



Universidad de Oviedo



Modularidad



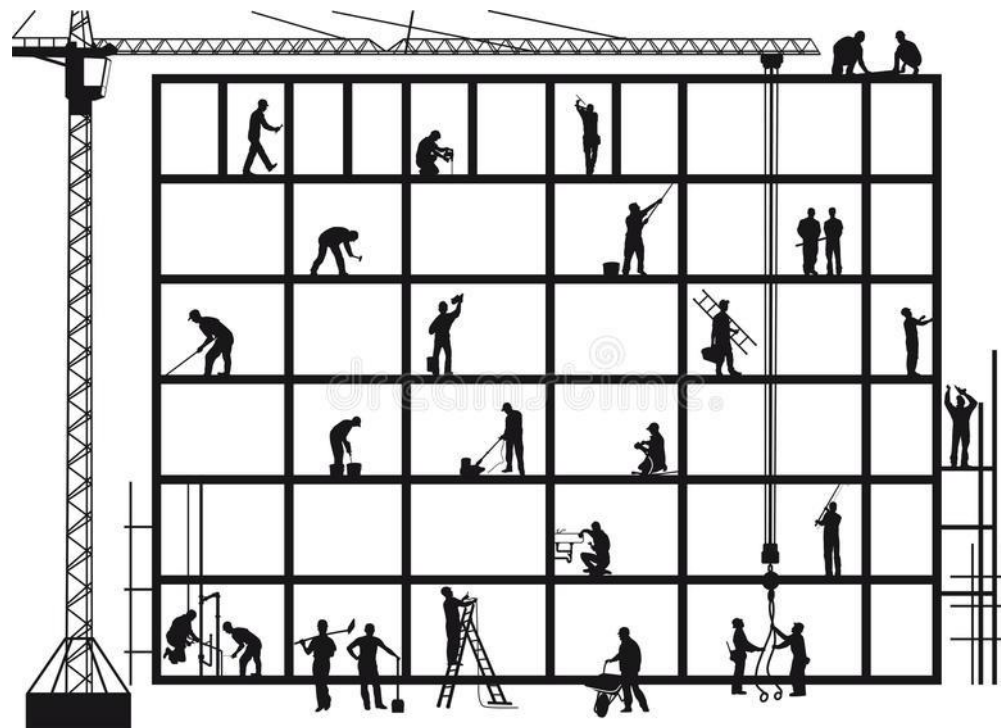
Curso 2019/2020

Jose Emilio Labra Gayo

Modularidad

Bloques de construcción

Descomposición modular: tiempo de construcción



Estilos de modularidad

Big Ball of Mud

Descomposición modular

Definiciones

Recomendaciones

Estilos de modularidad

Capas

Orientado a aspectos

Basados en dominio

Big Ball of Mud

Big Ball of Mud (Gran bola de lodo)

Descrito por Foote & Yoder, 1997

Elementos

Un montón de entidades entrelazadas entre sí

Restricciones

Ninguna



Big Ball of Mud

Atributos de calidad?

Time-to-market

Arranque rápido

Comenzar a desarrollar sin arquitectura

Resolver problemas bajo demanda

Coste

Solución barata a corto plazo

Adecuado para algunos problemas

"No todos los cobertizos necesitan columnas"



Big Ball of Mud

Problemas

Mantenimiento muy caro

Poca flexibilidad a partir de una etapa

Al inicio puede ser muy flexible

A partir de un punto, un cambio = dramático

Inercia

Cuando el sistema se convierte en *Big Ball of Mud* es difícil transformarlo en otra cosa

Pocos desarrolladores "*con prestigio*" saben dónde tocar

Los desarrolladores "*limpios*" huyen

Big Ball of Mud

Razones

Código de usar y tirar

Crecimiento improvisado

Necesidad de que siga funcionando

Reutilización mediante cortar/pegar

Código malo se reproduce en muchos sitios

Antipatrones

Malos olores (Bad smells)

Código/arquitectura

Descomposición modular

Módulo:

Pieza de software que ofrece conjunto de responsabilidades

Sentido en tiempo de desarrollo (no ejecución)

Separa interfaz del cuerpo

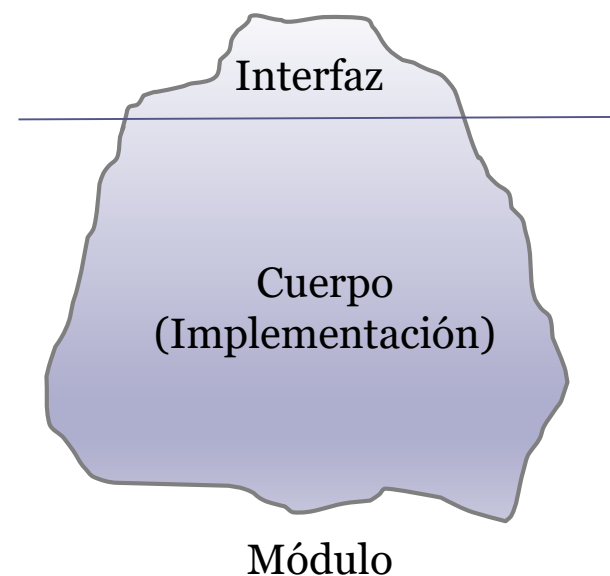
Interfaz

Describe qué es el módulo

Cómo utilizarlo \approx Contrato

Cuerpo

Cómo está implementado



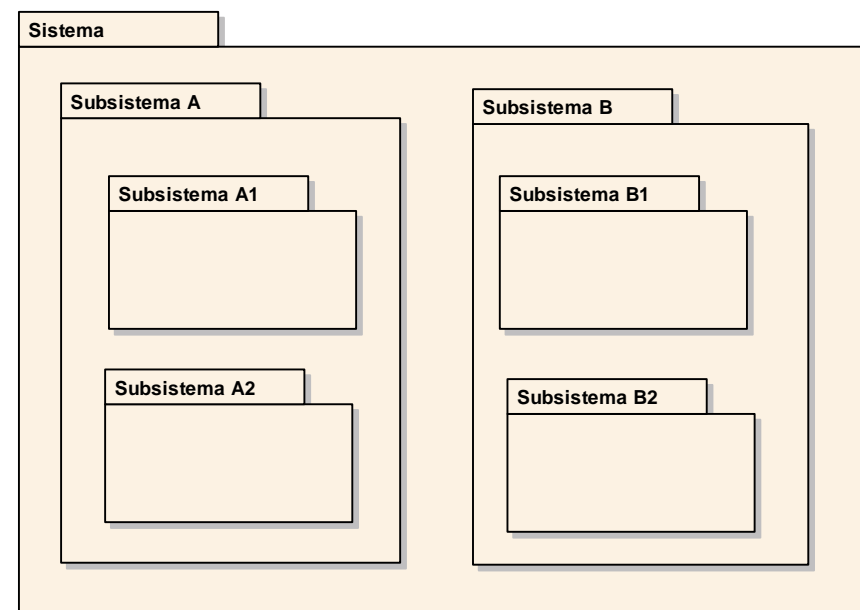
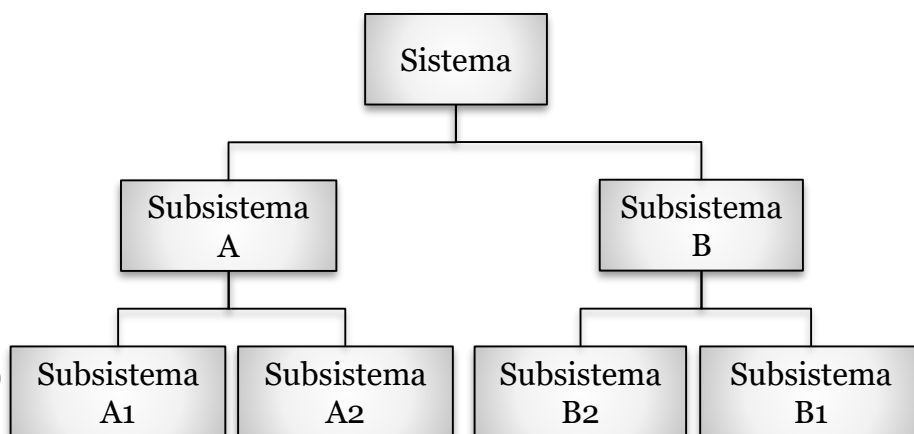
Descomposición modular

Restricciones

No puede haber ciclos

Un módulo sólo puede tener un padre

Varias representaciones



Atributos calidad modularidad

Comunicación

Permite comunicar aspecto general del sistema

Minimiza complejidad

Cada módulo expone sólo interfaz

Extensibilidad, mantenimiento

Facilita cambios y modificaciones

Funcionalidad localizada

Reusabilidad

Módulos que pueden usarse en otros contextos

Líneas de productos

Independencia

Desarrollo de módulos por diferentes equipos

Retos modularidad

Mala descomposición puede aumentar complejidad

Gestión de dependencias

Módulos de terceras partes pueden afectar evolución

Disposición del equipo

Descomposición puede afectar desarrollo y organización del equipo

Decisión: comprar vs desarrollar

Módulos COTS/FOSS

Recomendaciones modularidad

Alta cohesividad

Bajo acoplamiento

Ley de Conway

Ley de Postel

Principios SOLID

Ley de Demeter

Interfaces Fluidos

Principios cohesión/acoplamiento

Recomendaciones modularidad

Alta Cohesividad

Cohesividad = Coherencia de un módulo.

Cada módulo debe resolver una funcionalidad

Lema DRY (Don't Repeat Yourself)

La intención debe estar declarada en un único sitio

Granularidad

Módulos que puedan entregarse y reutilizarse

Cada módulo debe poder probarse por separado



Recomendaciones modularidad

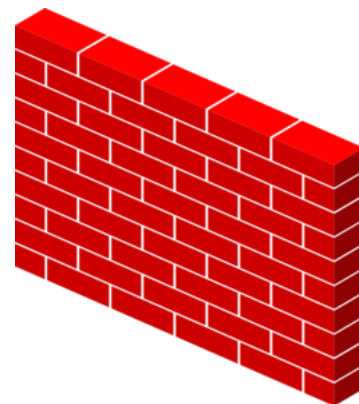
Acoplamiento bajo

Acoplamiento = Grado interacción entre módulos

Menor acoplamiento \Rightarrow Facilita modificabilidad

Despliegue independiente de unos módulos respecto a otros

Estabilidad frente a cambios de otros módulos



Ley de Conway

M. Conway, 1967

“Las organizaciones que diseñan sistemas...acaban produciendo sistemas que son copias de las estructuras de comunicación de dichas organizaciones”

Corolario:

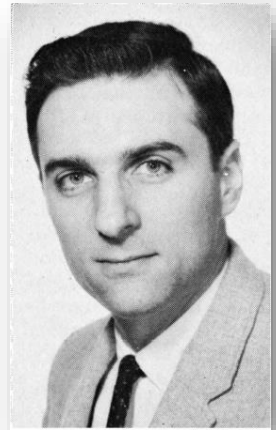
"La major estructura de un Sistema está influenciada por la estructura social de la organización"

Ejemplo:

Si hay 3 equipos (diseño, programación, base datos), el sistema tendrá 3 módulos de forma natural

Consejo:

Crear equipos después de la descomposición modular



Melvin Conway

Principio de robustez, ley de Postel

Ley de Postel (1980) definida para TCP/IP

"Sé liberal en lo que aceptas de otros y conservador en lo que envías"

Mejorar interoperabilidad

Enviar mensajes bien formados

Aceptar mensajes incorrectos

Aplicación al diseño de APIs

Procesar campos de interés ignorando el resto

Permitir que las APIs evolucionen después



Jon Postel

Recomendaciones modularidad

Principios SOLID

Pueden utilizarse para clases/módulos

SRP (Single Responsibility Principle)

OCP (Open-Closed Principle)

LSP (Liskov Substitution Principle)

ISP (Interface Segregation Principle)

DIP (Dependency Injection Principle)



Robert C. Martin

(S)ingle Responsibility

Un módulo debe tener una única responsabilidad

Responsabilidad = motivo para cambiar

No debe haber más de un motivo para cambiar un módulo

Sino, las responsabilidades se mezclan y acoplan



VS



(S)ingle Responsibility

Departamentos responsables

```
class Employee {  
  def calculatePay(): Money = { ??? } ← Finanzas  
  
  def saveDB() { ??? } ← Operaciones  
  
  def reportWorkingHours(): String = { ??? } ← Gestión  
}
```

Hay multiples razones para cambiar la clase Empleado

Solución: Separar preocupaciones (concerns)

Juntar las cosas que cambian por las mismas razones
Separar las cosas que cambian por razones diferentes

(O)pen/Closed

Abierto para extender

El módulo puede adaptarse a nuevos cambios

Cambiar/adaptar comportamiento del módulo

Cerrado para modificar

Los cambios pueden realizarse sin modificar el módulo

Cambiar sin modificar código fuente, binarios, etc.

