Otros patrones arquitecturales

Diseño de Sistemas Software
Curso 2023/2024

Óscar Segura Amorós
Carlos Pérez Sancho

Universitat d'Alacant
Universidad de Alicante
Departament de Llenguatges i Sistemes Informàtics
Departamento de Lenguajes y Sistemas Informáticos

Contenidos

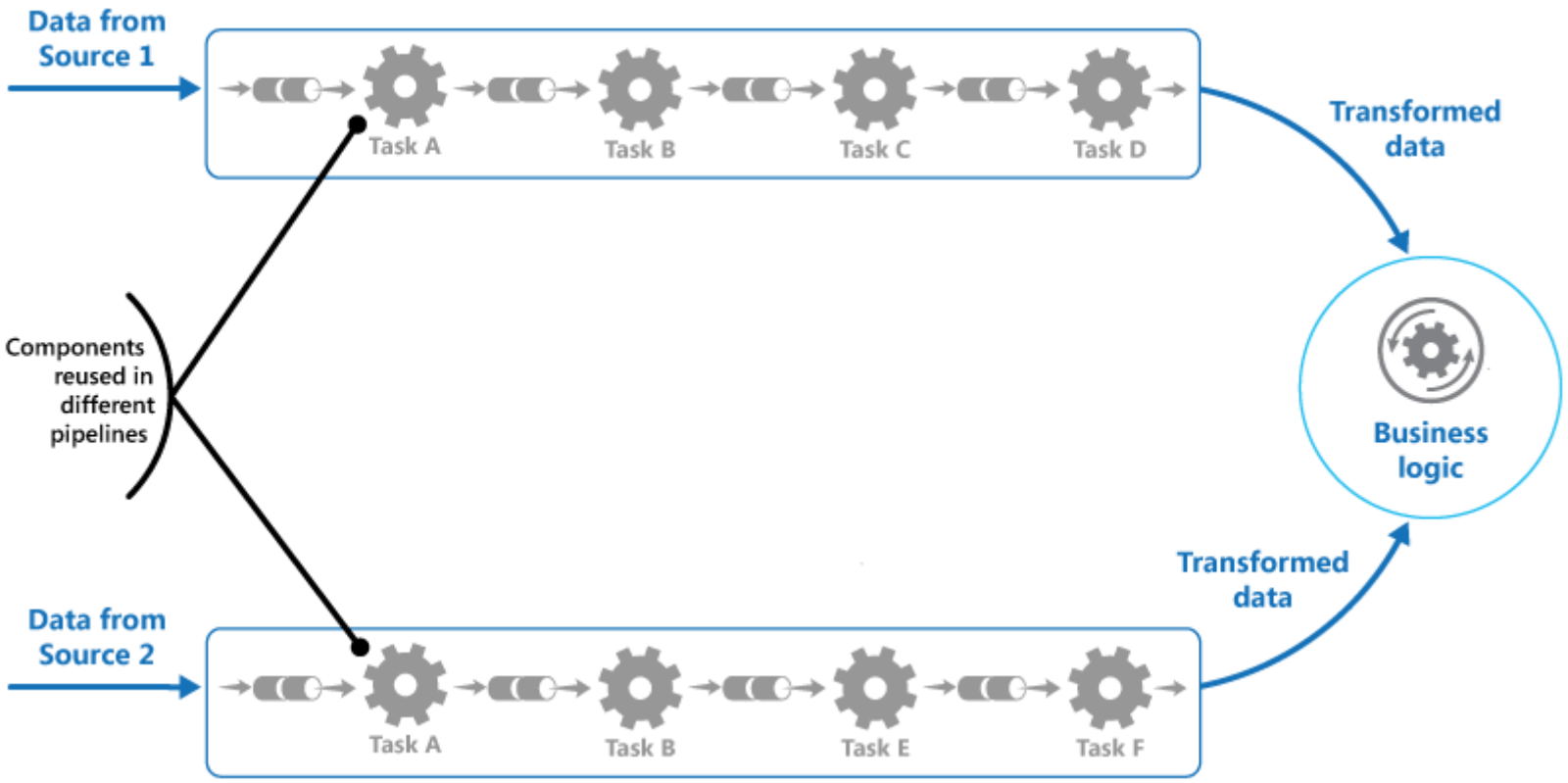
- ~~Arquitectura en Capas~~
- Arquitectura tuberías y filtros
- Arquitectura microkernel
- Arquitectura orientada a eventos
- Arquitectura de microservicios
- Patrones de distribución
  - Patrón Remote Facade
  - Patrón Data Transfer Object

Arquitectura tuberías y filtros

Patrones arquitecturales

Arquitectura tuberías y filtros

Los datos pasan por una serie de filtros, que transforman la información



Arquitectura tuberías y filtros

- Muy usado en sistema Unix, aunque no tiene por qué implementarse necesariamente mediante tuberías y línea de comandos
- Filosofía Unix
  - Modularidad
  - Simplicidad
  - Composición mediante el encadenamiento de programas sencillos para realizar tareas complejas
  - «Haz una cosa y hazla bien»

Arquitectura tuberías y filtros

- Otros usos comunes
  - Procesamiento de gráficos y multimedia
  - Compiladores
  - Preprocesamiento de datos para *data analysis* e inteligencia artificial

Arquitectura tuberías y filtros

- Ejemplo: Apertium
  https://www.apertium.org

- Desarrollo de aplicaciones derivadas mediante la reutilización de módulos (interNOSTRUM → Apertium)

Arquitectura tuberías y filtros

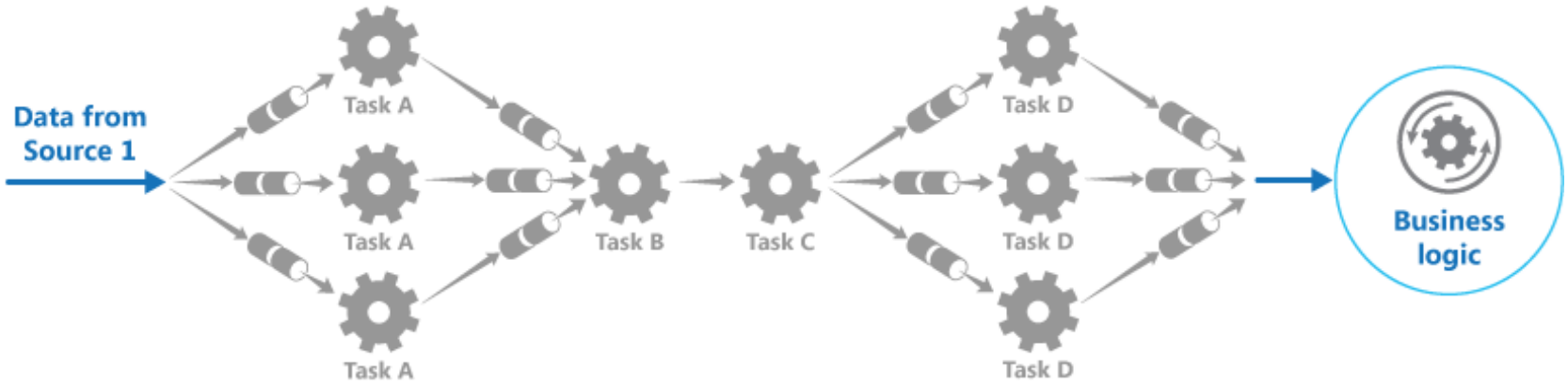
- Ventajas
  - Bajo acoplamiento entre componentes
  - Se pueden probar los componentes individualmente
  - Facilidad para añadir componentes nuevos o reordenar los existentes
  - Se puede hacer cambios en un componente de forma ágil
  - Los filtros actúan como “cajas negras”
  - Reusabilidad

Arquitectura tuberías y filtros

- Inconvenientes
  - Cada filtro debe procesar, transformar y enviar los datos de entrada, puede afectar al rendimiento
  - No apta para aplicaciones con mucha interactividad
  - Se pueden crear cuellos de botella en algún filtro, dejando a los restantes inactivos

Arquitectura tuberías y filtros

Se puede mejorar la escalabilidad paralelizando los filtros que requieren mayor capacidad de cómputo

