Arquitectura de Software y patrones arquitecturales

Diseño de Sistemas Software

Carlos Pérez
Santiago Meliá
Cristina Cachero

¿Qué es la Arquitectura de Software?

- Estructura de alto nivel que determina cómo los distintos componentes y módulos software se integran para construir la solución a un problema.
  [Hammer2014]

- Organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y el ambiente y los principios que orientan su diseño y evolución.
  [IEEE 1471]

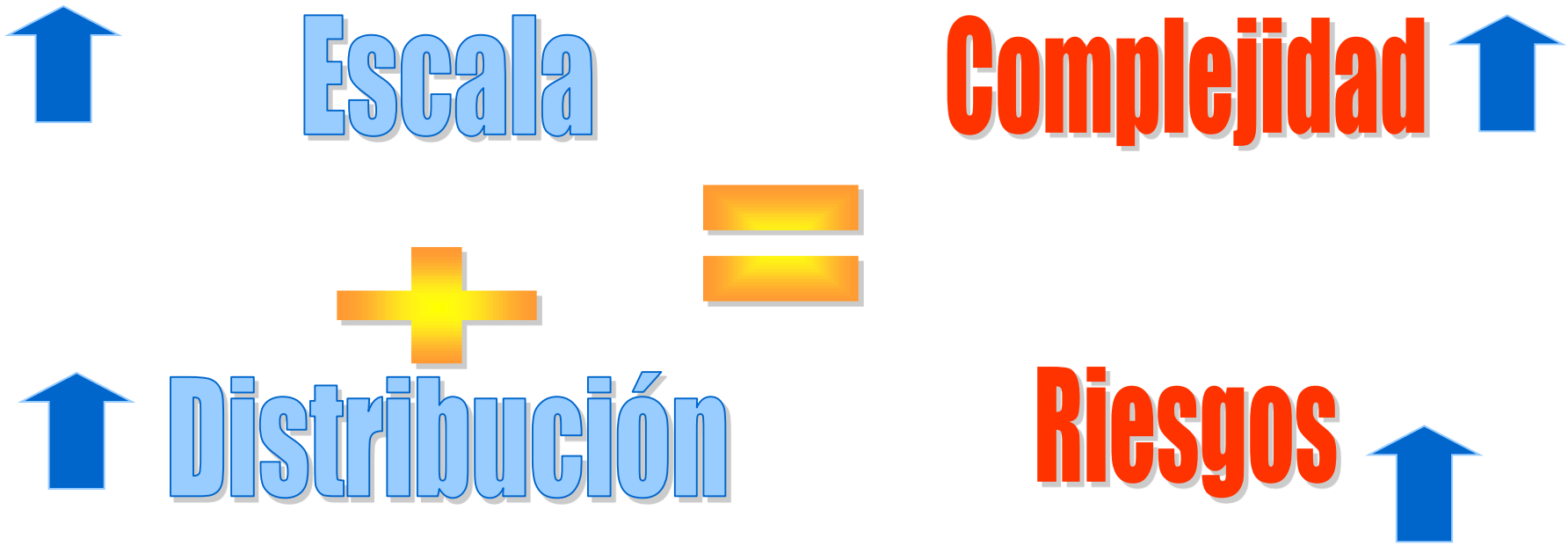
Decisiones arquitecturales

- Lógicas (decisiones independientes del hardware y software utilizado):
  - p.ej. definición de interfaces
- Físicas (decisiones dependientes del hardware y software utilizado):
  - p.ej. definición de capa de acceso a datos para acceso a BD relacional mediante ODBC, distribución de un sistema en distintas máquinas, etc.

¿Por qué es importante definir una arquitectura?

Premisa

- Las arquitecturas existen, independientemente de si han sido definidas formalmente o no.
- Dos factores primarios en la ingeniería de software que han incrementado la importancia de la arquitectura:

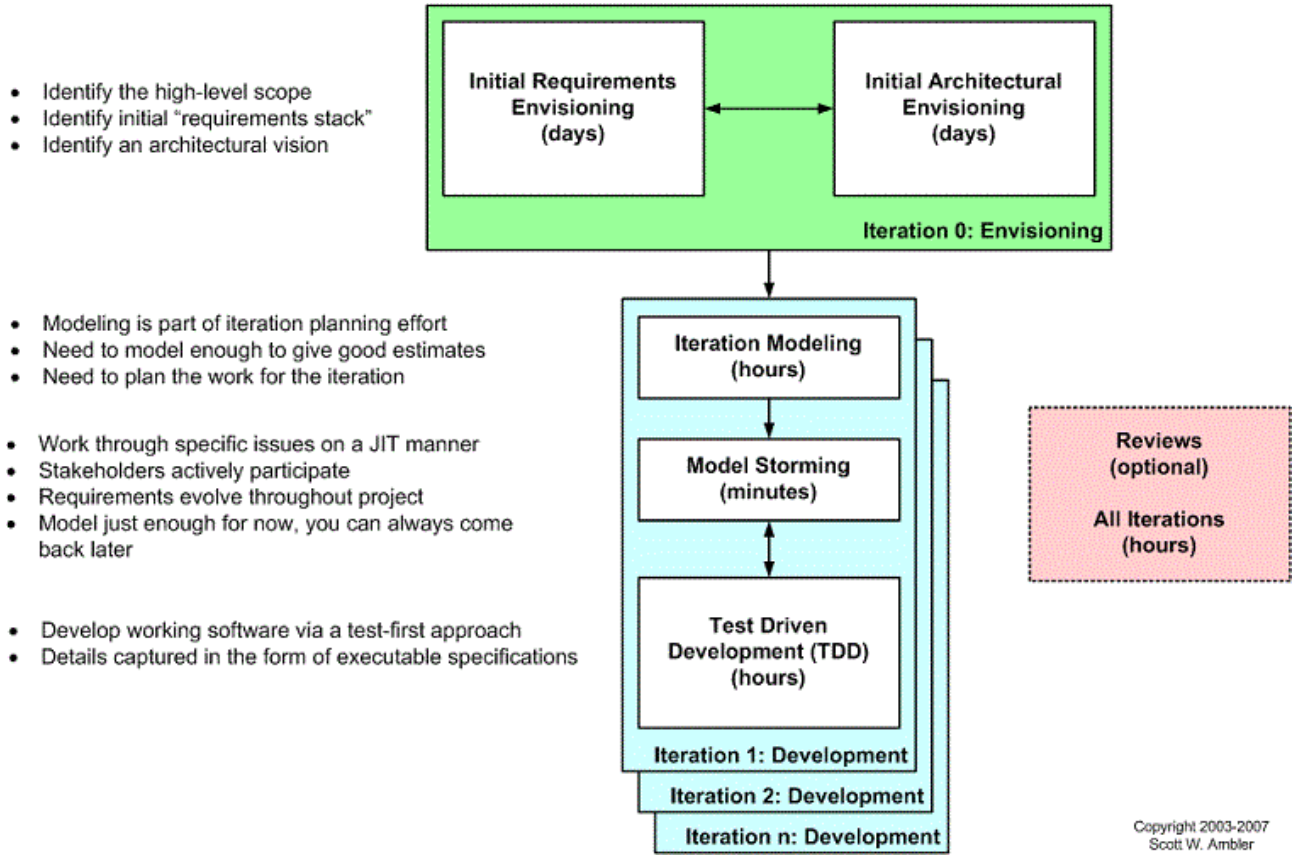


¿Por qué es importante definir una arquitectura?