Debe ser sencillo cambiar comportamiento de un módulo sin cambiar su código Fuente o tener que recompilar

(O)pen/Closed

Ejemplo:

```
class SelectProducts {  
    List<Product> filterByColor(List<Product> products,  
                                String color) {  
  
        . . .  
    }  
}
```

Si queremos filtrar por altura, tendríamos que cambiar el código fuente

Otra solución:

```
List<Product> filter(List<Product> products,  
                    Predicate<Product> criteria) {  
  
    . . .  
}
```

Ahora sí es posible filtrar por cualquier otro predicado sin cambiar el módulo

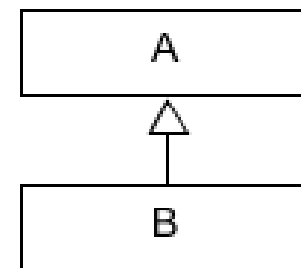
```
List<Product> redProducts = selector.filter(p -> p.color.equals("red"));  
List<Product> biggerProducts = selector.filter(p -> p.height > 30);
```

Principio sustitución Liskov

Los subtipos deben seguir el contrato de los supertipos

Un tipo B es un subtipo de A cuando:

$\forall x \in A$, si hay una propiedad Q tal que $Q(x)$
entonces $\forall y \in B$, $Q(y)$



“Los tipos derivados deben ser completamente sustituibles por los tipos base”

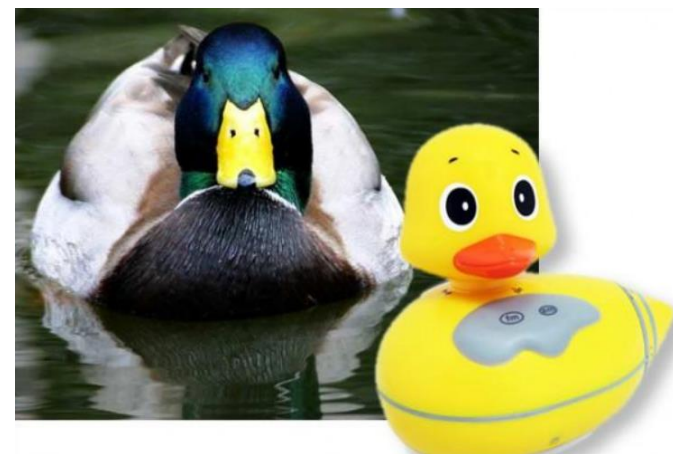
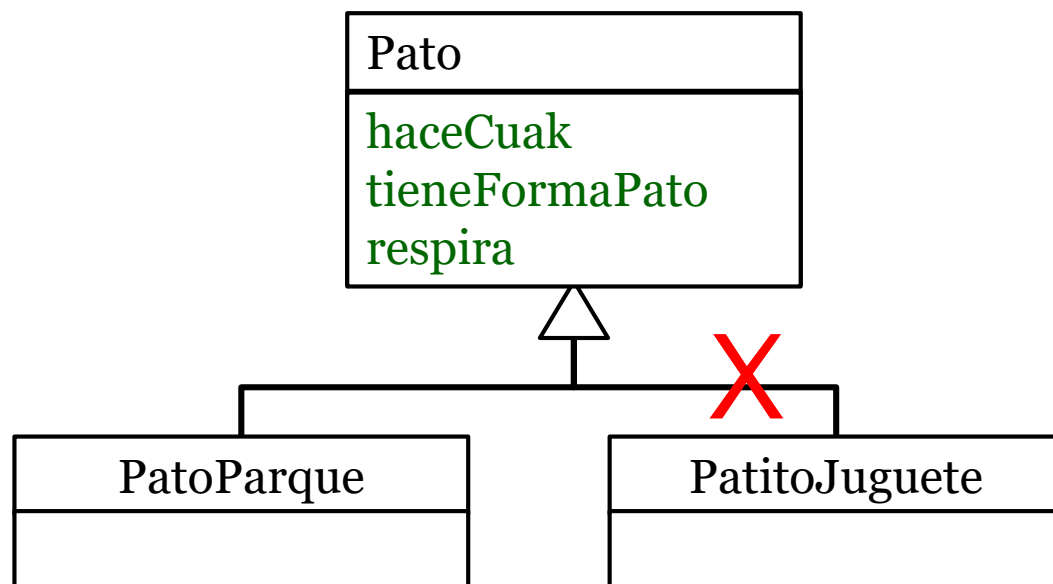
Errores habituales:

Heredar y modificar comportamiento clase base

Añadir funcionalidad a supertipos que los subtipos no cumplen

(L)iskov

Subtipos deben respetar el contrato de sus supertipos



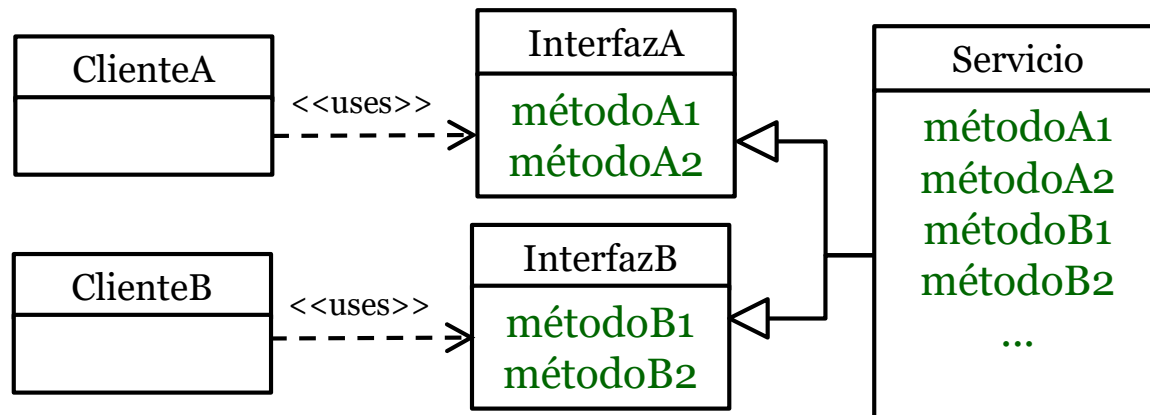
(I)nterface Segregation

Clientes no deben depender de métodos que no usan

Mejor utilizar interfaces pequeños y cohesivos

En caso contrario \Rightarrow dependencias no deseadas

Si un módulo depende de funciones que no utiliza y éstas cambian puede verse afectado



(D)ependency inversion

Módulos alto nivel no dependen de módulos bajo nivel

Todos dependen de abstracciones

Las abstracciones no dependen de los detalles

Puede obtenerse mediante inyección de dependencias, o con otros patrones como *plugin*, *service locator*, etc.

Recomendaciones modularidad

SOLI(D) (DIP - Dependency Inversion Principle)

Minimiza acoplamiento

Facilita creación de pruebas unitarias

Sustituyendo módulos de bajo nivel por dobles de pruebas

Inyección de dependencias

Varios frameworks: Spring, Guice, etc.



Otras recomendaciones modularidad

Ley de Demeter - Principio de menor conocimiento

Nombre a partir del sistema Demeter (1988)

Cada módulo sólo se comunica con módulos próximos

Objetivo: bajar acoplamiento

Bajar número de métodos invocados en cada método

Síntomas de mal diseño:

Usar más de un punto...

`a.b.método(...)` 

`a.método(...)` 

NOTA: Solución de compromiso

No siempre es positivo adherirse a esta ley



Otras recomendaciones modularidad

Interfaz fluida (fluent API)

Crear interfaces que faciliten su lectura

Permite encadenar métodos

Ejemplo:

```
Product p = new Product().setName("Pepe").setPrice(23);
```

Ventajas

Código más legible

Facilita lenguajes de dominio específico

Facilidades para auto-completado en IDEs

Truco: Métodos que modifican un objeto, devuelven dicho objeto

```
class Product {  
    ...  
    public Product setPrice(double price) {  
        this.price = price;  
        return this;  
    };  
};
```



No contradice la Ley de Demeter porque actúa sobre el mismo objeto

Principios cohesión/acoplamiento

Principios cohesión

Reuse/Release Equivalent Principle (REP)

Common Reuse Principle (CRP)

Common Closure Principle (CCP)

Principios acoplamiento

Acyclic dependencies Principle (ADP)

Stable Dependencies Principle (SDP)

Stable Abstractions Principle (SAP)



Robert C. Martin

Principios de cohesión

REP

Reuse/Release Equivalence Principle

Equivalencia entre reutilización/release

Agrupar en *release* componentes a reutilizar

Para reutilizar un elemento es necesario publicarlo en algún sistema de gestión de releases

Gestión de versiones de releases: números/nombres

Todas las entidades relacionadas deben agruparse

Las entidades se agrupan para ser reutilizadas

CCP

Common Closure Principle

Juntar en un modulo las entidades que cambian por las mismas razones y al mismo tiempo

Las entidades que cambian juntas deben pertenecer al mismo módulo

Objetivo: limitar la dispersión de cambios entre releases de módulos

Los cambios deben afectar al menor número de módulos publicados

Las entidades de un modulo deben ser cohesivas

Agrupar entidades para mantenimiento

Nota: Este principio es similar al SRP (Single Responsibility Principle), para módulos

