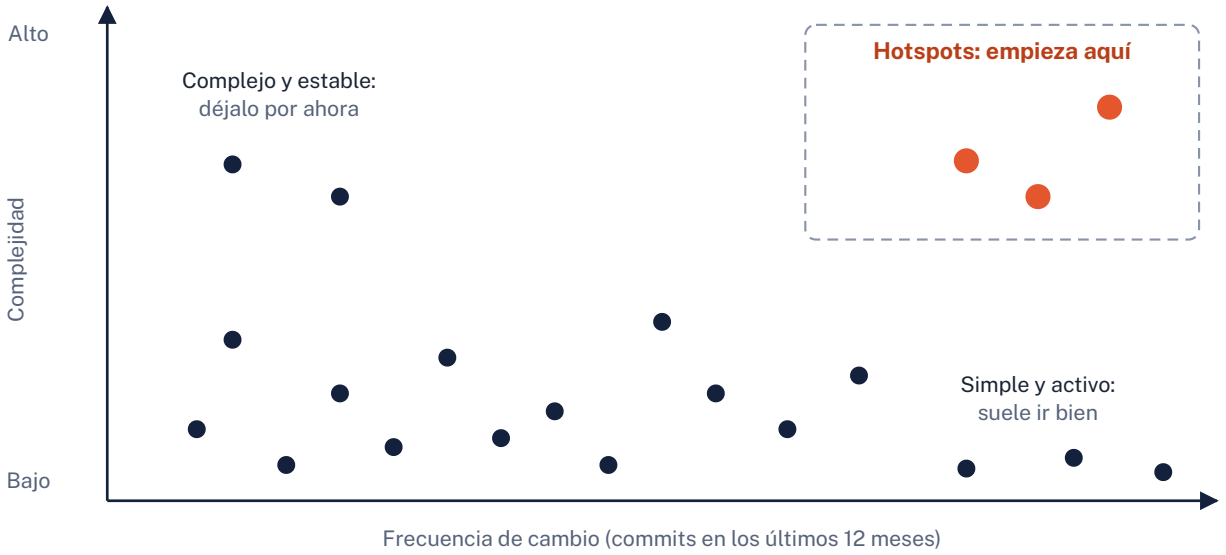
Hotspots (puntos calientes). *Your Code as a Crime Scene* (2015; segunda edición, 2024), de Adam Tornhill, aplica ideas forenses al historial del control de versiones. Un hotspot es código que es a la vez complejo y que cambia a menudo. La complejidad por sí sola no basta: un archivo complejo que nadie toca no cuesta intereses. La frecuencia de cambio por sí sola tampoco basta: un archivo de configuración sencillo puede cambiar a diario sin causar daño. Donde ambas coinciden, se pagan intereses en cada commit, y ahí es donde saldar la deuda rinde antes. En la mayoría de las bases de código, una pequeña fracción de los archivos concentra la mayor parte de la actividad de cambio.

Acoplamiento de cambio. Los archivos o servicios que cambian una y otra vez en los mismos commits están acoplados, tanto si el código muestra una dependencia como si no. Si cada cambio en el servicio de precios va acompañado de un cambio en el servicio de facturación, son una sola cosa que finge ser dos. El acoplamiento de cambio entre los límites de equipos o de servicios es una señal clara de deuda arquitectónica.

Ciclos de dependencias y violaciones de reglas. El análisis estático puede dibujar el grafo de dependencias real y compararlo con el previsto. Los ciclos (A usa B, B usa C, C usa A) significan que ninguno de los tres puede entenderse, probarse ni lanzarse por separado. Las reglas de arquitectura ("la capa de dominio no debe depender de la capa web") pueden comprobarse automáticamente en cada compilación; son funciones de aptitud (fitness functions), que se tratan en el Capítulo 11.

Tendencias de la entrega. El aumento del tiempo de entrega de los cambios, más defectos que se escapan en cada versión, una incorporación de personal más larga, más reuniones de coordinación por funcionalidad y una creciente reticencia a tocar ciertas áreas son indicadores rezagados de los intereses acumulados. Las métricas de entrega DORA (como el tiempo de entrega de cambios y la tasa de fallos de los cambios) ofrecen una forma coherente de seguir algunos de ellos a lo largo del tiempo.

Encontrar hotspots



Situar cada archivo según la frecuencia con que cambia y su complejidad muestra dónde se pagan intereses con más frecuencia: los pocos archivos de la esquina superior derecha.

Gestionar la deuda

Ninguna base de código sana tiene cero deuda, e intentar llegar a cero es en sí un desperdicio. El objetivo es que la deuda sea visible, deliberada y asumible. Los equipos que lo hacen bien combinan varias prácticas.

Un registro de deuda. Anota los elementos de deuda importantes donde el equipo planifica su trabajo: cuál es el problema, dónde está, cómo surgió, el principal estimado, los intereses estimados (con pruebas como los datos de hotspots o las incidencias recientes) y el desencadenante para saldarla. Un registro convierte las quejas vagas en una lista que puede priorizarse junto con las funcionalidades.

Una parte fija de la capacidad. Muchos equipos reservan una proporción estable de cada iteración, normalmente entre el 10 y el 20 por ciento, para saldar deuda y otros trabajos de mejora. No hay pruebas de que exista un número mágico; lo que importa es que esa parte esté protegida y no desaparezca cada vez que se acerca una fecha límite.

Refactorizar junto con las funcionalidades. El momento más barato para saldar deuda es cuando ya estás trabajando en esa área, porque tienes el contexto en la cabeza y las pruebas en marcha. Fowler lo llama refactorización oportunista. Su forma cotidiana es la regla del boy scout, popularizada por Robert C. Martin: deja el código un poco más limpio de lo que lo encontraste. Como los hotspots son, por definición, donde aterrizan las funcionalidades, esto concentra de forma natural el esfuerzo donde los intereses son más altos.

