UCAB UNIVERSIDAD CATÓLICA ANDRÉS BELLO

REUTILIZACIÓN ARQUITECTÓNICA

OBJETIVO

Identificar los beneficios y desventajas de los mecanismos de reutilización de diseño, tales como estilos y patrones arquitectónicos, *frameworks* y patrones de diseño.

TABLA DE CONTENIDO

UCAB

INTRODUCCIÓN

Siguiendo la **dinámica** de hacer similitudes entre la arquitectura civil y la arquitectura del *software,* podemos observar que en el mundo de la construcción civil existen diseños a diferentes escalas, que son reutilizados una y otra vez, pues son exitosos en su contexto. Por ejemplo, en las universidades tecnológicas es común que los laboratorios (que tienen exigencias muy particulares) sean ubicados en una edificación especial para ellos; todas las bibliotecas cuentan con espacios para la lectura y espacios para colocar los libros, y así sucesivamente.

Lo mismo sucede con la arquitectura del *software*: ya tenemos más de seis décadas de desarrollo del *software* y podemos observar que muchos diseños cuentan con estructuras similares, nuevamente a diferentes niveles. Estas **estructuras** similares se conocen como "mecanismos arquitectónicos", los cuales son conocidos como estilos y patrones arquitectónicos, *frameworks* y patrones de diseño, que no son más que la reutilización de diseños exitosos a diferentes niveles de abstracción. Estos mecanismos solo son exitosos bajo ciertas circunstancias; por ello, cuando se usen se debe evaluar si se están cumpliendo esas condiciones.

Como siempre, no hay "buen diseño", sino aquel que da menos problemas. En este tema se revisan los mecanismos arquitectónicos de mayor tendencia bajo la óptica de los beneficios y desventajas de utilizarlos.

01 Estilos y patrones arquitectónicos

Si observamos la historia de la arquitectura **convencional**, podemos comprobar que el hombre ha ido haciendo construcciones, ya sea para habitar, estudiar, trabajar, divertirse, etc. que se pudieran clasificar según su forma, materiales, periodo de construcción y demás.

Lo importante es que cada una de ellas era una solución para ese momento. Es decir, no fueron ni buenas ni malas.

1 Estilo arquitectónico: genera un **conjunto** de construcciones que se caracterizan por ser familiares, por ser parecidas.

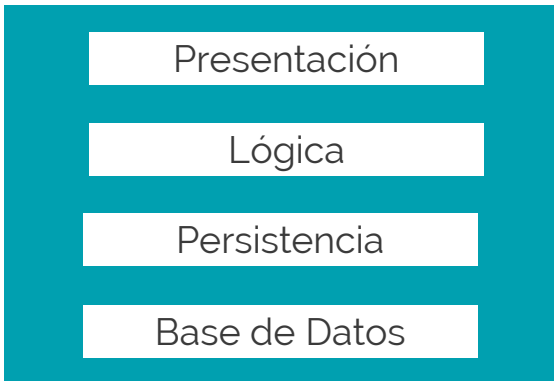
En el ámbito del *software* es igual. En estas últimas cinco décadas se ven familias de *software* que se pueden agrupar en un estilo arquitectónico.

Entonces, un estilo arquitectónico:

- Define una familia de sistemas de *software,* en términos de su organización estructural.
- Determina el vocabulario de los componentes y conectores que se usan para instanciar esa estructura, junto con las restricciones de cómo ellos pueden combinarse.

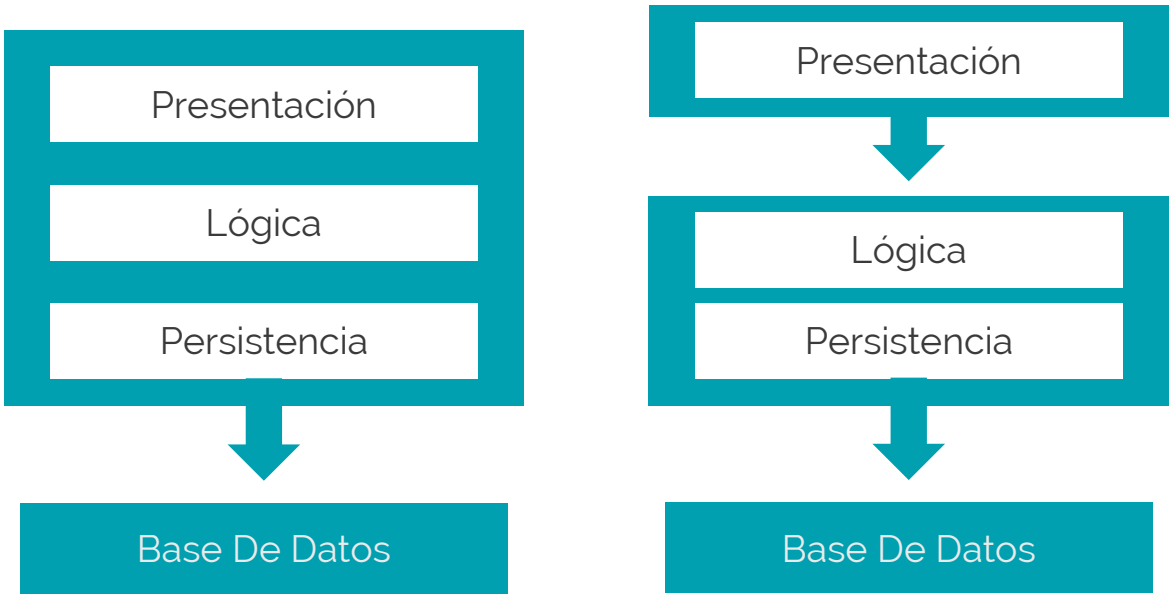
Veamos los estilos arquitectónicos más usados y los que actualmente son tendencia:

Capas (*layers*): en la arquitectura el concepto de capas, que separa diferentes preocupaciones en función de la **funcionalidad**, es tan antiguo como el propio *software*. Sin embargo, el estilo en capas (también conocido como patrón por otros autores) continúa manifestándose en diferentes formas, incluidas variantes modernas. Es uno de los más conocidos y usados, entre otras cosas por su simplicidad y por su similitud con el mundo de los negocios. Todas las organizaciones tienen una "cara"' (presentación), unas reglas de negocio (lógica) y necesitan muchos datos.



