

Introducción al diseño de software

Diseño de Sistemas Software
Curso 2020/2021

Carlos Pérez Sancho



Universitat d'Alacant
Universidad de Alicante

Departament de Llenguatges i Sistemes Informàtics
Departamento de Lenguajes y Sistemas Informáticos

¿Qué es el diseño de software?

Es una disciplina de la Ingeniería de Software...

“Software engineering is the discipline of developing and maintaining software systems that behave reliably and efficiently, are affordable to develop and maintain, and satisfy all the requirements that customers have defined for them.”

(IEEE Computing Curricula 2005)

“(Students should be able to) Demonstrate an understanding of and apply appropriate theories, models, and techniques that provide a basis for problem identification and analysis, software design, development, implementation, verification, and documentation.”

(IEEE Software Engineering 2014)

Diseño de software

- Análisis, diseño e implementación son disciplinas centrales de la ingeniería del software
- Están fuertemente relacionadas, y las decisiones sobre cómo realizar una de ellas afecta directamente a las demás
- Lenguajes, herramientas, actividades a realizar, etc. deben decidirse al comienzo de un proyecto

“Software design is both the process of defining the architecture, components, interfaces and other characteristics of a system, and the result of that process”.

(IEEE Computing Curricula 2005)

Diseño de software

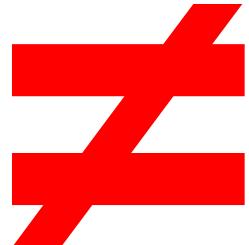
- **Diseño explícito:** realizado para planificar o documentar el software desarrollado
 - **Diseño implícito:** la estructura que el software tiene realmente, aunque no se haya diseñado formalmente
- **Todo el software tiene un diseño**

Fases del diseño

- **Diseño arquitectural:** estructura de alto nivel, identificación de los componentes principales junto con los requisitos funcionales y no funcionales
- **Diseño detallado:** los componentes se descomponen con un nivel de detalle más fino. Guiado por la arquitectura y los requisitos funcionales

¿Por qué es importante el diseño?

¿Por qué diseñar?



```
2. vim  
#include <stdio.h>  
  
int main() {  
    printf("Hello world!\n");  
}  
  
-- INSERT --
```



Ventajas de un buen diseño

- Ayuda a trasladar las especificaciones a los programadores de forma no ambigua
- Ayuda a escribir código de calidad
 - ... conforme a las especificaciones
 - ... fácil de mantener y extender
 - ... que los demás puedan comprender

Aumenta las probabilidades de finalizar con éxito el proyecto

Diseñar bien no es garantía de éxito

- Realizar un diseño previo no garantiza que el software resultante sea conforme a ese diseño ni a la especificación
- Existen disciplinas y técnicas para garantizar la calidad del software (p.ej. pruebas de código)

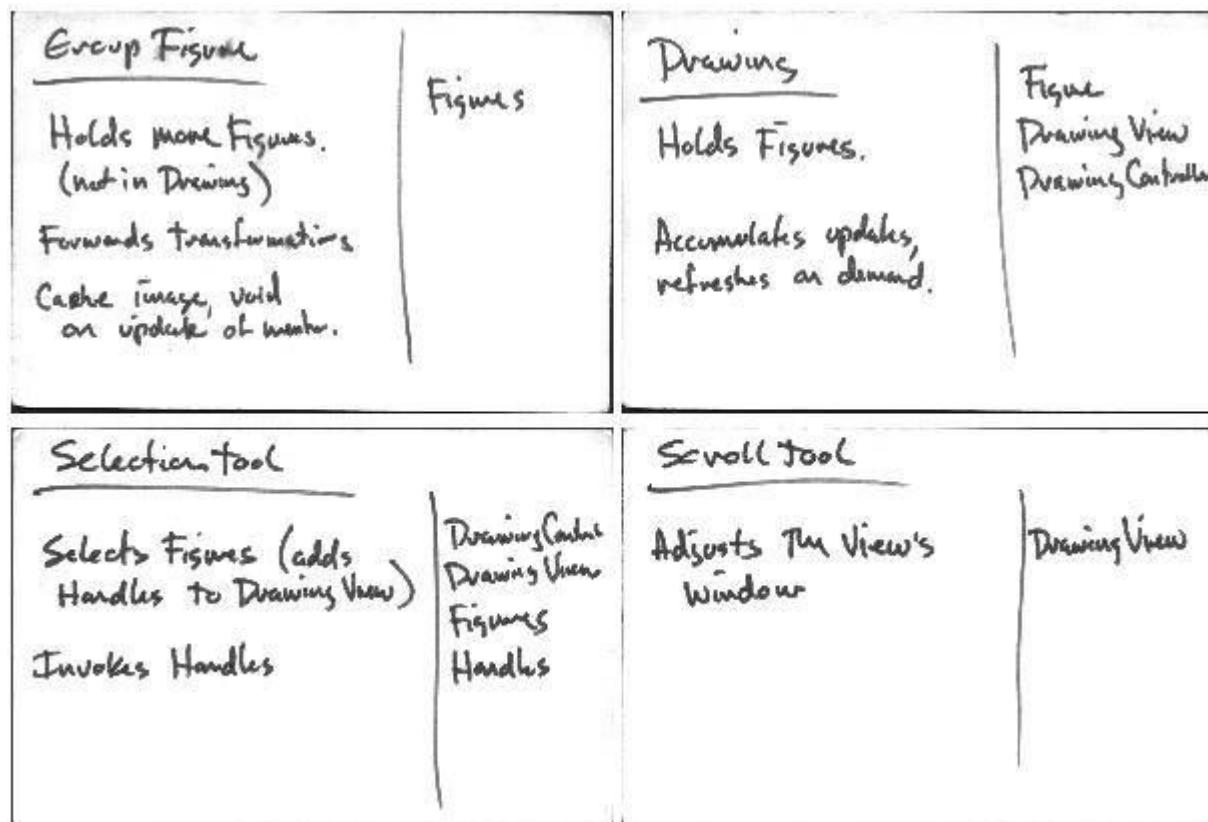
¿Cómo y cuánto hay que diseñar?

(Algunos) Paradigmas de diseño

- **Diseño orientado a objetos:** los sistemas de software se componen de objetos con un estado privado y que interactúan entre sí
- **Diseño orientado a funciones:** descompone el sistema en un conjunto de funciones que interactúan y comparten un estado centralizado
- **Diseño de sistemas de tiempo real:** reparto de responsabilidades entre software y hardware para conseguir una respuesta rápida
- **Diseño de interfaces de usuario:** tiene en cuenta las necesidades, experiencia y capacidades de los usuarios

Herramientas de Diseño Orientado a Objetos

- Tarjetas CRC



Herramientas de Diseño Orientado a Objetos

- UML, diagramas más usados según Sommerville (2010)
 - **Diagramas de actividad:** actividades involucradas en un proceso o en el procesamiento de datos
 - **Diagramas de caso de uso:** interacciones entre el sistema y su entorno
 - **Diagramas de secuencia:** interacciones entre los actores y el sistema y entre componentes del sistema
 - **Diagramas de clase:** clases de objetos y sus asociaciones
 - **Diagramas de estados:** cómo reacciona el sistema ante eventos internos y externos

UML no hace milagros



¡ATENCIÓN!

UML no es más que un formato de representación

Usar UML no garantiza que el diseño sea bueno

Formas de usar UML

- Como un medio para facilitar el debate sobre un sistema existente o propuesto
- Como una forma de documentar un sistema existente
- Como una descripción detallada que se puede usar para desarrollar una implementación
- Usando herramientas especializadas, los modelos se pueden usar para generar código automáticamente (Model-Driven Engineering, MDD)

La metodología impone las reglas

- En metodologías tradicionales hay que generar numerosos modelos antes de tocar una sola línea de código
- Las metodologías ágiles recomiendan modelar lo mínimo imprescindible, en grupo y de manera informal
⇒ los modelos se usan como herramienta de comunicación

“For a three-week timeboxed iteration, spend a few hours or at most one day (with partners) near the start of the iteration *at the walls...*”

(Larman, 2004, ch. 14.3)

¿Preguntas?

Bibliografía

- [IEEE Computing Curricula 2005](#)
- [IEEE Software Engineering 2014](#)
- Larman, C. (2004). ***Chapter 14. In Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd edition.*** Addison Wesley Professional.
[Leer en Safari Books Online](#)
- Tsui, F., Karam, O., Bernal, B. (2013). ***Chapters 2 & 7. In Essentials of Software Engineering, 3rd edition.*** Jones & Bartlett Learning.
[Leer en Safari Books Online](#)
- Sommerville, I. (2010). ***Software Engineering, Ninth edition.*** Addison-Wesley.

Patrones arquitecturales

Arquitectura en capas

Diseño de Sistemas Software
Curso 2020/2021

Carlos Pérez Sancho



Universitat d'Alacant
Universidad de Alicante

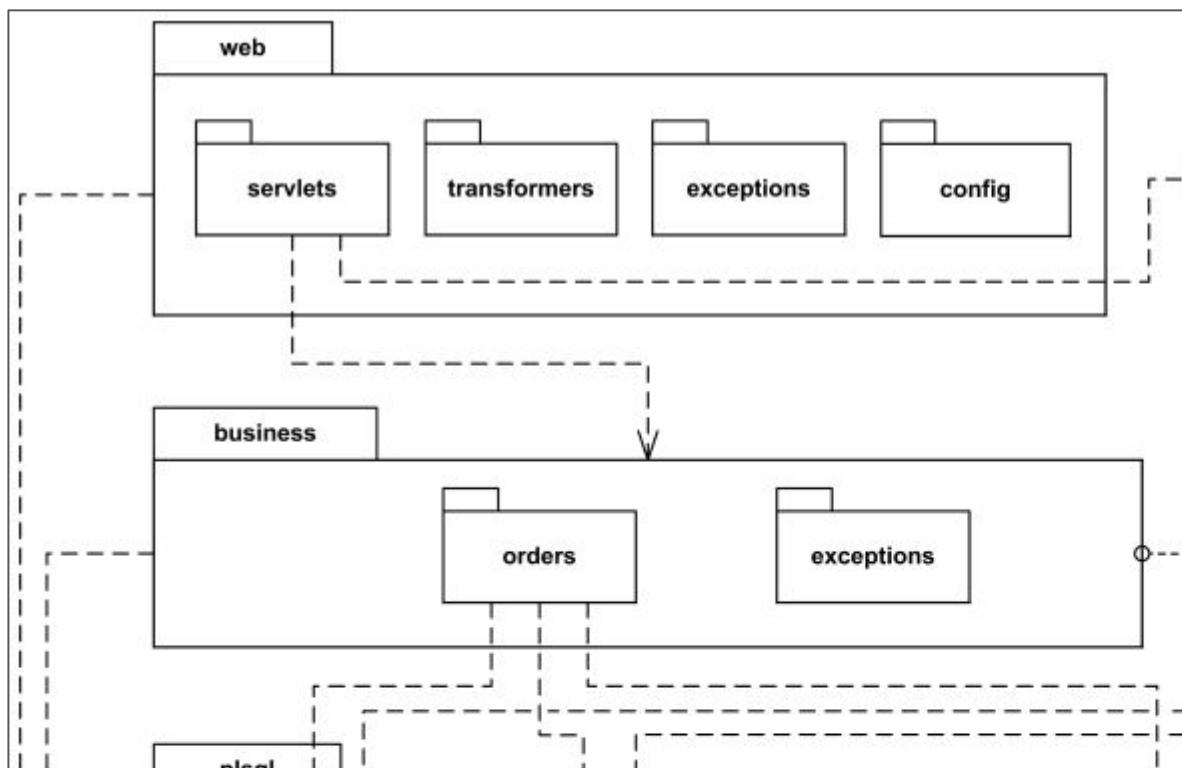
Departament de Llenguatges i Sistemes Informàtics
Departamento de Lenguajes y Sistemas Informáticos

Arquitectura de software

- Distintas definiciones
 - “Descomposición de un sistema en componentes de alto nivel”
 - “Decisiones difíciles de cambiar”
 - “La arquitectura es el puente entre los objetivos de negocio (a menudo abstractos) y el sistema resultante (concreto)”

Arquitectura de software

- Un diseño arquitectural es una abstracción que muestra únicamente los detalles relevantes



Arquitectura de software

- La definición de una arquitectura debe incluir también una descripción del comportamiento de sus componentes
- La arquitectura prescribe la manera en la que se deben implementar los componentes y cómo deben interactuar, normalmente mediante restricciones
- Se deben tener en cuenta los requisitos no funcionales (p.ej. rendimiento esperado) y la distribución lógica y física de sus componentes

Arquitectura de software

- Las aplicaciones que no tienen una arquitectura formal, normalmente son
 - Altamente acopladas
 - Frágiles
 - Difíciles de cambiar
 - No tienen una visión o dirección claras
 - Es muy difícil determinar las características arquitecturales de una aplicación sin comprender el funcionamiento interno de cada componente y módulo en el sistema

Arquitectura de software

- Hay muchas formas de diseñar mal, pero sólo unas pocas de hacerlo bien
- El éxito en el diseño arquitectural es complejo y desafiante, por eso los diseñadores han encontrado maneras de capturar y reutilizar el conocimiento adquirido en otros sistemas
- Los patrones arquitecturales son formas de capturar estructuras de diseño de eficacia probada, de manera que puedan ser reutilizadas

¿Qué es un patrón de diseño?

A hombros de gigantes

- La mayoría de problemas de diseño no son nuevos, sólo cambia el dominio de la aplicación
- No sólo se reutiliza el código, también podemos reutilizar soluciones anteriores
- Podemos reutilizar la experiencia de los expertos documentada a través de **patrones de diseño**

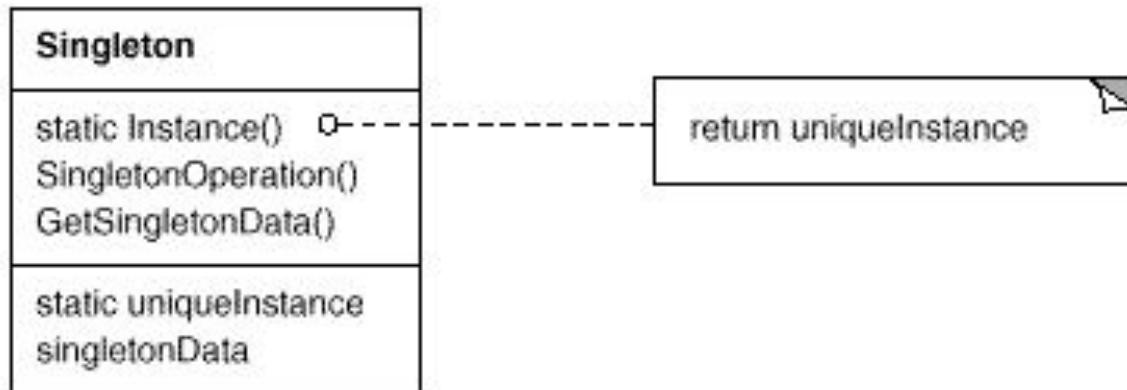
Patrones de diseño

- Son soluciones documentadas a problemas frecuentes en el diseño de software
 - Nombre del patrón
 - Problema y su contexto
 - Solución
 - Consecuencias
- Conocer los patrones de diseño más habituales ayuda a comprender mejor los sistemas, incluso observando directamente el código

Ejemplo

- Patrón **Singleton**
- Motivación: asegurar que una clase sólo tiene una única instancia, y proporcionar un punto de acceso global

- Solución:



- Consecuencias: acceso controlado a la única instancia, mejor que el uso de variables globales, ... (más en la bibliografía)

De vuelta a la arquitectura de software

Patrones arquitecturales

- Los patrones arquitecturales expresan esquemas de organización estructural fundamentales
- Proporcionan un conjunto de subsistemas predefinidos, especifican sus responsabilidades e incluyen reglas y principios para organizar sus relaciones
- Para los patrones más habituales es frecuente encontrar **frameworks** que permiten implementar una aplicación sin tener que escribirla desde cero