- Necesitamos definir una arquitectura software para...
  - Comprender el sistema
    - Los sistemas de software pueden ser grandes y complejos, y deben acomodarse a requisitos a menudo en conflicto. Una arquitectura proporciona un plano conveniente del sistema que debe ser producido. Abstrae muchos detalles de implementación, pero posiciona los elementos que deben cubrir los requisitos funcionales
  - Organizar el desarrollo
    - Ayuda a separar las distintas preocupaciones para que cada persona sólo se ocupe de su tarea. También identifica cómo se relacionan las distintas tareas, de manera que los puntos en que interseccionan están bien documentados y claramente especificados
  - Promover la reutilización a nivel de subsistema
    - Ayuda a identificar sistemas y subsistemas críticos

¿Por qué es importante definir una arquitectura?

- Promover el desarrollo continuo
  - Una buena arquitectura ayuda a controlar la evolución del sistema a lo largo del tiempo. Proporciona una visión general del diseño actual del sistema, y por tanto ayuda a que futuros diseñadores no malinterpreten o ignoren parte del diseño.



Consecuencias de no definir una arquitectura

- Alto acoplamiento entre componentes
- Los cambios son difíciles sin comprender completamente el funcionamiento interno
- No se puede prever el comportamiento ante el despliegue y la necesidad de escalar

¿Por qué es importante definir una arquitectura?

Ejemplo

- Una tienda virtual que vende juguetes no puede manejar la carga de usuarios por su alta demanda y sale de línea:
  - Dos semanas antes de navidad
  - Después de una campaña de marketing de 20 Millones de USD.
- La expansión de capacidad tomará 6 semanas.

- **¿Cuál es vuestro diagnóstico respecto a la situación?**
  - ¿Los problemas que tiene esta aplicación son funcionales?
  - ¿Están asociados a su arquitectura, diseño o implementación?

Cualidades de una buena arquitectura

- Cualidades deseables
  - Agilidad para extender y mantener el sistema
  - Rapidez y facilidad de despliegue de la aplicación en producción
  - Facilidad para realizar pruebas
  - Rendimiento: tiempo de respuesta desde el punto de vista del usuario
  - Escalabilidad: horizontal (más servidores) o vertical (servidores más potentes)
  - Facilidad de desarrollo
- Es prácticamente imposible favorecer todos estos aspectos a la vez

Estilos arquitecturales y patrones

- El estilo arquitectural determina **categorías de patrones**

Table 2-1	Architectural Styles
Architectural Style	***Patterns***
From Mud to Structure	Layers (Chapter 9), Pipes and Filters (Chapter 10), Blackboard (Chapter 11)
Distributed Systems	Broker (Chapter 12)
Interactive Systems	Model-View-Controller (Chapter 13), Presentation-Abstraction-Control (Chapter 14)
Adaptable Systems	Microkernel (Chapter 15), Reflection (Chapter 16)

Patrones POSA

- Estilos de Flujo de Datos
  - Tubería y filtros
- Estilos Centrados en Datos
  - Arquitecturas de Pizarra o Repositorio
- Estilos de Llamada y Retorno
  - Model-View-Controller (MVC)
  - Arquitecturas en Capas
  - Arquitecturas Orientadas a Objetos
  - Arquitecturas Basadas en Componentes
- Estilos de Código Móvil
  - Arquitectura de Máquinas Virtuales
- Estilos heterogéneos
  - Sistemas de control de procesos
  - Arquitecturas Basadas en Atributos
- Estilos Peer-to-Peer
  - Arquitecturas Basadas en Eventos
  - Arquitecturas Orientadas a Servicios
  - Arquitecturas Basadas en Recursos

Patrones Arquitecturales y Frameworks

- Framework: Es un subsistema de software parcialmente construido, de propósito general para un tipo específico de problema, el cual debe ser instanciado para resolver un problema particular.
- Características
  - Define la arquitectura para una familia de subsistemas
  - Provee bloques básicos de construcción y adaptadores.

- Típicamente un framework se construye a partir de patrones de diseño.
- El framework impone patrones de diseño para su uso.

Patrones Arquitecturales vs Frameworks

Ejemplos de Frameworks

Tecnológicos

- Struts y Java Server Faces (JSF) para Interfaces Web.
- Log4J para auditoría técnica o negocios en app.
- Toplink, Hibernate, ROME para mapeos objeto-relacionales.
- Enterprise Java Beans para construcción de servicios de negocios.
- Framework de patrones de diseño para J2EE de Sun Microsystems.
- Grinder como framework para pruebas de rendimiento.

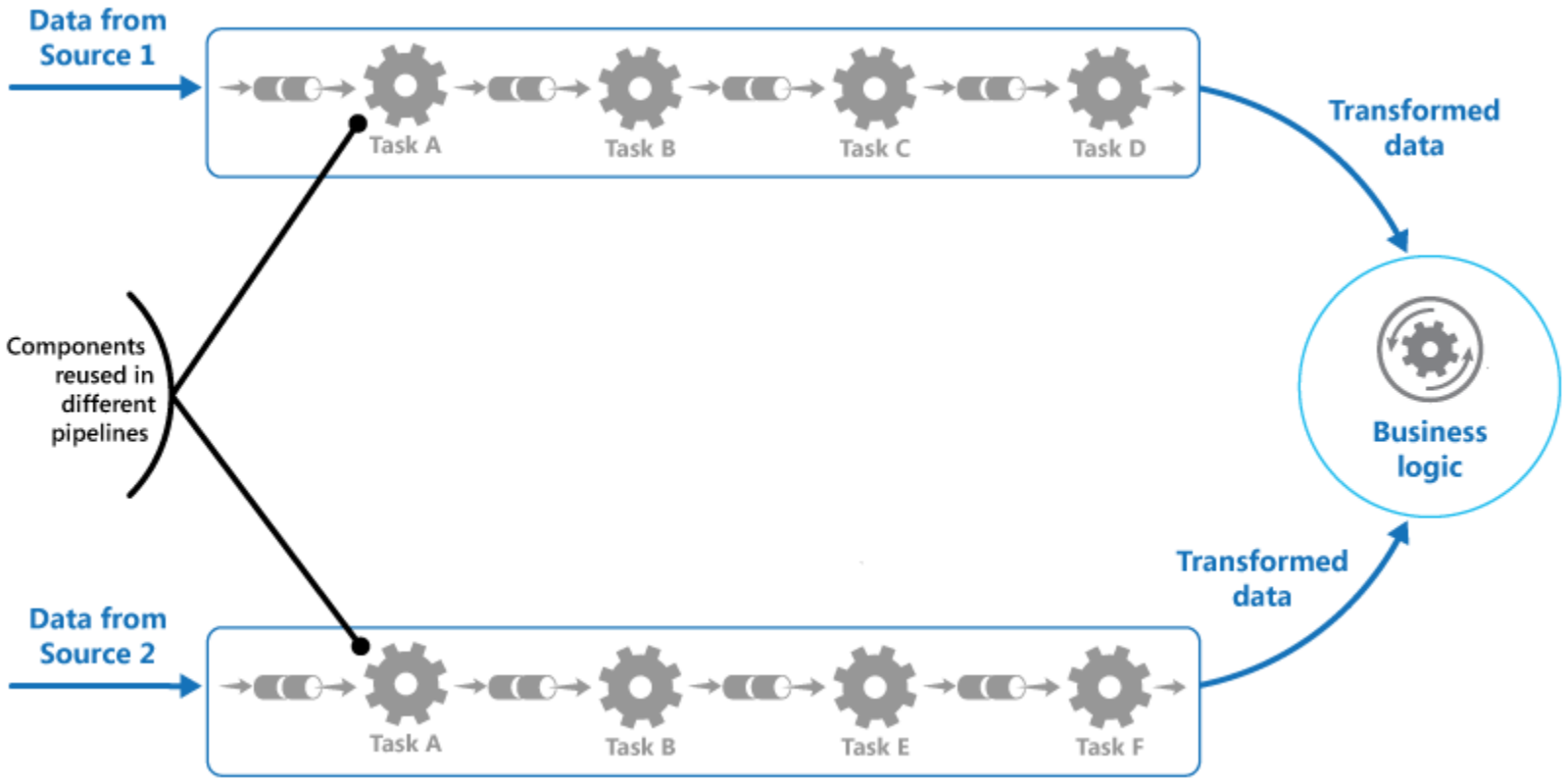
Conceptuales

- El Cubo para definición de arquitecturas.
- eTom para empresas de telecomunicaciones.
- RUP como metodología para definición de procesos de desarrollo.