Fuente: Capas. Adaptado de Richards y Ford (2020)

También se puede "estructurar" de otras maneras; distribuido, por ejemplo:



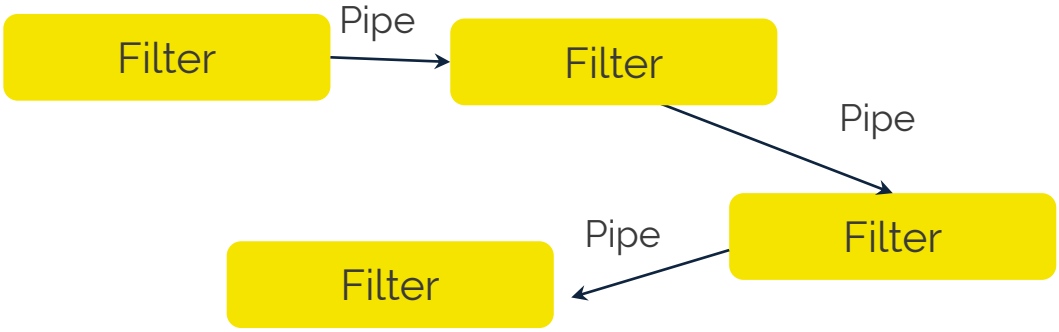
Fuente: Distribuido. Adaptado de Richards y Ford (2020)

Cada capa es **independiente** de las otras capas, por lo que tiene poco o ningún conocimiento del funcionamiento interno de las otras capas en la arquitectura. El aislamiento entre capas también permite reemplazar cualquier capa en la arquitectura sin afectar a ninguna otra.

La arquitectura en capas constituye un buen punto de partida para la mayoría de las aplicaciones, cuando aún no se sabe exactamente qué estilo de arquitectura se utilizará en última instancia. El estilo de arquitectura en capas es una buena opción para aplicaciones o sitios web pequeños y simples, para situaciones con un presupuesto y unas limitaciones de tiempo muy ajustadas.

Es, quizás, uno de los estilos de arquitectura de menor costo, lo que promueve la facilidad de desarrollo para aplicaciones más pequeñas.

Tuberías y filtros (*Pipes* and *filters*): uno de los estilos fundamentales en la arquitectura de *software*, que aparece una y otra vez, es la arquitectura de tuberías (también conocida como arquitectura de tuberías y filtros). Se conoce como el principio **subyacente** detrás de los lenguajes *Shell*, tales como *Unix* (ver siguiente figura):



Fuente: Tuberías y filtros. Adaptado de Richards y Ford (2020)

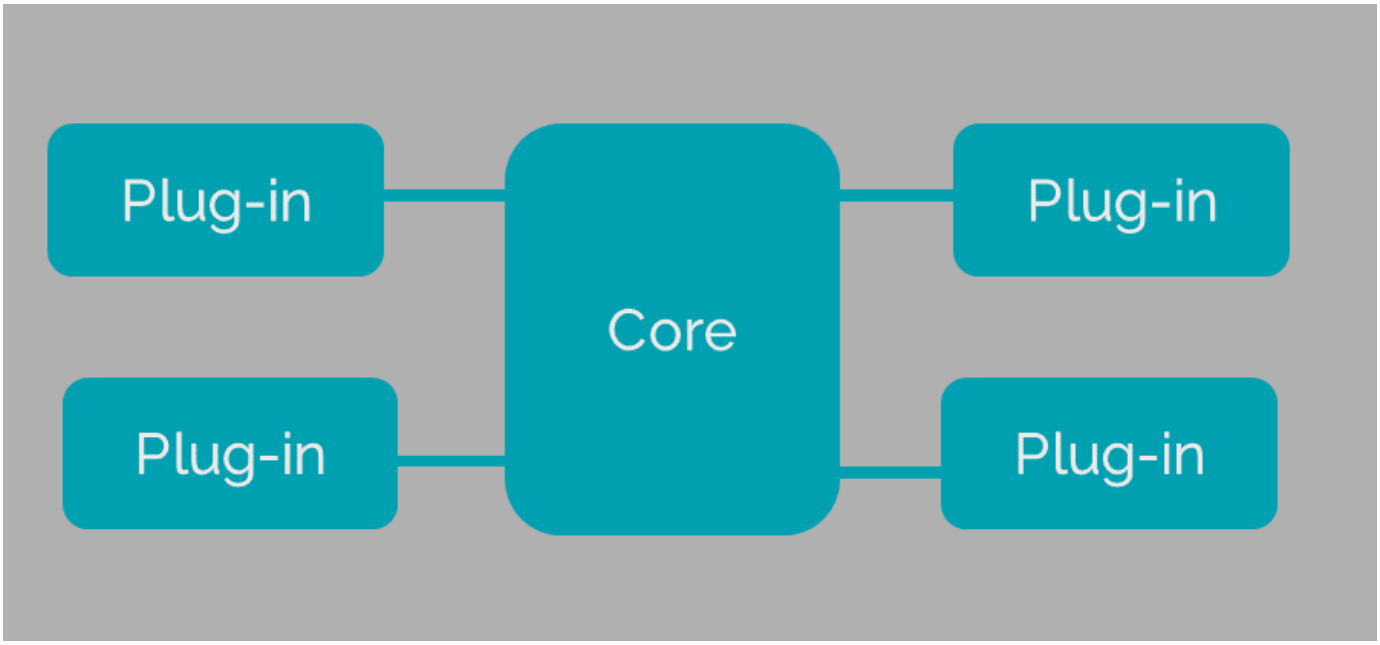
Las tuberías (*pipes*) en esta arquitectura forman el canal de comunicación entre filtros. Cada tubería es típicamente **unidireccional** y por razones de rendimiento acepta la entrada de una fuente y siempre dirige la salida a otra. Los filtros son autónomos, independientes de otros filtros. Estos deben realizar una sola tarea.

La naturaleza unidireccional y la simplicidad de cada una de las tuberías y filtros fomentan la reutilización. Este estilo aparece en una variedad de aplicaciones, especialmente en tareas que facilitan el procesamiento simple y unidireccional.

Por ejemplo, las herramientas de intercambio electrónico de datos (EDI) lo utilizan y crean transformaciones de un tipo de documento a otro, mediante conductos y filtros. Nótese que es un estilo que propicia extensibilidad y no es muy costoso.