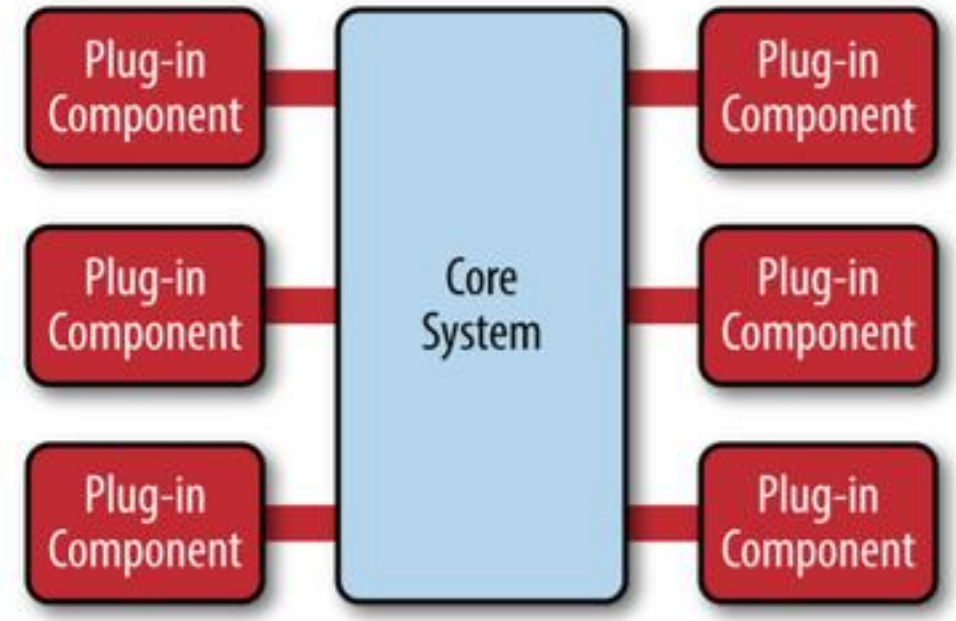


Arquitectura microkernel

Patrones arquitecturales

Arquitectura microkernel

También conocida como arquitectura de *plugins*, consiste en un núcleo central que se puede extender mediante módulos

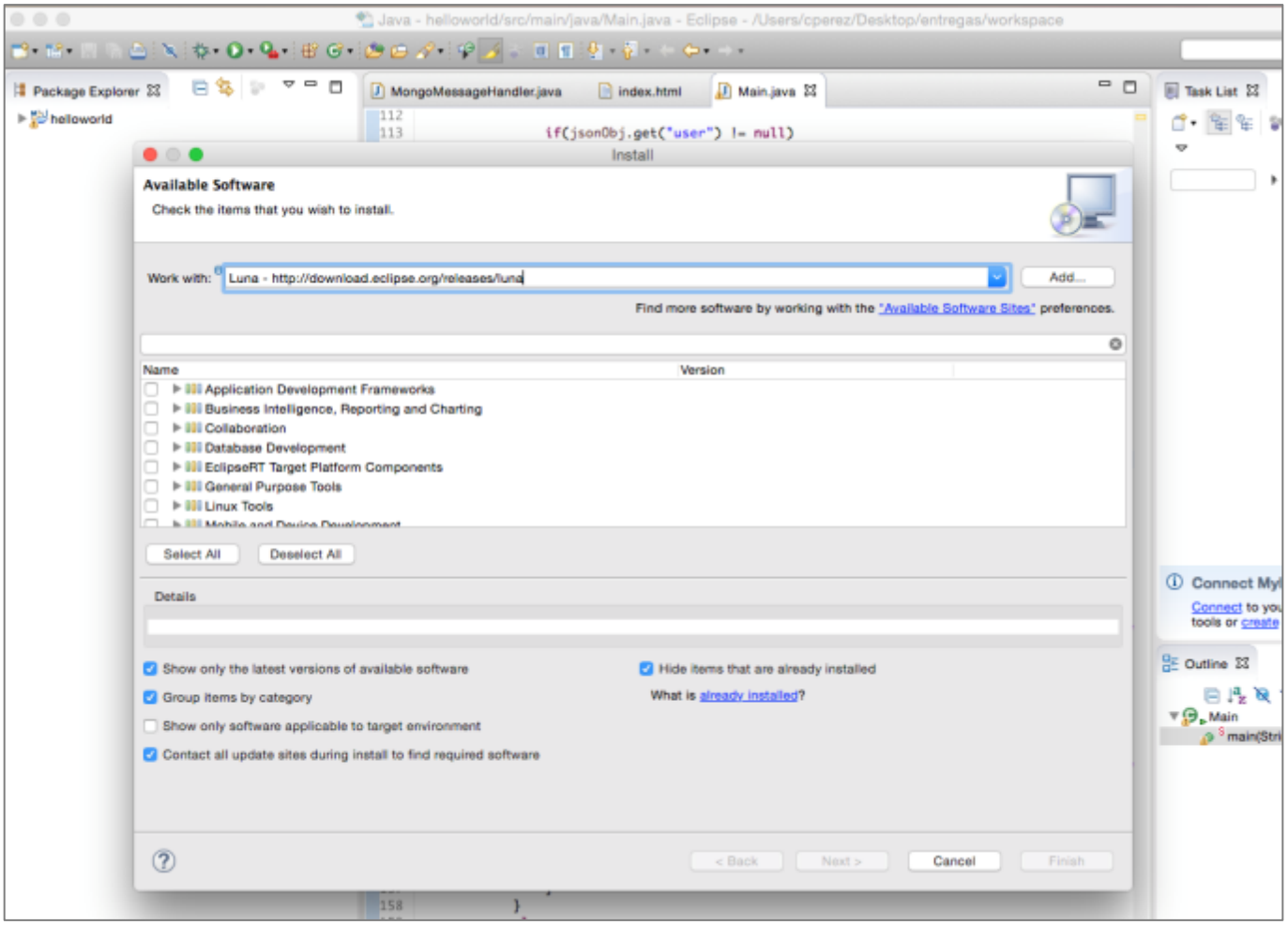


Arquitectura microkernel

- Componentes
  - **Núcleo central**: contiene la funcionalidad mínima para que el sistema funcione
  - ***Plugins***
    - Componentes independientes que añaden funcionalidades al núcleo central
    - Puede haber dependencias entre *plugins*
    - Se suelen organizar en un registro o repositorio para que el núcleo pueda obtener los *plugins* necesarios

Arquitectura microkernel

Ejemplo: Eclipse IDE



Arquitectura microkernel

- Muy adecuado para el diseño y desarrollo evolutivo e incremental, y aplicaciones “producto”
- Análisis del patrón
  - **Agilidad**: alta, es sencillo añadir nuevos componentes
  - **Despliegue**: sencillo, se pueden añadir nuevos plugins en tiempo de ejecución
  - **Pruebas**: sencillo, se pueden probar los plugins por separado
  - **Rendimiento**: normalmente alto, ya que se pueden instalar únicamente los plugins necesarios
  - **Escalabilidad**: baja, normalmente diseñado como un único ejecutable
  - **Desarrollo**: difícil, el diseño del interfaz de los plugins debe planearse cuidadosamente. La gestión de versiones y repositorios de plugins añade complejidad

Arquitectura orientada a eventos

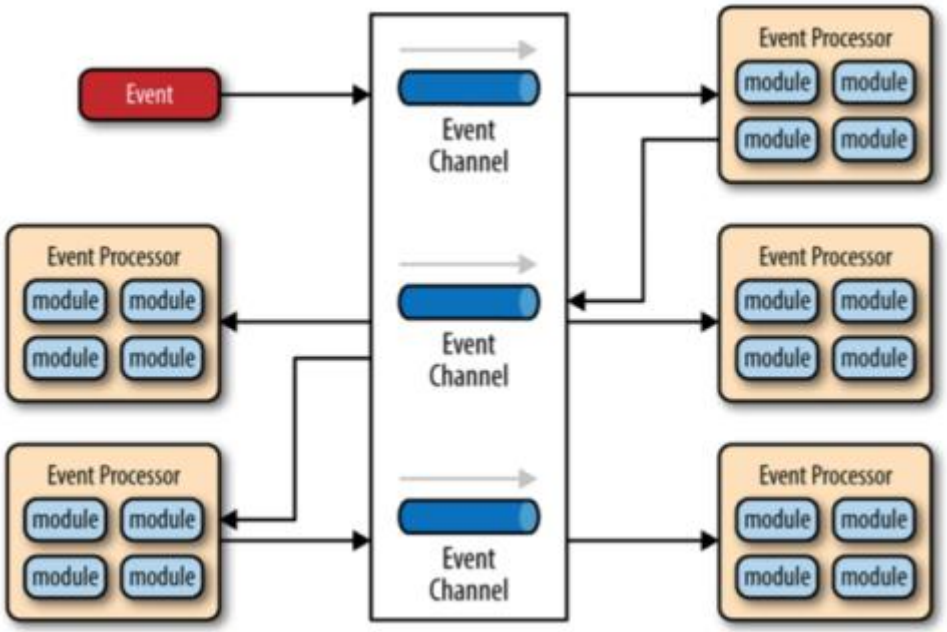
Patrones arquitecturales

Arquitectura orientada a eventos

- El sistema se compone de pequeños componentes que responden a eventos, y de algún mecanismo para gestionar las colas de eventos que se reciben