Arquitectura tuberías y filtros

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

- Ejemplo: Apertium

  https://www.apertium.org

- Conclusiones
  - Facilita el diagnóstico
  - Permite añadir fácilmente nuevos filtros entre dos módulos
  - Desarrollo de aplicaciones derivadas mediante la reutilización de módulos (interNOSTRUM → Apertium)

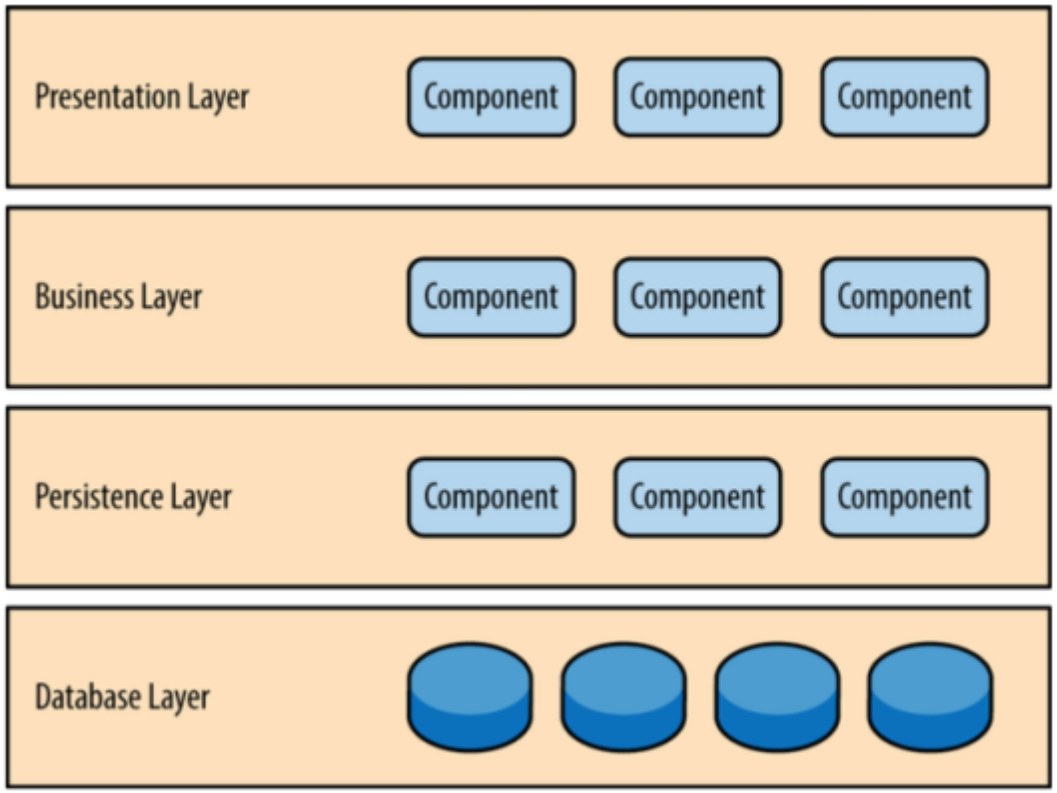
Arquitectura tuberías y filtros

- Análisis del patrón
  - Agilidad: alta, debido a la modularidad
  - Despliegue: lento, normalmente se trata de aplicaciones que se ejecutan en las máquinas de los usuarios finales
  - Pruebas: fácil, se pueden probar los filtros de forma independiente
  - Rendimiento: alto, aunque depende de la complejidad de los filtros
  - Escalabilidad: baja, distribuir los filtros en distintos nodos añade complejidad
  - Desarrollo: fácil, si se usan los mecanismos de comunicación del sistema operativo (tuberías)

Arquitectura en capas

Arquitectura en capas

- Los componentes se organizan en capas horizontales



Arquitectura en capas

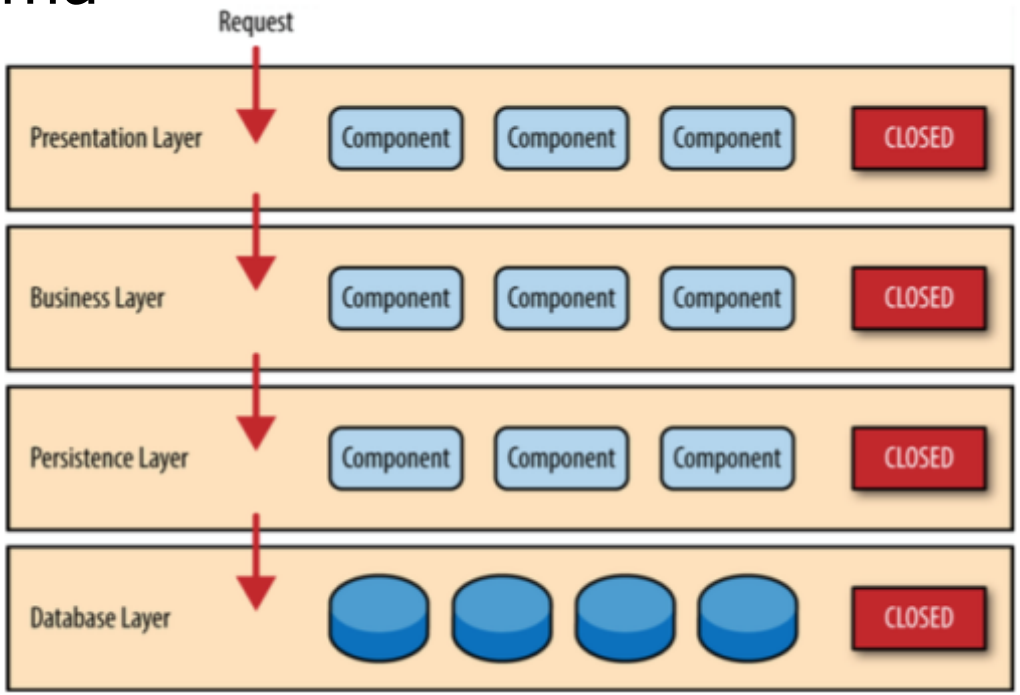
- Cada capa se encarga de una tarea específica, abstrayendo los detalles a las demás capas
- Arquitectura típica
  - **Presentación**: interacción con el usuario
  - **Lógica de negocio**: ejecución de las reglas de negocio
  - **Persistencia**: comunicación con la base de datos
  - **Base de datos**: almacenamiento persistente
- **La capa de lógica de negocio y la capa de persistencia no deben depender nunca de la presentación**

Arquitectura en capas

- La separación en capas es una organización lógica (a nivel de software), no implica necesariamente una separación física en distintos nodos
- Cuando la separación es física hablamos de arquitecturas 2-tiers (cliente/servidor), 3-tiers, n-tiers