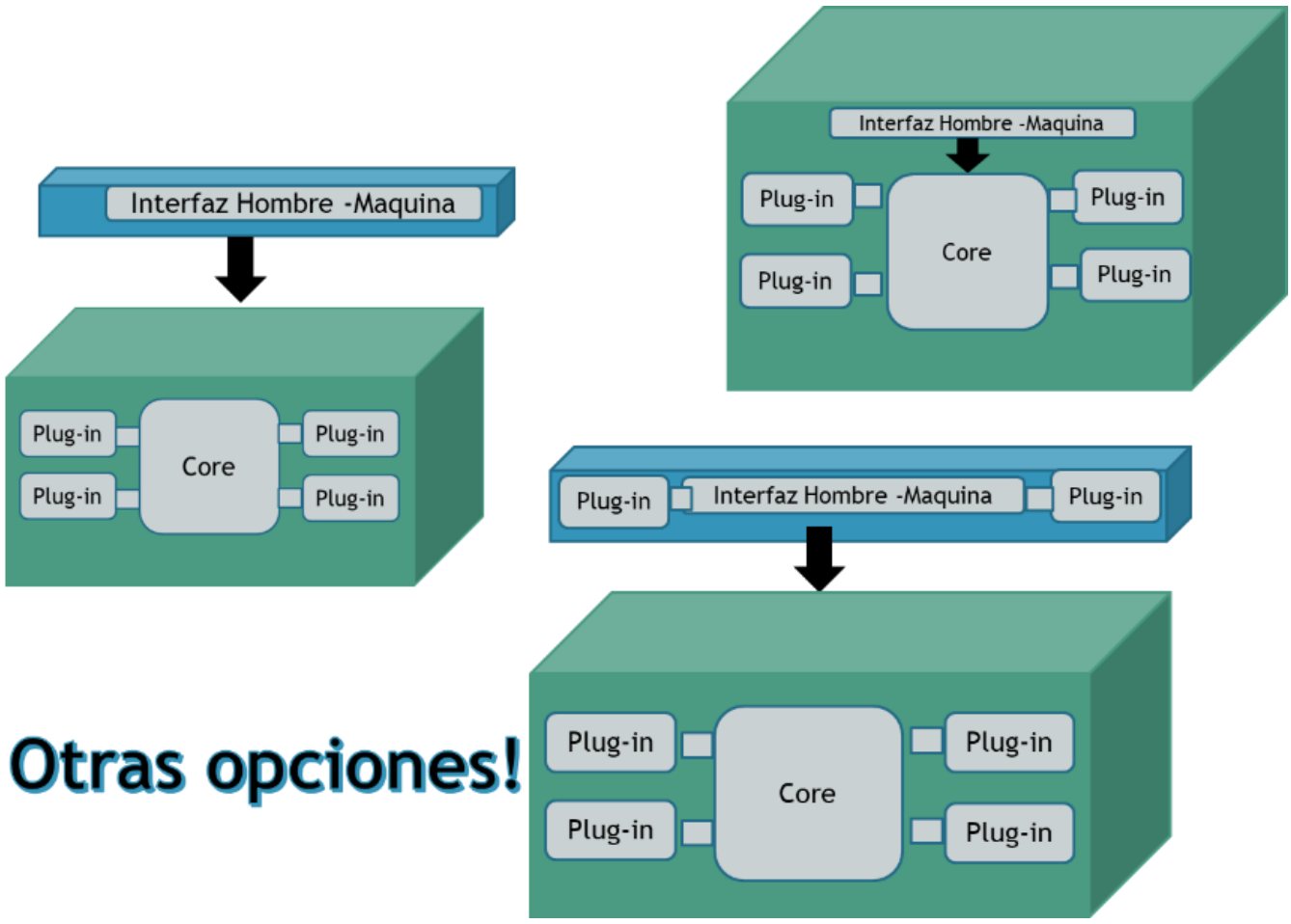
Microkernel: el estilo de arquitectura de microkernel (también conocido como arquitectura de *plug-in*) se acuñó hace varias décadas y todavía se usa ampliamente en la actualidad.

Este estilo es un **ajuste natural** para las aplicaciones basadas en productos (empaquetados y disponibles para su descarga e instalación como una implementación monolítica única, generalmente instalada en el sitio del cliente como un producto de terceros) (ver siguiente figura):



Fuente: Microkernel. Adaptado de Richards y Ford (2020)

Nuevamente, es un estilo con dos componentes. *Core* contiene la funcionalidad mínima requerida para ejecutar el sistema. Este estilo propicia extensibilidad y mantenibilidad.



Fuente: Otras opciones. Adaptado de Richards y Ford (2020)

Los *plug-in* son componentes **independientes** que contienen procesamiento específico, funciones adicionales y código propio, destinados a mejorar o ampliar el *core*. Idealmente, los *plug-in* deberían ser independientes entre sí. Esta independencia entre ellos propicia extensibilidad y rendimiento. El *core* necesita saber qué *plug-ins* están disponibles y cómo acceder a ellos.

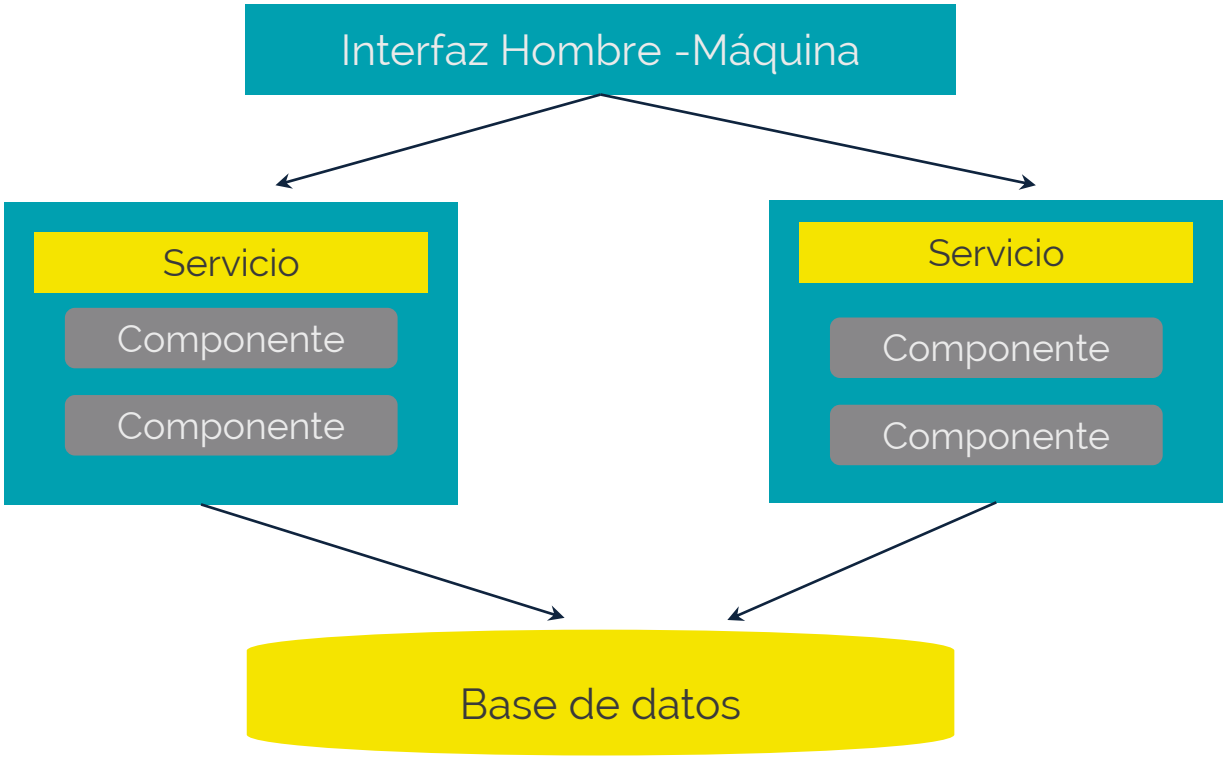
Una forma común de implementar esto es a través de un registro de *plug-in*. Este registro contiene su nombre, contrato de datos y detalles del protocolo de acceso remoto.

