



# Arquitectura y Patrones para Aplicaciones Web -APAW-

## WebArchitecture

Jesús Bernal Bermúdez

# Arquitectura Web

## ¿Qué es la Arquitectura software?

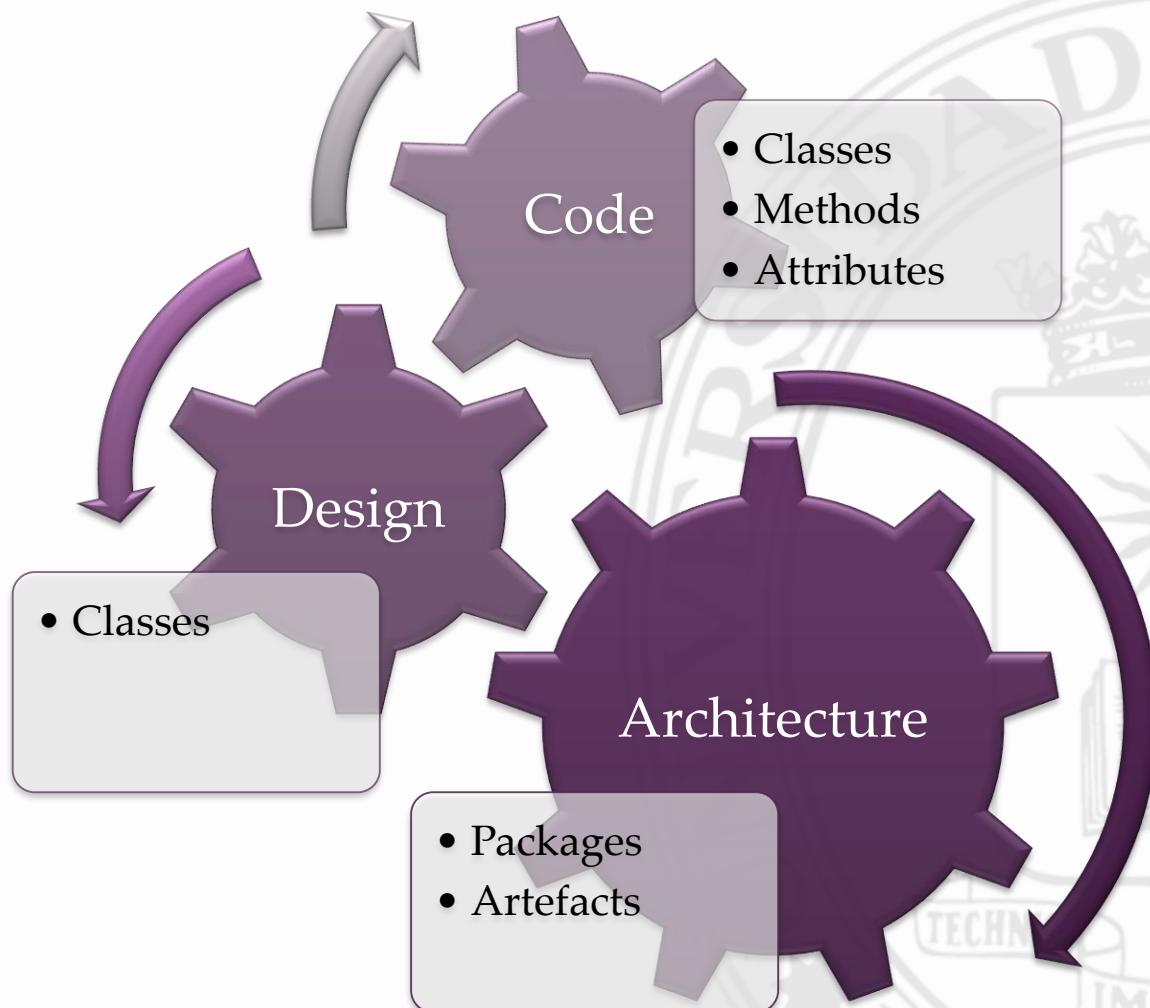


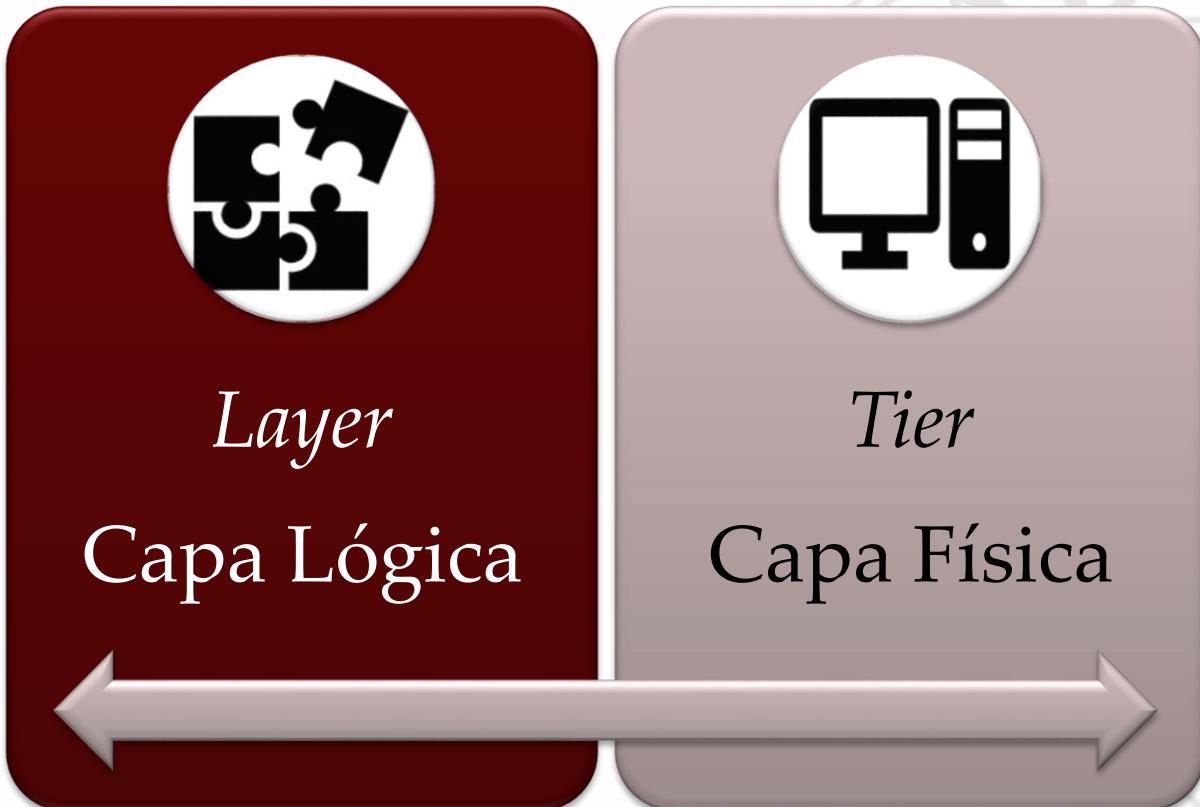
IEEE

Estructura fundamental de un sistema. Comprende sus **componentes**, sus **relaciones** con otros, su **entorno** y las **principales guías** de su diseño y evolución