Frameworks

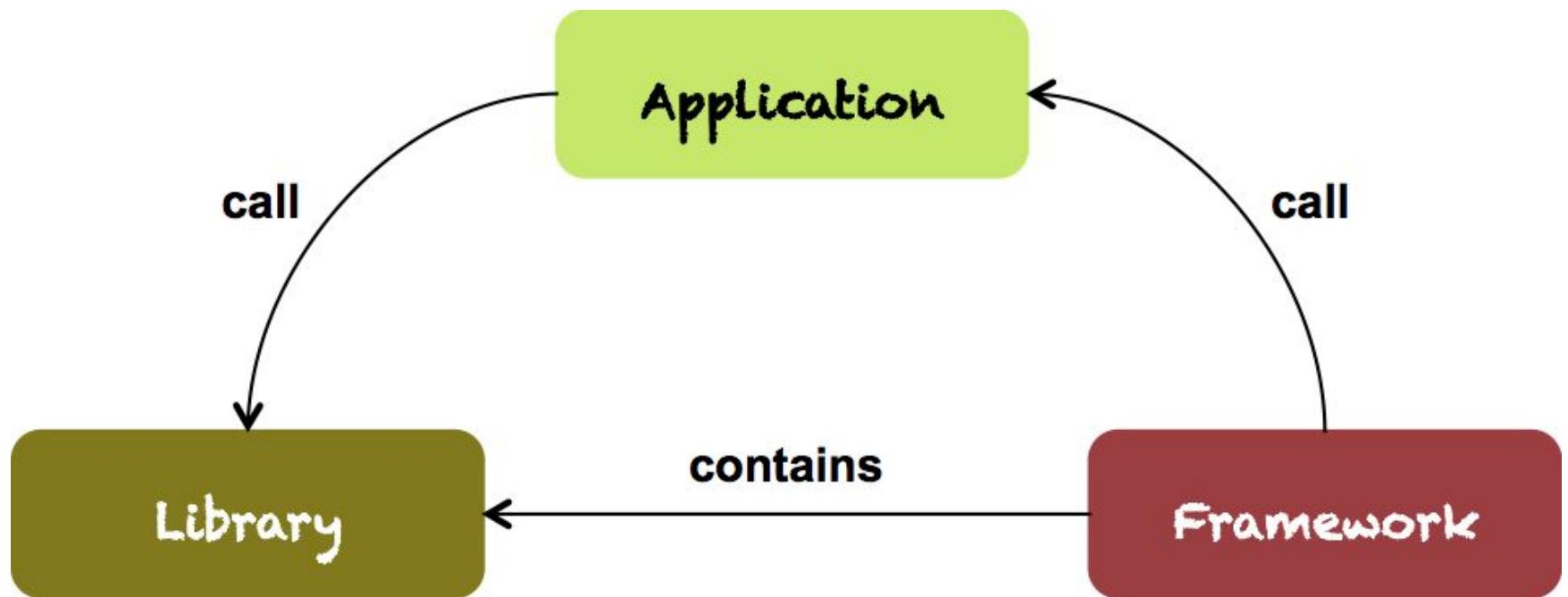
Frameworks

- Un framework es una aplicación reutilizable, “semi-completa” que se puede especializar para producir aplicaciones personalizadas
- Incorporan numerosos patrones de diseño en su implementación
- Muchos frameworks siguen el principio ***Convention over configuration***: no es necesario crear archivos de configuración si se respetan las convenciones de nombres (p.ej. mapeado de rutas en ASP.NET MVC)

Frameworks: principales características

- **Modularidad:** encapsulando los detalles de implementación detrás de interfaces estables
- **Reusabilidad:** definiendo componentes genéricos que se pueden reutilizar para crear nuevas aplicaciones
- **Extensibilidad:** proporcionando métodos “gancho” que permiten a las aplicaciones extender sus interfaces estables
- **Inversión de control** (Principio Hollywood): permite al framework (en lugar de a cada aplicación) determinar qué métodos de la aplicación serán invocados como respuesta a eventos externos

Frameworks: inversión de control



[Fuente](#)

Frameworks

- Beneficios de usar frameworks
 - Desarrollo más rápido
 - Código más estable y robusto
 - Mayor seguridad
 - Menor coste de desarrollo
 - Soporte de la comunidad

Frameworks

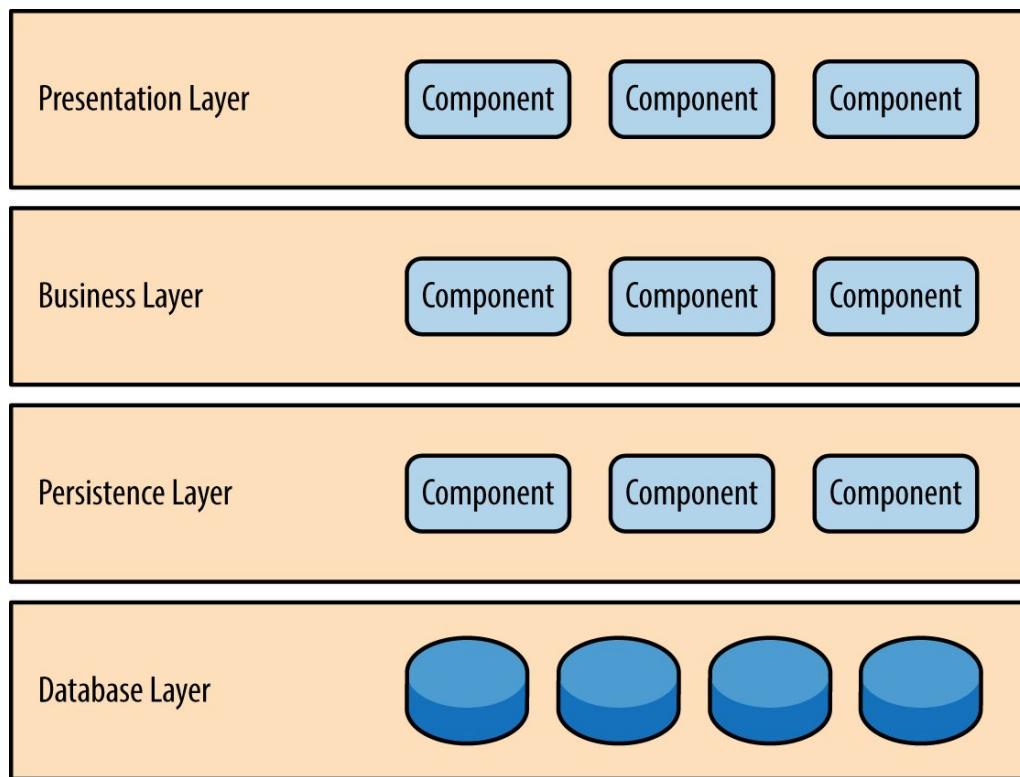
- Desventajas del uso de frameworks
 - Curva de aprendizaje
 - Problemas de integración con otros frameworks o librerías
 - Limitaciones
 - Es más difícil depurar el código
 - El código es público
- https://www.theregister.co.uk/2018/05/08/equifax_breach_may_2018/

Arquitectura en capas

Patrones arquitecturales

Arquitectura en capas

- Descompone una aplicación en componentes situados en distintos niveles horizontales llamados capas



Arquitectura en capas

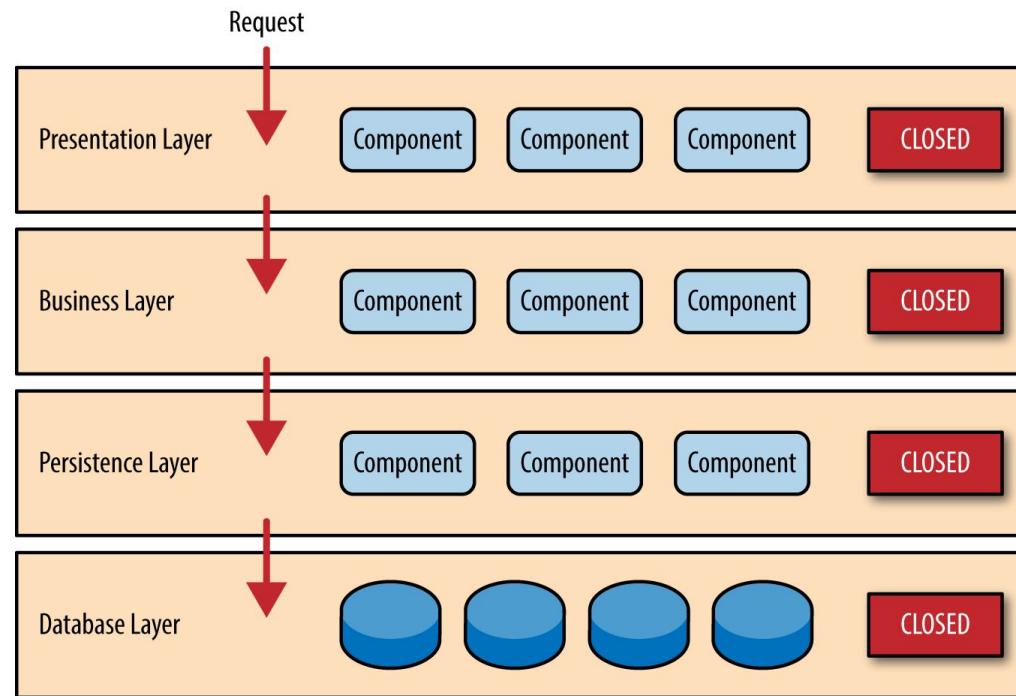
- Cada capa se encarga de una tarea específica, abstrayendo los detalles a las demás capas → separación de intereses
- Capas típicas (aunque el patrón no prescribe ninguna):
 - **Presentación:** interacción con el usuario
 - **Servicios:** funcionalidades de alto nivel
 - **Lógica de negocio:** ejecución de las reglas de negocio
 - **Persistencia (acceso a datos):** comunicación con la BBDD
 - **Base de datos:** almacenamiento de información

Arquitectura en capas

- La capa de lógica de negocio y la capa de persistencia no deben depender nunca de la capa de presentación
- La capa de lógica de negocio y la capa de persistencia pueden juntarse en una única capa en algunas circunstancias
- La capa de servicios ofrece funcionalidades compuestas a partir de las clases de la capa de lógica de negocio, ocultando su complejidad