Finalmente, al estar "aislada" la funcionalidad, es un estilo que propicia la facilidad de prueba.

D

Basados en servicios: aunque **la arquitectura** basada en servicios es una arquitectura distribuida, no tiene el mismo nivel de complejidad y costo que otras arquitecturas distribuidas, al igual que la basada en microservicios o por eventos, lo que la convierte en una opción muy popular para muchas aplicaciones relacionadas con las empresas.

Nótese que es una arquitectura distribuida en capas macro y que consta de una interfaz de usuario, servicios generales remotos implementados y una base de datos monolítica. Todo implementado por separado (ver la figura):



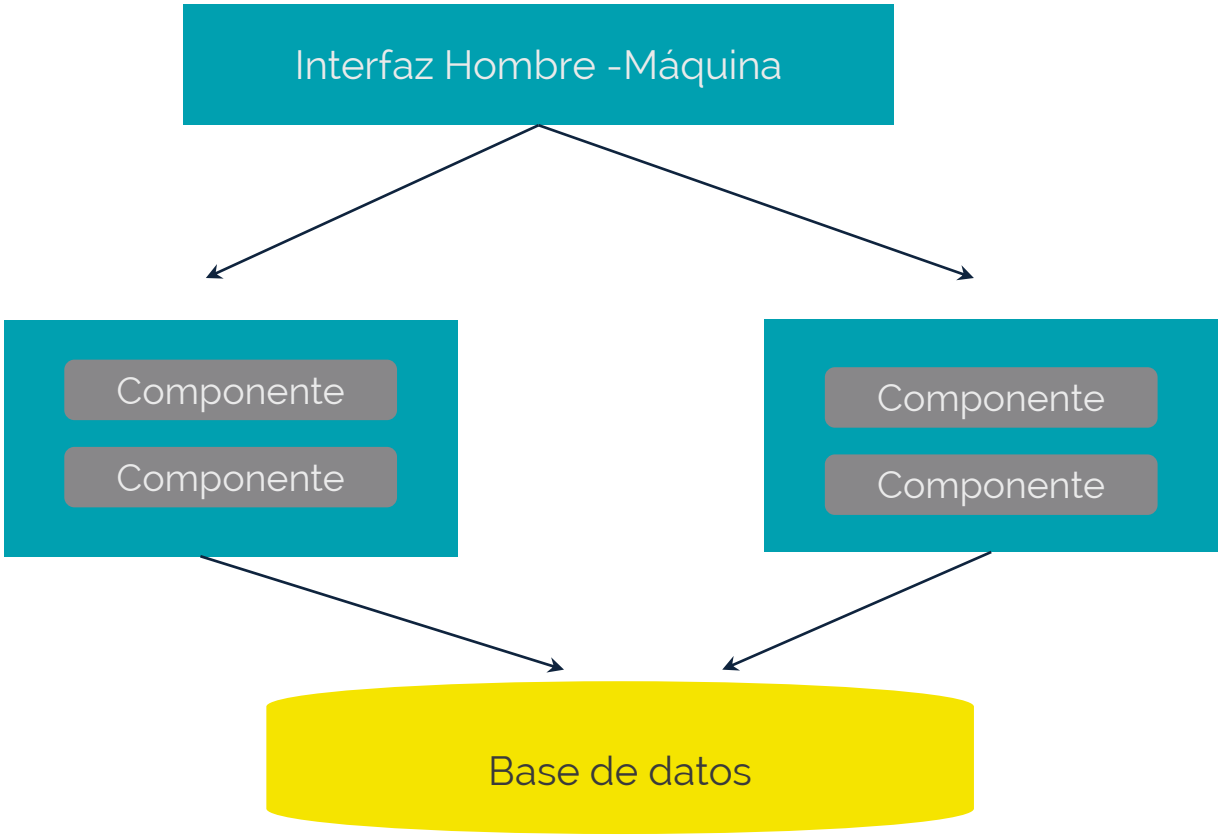
Fuente: Arquitectura basada en servicios. Adaptado de Richards y Ford (2020)

Los servicios son típicamente "partes de una aplicación" (generalmente llamadas "servicios de dominio") que son independientes. En la mayoría de los casos existe una única instancia de cada servicio de dominio. Sin embargo, según las necesidades de **escalabilidad**, tolerancia a fallas y rendimiento, pueden existir varias instancias de un servicio. Esto requiere algún tipo de equilibrio de carga entre la interfaz de usuario y el servicio de dominio.

Se accede a los servicios de forma remota desde una interfaz de usuario y mediante un protocolo de acceso remoto. Si bien el protocolo *REST* es el más común, también se puede usar **mensajería**, llamada a procedimiento remoto (RPC) o incluso *SOAP*. También puede ser una capa *API*.

Normalmente, utiliza una base de datos **compartida** de forma centralizada, con pocos servicios, de 4 a 12.