# Arquitectura Web

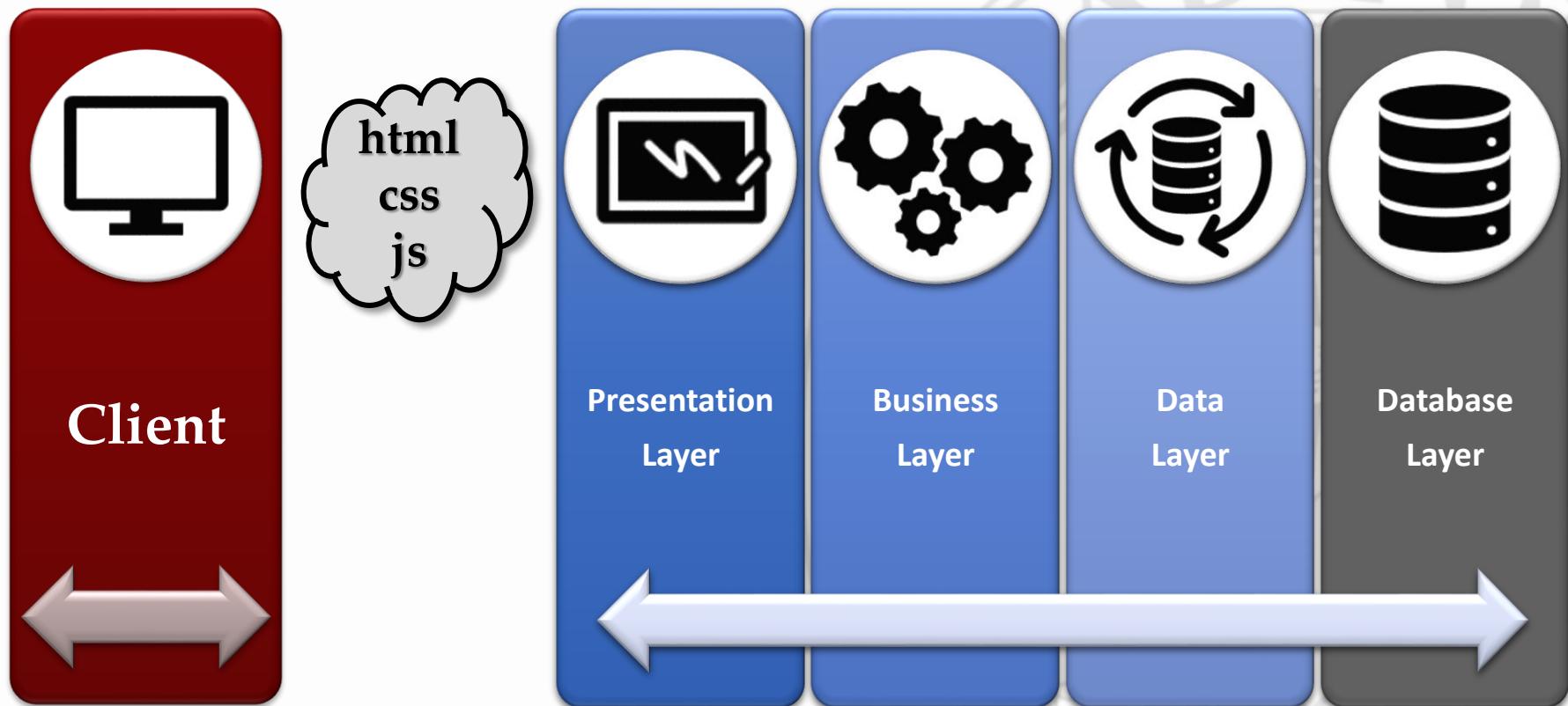
## Unidad de gestión





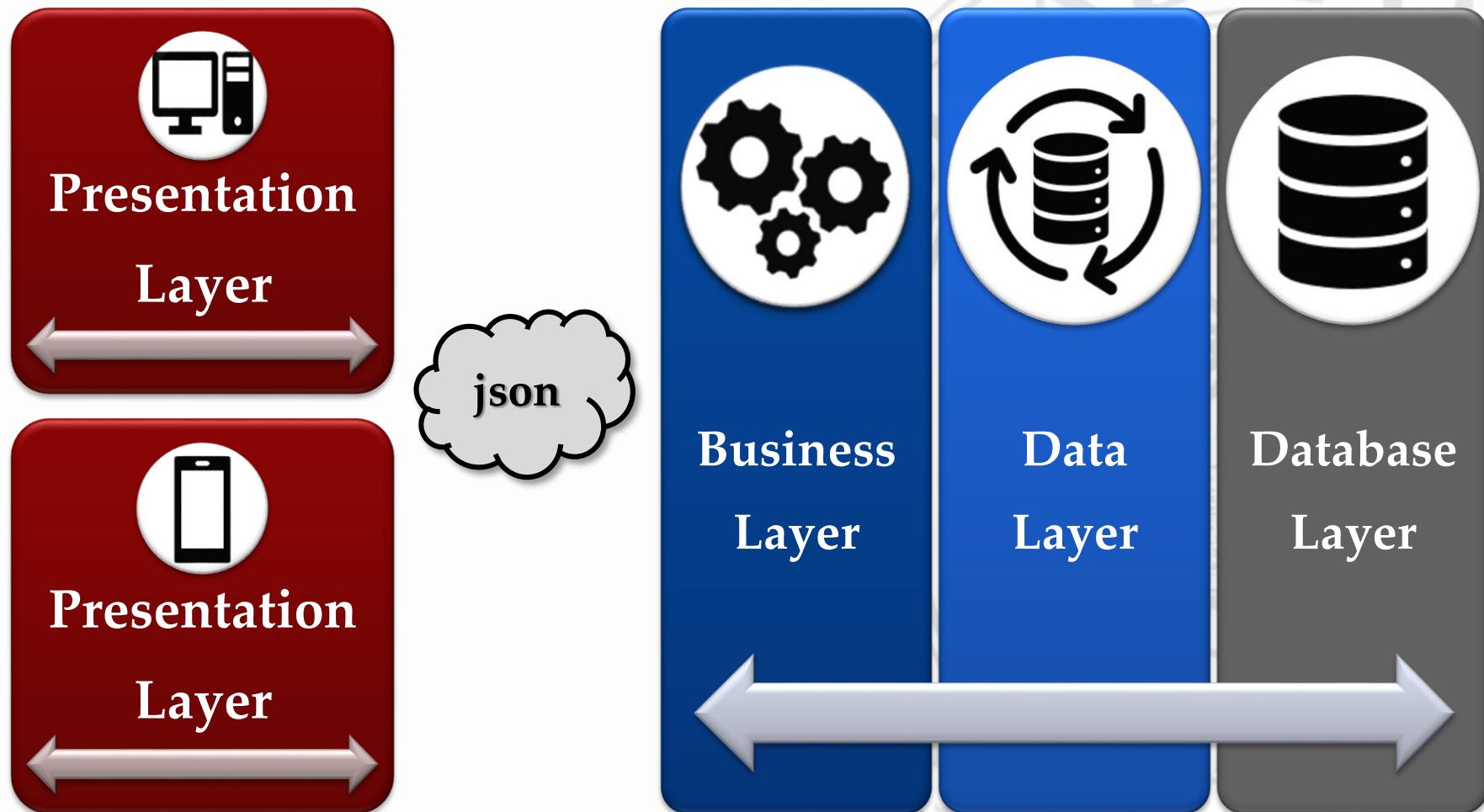
# Arquitectura por Capas

## Aplicación de Múltiples Páginas



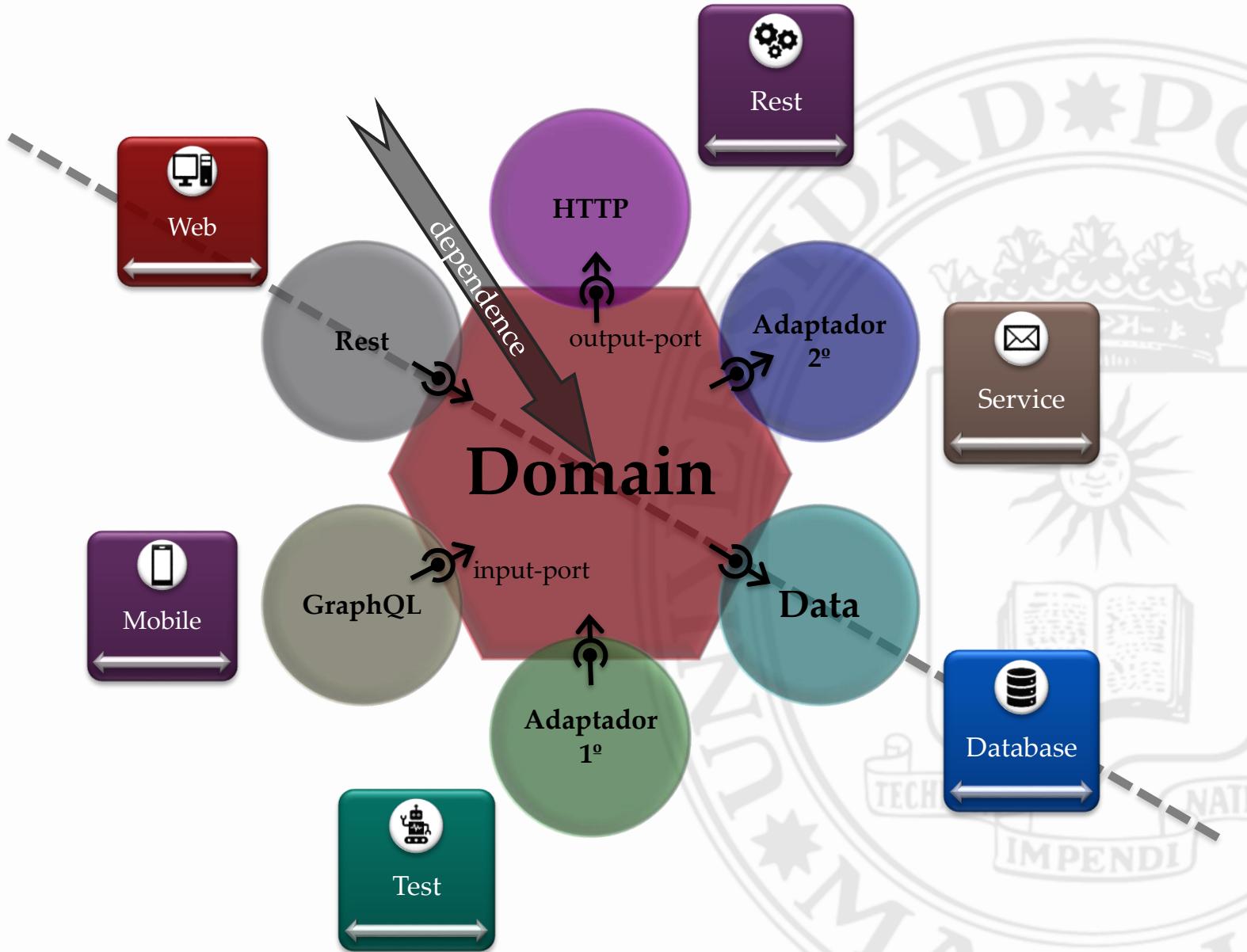
# Arquitectura por Capas

## Aplicación de Página Única (*SPA*)



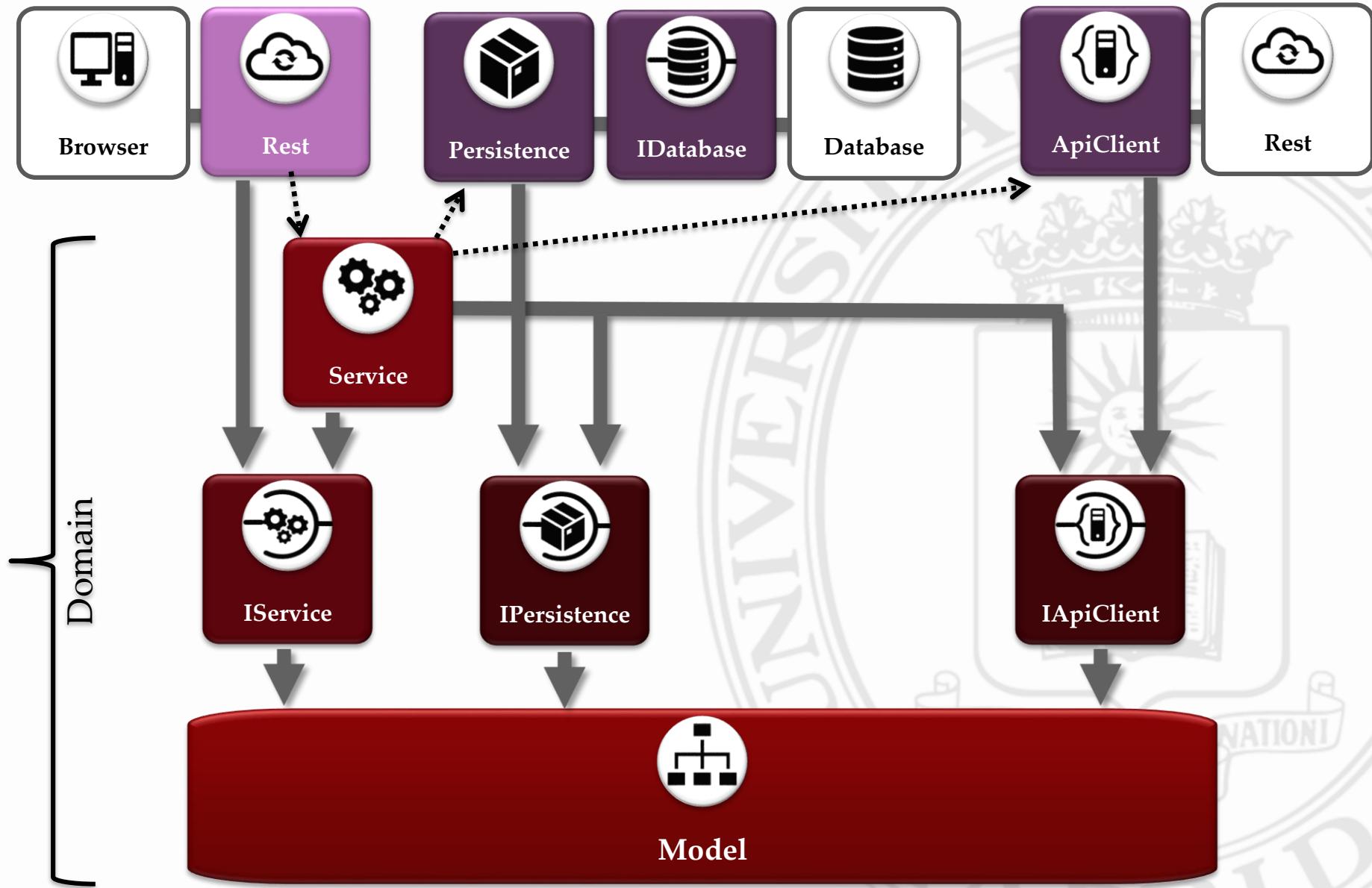
# Hexagonal Architecture

## Alistair Cockburn



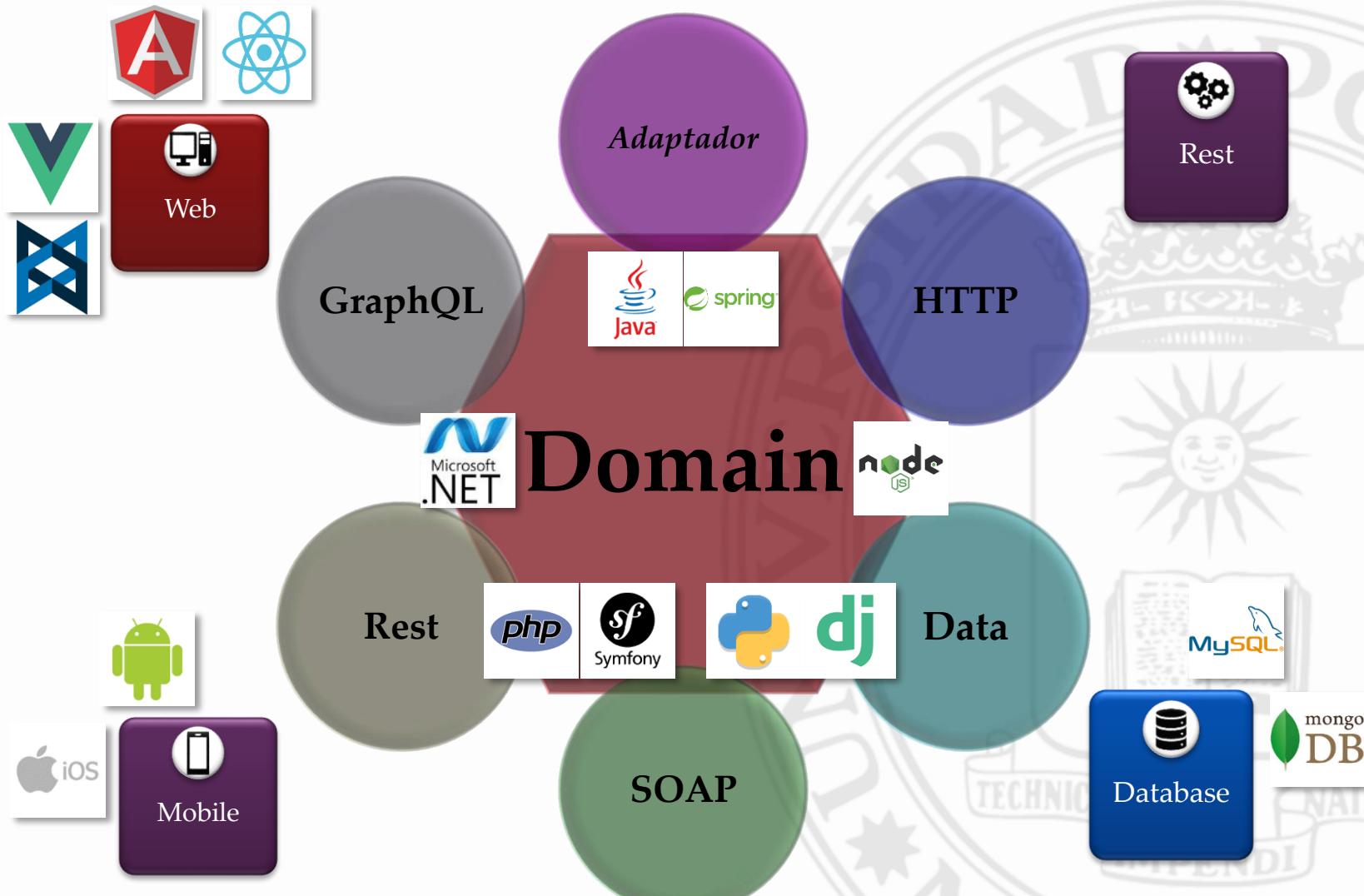
# Hexagonal Architecture

## Dependencias



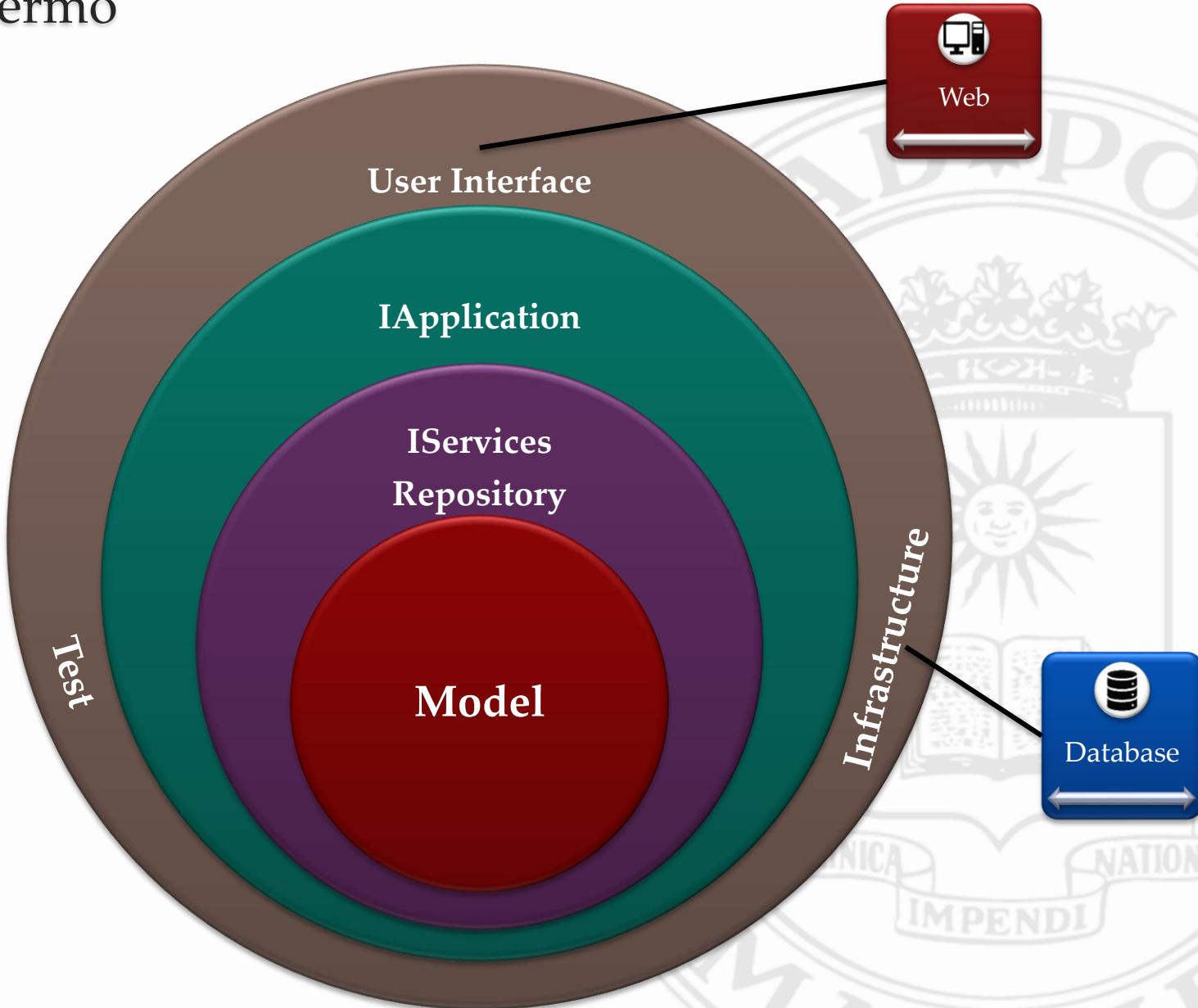
# Arquitectura Hexagonal

## Alistair Cockburn



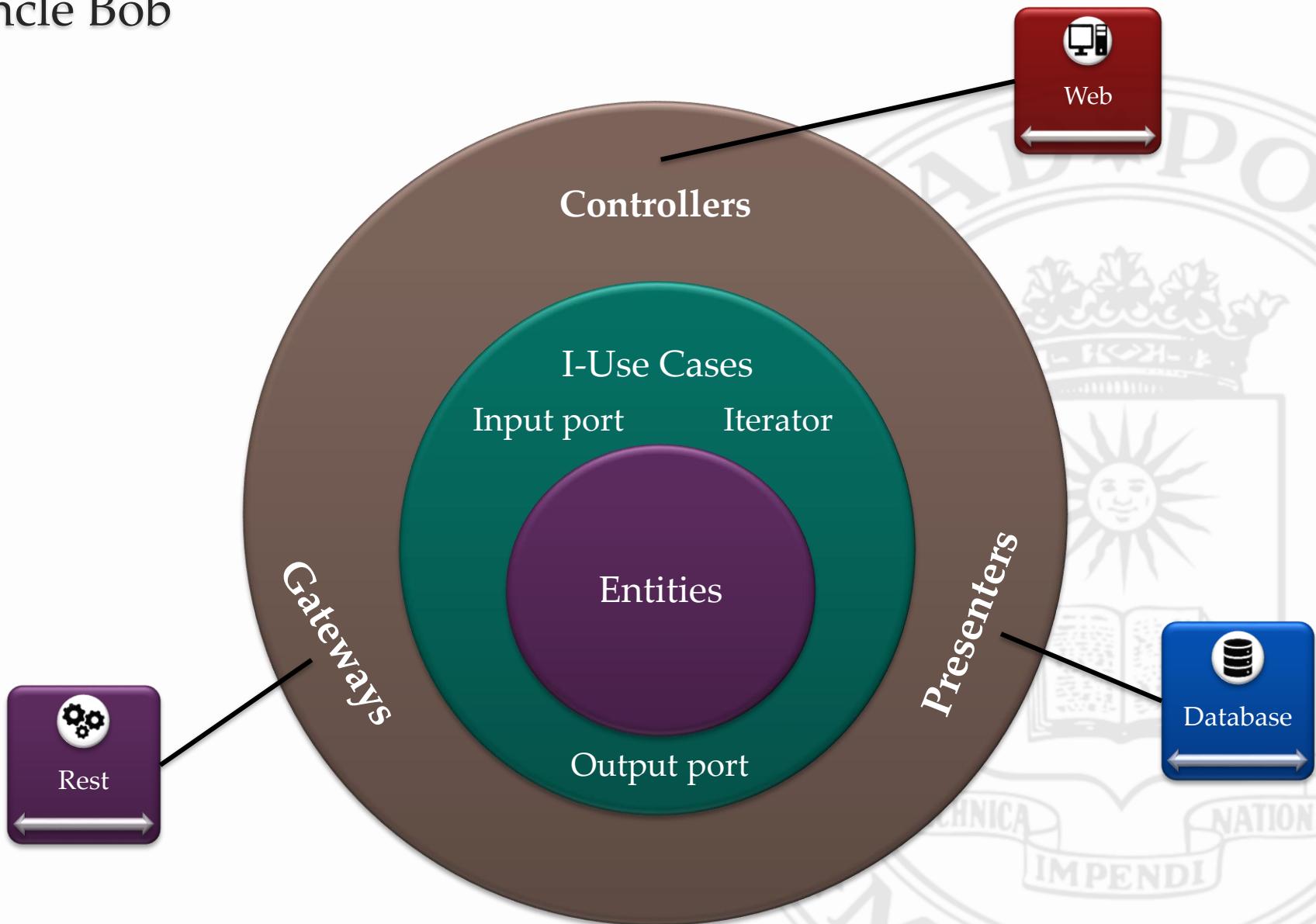
# Onion Architecture

## Jeffrey Palermo



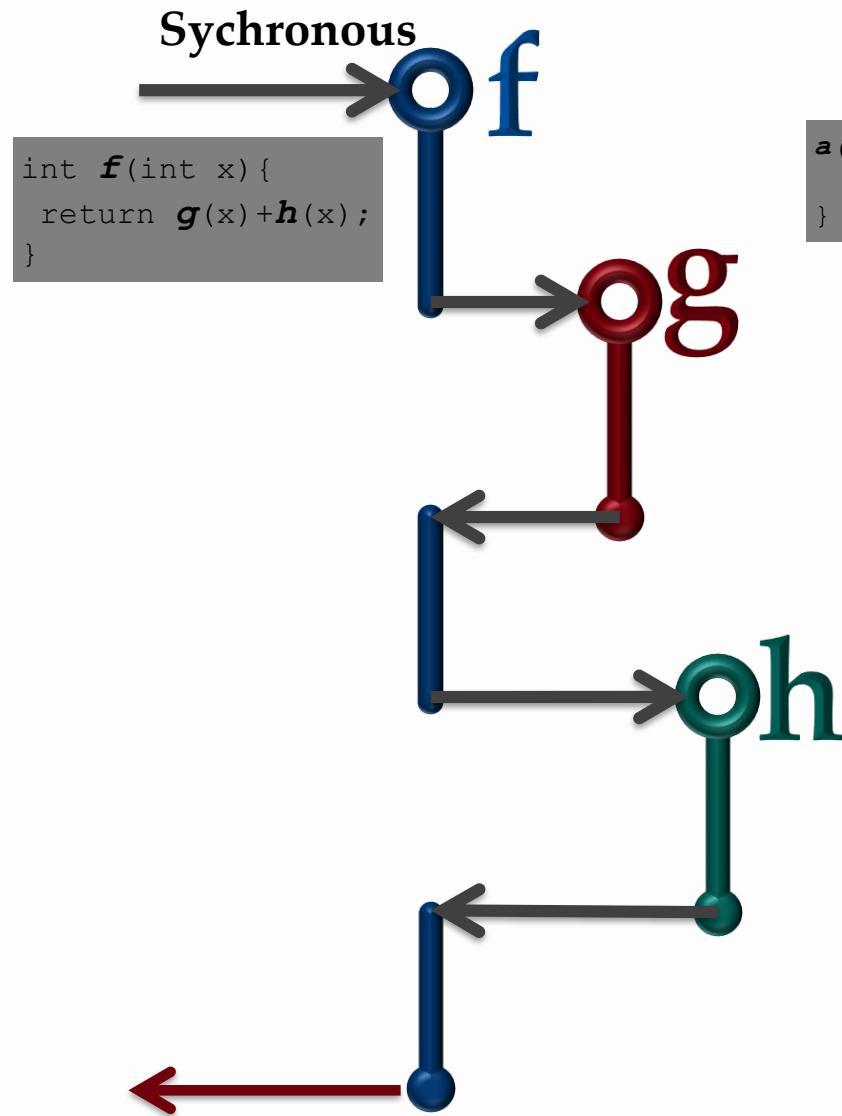
# Clean Architecture

## Uncle Bob



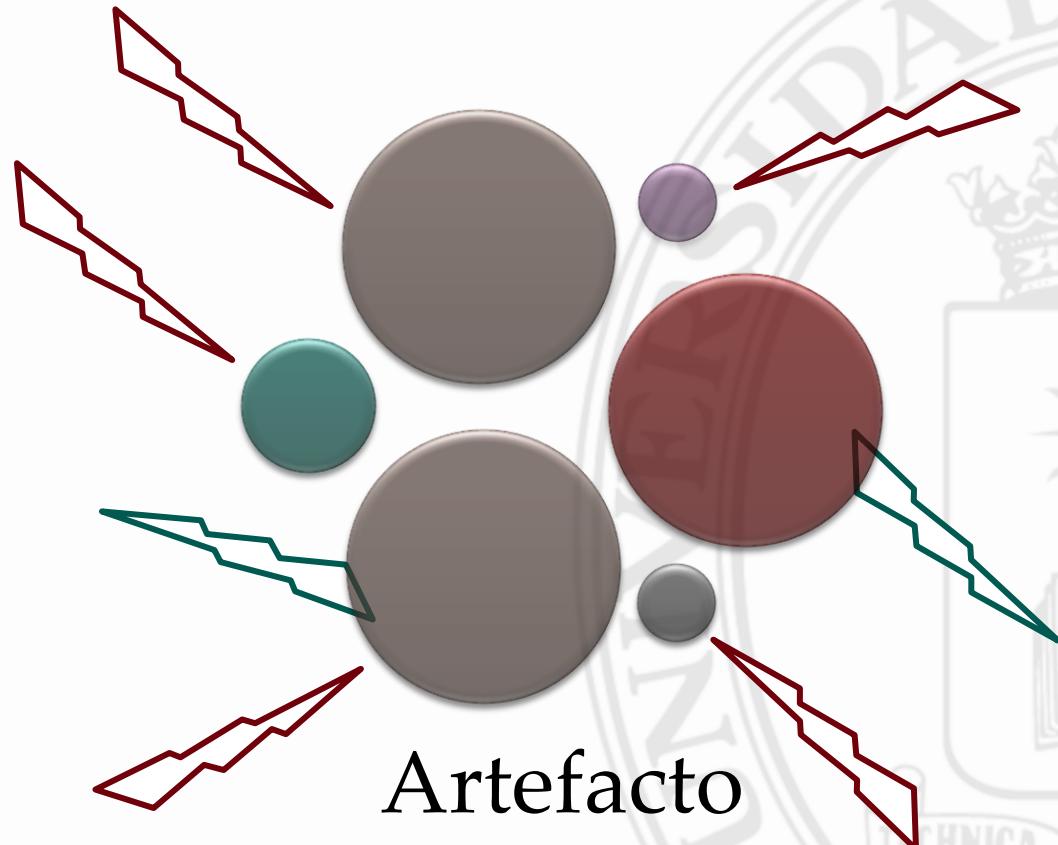
# Arquitectura dirigida por Eventos

## Síncrono - Asíncrono

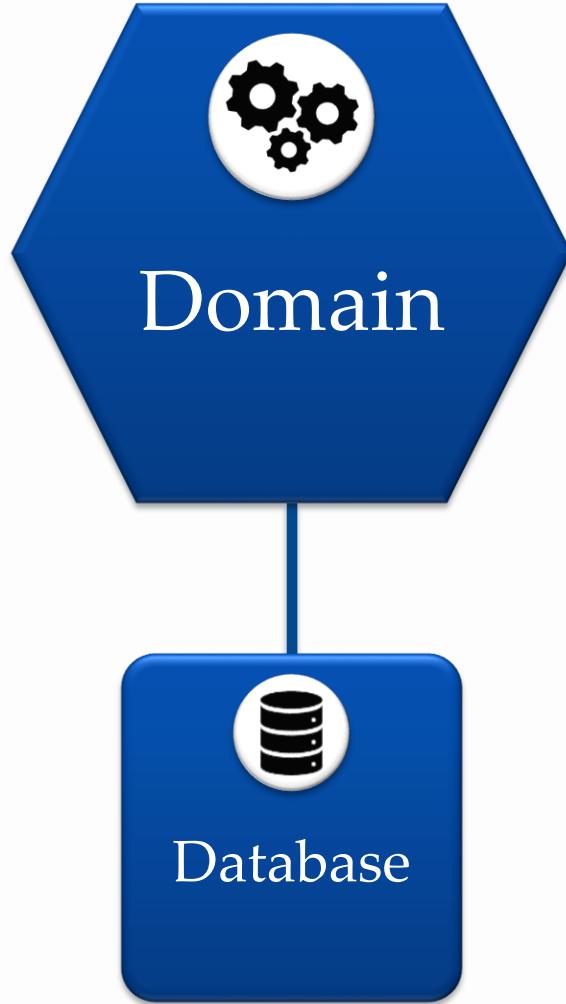


# Arquitectura dirigida por Eventos

*Event Driven Architecture*



# Aplicación Monolítica



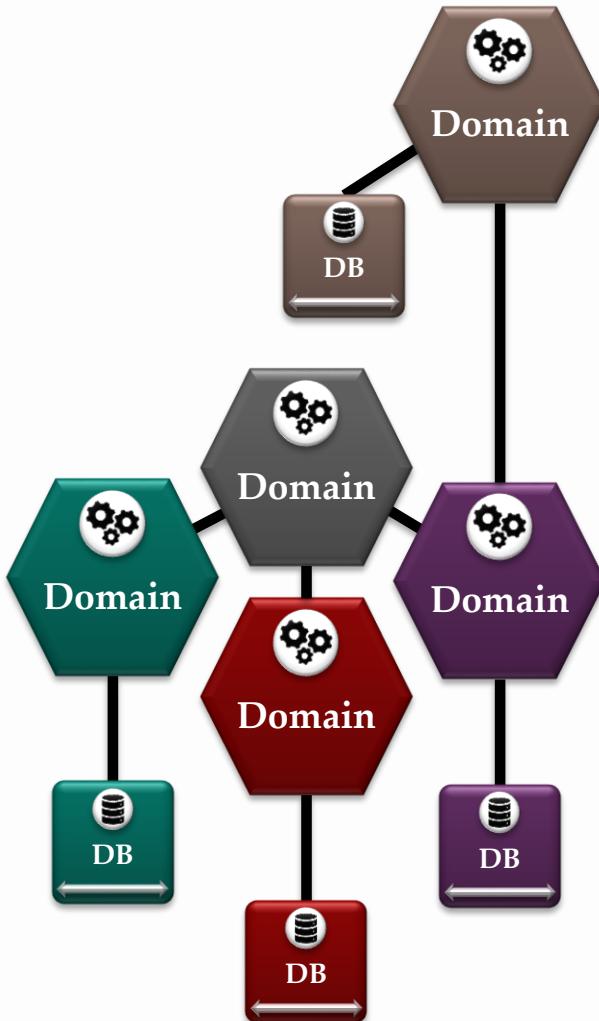
## Ventajas

- Un solo artefacto: fácil despliegue
- Fácil gestión de versiones

## Inconvenientes

- Complejidad aumenta en el tiempo: perdida de producción
- Grandes equipos de desarrollo: especialización y complejidad en la interacción entre los equipos
- Replicación costosa con difícil balanceo de carga
- Grandes BD: migraciones complejas
- Difícil reusabilidad de partes del proyecto
- Una sola tecnología y difícil migración tecnológica

# Microservicios



## Ventajas

- **Fuertemente modular.**

- Permite tener equipos más pequeños, multidisciplinarios.
- Filosofía de productos y no proyectos.
- Descentralización de BD y elimina la integración de BD.

- **Despliegues independientes.**

- Permite la evolución rápida y con menor riesgo.
- Replicar despliegues y balanceo de carga.
- Reusabilidad en diferentes proyectos.

- **Diversidad tecnológica.**

- Permite cambiar de tecnología en la evolución del proyecto con nuevos servicios.

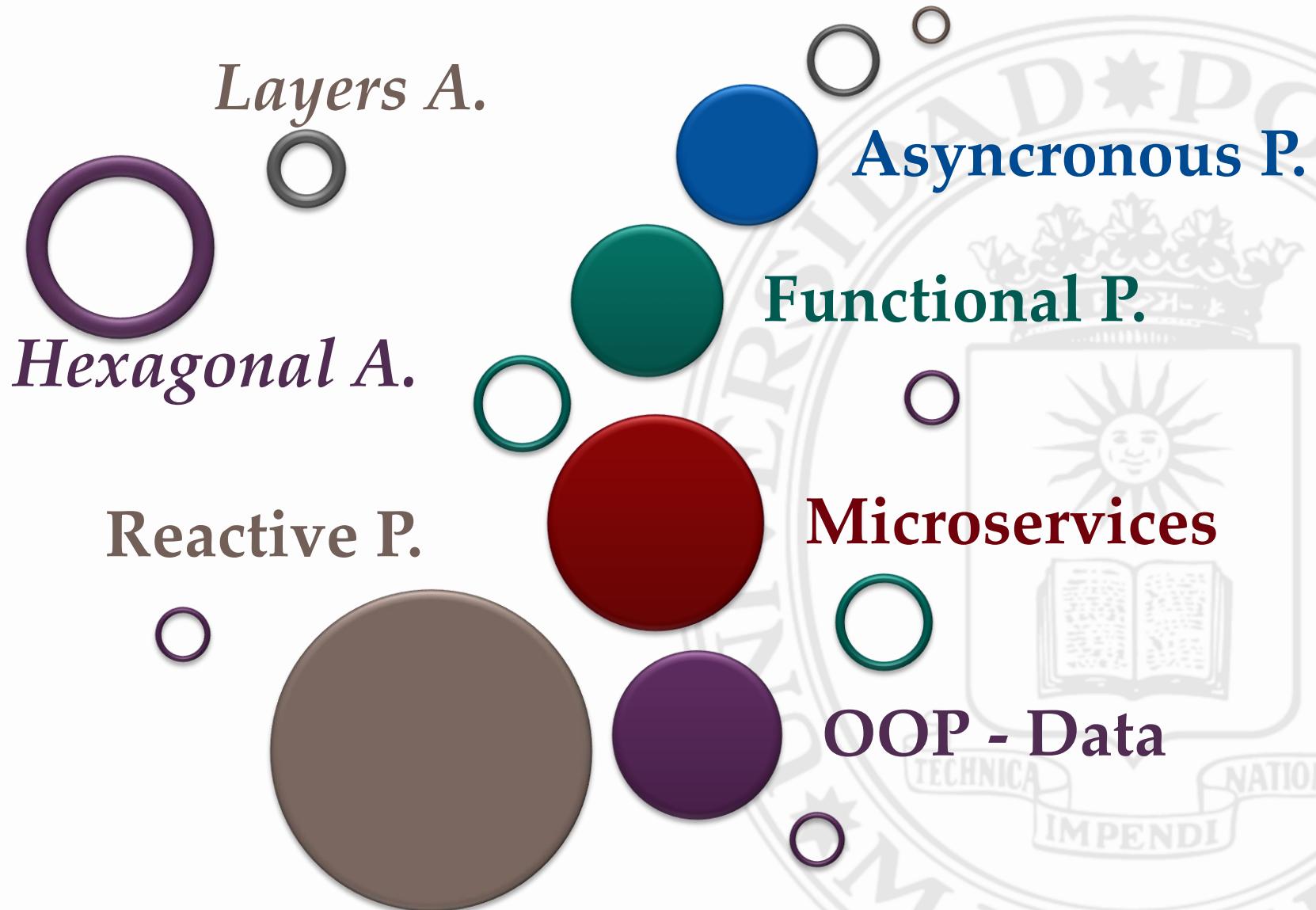
## Inconvenientes

- **Distribución.** Al ser un sistema distribuido, es más compleja su despliegue

- **Consistencia.** Es más complicado mantener la consistencia del sistema

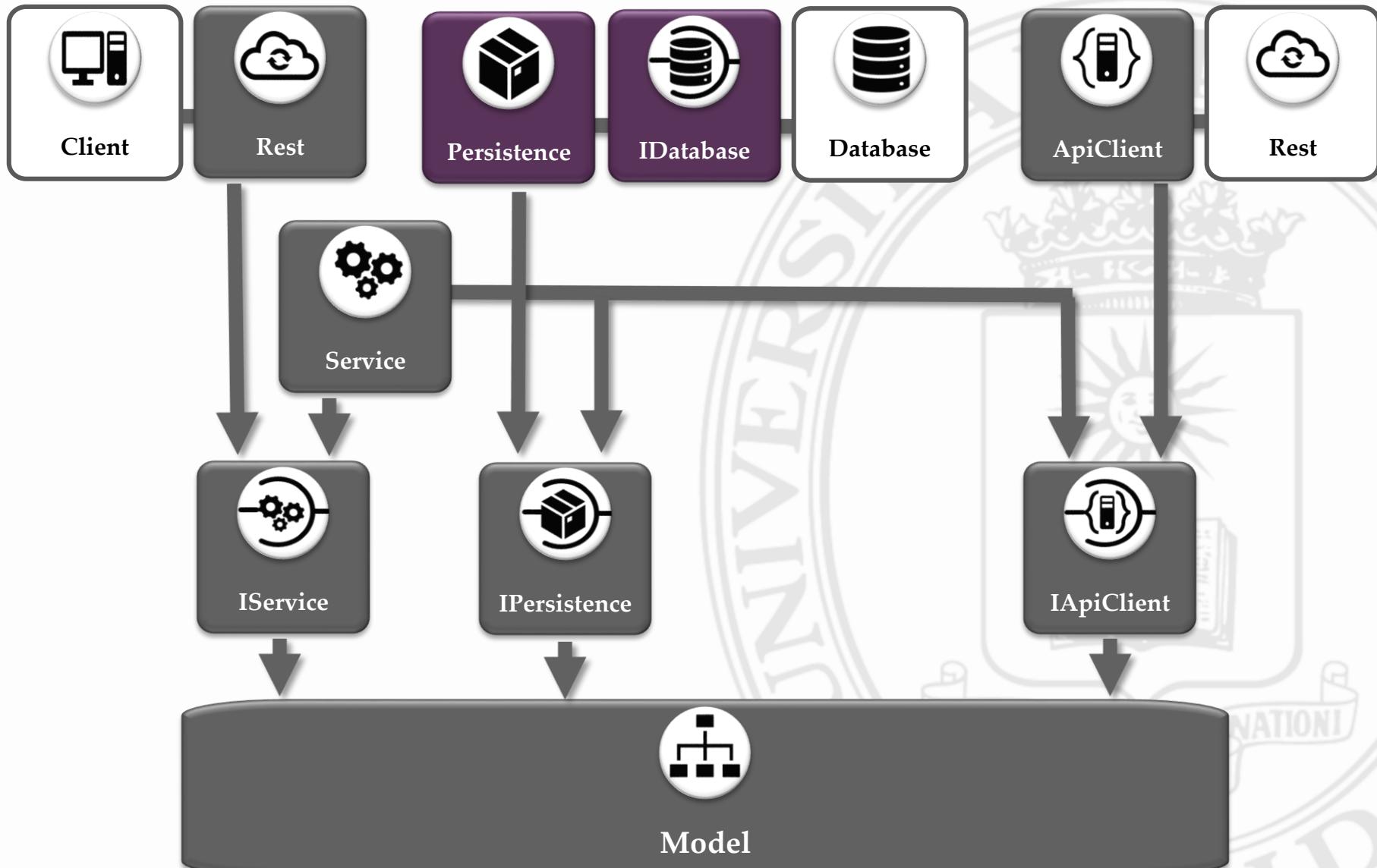
- **Complejidad operacional.** Se necesita acceder a varios servicios para realizar una operación

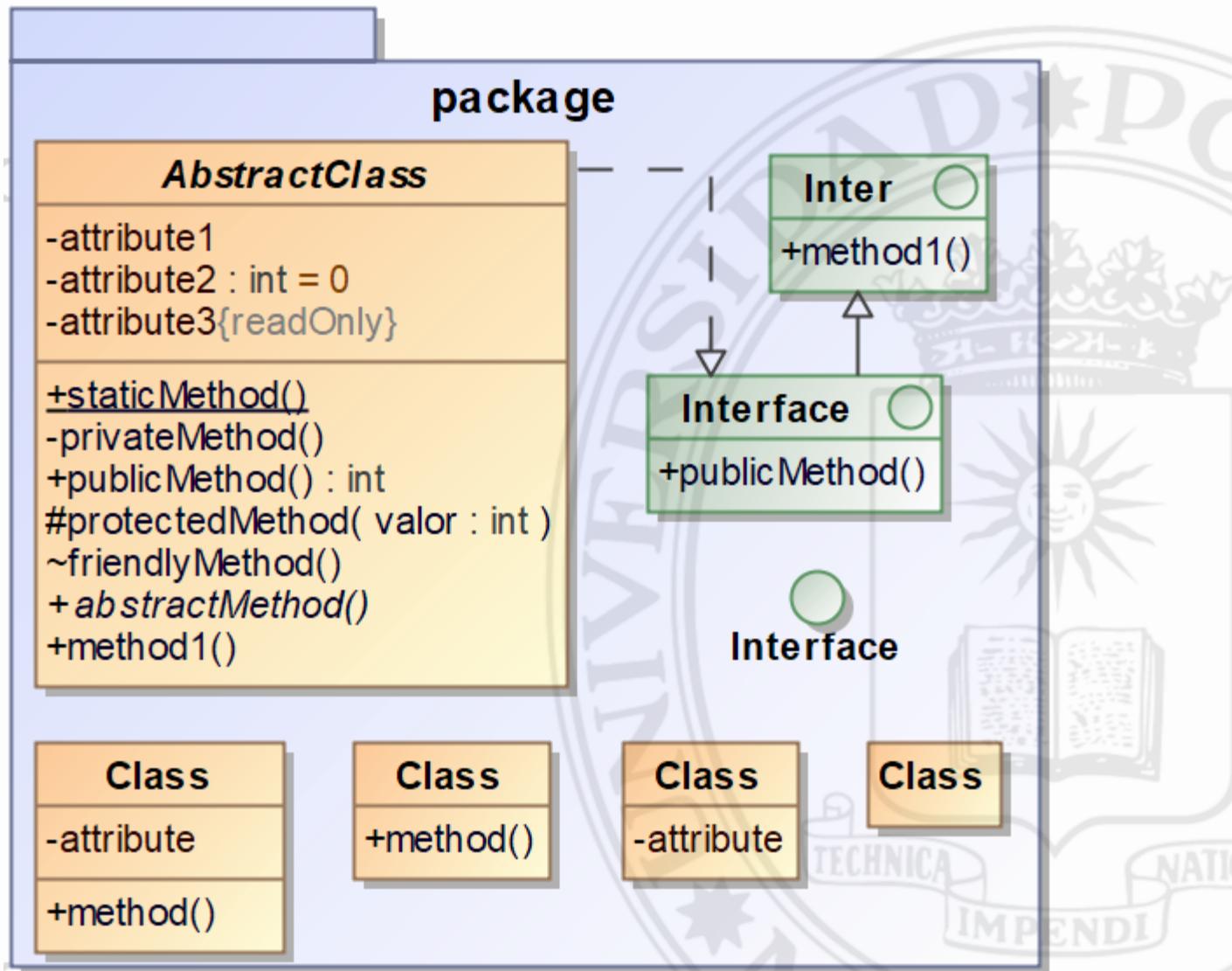
# Tendencias...

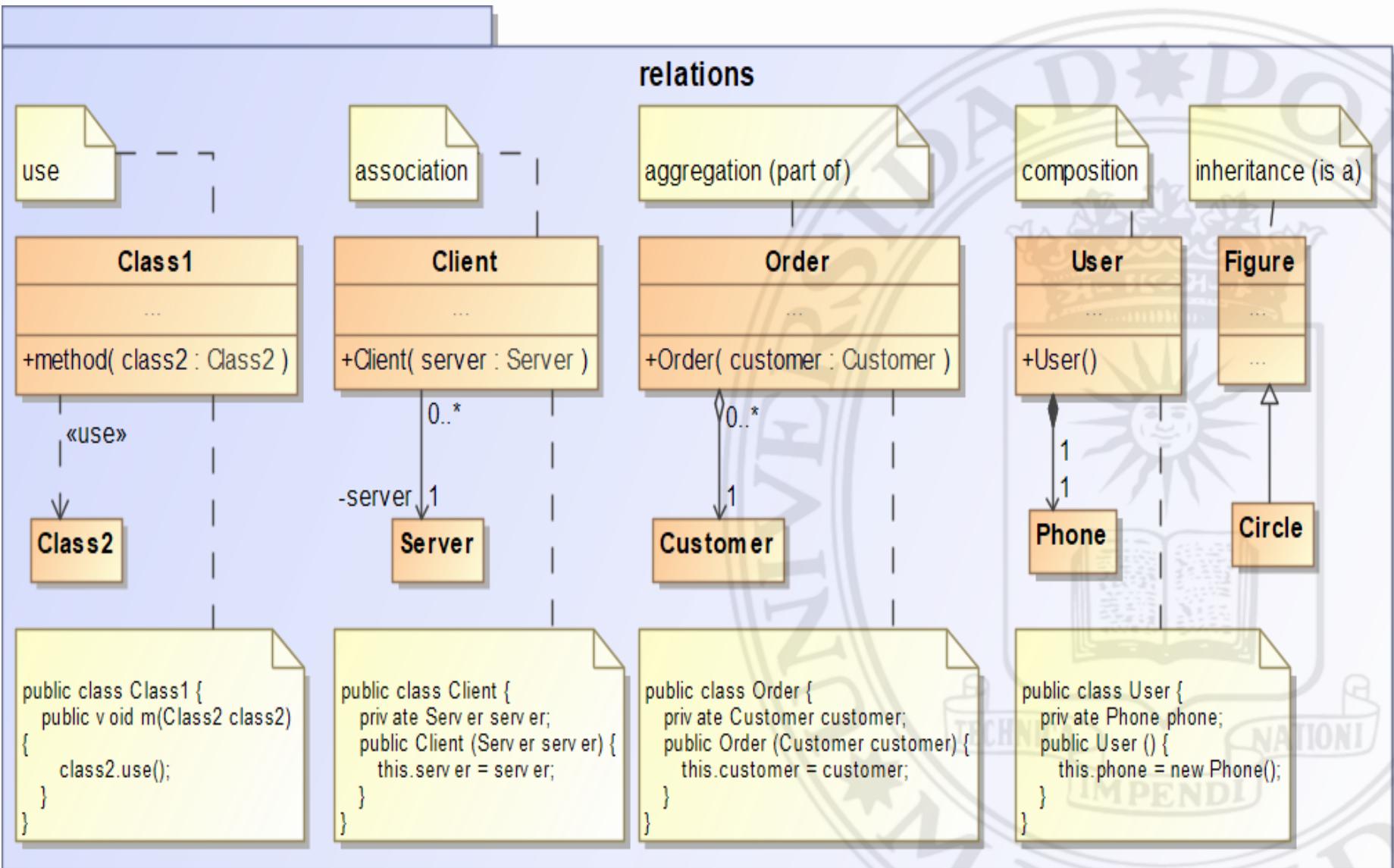


# Persistencia

## Adaptador de Bases de Datos

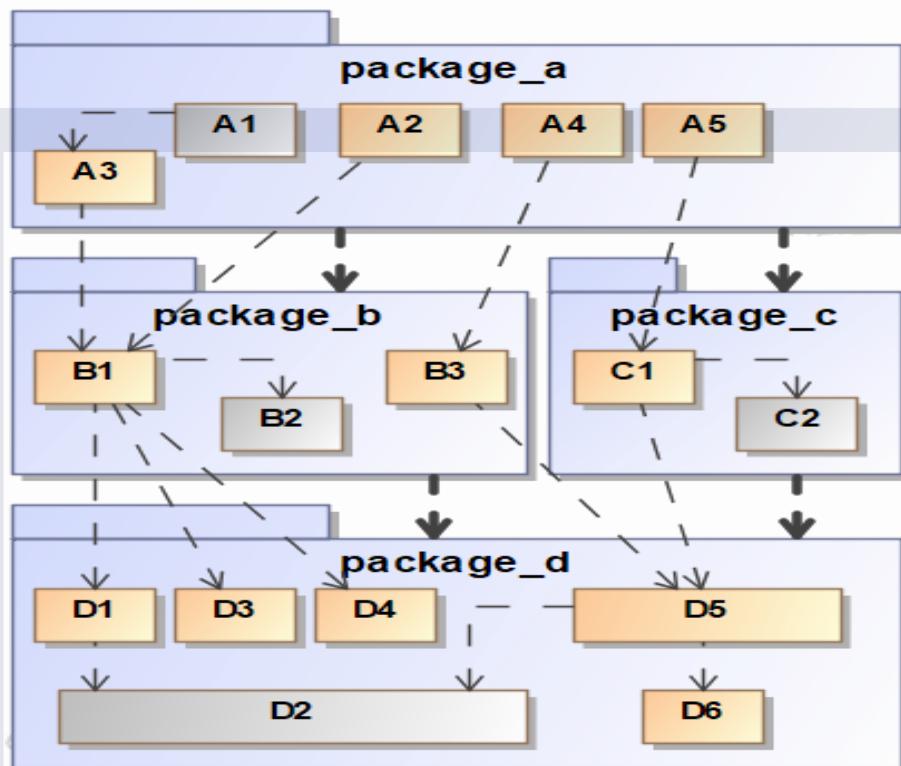






# Data

## Principios de Robert Martin



Dependencias

### Principio de Equivalencia entre Reusable y Entregable

- Se reúsa si no se mira el código fuente.