Arquitectura orientada a eventos

- Los procesadores de eventos se encadenan unos con otros mediante eventos que pasan a través del bus de eventos
- Cuando un procesador de eventos termina su trabajo, genera un evento para que se ejecuten los siguientes componentes

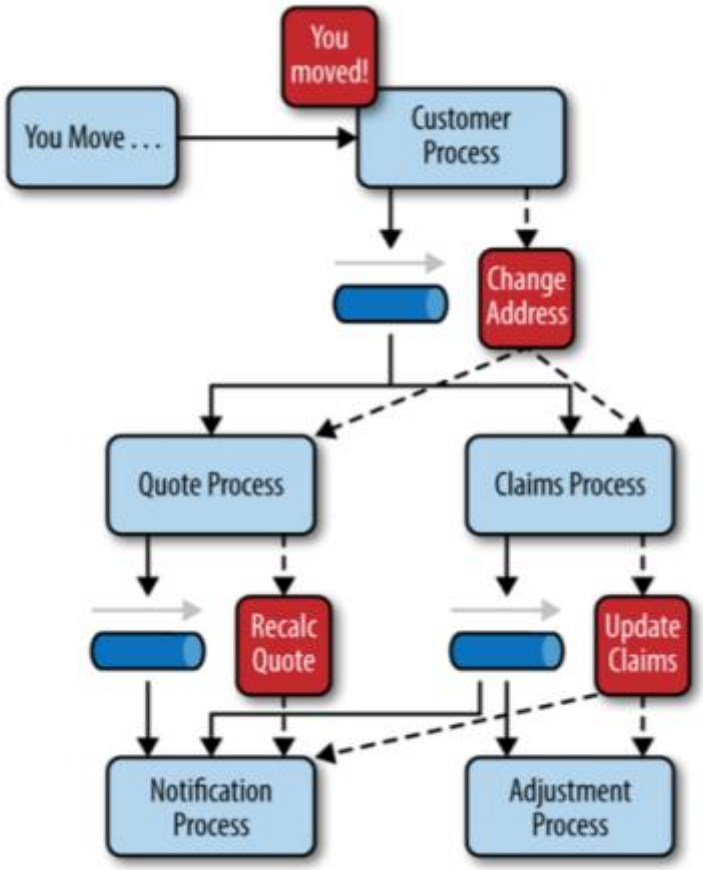


Arquitectura orientada a eventos

- Componentes
  - **Procesadores de eventos**
    - Contienen la lógica de negocio
    - Pequeñas unidades autocontenidas y altamente independientes del resto
  - **Bus de eventos**
    - Se encarga de gestionar las colas de eventos para que los procesadores no tengan que preocuparse de los detalles de implementación

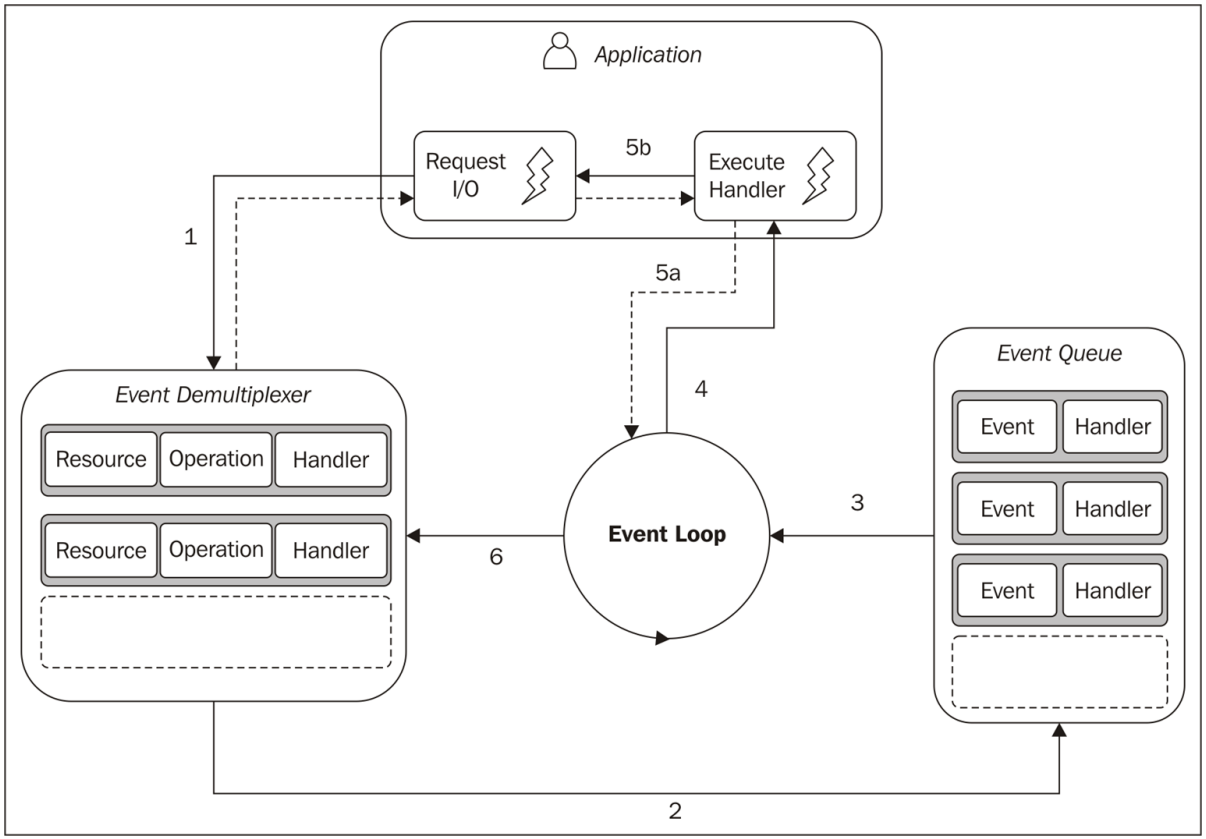
Arquitectura orientada a eventos

Ejemplo: un cliente de una aseguradora cambia de domicilio



Arquitectura orientada a eventos

Ejemplo: Node.js



Arquitectura orientada a eventos

- Ventajas
  - Bajo acoplamiento entre componentes
  - Los componentes pueden escalar por separado