Debido a que los servicios de dominio generalmente son de grano grueso, cada servicio de dominio generalmente se diseña utilizando un estilo de arquitectura en capas que consta de una capa de fachada *API*, una capa empresarial y una capa de persistencia (ver la figura):



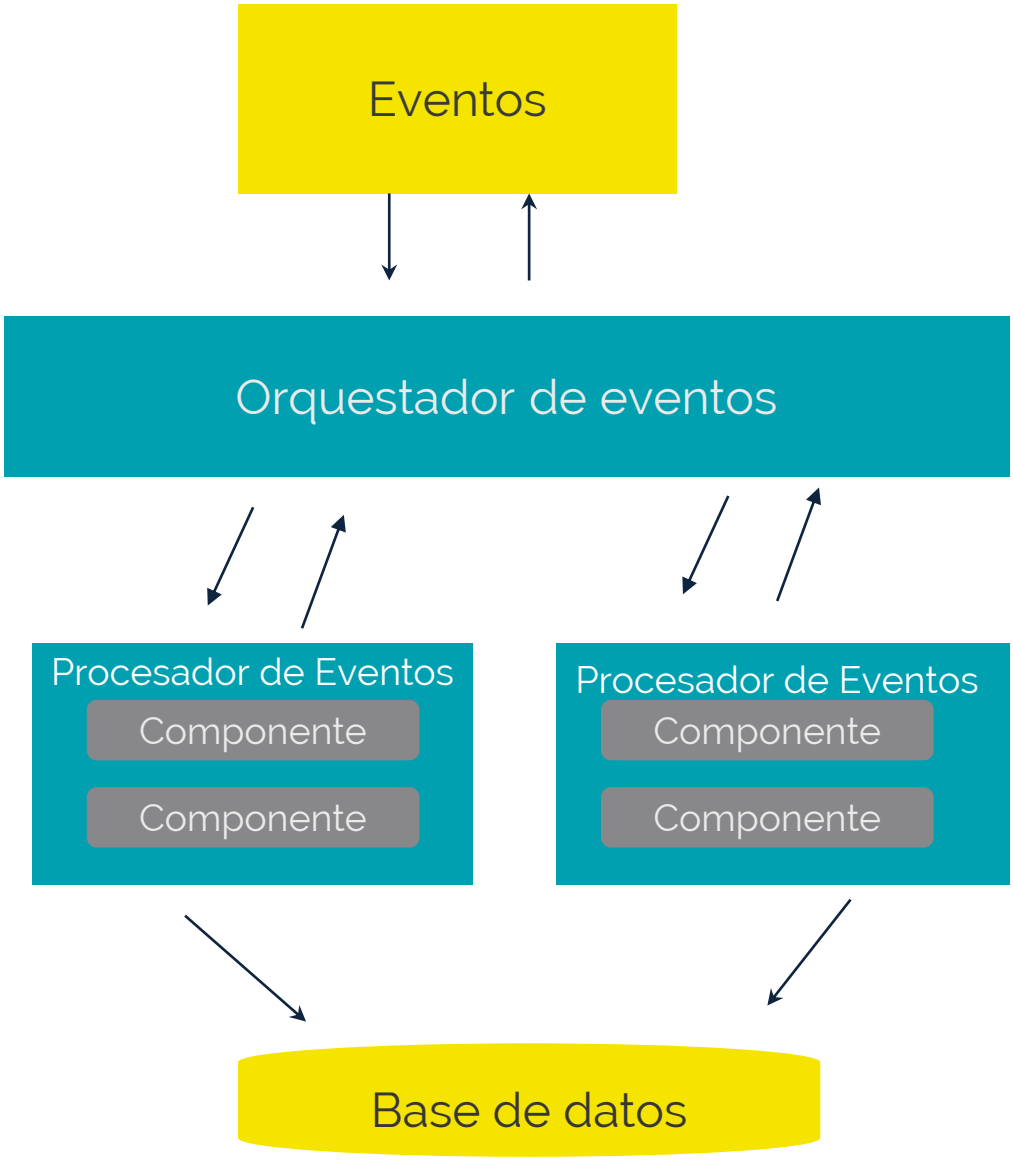
Fuente: Servicios de dominio. Adaptado de Richards y Ford (2020)

Dado que tenemos funcionalidad repartida en servicios, se permite hacer **cambios rápidos** (facilidad de cambio). Propicia la facilidad de prueba, debido al alcance limitado del dominio. La tolerancia a fallas y la disponibilidad general de la aplicación también se propician con este estilo, porque los servicios suelen ser independientes. Si un servicio deja de funcionar no debería afectar a los otros.

E

Basados en eventos: el estilo de arquitectura orientado por eventos es uno de arquitectura asincrónica, distribuida para aplicaciones altamente escalables y de alto rendimiento. Se puede utilizar tanto para aplicaciones pequeñas como para grandes y complejas.

Se constituye de componentes **desacoplados** que procesan los eventos de forma asíncrona. Se puede usar como un estilo de arquitectura independiente o incrustado en otros (como una arquitectura de microservicios basada en eventos) (ver la figura):



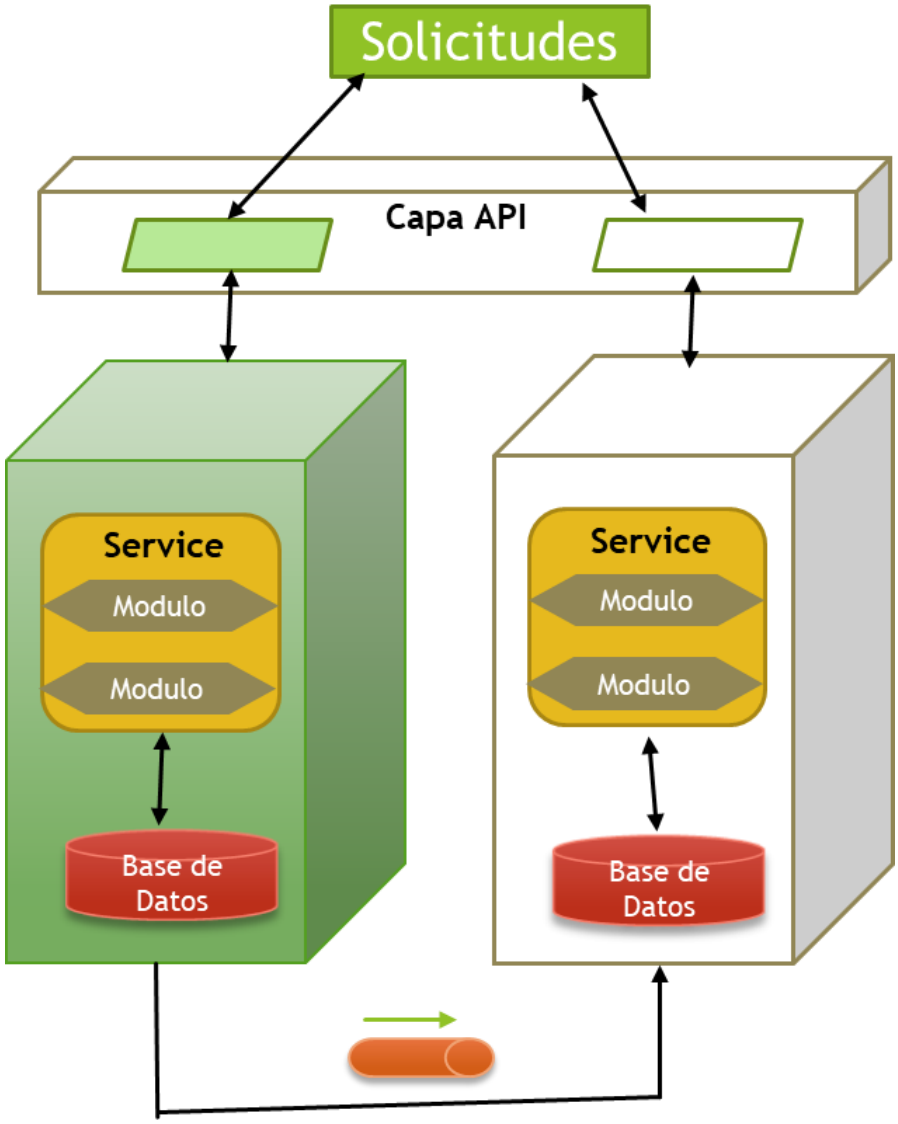
Fuente: Arquitectura basada en eventos. Adaptado de Richards y Ford (2020)