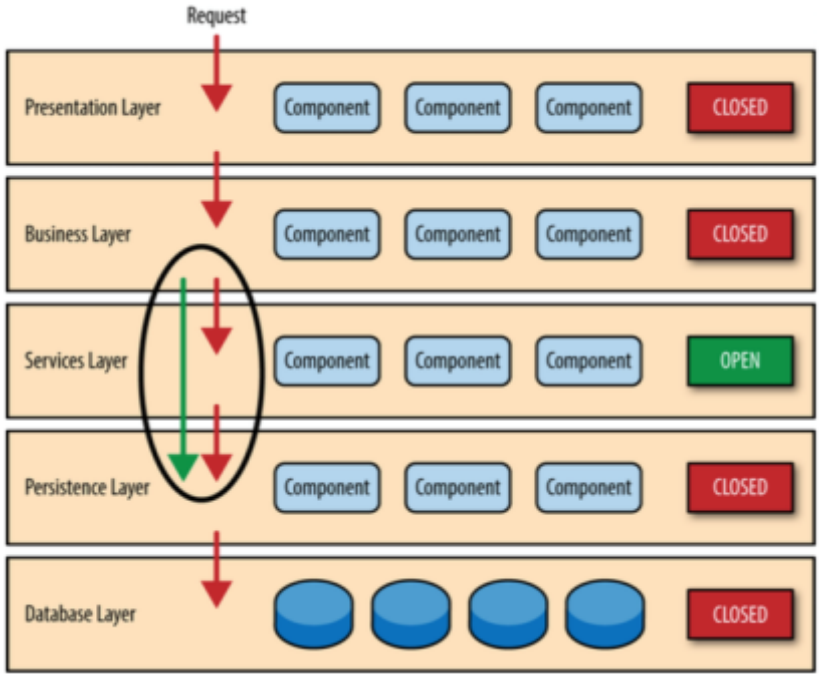
Arquitectura en capas

- **Arquitectura cerrada**: la comunicación va de una capa a la inmediatamente inferior
- Disminuye el impacto de los cambios y la complejidad del sistema



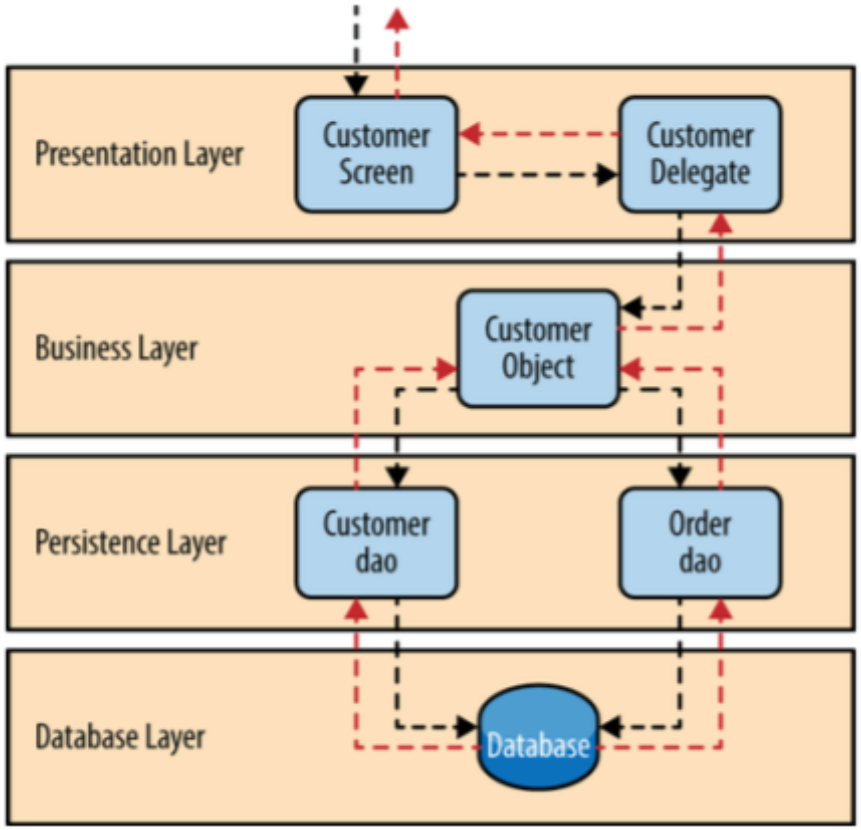
Arquitectura en capas

- **Arquitectura abierta**: las capas superiores se pueden comunicar con todas o algunas de las inferiores
- Necesario cuando la mayoría de las peticiones se limitan a pasar de una capa a otra sin ninguna lógica de negocio asociada



Arquitectura en capas

- Ejemplo: aplicación de gestión de clientes

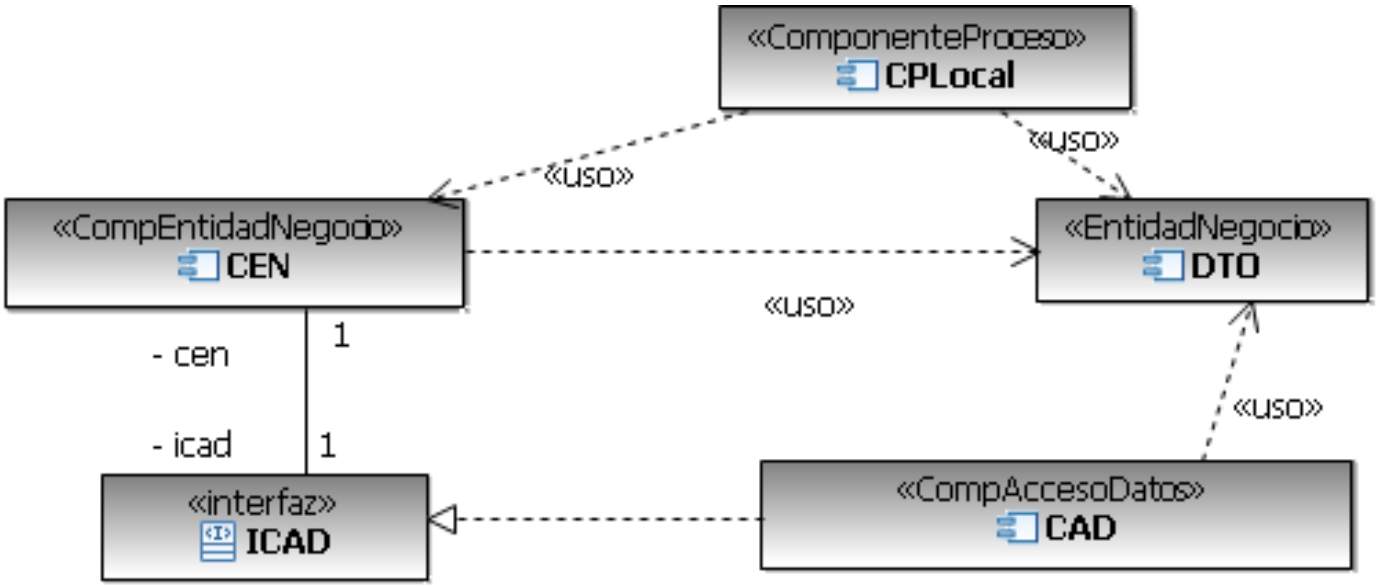


Arquitectura en capas

- Es la arquitectura más adecuada para la mayoría de aplicaciones
- Análisis del patrón
  - Agilidad: baja, los cambios normalmente afectan a varias capas y son lentos
  - Despliegue: lento, un cambio en un componente puede requerir el despliegue de toda la aplicación
  - Pruebas: el aislamiento entre capas facilita las pruebas
  - Rendimiento: generalmente bajo, por el sobrecoste de la comunicación entre capas
  - Escalabilidad: baja, si las capas no se distribuyen entre varios nodos
  - Desarrollo: fácil, es un patrón muy conocido y extendido. Permite repartir el trabajo de cada capa a distintos expertos (BBDD, diseño de GUI, etc.)