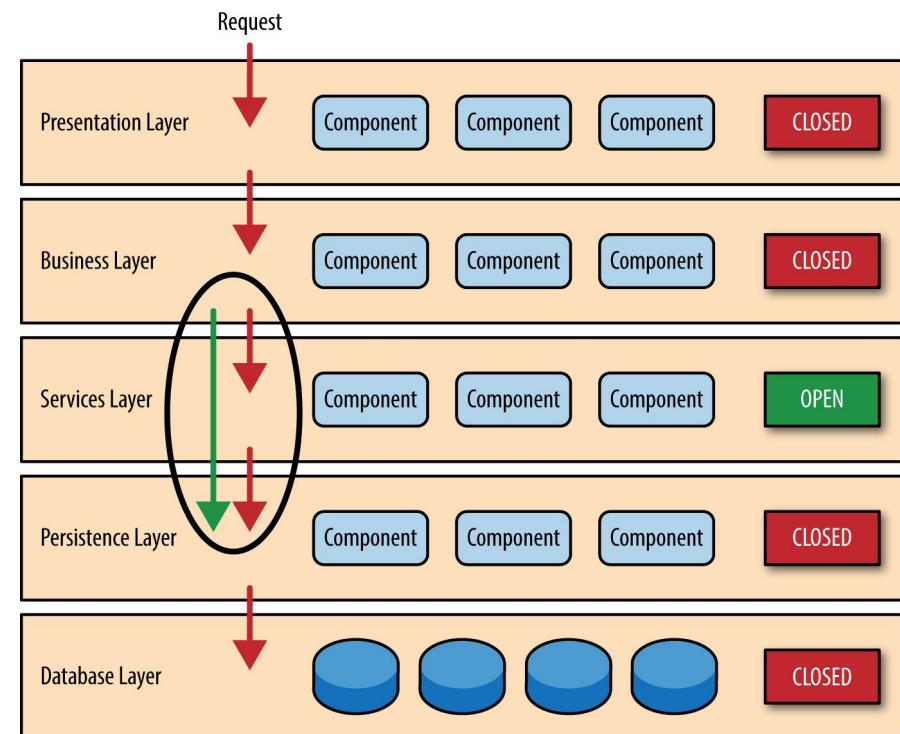
Arquitectura en capas

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



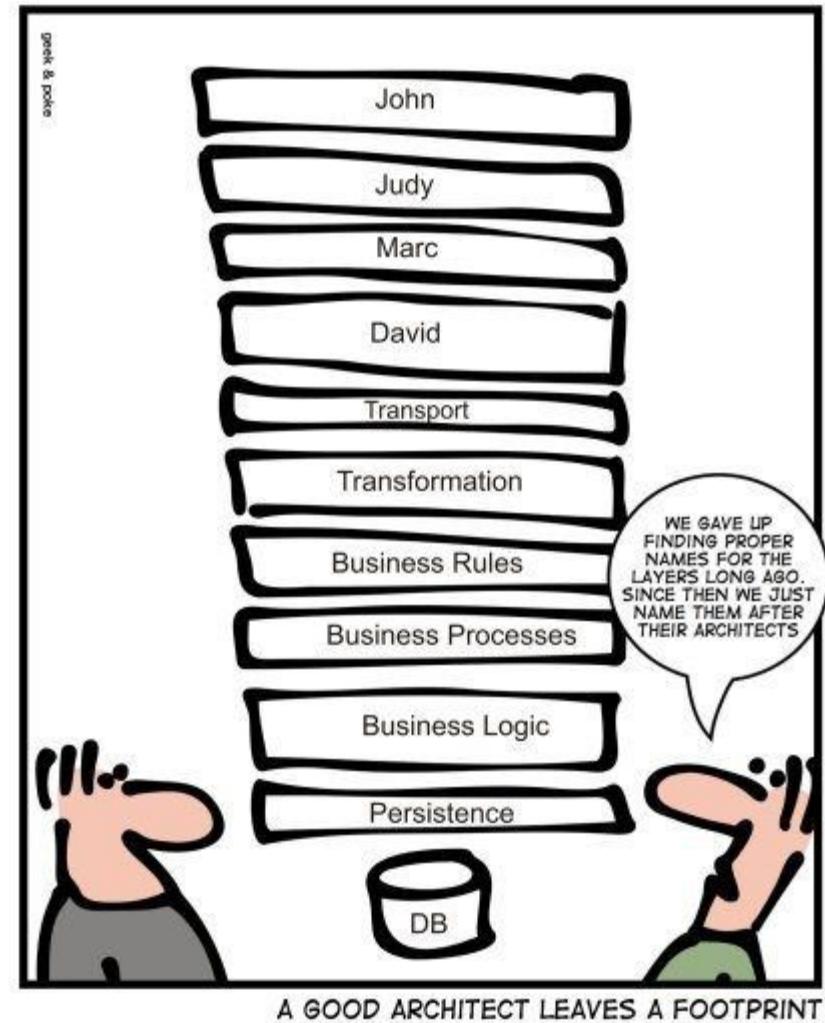
Arquitectura en capas

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



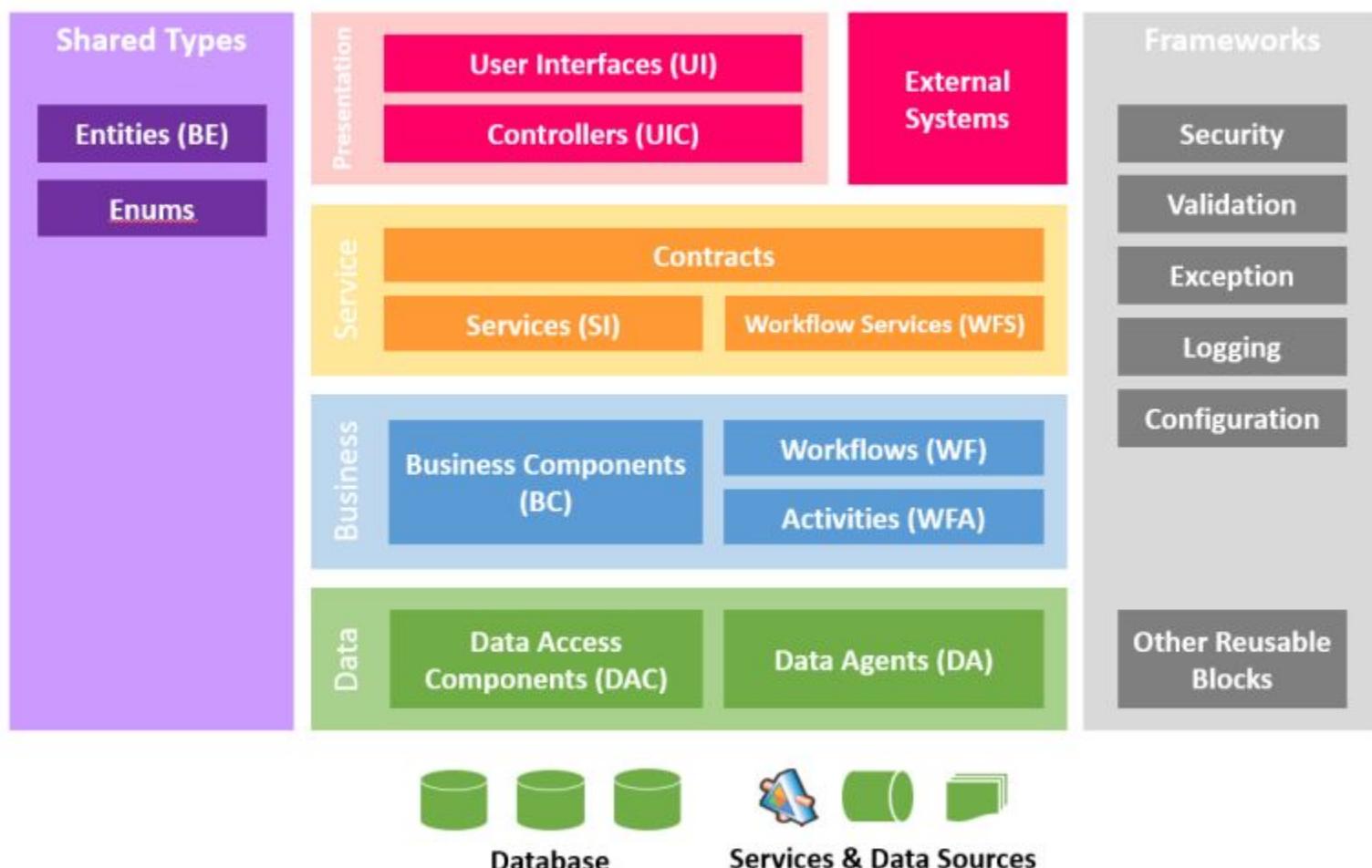
Arquitectura en capas

Se pueden crear tantas capas como sea necesario, siempre que cada capa tenga un propósito claro y separado de las demás



Ejemplo: arquitectura .NET

Layered Architecture



Fuente

Consideraciones

- Es la arquitectura más adecuada para la mayoría de aplicaciones
- No es necesaria cuando todas las peticiones pasan de una capa a otra sin procesamiento adicional (antipatrón sumidero)
- Cuando las capas no están distribuidas físicamente puede terminar convirtiéndose en una aplicación monolítica, lo que puede ser un riesgo para la escalabilidad

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.)

¿Preguntas?

Bibliografía

- Bass, L., Clements, P., Kazman, R. (2012). ***Software Architecture in Practice, Third Edition.*** Addison-Wesley Professional.
[Leer en Safari Books Online](#)
- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. (1996).
Pattern-Oriented Software Architecture, Volume 1, A System of Patterns. John Wiley & Sons.
[Leer en Safari Books Online](#)
- Richards, M. (2015). ***Software Architecture Patterns.*** O'Reilly Media, Inc.
[Leer en Safari Books Online](#)
- Fayad, M., Schmidt, D.C. (1997). ***Object-Oriented Application Frameworks.*** In Communications of the ACM, Special Issue on Object-Oriented Application Frameworks, Vol. 40, No. 10, October 1997.
[Enlace \(febrero 2017\)](#)

Lógica de negocio y acceso a datos

Diseño de Sistemas Software
Curso 2020/2021

Carlos Pérez Sancho



Universitat d'Alacant
Universidad de Alicante

Departament de Llenguatges i Sistemes Informàtics
Departamento de Lenguajes y Sistemas Informáticos

Supuesto inicial

- Queremos implementar la funcionalidad “Procesar pedido”
 - Un pedido se compone de varias líneas de pedido
 - Cada línea de pedido está asociada a un producto e indica cuántas unidades de producto se van a vender
 - Si no hay suficientes unidades del producto en stock, el pedido no se puede procesar
 - El formulario para crear el pedido permitirá buscar productos por categoría

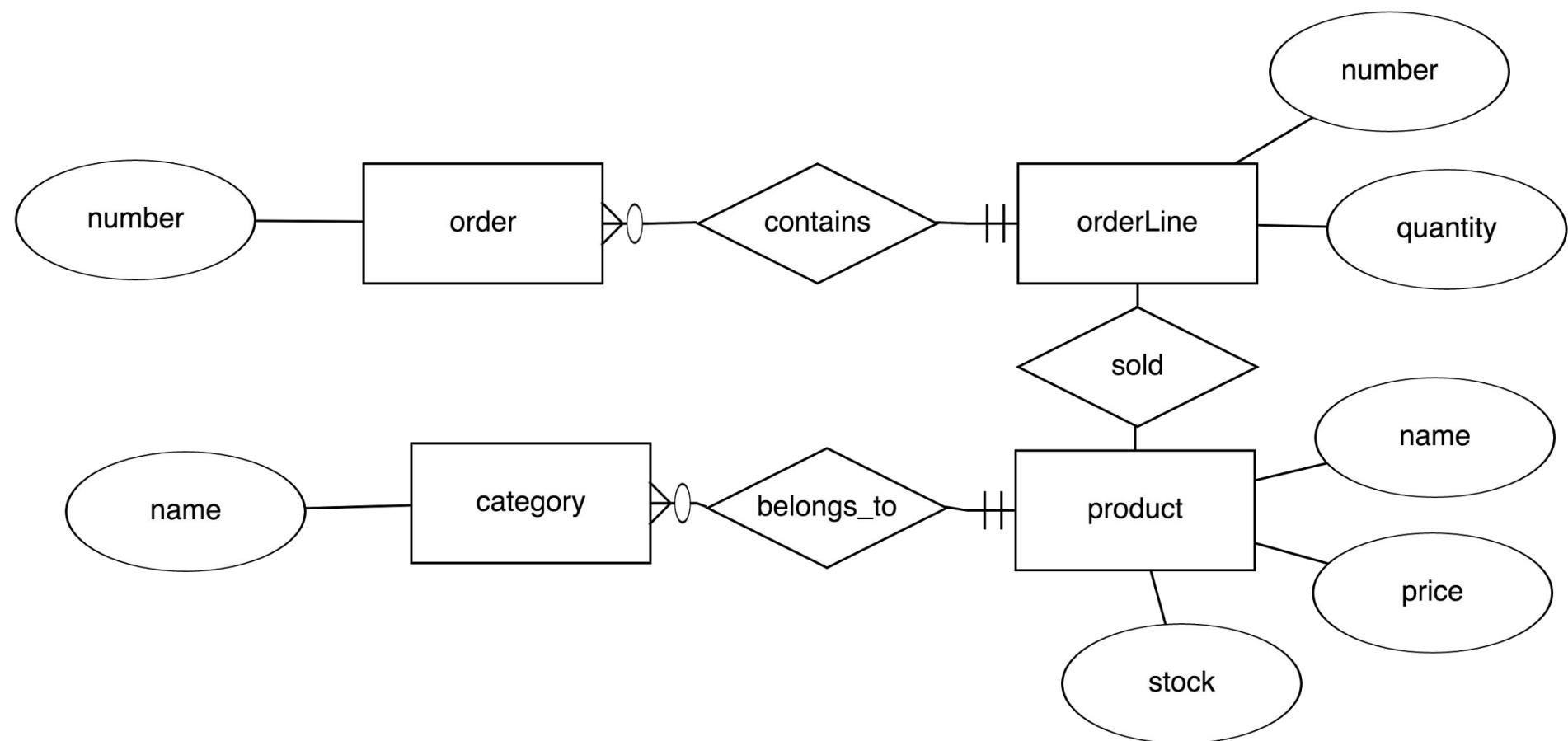
Mockup “Crear pedido”

The mockup displays a web-based application for creating a new order. At the top, there is a header bar with standard browser controls (back, forward, stop, refresh) and a search bar. Below the header, the title "New order" is centered. On the left side, a "Product" section contains a search input field, a list of products ("Product 1", "Product 2", "Product 3"), a quantity selector (set to 3), and an "Add" button. To the right, an "Order" section shows a table with four columns: "#", "Product", "Quantity", "Price", and "Subtotal". The table contains two items: Product 71 (Quantity 3, Price 15.2, Subtotal 45.6) and Product 2 (Quantity 1, Price 9.95, Subtotal 9.95). At the bottom, the total value "Total: 55.55" is displayed, along with "Cancel" and "Process" buttons.

#	Product	Quantity	Price	Subtotal
1	Product 71	3	15.2	45.6
2	Product 2	1	9.95	9.95

Created with Balsamiq - www.balsamiq.com

Diseño de base de datos



¿Cómo diseñamos la lógica de negocio?

Patrones de lógica de negocio

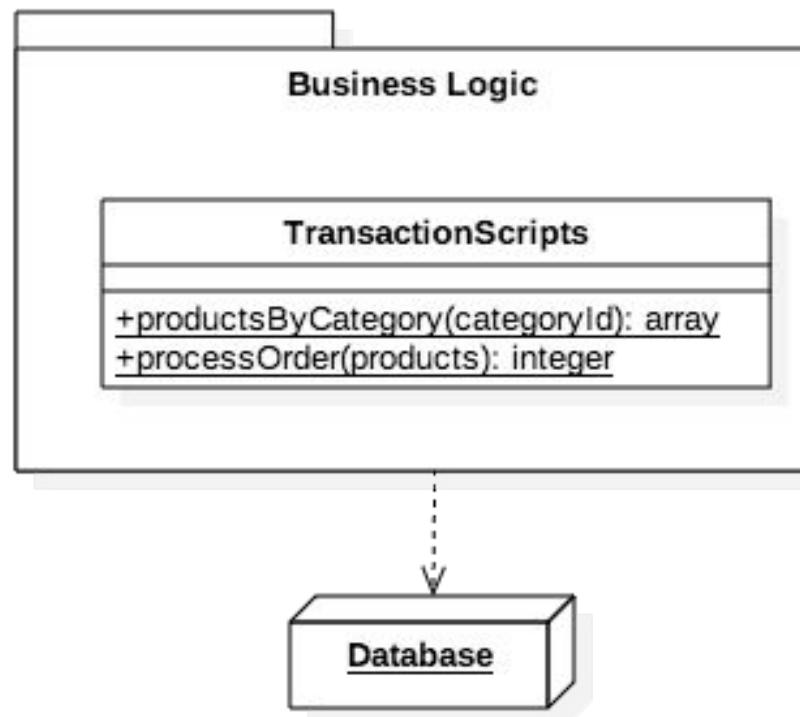
- Tres opciones (Fowler, 2002)
 - Transaction script
 - Table module
 - Domain model

Transaction script

Patrones de lógica de negocio y acceso a datos

Transaction Script

“Organiza la lógica de negocio en procedimientos, donde cada procedimiento gestiona una petición de la capa de presentación.”



Transaction Script

- Es la forma más sencilla de organizar la lógica de negocio
- Se crea una transacción para cada una de las funcionalidades de la aplicación
- Cada transaction script se organiza en un único método, haciendo llamadas directamente a la base de datos

Transaction Script

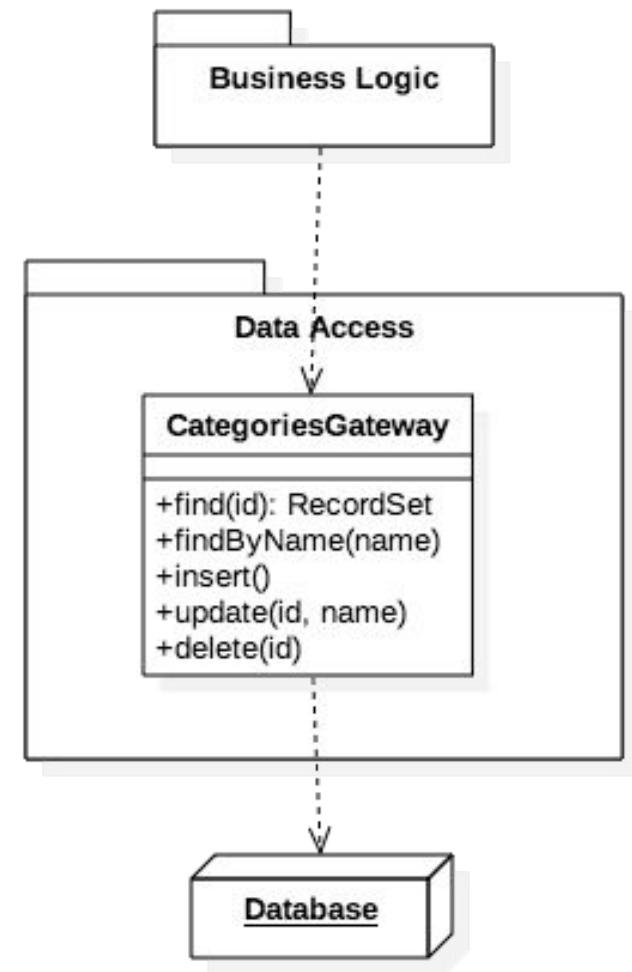
```
class RawScripts {
    public static function productsByCategory($category) {
        $products = DB::table('products')
            ->join('categories', 'products.category_id', '=', 'categories.id')
            ->where('categories.name', $category)
            ->select('products.id', 'products.name',
                'products.price', 'products.stock')
            ->get();
        return $products;
    }

    public static function processOrder($products) {
        // Ver proyecto en GitHub
    }
}
```

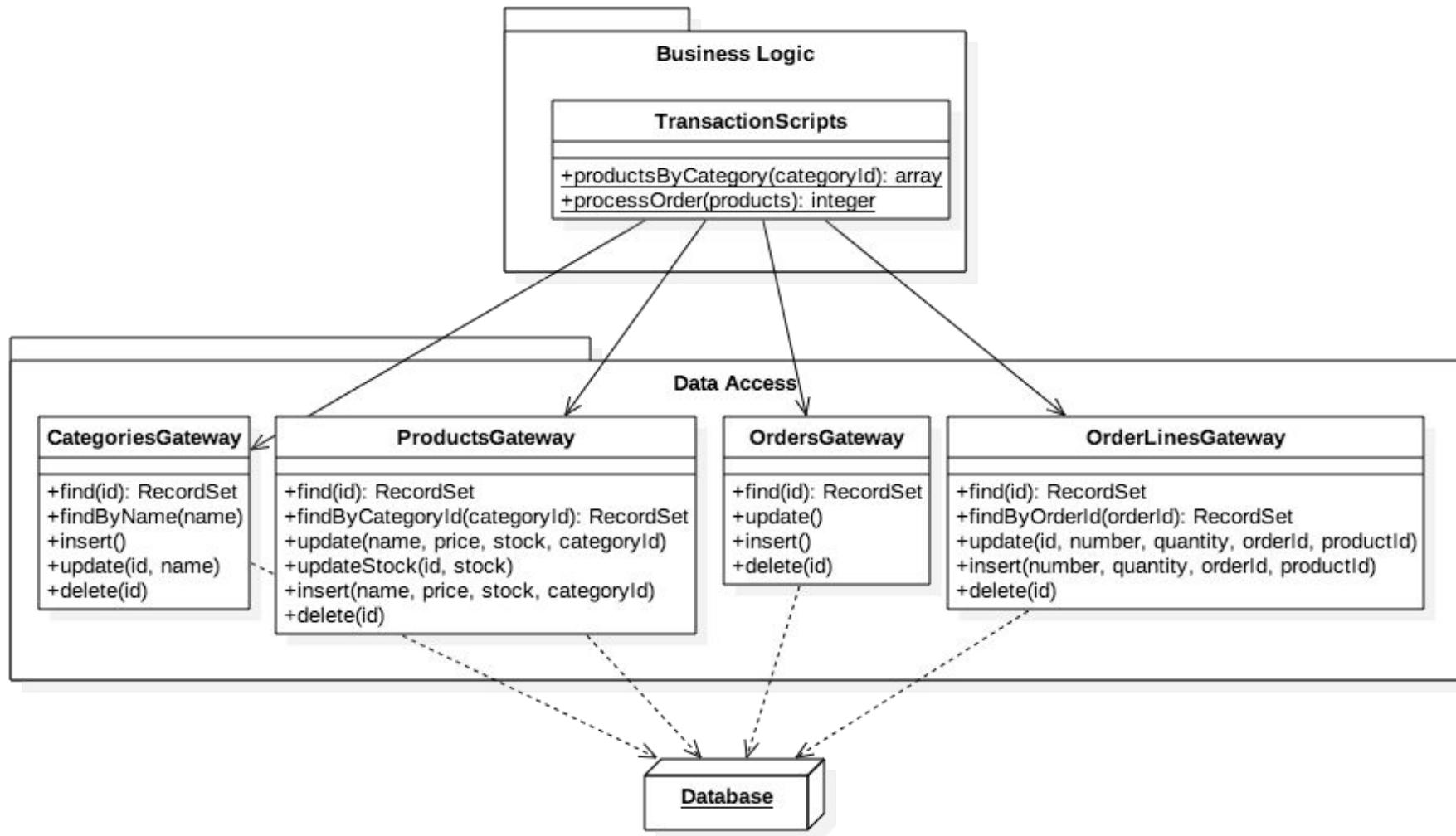
Table Data Gateway

Si queremos evitar hacer llamadas SQL directamente desde los transaction script, podemos combinarlos con el patrón de acceso a datos Table Data Gateway:

“Un objeto que actúa como pasarela para una tabla de la base de datos. Una única instancia gestiona todas las filas de la tabla.”