### Principio de Reusabilidad Común

- Las clases de un paquete se reutilizan juntas.

### Principio de Cierre Común

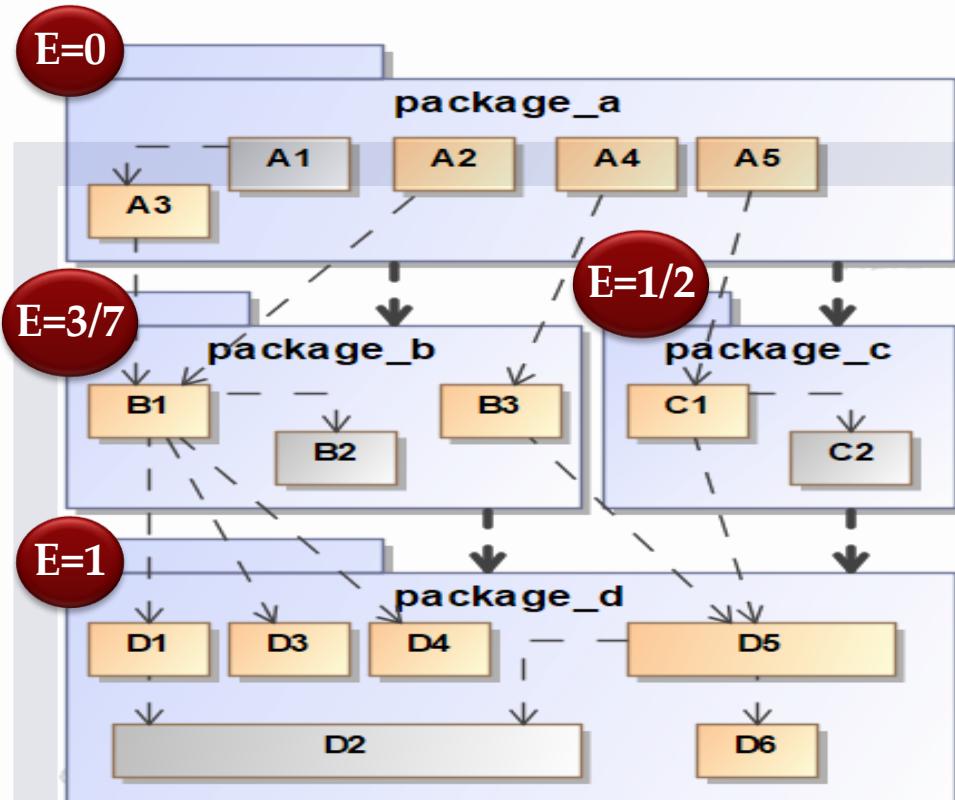
- Un cambio que afecte a una clase, afecta a todo el paquete.

### Principio de dependencias acíclicas

- Ciclo de dependencias sin bucle

# Data

## Principios de Robert Martin

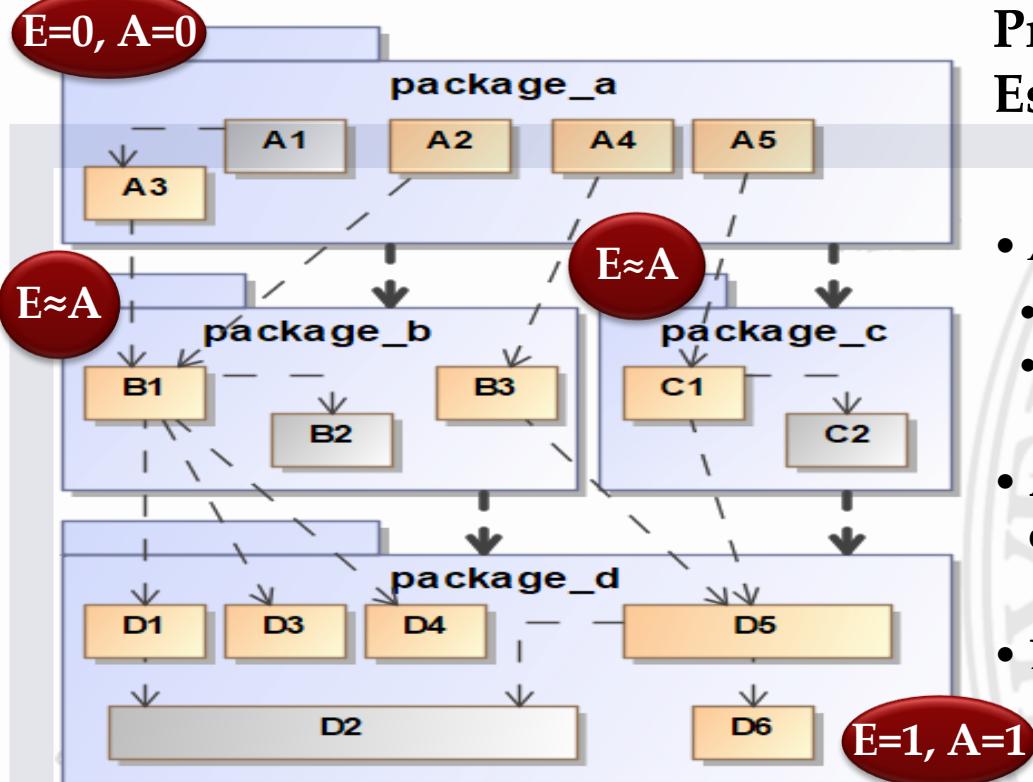


### Principio de Dependencias Estables

- Estable en el sentido de pesado, difícil de cambiar.
- Inestable en el sentido de ligero, fácil de cambiar.
- **Estabilidad** =  $\frac{Aa}{Aa+Ae} \rightarrow [1..0]$ , 1 es totalmente estable, 0 nada estable, es decir, inestable.
  - **Aa** - Acoplamiento aferente: N° de clases de afuera que dependen del paquete.
  - **Ae** - Acoplamiento eferente: el paquete depende de un N° de clases de fuera.
- **Inestabilidad** =  $1 - \text{Estabilidad}$

# Data

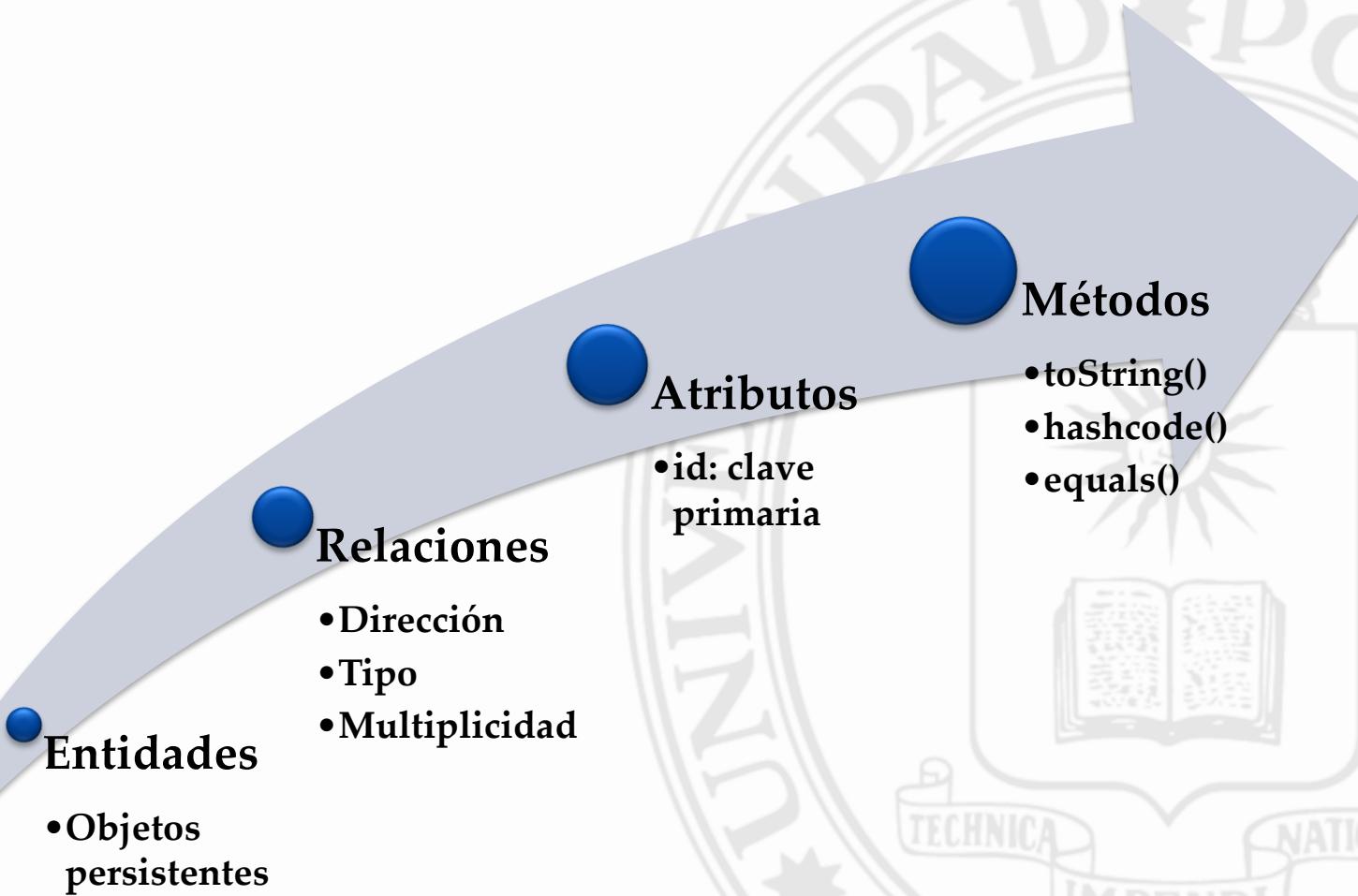
## Principios de Robert Martin

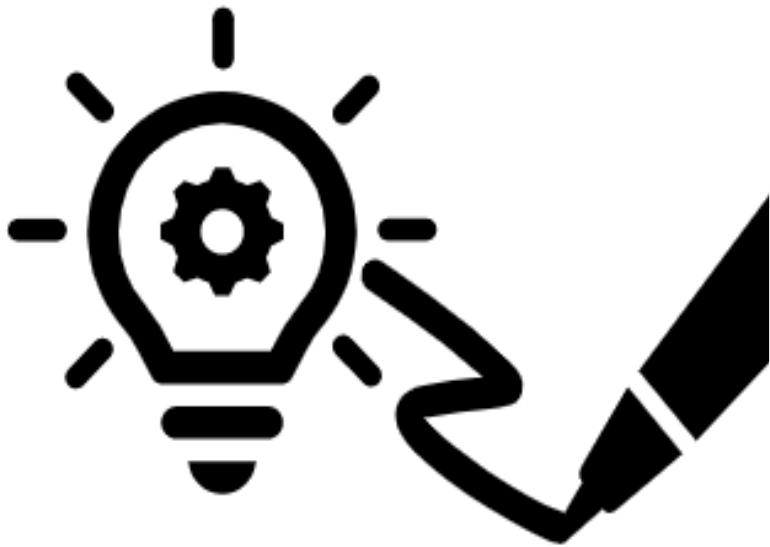


### Principio de Abstracciones Estables

- $\text{Abstracción} = \frac{Ca}{Ct}$ 
  - Ca = N° de clases abstractas
  - Ct = N° total de clases
- Abstracción debe ser  $\geq$  que la anterior, en cada capa
- Estabilidad  $\approx$  Abstracción

Dependencias





Biblioteca

Nos piden modelizar una **biblioteca**.

Existen libros, con su título, su ISBN y una lista de autores. Los libros están asociados a un conjunto de temas.

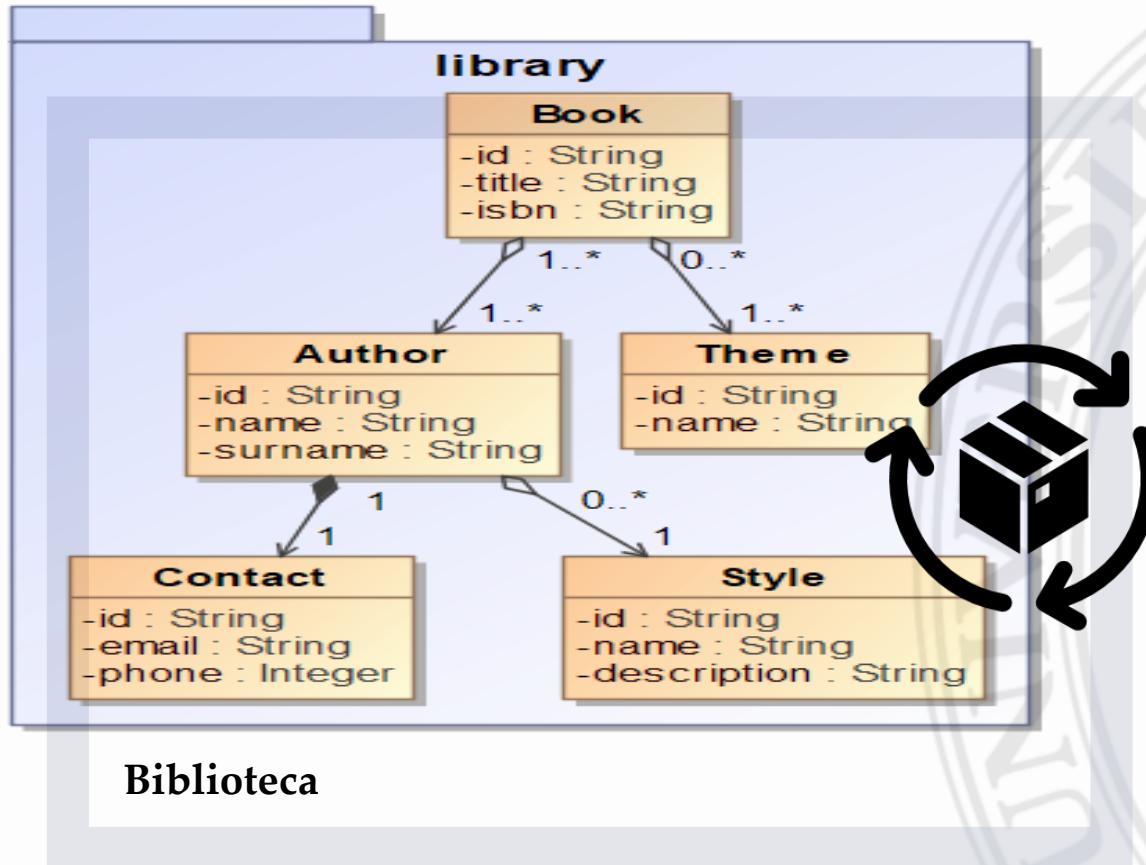
Un tema tiene un nombre y un tema puede estar asociado a muchos libros.

De cada autor queremos guardar su nombre y su apellido y sus datos de contacto, que están formados por un email y un teléfono.

Un autor tiene un estilo, pero un estilo puede ser utilizado por muchos autores. De un estilo se guarda el nombre y la descripción.

El cliente quiere poder gestionar libros, los autores...

# Data Modelización

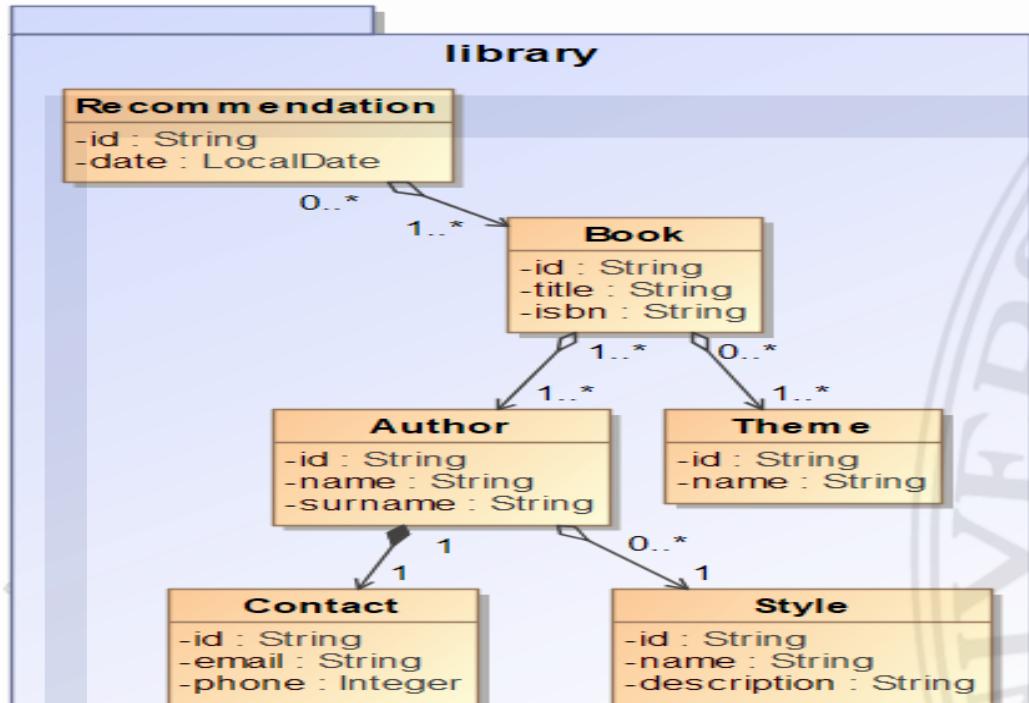


Nos piden modelizar una biblioteca.

Existen libros, con su título, su ISBN ...

Después de LIBERAR (en producción) la primera versión, el cliente nos pide ahora que en un libro se indique si está recomendado. El pretende recomendar cada mes una serie de libros.

# Data Modelización

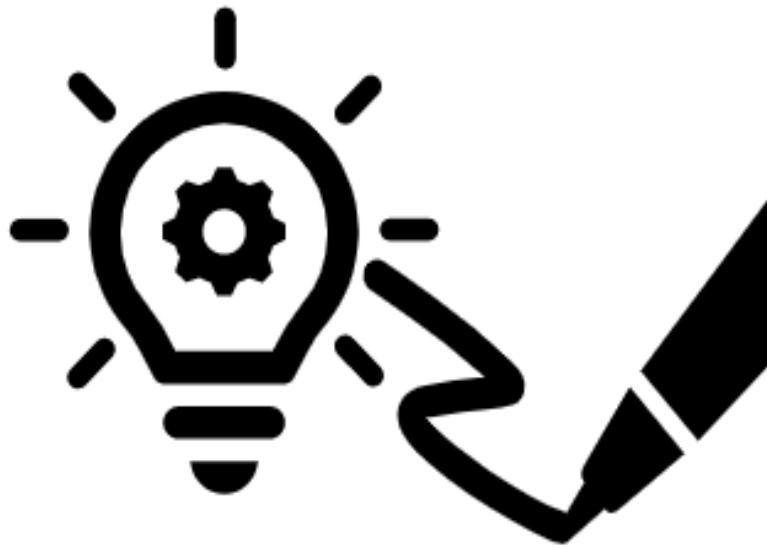


Biblioteca

Nos piden modelizar una biblioteca.

Existen libros, con su título, su ISBN ...

Después de liberar la primera versión, el cliente nos pide ahora que en un libro se indique si está recomendado. El pretende recomendar cada mes una serie de libros.



## Ejercicio

Nos piden modelizar:

- Company
- Doctor
- Car Hire
- Hotel
- Sport
- Shop
- Veterinary Clinic
- Department
- Movie
- ...

Con 4 clases y un total de 16 atributos.  
Proyecto y clases únicas para todos los grupos.

# Mapeo de objetos

## Mapeo Objeto-Relacional - ORM

- Es una técnica para convertir objetos a una base de datos relacional.

## Mapeo Objeto-Dокументo - ODM

- Convierte objetos a una base de datos tipo JSON.

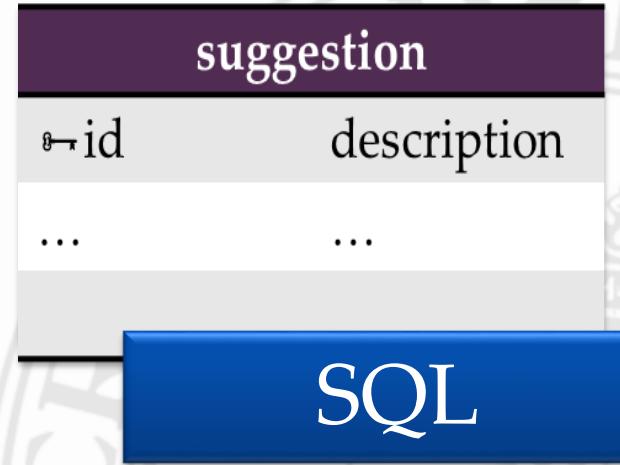
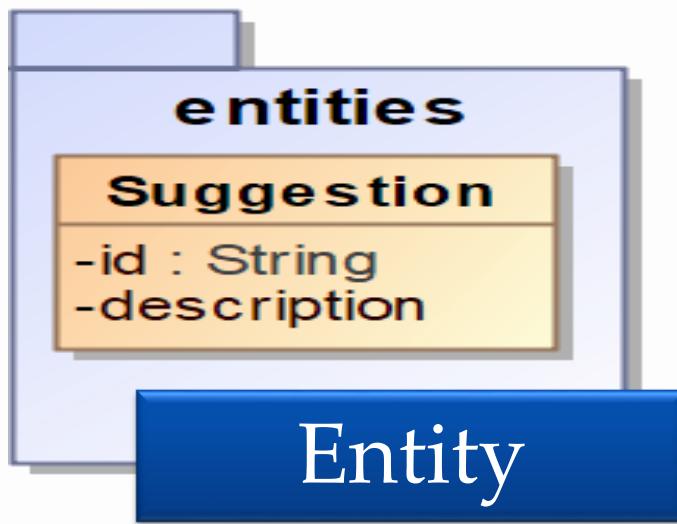
## Mapeo

- Diferentes opciones...
- Rendimiento de memoria.
- Rendimiento de ejecución.

## Dependencias

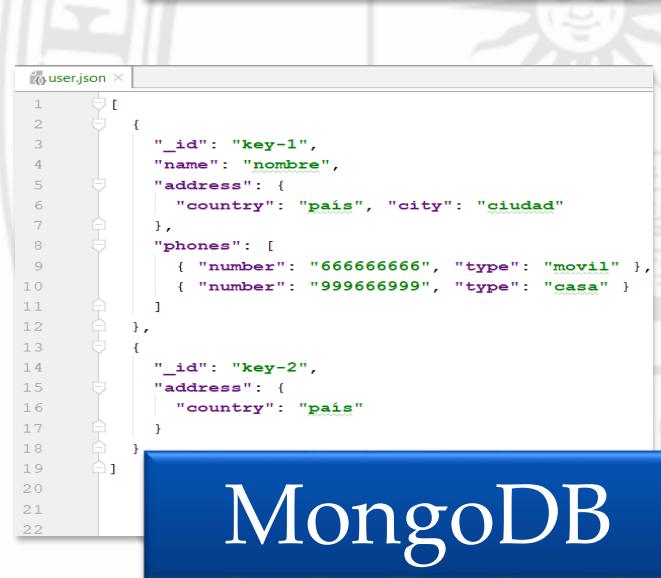
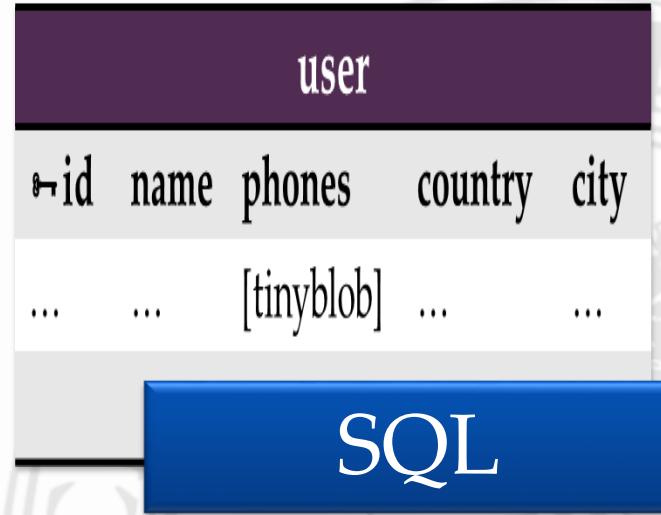
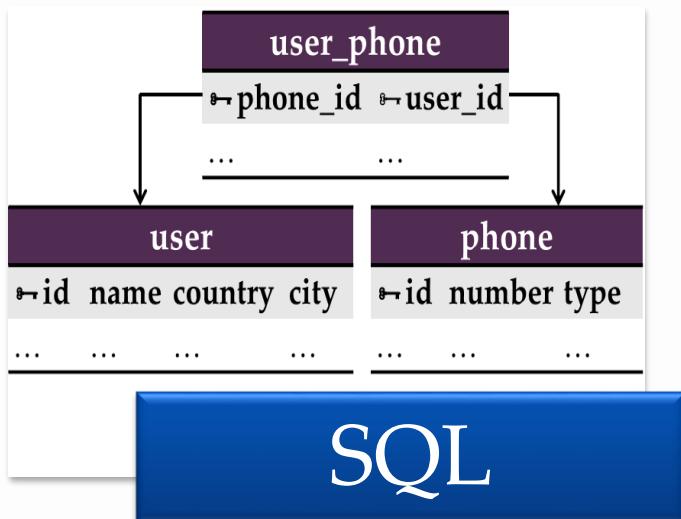
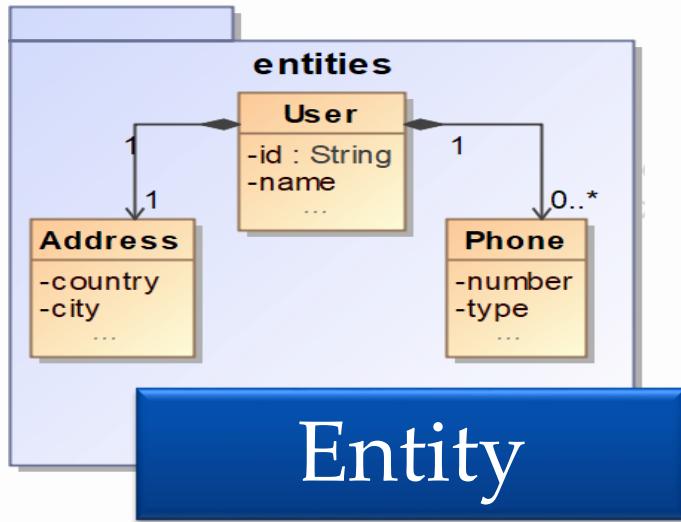
- No se debe depender del tipo de Bases de Datos

# Mapeo de objetos: sin relación

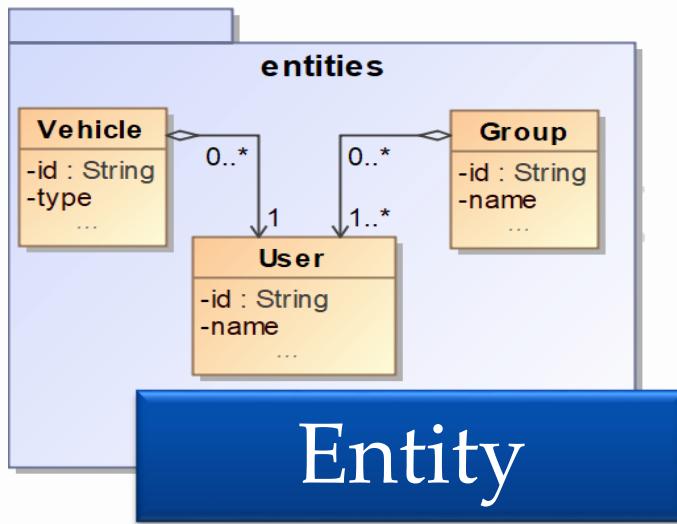


MongoDB

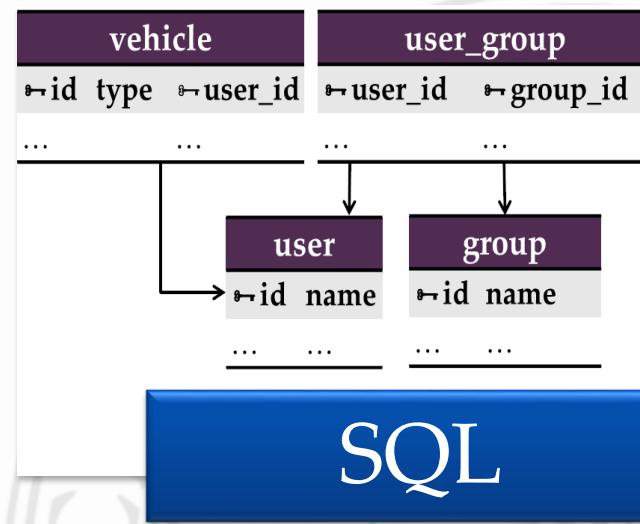
# Mapeo de objetos: composición



# Mapeo de objetos: agregación



## Entity



SQL

```
user.json ×
1   [
2     {
3       "_id": "key-1",
4       "name": "nombre",
5       "address": {
6         "country": "pais", "city": "ciudad"
7       },
8       "phones": [
9         { "number": "6666666666", "type": "movil" },
10        { "number": "999666999", "type": "casa" }
11      ]
12    },
13    {
14      "_id": "key-2",
15      "address": {
16        "country": "pais"
17      }
18    }
19  ]
20
21
```

# MongoDB

```
vehicle.json x
1   {
2     "_id" : "key-1",
3     "type" : "tipo",
4     "user" : DBRef("user", "user-key-1")
5   }

group.json x
1   {
2     "_id" : "key-1",
3     "name" : "nombre",
4     "users" : [
5       DBRef("user", "user-key-1"),
6       DBRef("user", "user-key-2")
7     ]
8   }
9
10
```

# MongoDB

## Singleton

- Se garantiza que una clase sólo tenga una instancia y se proporciona un acceso global a ella.