Pago estructural planificado. La deuda arquitectónica rara vez cede solo con pasos pequeños. Necesita un trabajo explícito y financiado, con un objetivo, un plan para migrar por incrementos y el apoyo de las partes interesadas. Técnicas como el patrón strangler fig (Capítulo 11, Modernizar sistemas heredados) permiten a un equipo sustituir una estructura pieza a pieza sin una reescritura big-bang.

Prevención. Las comprobaciones automáticas de la arquitectura en la compilación, una revisión de código que mira la estructura además del estilo y unos registros de decisiones actualizados frenan la erosión y la deriva en su origen.

Explicar la deuda a las partes interesadas no técnicas

A los patrocinadores y a los responsables de producto no les hace falta oír hablar de dependencias cíclicas; necesitan entender el efecto sobre lo que les importa: velocidad, costo, riesgo y experiencia del cliente. Aquí la metáfora financiera es un regalo, siempre que la uses con cifras.

Habla en términos de **tiempo y dinero**: "Cada cambio en el proceso de pago nos cuesta unos tres días-desarrollador adicionales por cómo están enredados los pagos y los pedidos". Habla en términos de **riesgo**: "Dos de nuestros últimos cuatro incidentes empezaron en este módulo". Habla en términos de **opciones**: "Hasta que separemos esto, no podemos ofrecer los nuevos métodos de pago que ha pedido el equipo de ventas". Y presenta el pago de la deuda como una **inversión con un periodo de recuperación**, no como una súplica para conseguir tiempo de ingeniería. Evita dos trampas: presentar la deuda como un fallo moral de quienes estuvieron antes (normalmente es prudente o inadvertida) y pedir un "sprint de refactorización" sin límite y sin un resultado medible.

EJEMPLO PRÁCTICO

Ejemplo práctico: estimar los intereses de una deuda

La lógica de pedidos y de pagos de una tienda en línea comparte un gran módulo y las mismas tablas de la base de datos. El equipo sospecha que esto le cuesta mucho. Reúne pruebas de las incidencias y del historial del control de versiones de los últimos seis meses, y luego hace una estimación.

Cambios que tocan el módulo	48 cambios en seis meses, unos 8 al mes (es el principal hotspot).
Esfuerzo adicional por cambio	Comparando con cambios similares en otras partes, los desarrolladores estiman unos 1.5 días adicionales por cambio para entender el código, sortear las tablas compartidas y hacer pruebas de regresión adicionales.
Intereses en esfuerzo	8 cambios × 1.5 días = 12 días-desarrollador al mes.
Intereses por defectos	3 incidentes en producción en seis meses atribuidos a este módulo, cada uno con un costo de unos 4 días-desarrollador para investigarlo y corregirlo: 12 días ÷ 6 = 2 días al mes.
Intereses totales	Unos 14 días-desarrollador al mes. Con un costo total de £600 por día-desarrollador, aproximadamente £8400 al mes, o unos £100 000 al año.
Principal	Separar los pedidos de los pagos detrás de una interfaz y dividir las tablas, de forma incremental: 45 días-desarrollador estimados, unas £27 000.
Intereses residuales tras saldar la deuda	Los desarrolladores esperan que quede cierta sobrecarga: 3 días-desarrollador al mes estimados.
Recuperación	Ahorro mensual de unos 11 días-desarrollador. 45 ÷ 11 son unos 4 meses para recuperar el principal.
Recomendación	Saldarla, en incrementos de unas dos semanas cada uno, empezando por las tablas que intervienen con más frecuencia en los incidentes. Seguir el tiempo de entrega de los cambios en el proceso de pago para confirmar el ahorro.

Cada cifra es una estimación con amplios márgenes de error. Duplicar el principal hasta 90 días sigue dando una recuperación en unos ocho meses, así que la conclusión es sólida; vale la pena mostrar esa comprobación a las partes interesadas.

El mismo método puede mostrar que no vale la pena saldar cierta deuda: un módulo desordenado que se cambia dos veces al año puede tener un periodo de recuperación de varios años.

El argumento se presenta en los términos del patrocinador (días, dinero, incidentes, recuperación), no en términos de calidad del código.

ATENCIÓN

Ideas erróneas habituales

- "Deuda técnica significa código malo". Mucha deuda viene de buen código escrito con un entendimiento que desde entonces ha mejorado, o de requisitos que han cambiado.
- "Hay que saldar toda la deuda". La deuda en código que nunca cambia no cuesta nada; salda donde están los intereses.
- "Lo arreglaremos en una gran reescritura". Las reescrituras big-bang suelen pasarse de plazo y perder comportamientos ocultos; saldar la deuda por incrementos suele ser más seguro.
- "Solo a los desarrolladores les importa la deuda". Sus efectos (funcionalidades más lentas, más incidentes, oportunidades bloqueadas) son efectos sobre el negocio, y los patrocinadores deberían ayudar a decidir qué saldar.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué un módulo muy desordenado puede tener poca prioridad a la hora de saldar su deuda?

Porque los intereses solo se pagan cuando el código se cambia o causa fallos. Si nadie lo toca y es estable, la deuda no cuesta casi nada.

▼ ¿Por qué la deuda imprudente e inadvertida no se cura solo con refactorización?

Porque el equipo no sabía que estaba creando deuda; a menos que mejoren sus habilidades y sus prácticas de revisión, seguirá creando el mismo tipo de deuda tan rápido como la salda.

▼ ¿Cuál es la diferencia entre la erosión y la deriva en los términos de Perry y Wolf?

La erosión es el resultado de violar las reglas de la arquitectura; la deriva es el resultado de cambios hechos sin tener en cuenta la arquitectura, que no rompen ninguna regla explícita pero desdibujan sus conceptos.

▼ **¿Por qué los análisis de hotspots combinan la frecuencia de cambio con la complejidad en lugar de usar cualquiera de las dos por separado?**

El código complejo que nunca cambia acumula pocos intereses, y el código sencillo que cambia a menudo es barato de cambiar; solo la combinación muestra dónde la deuda está costando esfuerzo una y otra vez.