Transaction Script + Table Data Gateway



Transaction Script

```
use App\.DataAccessLayer\TableDataGateways\CategoriesGateway as CG;
use App\.DataAccessLayer\TableDataGateways\OrdersGateway as OG;
use App\.DataAccessLayer\TableDataGateways\OrderLinesGateway as OLG;
use App\.DataAccessLayer\TableDataGateways\ProductsGateway as PG;

class Scripts {
    public static function productsByCategory($category) {
        $category = CG::findByName($category);
        if ($category)
            return PG::findCategoryId($category->id);
        else
            return null;
    }

    public static function processOrder($products) {
        // View project at GitHub
    }
}
```

Table module

Patrones de lógica de negocio y acceso a datos

Table Module

“Una única instancia gestiona la lógica de negocio para todas las filas en una tabla o vista de la base de datos.”

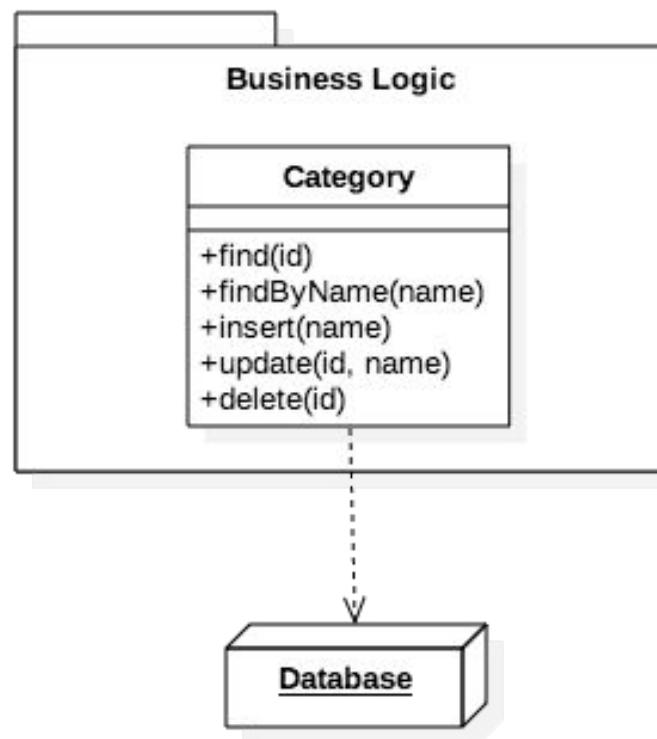


Table Module

- Una única instancia del objeto Table Module permite gestionar todos los registros de la tabla
- Todos los métodos excepto insert() necesitan un identificador para localizar el registro en la base de datos
- Los objetos table module también pueden usar el patrón Table Data Gateway para comunicarse con la base de datos
 - Table Module implementa la lógica de negocio
 - Table Data Gateway implementa el acceso a datos

Table Module + Table Data Gateway

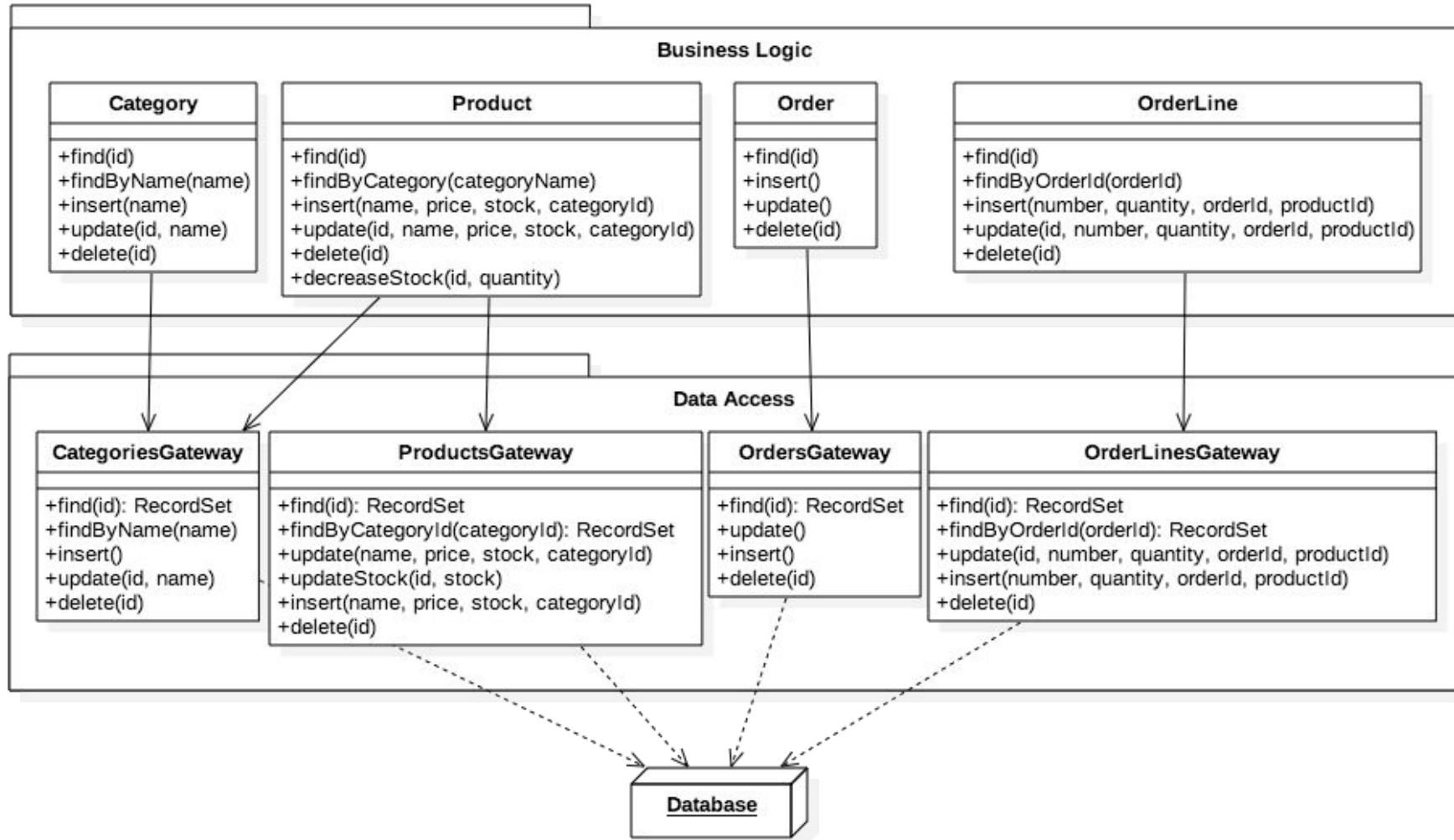


Table Module

```
use App\.DataAccessLayer\TableDataGateways\ProductsGateway as PG;

class Product {

    public static function findByCategory($category) {
        $category = Category::findByName($category);
        if ($category) return PG::findByCategoryId($category->id);
        else return null;
    }

    public static function decreaseStock($id, $quantity) {
        $product = PG::find($id);
        if ($product && $product->stock >= $quantity) {
            PG::updateStock($id, $product->stock - $quantity);
            return true;
        }
        else return false;
    }

    // View project at GitHub
}
```

Table Module

- Los objetos Table Module sólo contienen la lógica de negocio que corresponde a cada objeto individual
- Las operaciones complejas deben ir en la capa de servicio

Table Module + Capa de servicio

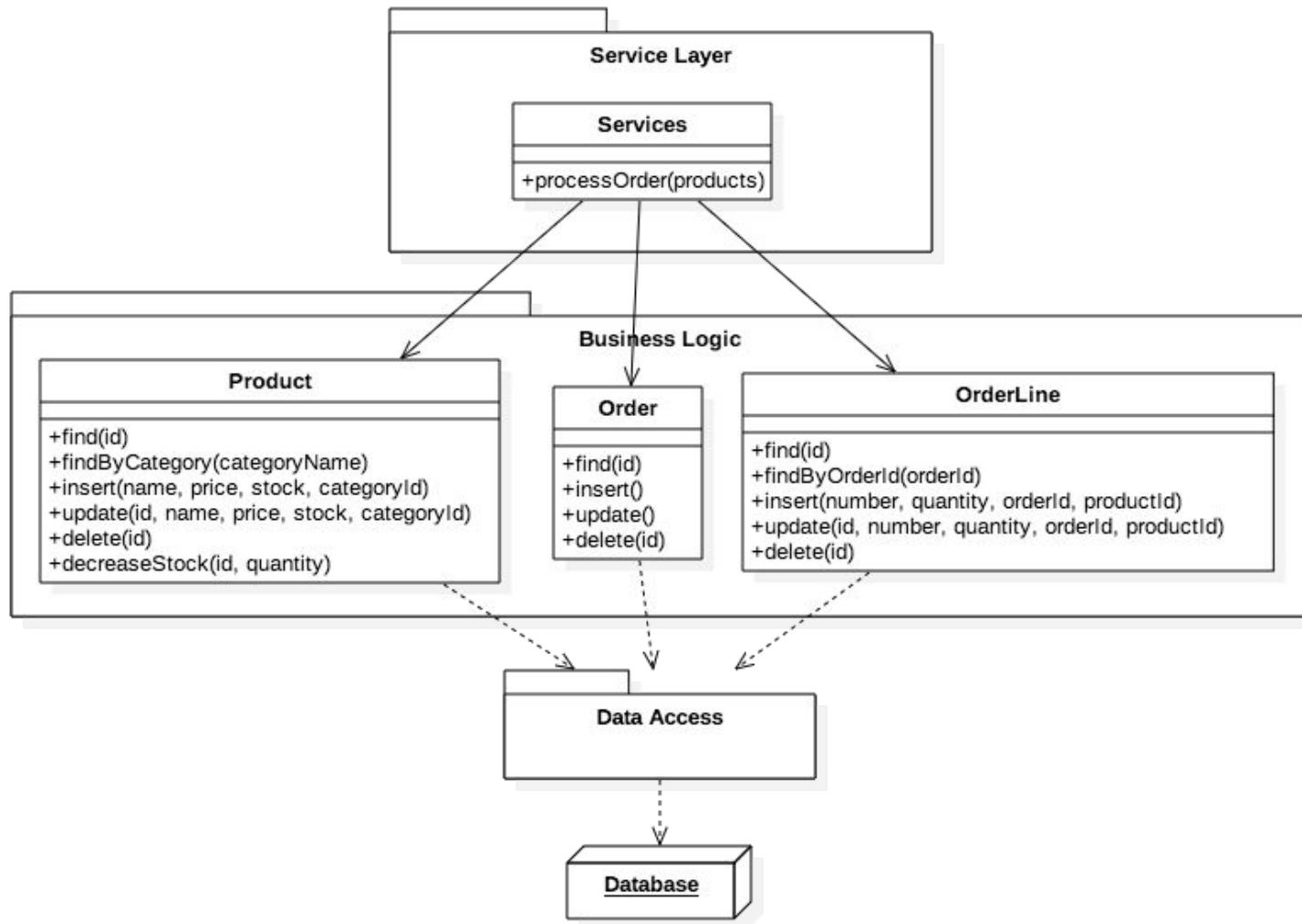


Table Module

```
use App\BusinessLogicLayer\TableModules\Order;
use App\BusinessLogicLayer\TableModules\OrderLine;
use App\BusinessLogicLayer\TableModules\Product;

class TableModuleServices {
    public static function processOrder($products) {
        $rollback = false;
        $lineNumber = 1;

        DB::beginTransaction();

        $orderId = Order::insert();
        foreach ($products as $productId => $quantity) {
            if (Product::decreaseStock($productId, $quantity)) {
                OrderLine::insert($lineNumber, $quantity, $orderId,
                    $productId);
                $lineNumber++;
            }
        }
    }
}

// View project at GitHub
```