Arquitectura en capas

- Patrones relacionados



Arquitectura en capas

- **Objetos de dominio** (*Business Objects*, BO): contienen la lógica de dominio
- Dos tipos
  - Contienen los atributos y métodos de las clases de dominio
  - Contienen sólo los métodos → implementados en OOH4RIA como **Componentes Entidad de Negocio (CEN)**

    Los atributos se mantienen por separado en otras clases.

Arquitectura en capas

- **Objetos de transferencia de datos** (*Data Transfer Object*, DTO): representan las clases de dominio, pero sólo almacenan atributos
- Se usan para almacenar información y ser transferidos entre distintas capas
- Implementados en OOH4RIA como **Entidades de Negocio (EN)**

Arquitectura en capas

- **Componentes de acceso a datos (CAD),** (*Data Access Object*, DAO): contienen la lógica propia del sistema de persistencia elegido
- Los CEN y CAD pertenecen a capas diferentes, por lo que para mantener la separación entre capas se puede usar el patrón **Inyección de dependencias** (Fowler, 2002)

Arquitectura en capas

- **Inyección de dependencias**: en lugar de instanciar directamente la clase de la que se depende, se recibe (inyecta) la dependencia (p.ej. como parámetro) desde una fuente externa
- Disminuye el acoplamiento y facilita las pruebas

CustomerCEN.cs

```
 */
public partial class CustomerCEN
{
private ICustomerCAD _ICustomerCAD;

public CustomerCEN()
{
        this._ICustomerCAD = new CustomerCAD ();
}

public CustomerCEN(ICustomerCAD _ICustomerCAD)
{
        this._ICustomerCAD = _ICustomerCAD;
}
```

CustomerCAD.cs

```
public partial class CustomerCAD : BasicCAD, ICustomerCAD
```

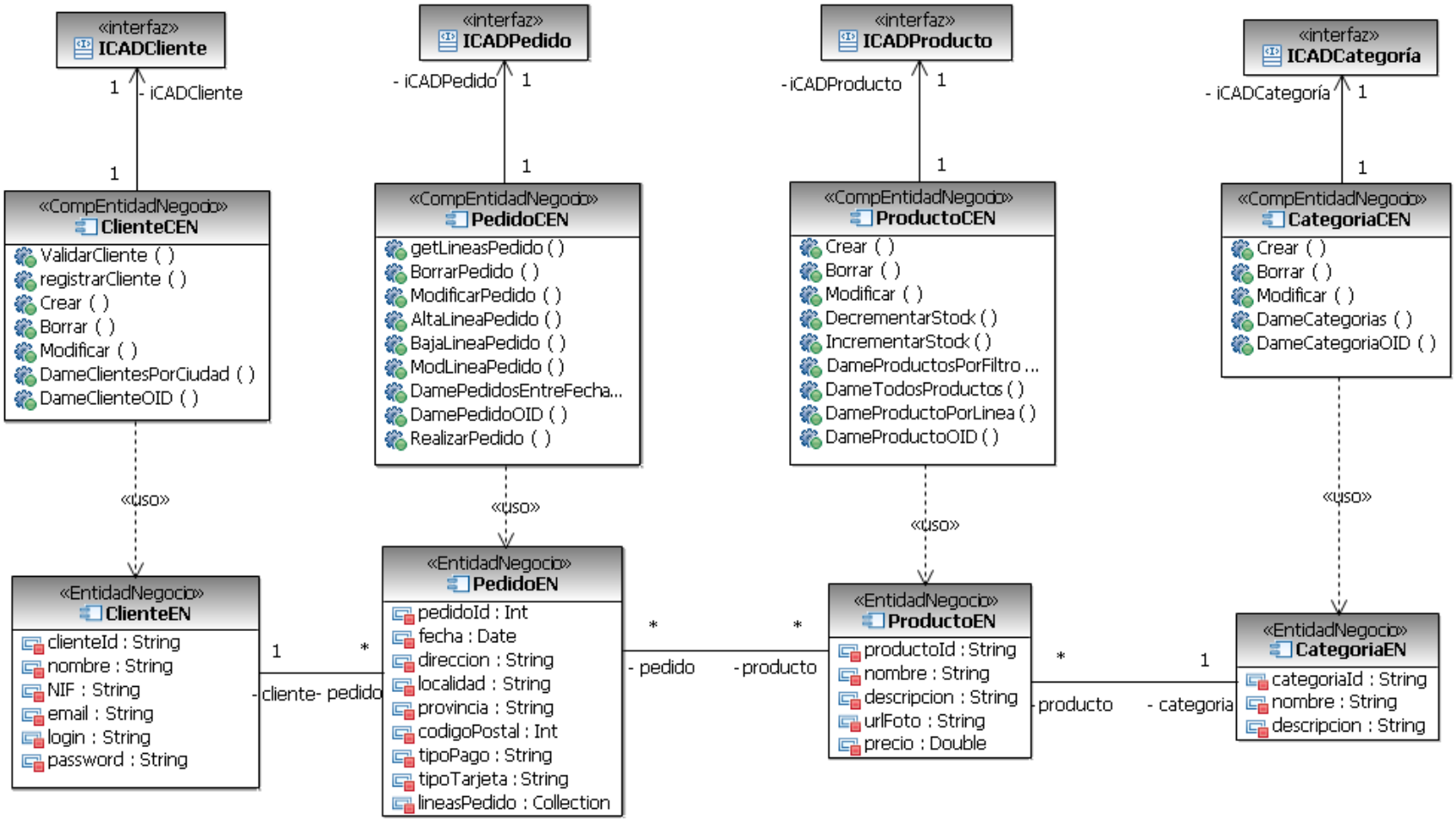
Código cliente

```
// Sin inyección de dependencias
CustomerCEN cen1 = new CustomerCEN();

// Inyección del objeto CAD
CustomerCEN cen2 = new CustomerCEN(new CustomerCAD());
CustomerCEN cen3 = new CustomerCEN(new MockCustomerCAD());
```