CRP

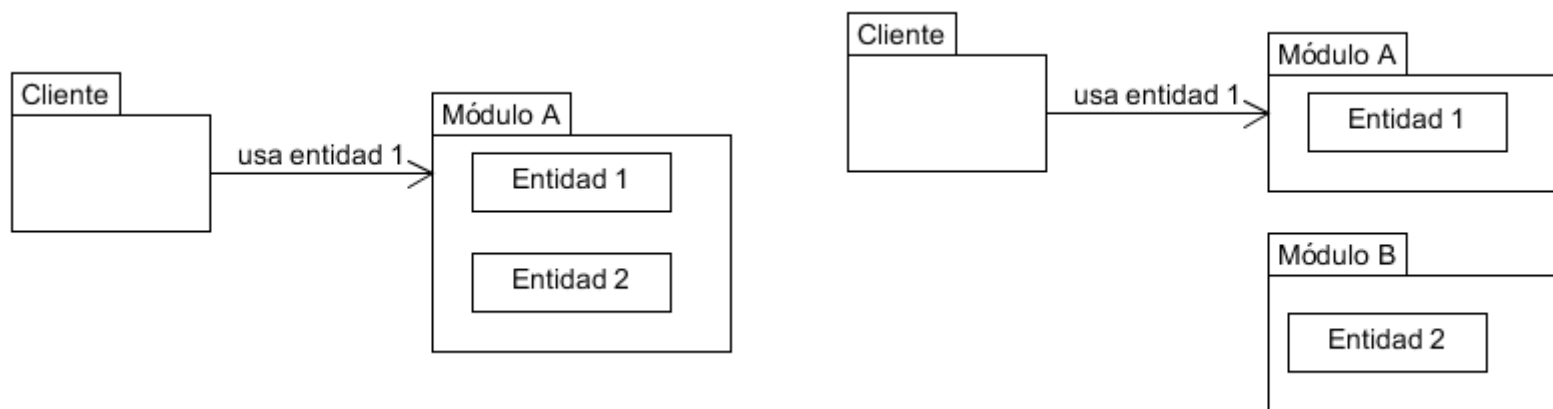
Common Reuse Principle

Los módulos sólo deberían depender de entidades que necesiten

No deberían depender de cosas que no necesiten

En caso contrario, un consumidor se verá afectado por cambios de entidades que no usa

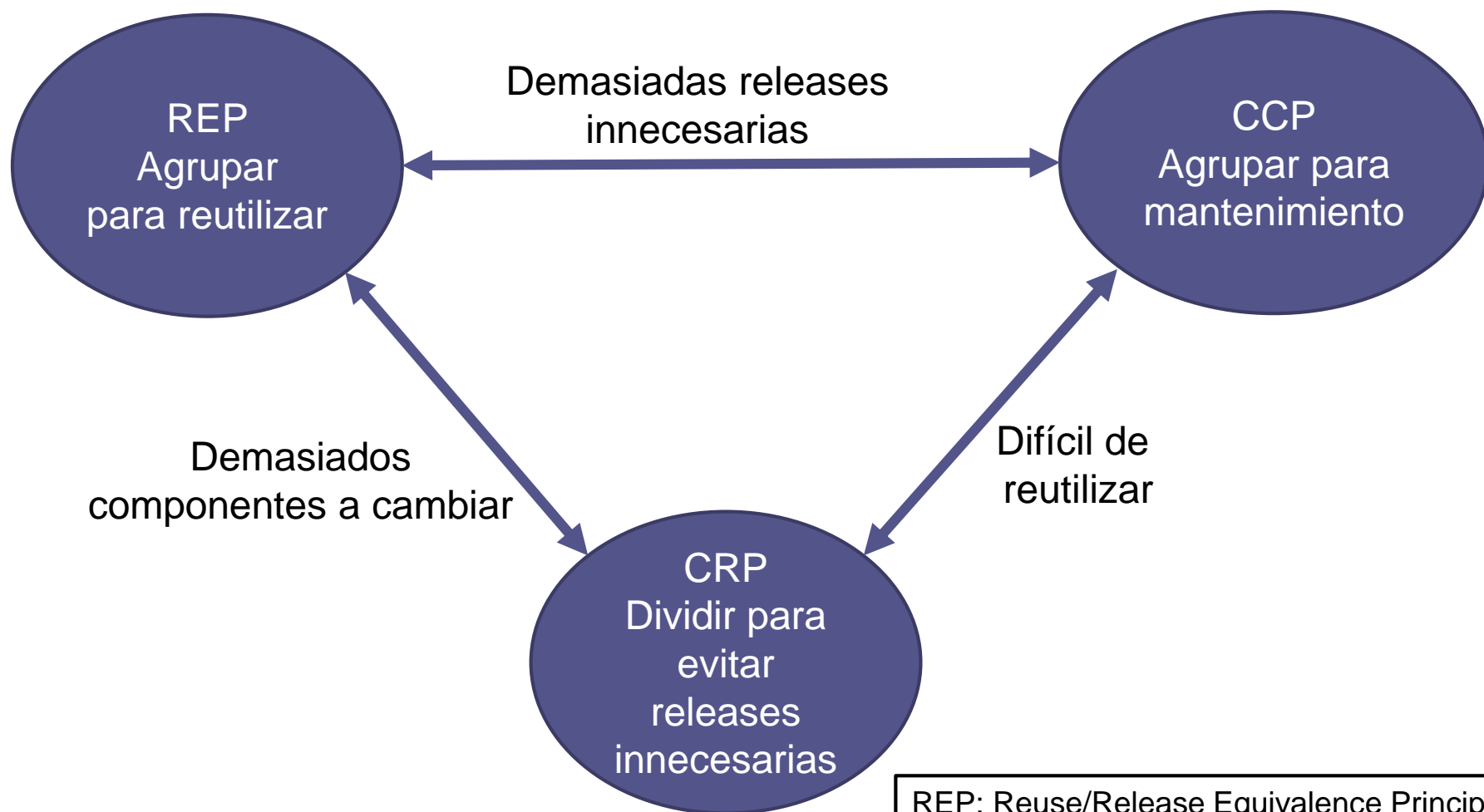
Dividir módulos para evitar *releases* innecesarias



Nota: Este principio está relacionado con ISP (Interface Segregation Principle)

Tensión entre principios cohesión de componentes

Coste de abandonar uno de los principios



REP: Reuse/Release Equivalence Principle
CCP: Common Closure Principle
CRP: Common Reuse Principle

Principios de acoplamiento

ADP

Acyclic Dependencies Principle

La estructura de dependencias de módulos debe formar un grafo dirigido acíclico

Evitar ciclos

Un ciclo puede hacer un pequeño cambio muy difícil

Muchos módulos pueden verse afectados

Problema para identificar el orden de construcción

NOTA: Los ciclos pueden evitarse mediante el DIP (Dependency Inversion Principle)

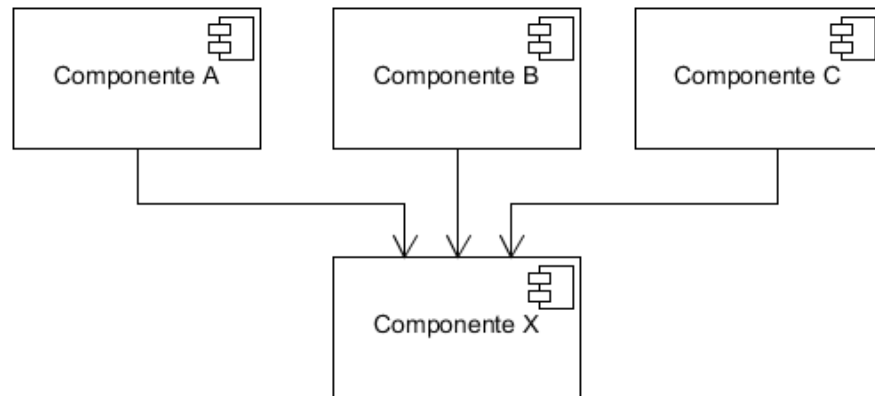
SDP

Stable Dependencies Principle

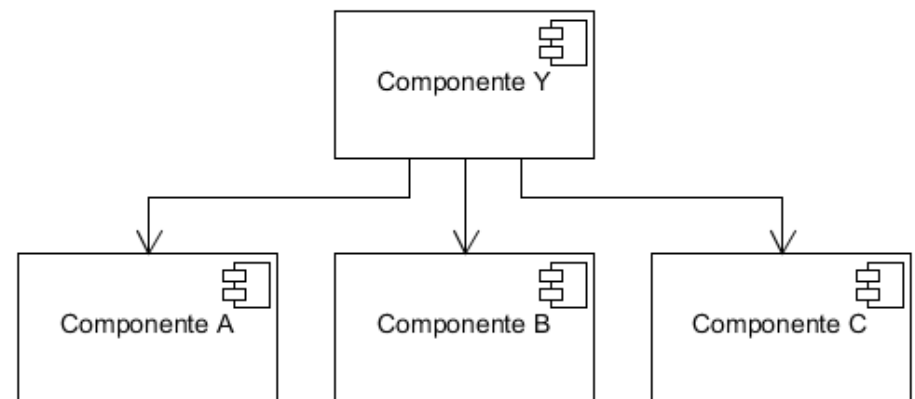
Las dependencias entre componentes en un diseño deberían ir en la dirección de estabilidad

Un componente debería depender solamente de componentes más estables que él

> Estabilidad = menores razones para cambiar



Componente X es estable



Componente Y es inestable

Tiene al menos 3 razones para cambiar

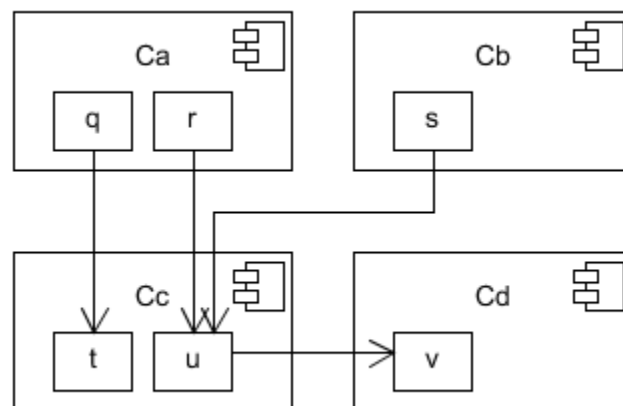
Midiendo la estabilidad

Fan-in: Dependencias de entrada

Fan-out: Dependencias de salida

$$\text{Inestabilidad } I = \frac{\text{Fan-out}}{\text{Fan-in} + \text{Fan-out}}$$

Valor entre 0 (estable) y 1 (inestable)



$$I(Ca) = \frac{2}{0+2} = 1$$

$$I(Cb) = \frac{1}{0+1} = 1$$

$$I(Cc) = \frac{1}{3+1} = \frac{1}{4}$$

$$I(Cd) = \frac{0}{1+0} = 0$$

El *Stable Dependencies Principle* indica que las dependencias deberían ir de mayor inestabilidad a menor