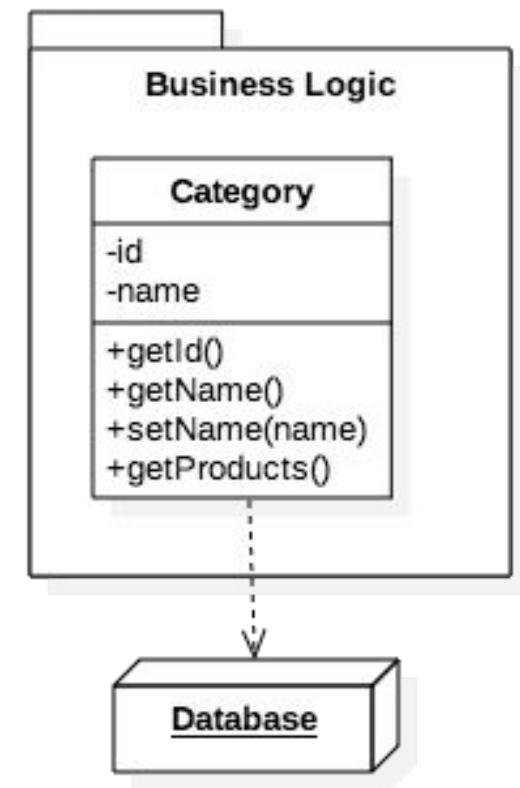
Domain model

Patrones de lógica de negocio y acceso a datos

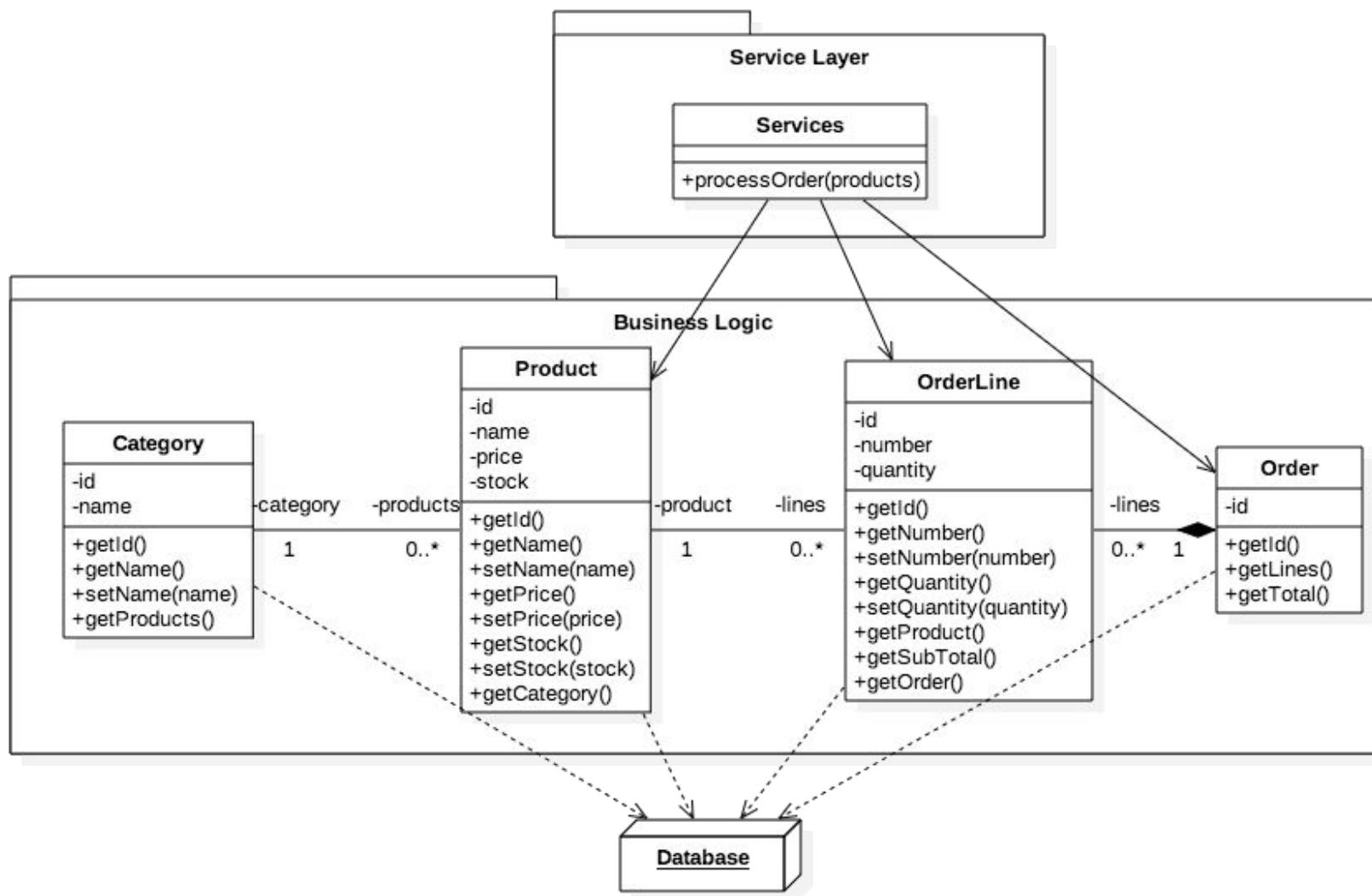
Domain Model

Cuando la lógica de negocio es compleja la mejor opción es implementar un modelo de dominio:

“Un objeto del modelo de dominio incorpora tanto el comportamiento como los datos.”



Domain Model

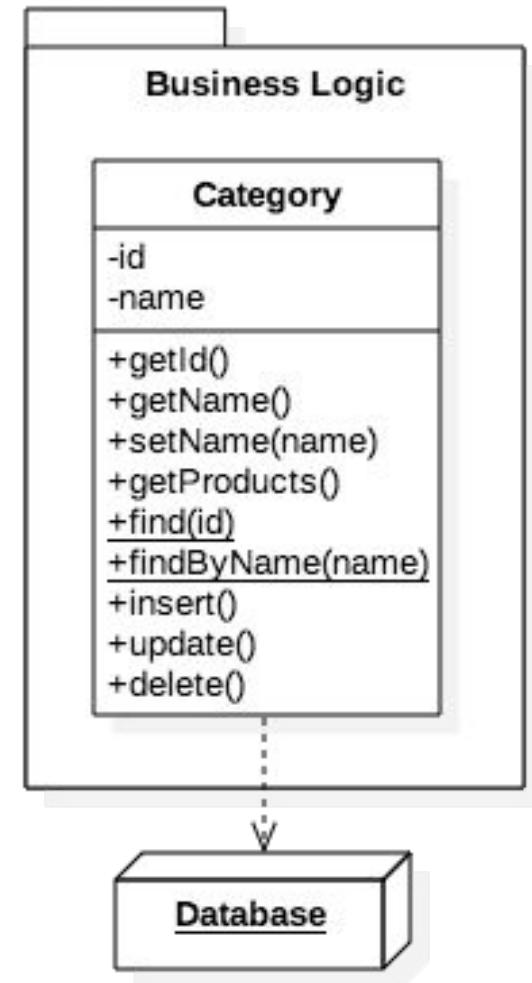


Domain Model

- Transformar el modelo de dominio a una estructura de tablas relacional puede ser complejo
 - Los modelos sencillos tienen un objeto por cada tabla
 - Modelos más complejos pueden tener una estructura distinta a la de la base de datos
- Dependiendo de la complejidad el acceso a datos se puede hacer de dos maneras
 - Modelos sencillos → **ActiveRecord**
 - Modelos complejos → **Data Mapper**

ActiveRecord

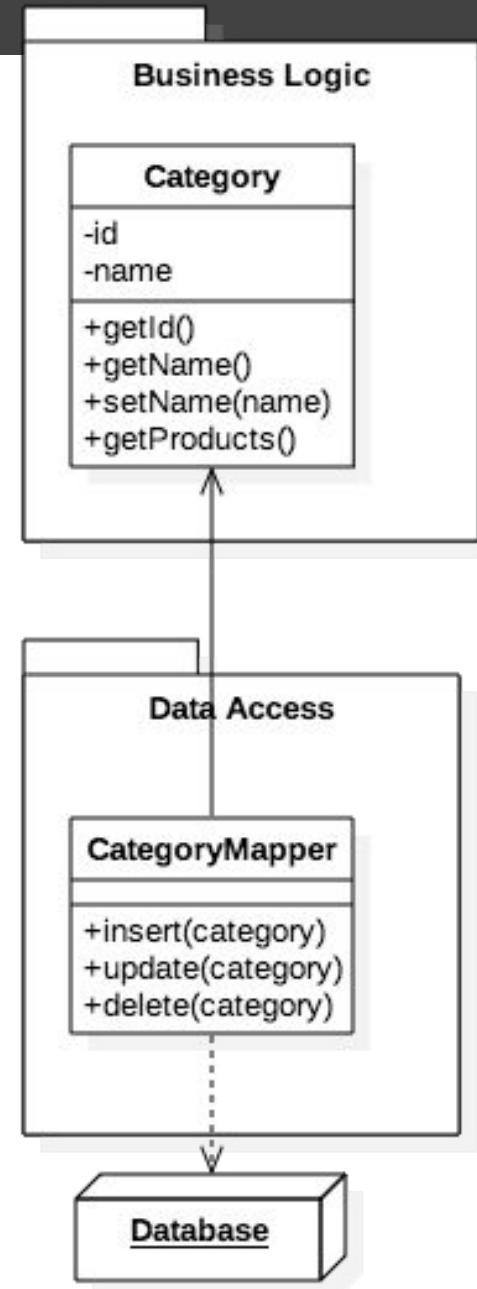
“Un objeto que envuelve una fila de una tabla o vista, encapsula el acceso a la base de datos y añade comportamiento a esos datos.”



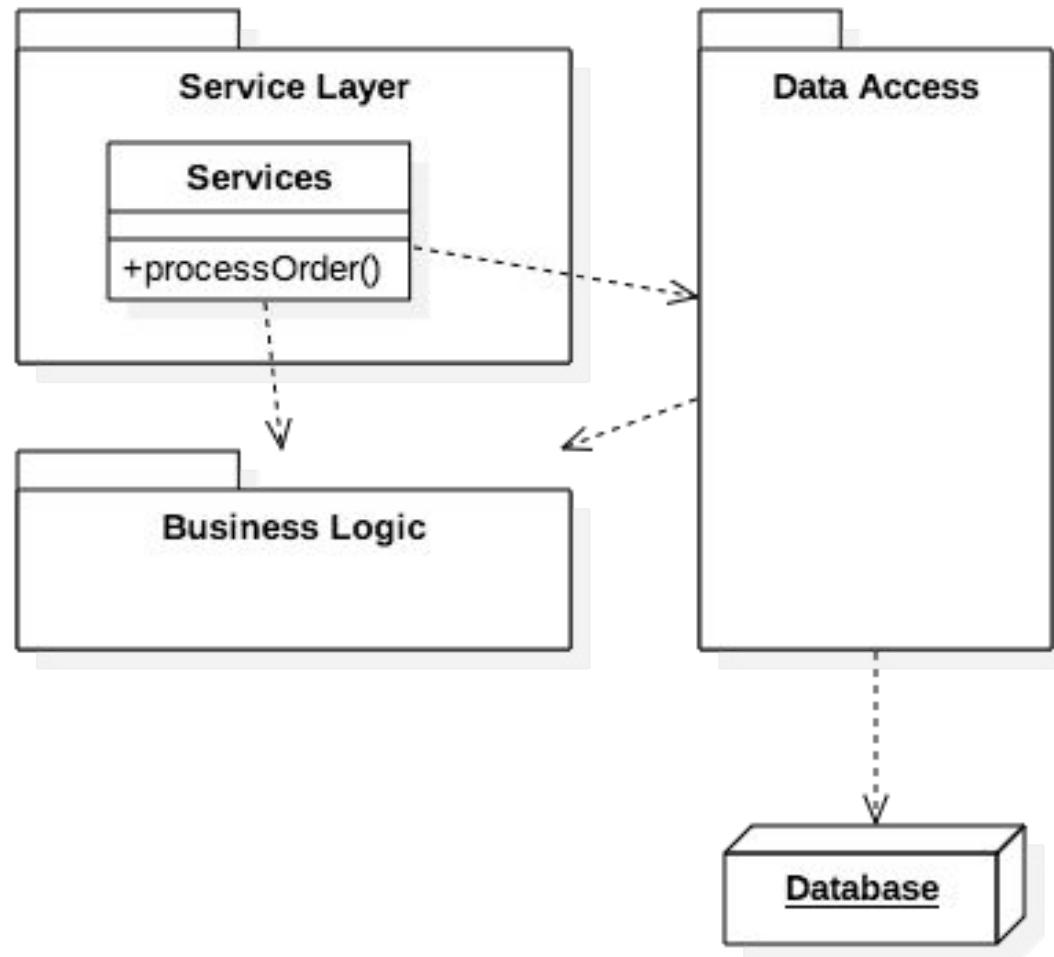
Data Mapper

Si modelo de dominio es complejo conviene mantenerlo separado del acceso a datos. En estos casos se usa el patrón Data Mapper:

“Una capa de mapeadores se encarga de mover datos entre los objetos y la base de datos, manteniéndolos independientes entre ellos e independientes del mapeador.”



Data Mapper



Resumen

Patrones de lógica de negocio y acceso a datos

Resumen

Complejidad	Lógica de negocio	Acceso a datos
↓	Transaction Script	-
		Table Data Gateway
	Table Module	-
↓	Domain Model	Table Data Gateway
		ActiveRecord
		Data Mapper

Ejercicios

Patrones de lógica de negocio y acceso a datos

Ejercicios

- Dibuja los diagramas de secuencia para el método `processOrder()` de los objetos
 - `App\BusinessLogicLayer\TransactionScripts\Scripts`
 - `App\ServicesLayer\TableModuleServices`
 - `App\ServicesLayer\ActiveRecordServices`

¿Preguntas?

Bibliografía

- Fowler, M. (2002). Patterns of Enterprise Application Architecture. Addison-Wesley Professional.

[Leer en Safari Books Online](#)

Modelo de dominio y mapeo objeto-relacional (ORM)

Diseño de Sistemas Software
Curso 2020/2021

Carlos Pérez Sancho



**Universitat d'Alacant
Universidad de Alicante**

Departament de Llenguatges i Sistemes Informàtics
Departamento de Lenguajes y Sistemas Informáticos

Modelos de dominio

Implementación de modelos de dominio

- Las funcionalidades de un sistema requieren la ejecución de distintos tipos de acciones
 - Validación de los datos de entrada
 - Ejecución de las reglas de negocio
 - Comprobación de restricciones en los datos
(p.ej. integridad referencial, normalmente se realiza en la base de datos)

Implementación de modelos de dominio

- Ejemplo: actualizar stock de producto
 - Validación: si la cantidad es negativa (reducción de stock) debe haber suficiente stock en el almacén
 - Reglas de negocio: si la cantidad restante es menor que un umbral, crear una alerta para reponer

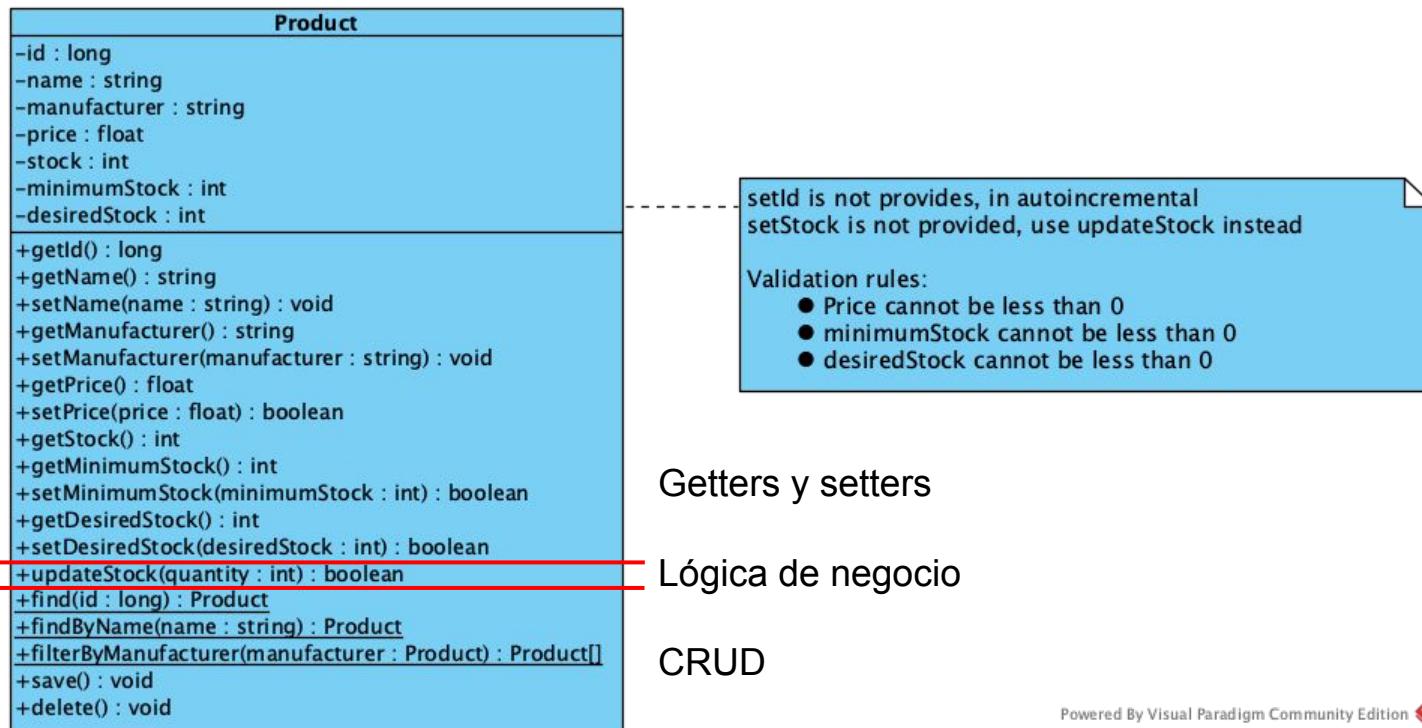
Tipos de modelos de dominio

- Al implementar la lógica de negocio con un modelo de dominio se puede optar entre 2 tipos distintos, dependiendo de cómo se distribuyan estas funcionalidades
 - Modelos de dominio ricos
 - Modelos de dominio anémicos