Arquitectura orientada a eventos

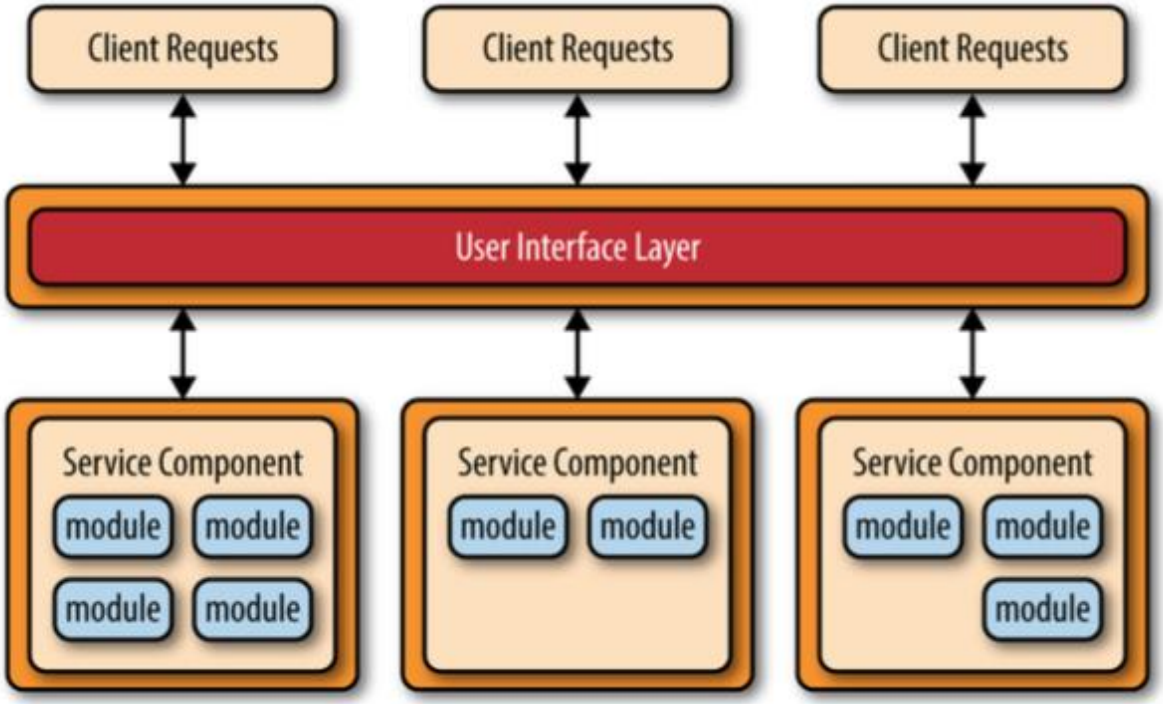
- Inconvenientes
  - Require herramientas especializadas para gestionar eventos
  - Por su naturaleza asíncrona puede dar lugar a condiciones de carrera

Arquitectura de microservicios

Patrones arquitecturales

Arquitectura de microservicios

Arquitectura distribuida, el cliente se comunica con los servicios mediante algún protocolo de acceso remoto



Arquitectura de microservicios

- La implementación más frecuente utiliza HTTP como protocolo de comunicación, definiendo interfaces REST (REpresentational State Transfer) para acceder a los servicios
  - Cada componente (servicio) ofrece datos a través de URLs
  - Los componentes también se pueden comunicar entre sí a través de estas URLs (un componente puede ser a la vez servidor y cliente de otros servicios)

Arquitectura de microservicios

- Ejemplo de interfaz REST

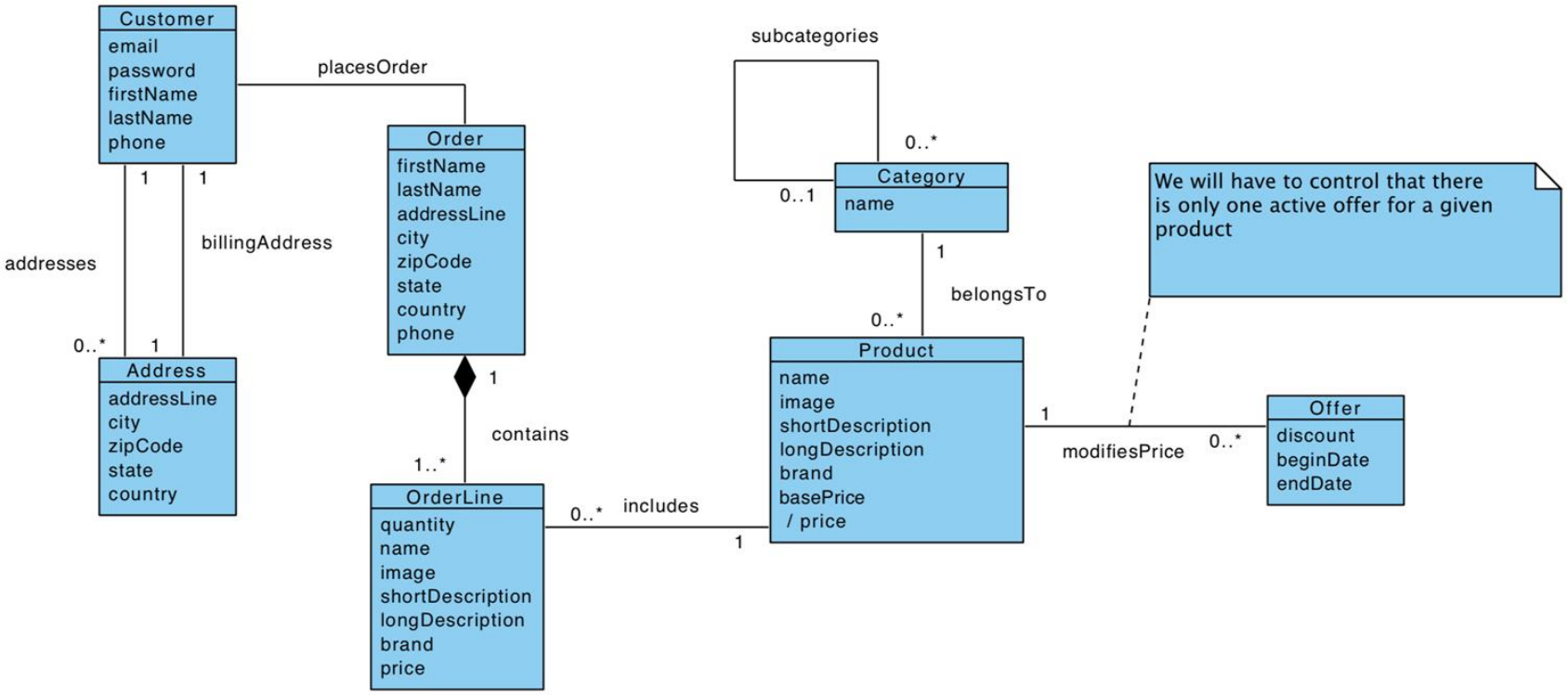
GET /products	Obtener todos
POST /products	Crear un producto
GET /products/{id}	Obtener un producto
PUT /products/{id}	Modificar un producto
DELETE /products/{id}	Borrar un producto

Arquitectura de microservicios

- Los servicios deben estar totalmente desacoplados entre sí
  - Mayor agilidad de desarrollo
  - Mayor escalabilidad
  - Posibilidad de implementar los servicios con distintas tecnologías