FUENTES

- The WyCash Portfolio Management System, Ward Cunningham (OOPSLA 1992): https://c2.com/doc/oopsla92.html
- Technical Debt, Martin Fowler: https://martinfowler.com/bliki/TechnicalDebt.html
- Technical Debt Quadrant, Martin Fowler (2009): https://martinfowler.com/bliki/TechnicalDebtQuadrant.html
- Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt (SEI, 2015): https://sei.cmu.edu/library/measure-it-manage-it-ignore-it-software-practitioners-and-technical-debt
- Ward Explains Debt Metaphor (c2 wiki): https://wiki.c2.com/?WardExplainsDebtMetaphor

SECCIÓN 1.8 DE 1.8 · INTERMEDIO · 19 MIN DE LECTURA

La arquitectura a lo largo del ciclo de vida: arquitectura inicial, emergente y continua

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.8 Cómo se hace el software
- 1.4 El papel del arquitecto
- 1.6 Decisiones y compromisos

Términos anteriores que utiliza: Ágil, Entrega continua, Último momento responsable, Función de aptitud (fitness function), Atributo de calidad

LA IDEA EN BREVE

Cuando planeas un viaje largo, reservas los vuelos y el hotel de la primera noche con mucha antelación, porque son caros de cambiar y podrían agotarse. No planeas cada comida de cada día antes de salir; eso lo decides sobre la marcha, cuando sabes cómo te sientes y qué hay cerca. La buena arquitectura de software funciona igual. Las pocas decisiones grandes y difíciles de cambiar se toman pronto y con cuidado. Todo lo demás se decide por el camino, a medida que el equipo aprende, y el plan se revisa y se ajusta durante todo el viaje en lugar de escribirse una vez y archivarse.

LO QUE NECESITAS SABER

- El gran diseño inicial (big design up front) conviene cuando los requisitos son estables y el cambio es muy costoso; el diseño puramente emergente conviene a sistemas pequeños de puertas de doble sentido; la mayoría de los equipos necesitan algo intermedio.
- El principio del Manifiesto Ágil según el cual las mejores arquitecturas surgen de equipos autoorganizados trata de quién diseña y cómo, y no afirma que la arquitectura sea innecesaria.
- El modelo dirigido por riesgos de Fairbanks ajusta el esfuerzo de arquitectura en proporción al riesgo de fallo: identifica y prioriza los riesgos, aplica técnicas, evalúa la reducción y detente cuando el riesgo restante sea aceptable.
- La pista arquitectónica (architectural runway) de SAFe y la arquitectura continua de Erder y Pureur tratan la arquitectura como un trabajo continuo que debe ir por delante de las funcionalidades que dependen de ella.

- Un esqueleto andante (walking skeleton) demuestra pronto la arquitectura de extremo a extremo; los spikes convierten las decisiones inciertas en decisiones basadas en pruebas.
- En la entrega continua y DevOps, la desplegabilidad y la operabilidad se convierten en atributos de calidad clave, una arquitectura débilmente acoplada hace posible un buen rendimiento de la entrega, y el sistema en ejecución se convierte en la principal fuente de retroalimentación sobre la arquitectura.

El viejo debate: ¿inicial o emergente?

Durante la mayor parte de la historia del software, la arquitectura era algo que se hacía al principio. En un proceso secuencial ("en cascada"), primero se recopilaban los requisitos, después se elaboraban y aprobaban una arquitectura y un diseño detallado, y luego se construía el sistema según ese diseño. Esto es el gran diseño inicial (big design up front) (BDUF). Su lógica es sólida en algunos contextos: cuando los requisitos son estables y se entienden bien, cuando los cambios después de la entrega son extremadamente costosos (software de vuelo, dispositivos médicos, sistemas ligados al hardware) o cuando muchos contratistas deben construir partes que encajen entre sí, un diseño minucioso antes de la construcción reduce el riesgo.

Sus debilidades son igual de conocidas. Los requisitos cambian, y un diseño elaborado hecho pronto incorpora supuestos que más tarde resultan erróneos. La retroalimentación del uso real solo llega al final, cuando el cambio es más costoso. Los equipos diseñan para necesidades imaginadas ("puede que necesitemos admitir cinco monedas") y pagan por una flexibilidad que nadie usa. Y los documentos que tardaron meses en aprobarse tienden a seguirse cuando ya han dejado de ser ciertos.

Los métodos ágiles reaccionaron contra esto. Extreme Programming, en particular, defendió el diseño emergente: construir lo más sencillo que funcione para las funcionalidades del momento, mantener el código limpio con refactorización continua y pruebas automatizadas, y dejar que el diseño crezca a la vez que el sistema. Las reglas del diseño simple de Kent Beck y el principio "you aren't gonna need it" (YAGNI, no lo vas a necesitar) expresan la misma idea. El diseño emergente funciona bien dentro de un módulo y en sistemas pequeños donde cada decisión es una puerta de doble sentido. Su debilidad aparece a escala: algunas decisiones (el modelo de datos de un sistema de registro, los límites entre equipos, el modelo de seguridad, la elección entre integración síncrona y asíncrona) son costosas de revertir y dan forma a todo lo que se construye encima. Dejar que surjan solas suele significar que las toma por accidente quien escriba el primer código.