Los procesadores manejan el evento, ya sea recuperando o actualizando información en una base de datos. El orquestador reacciona a una situación particular y toma medidas en función del evento.

Este estilo ofrece una **característica** única, en el sentido de que se basa únicamente en la comunicación asincrónica, tanto para el procesamiento de disparar y olvidar (no se requiere respuesta) como para el procesamiento del evento/respuesta (respuesta requerida del consumidor del evento). La comunicación asincrónica es una técnica poderosa para aumentar la capacidad de respuesta (eficiencia) general de un sistema.

F **Basados en microservicios:** es un estilo de arquitectura muy popular que ha ganado un impulso significativo en los últimos años:

El término "arquitectura de microservicios" ha surgido en los últimos años para describir una forma particular de diseñar **aplicaciones** de software como conjuntos de servicios desplegables de forma independiente. Si bien no existe una definición precisa de este estilo arquitectónico, existen ciertas características comunes en torno a su organización, a la capacidad empresarial, a la implementación automatizada, a la inteligencia en los endpoint y al control descentralizado de idiomas y datos.



Fuente: Arquitectura basada en microservicios. Traducido de Fowler (2014)

Surge del diseño impulsado por dominios (DDD), un proceso de diseño lógico para proyectos de *software*. El contexto limitado es un concepto en particular de DDD, en el que se inspiran los microservicios. Dentro de un contexto limitado las partes internas, como el código y los esquemas de datos, se combinan para producir el trabajo; pero nunca se acoplan a nada fuera del contexto limitado, como una base de datos o una definición de clase de otro contexto limitado. Esto permite que cada contexto defina solo lo que necesita, en lugar de adaptarse a otros componentes. Así, cada **microservicio** incluye todo lo necesario para operar dentro de la aplicación, incluyendo clases, otros subcomponentes y esquemas de base de datos. Los microservicios llevan al extremo el concepto de una arquitectura con particiones de dominio. Cada servicio está destinado a representar un dominio o subdominio.

¡Recuerda! Cuando se favorece la reutilización, también se favorece el acoplamiento para lograr esa reutilización, ya sea por herencia o composición. Sin embargo, si se requieren altos grados de desacoplamiento, entonces se favorece la duplicación sobre la reutilización. El objetivo principal de los microservicios es un alto desacoplamiento, modelando físicamente la noción lógica de contexto limitado.

Los arquitectos esperan que cada servicio incluya todas las partes necesarias para operar de forma independiente, incluidas las bases de datos y otros componentes dependientes.

Otros autores no hablan de estilo arquitectónico, sino de patrón arquitectónico.

2 Patrones arquitectónicos: se refiere a un nivel de abstracción más bajo. Buschmann et al. (1996) definen "patrón" como una **regla** que consta de tres partes, la cual expresa una relación entre un contexto, un problema y una solución. Estas son:

- Contexto: es una situación de diseño en la que aparece un problema de diseño.
- Problema: es un conjunto de fuerzas que aparecen repetidamente en el contexto
- Solución: es una configuración que equilibra estas fuerzas.

¿Quién en su casa no tiene el lavadero muy cerca de la cocina?

Este es un patrón arquitectónico en la arquitectura de obras civiles.

La selección de un **patrón** arquitectónico es, por lo tanto, una decisión fundamental de diseño en el desarrollo de un sistema de *software*. Buschmann propone los siguientes patrones arquitectónicos. Notarás que en algunos casos coinciden con los estilos arquitectónicos que acabamos de describir:

Patrón arquitectónico	Descripción
Presentation Abstraction Control	Define una estructura para sistemas de *software* interactivos de agentes de cooperación organizados de forma jerárquica. Cada agente es responsable de un aspecto específico de la funcionalidad de la aplicación y consiste de tres componentes: presentación, abstracción y control.
Microkernel	Aplica para sistemas de *software* que deben estar en capacidad de adaptar los requerimientos de cambio del sistema. Separa un núcleo funcional mínimo del resto de la funcionalidad y de partes específicas pertenecientes al cliente.
Reflection	Provee un mecanismo para sistemas cuya estructura y comportamiento cambia dinámicamente. Soporta la modificación de aspectos fundamentales como estructuras tipo y mecanismos de llamadas a funciones.
Layers	Consiste en estructurar aplicaciones que pueden ser descompuestas en grupos de subtareas, las cuales se clasifican de acuerdo a un nivel particular de abstracción.

Pipes and filters	Provee una estructura para los sistemas que procesan un flujo de datos. Cada paso de procesamiento está encapsulado en un componente filtro (*filter*). El dato pasa a través de conexiones (*pipes*), entre filtros adyacentes.
Blackboard	Aplica para problemas cuya solución utiliza estrategias no determinísticas. Varios subsistemas ensamblan su conocimiento para construir una posible solución parcial o aproximada.
Broker	Puede ser usado para estructurar sistemas de *software* distribuido con componentes desacoplados que interactúan por invocaciones a servicios remotos. Un componente *broker* es responsable de coordinar la comunicación, como el reenvío de solicitudes, así como también la transmisión de resultados y excepciones.
Model View controler	Divide una aplicación interactiva en tres componentes. El modelo (*model*) contiene la información central y los datos. Las vistas (view) despliegan información al usuario. Los controladores (*controlers*) capturan la entrada del usuario. Las vistas y los controladores.