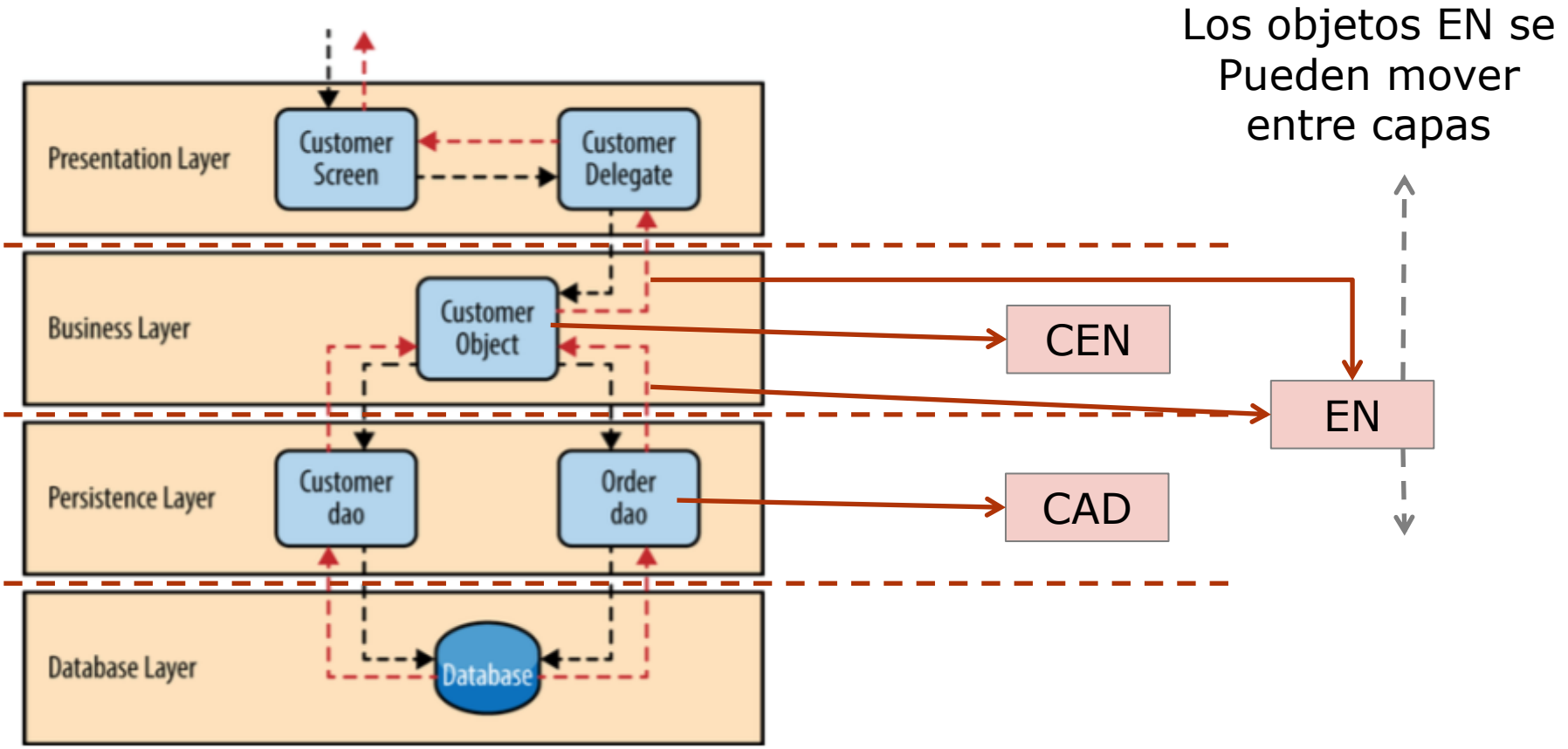
Arquitectura en capas

- Estructura del código generado por OOH4RIA



Arquitectura en capas

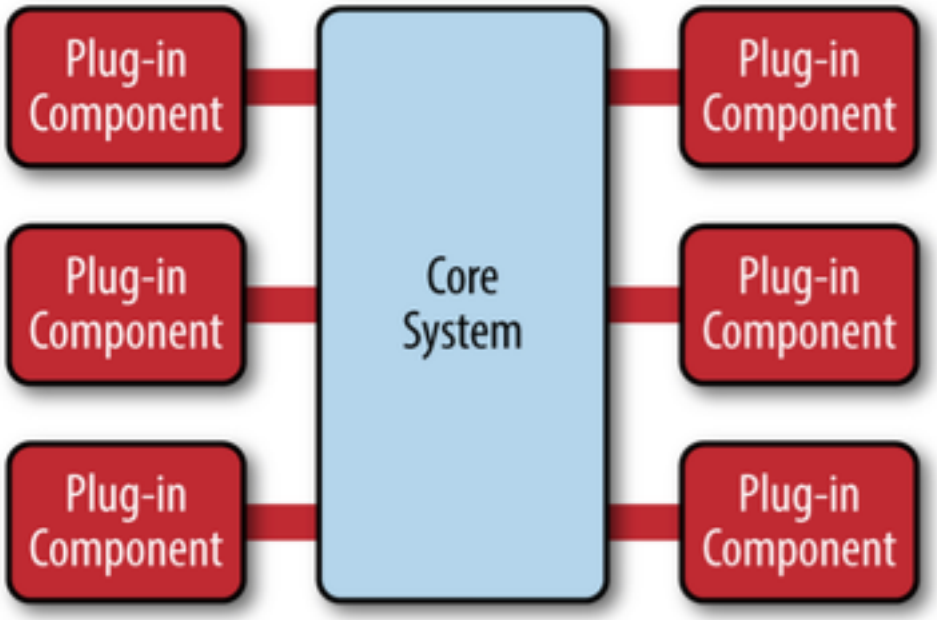
- Ubicación de los patrones en distintas capas