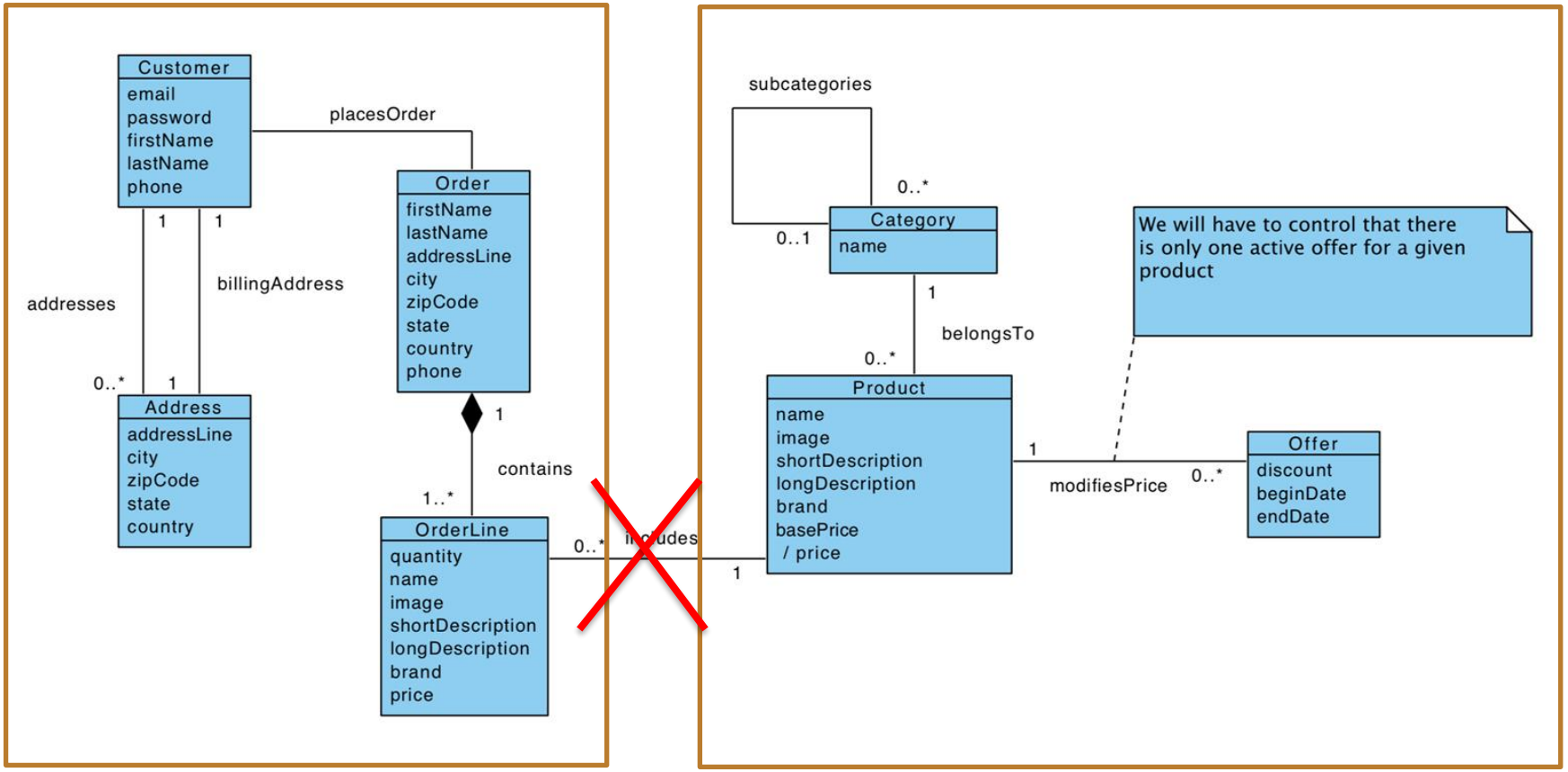
Arquitectura de microservicios

- Ejemplo



Arquitectura de microservicios

- Dividimos en dos servicios

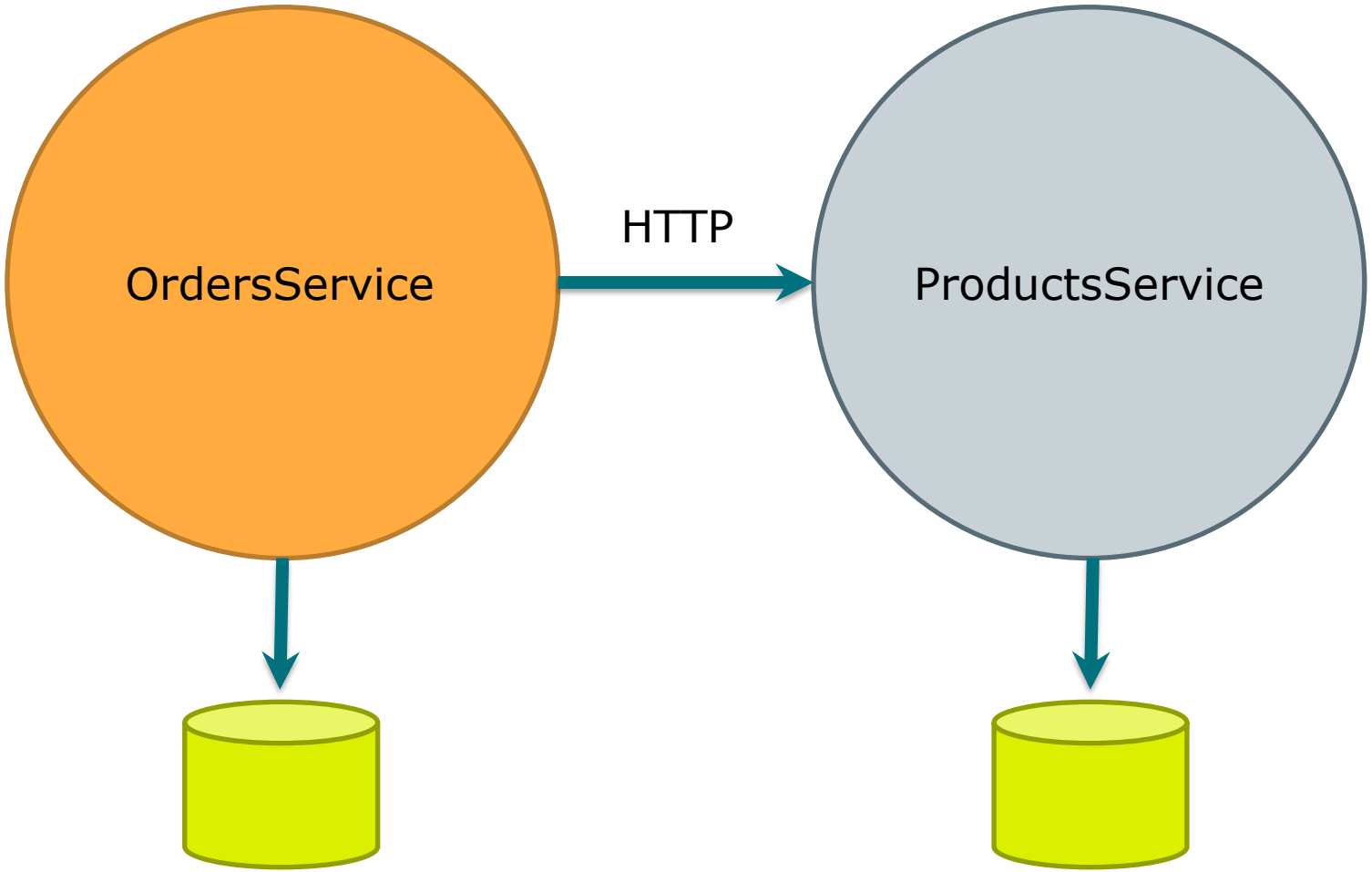


OrdersService

ProductsService

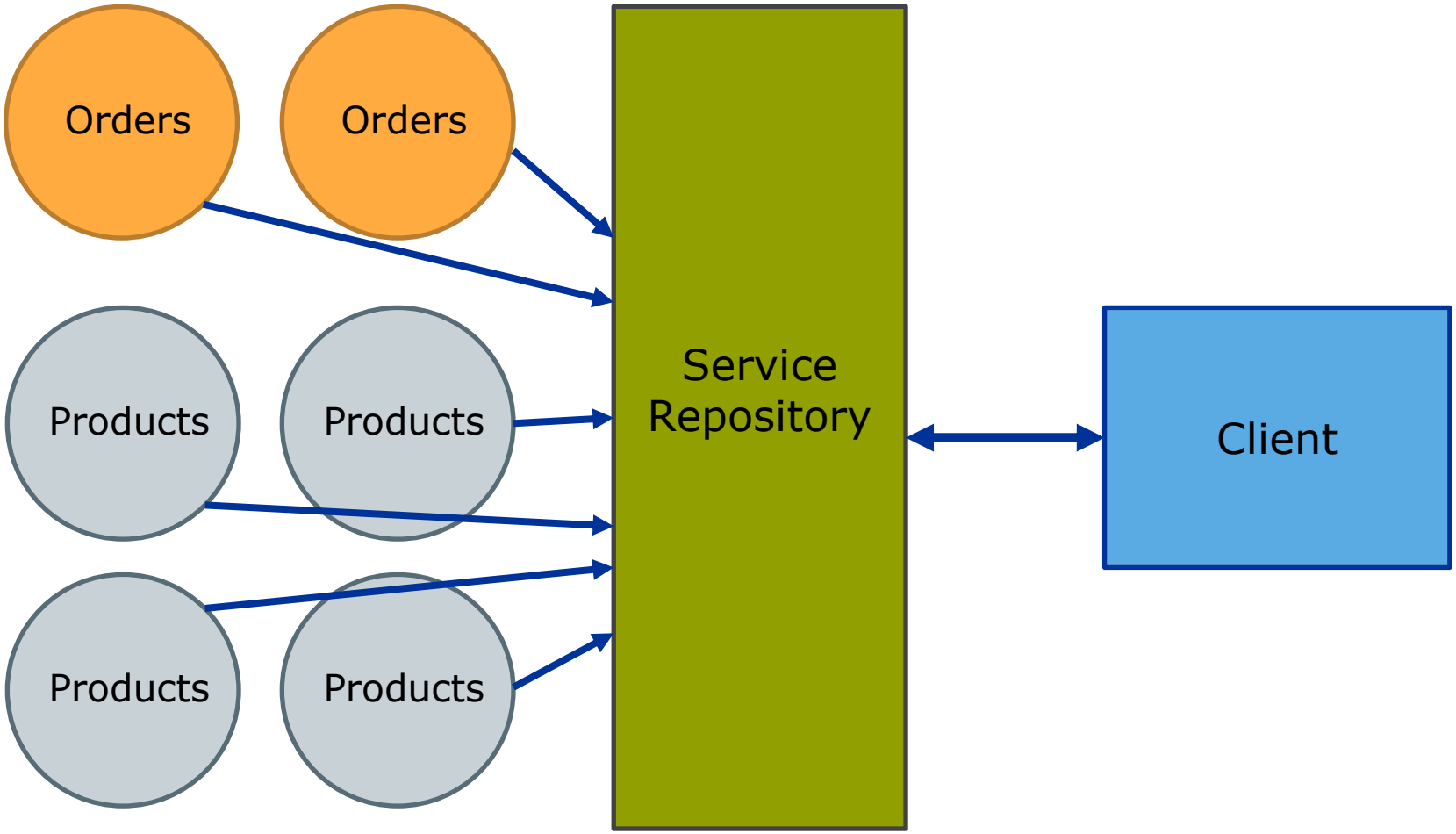
Arquitectura de microservicios

- Dividimos en dos servicios



Arquitectura de microservicios

- Los servicios pueden escalar individualmente



Arquitectura de microservicios

- Adecuada para el desarrollo de aplicaciones y servicios web
- Análisis del patrón
  - **Agilidad**: alta, los cambios afectan a componentes aislados
  - **Despliegue**: sencillo, favorece la integración continua
  - **Pruebas**: sencillo, debido a la independencia de los servicios
  - **Rendimiento**: bajo, debido a la naturaleza distribuida
  - **Escalabilidad**: alta, permite escalar los servicios por separado
  - **Desarrollo**: fácil, la independencia de los servicios reduce la necesidad de coordinación. El uso de protocolos de comunicación estándar facilita el desarrollo