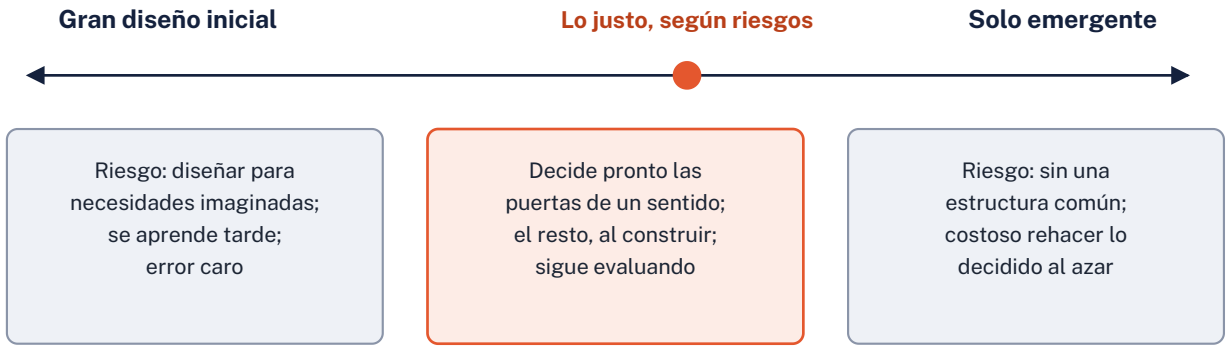
Lo que dice y lo que no dice el Manifiesto Ágil

Uno de los doce principios del Manifiesto Ágil (2001) dice: “Las mejores arquitecturas, requisitos y diseños emergen de equipos autoorganizados.” Se cita a menudo como si significara que los equipos ágiles no necesitan arquitectura ni arquitectos. No dice eso.

Leído con atención, el principio afirma algo sobre *quién* produce las mejores arquitecturas (las personas que construyen el sistema, organizadas por sí mismas en lugar de dirigidas desde fuera) y sobre *cómo* (surgen a través del trabajo, en lugar de entregarse terminadas desde arriba). No dice nada de que la arquitectura no sea importante, y el mismo documento contiene otro principio: la atención continua a la excelencia técnica y al buen diseño mejora la agilidad. Un equipo que ignora las decisiones estructurales no está siguiendo el manifiesto; está ignorando la mitad.

Lo que sí descarta el principio es una arquitectura impuesta desde una torre de marfil por personas que no vivirán con sus consecuencias, y una fase en la que la arquitectura se termina y se congela antes de que nadie escriba código. En la práctica, la conciliación más habitual es que la arquitectura se convierte en una actividad continua, compartida dentro del equipo, con un arquitecto (o ingenieros con mentalidad arquitectónica) que actúa como guía, facilitador y custodio de la visión de conjunto, en lugar de como guardián que da o niega el paso.

Un espectro, no una elección



La mayoría de los equipos que tienen éxito se sitúan entre los extremos: deciden pronto las cuestiones costosas de revertir y dejan que el resto surja.

¿Cuánta arquitectura? El modelo dirigido por riesgos

Si ninguno de los extremos es correcto, ¿cuánto trabajo de arquitectura es suficiente? George Fairbanks dio una de las respuestas más claras en *Just Enough Software Architecture: A Risk-Driven Approach* (2010). Su modelo dirigido por riesgos dice que el esfuerzo que dedicas a la arquitectura debe ser proporcional al riesgo de fallo. Tiene tres pasos, que se repiten según sea necesario: **identificar y priorizar los riesgos** (idealmente como escenarios de fallo

concretos, por ejemplo "la página de reservas agota el tiempo de espera cuando 5000 pacientes inician sesión a las 8 a. m."); **seleccionar y aplicar técnicas** que reduzcan esos riesgos (un modelo, un prototipo, una prueba de carga, un análisis, un patrón de diseño); y **evaluar la reducción del riesgo**, deteniéndose cuando el riesgo restante sea aceptable.

El modelo es eficaz porque ofrece una regla para detenerse. Una herramienta interna sencilla, construida con tecnología conocida y con pocos usuarios, se enfrenta a pocos riesgos arquitectónicos, y puede bastar con un boceto en una pizarra. Una plataforma de pagos que gestiona millones de transacciones, sujeta a regulación, se enfrenta a muchos, y merece un modelado, unos prototipos y una revisión cuidadosos. Un mismo equipo puede aplicar cantidades muy distintas de arquitectura a distintas partes de un mismo sistema. Fairbanks también subraya que el tiempo dedicado a diseñar es tiempo que no se dedica a construir ni a probar, así que diseñar de más es un costo real, no una opción segura por defecto.

SI ESTO ES NUEVO PARA TI

Paso a paso: la arquitectura justa para el sistema de reservas de citas

La idea más difícil de esta sección es el modelo dirigido por riesgos de Fairbanks: no existe una cantidad correcta y fija de arquitectura, solo una cantidad que se ajusta a los riesgos. Sigues diseñando mientras el riesgo de fallo es demasiado alto y te detienes cuando es aceptable. Síguelo a través del sistema de reservas de citas del hospital.

- Enumera los riesgos como fallos concretos, por ejemplo 'la página de reservas agota el tiempo de espera cuando 5000 pacientes inician sesión a las 8 a. m.' y 'el sistema de administración de pacientes, que proporciona las franjas horarias libres de las consultas, no puede responder a 50 solicitudes por segundo'. Las partes construidas con tecnología conocida para pocos usuarios plantean pocos riesgos serios.
- Prioriza. La avalancha de las 8 a. m. afectaría a todos los pacientes que reservaran a esa hora, así que va primero. Las partes de bajo riesgo reciben un boceto en la pizarra y nada más.
- Aplica una técnica adecuada a cada riesgo. Para la avalancha de la mañana, el equipo construye un pequeño prototipo y lo prueba con 5000 pacientes simulados que inician sesión a la vez. Para el sistema de administración de pacientes, un breve experimento, acordado con su proveedor, mide cuántas solicitudes por segundo puede responder.
- Evalúa cuánto se ha reducido cada riesgo. Supón que el prototipo aguanta la avalancha, pero el sistema de administración de pacientes se ralentiza mucho antes de llegar a 50 solicitudes por segundo. El primer riesgo ya es aceptable; el segundo no, así que el equipo da otra vuelta, quizá probando la caché de franjas horarias de corta duración de la Sección 1.5.
- Detente cuando todos los riesgos principales sean aceptablemente bajos. Seguir diseñando no es una opción segura por defecto, advierte Fairbanks, porque quita tiempo a la construcción y a las pruebas.
- **La lección:** deja que los riesgos fijen el esfuerzo. Una parte de un sistema puede recibir un boceto en la pizarra y otra, semanas de prototipos y pruebas: eso es el modelo funcionando.