## Builder

- Separa la construcción de objeto complejo de su representación, permitiendo diferentes construcciones.

## Composite

- Permite estructuras en árbol tratando por igual a las hojas que a los elementos compuestos.

## Method Factory

- Define una abstracción para crear objetos, y son las subclases que deciden la clase concreta a instanciar.

## Abstract Factory

- Proporciona un interface para crear familias de objetos relacionadas.

## Inversion of Control

- Desacoplar mediante la inyección de dependencias.

## Data Access Object - DAO

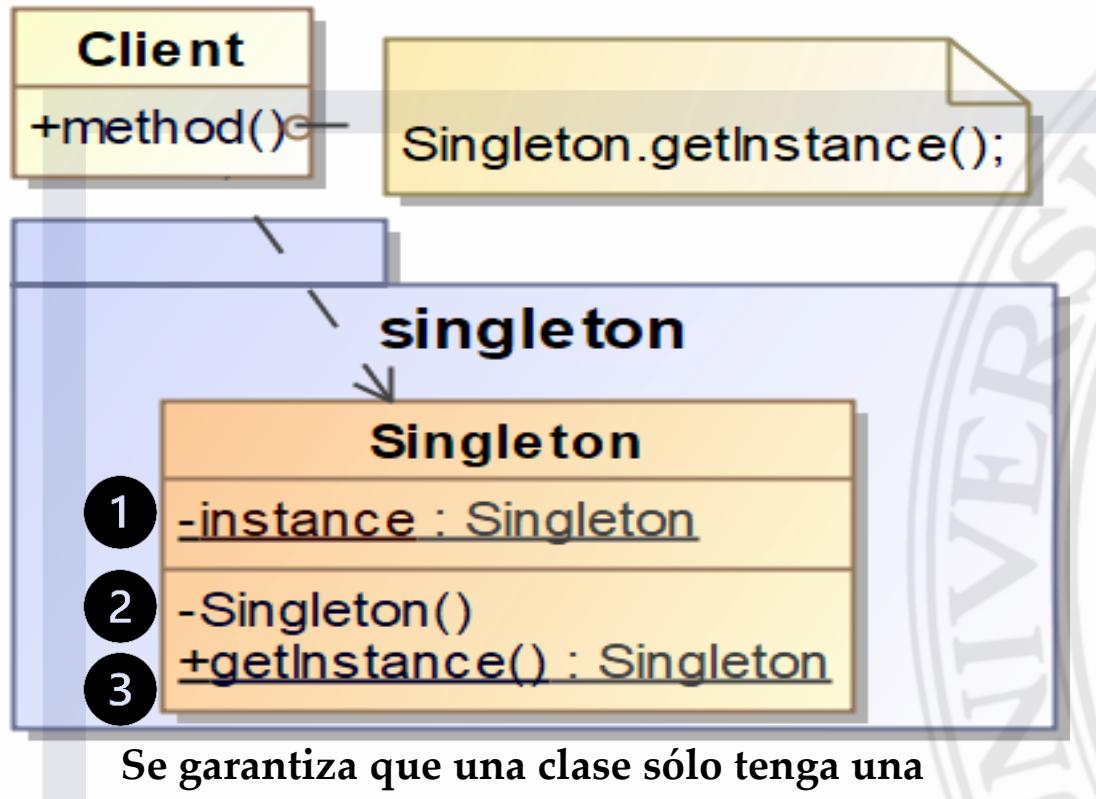
- Desacoplar el Dominio de la aplicación del Adaptador de las Bases de Datos.

## *Spring-Data*

- Framework que implementa el patrón DAO

# Singleton

Propósito: Creación. Ámbito: objeto



① Atributo privado estático de la propia clase.

- Creación temprana (*eager*).

② Constructor privado.

③ Método estático y público para devolver el atributo.

- Creación perezosa (*lazy*)

# Singleton

## Patrón Único

### logger

#### Logger

-logs : String

+Logger()

+addLog( log : String )

+getLogs() : String

+clear()

Logger

### solution

#### Logger

-logger : Logger

-logs : String

-Logger()

+getLogger() : Logger

+addLog( log : String )

+getLogs() : String

+clear()

Logger & Singleton

## factory

### ReferencesFactory

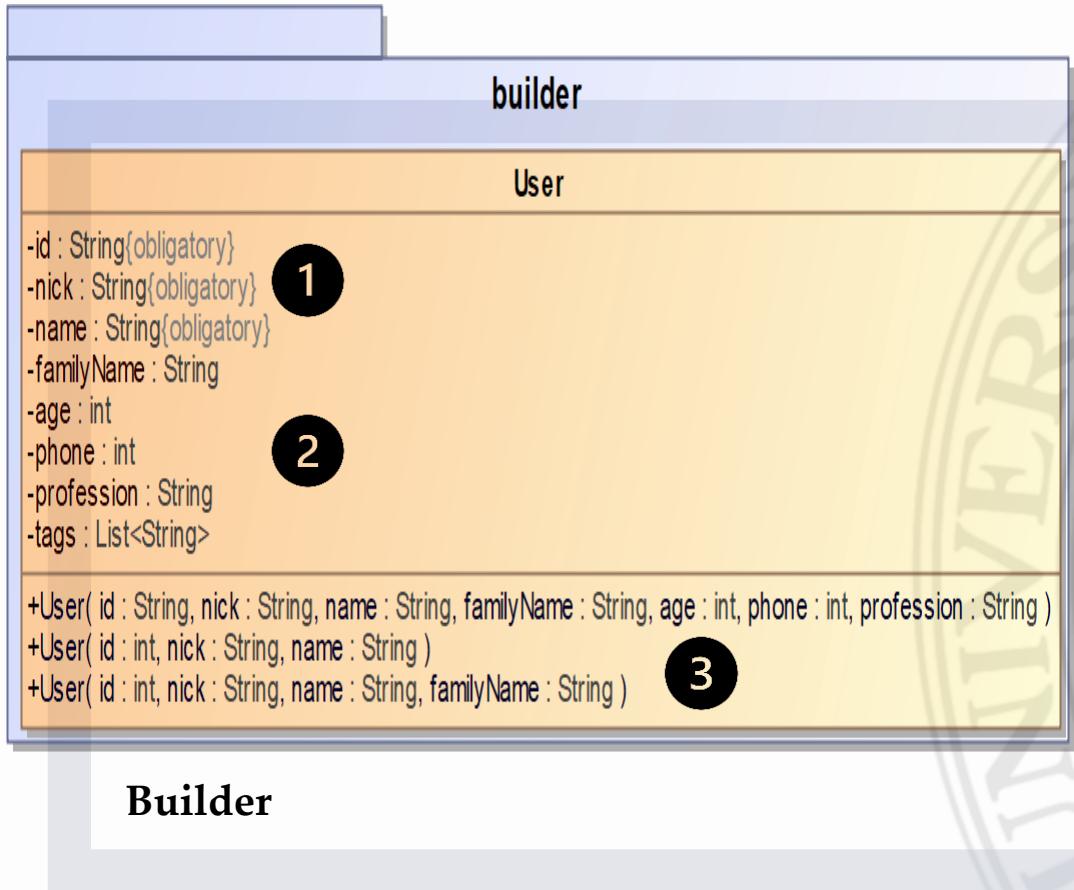
-references : Map<String, Integer>  
-reference : int

+ReferencesFactory()  
+getReference( key : String ) : int  
+removeReference( key : String )

Aplicar el patrón *Singleton* a *ReferencesFactory*, con creación temprana

# Builder

Propósito: Creación. Ámbito: objeto



```
User user = new User("id1","Paco","Jose","De Miguel",25,666666666,"Profesor",
    Arrays.asList("Director", "Socio", "Consejo"));
```

① Atributos obligatorios.

② Atributos opcionales.

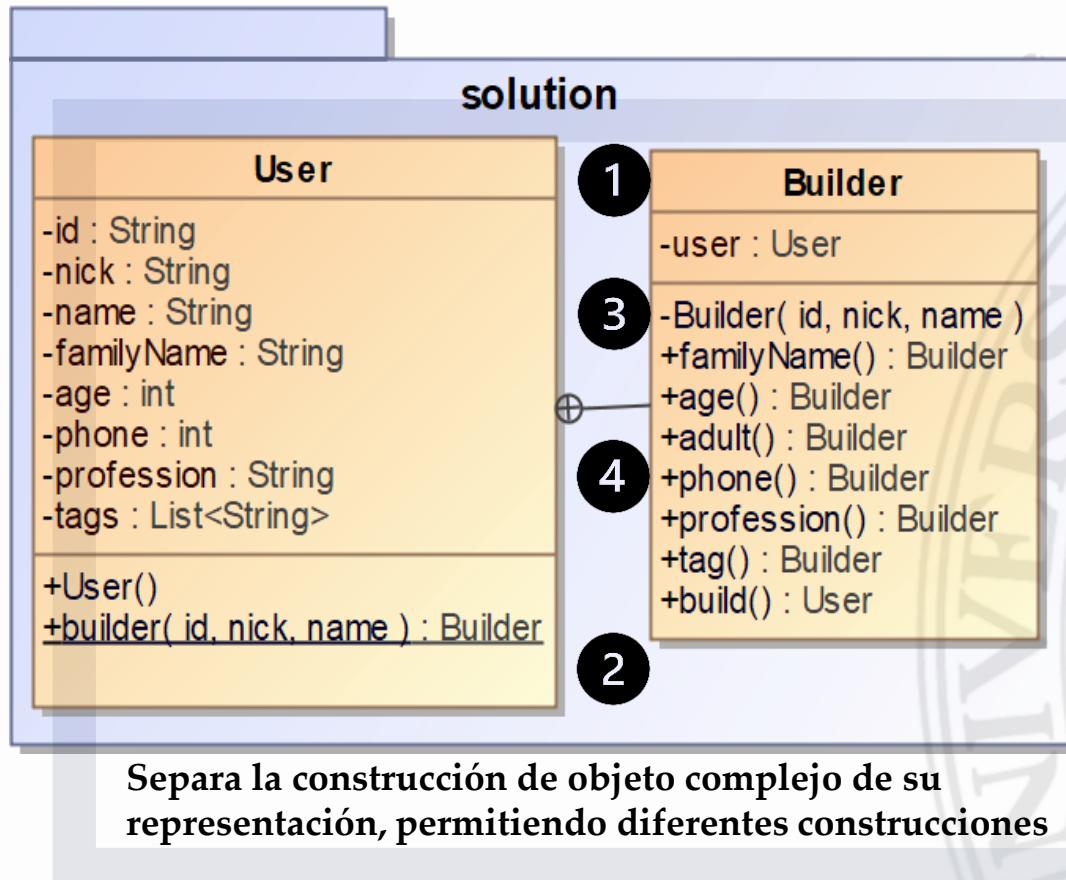
③ *Code Smell*

- Constructores con demasiados parámetros
- Muchos constructores
- Poco usables.



# Builder

Propósito: Creación. Ámbito: objeto



```
User user = User.builder("1", "Paco", "Jose").familyName("De Miguel").phone(66666666).adult()
    .profession("Profesor").tag("Director").tag("socio").tag("Consejo").build();
```

```
User user = User.builder("1", "Paco", "Jose").phone(66666666).familyName("De Miguel").build();
```

① Clase Interna *Builder*.

② Método estático que crea una instancia del *Builder*.

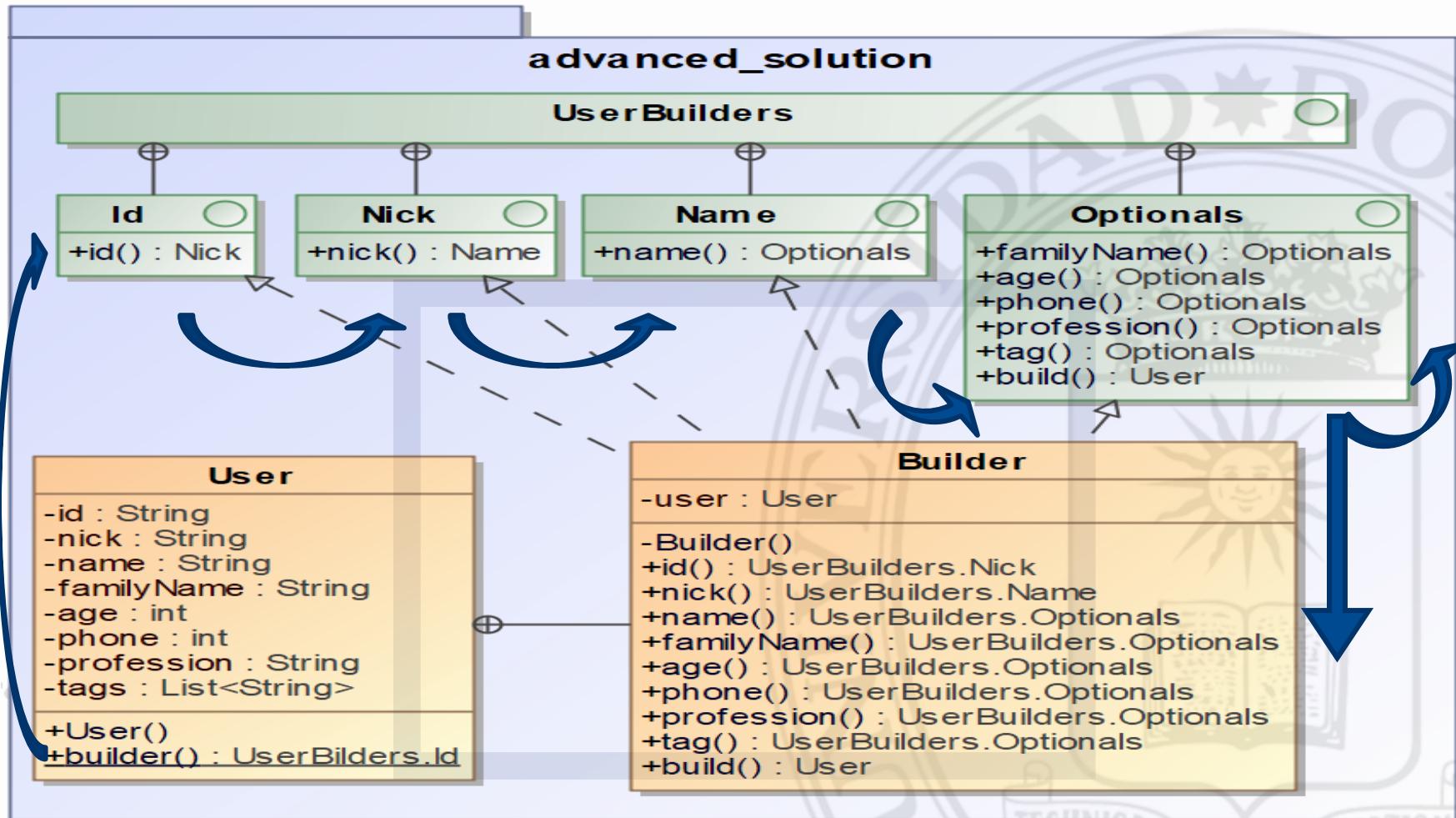
③ Constructor con los parámetros obligatorios.

④ Métodos devuelven *this*

- ¿Qué pasa si son muchos parámetros obligatorios?
- ¿Qué pasa si tenemos un orden de construcción?

# Builder

Propósito: Creación. Ámbito: objeto



Separa la construcción de objeto complejo de su representación, permitiendo diferentes construcciones

```
User user = User.builder().id("1").nick("Paco").name("Jose").tag("Director").age(18).build();
```



<https://github.com/miw-upm/apaw>

- Package: *es.upm.miw.pd.builder*

## Builder in action

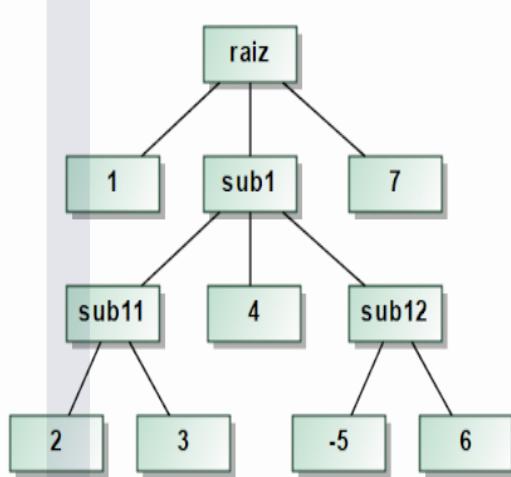
- *User*
- Soluciones

## 📝 Ejercicios

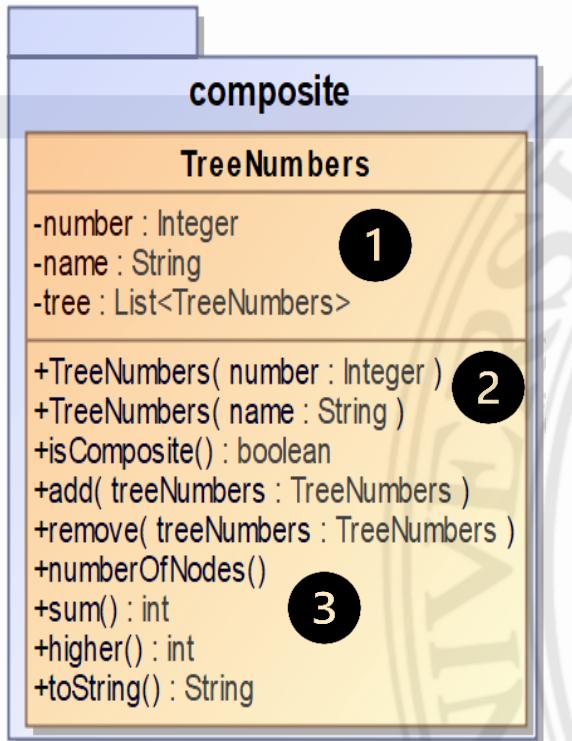
- Aplicar el patrón *builder* a la clase *Article*, atributos obligatorios: *id*, opcionales: *resto*.
- Aplicar el patrón *builder* a la clase *Article*, atributos obligatorios: *id, reference, description & retailPrice*, opcionales: *resto*.
- Aplicar solución con interface para secuenciar los obligatorios.
- Añadir a la clase *Article* el atributo *Provider*, debería lanzarse el *builder* de *Provider*, con los atributos *id, company* **obligatorios**.

# Composite

## Motivación



Composite

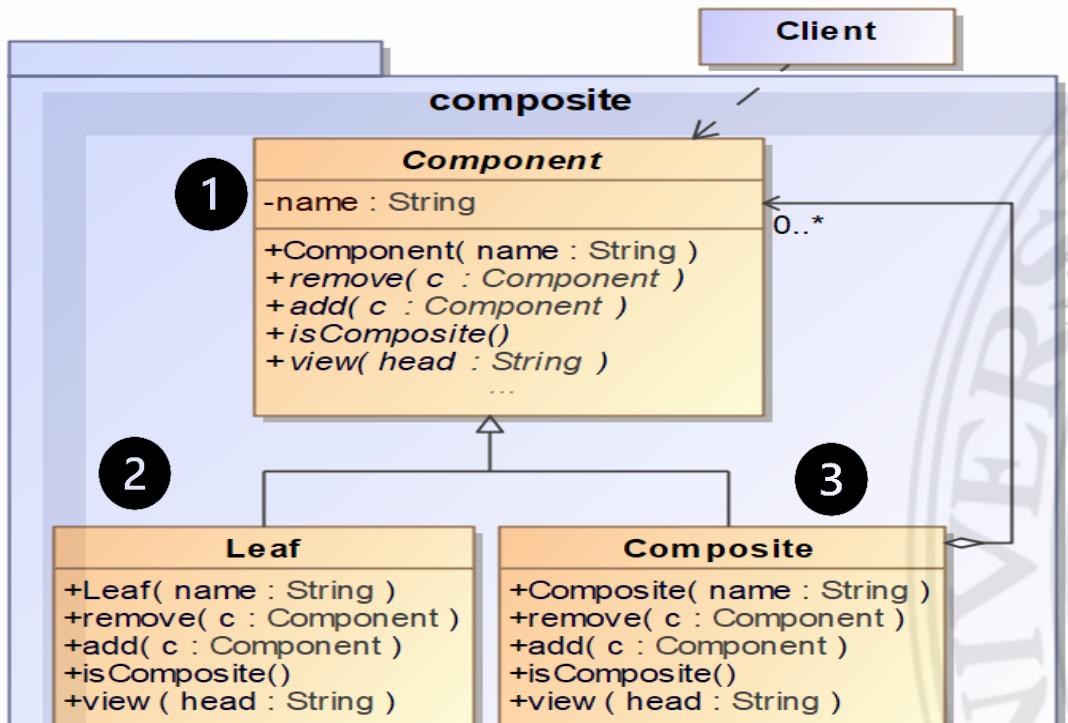


¿Cumple  
DOO?

- ① ¿Cohesión?
- ② ¿SOLID única responsabilidad?
- ③ Anti patrón:  
*Código espagueti*
- Añadir un nuevo tipo de hoja!!!

# Composite

Propósito: Estructural. Ámbito: objeto



Permite estructuras en árbol tratando por igual a las hojas que a los elementos compuestos

## ① Rol Componente

- Clase padre abstracta, es la única visible al cliente.
- Los posibles métodos se definen aquí.

## ② Rol Hoja

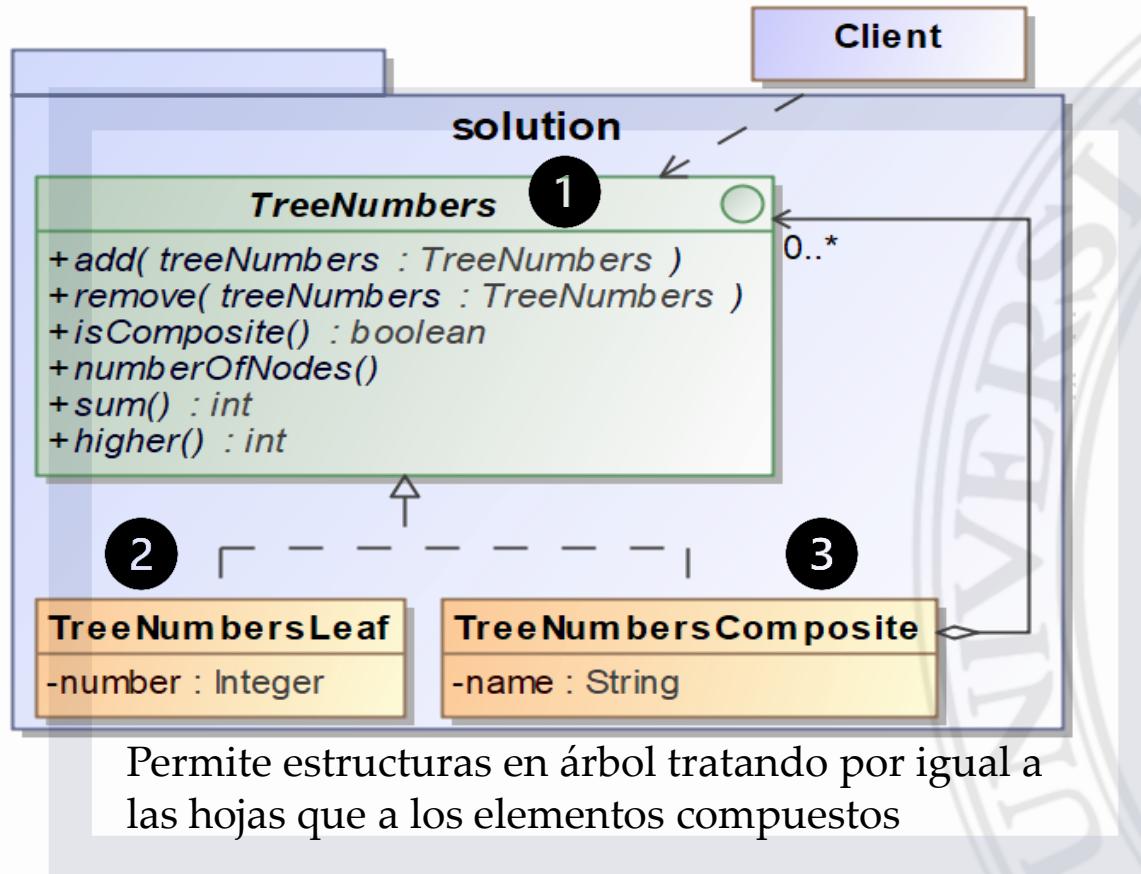
- Hereda de Componente.

## ③ Rol compuesto

- Hereda de *componente*.
- Contiene una lista de *Componente*.

# Composite

Propósito: Estructural. Ámbito: objeto



## ① Rol Componente

- Clase padre abstracta, es la única visible al cliente.
- Los posibles métodos se definen aquí.

## ② Rol Hoja

- Hereda de Componente.

## ③ Rol compuesto

- Hereda de *componente*.
- Contiene una lista de *Componente*.

# Composite

 {→}

## Editor de Expresiones Matemáticas

- Se quiere construir un editor de expresiones matemáticas con valores enteros.
- Especificar el diagrama de clases que permita representar las expresiones.

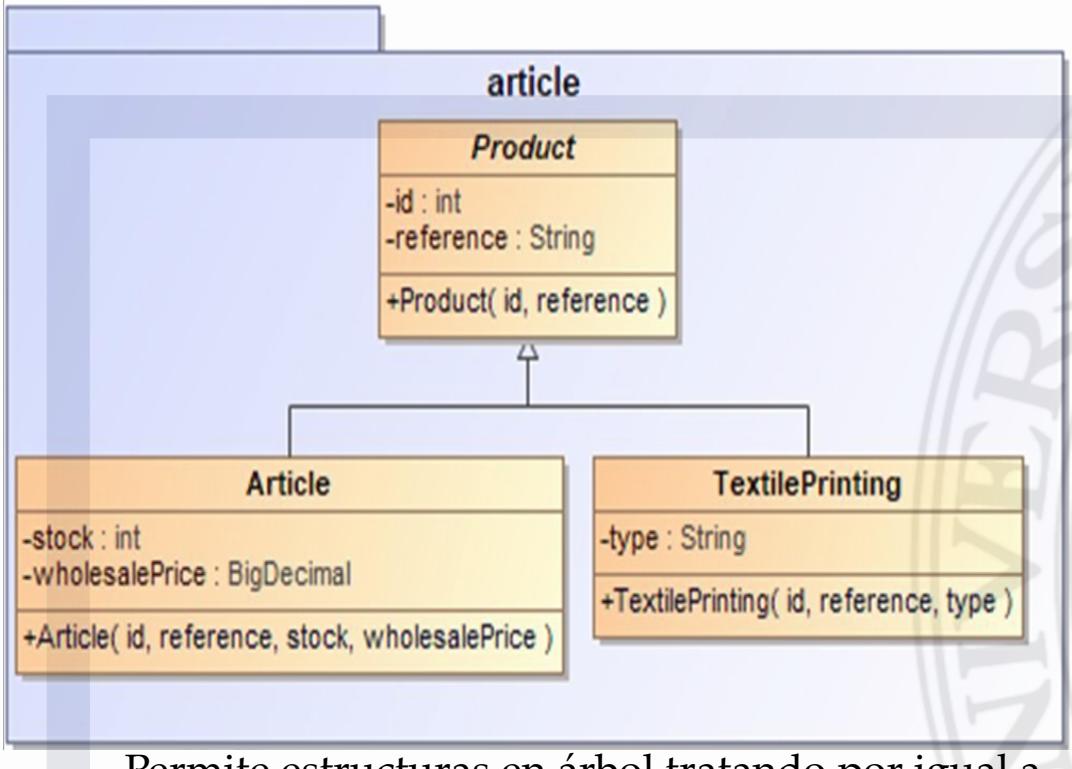
## Expresión

- Una expresión válida estará formada o bien por un número, o bien por la suma, resta, división o multiplicación de dos expresiones.

## Ejemplos de expresiones válidas:

- 5
- $(1+(8*3))$
- $((7+3) * (1+5))$

# Composite

 {→}

Permite estructuras en árbol tratando por igual a las hojas que a los elementos compuestos

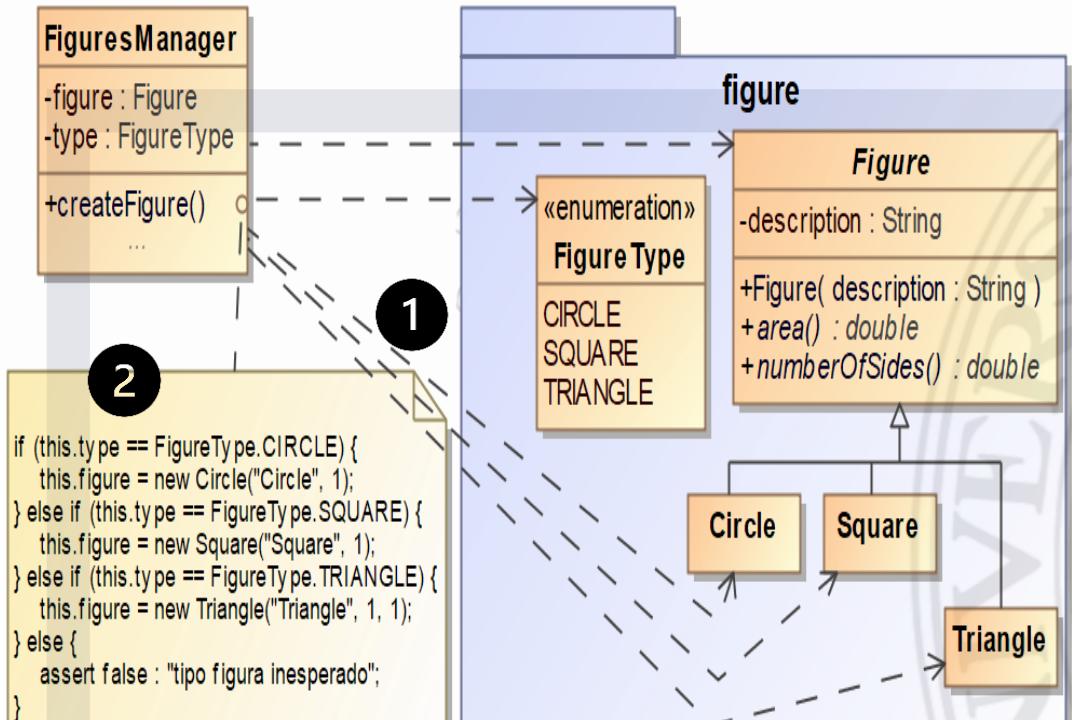
Se parte de un código en producción, y **no se quieren alterar las clases existentes.**

Aplicar el patrón compuesto para realizar agrupaciones en árbol de solo *artículos*.