Modelos de dominio ricos

Modelos de dominio ricos

- En un modelo rico todos los métodos están en los modelos (CRUD + lógica de negocio)



Modelos de dominio ricos

- Implementación de las reglas de validación
 - Los métodos set (*setters*) deben comprobar la validez de los valores de entrada
 - Dos estilos de implementación
 - Devolver valores booleanos
 - Lanzar excepciones

Modelos de dominio ricos

- Implementación con Laravel

- Los atributos son públicos y NO HAY QUE DECLARARLOS EN EL MODELO, Eloquent los coge automáticamente de la estructura de la tabla
- No son necesarios getters ni setters, aunque se pueden implementar
 - **Accessors:** permiten crear valores calculados
 - **Mutators:** permiten realizar validaciones o transformaciones sobre los valores de entrada
- Se pueden crear métodos CRUD para realizar consultas usando **Scopes**

Modelos de dominio ricos

- Migración

```
public function up()
{
    Schema::create('products', function (Blueprint $table) {
        $table->bigIncrements('id');
        $table->timestamps();
        $table->string('name');
        $table->string('manufacturer');
        $table->float('price');
        $table->unsignedInteger('stock');
        $table->unsignedInteger('minimumStock');
        $table->unsignedInteger('desiredStock');
    });
}
```

Modelos de dominio ricos

- Modelo (1/2)

¡Los atributos no se declaran aquí!

```
class Product extends Model
{
    public function getDescriptionAttribute() {
        return "{$this->name} ({$this->manufacturer})";           Accessor
    }

    public function setPriceAttribute($value) {
        if ($value >= 0)
            $this->attributes['price'] = $value;
        else
            throw new \Exception('Invalid price');                  Mutator
    }

    public function scopeManufacturer($query, $manufacturer) {
        return $query->where('manufacturer', $manufacturer);       Scope
    }
}
```

Modelos de dominio ricos

- Modelo (2/2)

```
public function updateStock($quantity) {  
    $newStock = $this->stock - $quantity;  
  
    if ($newStock < 0)  
        throw new \Exception('There is not enough stock');  
  
    $this->stock = $newStock;  
    if ($newStock < $this->minimumStock)  
        StockService::createAlert($this);  
}  
}
```

Lógica de negocio

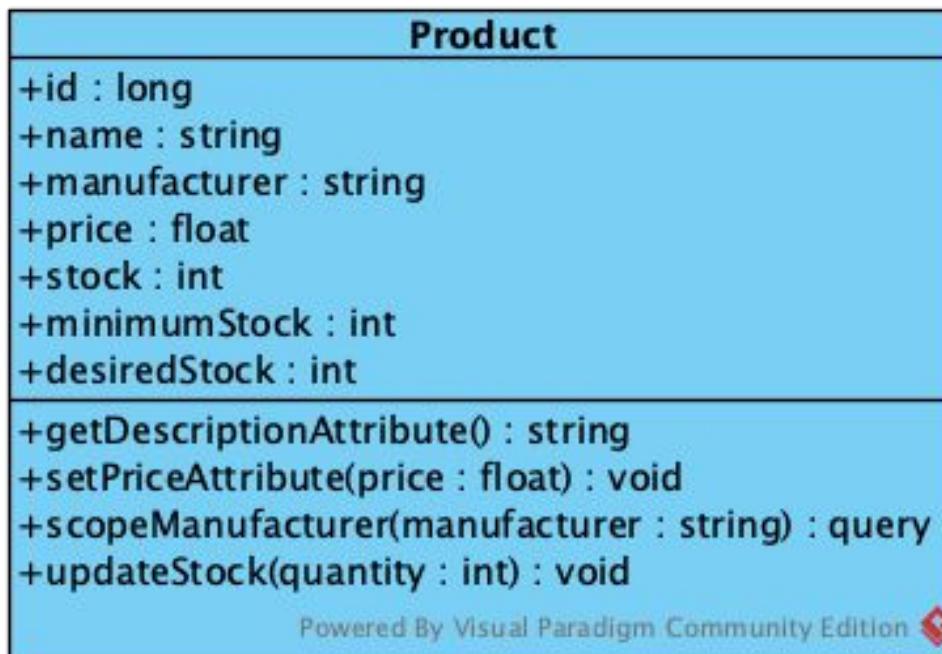
Modelos de dominio ricos

- Uso

```
>>> $p->description
=> "Monitor (Asus)"
>>> $p->price = -1
Exception with message 'Invalid price'
>>> $p->manufacturer('Asus')->get()
=> Illuminate\Database\Eloquent\Collection {#3016
    all: [
        App\Product {#3007
            id: 1,
            created_at: "2020-03-08 17:46:06",
            updated_at: "2020-03-08 17:46:06",
            name: "Monitor",
            manufacturer: "Asus",
            price: 150.0,
            stock: 20,
            minimumStock: 5,
            desiredStock: 20,
        },
    ],
}
>>> █
```

Modelos de dominio ricos

- Diagrama de implementación



Modelos de dominio anémicos

Modelos de dominio anémicos

- Un modelo de dominio **anémico** contiene objetos ligeros que solamente almacena datos y sus relaciones
- Estos objetos se pueden usar como objetos de transferencia de datos (Data Transfer Objects, DTO) para pasar información entre capas
- La validación de datos y la lógica de negocio se implementan fuera de estos objetos

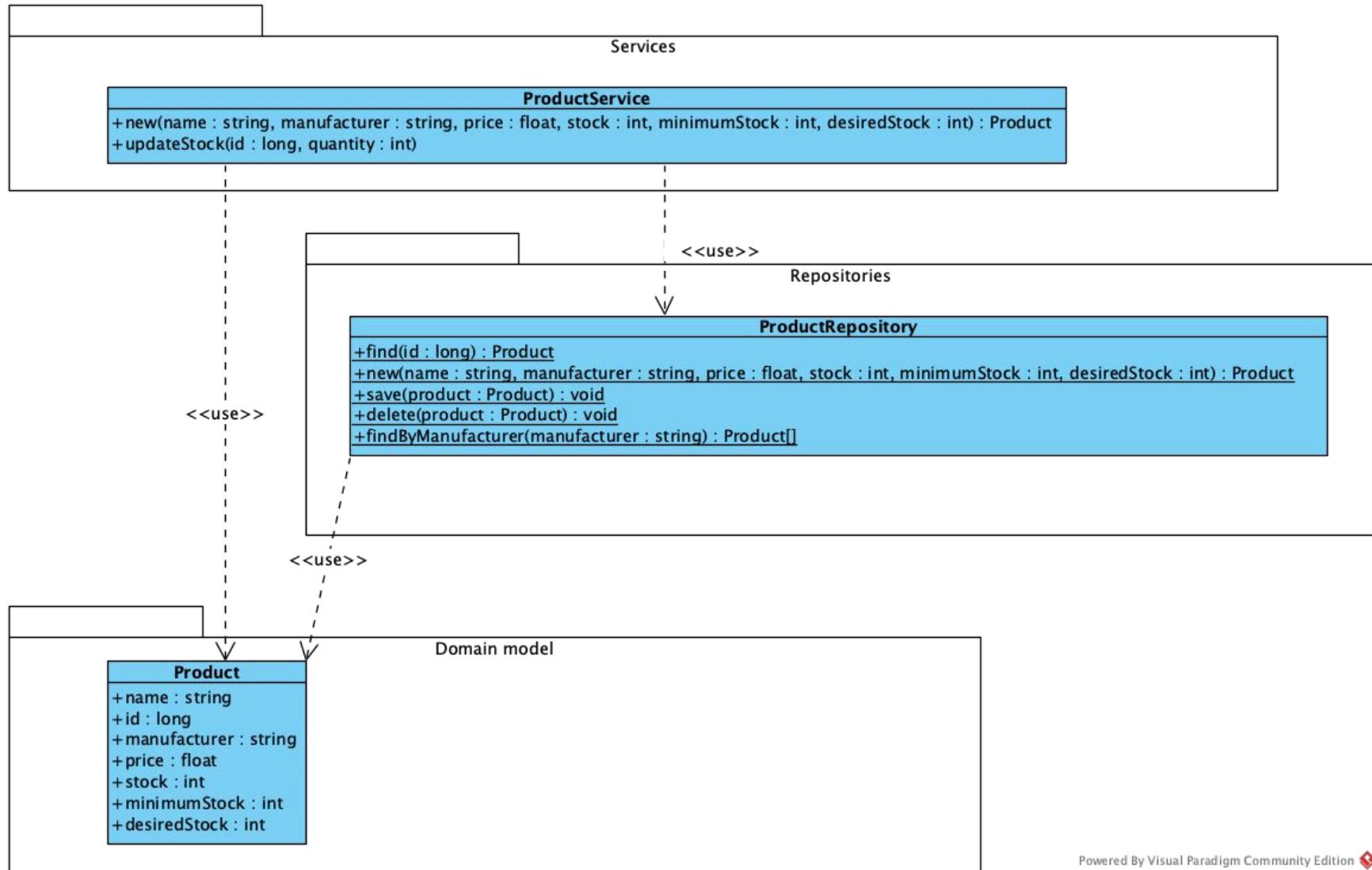
Modelos de dominio anémicos

- ¿Dónde ubicamos entonces la validación de datos y la lógica de negocio?
 - La validación de los datos de entrada se suele hacer en los controladores de la capa de presentación
 - La lógica de negocio puede situarse en 2 capas distintas
 - Capa de presentación: ubicarla en los controladores hace perder la encapsulación y favorece la aparición de código duplicado (no recomendado)
 - Capa de servicios: objetos que implementan la lógica de negocio y usan el modelo de dominio para representar los datos

Modelos de dominio anémicos

- Es conveniente encapsular los métodos del CRUD para no depender de una implementación concreta
 - p.ej. podríamos cambiar de Eloquent (ActiveRecord) a Doctrine (Data Mapper)
- Para desacoplar se suele usar el **Patrón Repositorio**

Patrón repositorio



Modelos de dominio anémicos

- Modelo (si tuviera relaciones con otros objetos, habría que añadirlas aquí)

```
class Product extends Model
{
    //
}
```

Modelos de dominio anémicos

- Repositorio

Modelos de dominio anémicos

• Servicio

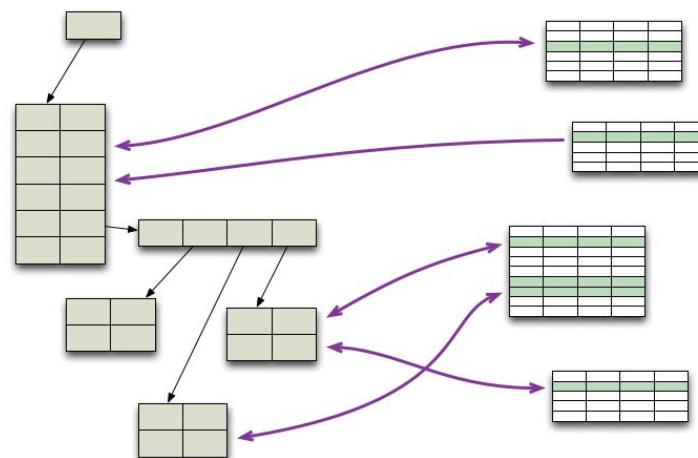
Modelos de dominio

- El patrón Repositorio también se puede usar con modelos de dominio ricos para desacoplar los controladores del uso de Eloquent

Mapeo objeto-relacional

Mapeo objeto-relacional

- Para implementar un modelo de dominio hay que encontrar la manera de adaptar la estructura del modelo a la estructura de la base de datos



<https://martinfowler.com/bliki/OrmHate.html>

Mapeo objeto-relacional

- Las **bases de datos orientadas a objetos** son una forma natural de persistir objetos, pero no son muy comunes
- Las **bases de datos relacionales** son mucho más comunes, pero tienen una estructura distinta a los modelos orientados a objetos (*impedance mismatch*)
- Las **bases de datos NoSQL** son más populares que las orientadas a objetos y permiten expresar la estructura de objetos complejos (p.ej. MongoDB almacena documentos JSON con una estructura de árbol), pero no tienen la potencia de las relacionales para trabajar con relaciones

Mapeo objeto-relacional

- Los sistemas de **mapeo objeto-relacional** (Object-Relational Mapping, ORM) se encargan de almacenar y recuperar los objetos en **bases de datos relacionales**
- Estos sistemas permiten crear una **capa de acceso a datos** sin necesidad de implementarla
- Ejemplos de sistemas ORM
 - Java: Hibernate, MyBatis, ...
 - C#: Entity Framework, Nhibernate, ...
 - PHP: Doctrine, Eloquent, ...
 - Python: SQLAlchemy, Django ORM, ...
 - ...

Mapeo objeto-relacional

- Los sistemas ORM existentes normalmente implementan uno de los dos patrones para persistir modelos de dominio
 - **ActiveRecord**
 - (+) es más sencillo de implementar
 - (-) incrementa el acoplamiento con la base de datos y dificulta la refactorización
 - (-) es difícil optimizar el SQL y puede perjudicar el rendimiento
 - **DataMapper**
 - (+) permite realizar transformaciones complejas
 - (+) facilita la optimización del acceso a la base de datos
 - (-) es más difícil de entender y configurar

Mapeo objeto-relacional

- Dos estrategias
 - **Model first:** primero se diseña el modelo de dominio, y luego se crea la base de datos con la estructura necesaria. Situación ideal para usar el patrón ActiveRecord.
 - **Database first:** se parte de una base de datos ya existente, por lo que hay que adaptar el modelo de dominio a su estructura. En estos casos puede ser más conveniente usar un Data Mapper.

Patrones de comportamiento

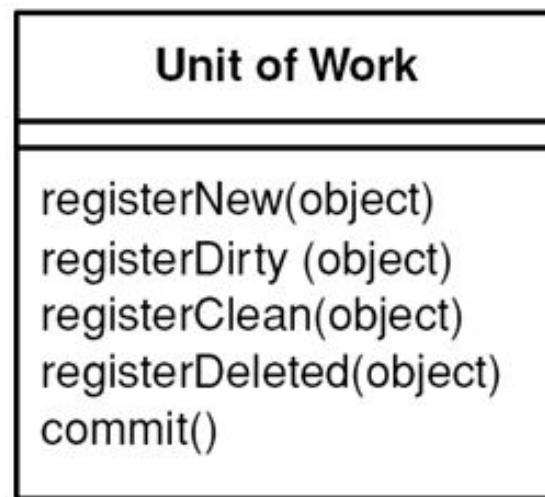
Mapeo objeto-relacional

Patrones de comportamiento

- Los patrones de comportamiento gestionan cómo se almacenan y recuperan los objetos de una base de datos relacional
- Problemas a resolver
 - Se debe mantener la integridad referencial cuando se crean y modifican múltiples objetos (**Unit of Work**)
 - No debe haber múltiples copias del mismo objeto en memoria (**Identity Map**)
 - Cargar los objetos relacionados en un modelo de dominio puede acabar recuperando la base de datos completa (**Lazy Load**)

Unit of Work

“Mantiene una lista de objetos afectados por una transacción y coordina la escritura de los cambios y la resolución de problemas de concurrencia.”