<http://wiki.c2.com/?StableDependenciesPrinciple>

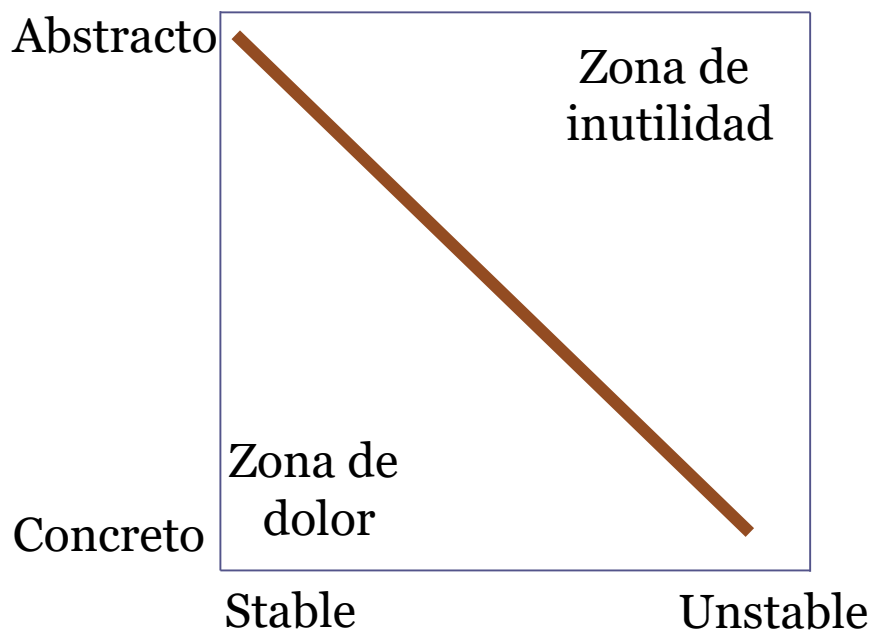
SAP

Stable Abstractions Principle

Un modulo debería ser tan abstracto como estable

Los paquetes más estables deberían tener máximo grado de abstracción

Paquetes inestables deberían ser concretos



Ejemplos

- Abstracto/estable = Interfaces con muchos módulos dependientes
- Concreto/Inestable = Implementaciones con muchos módulos dependientes
- Zona de dolor = esquemas de base datos
- Zona de inutilidad = interfaces sin implementaciones

Otras recomendaciones modularidad

Facilitar configuración externa del módulo

Crear un módulo externo de configuración

Proporcionar implementación por defecto

Principios GRASP (General Responsibility Assignment Software Patterns)

Sistemas de módulos

OSGi: Sistema de módulos para Java

Módulo = bundle

Control de encapsulación

Permite instalar, arrancar, detener, desinstalar
módulos en tiempo de ejecución

Utilizado en Eclipse

Módulos = Micro-servicios

Implementación: Apache Felix, Equinox

Proyecto Jigsaw (Java 9)

.Net soporta módulos mediante Assemblies

Sistemas de módulos

En NodeJs

Basado inicialmente en CommonJs

`require` importa un módulo

`exports` declara un objeto que se estará disponible

person.js

```
const VOTING_AGE = 18
const person = {
  name: "Juan",
  age: 20
}
function canVote() {
  return person.age > VOTING_AGE
}
module.exports = person;
module.exports.canVote = canVote;
```

```
const person = require('./person');

console.log(person.name);
console.log(person.canVote());
```

Sistemas de módulos

En Javascript (ES6), require Babel en Node

`import` importa un módulo

`export` declara un objeto disponible

person.js

```
const VOTING_AGE = 18;
export const person = {
  name: "Juan",
  age: 20
};
export function canVote() {
  return person.age > VOTING_AGE
}
```

```
import { canVote, person } from './person';
console.log(person.name);
console.log(person.canVote());
```

Estilos de modularidad

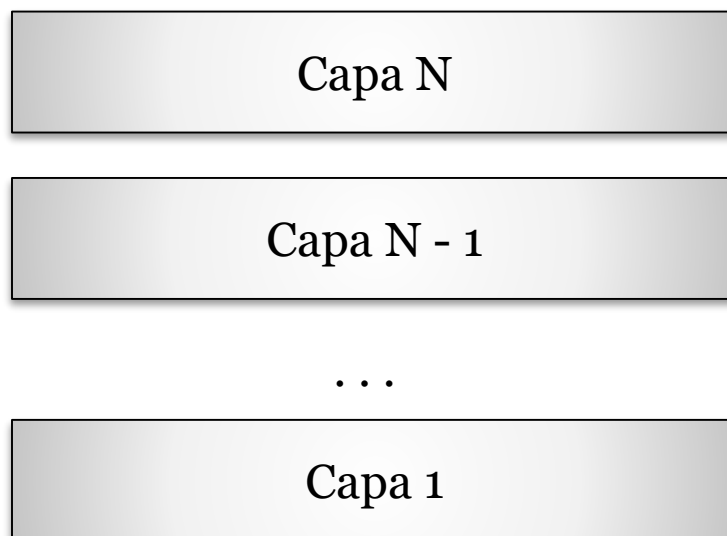
Capas

Capas

Partición del software en capas (*layers*)

Orden estricto entre capas

Cada capa expone un interfaz (API) que puede utilizarse por las capas superiores

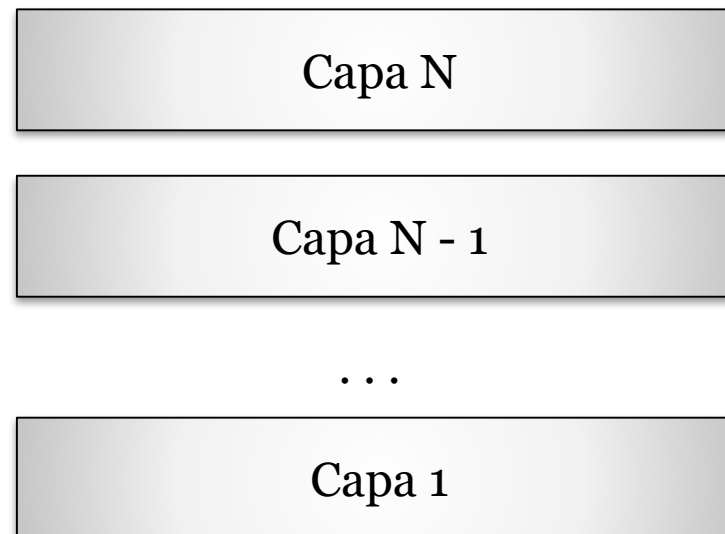


Capas

Elementos

Capa: conjunto de funcionalidades expuestas mediante un interfaz en un nivel N

Relación de orden: relación ordenada de las capas



Capas

Restricciones

Cada pieza de software está en una capa

Existen al menos 2 capas

Una capa puede ser:

- Cliente: consume servicios de capas inferiores

- Servidor: proporciona servicios a capas superiores

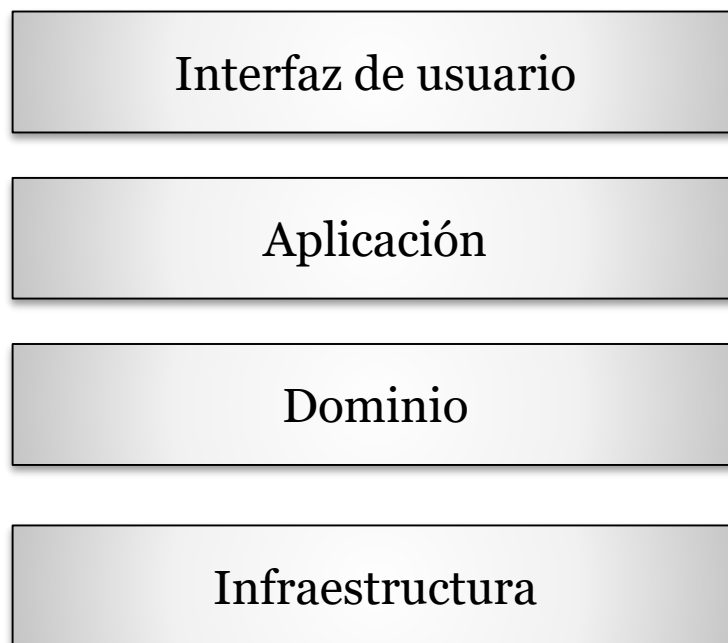
Capa N sólo puede utilizar funcionalidad capa N-1

- Variante laxa: invocar funcionalidad capas 1 a N-1

- No hay ciclos

Capas

Ejemplo



Capas

Capas \neq Módulos

Una capa puede ser un módulo...

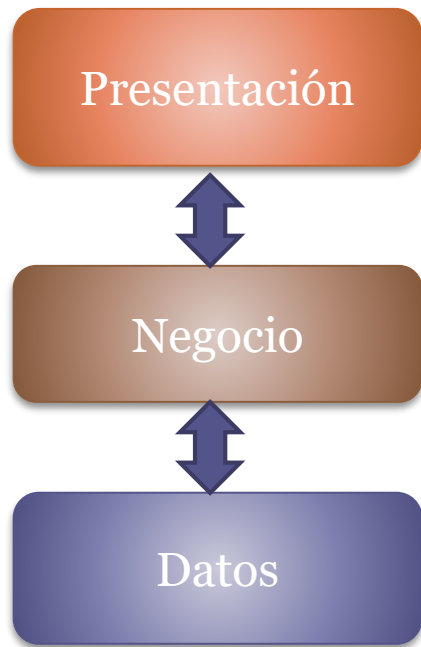
...pero los módulos pueden descomponerse en otros módulos (las capas no)

Segmentando una capa se obtienen módulos

Capas

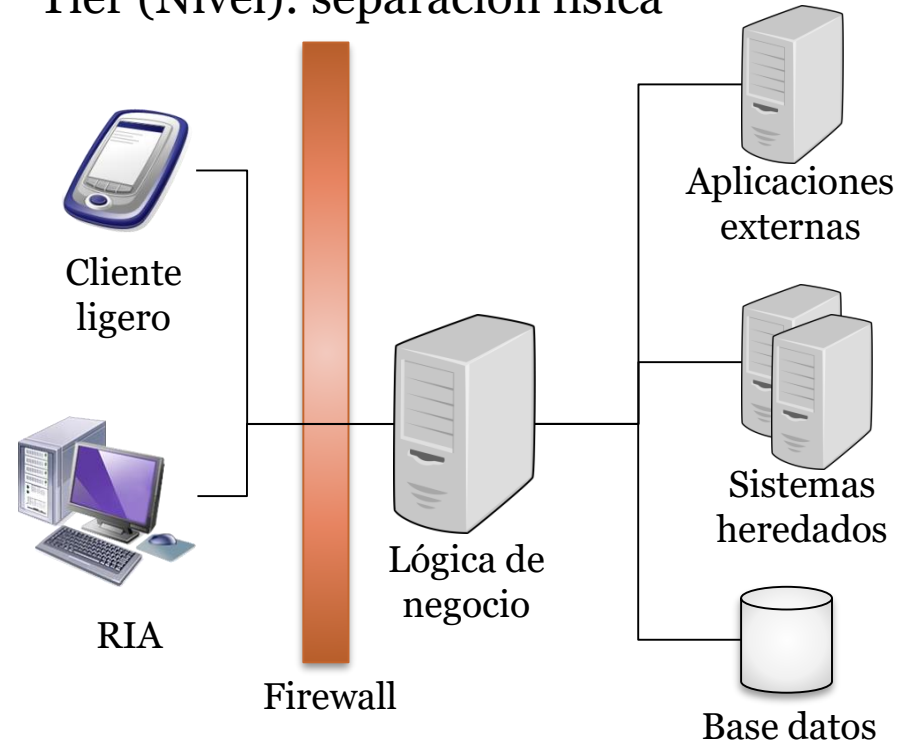
Capas \neq Tiers

Layer (capa): separación conceptual



3-Capas (3-Layers)

Tier (Nivel): separación física



Presentación

Negocio

Datos

3-niveles (3-tier)