La agrupación de artículos se identifican con un *nombre* y los artículos por su *reference*

# Factory Method

## Motivación



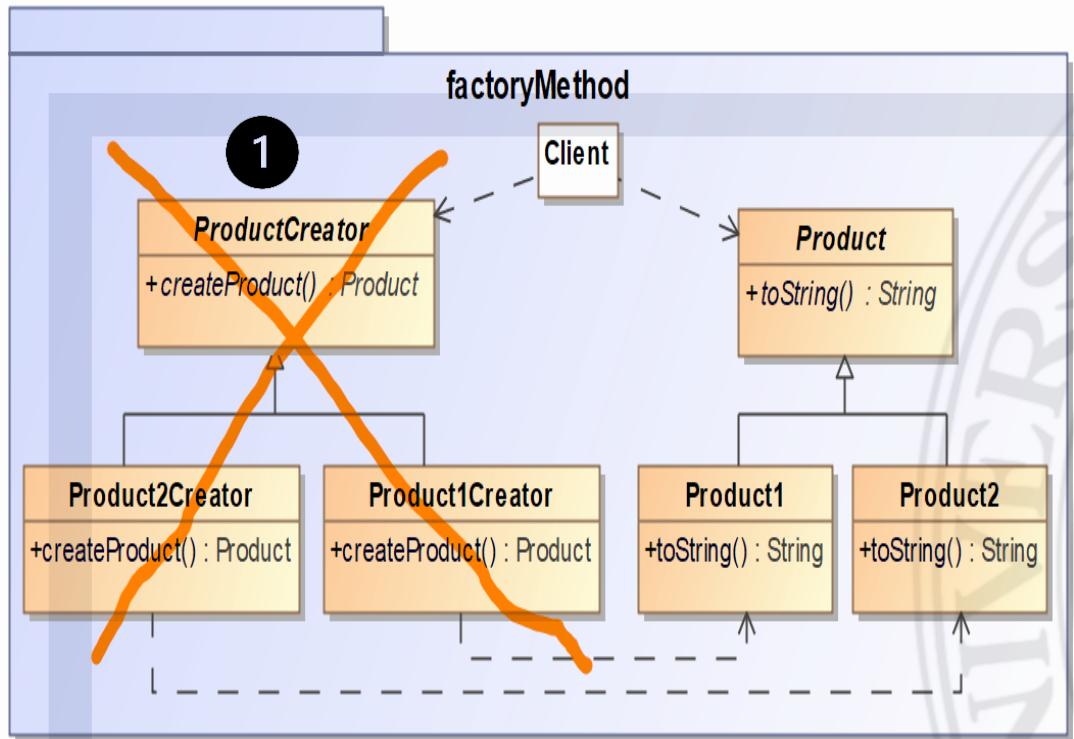
*Factory Method*

El cliente:  
*FiguresManager*,  
debe tener la  
responsabilidad  
de crear figuras

- **1** Dependencias inadecuadas.
- **2** Anti patrón:  
*código espagueti*.

# Factory Method

Propósito: Creación. Ámbito: clase



*Factory Method*

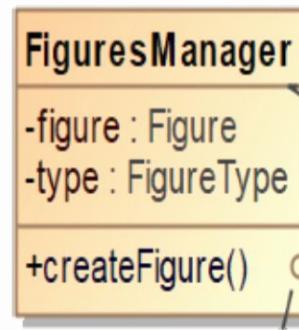
## Propósito

- Define una abstracción para crear objetos, y son las subclases que deciden la clase concreta a instanciar

### ① Función lambda

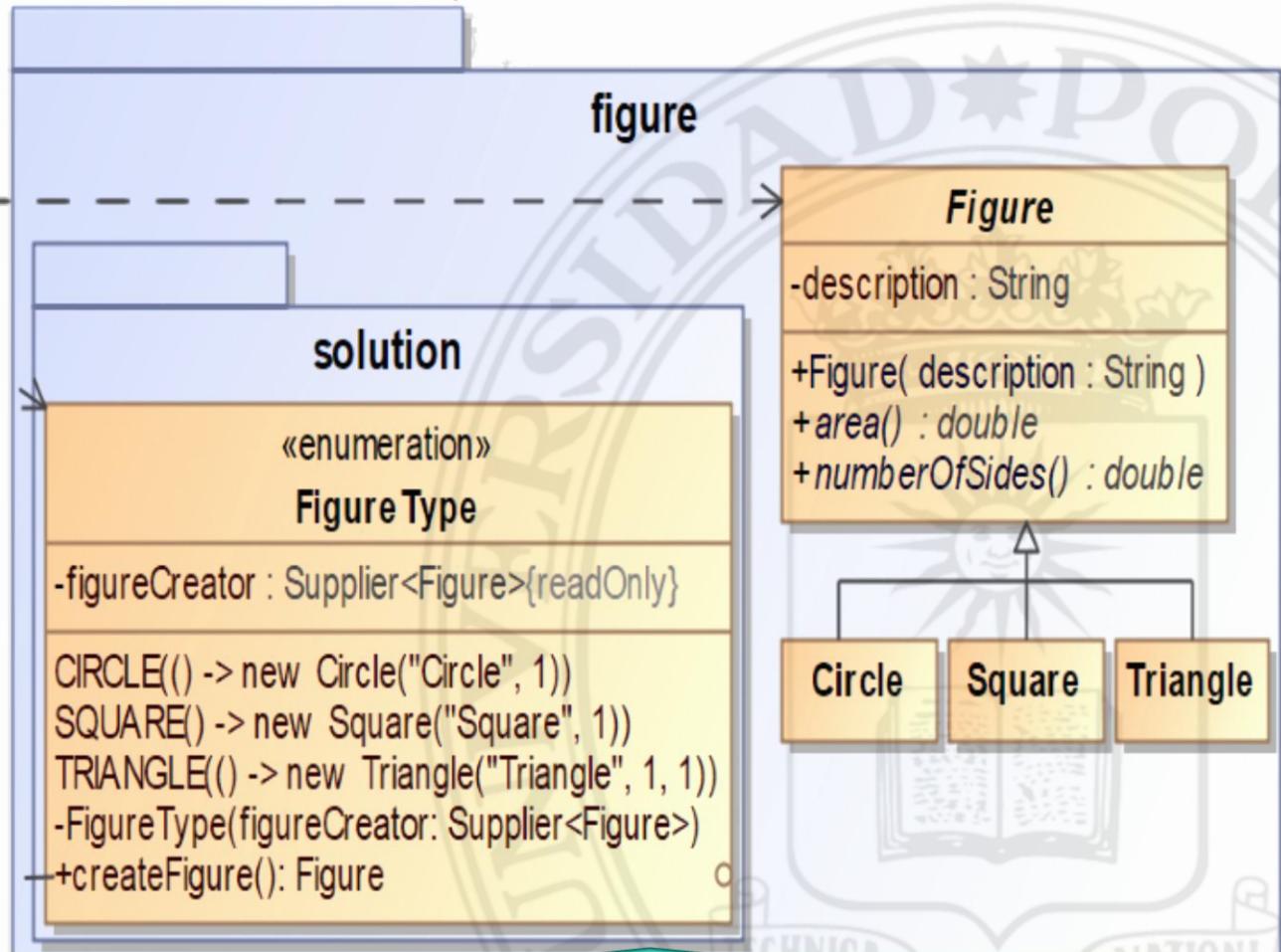
- `Supplier<T>`

# Method Factory



```
this.figure=type.createFigure();
```

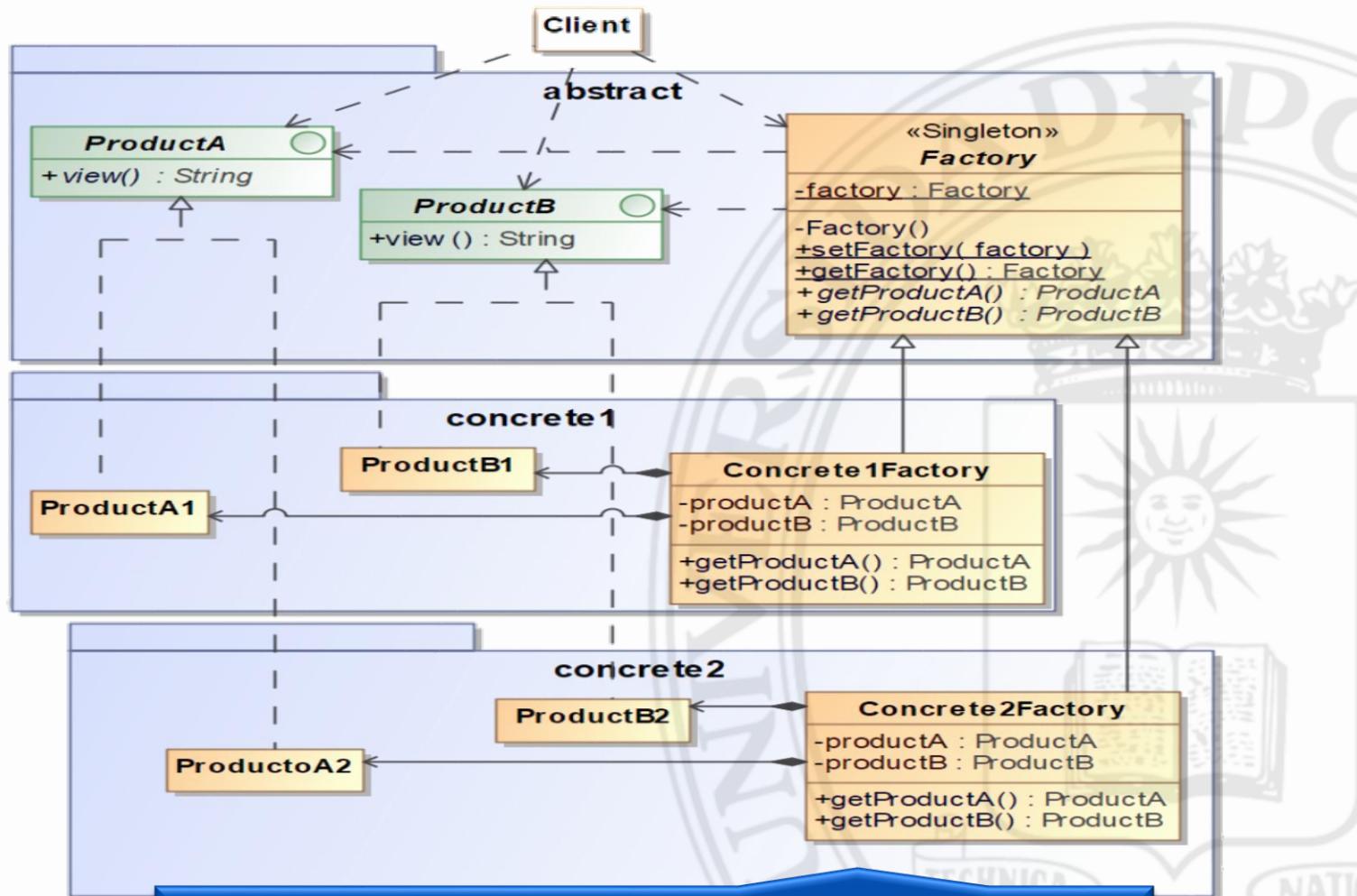
```
return this.figureCreator.get();
```



Solución mediante función Lambda

# Abstract Factory

Propósito: Creación. Ámbito: objeto



Proporciona un interface para crear familias de objetos relacionadas

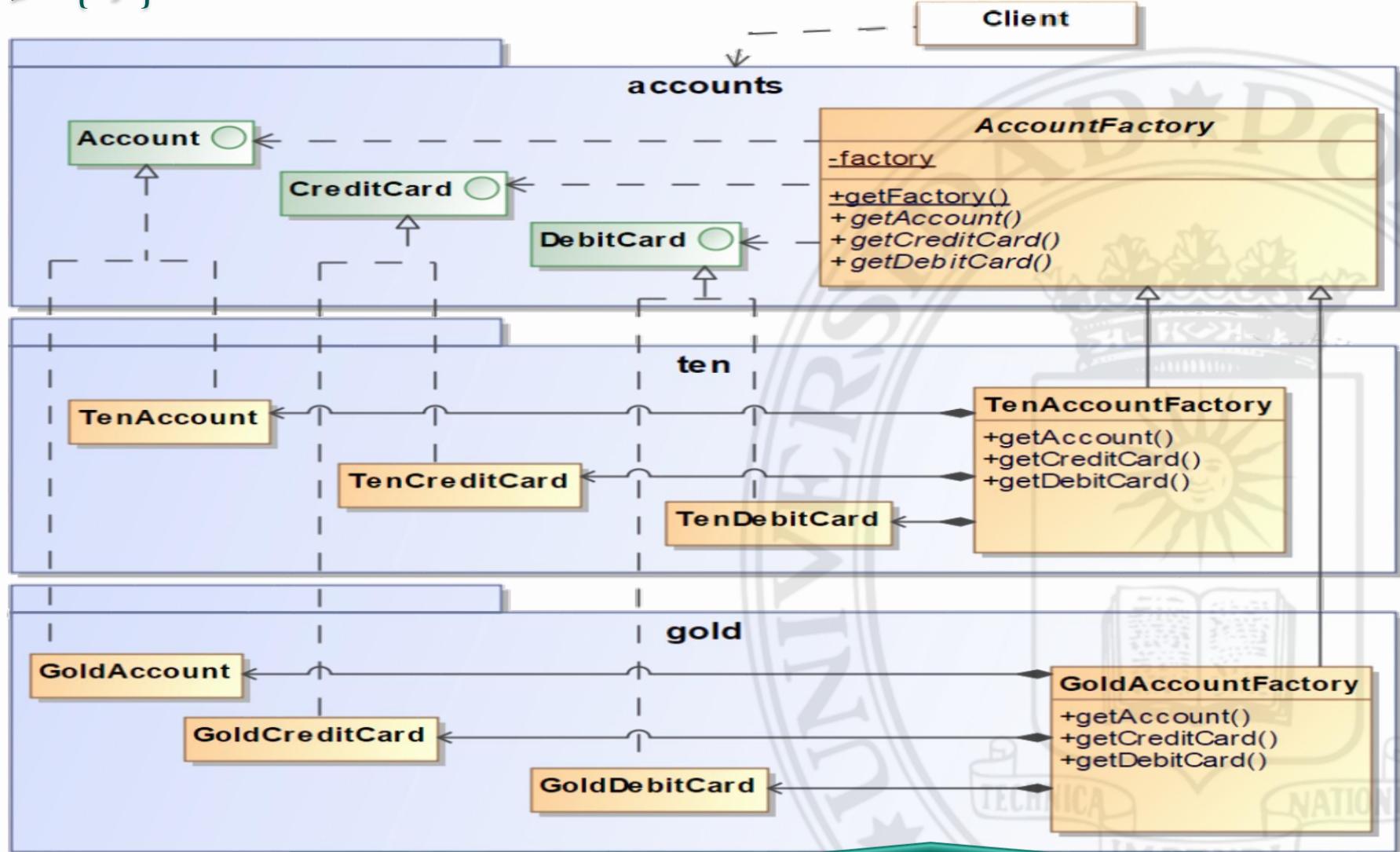
# Abstract Factory



Tipo Cuenta	Cuenta	Tarjeta débito	Tarjeta crédito
Joven	1%	Gratuita	Gratuita Max.600
10	1'5%	Gratuita	10 € Max. 2000€
Oro	2%	5 €	20€ Max. 4000€

Aplicar el patrón *Abstract Factory*,  
solo con nombres de clases

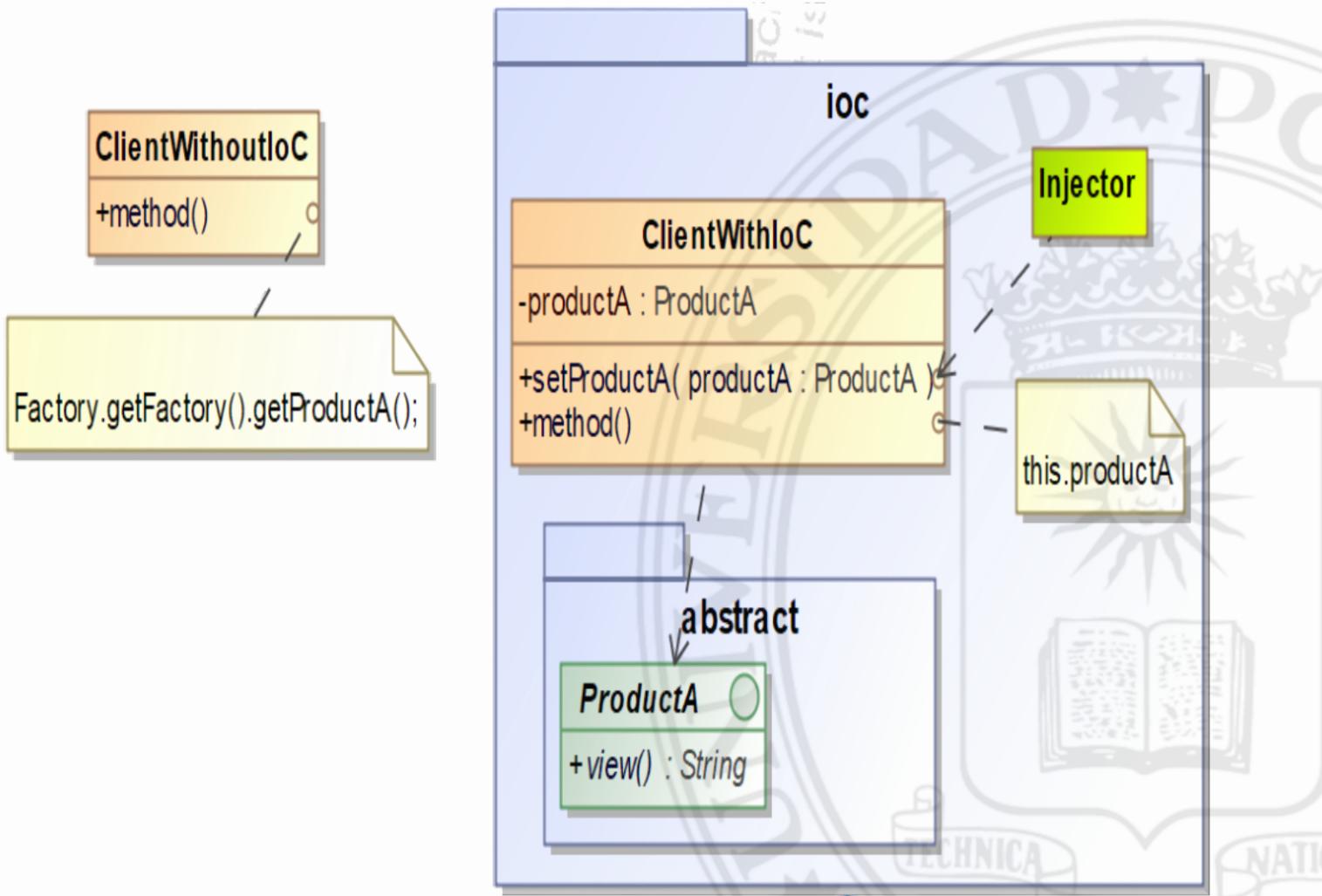
# Abstract Factory



Solución

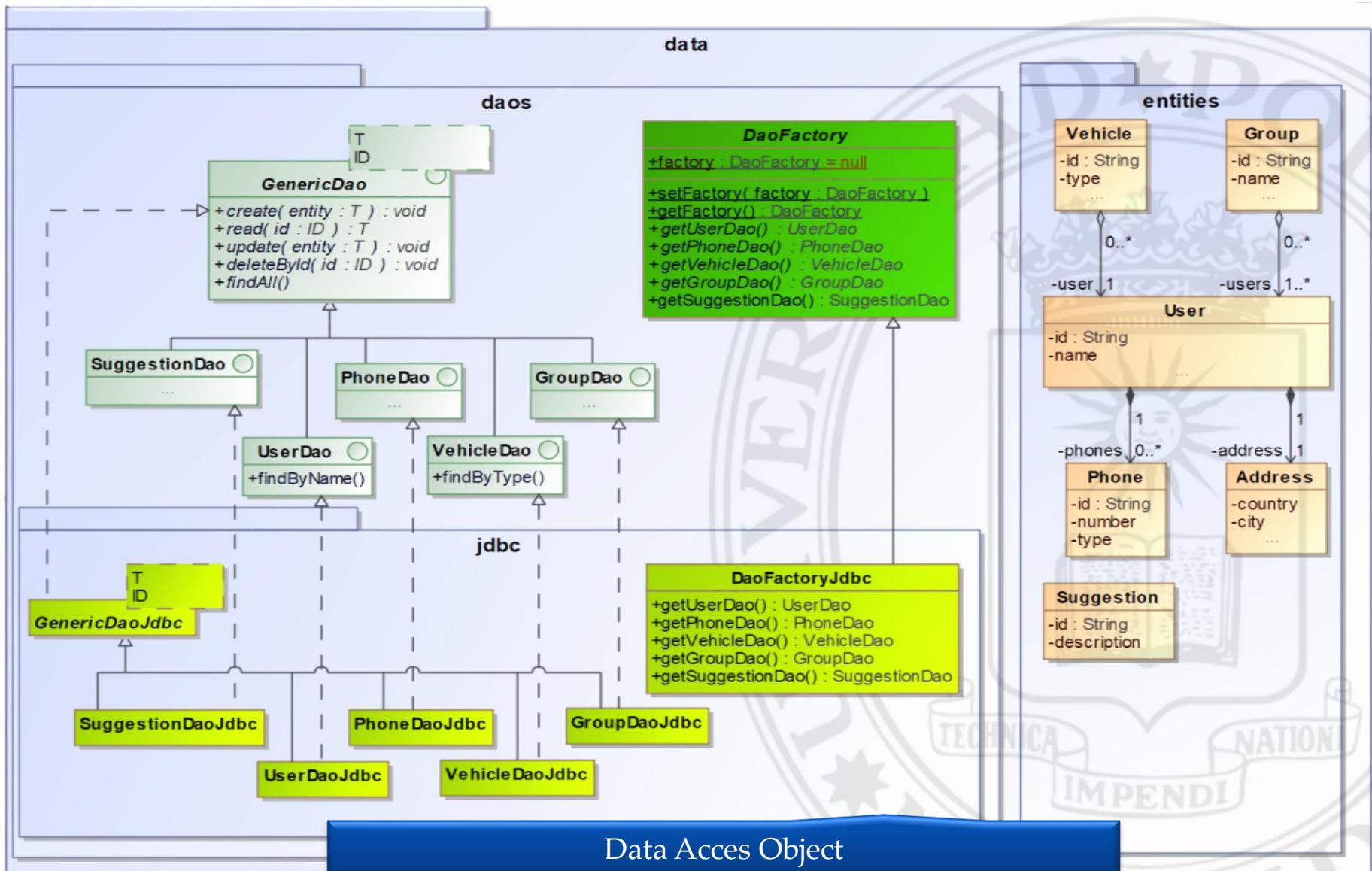
# Inversion Of Control

## Inversión de Control



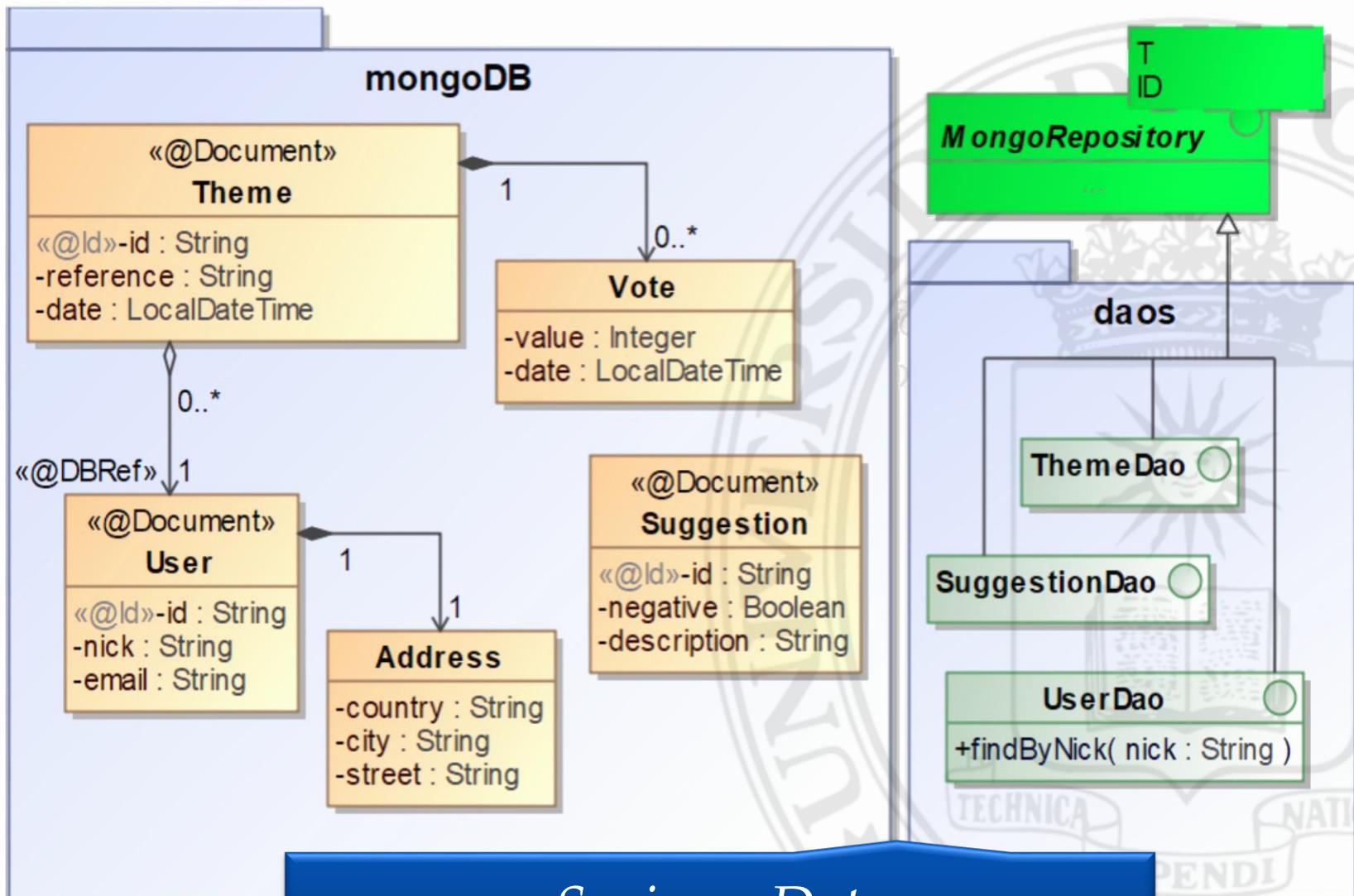
Mínima dependencia, quitar dependencia con *Factory*

# Data Access Object DAO



# Data Acces Object

## *Spring - Data*



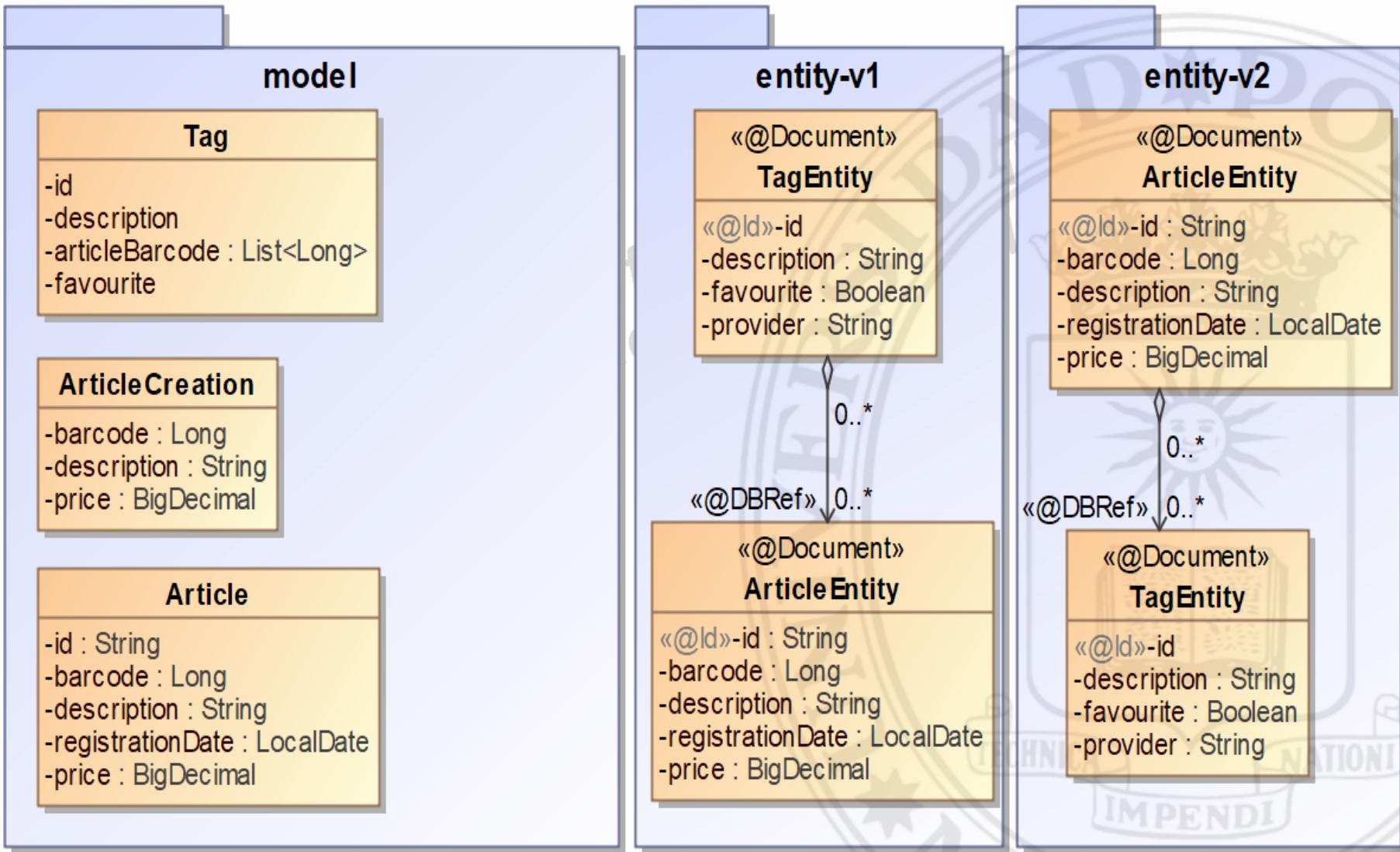
IntegrationTest: IT

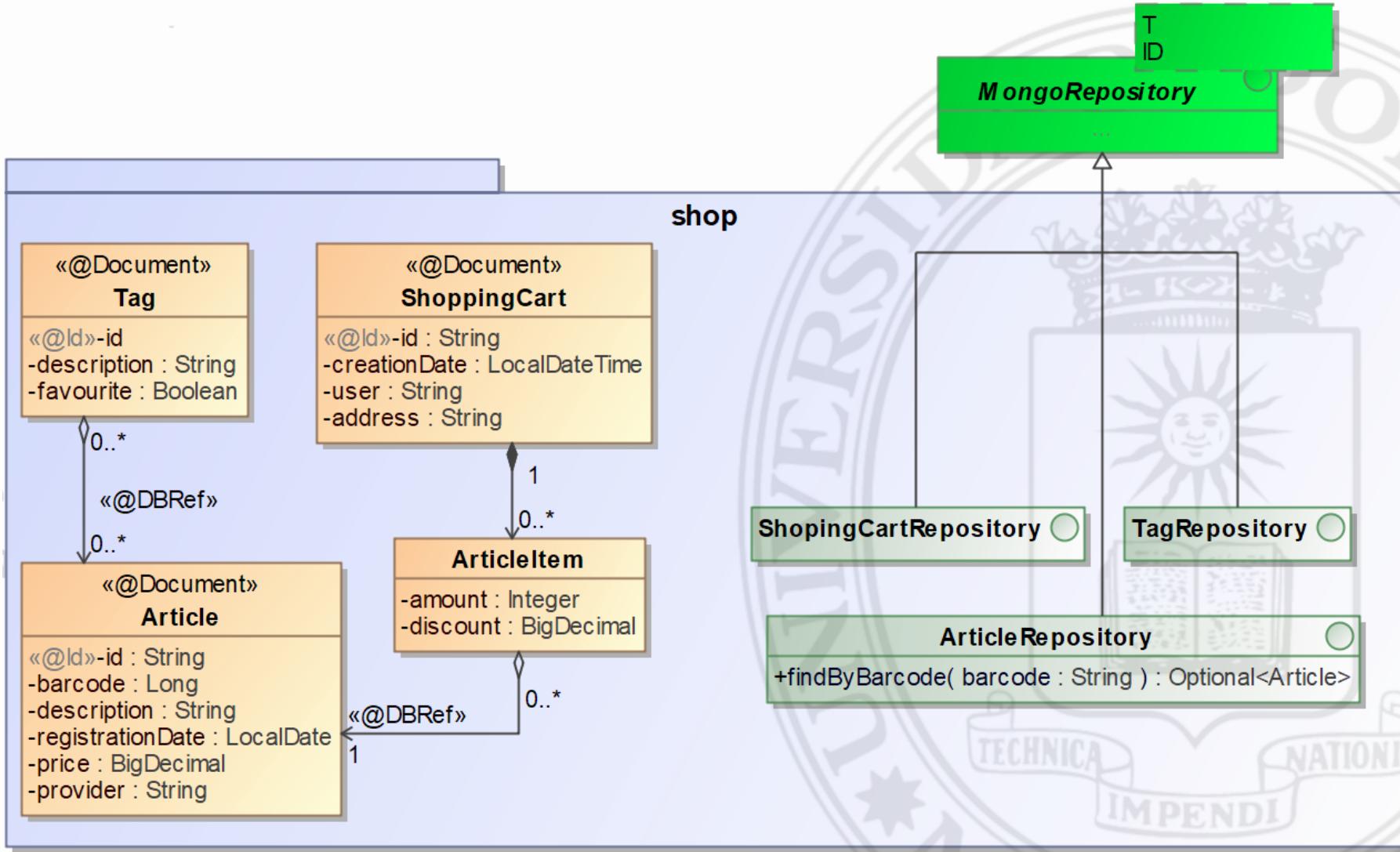
Spring-Test  
JUnit5

*SeedDatabaseService*

seedDatabase()  
deleteAll()