Fuente: Patrones arquitectónicos. Extraída de Buschmann et al. (1996)

Otro mecanismo arquitectónico reutilizable en *software* son los *frameworks*. Veamos qué son.

02 Framework

Un *framework* es más que una **jerarquía** de clases: es una jerarquía de clases más un modelo de interacción entre los objetos instanciados a partir del mismo. Un *framework* es una técnica de reutilización, es un diseño reutilizable de todo, o partes del sistema que están representadas por un conjunto de clases abstractas y la forma en que sus instancias interactúan.

- Un *framework* es un esqueleto de una aplicación que puede ser personalizado por un **desarrollador** de aplicaciones; son más personalizables que los componentes y tiene interfaces más complejas. Son una clase de arquitectura de dominio específico.

- La diferencia principal es que un *framework* es un diseño orientado a objetos, mientras que una arquitectura de un dominio específico puede no serlo.

03 Patrón de diseño

Un pattern describe un **problema** a ser resuelto, una solución y el contexto en el cual la solución trabaja. Nomina una técnica, describe su costo y su beneficio. Estos *pattern* fueron descubiertos al examinar varios *frameworks* y fueron seleccionados como representativos de *software* reutilizable. Los *design pattern* son elementos microarquitecturales.

El primero en identificarlos fue Gamma (Gamma et al, 1995) y los clasificó según la siguiente tabla:

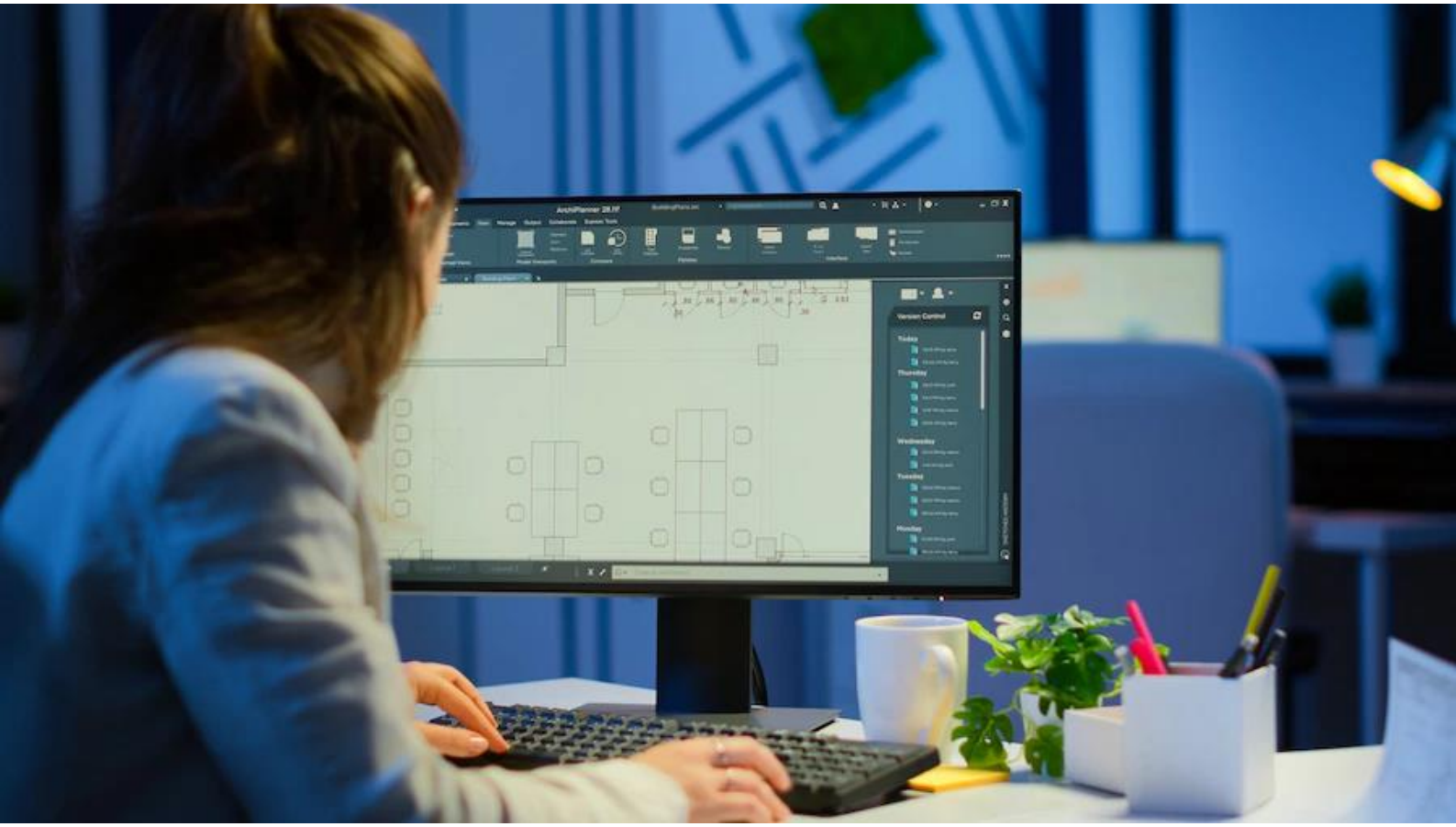
ALCANCE	PROPÓSITO		
CLASE	CREACIÓN	ESTRUCTURAL	COMPORTAMIENTO
	Factory method	*Adapter Clss*	*Interpreter*
OBJECT	*Abstract factory*	*Adapter Object*	*Chain of*
			Responsibility
	Builder	*Bridge*	*Iterator*
	Protitype	*Composite*	*Mediator*
	Singleton	*Decorator*	*Memento*
		Facade	*State*
		Flyweight	*Straregy*
		Proxy	*Visitor*

Fuente: Design pattern. Extraída de Gamma et al. (1995)

El **propósito**, entonces, es garantizar que nuestros sistemas de *software* cuenten con la arquitectura de software que menos problemas dé y que satisfaga las cualidades arquitectónicas que el *software* va solicitando a medida que va evolucionando.