Arquitectura microkernel

Arquitectura microkernel

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

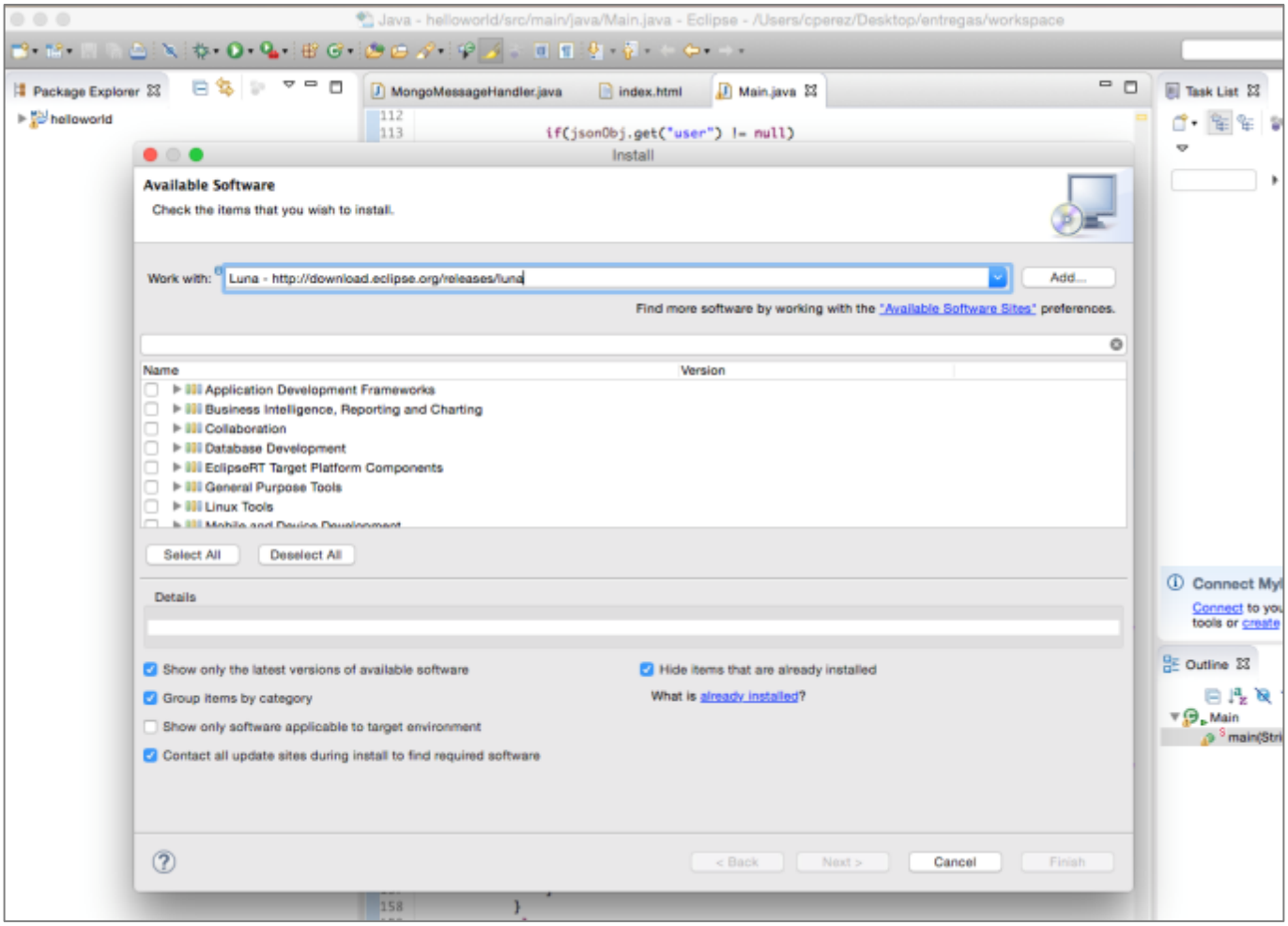


Arquitectura microkernel

- 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



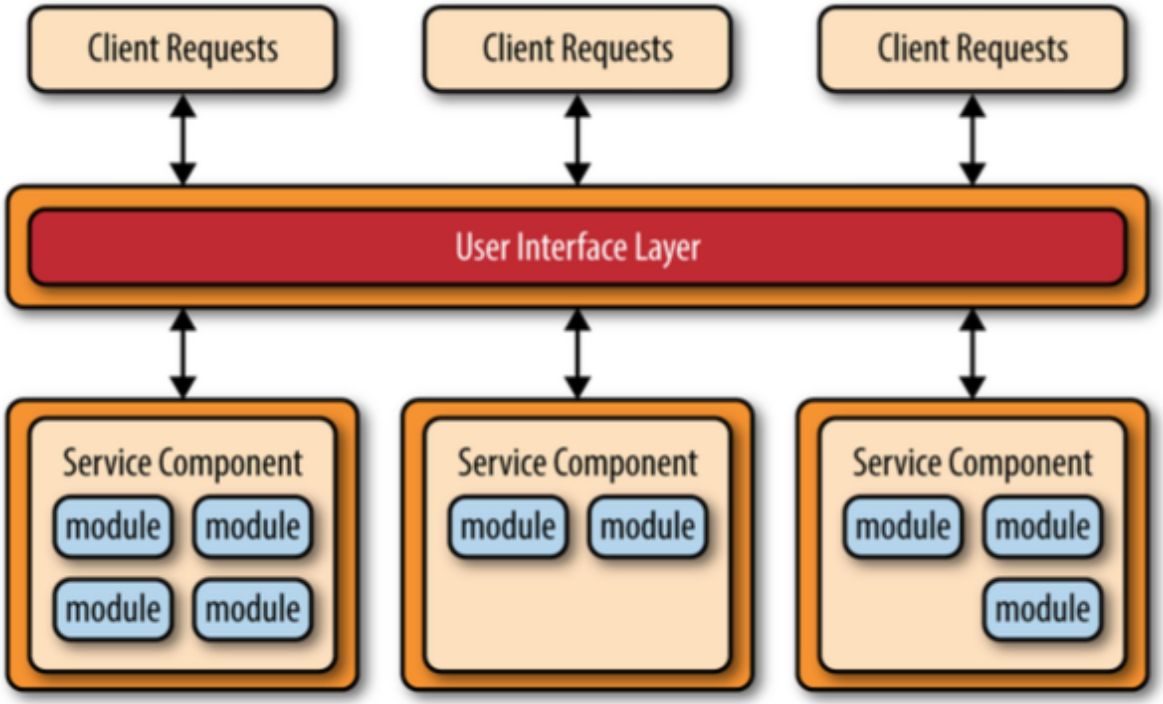
Arquitectura microkernel

- 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 de microservicios

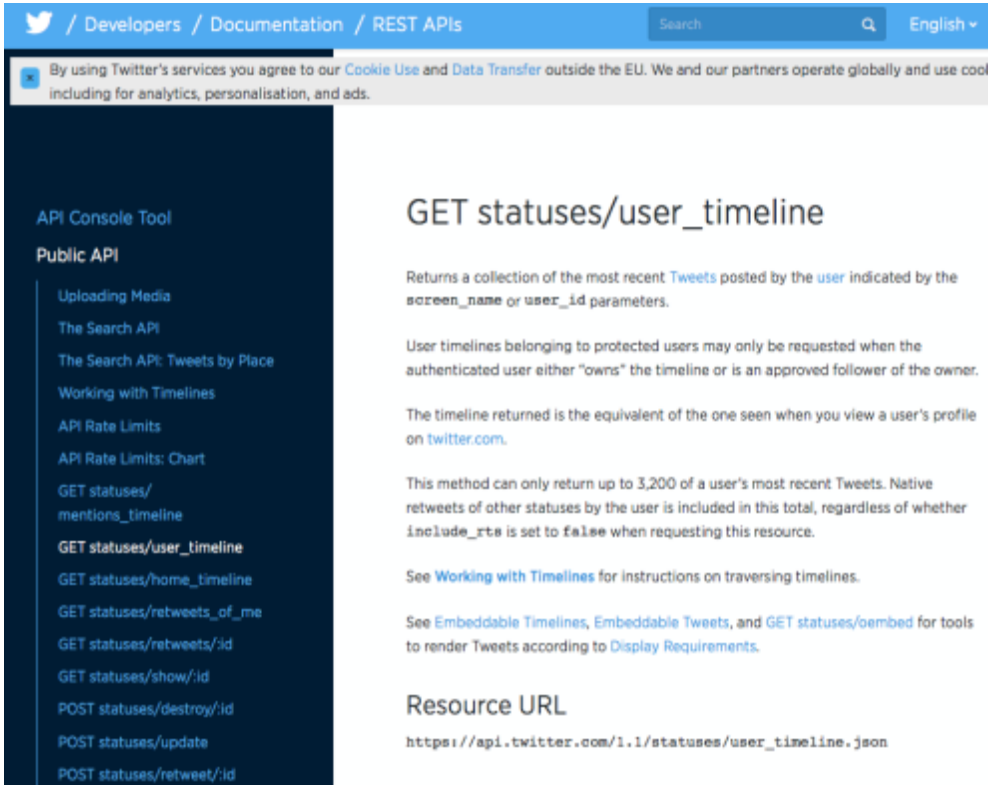
Arquitectura de microservicios

- Arquitectura distribuida, el cliente se comunica con los servicios mediante algún protocolo de acceso remoto (REST, SOAP, RMI, etc.)



Arquitectura de microservicios

- La implementación más frecuente utiliza HTTP como protocolo de comunicación, definiendo interfaces REST (*REpresentational State Transfer*) para acceder a los servicios