Unit of Work

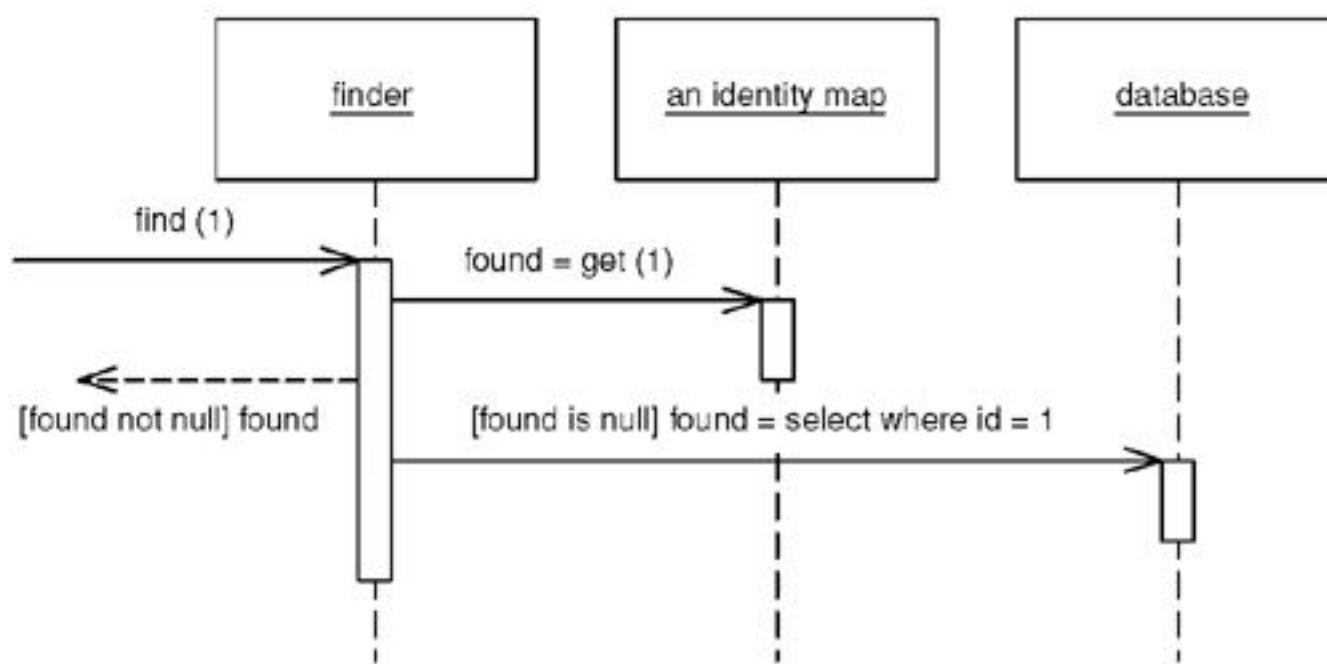
- Llamar a la base de datos para cada cambio en el modelo de dominio puede afectar al rendimiento
- En lugar de llamar directamente a la base de datos, el sistema notifica al objeto *Unit of Work*, de manera que pueda llevar un registro de los objetos nuevos, recuperados de la base de datos, modificados y borrados
- Cuando es necesario, abre una transacción con la base de datos y realiza todos los cambios en orden

Unit of work

- Laravel no proporciona este patrón

Identity Map

“Asegura que cada objeto se carga una única vez llevando un registro en un mapa de cada objeto. Busca los objetos en el mapa antes de recuperarlos de la base de datos.”



Identity Map

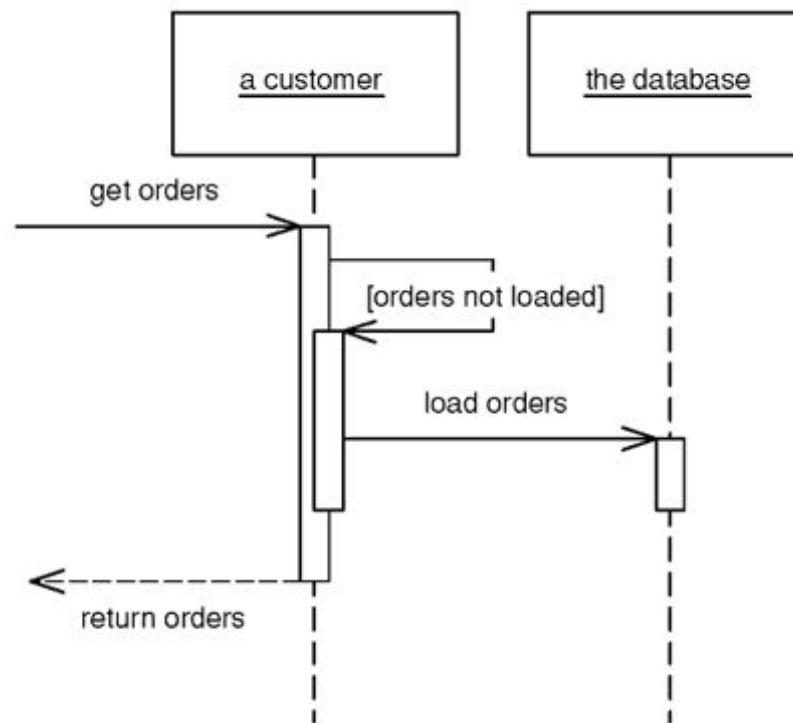
- Cargar un registro varias veces de la base de datos puede dar lugar a múltiples objetos con los mismos datos
- Problemas
 - Gasto excesivo de recursos
 - Inconsistencia cuando uno de ellos se modifica
- El objeto *Identity Map* lleva un registro de los objetos cargados y devuelve referencias para los que ya existe una instancia en memoria
- También actúa de caché para la base de datos

Identity Map

- Laravel no proporciona este patrón

Lazy Load

“Un objeto que no contiene datos, pero sabe cómo obtenerlos.”



Lazy Load

- Cuando se carga un objeto del modelo de dominio puede ser útil cargar sus objetos relacionados
- Sin embargo, esto puede degradar el rendimiento si el número de objetos relacionados es elevado, o incluso acabar cargando la base de datos completa en una reacción en cadena
- *Lazy Load* sólo carga un objeto cuando se usa

Lazy Load

- Alternativas de implementación
 - **Lazy initialization:** usa un valor nulo para los objetos hasta que se cargan, necesita que las propiedades estén encapsuladas en métodos get
 - **Virtual proxy:** los objetos se sustituyen por un objeto vacío que carga los datos cuando es necesario, permite separar la lógica necesaria para cargar los datos de los objetos del modelo de dominio

Lazy Load

- Laravel usa lazy-loading por defecto
- Se pueden cargar modelos relacionados para mejorar el rendimiento (eager loading)

```
$books = App\Book::with('author')->get();  
  
foreach ($books as $book) {  
    echo $book->author->name;  
}
```

Patrones estructurales

Mapeo objeto-relacional

Mapeo de objetos pequeños

- A veces no es necesario mapear todos los objetos de dominio a tablas en la BBDD
 - Los objetos pequeños se pueden guardar junto a su contendor
 - Las colecciones o jerarquías de pequeños objetos se pueden guardar juntas, de manera que se pueda acceder a ellas en una sola operación

Embedded Value

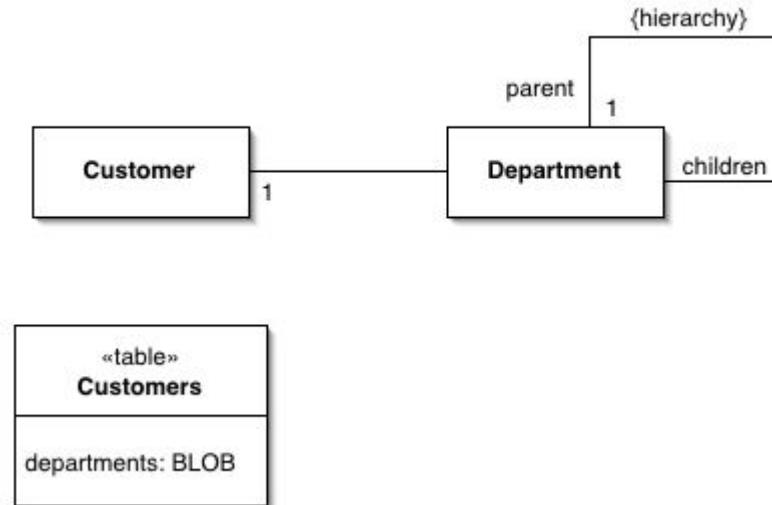
“Mapea un objeto en varios campos de otra tabla.”

Employment
ID
person: person
period: DateRange
salary: Money

«table» Employments
ID: int
personID: int
start: date
end: date
salaryAmount: decimal
salaryCurrency: char

Serialized LOB

- En ocasiones hay objetos que tienen una propiedad que contiene una estructura jerárquica de pequeños objetos
- Una forma eficiente de almacenarlos es transformarlos en un único objeto grande (LOB), ya sea en forma binaria (BLOB) o textual (CLOB), y almacenarlo en una única columna



Serialized LOB

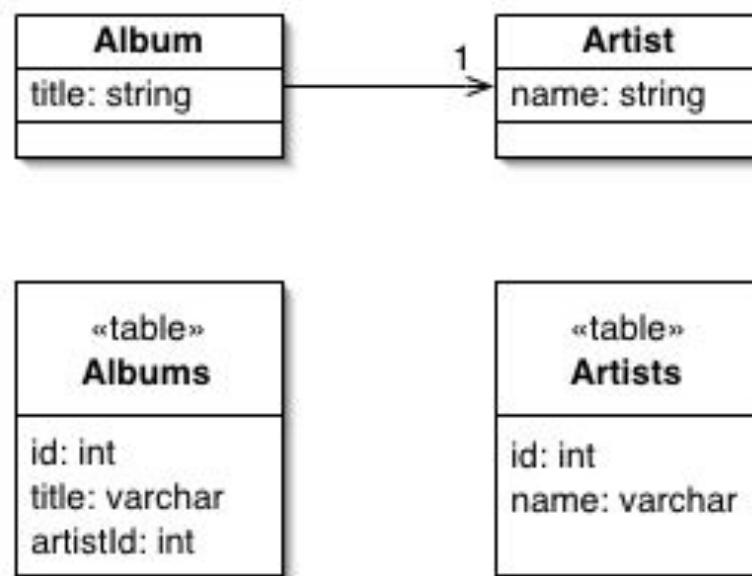
- ¡Atención! Los campos BLOB no deberían utilizarse para almacenar imágenes
- Es preferible almacenar las imágenes en una carpeta y guardar su ruta en la base de datos

Mapeo de relaciones

- Cuando un objeto contiene colecciones o referencias (compartidas) a otros objetos, no deben guardarse como valores en la misma tabla
- Necesitamos representar esas referencias para mantener la base de datos en forma normal

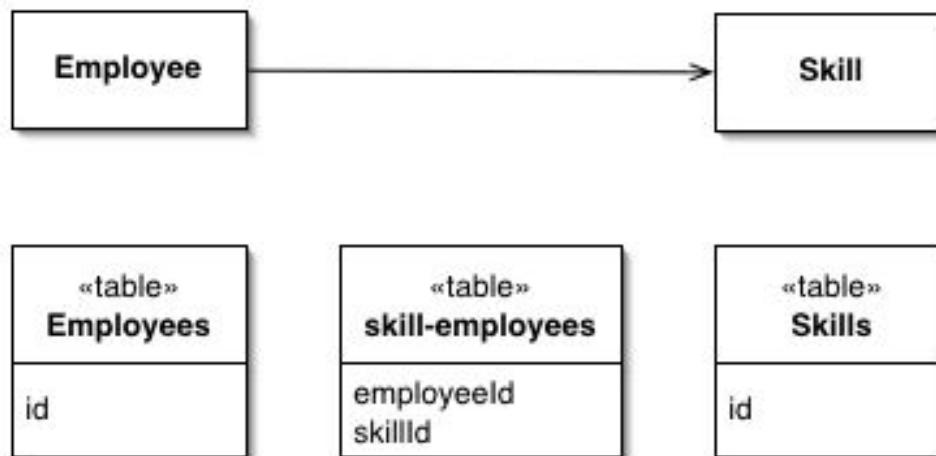
Mapeo de clave ajena

“Mapea una asociación entre objetos a una referencia de clave ajena entre tablas.”



Mapeo de tabla de asociación

"Guarda una asociación como una tabla con claves ajenas a las tablas que están vinculadas por la asociación".



Mapeo de la herencia

- Las bases de datos relacionales no soportan la herencia
- Si decidimos implementar una jerarquía de herencia en el modelo de dominio necesitamos:
 - Poder usar las clases hijas como si se tratase de la clase padre (polimorfismo)
 - Que todas las clases de la jerarquía compartan un campo identificador común

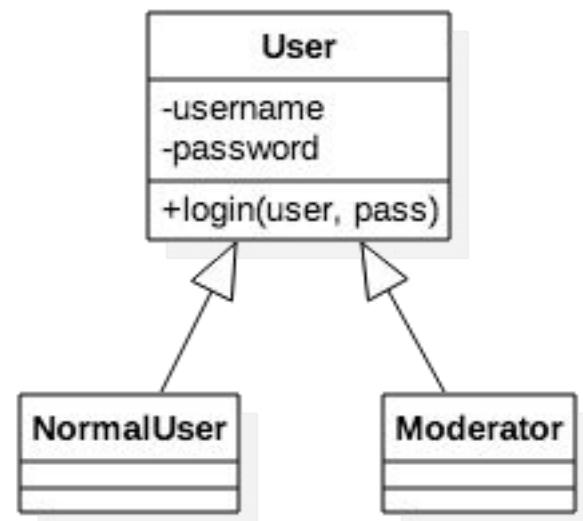
Mapeo de la herencia

Supongamos que cada objeto se almacena en una tabla diferente

```
$user = new NormalUser();
$user->save(); // Obtiene el id 1
```

```
$user = new Moderator();
$user->save(); // Obtiene el id 1
```

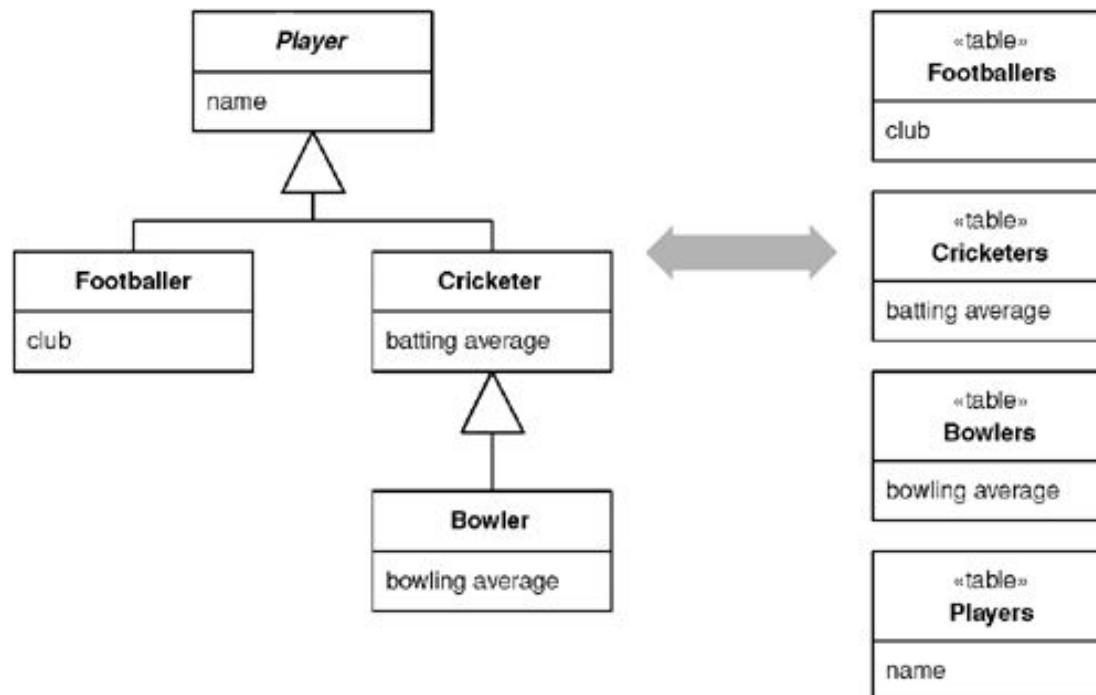
```
$user = User::find(1);
```



¿Cuál devolverá?