La arquitectura intencional y la pista arquitectónica

Las grandes organizaciones se enfrentan a un problema de coordinación que un solo equipo no tiene: decenas de equipos deben construir partes que encajen entre sí. El Scaled Agile Framework (SAFe), un marco para escalar las prácticas ágiles ampliamente adoptado y también ampliamente debatido, describe la arquitectura como un equilibrio entre el *diseño emergente* dentro de los equipos y la *arquitectura intencional*: la planificación deliberada, entre equipos, de las estructuras y las pautas que los equipos comparten. Al resultado lo llama pista

arquitectónica (architectural runway): el código, los componentes y la infraestructura técnica existentes necesarios para implementar las funcionalidades a corto plazo sin rediseños ni retrasos excesivos. El trabajo para ampliar la pista (por ejemplo, introducir una plataforma de mensajería antes de las funcionalidades que la necesitan) se planifica de forma explícita junto con el trabajo de funcionalidades, y las funcionalidades consumen la pista a medida que la usan.

Descrita de forma neutral, la metáfora de la pista recoge algo que es cierto para cualquier organización: si la infraestructura que necesita una funcionalidad aún no existe, esa funcionalidad será lenta, y construir la infraestructura requiere tiempo de antelación. Los críticos de SAFe sostienen que puede reintroducir una planificación pesada y jerárquica con vocabulario ágil; sus defensores sostienen que cierta coordinación central es inevitable a escala. No necesitas adoptar SAFe para usar la idea: "¿qué cimientos necesitarán las funcionalidades del próximo trimestre y los estamos construyendo a tiempo?" es una buena pregunta para cualquier arquitecto.

Arquitectura continua

Murat Erder y Pierre Pureur, en *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World* (2015) y *Continuous Architecture in Practice* (2021, con Eoin Woods), describen la arquitectura como un flujo continuo de decisiones y no como una fase o un documento. Sus seis principios son: diseñar la arquitectura de productos, no solo soluciones para proyectos; centrarse en los atributos de calidad, no en los requisitos funcionales; retrasar las decisiones de diseño hasta que sean absolutamente necesarias; diseñar la arquitectura para el cambio aprovechando la fuerza de las partes pequeñas y débilmente acopladas; diseñar la arquitectura para construir, probar, desplegar y operar; y organizar los equipos a imagen del diseño del sistema en el que se trabaja.

Dos de esos principios recuerdan secciones anteriores de este capítulo: retrasar las decisiones es el último momento responsable, y organizar los equipos a imagen del sistema es un uso deliberado de la ley de Conway. El énfasis práctico está en que la unidad del trabajo arquitectónico sea la *decisión*, registrada en el momento en que se toma, y en los ciclos de retroalimentación (pruebas automatizadas, funciones de aptitud, monitorización en producción) que le dicen al equipo si la arquitectura sigue cumpliendo sus objetivos de calidad.

Esqueletos andantes y spikes

Dos técnicas sencillas unen la arquitectura y la entrega desde la primera semana. Un esqueleto andante (walking skeleton), término asociado a Alistair Cockburn, es una implementación diminuta del sistema que realiza una pequeña función de extremo a extremo a través de todos los componentes arquitectónicos principales. En un sistema de reservas de citas de un hospital,

podría permitir que un usuario de prueba inicie sesión, vea las franjas horarias libres de una consulta fijada en el código, obtenidas del sistema de administración de pacientes, reserve una y reciba un mensaje de texto, todo desplegado en un entorno similar al de producción a través del pipeline de entrega real. No hace casi nada útil, pero demuestra que las piezas se conectan, que el pipeline funciona y que la arquitectura se puede construir. Cada funcionalidad posterior añade carne a un esqueleto que ya camina.

Un spike (investigación acotada) (otro término de Extreme Programming) es un experimento breve, con un tiempo limitado, para responder a una pregunta o reducir un riesgo: ¿puede la interfaz del PAS gestionar 50 solicitudes por segundo? ¿Cuánto tarda el proveedor de identidad en emitir un token? ¿Puede la biblioteca que hemos elegido generar archivos PDF accesibles? Los spikes suelen desecharse; su resultado es conocimiento, registrado en una decisión. Son la forma más barata de llevar una decisión de las conjeturas a las pruebas antes de su último momento responsable.

Las actividades de arquitectura en un ciclo iterativo

En un proceso iterativo, el trabajo de arquitectura no ocurre una sola vez; se repite en cada ciclo, con una profundidad que se ajusta a los riesgos que tiene delante el equipo. El plan de estudios de iSAQB describe las tareas centrales del arquitecto de una forma que encaja de manera natural en un ciclo así, y El papel del arquitecto las presentó. Aquí están en forma de bucle:

Aclarar los requisitos: encontrar a las partes interesadas y sus inquietudes, precisar los objetivos de calidad y las restricciones, e identificar los riesgos (consulta Partes interesadas e inquietudes). **Diseñar las estructuras**: tomar o revisar las decisiones significativas, eligiendo estilos y patrones y definiendo límites e interfaces. **Comunicar y acordar**: explicar las decisiones a las personas que construyen, operan y pagan el sistema, con vistas adecuadas a cada público, y registrarlas. **Acompañar la implementación**: trabajar con los equipos mientras construyen, responder preguntas, revisar código y ajustar el diseño cuando la realidad se resiste. **Evaluar**: comprobar que el sistema cumple sus objetivos de calidad, mediante revisiones, pruebas, mediciones en producción y funciones de aptitud automatizadas. **Evolucionar**: actuar según lo que encontró la evaluación, saldar deuda y adaptarse a los requisitos cambiados, lo que te lleva de nuevo a aclarar los requisitos.