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

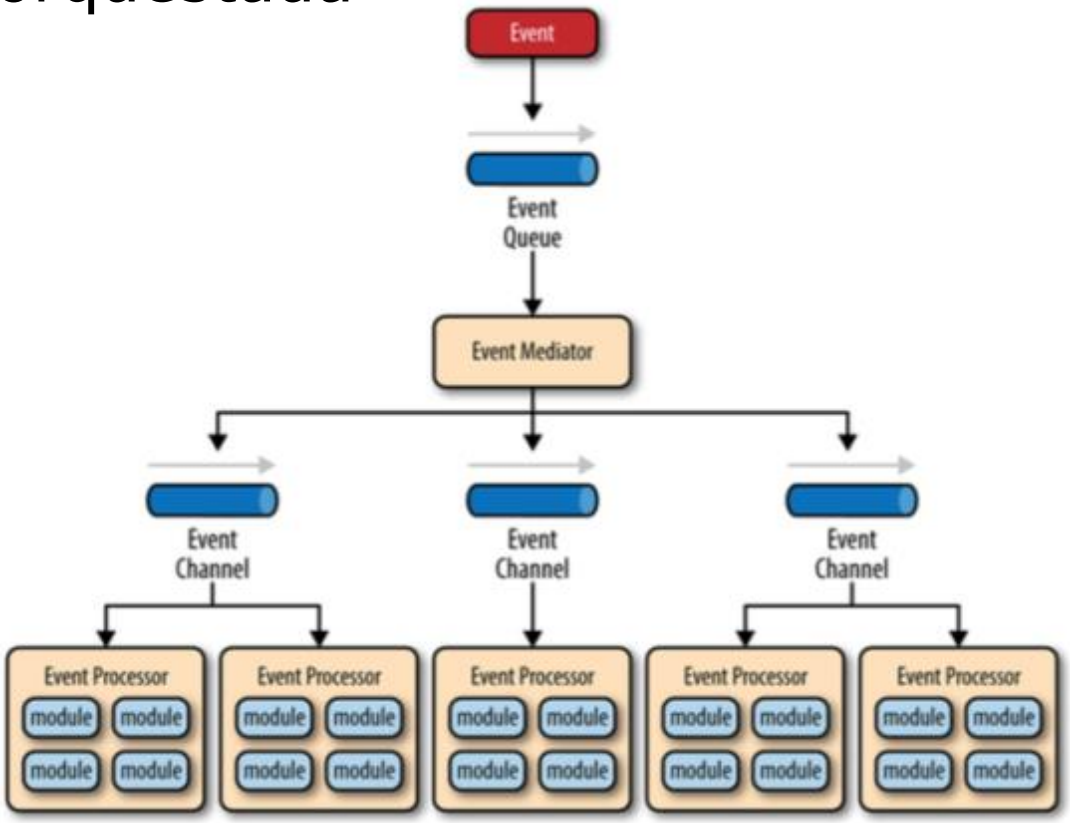
Arquitectura orientada a eventos

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
- Dos topologías alternativas
  - Mediador
  - *Broker*

Arquitectura orientada a eventos

- Topología **mediador**: el procesamiento de eventos implica varios pasos que deben ejecutarse de manera orquestada

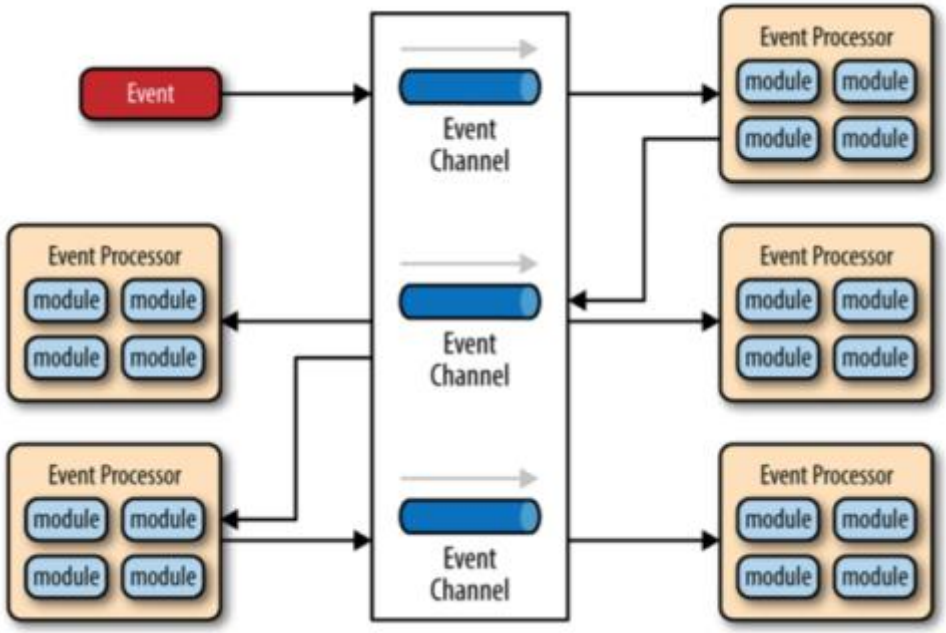


Arquitectura orientada a eventos

- Componentes
  - Procesadores de eventos
    - Contienen la lógica de negocio
    - Pequeñas unidades autocontenidas y altamente independientes del resto
  - Mediador
    - Recoge los eventos de inicio y los envía a los procesadores en el orden apropiado según el tipo de evento
    - Se puede implementar mediante soluciones *open source*, y definir usando lenguajes de definición de procesos (*business process execution language*, BPEL)

Arquitectura orientada a eventos

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

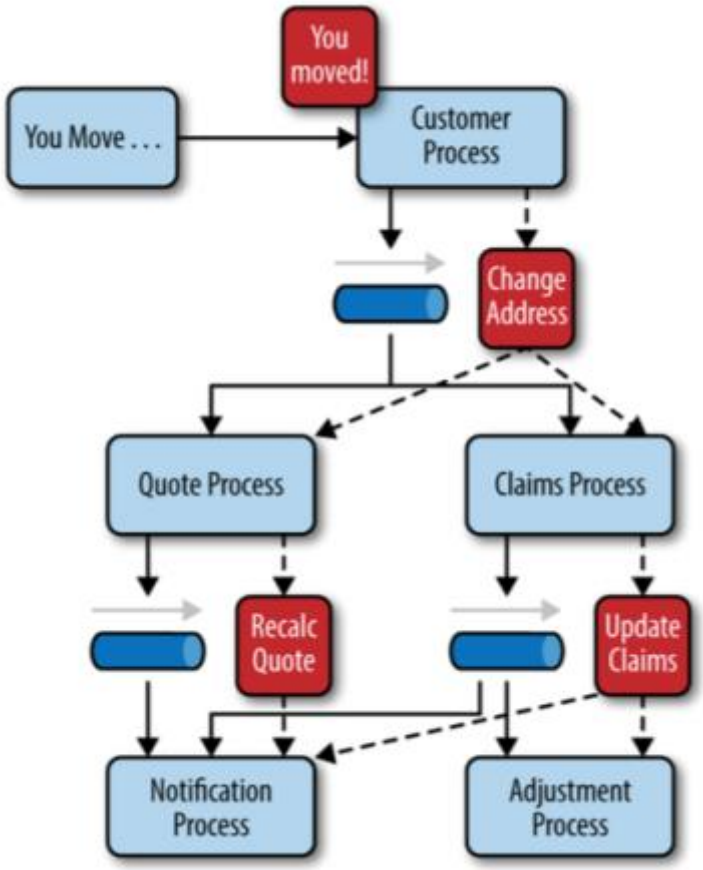


Arquitectura orientada a eventos

- Componentes
  - Procesadores de eventos
  - Broker: más ligero que el mediador, se encarga únicamente 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

- Es una arquitectura compleja, de naturaleza asíncrona y distribuida
- Análisis del patrón
  - Agilidad: alta, los cambios normalmente afectan a uno o pocos componentes
  - Despliegue: sencillo, debido al bajo acoplamiento. Más complicado en el caso del mediador, ya que se debe actualizar cada vez que hay un cambio en los procesadores de eventos
  - Pruebas: complicado, debido a la naturaleza asíncrona y la necesidad de herramientas especializadas para generar eventos
  - Rendimiento: alto, debido a la posibilidad de paralelizar la ejecución de componentes
  - Escalabilidad: alta, los componentes pueden escalar por separado
  - Desarrollo: difícil, la gestión de errores es compleja

Bibliografía

- [Richards2015] Software Architecture Patterns. Mark Richards. O'Reilly, 2015
- [Hammer2014] Pattern-Oriented Software Architecture for Dummies. Robert Hammer. Addison Wesley, 2014
- [Bass et al. 2003] Software Architecture in Practice. Len Bass, Paul Clemens and Rick Kazman. Addison Wesley, 2003
- [Buschman96] Buschmann, Frank et al.: Pattern Oriented Software Architecture, Volume 1: A System of Patterns, Willey & Sons, 1996.
- [Fowler03]Fowler, Martin: Patterns of Enterprise Application Architecture, Addison Wesley, 2003.