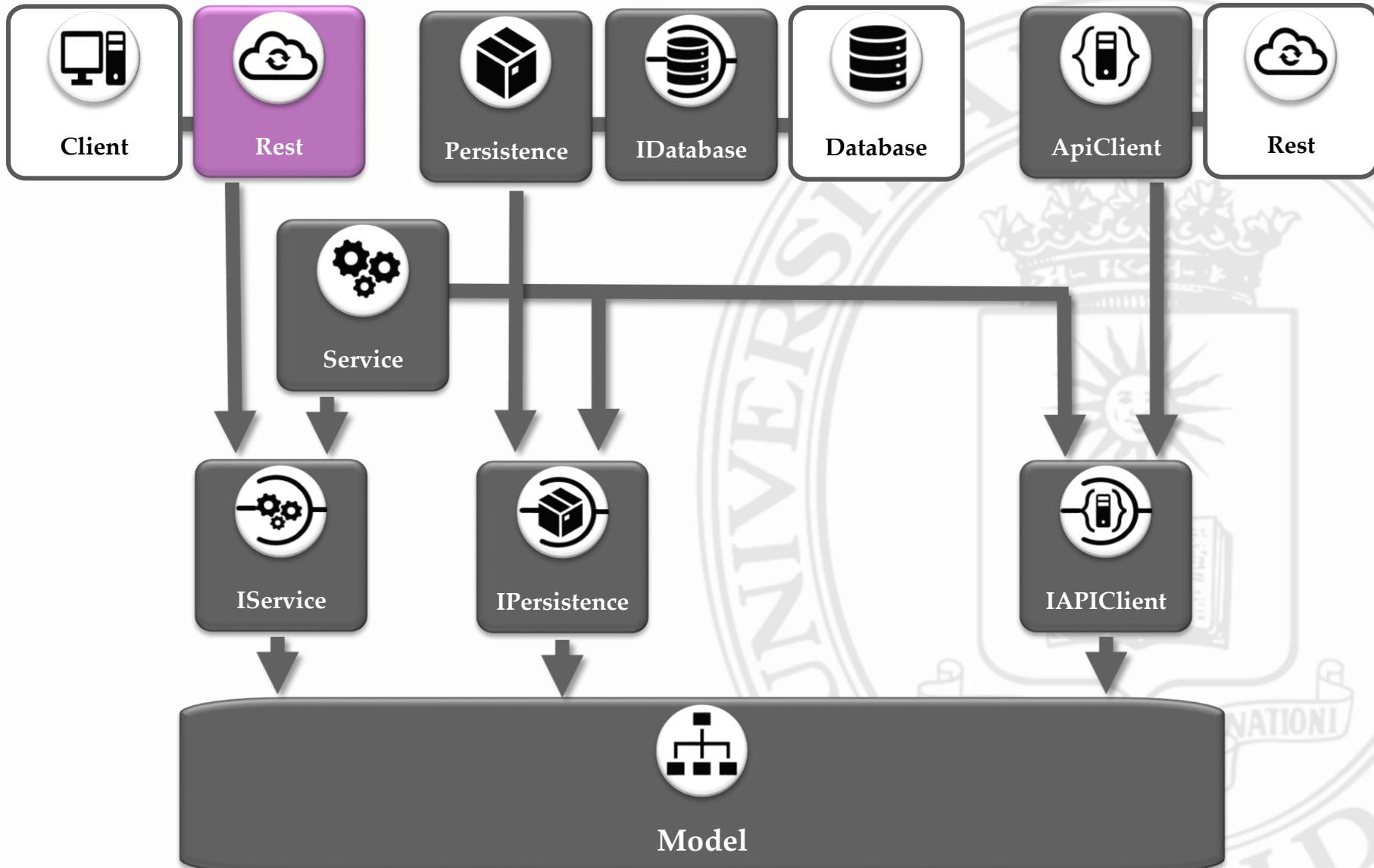
# Dominio Modelo





# Rest

## Adaptador Rest



# Rest

HTTP- *Hyper Text Transfer Protocol*

**URI**

- Server
- Path?
- Params?

**Header**

- Method
- Params?

*Body?*

**HTTP - stateless**

**Header**

- State
- Params?

*Body?*

# ¿Qué es REST?

*Representational State Transfer*

## REST

*Representational  
State Transfer*

Es un estilo de  
arquitectura software  
sobre HTTP

## API

*Application  
Programming  
Interface*

Describe un interfaz

**End-point**

Cada una de las  
posibles peticiones

# ¿Qué es REST?

*Estilo de arquitectura software sobre HTTP*

## HTTP

- Utiliza el protocolo HTTP, sin ninguna capa adicional.

## Relación Cliente-Servidor

- El cliente realiza una petición y el servidor da una respuesta.

## Sin estado

- No se guardan datos de la petición del cliente en el servidor.
- En cada petición el cliente debe enviar toda la información.

## Tipo de Representación

- Se permite la elección del tipo de representación (JSON, HTML, XML,...) a través de los tipos MIME

## Cacheable

- En las respuestas se indica si es cacheable.

## Operaciones CRUD

- Se permiten cuatro operaciones: Create (POST), Read (GET), Update (PUT/PATCH) y Delete (DELETE).

# REST

## Buenas prácticas

### SSL

- Siempre!!!

### Documentar

- OpenAPI

### Versionado

- Para versiones utilizar v\* en el nivel mas alto

### Recursos

- Autodescriptivos
- Sustantivos en plural

### ID

- Procurar id's no deducibles

### Granularidad

- Controlar el tamaño de la respuesta

### Error

- Devolver mensaje de error en el cuerpo

### Pretty print

- Con gzip

### HATEOAS?

- Hypermedia as the Engine of Application State

# REST

## Buenas prácticas

### URI's

“/” jerarquía de los recursos

“,” y “;” para partes no jerárquicas

NO referenciar acciones ~~/v0/get-orders~~

NO referenciar formatos ~~/v0/orders/pdf~~

### Búsquedas:

GET /orders/fixed-search

### GET

GET /orders/search?q=name:jes

GET /search/repo?q=tetris;language:java&sort=stars

### Respuestas

Respuestas parciales: ...?fields=id,description

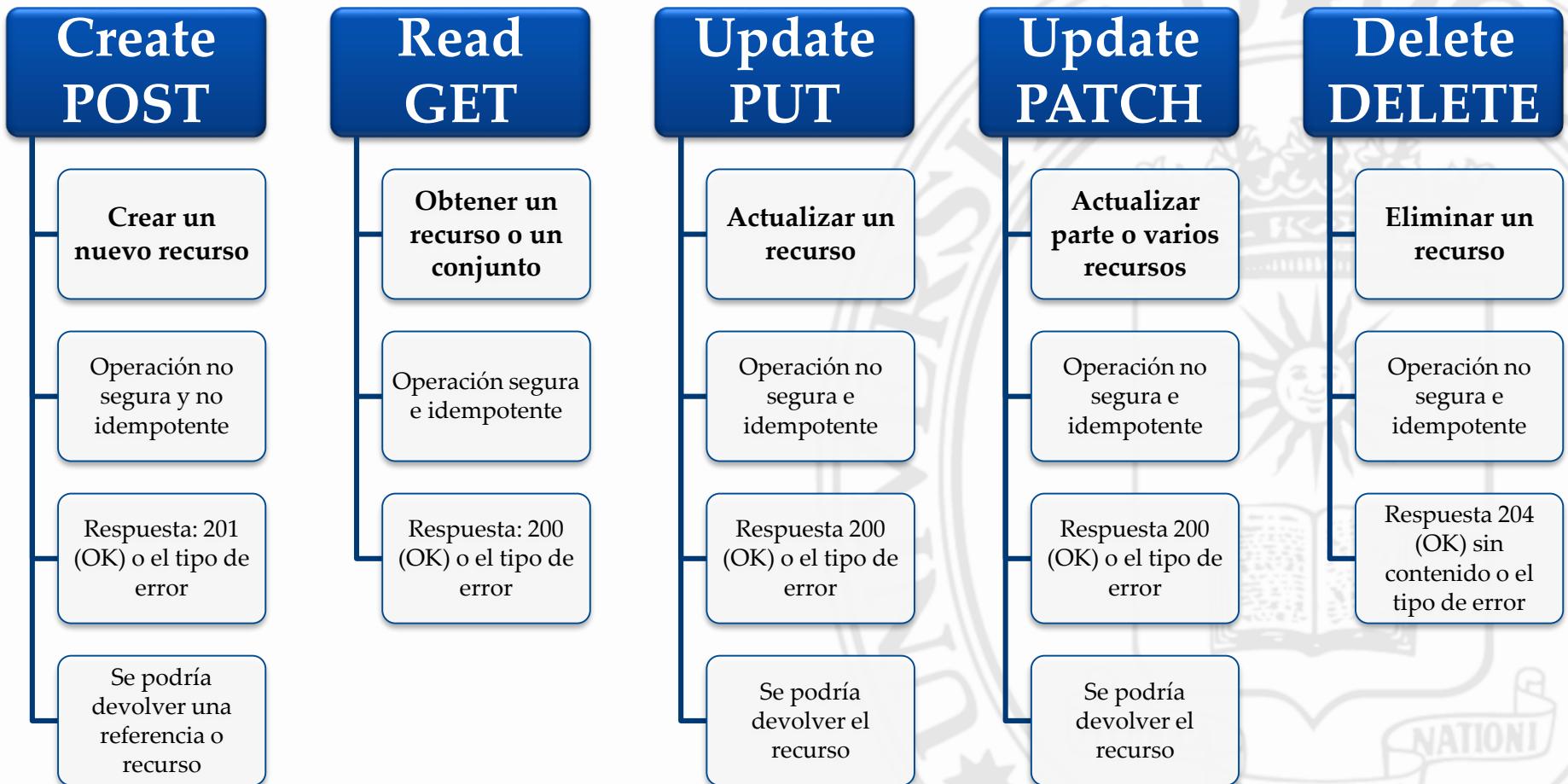
Paginación: ...?page=1&size=30

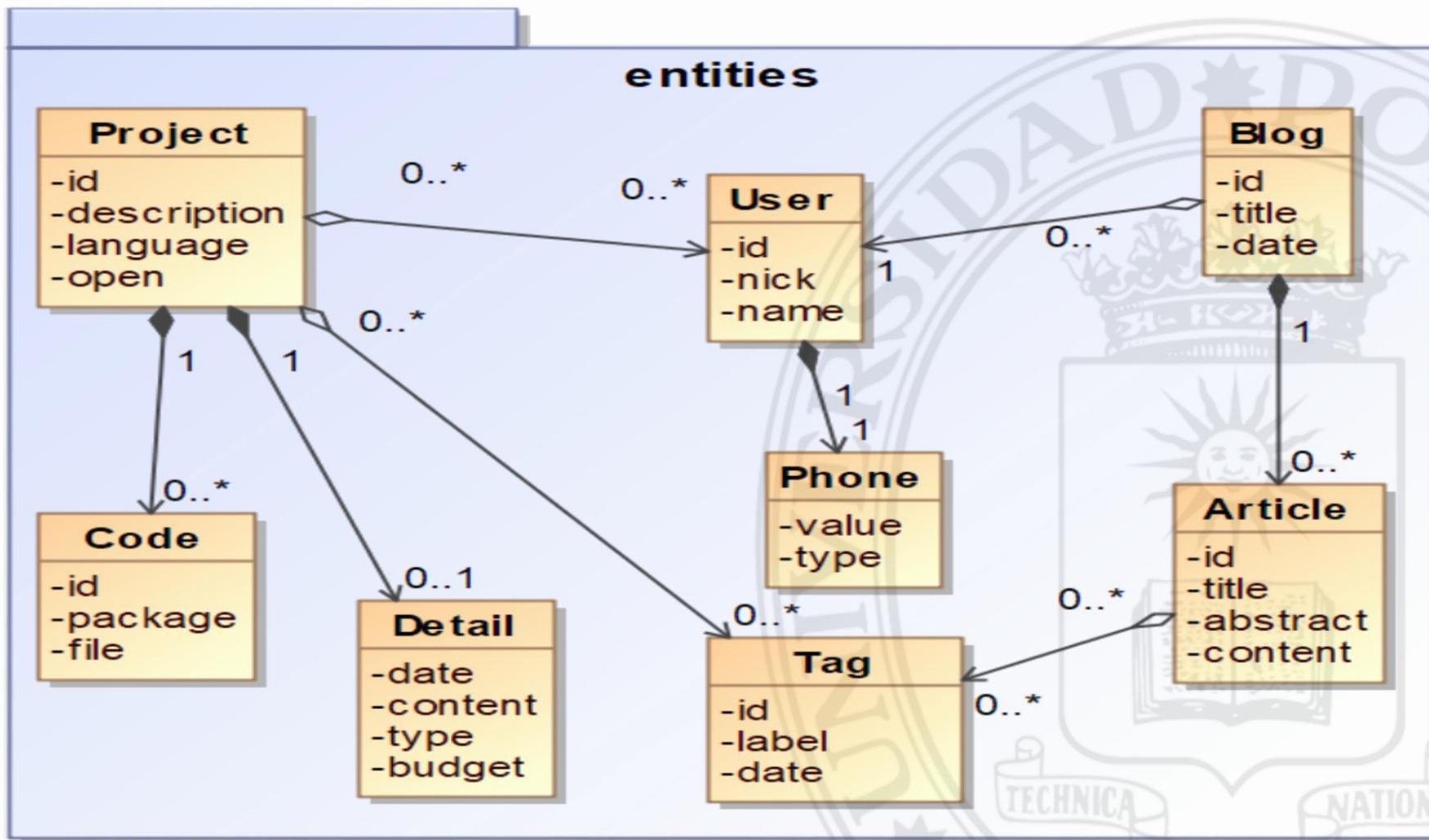
Filtros: ...?type=3,5

Orden: ...?sort=name,surname&desc=id

# REST

## Buenas prácticas





Crear un API Rest

# Rest



## Server URI

- `http://server.com/api/v0/*`
- `http://api.server.com/v0/**`

## End-points

- `GET /users` response: [{id, nick}]
- `POST /users` request: {nick, name...}
- `GET /users/{id}` response:{id, nick, name, phone{value, type}}
- `PUT /users/{id}` request:{...}
- `PATCH /users/{id}` request:[{nick:new-nick, phone/value:new-value}] Atomico
- `DELETE /users/{id}`
- `GET /blogs` response:{id,title}
- `DELETE /blogs/{id}`
- `GET /blogs/{id}/user`
- `POST /blogs` request: {...}
- `GET /blogs/{id}/articles/{id}/abstract` response:{abstract}
- `GET /tags` response:{...}
- `GET /projects/search?q=language:java&sort=id`
- `GET /projects/{id}/tags?sort=-date,label`
- `GET /projects?fields=language,description`
- `GET /users?page=2&size=20`
- `GET /search?q=language:java+language:typescript&sort=id`
- `GET /projects/opened`
- `GET /projects/search?q>tag:id`
- `GET/projects/{id}/open`

# Rest

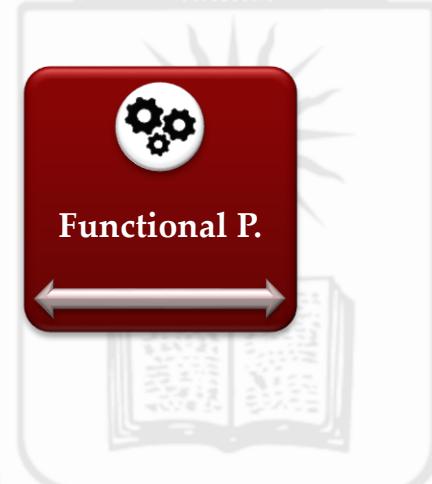
## Responsabilidades

Define el *path* y el  
método del recurso

Gestionar la validación de  
los *dtos*

Reorganizar la recepción  
de datos de la petición

Delega en el *Servicio* la  
ejecución de la petición



## rest\_adapter

```
«@RestController»  
«@RequestMapping("/users")»  
UserResource
```

-userDomain : UserDomain

```
«@Autowired»+User( userDomain : UserDomain )  
«@PostMapping»+create( @RequestBody userCreationDto : UserCreationDto )  
«@GetMapping»+readAll() : List<User>  
«@GetMapping("/{id}")»+read( @PathVariable id : String ) : User  
«@PutMapping("/{id}")»+update( @PathVariable id : String, @RequestBody userDto : UserDto )  
«@PutMapping("/{id}/name")»+updateName( @PathVariable id : String, @RequestBody userDto : UserDto )  
«@PatchMapping("/{id}")»+updateAll( @PathVariable id : String, @RequestBody updateUsersDto : UpdateUsersDto )  
«@DeleteMapping("/{id}")»+delete( @PathVariable id : String )  
«@GetMapping("/search")»+find( @RequestParam q : String ) : List<User>  
«@GetMapping("/{id}/name")»+readName( @PathVariable id : String ) : UserDto
```

GET /users/666/name

GET /users/search?q=name:j&order=asc

# Rest

## *Spring - Test*

*Spring-Test: WebTestClient*

URI & params?

Method: *post, get...*

Body?

**Exchange**

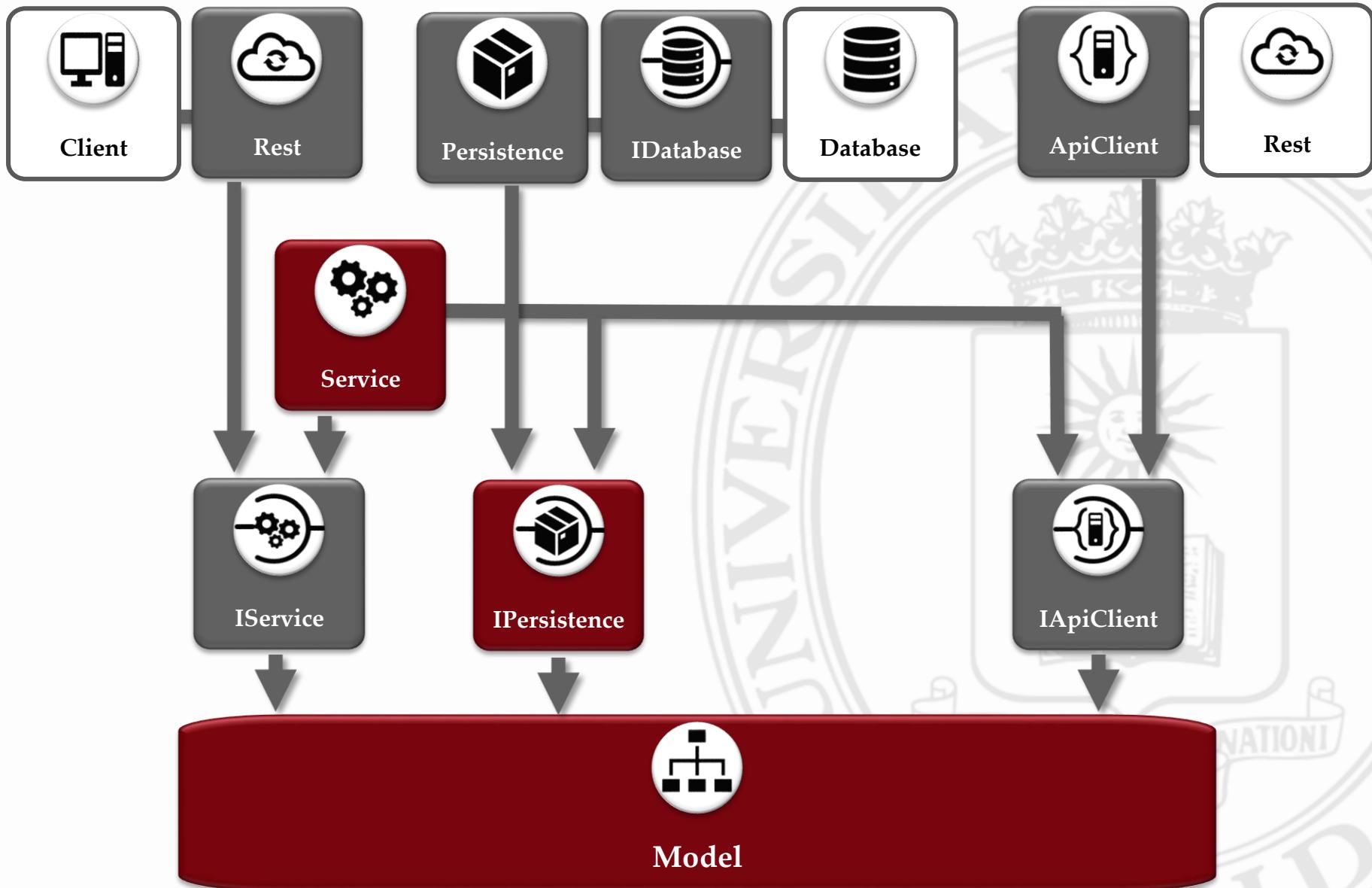
**expect\***

- expectStatus, expectBody...

<https://github.com/miw-upm/apaw-practice>

- *main* es.upm.miw.apaw\_practice.adapters.rest.shop
- *test* es.upm.miw.apaw\_practice.adapters.rest.shop

# Domain



## Programación Funcional & flujos

- Programación declarativa: ¿Qué?
- Basado en Funciones: funciones Lambda.
- Funciones de orden superior. Manejan funciones como parámetros de entrada y salida.
- Flujos de datos.
- Sin estado, sin orden y sin efectos colaterales.
- Valores inmutables: paso de parámetros por valor.

# Domain Patrones

## Facade

- Proporciona un interface unificado para un conjunto de interfaces de un subsistema.

## Strategy

- Define un conjunto de algoritmo haciéndolos intercambiables dinámicamente.

## Observer: OOP

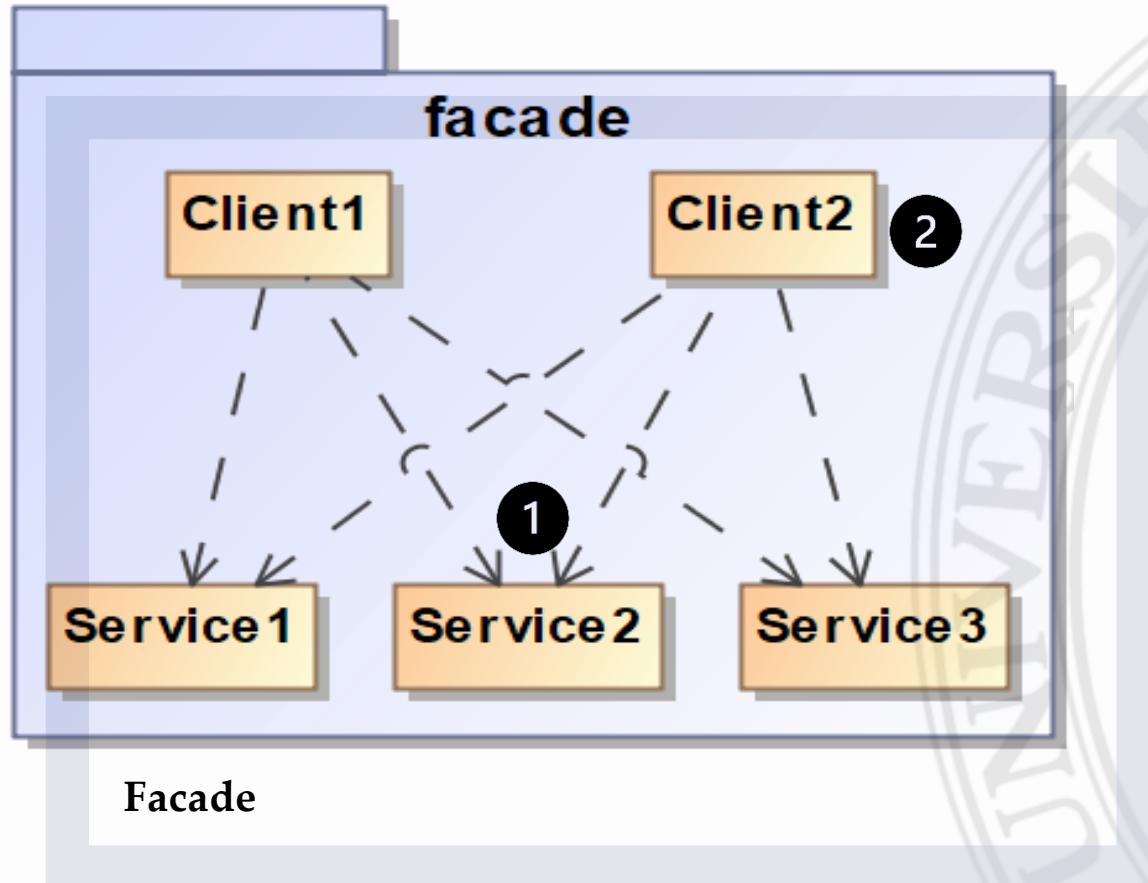
- Se define una dependencia entre uno a muchos, del tal manera, que cuando cambie el sujeto avise a todos los objetos dependientes.

## Publisher: FP

- Se permite que un publicador envíe mensajes de forma asincrónica a varios subscriptores interesados, sin crear dependencias.

# Facade

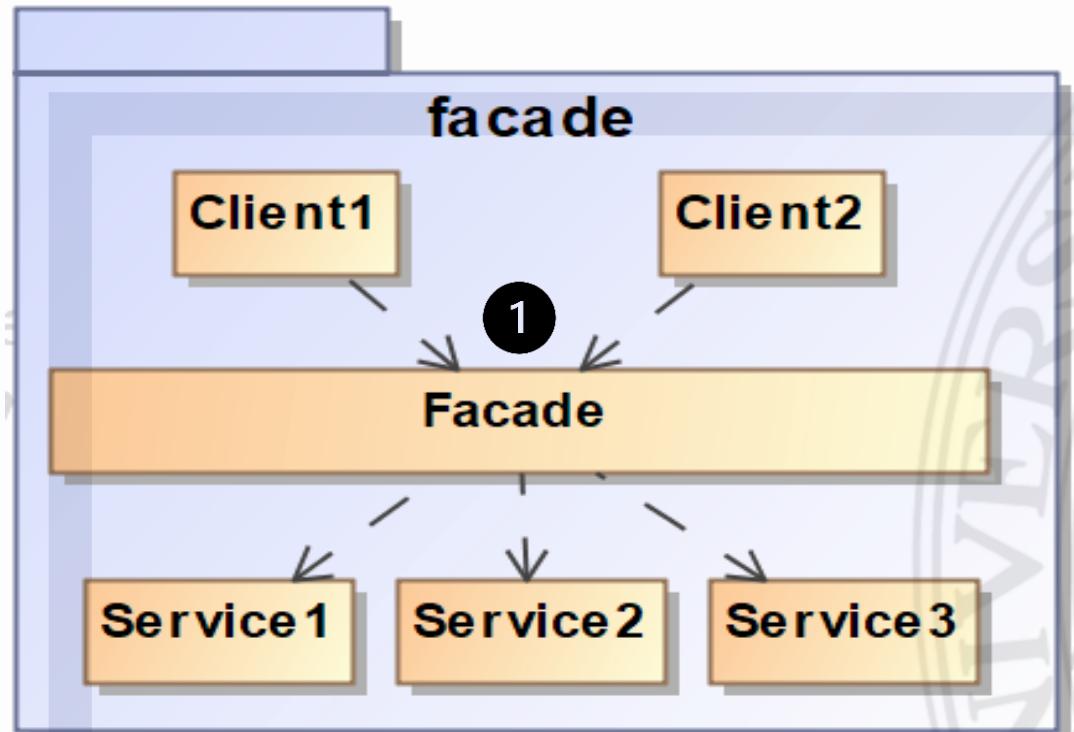
Propósito: Estructural. Ámbito: objeto



- ① Subsistema complejo, difícil de utilizar. Librerías de terceros.
- ② En el desarrollo del Cliente2 no se puede reutilizar el código del Cliente1.