Patrones de distribución

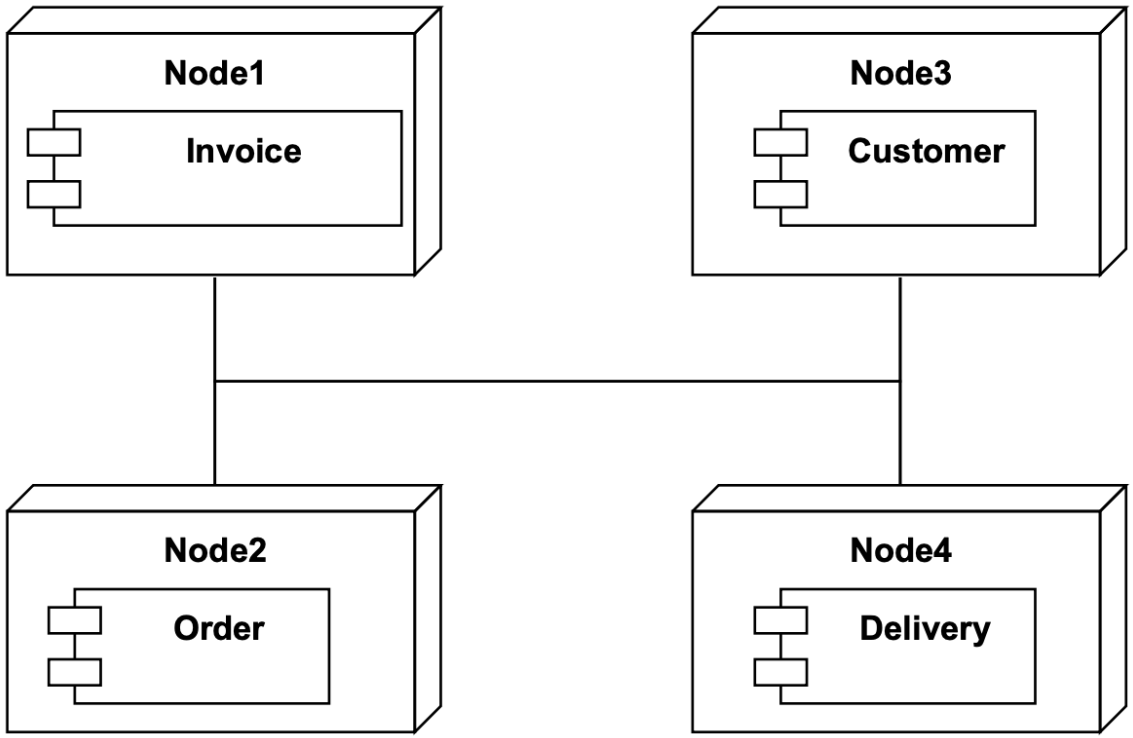
Patrones arquitecturales

Introducción

- Un **sistema distribuido** es aquél que se ejecuta repartido en distintos procesos o máquinas.
- Realizar un **diseño distribuido de objetos** implica ubicar los objetos del sistema en los diferentes nodos donde se ejecutarán.

Introducción

¿Cuál es el problema de este diseño?



Aplicación distribuida con diferentes componentes en distintas máquinas [Fowler, 2002]

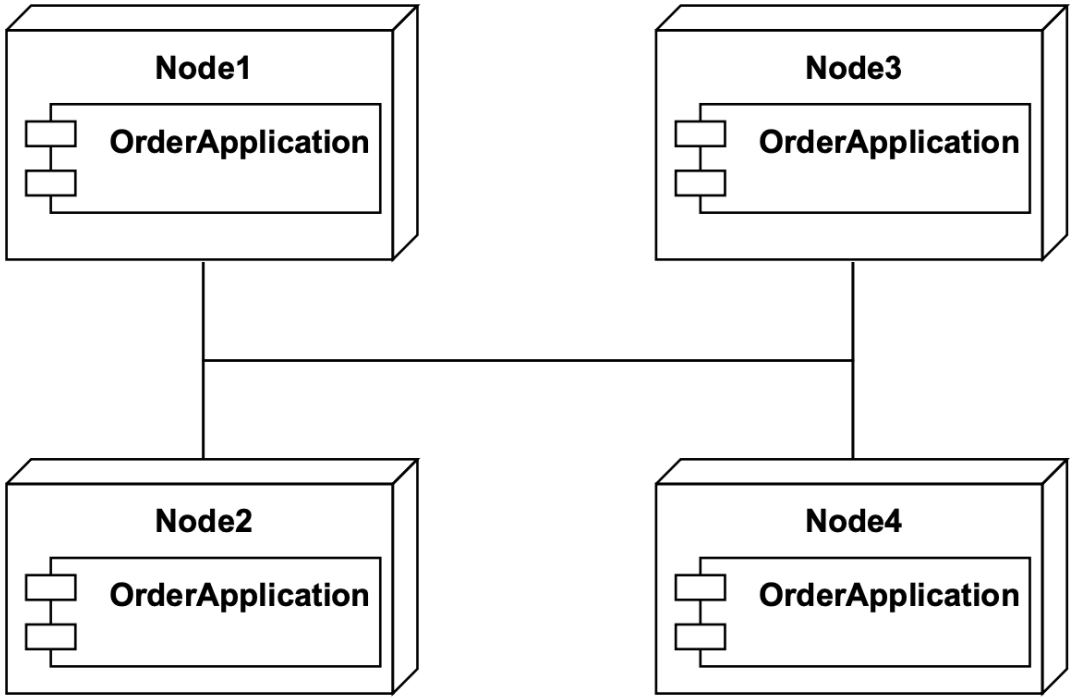
Introducción

- Al distribuir objetos tenemos que tener en cuenta el rendimiento del sistema.
- La comunicación entre procesos o entre sistemas remotos implica una disminución del rendimiento.

MÁS RÁPIDO	Llamadas locales
↓	Llamadas entre procesos
MÁS LENTO	Llamadas remotas

Clustering

- **Mejora:** poner varias copias de la aplicación completa en distintos nodos (***clustering***)



[Fowler, 2002]

Primera ley del diseño de objetos distribuidos:

“¡No distribuyas los objetos!”