El ciclo de la arquitectura

Cada iteración vuelve a pasar por las seis actividades, con una profundidad proporcional a los riesgos que tiene delante el equipo, en torno a un sistema en ejecución que proporciona retroalimentación.

La arquitectura en DevOps y en la entrega continua

La entrega continua, en la que cada cambio se compila, se prueba y se mantiene listo para lanzarse, y DevOps, que une la responsabilidad del desarrollo y de las operaciones, cambian tanto los objetivos de la arquitectura como sus ciclos de retroalimentación.

Cambian los objetivos porque **la desplegabilidad, la capacidad de prueba y la operabilidad se convierten en atributos de calidad de primer orden**. Un sistema no puede lanzarse muchas veces al día si los componentes deben desplegarse juntos en un orden fijo, si las pruebas necesitan una copia completa de producción o si nadie puede saber, por la monitorización, si un lanzamiento empeoró las cosas. La investigación resumida en *Accelerate* (Nicole Forsgren, Jez Humble y Gene Kim, 2018) concluyó que una arquitectura débilmente acoplada, que permita a los equipos probar y desplegar sus partes de forma independiente, estaba entre los indicadores más fuertes del rendimiento de la entrega continua. Por tanto, la buena práctica de entrega no solo ayuda a la arquitectura; la arquitectura es una de sus condiciones previas.

Cambian los ciclos de retroalimentación porque el sistema en ejecución se convierte en la revisión de arquitectura más honesta. Las métricas de producción, las trazas y las revisiones de incidentes muestran dónde es débil la arquitectura con mucha más fiabilidad que los diagramas. Las reglas de arquitectura pueden hacerse cumplir como funciones de aptitud automatizadas en el pipeline (Capítulo 11), la infraestructura se define como código y se revisa

como el código (Capítulo 10), y técnicas como las feature flags y los lanzamientos canario hacen que muchas decisiones sean baratas de revertir. El papel del arquitecto cambia en consecuencia: menos aprobación de documentos y más modelado de la plataforma, del pipeline y de las barreras de protección que permiten a los equipos tomar buenas decisiones con rapidez.

Lista de comprobación: ¿es proporcionado tu trabajo de arquitectura?

Repasa esta lista al inicio de un proyecto y en cada hito importante. Cada elemento es una pregunta sobre si el esfuerzo se ajusta al riesgo.

- [ ] **Puedes nombrar los cinco principales riesgos arquitectónicos**
  Redactados como escenarios de fallo concretos, priorizados, y cada uno con un responsable y un plan para reducirlo.
- [ ] **Tu esfuerzo de diseño se concentra en esos riesgos**
  Las áreas de bajo riesgo y tecnología conocida reciben un boceto; las áreas de alto riesgo reciben prototipos, mediciones y revisión.
- [ ] **Las puertas de un solo sentido se identifican y se deciden de forma deliberada**
  Las decisiones costosas de revertir se conocen, se toman en el último momento responsable y se registran; las puertas de doble sentido se dejan a los equipos.
- [ ] **Existe pronto un esqueleto andante**
  Un camino de extremo a extremo a través de los componentes principales se ejecuta en un entorno similar al de producción en las primeras iteraciones.
- [ ] **Las decisiones se registran brevemente a medida que se toman**
  Existe un breve registro de decisión para cada decisión significativa; nadie tiene que reconstruir el razonamiento de memoria.
- [ ] **Los objetivos de calidad se comprueban de forma continua**
  Los atributos de calidad clave tienen pruebas automatizadas, funciones de aptitud o mediciones en producción, no solo una revisión puntual.
- [ ] **La arquitectura no bloquea la entrega**
  Los equipos rara vez esperan una decisión o una aprobación de arquitectura; si lo hacen, el proceso es demasiado pesado para el riesgo.
- [ ] **La entrega no socava la arquitectura**
  Se hace un seguimiento de los hotspots, el acoplamiento de cambio y las violaciones de reglas; la erosión se detecta en semanas, no en años.
- [ ] **La cantidad de documentación se ajusta a sus lectores**
  Cada documento o vista responde a una inquietud real de una parte interesada y se mantiene actualizado; los documentos que nadie lee se retiran.

ATENCIÓN

Dos modos de fallo opuestos

La astronáutica de la arquitectura: meses de diagramas, frameworks construidos para necesidades que nunca llegan y un diseño que queda obsoleto antes del primer lanzamiento. **La arquitectura accidental**: nadie piensa en la estructura hasta que el sistema es grande, lento de cambiar e imposible de razonar, y un rescate costoso se vuelve inevitable. El remedio para el primero es la retroalimentación del código en ejecución; el remedio para el segundo es la atención deliberada a las pocas decisiones que más importan. El modelo dirigido por riesgos es una forma de navegar entre ambos.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Cuándo es el gran diseño inicial una opción sensata?

Cuando los requisitos son estables y se entienden bien, cuando el cambio después de la entrega es extremadamente costoso o cuando muchas partes independientes deben construir piezas que encajen exactamente entre sí.

▼ ¿Qué regla para detenerse ofrece el modelo dirigido por riesgos para el trabajo de arquitectura?

Detenerse cuando los riesgos restantes sean aceptablemente bajos; el esfuerzo debe ser proporcional al riesgo de fallo.

▼ ¿Cuál es la diferencia entre un esqueleto andante y un spike?

Un esqueleto andante es una implementación mínima de extremo a extremo a través de la arquitectura real que se conserva y sobre la que se sigue construyendo. Un spike es un experimento breve para responder a una pregunta, que normalmente se desecha una vez que ha producido conocimiento.

▼ ¿Por qué la entrega continua hace que la arquitectura sea más importante y no menos?

Los lanzamientos frecuentes e independientes requieren una estructura débilmente acoplada, que se pueda probar y desplegar; sin ella, el pipeline de entrega no puede funcionar con rapidez.