# Facade

Propósito: Estructural. Ámbito: objeto



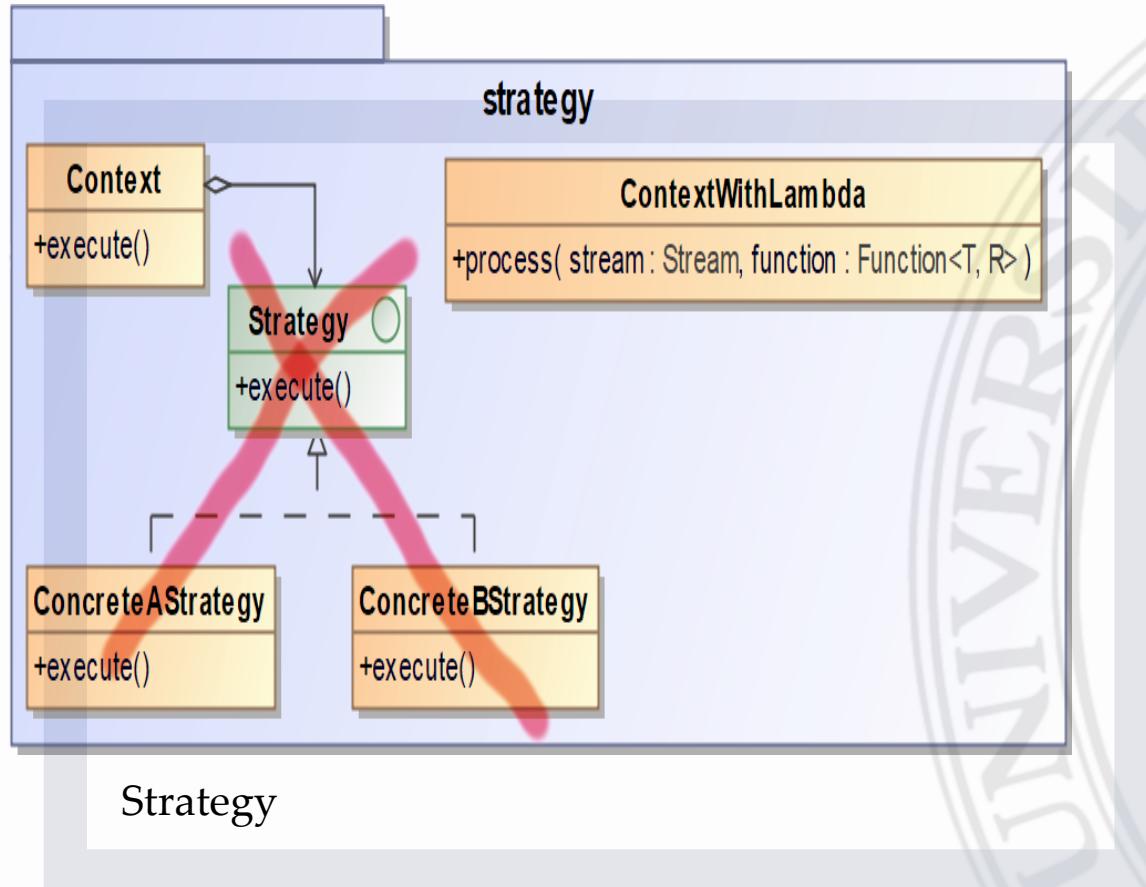
Proporciona un interface unificado para un conjunto de interfaces de un subsistema

① El código aportado en el desarrollo del *Client1* se reutiliza con facilidad para el *Client2*

Podría ser necesario ampliar la fachada para el desarrollo de Cliente2

# Strategy

Propósito: Comportamiento. Ámbito: objeto

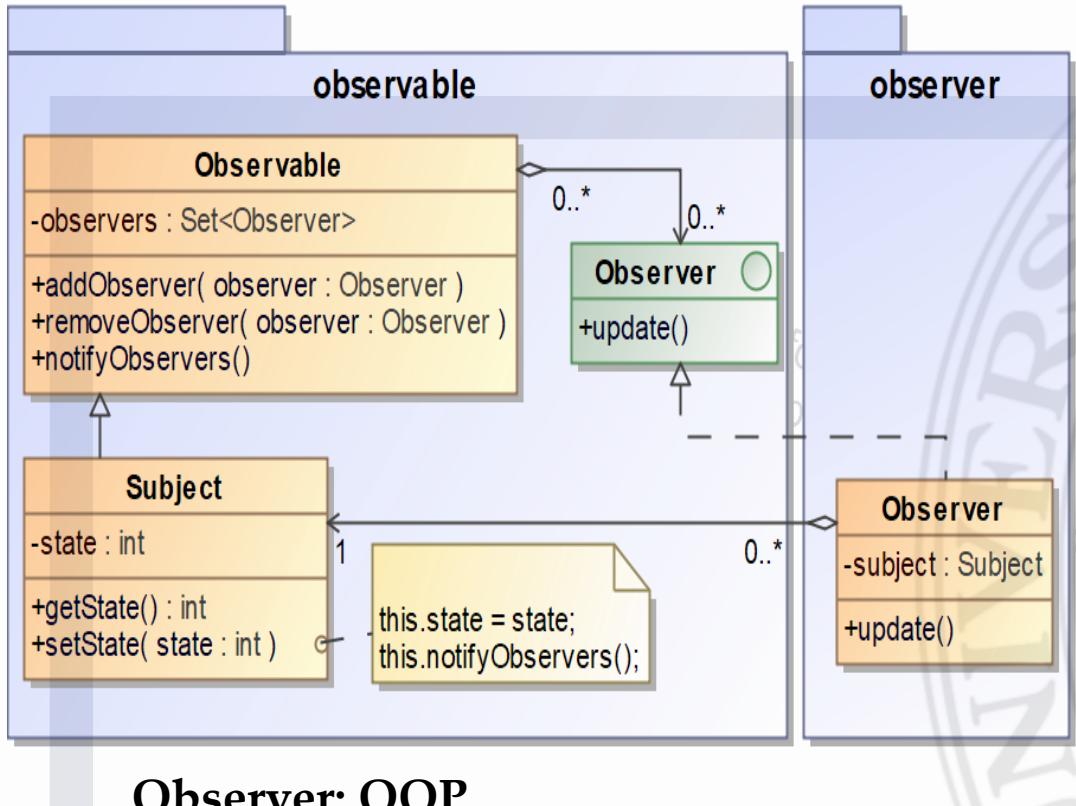


## Propósito

- Define un conjunto de algoritmo haciéndolos intercambiables dinámicamente

# Observer

Propósito: Comportamiento. Ámbito: objeto



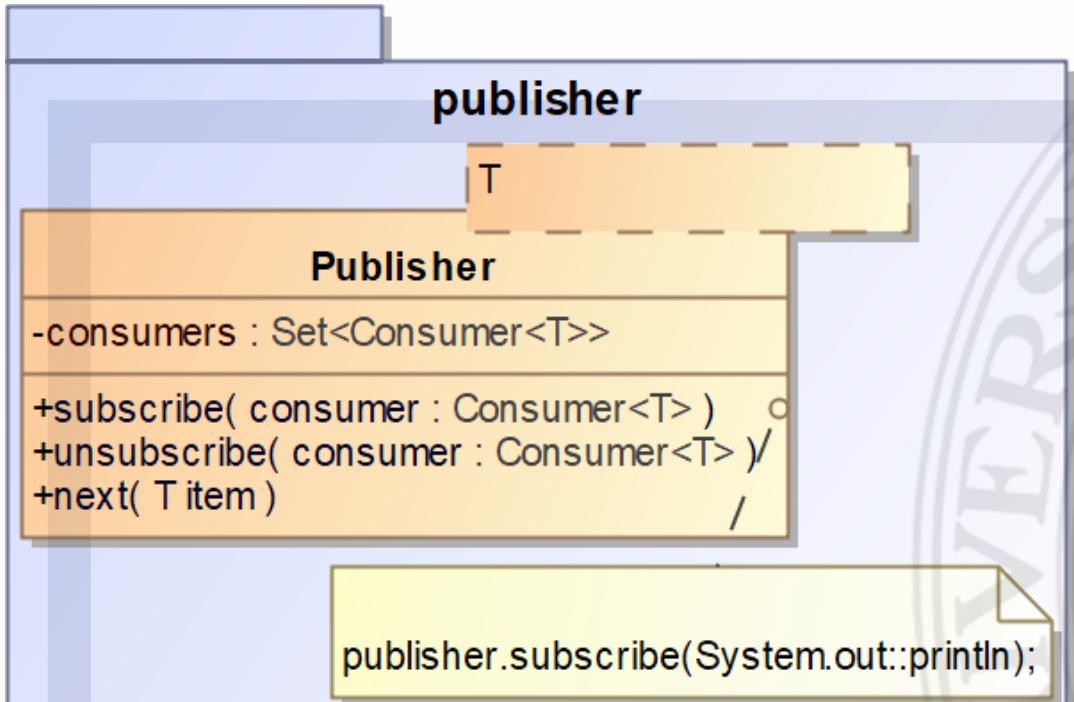
**Observer: OOP**

## Propósito

- Se define una dependencia entre uno a muchos, de tal manera, que cuando cambie avise a todos los objetos dependientes

# Publisher

Propósito: Comportamiento. Ámbito: objeto



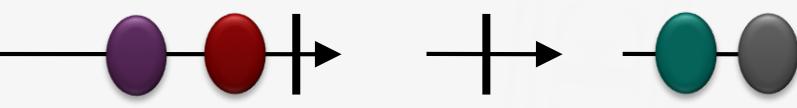
**Publisher: FP**

## Propósito

- Se permite que un publicador envíe mensajes de forma asincrónica a varios subscriptores interesados, sin crear dependencias.

# Programación Reactiva

# *Reactive Programming*

- Flujo de datos asíncronos
  - Programación Funcional
  - Programación Asíncrona



# Programación Reactiva

## Proyectos



### ReactiveX

- JavaScript, Java, Python, C#, PHP, C++, Swift, Go, Kotlin, Ruby, Scala,...
- <http://reactivex.io/>

RxJS



### RxJS - Javascript

- Angular
- Clase: *Observable*
- <https://rxjs.dev/guide/overview>

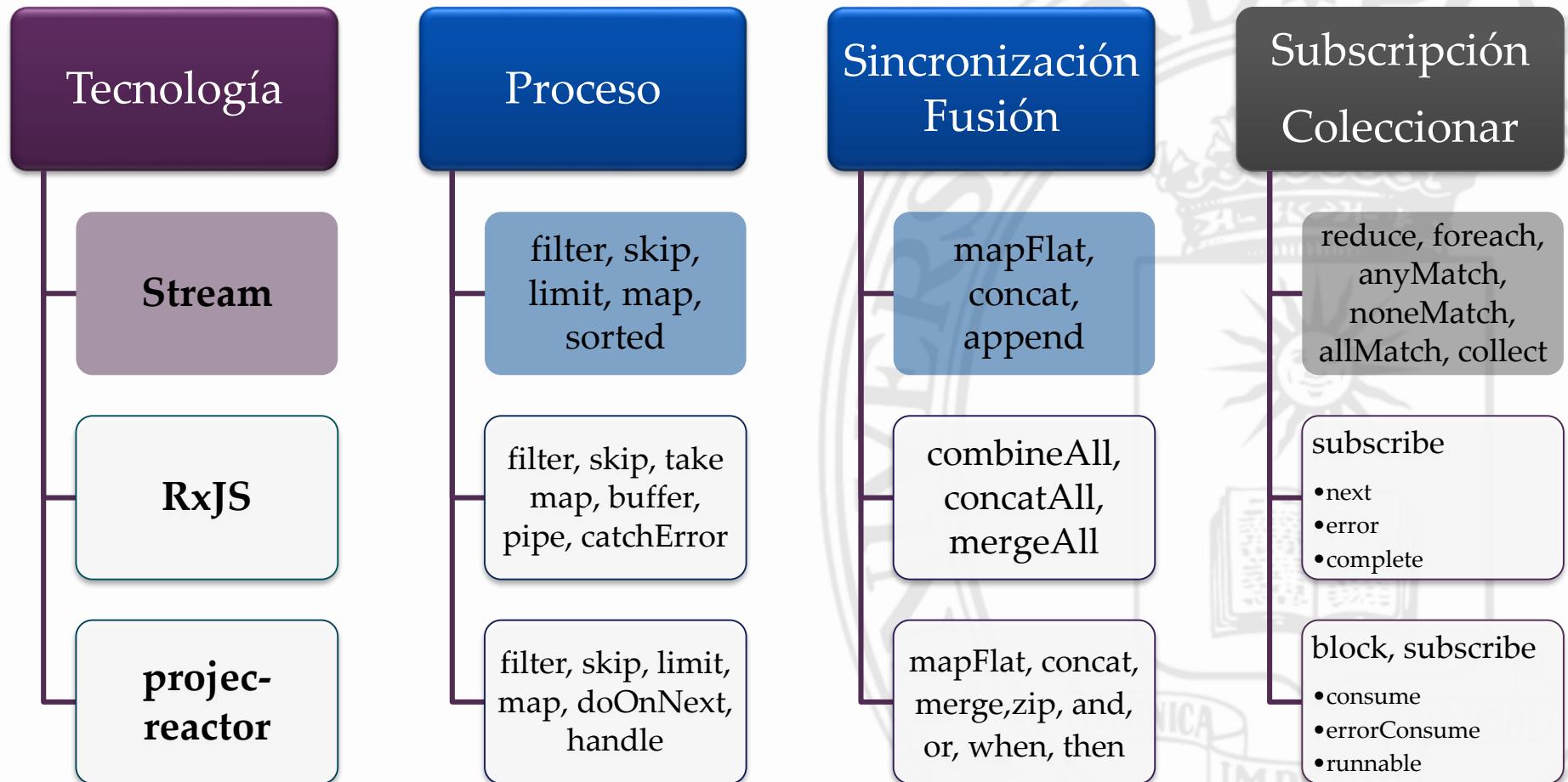
Reactive



### Project-reactor - Java

- Java - Spring
- Clases: Mono & Flux
- <https://projectreactor.io/docs/core/release/reference>
- <https://projectreactor.io/docs/core/release/api>

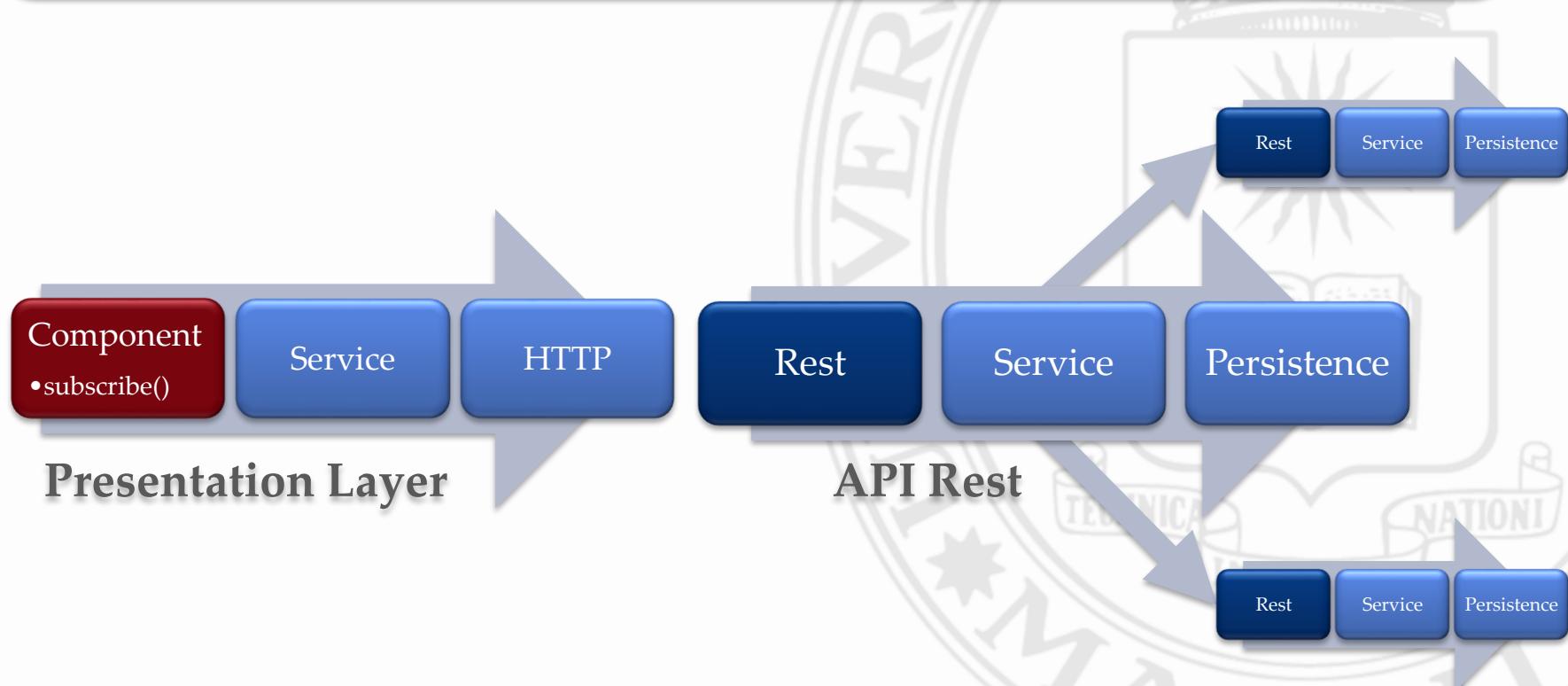
# Programación Reactiva



# Programación Reactiva

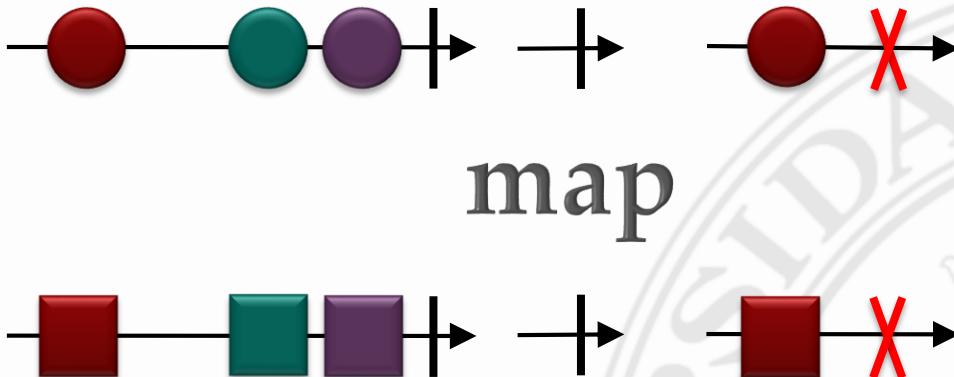
## Proyectos

Métodos que devuelven un tipo *Publisher* no deberían suscribirse porque ello *podría romper la cadena del publicador*



# Programación Reactiva

## Funciones



```
public Flux<Integer> convertToInteger(Flux<String> flux) {  
    return flux.map(Integer::valueOf);  
}
```

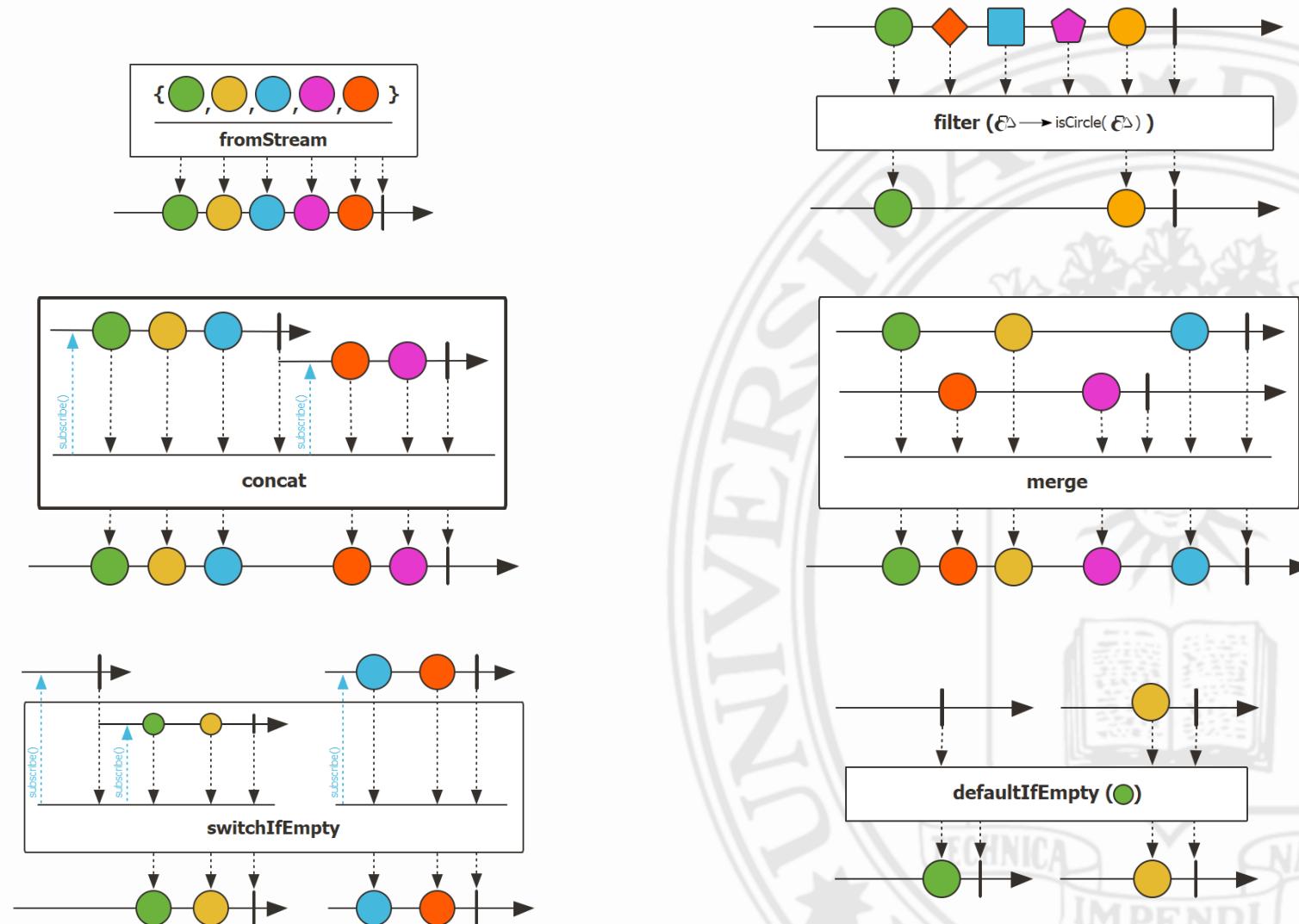


```
convertToNumber(flow: Observable<string>): Observable<number> {
  return flow.pipe(
    map(item => Number(item))
  );
}
```

```
createArticle() {
  this.articleService.create(this.newArticle).subscribe(
    item => this.item = item;
  );
}
```

# Programación Reactiva

## Funciones



# Programación Reactiva



<https://github.com/miw-upm/apaw>

- Package: *es.upm.miw.reactive*

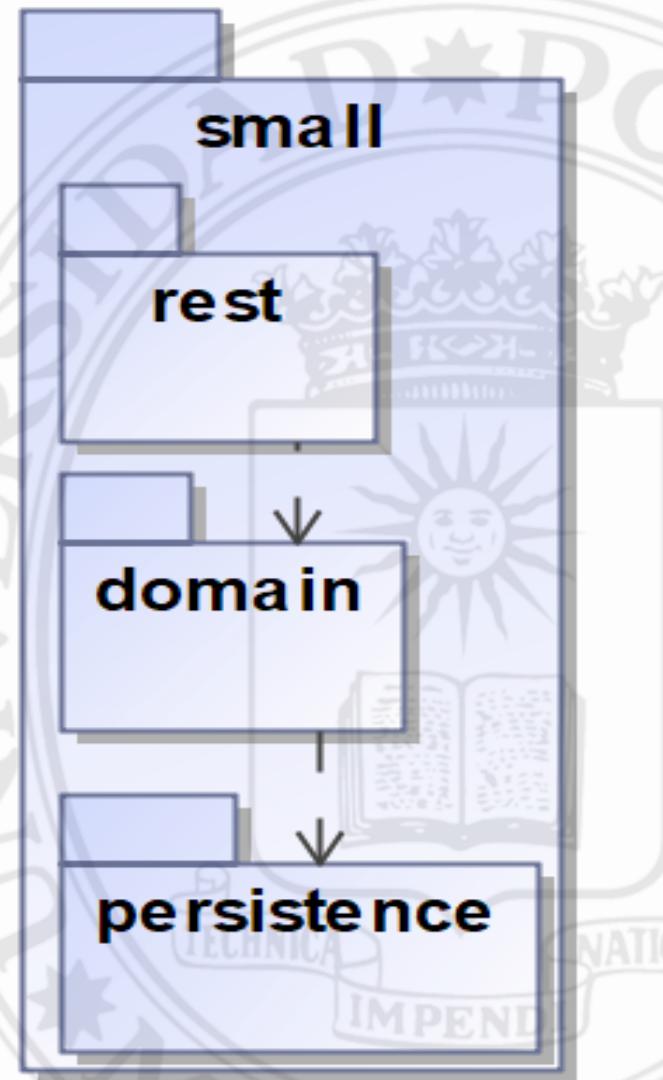
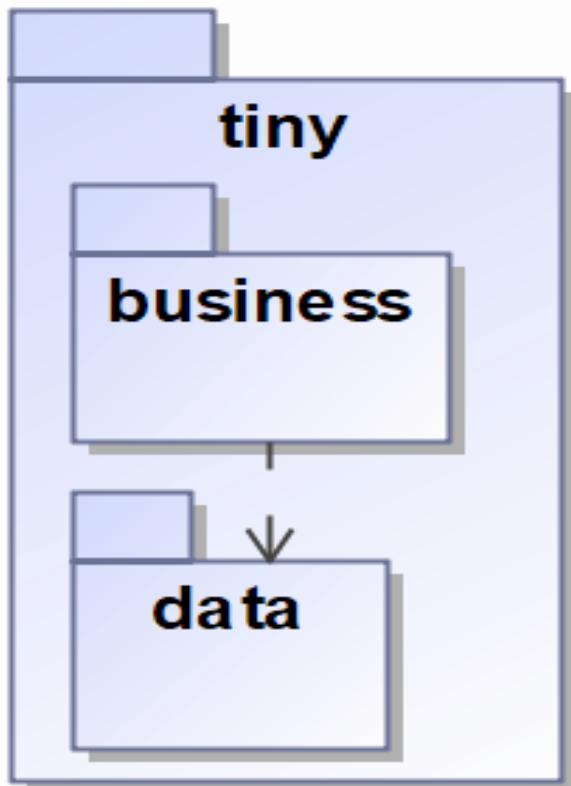
## Reactive in action