Estrategias de distribución

Desafortunadamente, hay algunas situaciones en las que no se puede evitar la distribución de objetos:

- Sistemas cliente-servidor.
- Servidor de aplicación y base de datos (comunicación SQL).
- Servidor web y servidor de aplicación.
- Sistemas de terceros que necesitan funcionar en su propio proceso.
- **Restricciones impuestas por las características de cada proyecto.**

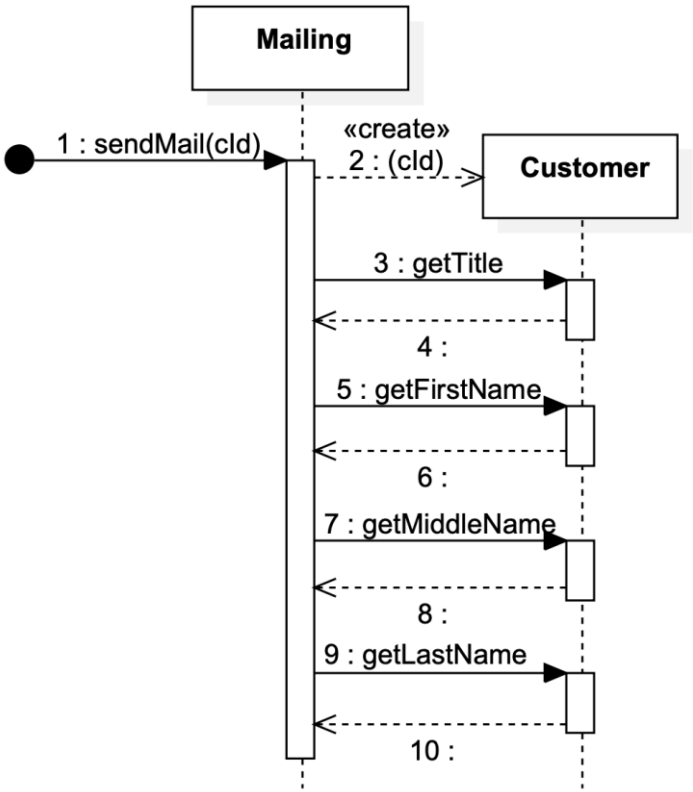
Estrategias de distribución

Qué hacer cuando los objetos deben estar distribuidos:

- Usar **interfaces de grano fino** para los objetos locales (como de costumbre), permite hacer un mejor diseño orientado a objetos.
- Usar **interfaces de grano grueso** para acceder a los objetos remotos para disminuir en la medida de lo posible la pérdida de rendimiento que implica hacer llamadas remotas.

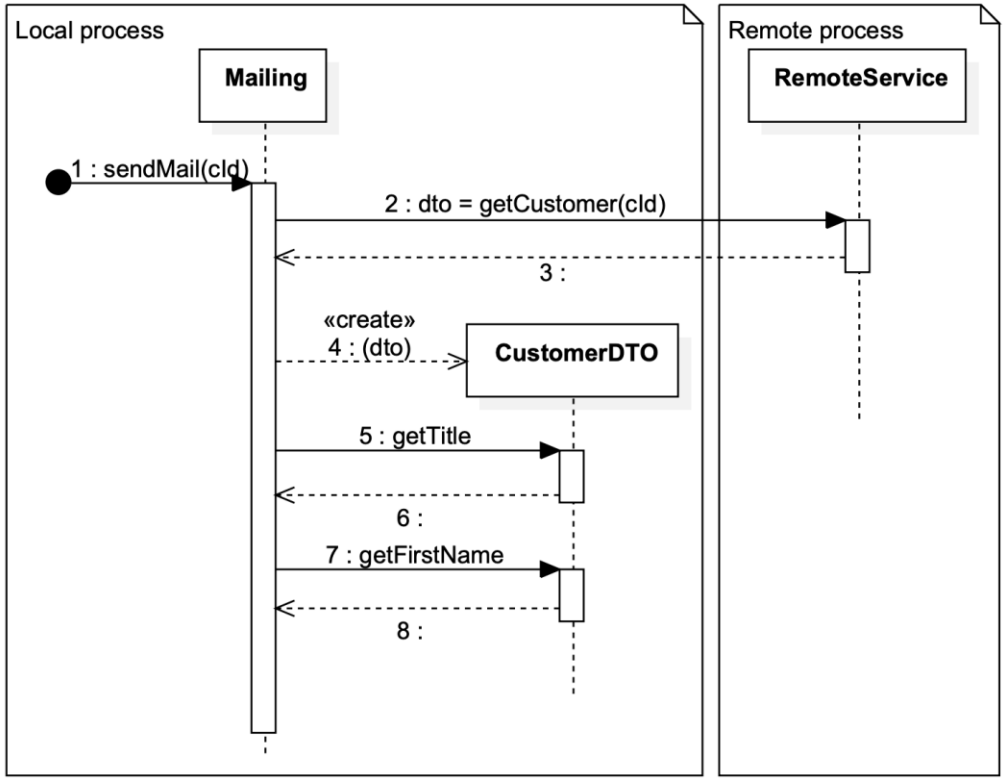
Estrategias de distribución

INTERFAZ DE GRANO FINO



El cliente solicita los datos uno a uno.

INTERFAZ DE GRANO GRUESO



El cliente solicita todos los datos en una sola llamada al procedimiento remoto.

Estrategias de distribución

Patrones de distribución involucrados:

- Remote Facade
- Data Transfer Object

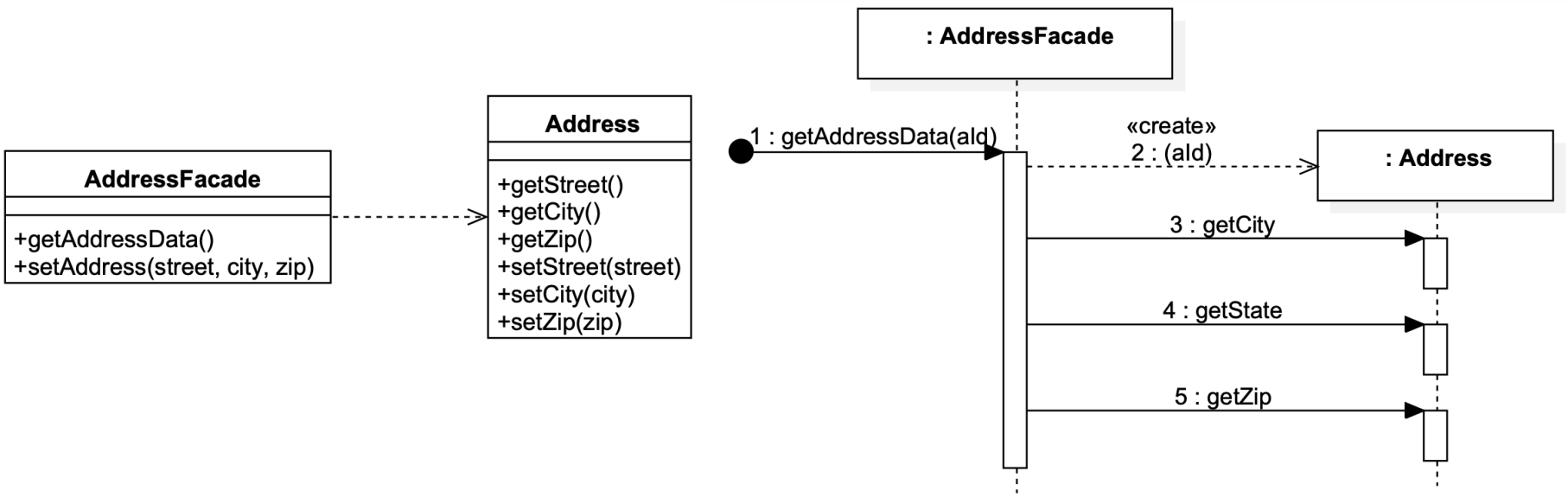
Patrón Remote Facade

Patrones de distribución

Patrón Remote Facade

Remote Facade

Provee una fachada de grano grueso para acceder a objetos de grano fino, para mejorar el rendimiento a través de la red.



[Fowler, 2002]

Patrón Remote Facade

- Una **Fachada Remota** reemplaza todos los métodos get() y set() con un único método para acceder a todas las propiedades del objeto.
- Provee un interfaz de grano grueso sin modificar los objetos de dominio.
- **¡Las Fachadas Remotas no contienen lógica de negocio!**
- Se puede usar una única Fachada Remota para acceder a varios objetos de dominio.
- Las Fachadas Remotas pueden implementarse con estado (p.ej. SOAP) o sin estado (p.ej. REST).