Capas

Ventajas

Separación de niveles abstracción

Facilita evolución independiente de cada capa

Manteniendo el API, pueden ofrecerse diferentes implementaciones de las capas

Reutilización

Cambios en una capa afectan solamente a capa inferior/superior

Pueden crearse interfaces estándar a modo de librerías y marcos de aplicaciones

Capas

Problemas

No siempre puede aplicarse

No siempre hay niveles de abstracción diferentes

Rendimiento

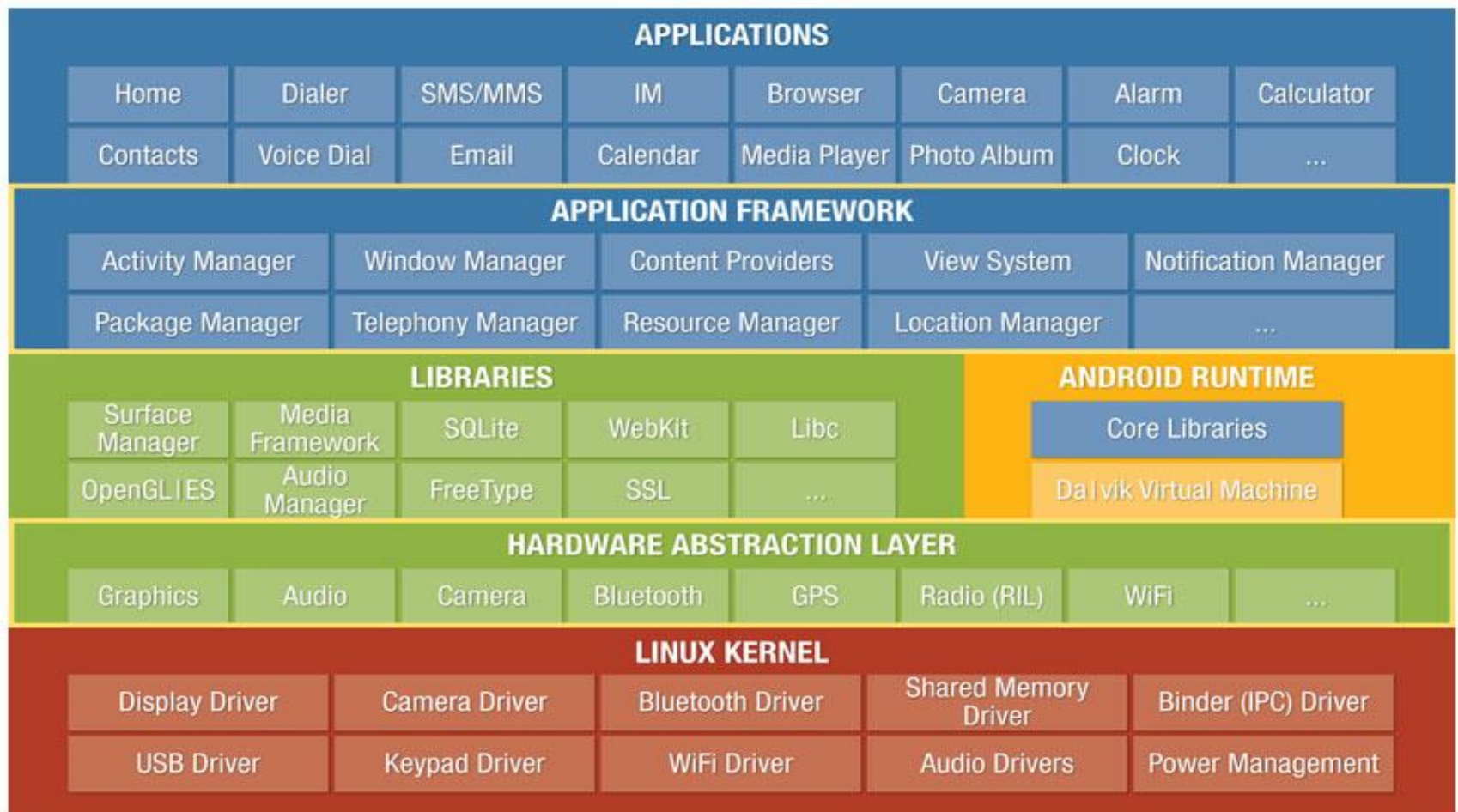
Acceso a través de capas puede ralentizar el sistema

Atajos

En ocasiones es necesario saltarse el nivel de capas

Capas

Ejemplo: Android



Capas

Variantes:

Máquinas virtuales, APIs

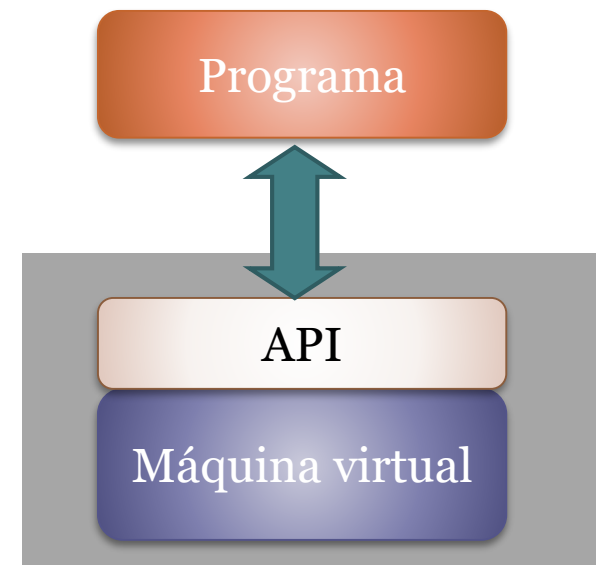
3-capas, N-capas

Máquina virtual

Capa opaca

Oculto una determinada implementación

Sólo puede accederse mediante un API



Máquina virtual

Ventajas

- Portabilidad

- Simplicidad en desarrollo de software

 - Programación a nivel más alto

- Facilidades para simulación

Problemas

- Ejecución más lenta

 - Técnicas JIT

- Sobrecarga computacional debido a la nueva capa

Máquina virtual

Aplicaciones

Lenguajes de programación

JVM: Java Virtual Machine

CLR .Net

Software de emulación

3-capas (N-capas)

Descomposición conceptual

Presentación

Lógica de negocio ó dominio

Datos



Presentación

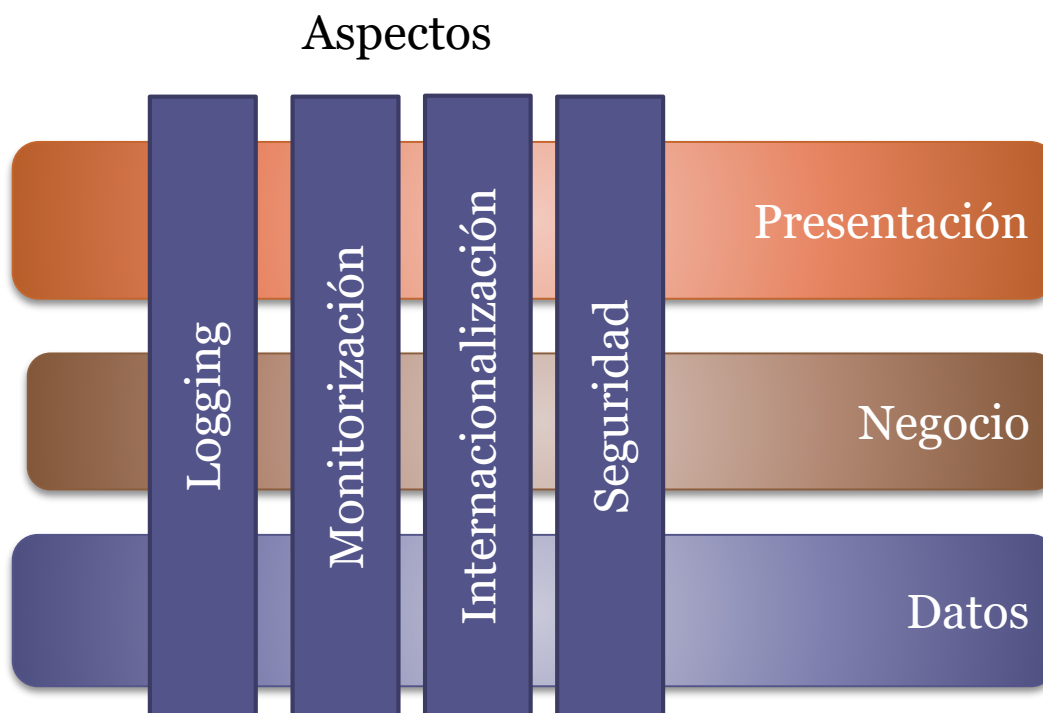
Negocio

Datos

Aspectos

Orientación a aspectos

Aspectos: Módulos que implementan características transversales



Orientación a Aspectos

Elementos:

Crosscutting concern (característica transversal).

Funcionalidad que se requiere en numerosas partes de una aplicación

Ejemplo: logging, monitorización, i18n, seguridad,...

Generan código enredado (*tangling*)

Aspecto. Captura un *crosscutting-concern* en un módulo

Orientación a aspectos

Ejemplo: Reservar asientos de avión

Varios métodos de reserva:

- Reservar un asiento

- Reservar una fila

- Reservar un par de asientos juntos

- ...

En cada reserva es necesario:

- Comprobar permisos (seguridad)

- Concurrencia (bloquear asientos)

- Transacciones (realizar la operación en un solo paso)

- Crear un log de la operación

- ...

Orientación a aspectos

Solución tradicional

```
class Avión {  
    void reservaAsiento(int fila, int número) {  
        // ... Log petición de reserva  
        // ... chequear autorización  
        // ... chequear que está libre  
        // ... bloquear asiento  
        // ... Iniciar transacción  
        // ... Log inicio de operación  
        // ... Realizar reserva  
        // ... Log fin de operación  
        // ... Ejecutar o deshacer transacción  
        // ... Desbloquear  
    }  
    ...  
    public void reservaFile(int fila) {  
        // ... Más o menos lo mismo!!!!  
    }  
    ...  
}
```