FUENTES

- Principles behind the Agile Manifesto: https://agilemanifesto.org/principles.html

- A Risk-Driven Model for Agile Software Architecture, George Fairbanks (Methods and Tools): https://www.methodsandtools.com/archive/agilearchi.html
- Is Design Dead?, Martin Fowler: https://martinfowler.com/articles/designDead.html
- Architectural Runway (SAFe): https://framework.scaledagile.com/architectural-runway
- Continuous Architecture (Erder, Pureur and Woods): https://continuousarchitecture.com/

Lecturas recomendadas

- *Software Architecture in Practice, 4th edition*, Len Bass, Paul Clements and Rick Kazman (2021), Addison-Wesley. El libro de texto de referencia del SEI: estructuras, atributos de calidad y tácticas, y la definición que se usa en todo este curso.
  https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/
- *Fundamentals of Software Architecture, 2nd edition*, Mark Richards and Neal Ford (2025), O'Reilly Media. Una introducción práctica y amena al papel del arquitecto, al razonamiento en términos de compromisos y a los estilos arquitectónicos.
- *ISO/IEC/IEEE 42010:2022 Software, systems and enterprise: Architecture description*, ISO/IEC JTC 1/SC 7 and IEEE (2022), ISO. La norma internacional sobre descripciones de arquitectura, partes interesadas, inquietudes y vistas.
  https://www.iso.org/standard/74393.html
- *Documenting Software Architectures: Views and Beyond, 2nd edition*, Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord and Judith Stafford (2010), Addison-Wesley. El origen de las categorías de estructura de módulos, de componentes y conectores y de asignación.
  https://www.sei.cmu.edu/library/documenting-software-architectures-views-and-beyond-second-edition/
- *Who Needs an Architect?*, Martin Fowler (2003), IEEE Software. Un ensayo breve e influyente sobre qué es la arquitectura y qué tipo de arquitecto necesita un equipo.
  https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf
- *Just Enough Software Architecture: A Risk-Driven Approach*, George Fairbanks (2010), Marshall & Brainerd. Explica cuánto trabajo de arquitectura hay que hacer, en función de los riesgos a los que se enfrenta realmente un proyecto.
  https://www.georgefairbanks.com/book/
- *The Software Architect Elevator*, Gregor Hohpe (2020), O'Reilly Media. Sobre el papel del arquitecto en las grandes organizaciones, como enlace entre la estrategia de negocio y la entrega técnica.
- *The WyCash Portfolio Management System (OOPSLA '92 experience report)*, Ward Cunningham (1992), OOPSLA 1992. El breve informe en el que apareció por primera vez la metáfora de la deuda técnica.
  https://c2.com/doc/oopsla92.html

Glosario

Acoplamiento de cambio
Dos o más archivos o componentes que tienden a cambiar en los mismos commits, lo que revela una dependencia que puede no ser visible en el código.

Amplitud técnica
Saber que existen muchas tecnologías y enfoques, en qué son buenos y malos y cuándo usarlos, aun sin una experiencia práctica profunda en cada uno.

Arquitectónicamente significativo
Describe una decisión o un requisito que es costoso de cambiar más adelante, afecta a muchas partes de un sistema o determina un atributo de calidad clave.

Arquitectura de software
Las estructuras fundamentales de un sistema de software y las decisiones importantes que les dieron forma: sus partes principales, cómo se relacionan, sus propiedades importantes y los principios que guían cómo se construye y evoluciona.

Arquitectura de soluciones
La arquitectura de la solución a un problema de negocio concreto, que a menudo abarca varias aplicaciones, proveedores y equipos, como un nuevo proceso de reclamaciones en línea para una aseguradora.

Arquitectura empresarial
La arquitectura a nivel de toda una organización: cómo encajan sus capacidades de negocio, su información, sus aplicaciones y su tecnología, y cómo cambian con el tiempo.

Atributo de calidad
Propiedad medible de un sistema que indica lo bien que hace su trabajo, como el rendimiento, la disponibilidad, la seguridad o la modificabilidad. El Capítulo 2 los trata en profundidad.

Big design up front (BDUF, gran diseño inicial)
Un enfoque en el que la arquitectura y el diseño se completan en detalle antes de que empiece la implementación.

Ciclo de dependencias
Un bucle en el que los módulos dependen unos de otros directa o indirectamente, de modo que ninguno puede entenderse, probarse ni lanzarse por separado.

Código muerto
Código que ya no debería ejecutarse pero que sigue presente en el sistema desplegado, donde puede activarse por error.

Componente
Una unidad que existe cuando el sistema se ejecuta, como un servicio, un proceso o una base de datos, y que interactúa con otros componentes a través de conectores.

Compromiso
Elección en la que obtener un beneficio implica aceptar un costo o renunciar a otro beneficio, como una entrega más rápida a cambio de un costo de operación más alto.

Conector
El mecanismo de tiempo de ejecución mediante el que interactúan los componentes: una llamada HTTP, una cola de mensajes, una base de datos compartida, un flujo de eventos o una llamada a función.

Costo del cambio
El esfuerzo, el dinero, el tiempo y el riesgo necesarios para modificar una decisión una vez que se ha construido trabajo sobre ella.

Deriva de la arquitectura
Cambios realizados sin conocer la arquitectura prevista que, aunque no rompen reglas explícitas, alejan al sistema de ella y difuminan sus conceptos.

Descripción de arquitectura
Un producto de trabajo (documentos, diagramas, modelos, registros de decisiones) que expresa una arquitectura. La arquitectura existe tanto si se describe como si no; la descripción es la forma en que las personas la comparten.

Deuda arquitectónica
Deuda técnica en la estructura de un sistema, como límites equivocados, una base de datos compartida o dependencias cíclicas, que es costosa de saldar porque afecta a muchas partes.

Deuda técnica
El costo futuro adicional de los cambios causado por atajos o por un diseño que ya no encaja, que se paga con un trabajo más lento y arriesgado hasta que se salda mejorando el diseño.