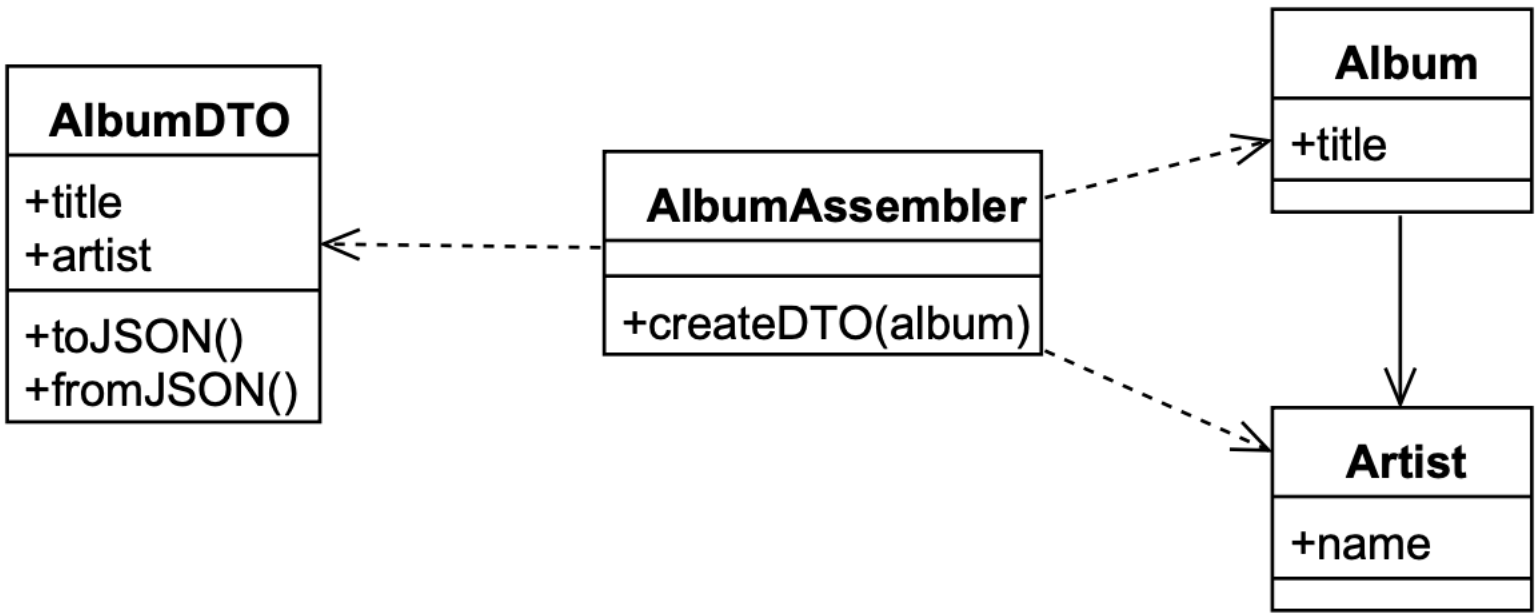
Patrón Data Transfer Object

Patrones de distribución

Patrón Data Transfer Object

Data Transfer Object

Un objeto que transporta datos entre procesos para reducir el número de llamadas.



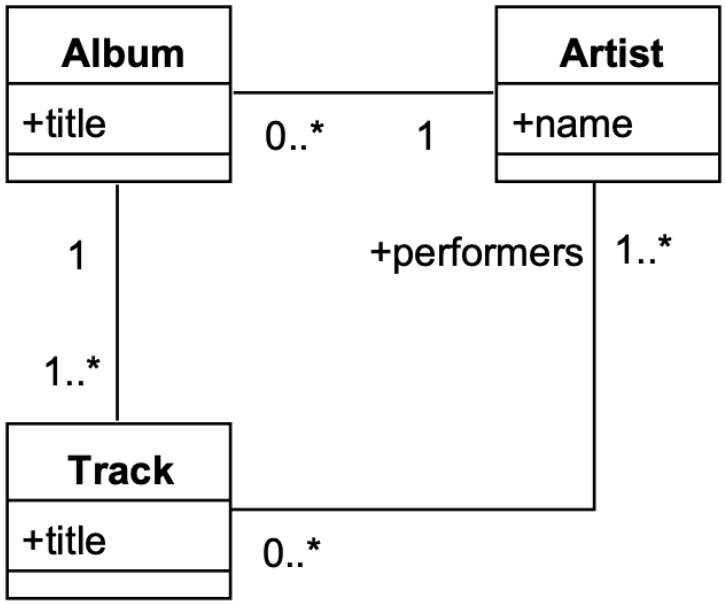
[Fowler, 2002]

Patrón Data Transfer Object

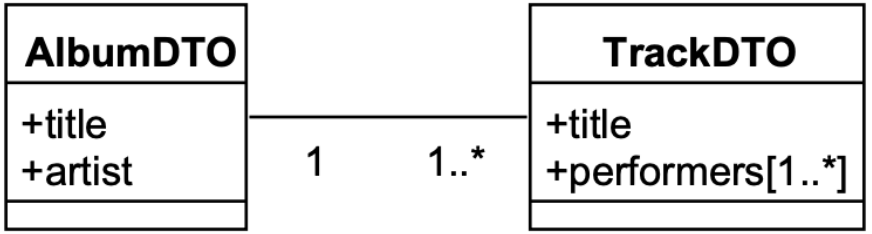
- **Contiene múltiples datos** para transportarlos en una única llamada remota.
- Debe ser **serializable** para poder transferir los datos sobre la conexión:
  - **Formato binario:** es más compacto pero más sensible a errores (p.ej. clientes no actualizados).
  - **Formato textual:** (p.ej. JSON) necesita más ancho de banda pero es más robusto ante los cambios.

Patrón Data Transfer Object

Los objetos DTO normalmente contienen datos de varios objetos de dominio



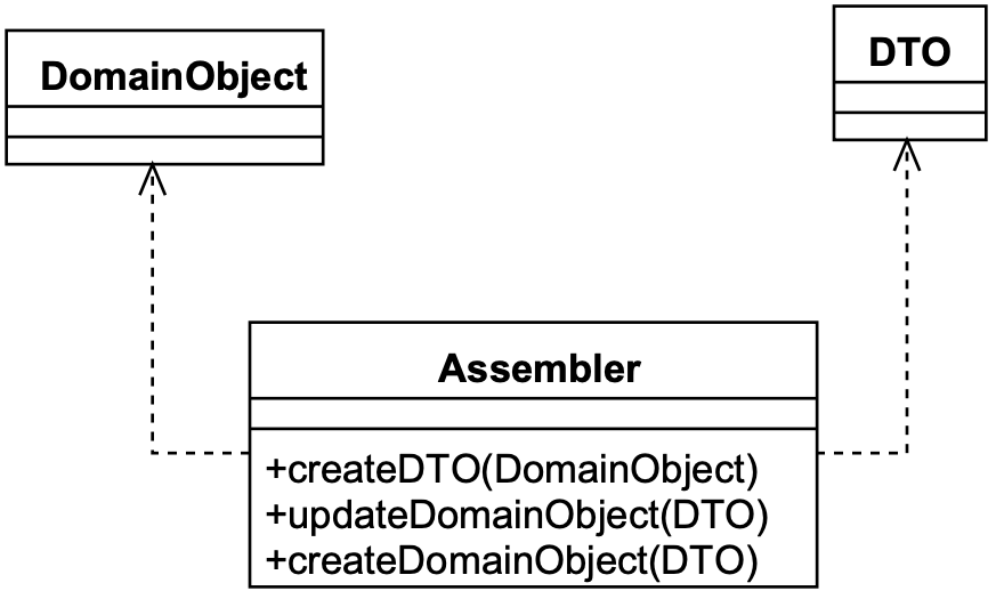
Modelo de dominio



Objetos DTO con información agregada

Patrón Data Transfer Object

- Los DTO y los objetos de dominio deben estar desacoplados, ya que **los DTO deben desplegarse en los dos lados** del sistema distribuido, pero los objetos de dominio no.
- Se puede usar un objeto ensamblador para construir objetos DTO a partir de los objetos de dominio.



¿Preguntas?

Bibliografía

- [Richards2015] Software Architecture Patterns. Mark Richards. O'Reilly, 2015