Seguridad

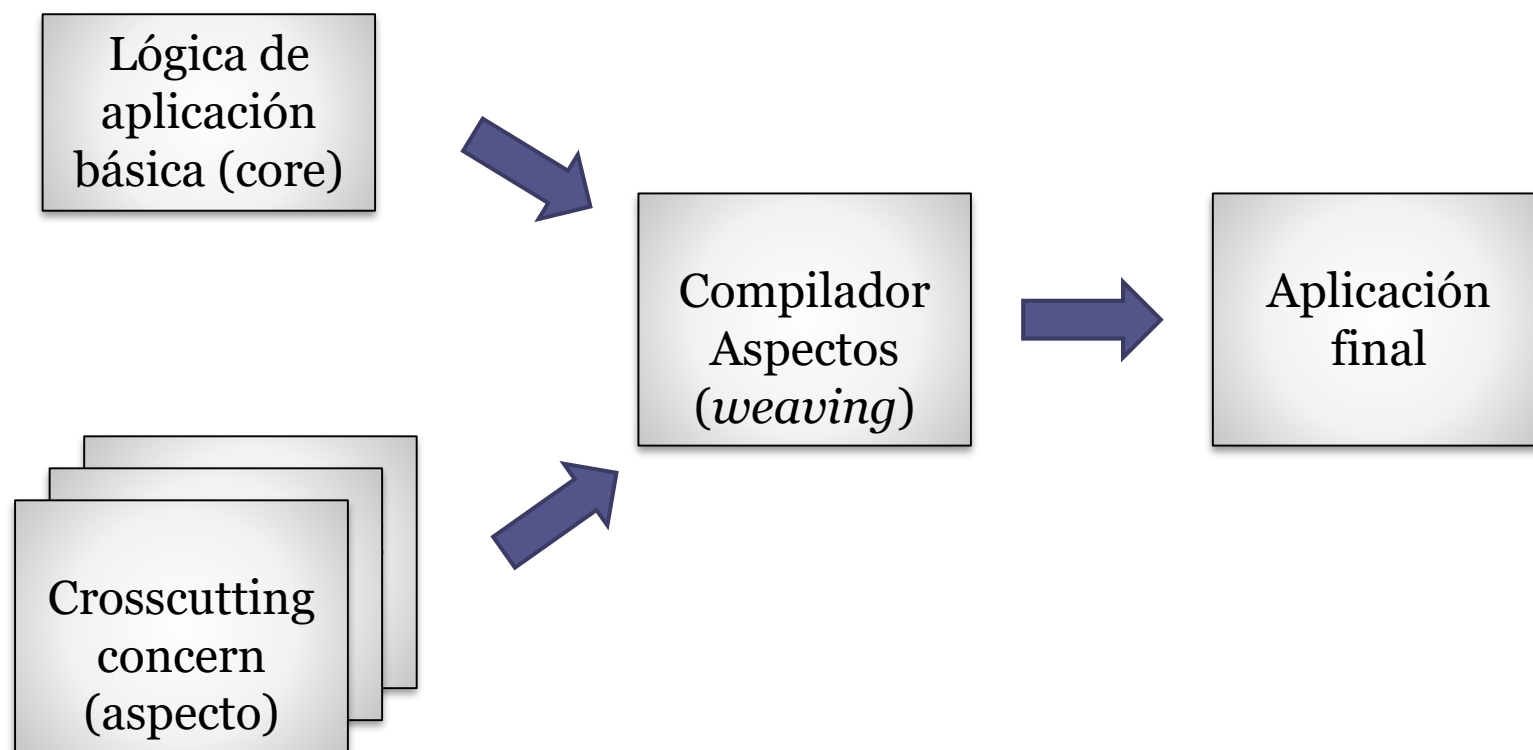
Concurrencia

Auditoría (log)

Transacciones

Orientación a aspectos

Estructura



Orientación a aspectos

Definiciones

Join point: Punto en el que se puede introducir un aspecto

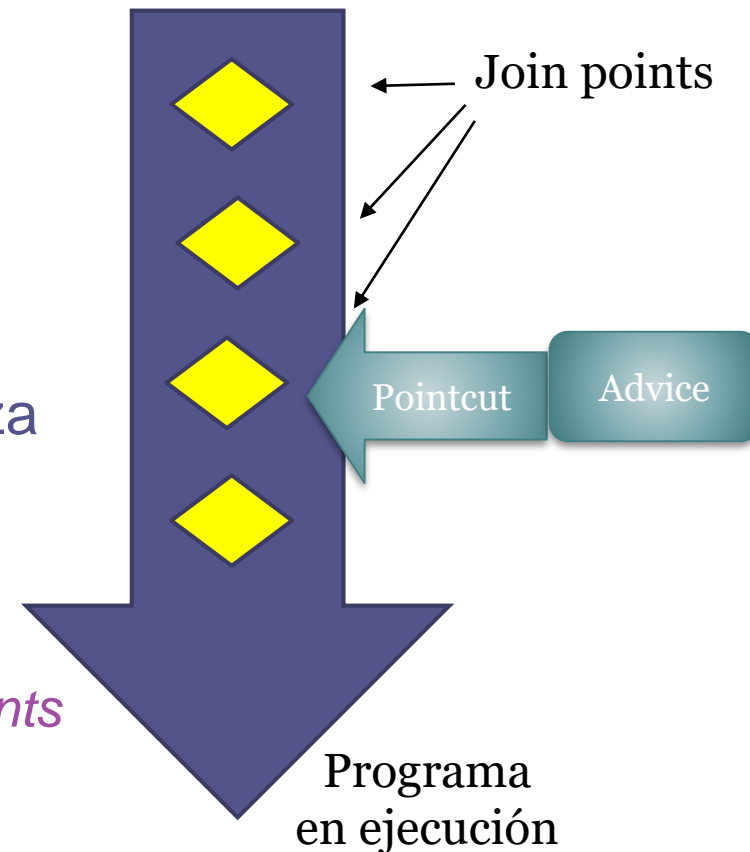
Aspecto:

Formado por:

Advice: define el trabajo que realiza un aspecto

Pointcut: Dónde se va a introducir un aspecto

Puede encajar uno o varios *join points*



Orientación a aspectos

Ejemplo de aspecto en @Aspectj

```
@Aspect
public class Seguridad {

    @Pointcut("execution(* es.uniovi.asw.Avión.reserva*(..))")
    public void accesoSeguro() {}

    @Before("accesoSeguro()")
    public void asegura(JoinPoint joinPoint) {
        // Realiza la autenticación
    }
}
```

Métodos de la forma
reserva*

Se ejecuta antes de
invocar a dichos
métodos

Permite acceder a
información del
punto de unión
Argumentos

Orientación a aspectos

Restricciones:

Un aspecto afecta a uno o más módulos tradicionales.

Un aspecto captura todas las definiciones de una *crosscutting-concern*

El aspecto es introducido en el código
Herramientas de introducción automática

Orientación a aspectos

Ventajas

- Diseño más simple

 - Aplicación básica limpia

- Facilitar modificación y mantenimiento del sistema

 - Crosscutting concerns localizados

- Reutilización

 - Los *crosscutting concerns* pueden reutilizarse en otros sistemas

Orientación a aspectos

Problemas

Necesidad de herramientas externas.

Compilador de aspectos: AspectJ

Otras herramientas: Spring, JBoss

Depuración más compleja

Un error en un módulo de aspectos podría tener consecuencias imprevistas en otros módulos

Flujo de programa más complicado

Necesidad de habilidades del equipo de desarrollo

No todos los desarrolladores lo conocen

Orientación a aspectos

Aplicaciones

AspectJ = Extensión de Java con AOP

Guice = Framework de inyección de dependencias

Spring = Marco de aplicaciones empresariales con inyección de dependencias y AOP

Variantes

DCI (Data-Context-Interaction): Se centra en identificar roles a partir de los casos de uso

Apache Polygene

Basados en dominio

Basados en dominio

- Domain driven design

- Estilo hexagonal

- Modelos centrados en datos

- Domain Driven Design de N-capas

- Data driven

- Patrones

 - CQRS

 - Event sourcing

 - Naked Objects

Modelos de datos vs dominio

Modelos de datos

Físico: Representación
datos

Tablas, columnas, claves, ...

Lógico: Estructura de los
datos

Entidades y relaciones

Modelos de dominio

Modelo conceptual del
dominio.

Vocabulario y contexto

Entidades, relaciones

Comportamiento

Reglas de negocio

Estilos basados en dominio

Centrar el enfoque en el dominio y la lógica

Se anticipan cambios en el dominio

Colaboración desarrolladores y expertos de dominio

Basados en dominio

Elementos

Modelo de dominio: suele estar formado por

Contexto

Entidades

Relaciones

Aplicación

Manipula elementos del dominio

Basados en dominio

Restricciones

Modelo de dominio refleja un contexto

Aplicación centrada en dominio

La aplicación debe adaptarse a los cambios en el modelo de dominio

No hay restricciones topológicas

Basados en dominio

Ventajas:

- Facilita comunicación del equipo

 - Lenguaje ubicuo

- Refleja estructura del dominio

 - Facilidad para afrontar cambios en dominio

 - Compartir y reutilizar modelos

- Refuerza calidad y consistencia de datos

- Facilita realización de pruebas del sistema

 - Creación de dobles de pruebas

Basados en dominio

Problemas:

- Requiere colaboración con expertos del dominio

- Análisis estancado

 - Establecer límites del contexto

- Modelo anémicos

 - Objetos sin comportamiento (delegado a otra capa)

- Dependencia tecnológica

 - Evitar modelos de dominio dependientes de tecnologías de persistencia concretas

- Sincronización

 - Establecer técnicas para sincronizar sistema con cambios del dominio

Basados en dominio

Variantes

DDD - *Domain driven design*

Estilo hexagonal

Centrados en datos

N-Layered Domain Driven Design

Patrones relacionados:

CQRS (Command Query Responsibility
Segregation)

Event Sourcing

Naked Objects

DDD - Domain Driven Design

Filosofía de desarrollo

Objetivo: Comprensión del dominio

Involucrar expertos de dominio - equipo desarrollo

Lenguaje ubicuo

Vocabulario común que deben conocer tanto los expertos de dominio como el equipo de desarrollo

DDD - Domain Driven Design

Elementos

Límites contextuales (*boundary context*)

Límites del dominio

Entidades

Un objeto con identidad

Objetos valor:

Contienen atributos pero no identidad

Agregados:

Colección de objetos ligados por una entidad raíz

Repositorios

Servicio de almacenamiento

Factoría

Se encarga de la creación de objetos solamente

Servicios

Operaciones externas

DDD - Domain Driven Design

Restricciones

Elementos de un agregado no son accesibles desde el exterior. Solamente a través de la entidad raíz

Repositorios son los que gestionan almacenamiento

Objetos valor son inmutables

Normalmente tienen solamente atributos

DDD - Domain driven design

Ventajas

Organización de código

Identificación de partes importantes

Mantenimiento/evolución del sistema

Facilidades para refactorizar

Se adapta a Behaviour Driven Development

Facilita comunicación

```
graph LR; A([Espacio de problema  
Expertos de dominio]) --- B[Lenguaje Ubicuo]; B --- C([Espacio de solución  
Equipo de desarrollo]);
```

Espacio de problema
Expertos de dominio

Lenguaje Ubicuo

Espacio de solución
Equipo de desarrollo

DDD - Domain driven design

Problemas

Involucrar expertos de negocio en desarrollo

No siempre es fácil

Complejidad aparente

Impone restricciones en desarrollo

Estilo útil para dominios relativamente complejos

Estilo hexagonal

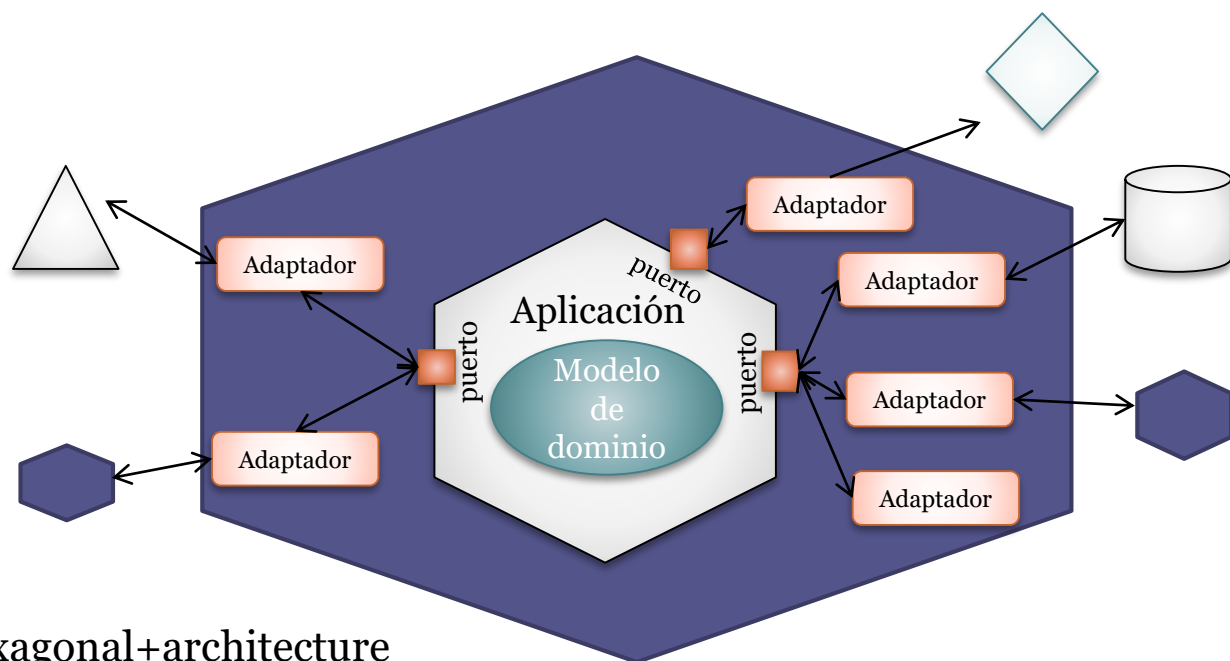
Otros nombres:

Puertos y adaptadores, onion, clean, etc.