Diseño detallado
Decisiones sobre el interior de una parte de un sistema (sus clases, funciones, algoritmos y estructuras de datos locales) que pueden cambiarse sin afectar al resto.

Diseño emergente
Un enfoque en el que el diseño crece de forma incremental, moldeado por las necesidades de cada nueva funcionalidad y por la refactorización continua.

Elemento
Una parte de un sistema que un arquitecto analiza como una unidad, como un módulo, un servicio, una base de datos, una cola o un servidor.

Erosión de la arquitectura
La degradación gradual de un sistema a medida que los cambios infringen su arquitectura prevista, por ejemplo, saltándose capas o añadiendo dependencias prohibidas.

Esqueleto andante (walking skeleton)
Implementación diminuta del sistema que realiza una pequeña función de extremo a extremo a través de todos los componentes arquitectónicos principales; normalmente se construye primero.

Estructura
Conjunto de elementos y las relaciones entre ellos, visto desde un ángulo: por ejemplo, los módulos de código y sus dependencias, o los servicios en ejecución y cómo se comunican.

Feature flag
Un interruptor de configuración que activa o desactiva un comportamiento en tiempo de ejecución sin desplegar código nuevo.

Función de aptitud (fitness function)
Una comprobación automatizada que te dice si un sistema sigue teniendo una propiedad arquitectónica, como una prueba que falla cuando un módulo depende de otro que no debe usar.

Hotspot (punto caliente)
Una parte del código que es a la vez compleja y modificada con frecuencia, que se encuentra combinando el historial del control de versiones con una medida de complejidad; un lugar probable donde saldar deuda.

Inquietud
Un interés que una parte interesada tiene en un sistema, como su costo, su rendimiento, su seguridad, su facilidad de cambio o su cumplimiento legal.

Inquietud transversal (cross-cutting concern)
Una inquietud que afecta a muchas partes de un sistema a la vez, como el registro, la seguridad, el manejo de errores o la persistencia, y que por eso suele decidirse una sola vez para todo el sistema.

Interés (de la deuda técnica)
El esfuerzo, el retraso o el riesgo adicionales que se pagan en cada cambio porque la deuda no se ha saldado.

Ley de Conway
La observación de Melvin Conway de 1968 de que las organizaciones diseñan sistemas cuya estructura refleja la propia estructura de comunicación de la organización.

Matriz de decisión ponderada
Tabla que puntúa cada opción según criterios ponderados para que el razonamiento detrás de una elección sea explícito y pueda cuestionarse.

Matriz poder/interés
Mapa de dos por dos que sitúa a las partes interesadas según su poder sobre un proyecto y su interés en él, y que se usa para planificar con qué intensidad implicar a cada grupo.

Modelo dirigido por riesgos
Enfoque de George Fairbanks en el que la cantidad de trabajo de arquitectura la fijan los riesgos a los que se enfrenta un proyecto: identificar los riesgos, aplicar técnicas y evaluar la reducción.

Monolito
Sistema construido y desplegado como una sola unidad, aunque su código esté dividido internamente en módulos bien separados.

Parte interesada
Persona, equipo u organización con interés en un sistema, como usuarios, patrocinadores, desarrolladores, operadores, responsables de seguridad o reguladores.

Pirámide del conocimiento
Modelo con tres capas: lo que sabes, lo que sabes que no sabes y lo que no sabes que no sabes. Los arquitectos amplían deliberadamente la capa intermedia.

Pista arquitectónica (architectural runway)
En el Scaled Agile Framework, el código, los componentes y la infraestructura existentes necesarios para implementar funcionalidades a corto plazo sin rediseños ni retrasos excesivos.

Principal (de la deuda técnica)
El esfuerzo necesario para corregir una porción de deuda técnica, es decir, para llevar el código o la estructura al estado en que debería estar.

Prueba de concepto
Pequeña porción de código funcional, con un tiempo limitado, construida para comprobar si una idea, una tecnología o una opción de diseño funciona antes de comprometerse con ella.

Punto de vista
Especificación de cómo construir y leer un tipo de vista de arquitectura: a qué partes interesadas e inquietudes sirve y qué convenciones de modelado usa.

Refactorización
Cambiar la estructura interna del código sin cambiar su comportamiento externo, para que sea más fácil de entender y modificar.

Registro de decisiones de arquitectura (ADR)
Un documento breve que registra una decisión significativa: su contexto, las opciones consideradas, la decisión, su estado y sus consecuencias.

Relación
Conexión entre elementos, como un módulo que usa otro, un servicio que llama a otro o un servicio desplegado en un servidor.

Reversibilidad
Lo barato y seguro que resulta deshacer una decisión. Las decisiones muy reversibles pueden tomarse rápido; las irreversibles merecen más análisis.

Serverless
Modelo de nube en el que el proveedor ejecuta tus funciones o flujos de trabajo bajo demanda y factura por uso, de modo que no gestionas servidores tú mismo.

Sesgo cognitivo
Un error sistemático de razonamiento, como el anclaje o la falacia del costo hundido, que lleva a las personas a juzgar las opciones de forma injusta.

Spike (investigación acotada)
Investigación o prototipo breve y con tiempo limitado construido para responder a una pregunta técnica o reducir un riesgo, que normalmente se desecha después.

Staff engineer
Rol sénior de contribuidor individual, por encima de ingeniero sénior, que ejerce liderazgo técnico entre equipos sin dirigir personas; principal engineer suele ser el siguiente nivel.

TOGAF
The Open Group Architecture Framework, un marco y método muy utilizado para la arquitectura empresarial, publicado por The Open Group; su versión actual es el TOGAF Standard, 10th Edition (2022).

Último momento responsable
El punto en el que retrasar más una decisión eliminaría una opción importante; decidir entonces aprovecha la mayor cantidad de información sin causar daño.

Vista
Representación de una o más estructuras de un sistema, dirigida a inquietudes concretas de partes interesadas concretas, como un diagrama de despliegue para el personal de operaciones.