- *StreamAsynchronous*
- *ReactiveService*
- *ReactiveComponent*

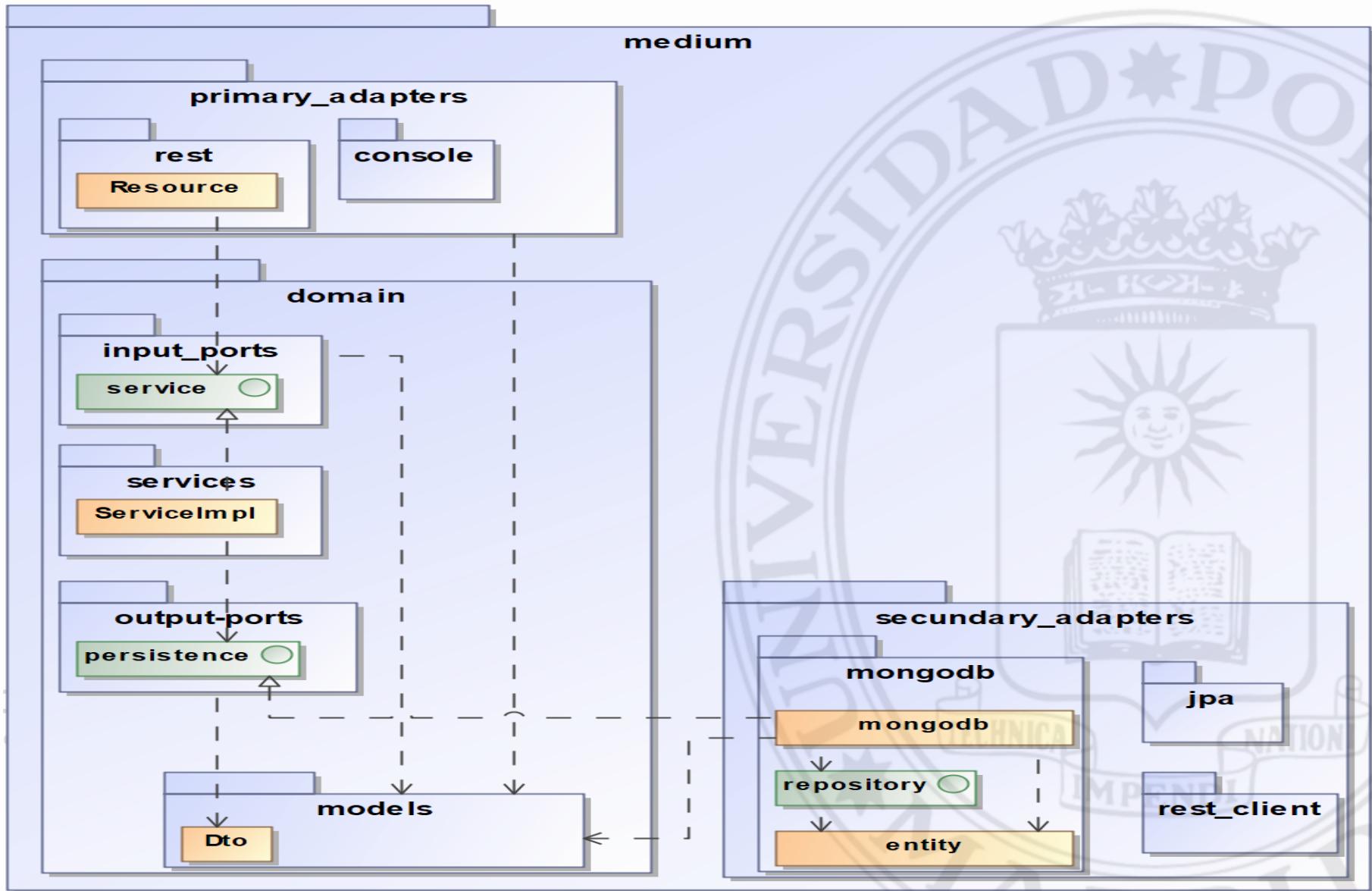
## Ejercicios

- Realizar nuevos métodos

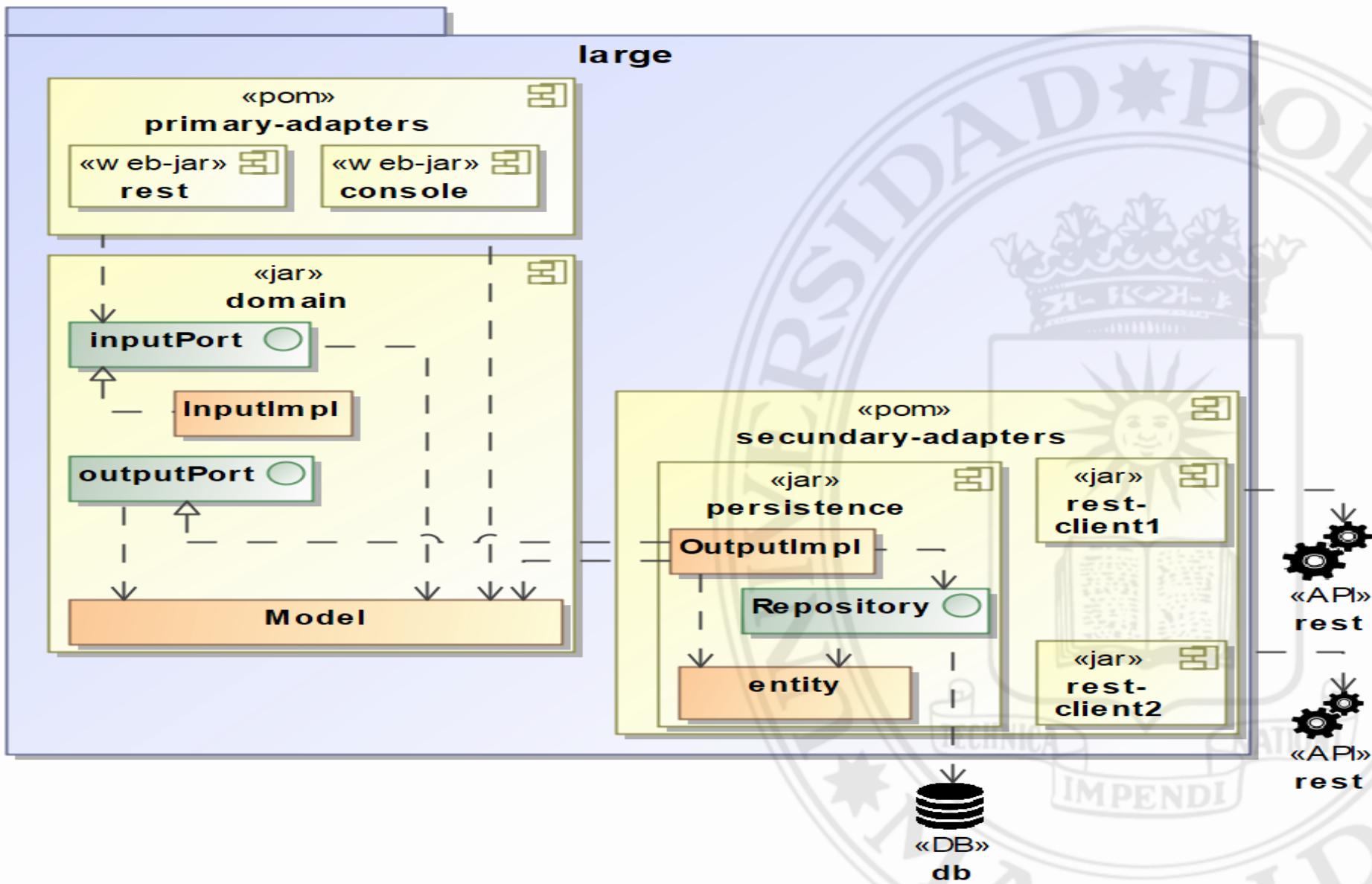
# Arquitectura por Capas Paquetes

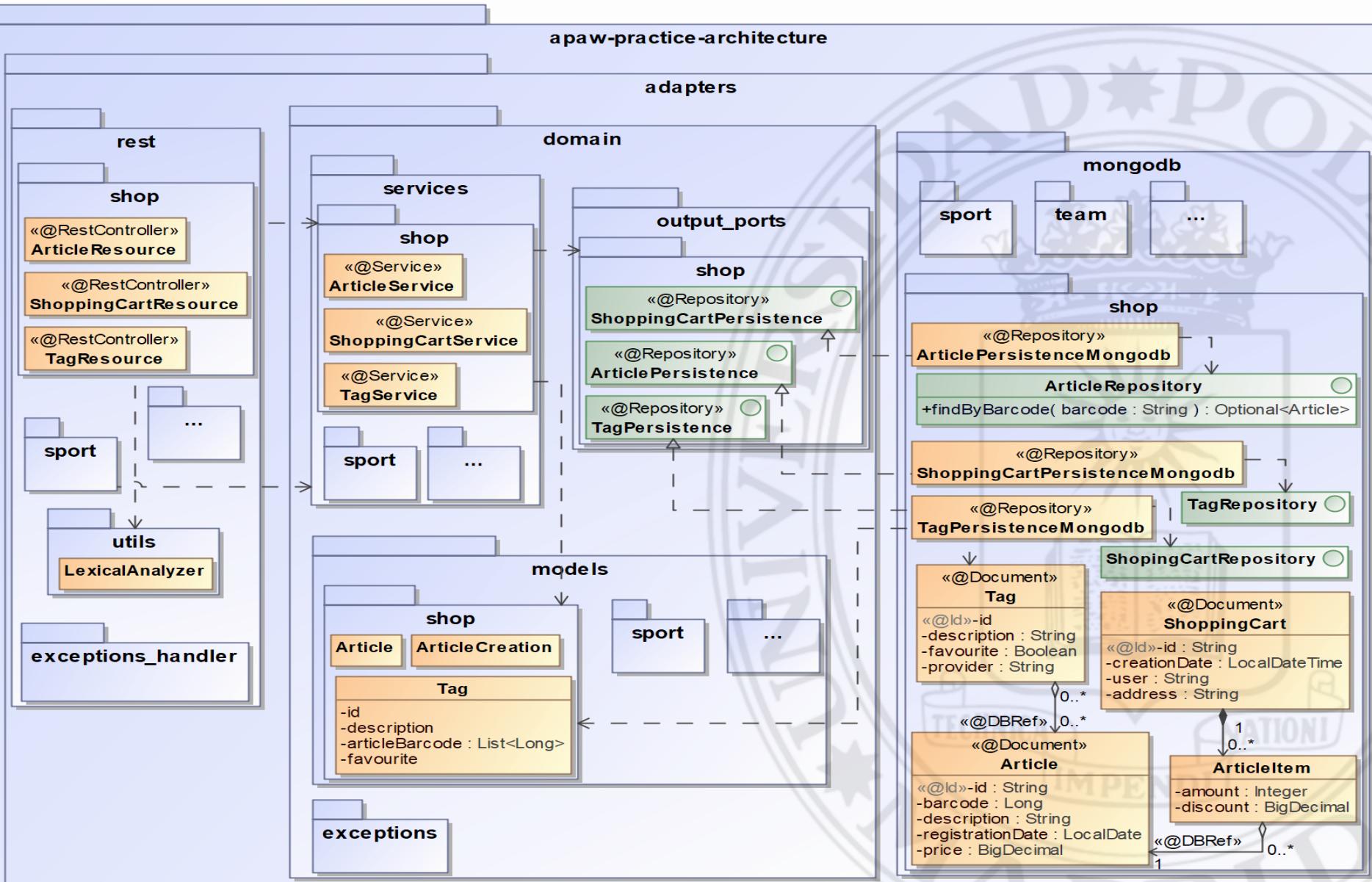


# Arquitectura Paquetes



# Arquitectura por Capas Paquetes



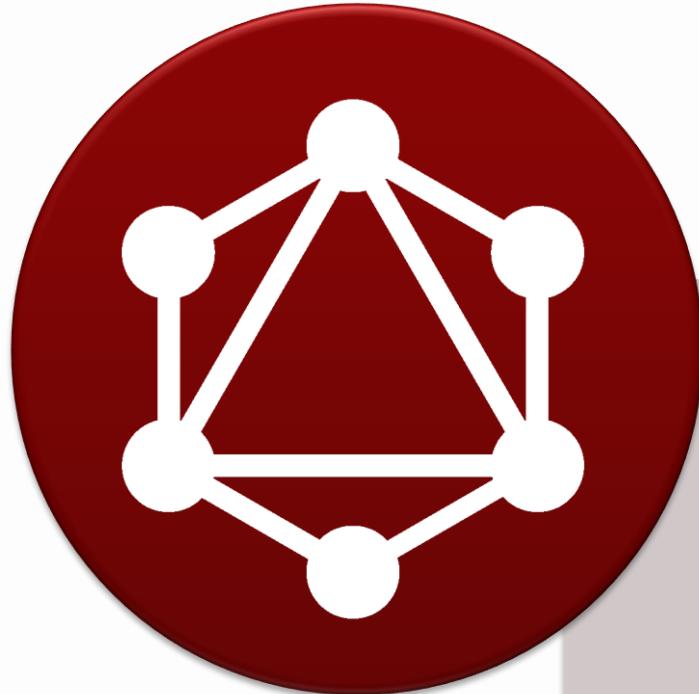




<https://github.com/miw-upm/apaw-practice>

- *main* es.upm.miw.apaw\_practice.business.services.shop
- *test* es.upm.miw.apaw\_practice.business.services.shop

# ¿Qué es GraphQL?



Es un lenguaje  
de consulta y  
modificación  
de un API.

# GraphQL

## Características

### GraphQL

- Desarrollado por Facebook y liberado en 2015
- Puede funcionar sobre HTTP y en un solo end-point:  
/graphql
  - GET ?query=\*\*\*
  - POST {"query":....}
- El servidor describe un esquema (*schema.graphqls*) del API.
- El cliente especificar los detalles de los datos solicitados.
- Cliente de prueba: GraphiQL.
- Existen librerías para diferentes lenguajes.
  - JavaScript, Java, Python, Go...

# GraphQL

## Características

```
#article.graphqls x
```

```
18   input ArticlePriceUpdating {
19     barcode:Int!
20     price: String!
21   }
22
23   type Article {
24     id: String!
25     registrationDate: String!
26     barcode: Int!
27     description: String
28
29
30 }
```

Datos

```
#Article
extend type Query {
  articles:[Article]
}
extend type Mutation {
  createArticle(articleCreation:ArticleCreation):Article
}
```

Operaciones

GraphiQL  Prettify

```
1 query {
2   articles {
3     barcode
4     registrationDate
5   }
6 }
```

Petición

```
{
  "data": {
    "articles": [
      {
        "barcode": 84001,
        "registrationDate": "2020-09-25"
      },
      {
        "barcode": 84002,
        "registrationDate": "2020-09-25"
      },
      {
        "barcode": 84003,
        "registrationDate": "2020-09-25"
      }
    ]
  }
}
```

Respuesta

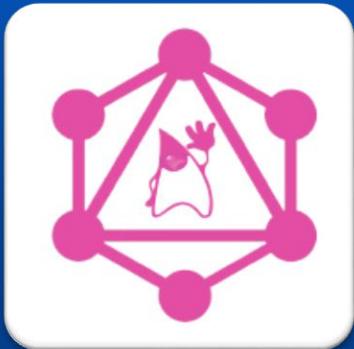
# GraphQL

## Implementaciones



### graphql-java-kickstart

- <https://www.graphql-java-kickstart.com>
- <https://github.com/graphql-java-kickstart/samples>



### graphql-java

- <https://www.graphql-java.com/>

<https://github.com/miw-upm/apaw-shop-architectures>

- Proyecto: *shop-three-layers-graphql*

## Reactive in action

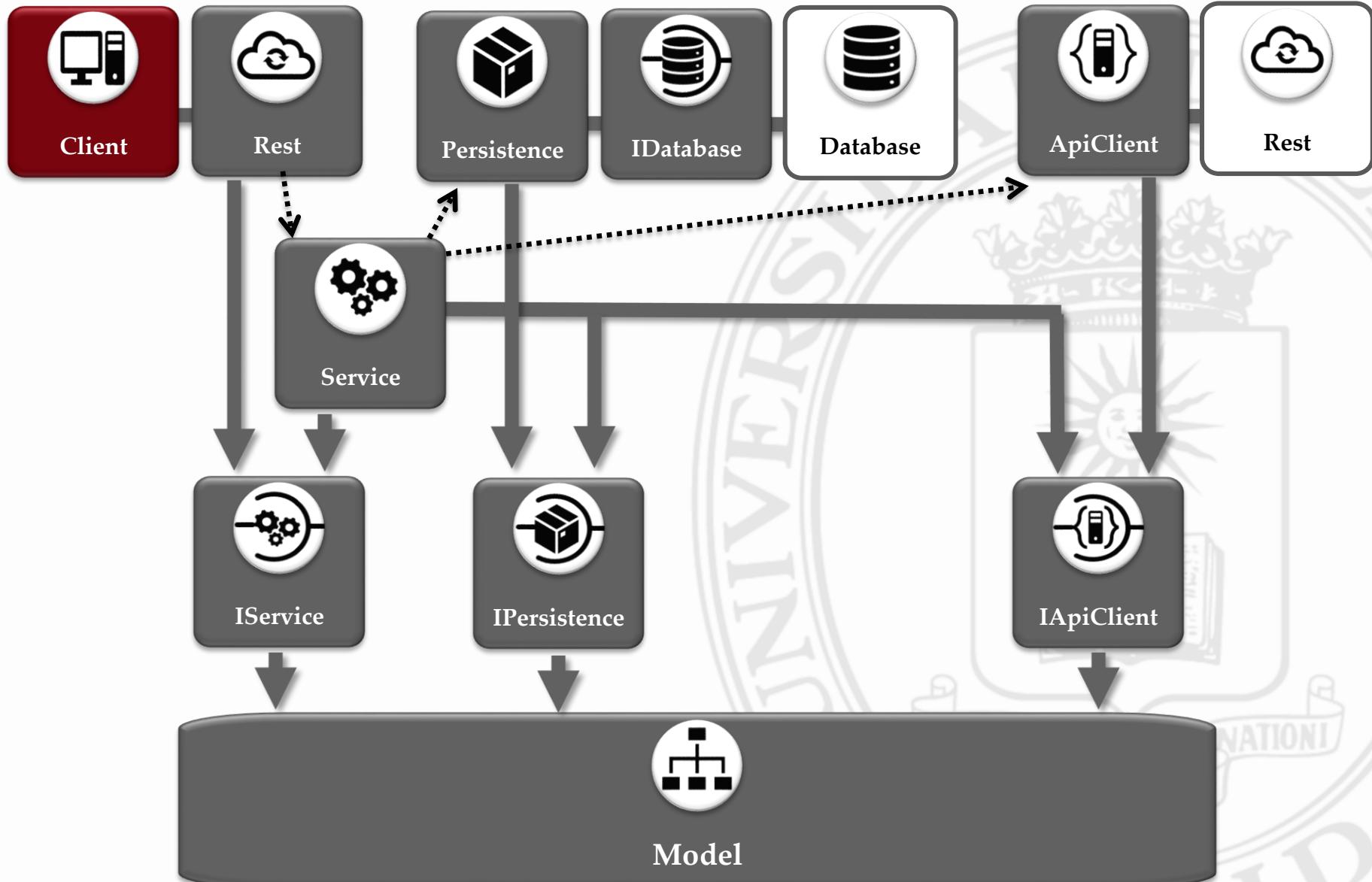
- *schema.graphqls*
- \*Query & \*Mutation

## 📝 Ejercicios

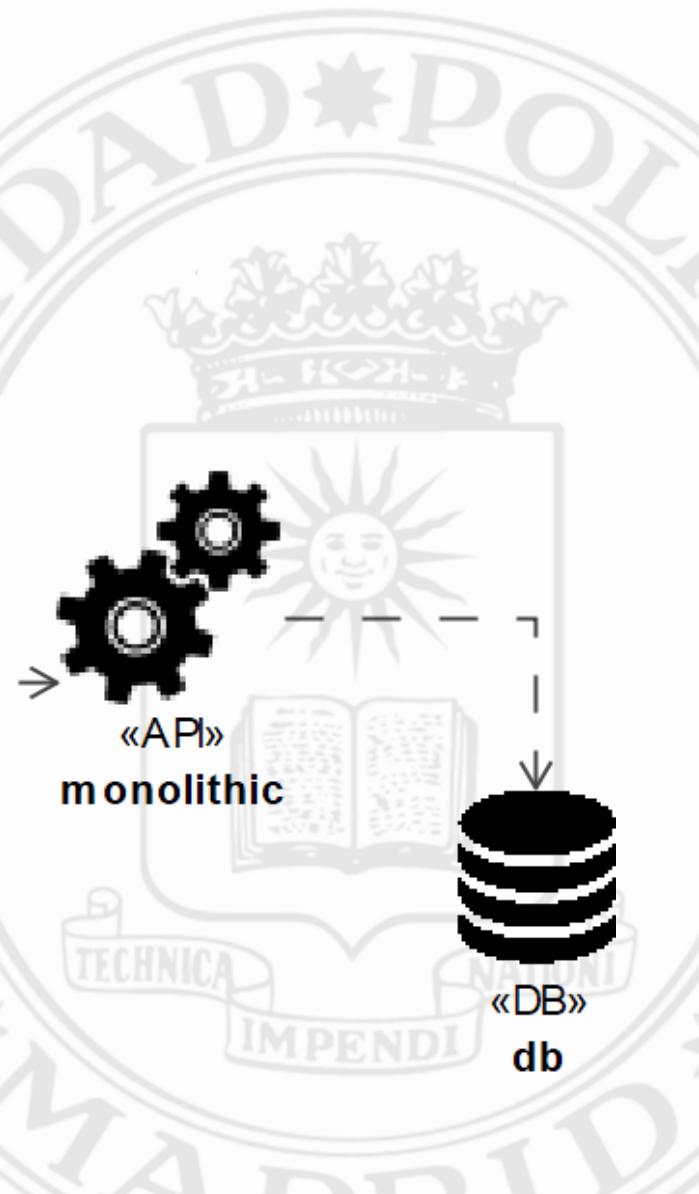
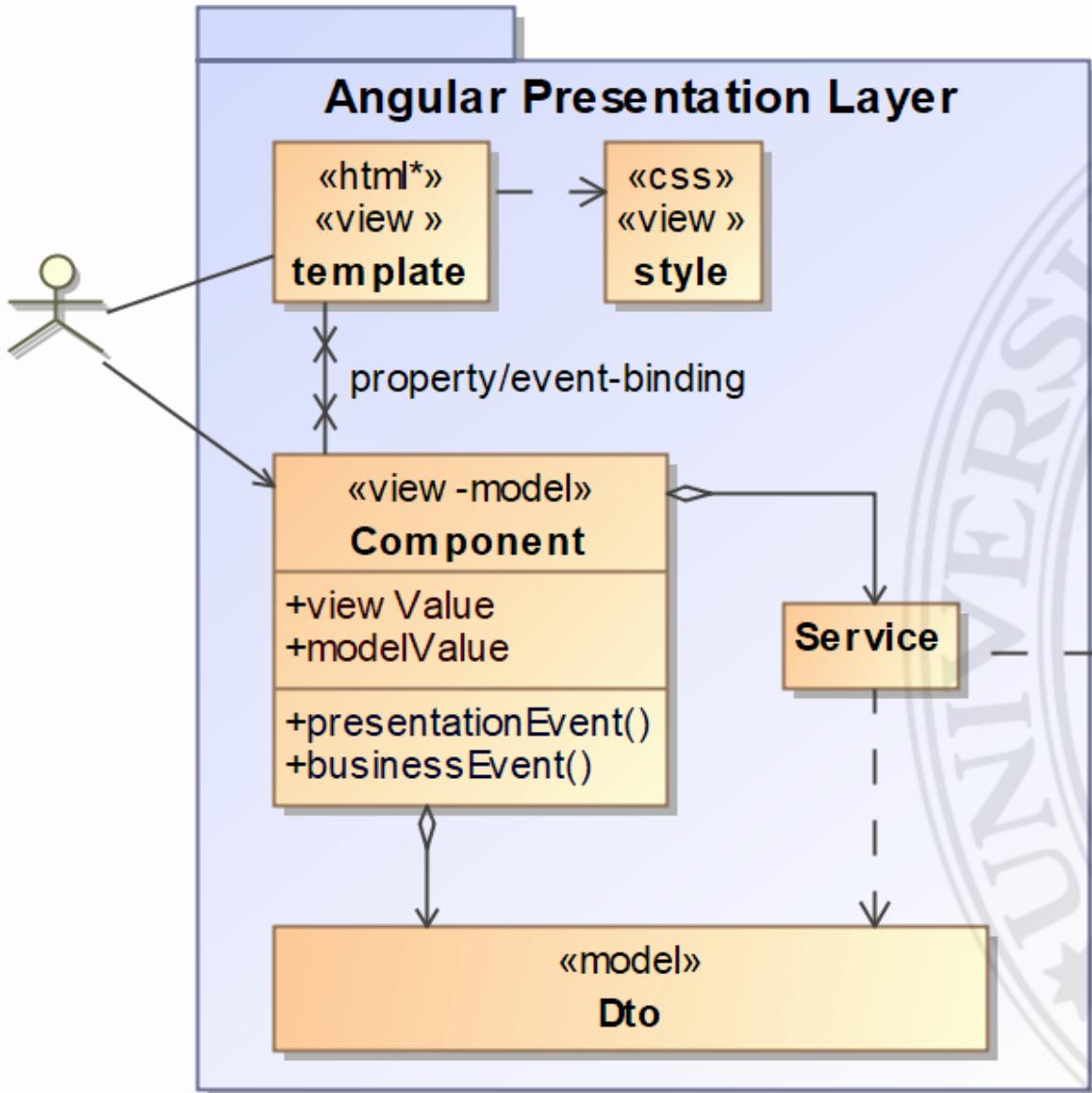
- Probar varias peticiones

# Presentación

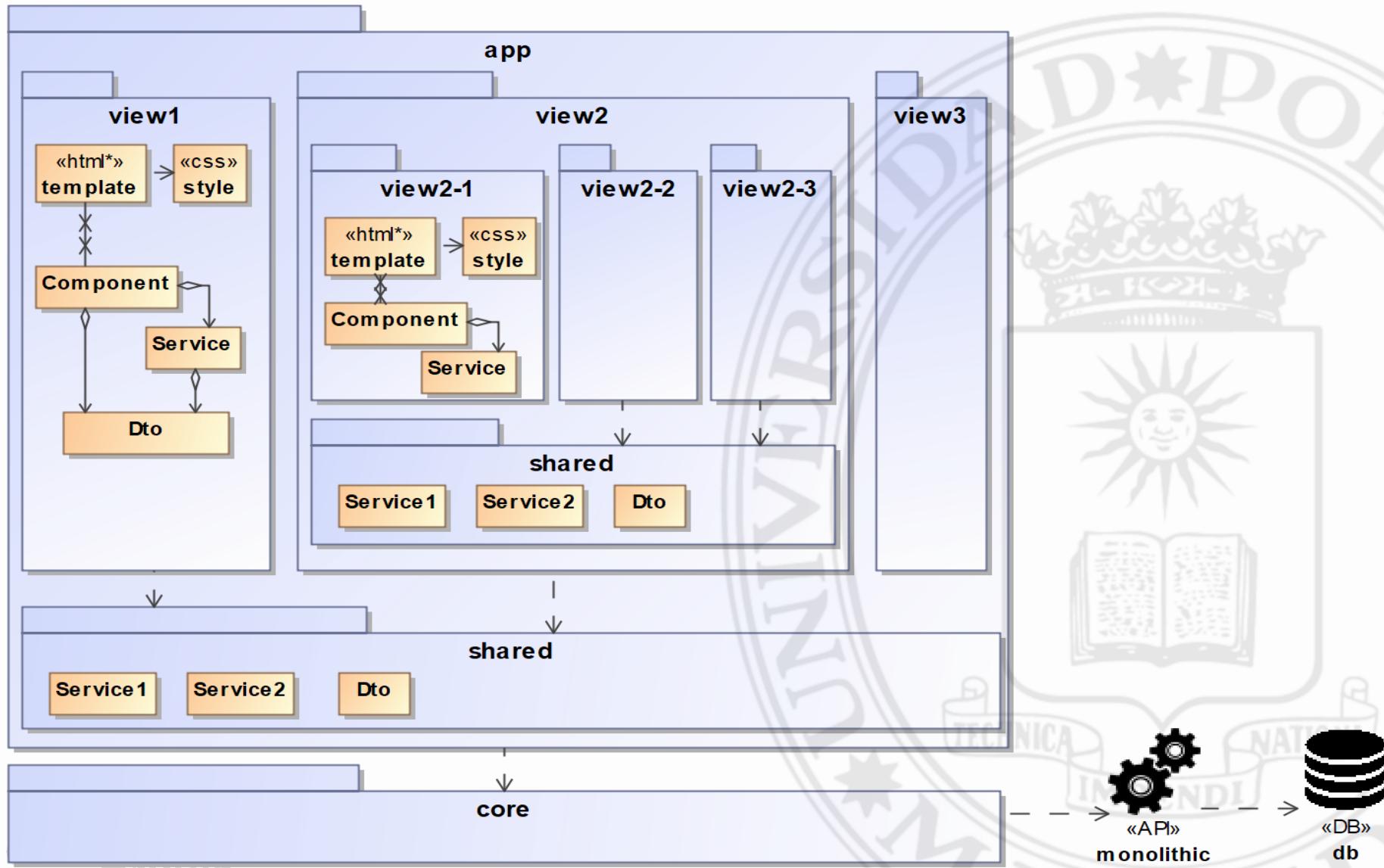
## Cliente



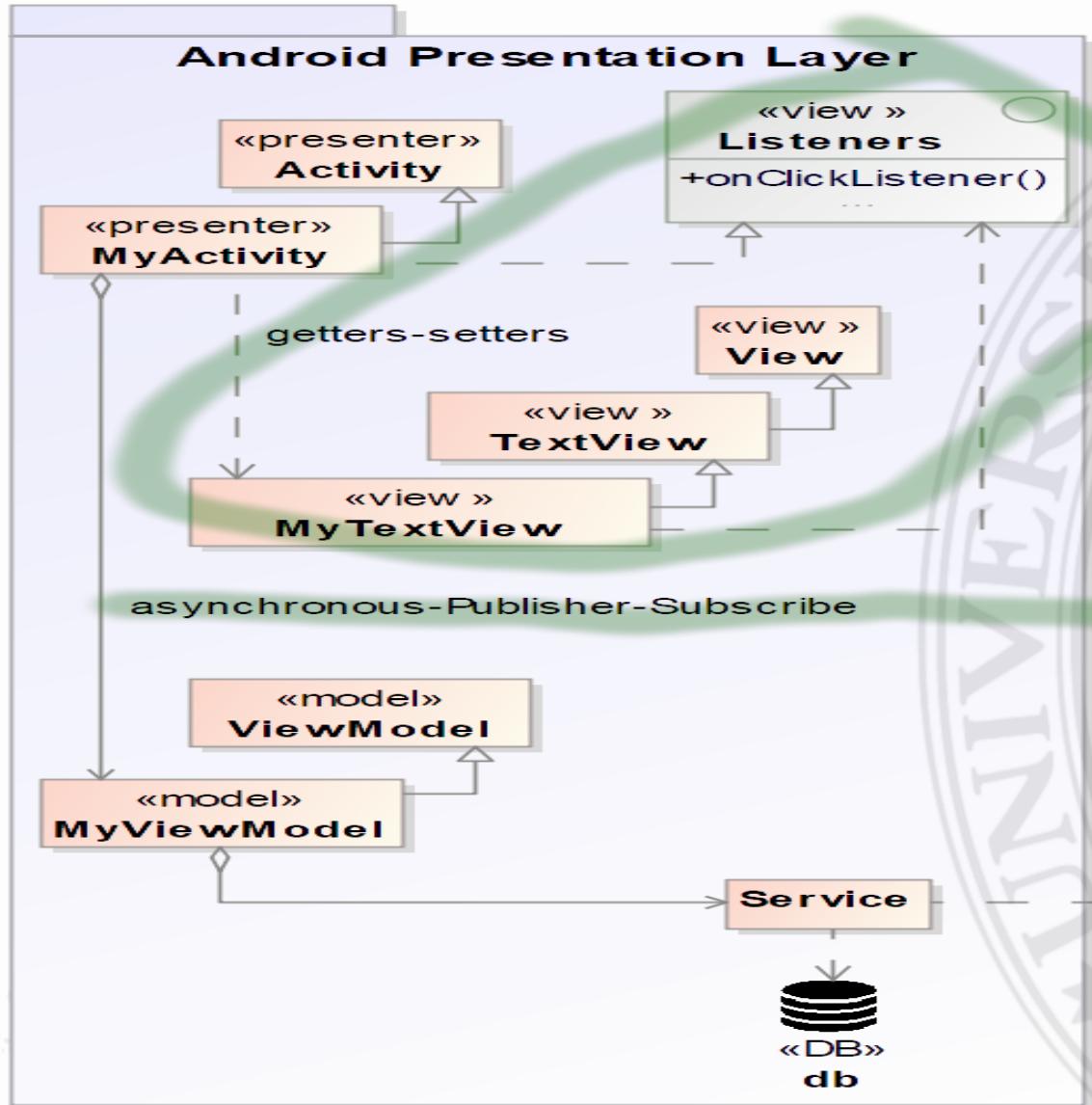
# Angular Presentación



# Angular Presentación



# Android Presentación



Por hacer...

# Microservicios

- Pattern
- Spring-Cloud