Enfoque en modelo de dominio

Infraestructuras y frameworks están en el exterior

Acceso mediante puertos y adaptadores



<http://alistair.cockburn.us/Hexagonal+architecture>

<http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html>

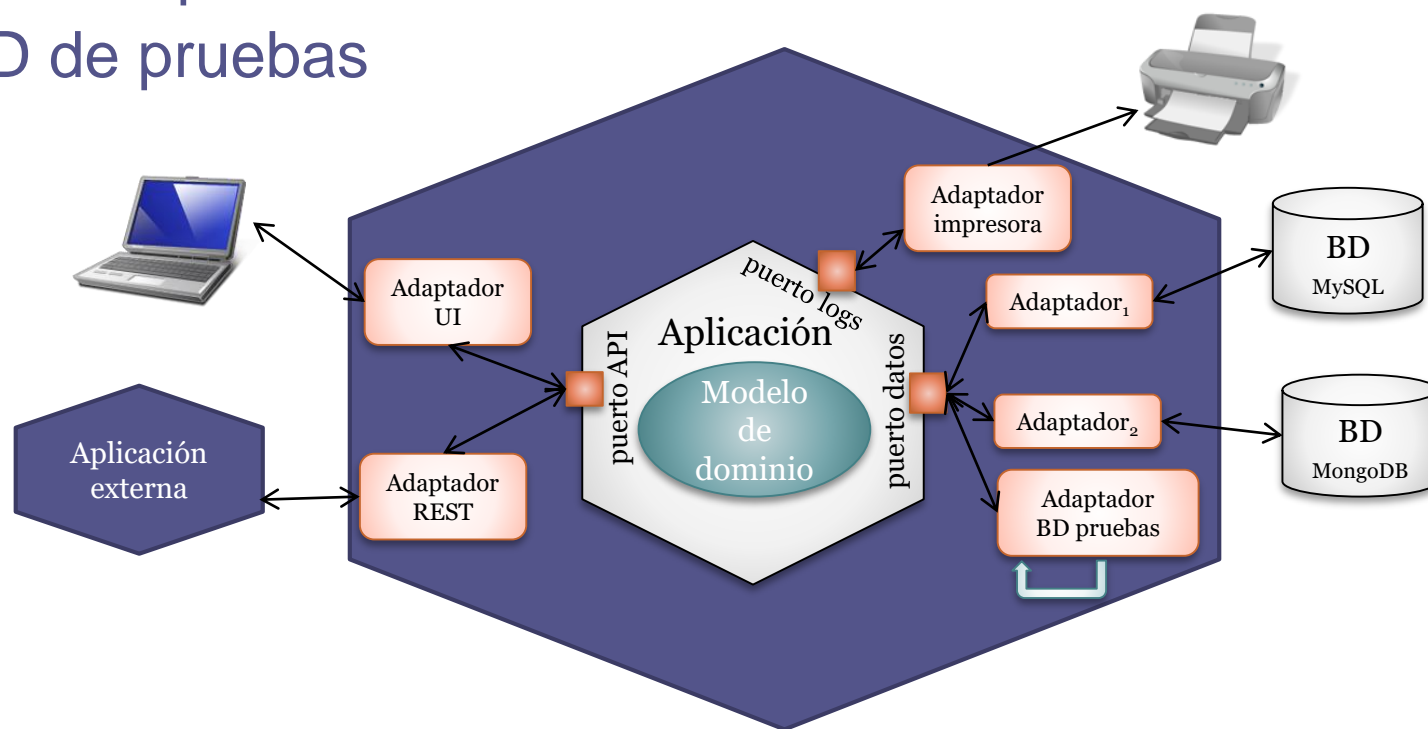
Estilo hexagonal

Ejemplo

Aplicación en capas tradicional

Se incorporan nuevos servicios

BD de pruebas



Estilo hexagonal

Elementos

Modelo de dominio

Representa lógica de negocio: Entidades y relaciones
Objetos planos (POJO)

Puertos

Interfaz de comunicación
Habitualmente: Usuario, Base de datos

Adaptadores

Un adaptador por cada elemento externo
Ejemplo: REST, Usuario, BD SQL, BD mock,...

Estilo hexagonal

Ventajas

Comprensión

Facilita la comprensión del dominio

Atemporalidad

Menor dependencia de tecnologías y frameworks

Adaptabilidad (*time to market*)

Facilidad para adaptar aplicación a cambios dominio

Testabilidad

Puede testearse sustituyendo BD reales por BD mock

Estilo hexagonal

Problemas

Puede ser difícil separar dominio de persistencia

Muchos frameworks mezclan ambos

Asimetría de puertos & adaptadores

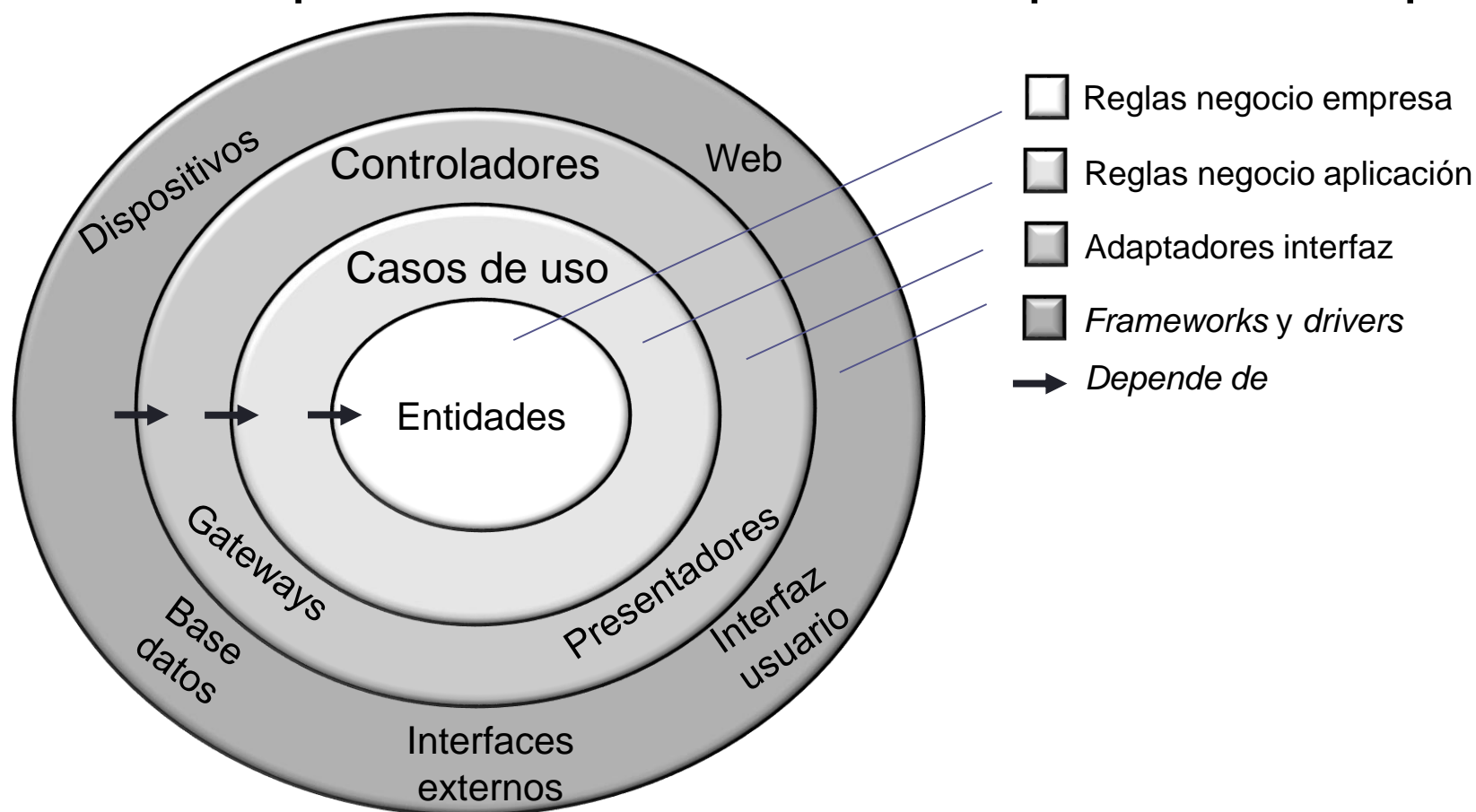
No todos son iguales

Puertos activos (ej. usuario) vs pasivos (ej. logger)

Arquitectura limpia

La misma que la arquitectura hexagonal

Presentada por *Uncle Bob* - Libro arquitectura limpia



Centrados en datos

Dominios sencillos basados en datos

Operaciones CRUD

Create-Retrieve-Update-Delete

Ventajas:

Generación pseudo-automática (*scaffolding*)

Velocidad rápida de desarrollo (time-to-market)