Para realizar esta difícil tarea, en la actualidad el arquitecto cuenta con muchos mecanismos de reutilización de diseño, tales como los estilos y patrones arquitectónicos, los *frameworks* y los patrones de diseño.

Todos ellos son exitosos según un contexto. Nosotros tenemos que tener claro, entonces, cuál es nuestro contexto para así poder reutilizarlo.



REFERENCIAS

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. y Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley.

Gamma, E., Helm, R., Johnson, R. y Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley.

Richard, M. y Ford, N. (2020). *Fundamentals of Software Architecture: An Engineering Approach.* O'Reilly Inc.

Referencias de las imágenes

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. y Stal, M. (1996). Patrones arquitectónicos [Imagen]. Recuperado de *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley.

Fowler, M. (25 de marzo 2014). Arquitectura basada en microservicios [Imagen]. Recuperado de *Microservices*. martinFowler. https://martinfowler.com/articles/microservices.html

Gamma, E., Helm, R., Johnson, R. y Vlissides, J. (1995). Design pattern [Imagen]. Recuperado de *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley.

Richard, M. y Ford, N. (2020). Arquitectura basada en eventos[Imagen]. Recuperado de *Fundamentals of Software Architecture*: *An Engineering Approach.* O'Reilly Inc.

Richard, M. y Ford, N. (2020). Arquitectura basada en servicios [Imagen]. Recuperado de *Fundamentals of Software Architecture*: *An Engineering Approach.* O'Reilly Inc.

Richard, M. y Ford, N. (2020). Capas [Imagen]. Recuperado de *Fundamentals of Software Architecture*: *An Engineering Approach.* O'Reilly Inc.

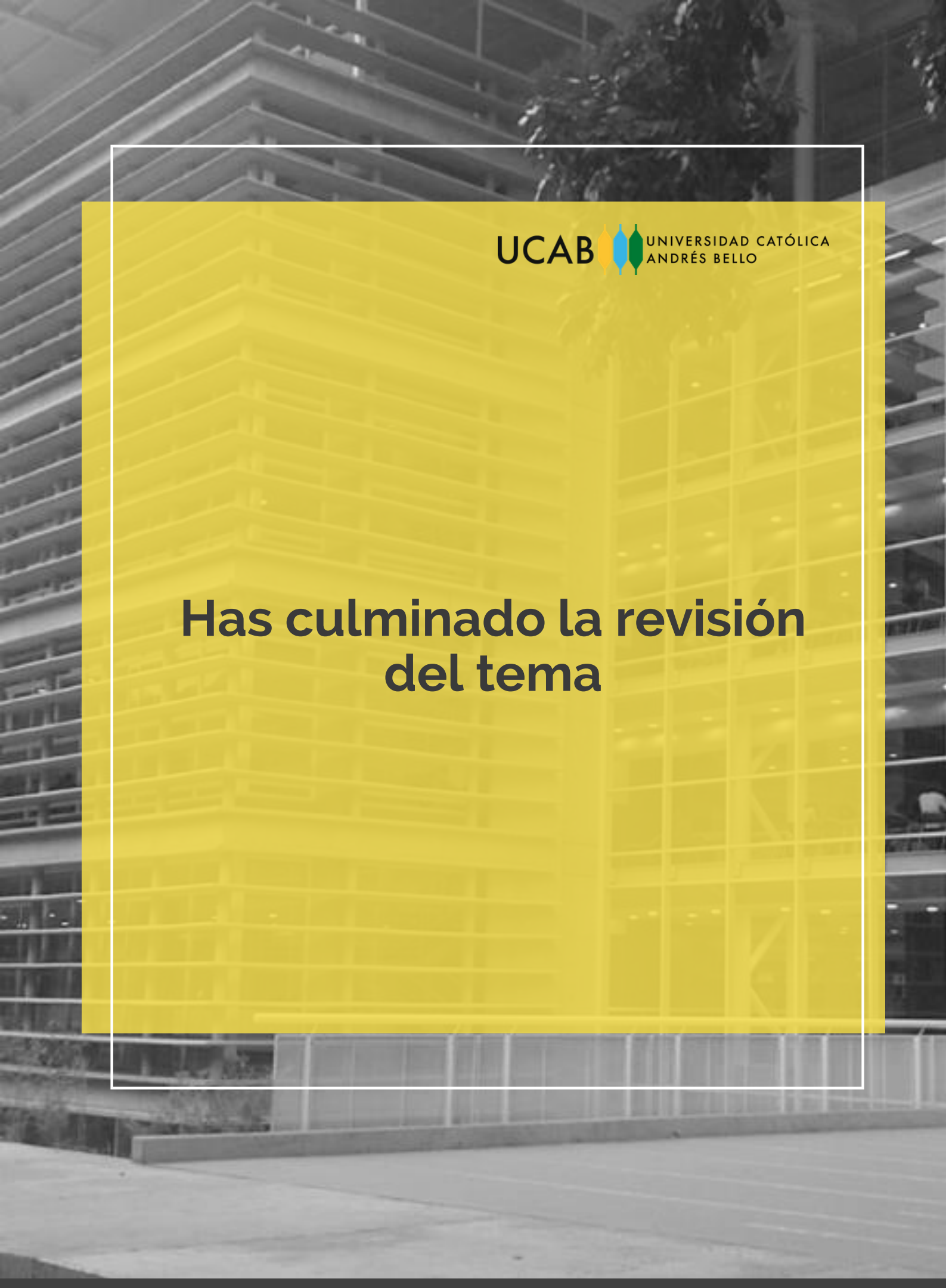
Richard, M. y Ford, N. (2020). Distribuido [Imagen]. Recuperado de *Fundamentals of Software Architecture*: *An Engineering Approach.* O'Reilly Inc.

Richard, M. y Ford, N. (2020). Microkernel [Imagen]. Recuperado de *Fundamentals of Software Architecture*: *An Engineering Approach.* O'Reilly Inc.

Richard, M. y Ford, N. (2020). Otras opciones [Imagen]. Recuperado de *Fundamentals of Software Architecture*: *An Engineering Approach.* O'Reilly Inc.

Richard, M. y Ford, N. (2020). Servicios de dominio [Imagen]. Recuperado de *Fundamentals of Software Architecture*: *An Engineering Approach.* O'Reilly Inc.

Richard, M. y Ford, N. (2020). Tuberías y filtros [Imagen]. Recuperado de *Fundamentals of Software Architecture*: *An Engineering Approach.* O'Reilly Inc.



UCAB UNIVERSIDAD CATÓLICA ANDRÉS BELLO

Has culminado la revisión del tema