Problemas

Evolución a dominios más complejos

Dominios anémicos

Clases que solamente tienen *getters/setters*

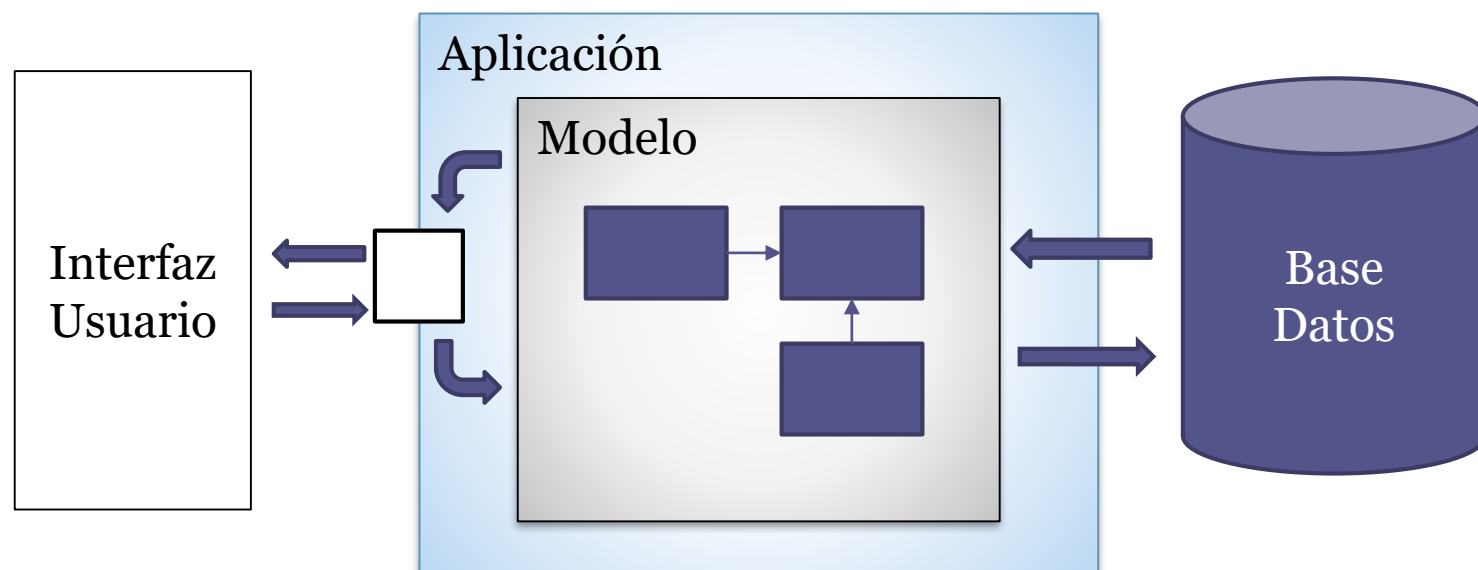
CQRS

Command Query Responsibility Segregation

Separar el modelo en 2 partes

Command (*modificación*): Realiza cambios

Query (*consulta*): Sólo realiza consultas, actualiza interfaz



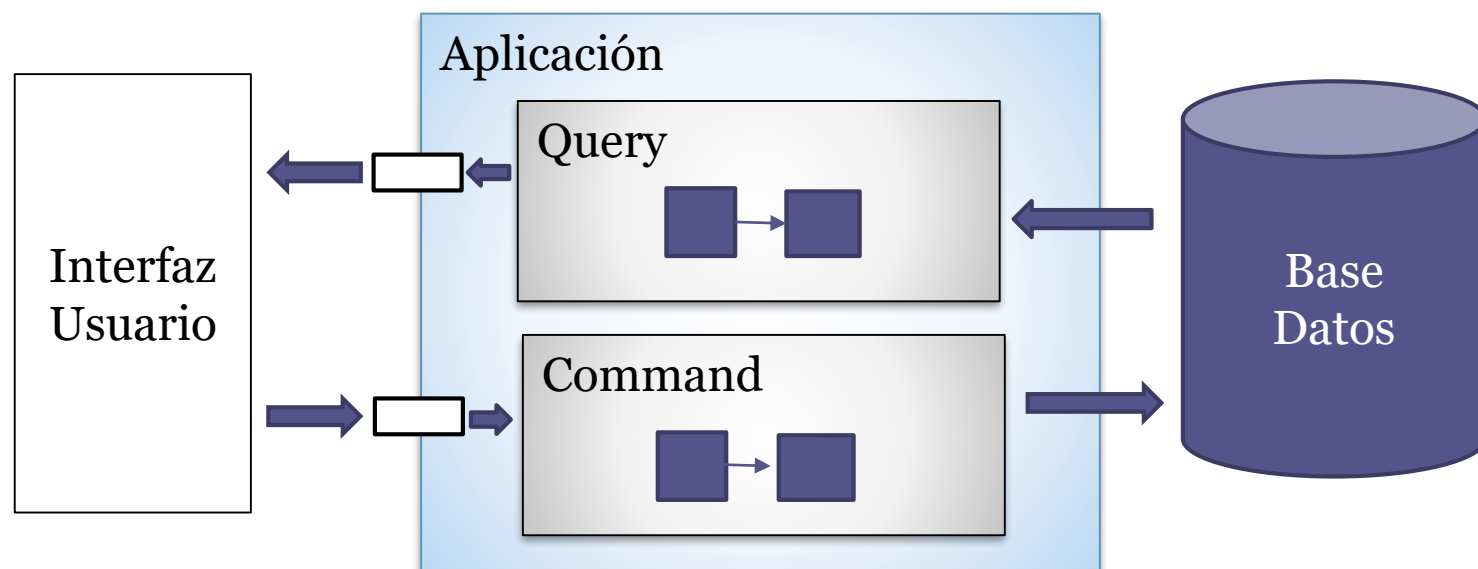
CQRS

Command Query Responsibility Segregation

Separar el modelo en 2 partes

Command (*modificación*): Realiza cambios

Query (*consulta*): Sólo realiza consultas, actualiza interfaz



CQRS

Ventajas

Escalabilidad

Optimizar consultas (sólo lectura)

Comandos asíncronos

Facilita descomposición de equipos

CQRS

Problemas

Operaciones híbridas (consulta/comando)

Ejemplo: *pop()* en una pila

Complejidad

En entornos CRUD puede ser excesivo

Sincronización

Posibilidad de consultas sobre datos no actualizados

Aplicaciones

Axon Framework

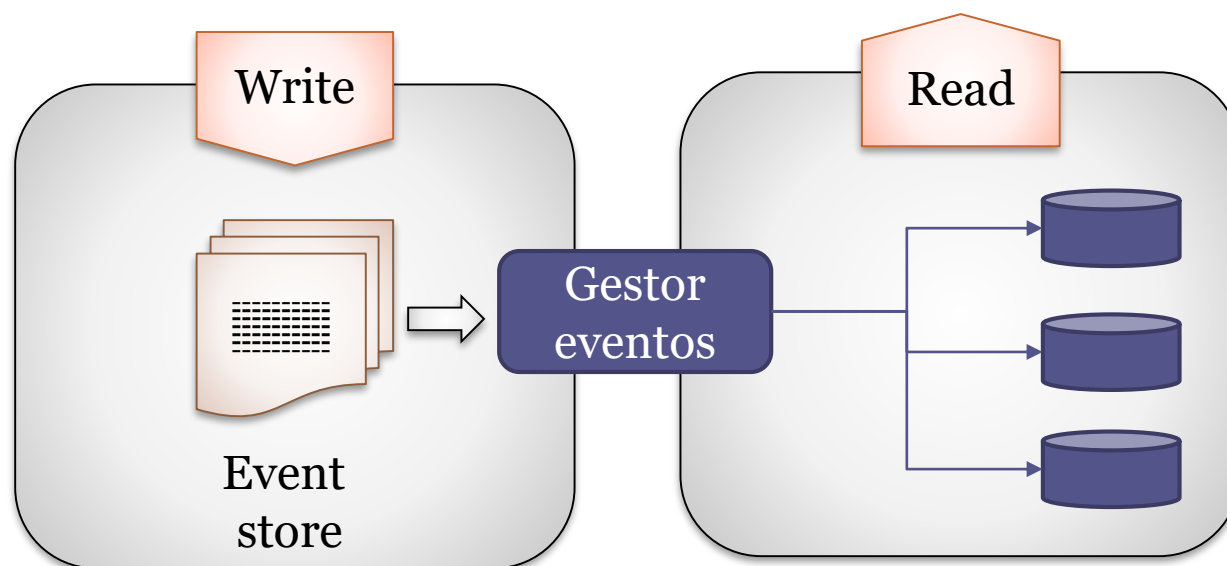
Event Sourcing

Capturar cambios del estado mediante eventos

Permite seguir traza de cómo se llegó a un determinado estado

Event Store

Siempre se añaden eventos (no se cambian)



Event Sourcing

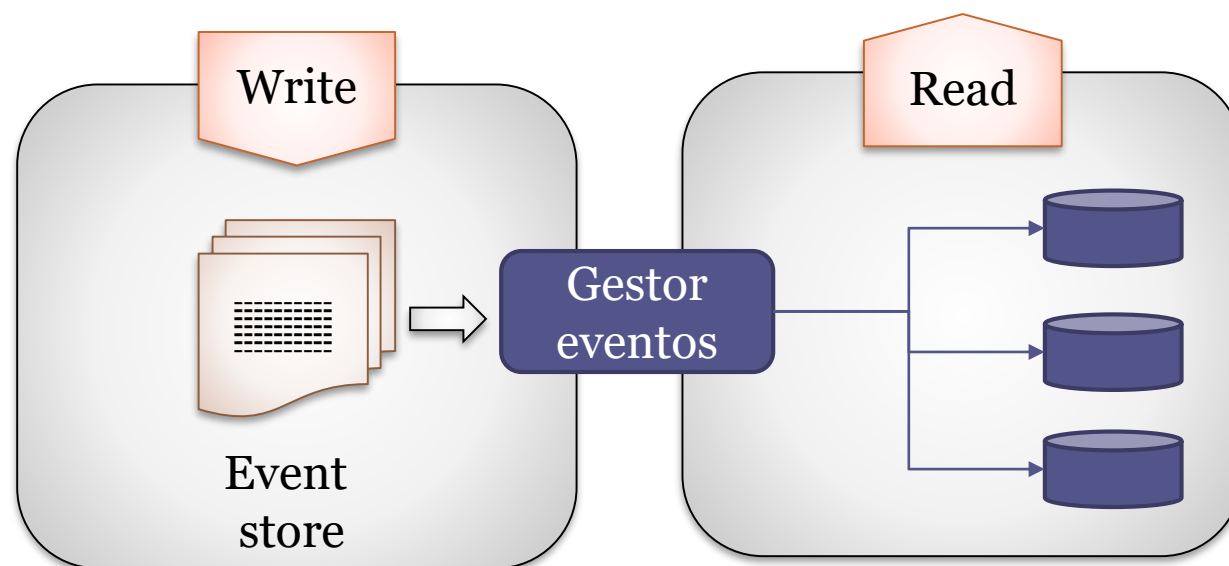
Elementos

Almacén de eventos

Almacena cambios en el estado

Eventos

No cambian, se enuncian en pasado



Event Sourcing

Ventajas

Tolerancia a fallos

Trazabilidad

Determinar estado de aplicación en cada momento

Reconstrucción

Si aparecen eventos erróneos, se pueden deshacer sus acciones y reconstruir el resto

Escalabilidad

BD de sólo append

Event Sourcing

Problemas/retos

Gestión de eventos

- Sincronización, consistencia

Complejidad de desarrollo

- Añade un nivel de indirección

Gestión de recursos

- Granularidad de los eventos

- Almacenamiento de eventos crece con tiempo

 - Requiere crear instantáneas (snapshots)

Event sourcing

Aplicaciones

Sistemas de bases de datos

Datomic, EventStore

Naked Objects

Objetos de dominio encapsulan ***toda*** la lógica de negocio

Interfaz de usuario = representación directa de objetos de dominio

Puede crearse automáticamente

Puede generarse API automáticamente

RESTful Objects

Naked Objects

Ventajas

- Adaptación al dominio

- Mantenimiento

Problemas/retos

- Difícil de adaptar Interfaz a casos especiales

Aplicaciones

- Naked Objects (.Net), Apache Isis (Java)