Patrón Class table inheritance

- Se usa una única tabla para cada clase en la jerarquía
- Cada tabla almacena únicamente los atributos nuevos
 - (-) Necesita hacer join para cada consulta, es un problema para el rendimiento
 - (+) No hay columnas irrelevantes



Patrón Concrete table inheritance

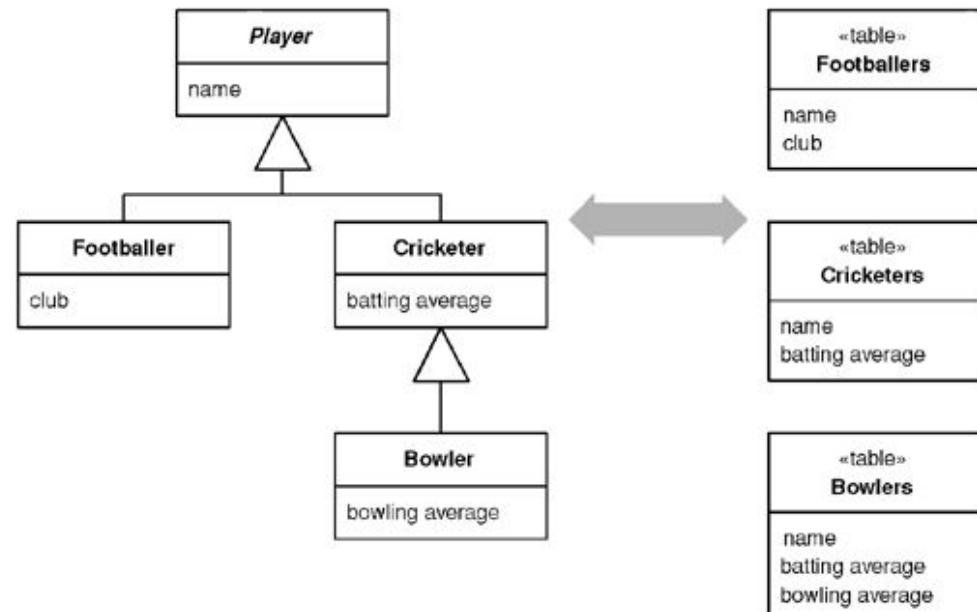
- Se usa una única tabla para cada clase “hoja”
- Cada tabla almacena todos los atributos (heredados + nuevos)

(-) Es complicado gestionar los identificadores si queremos que todas las subclases comparten campo identificador

(-) No se pueden representar relaciones con las clases abstractas

(+) No hay columnas irrelevantes

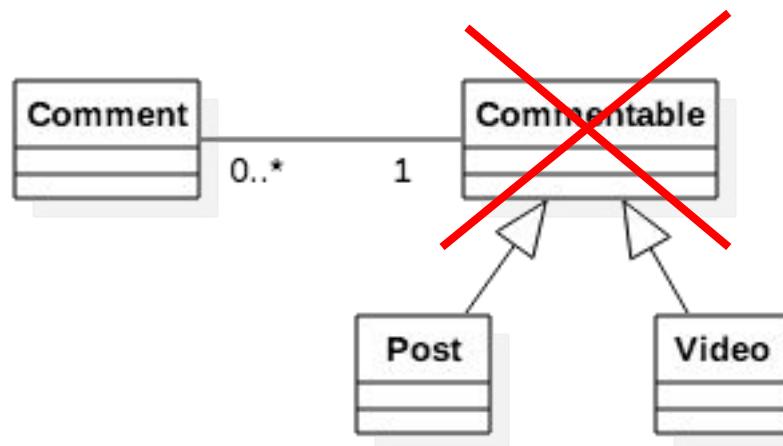
(+) No necesita hacer join



Patrón Concrete table inheritance

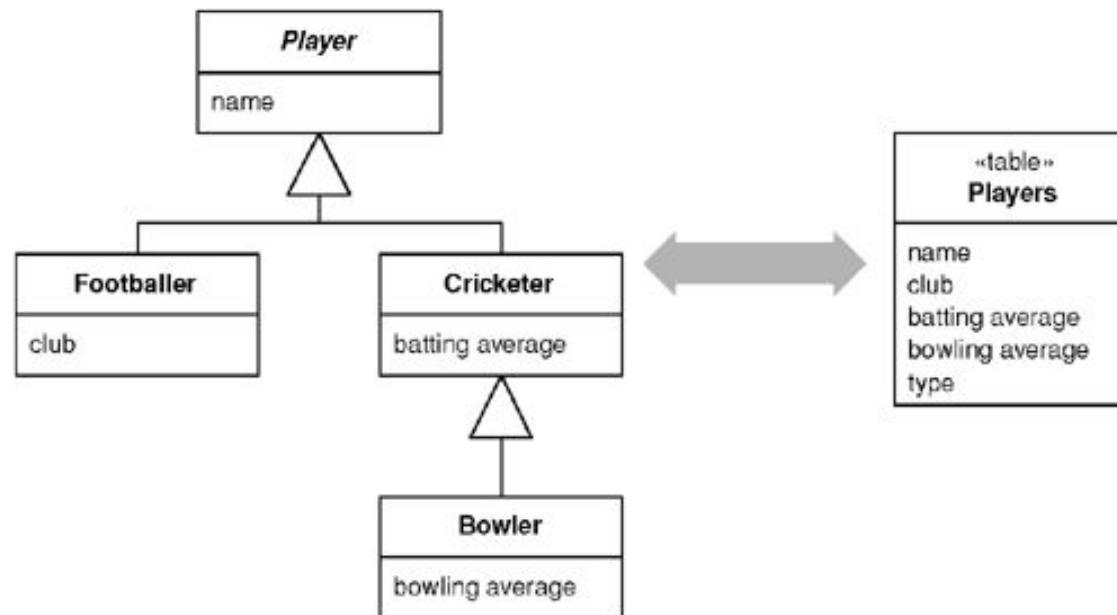
- Implementación en Laravel mediante relaciones polimórficas
 - Permite acceder a objetos de distinto tipo a través de una relación (`$comment->commentable` devolverá un objeto que podrá ser un Post o un Video)
 - La clase base no se implementa

<https://laravel.com/docs/6.x/eloquent-relationships#polymorphic-relationships>



Patrón Single table inheritance

- Se usa una única tabla para todas las clases
- Almacena la unión de todos los atributos de las clases hijas
 - (-) Las columnas que no usan todas las clases gastan espacio
 - (+) Evita joins innecesarios
 - (+) Mismo campo identificador para todas las subclases



Patrón Single table inheritance

- Laravel tiene una extensión que permite implementar este patrón manteniendo la jerarquía de clases en el código
 - No permite instanciar objetos de la clase padre → no podemos usar el polimorfismo \$user = User::find(1)

<https://github.com/Nanigans/single-table-inheritance>

- En la práctica es más sencillo juntar todas las clases hijas en una sola con la unión de todos los atributos y métodos, y un atributo adicional indicando el tipo de cada objeto

Pero entonces...

¿qué sucede si queremos tener
distintos tipos de usuarios?

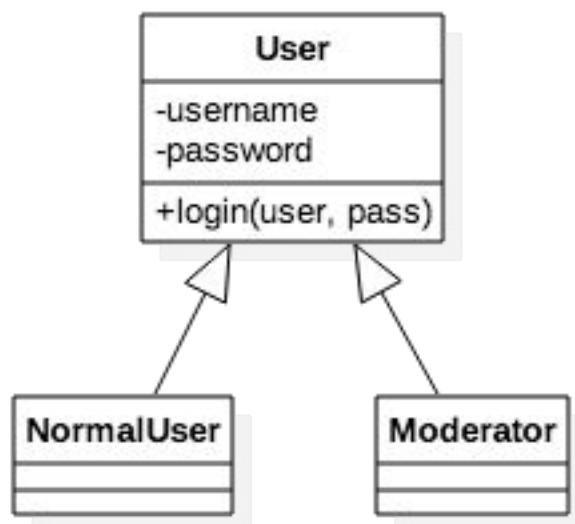
Autenticación de usuarios

Mapeo objeto-relacional

Autentificación de usuarios

- Requisitos:

- En un foro podemos tener usuarios normales y moderadores
- Los usuarios normales se pueden convertir en moderadores a propuesta de otros moderadores



¿PROBLEMAS?

- No podemos cambiar el tipo de un usuario, hay que destruirlo y crear uno nuevo
- Tendríamos que hacer comprobación de tipos (`instanceof`) para comprobar el tipo de usuario cada vez que se ejecuta una funcionalidad

Autentificación de usuarios

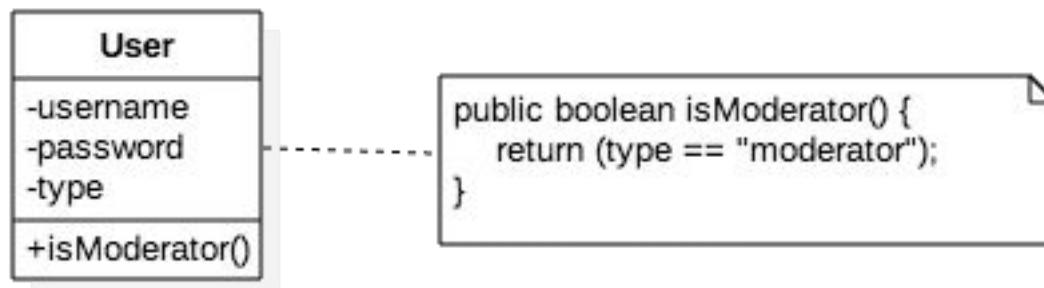
- Cuando el usuario pide acceso a una funcionalidad se hacen dos comprobaciones:
 - **Autentificación:** comprueba la identidad del usuario (recuperando el objeto User de la sesión activa)
 - **Autorización:** comprueba si el usuario identificado tiene permisos para acceder a la funcionalidad solicitada
- Al dibujar una pantalla se muestran únicamente los elementos a los que el usuario tiene acceso

Autentificación de usuarios

- Si la autentificación falla se redirige al usuario al formulario de login
- Cuando el usuario introduce las credenciales correctas se crea una instancia de la clase User y se almacena en la sesión (objeto global que almacena información compartida entre todas las pantallas)

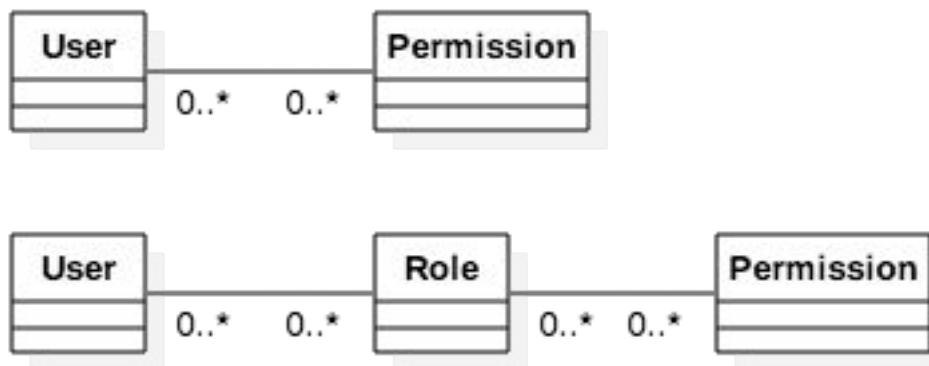
Autentificación de usuarios

- La clase User no implementa las funcionalidades, se usa para otorgar acceso a las pantallas que las ofertan
- Se puede simplificar el diseño usando una única clase



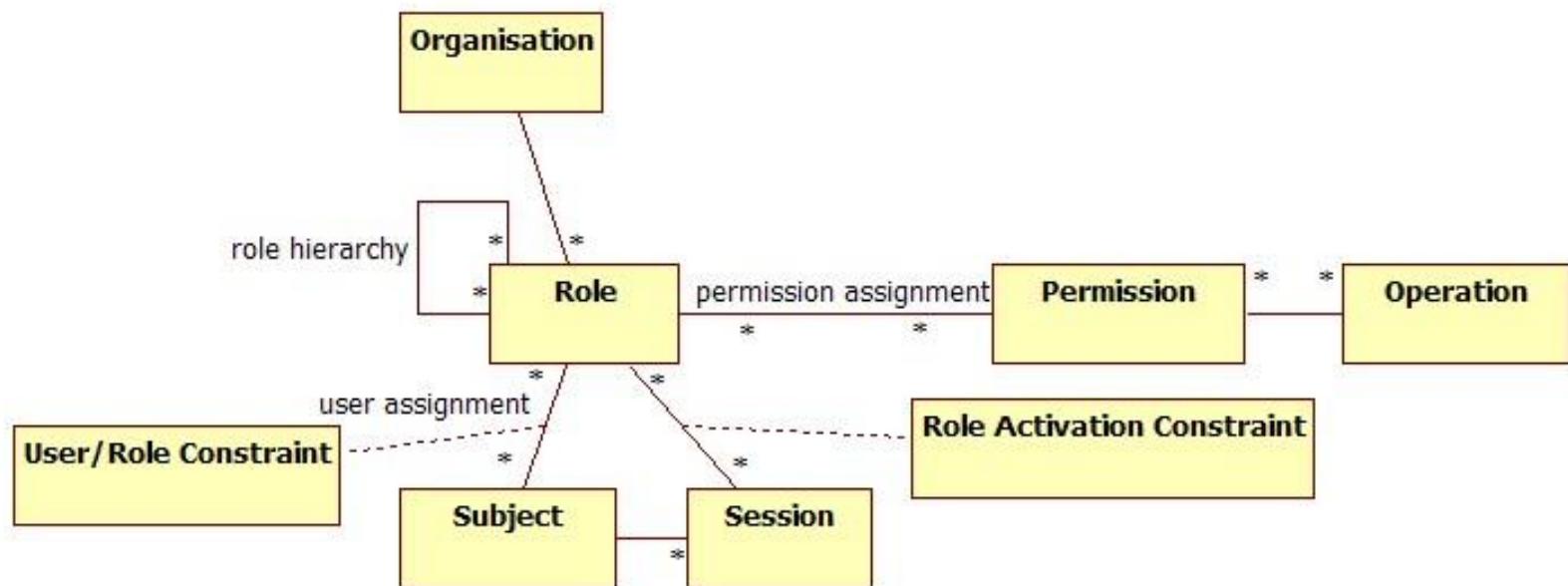
Autenticación de usuarios

- Si queremos tener mayor control sobre lo que pueden hacer los usuarios podemos añadir permisos



Autentificación de usuarios

- Generalización: patrón Role-Based Access Control (RBAC) ([estándar NIST](#))



Fuente: [Wikipedia](#)

¿Preguntas?

Bibliografía

- Fowler, M. (2002). Patterns of Enterprise Application Architecture. Addison-Wesley Professional.

[Leer en Safari Books Online](#)

Patrones de capa de presentación

Diseño de Sistemas Software
Curso 2020/2021

Carlos Pérez Sancho



Universitat d'Alacant
Universidad de Alicante

Departament de Llenguatges i Sistemes Informàtics
Departamento de Lenguajes y Sistemas Informáticos

Contenidos

1. Introducción
2. Patrón Model-View-Controller
3. Patrón Model-View-Presenter
4. Patrón Model-View-ViewModel
5. Aplicaciones web RIA

Introducción

Introducción

En una arquitectura en capas, la **Capa de Presentación** tiene las siguientes responsabilidades:

- Gestionar la interacción con el usuario
- Comunicarse con las capas inferiores que proveen las funcionalidades deseadas
- Mostrar/actualizar la información como resultado de las llamadas a la lógica de negocio

Para estructurar correctamente el código de la capa de presentación conviene distribuir estas responsabilidades entre distintos objetos.

Introducción

Los patrones estructurales más usados para la capa de presentación son: [Fowler, 2003, Osmani, 2015]

- Model-View-Controller (MVC)
- Model-View-Presenter (MVP)
- Model-View-ViewModel (MVVM)

Para todos ellos se considera que el **Modelo** representa los objetos del modelo de dominio en la **capa de lógica de negocio**.

Los demás objetos (**Vista** y **Controlador/Presenter/ViewModel**) pertenecen a la **capa de presentación**.

Introducción

Estos patrones son independientes del tipo de aplicación (escritorio, web, móvil, ...).

Dependiendo del framework y de las características de la aplicación será más sencillo emplear unos u otros.

- Normalmente los frameworks para aplicaciones web de servidor están preparados para usar el patrón MVC
- La mayoría de frameworks para aplicaciones web cliente (*frontend*) permite usar cualquiera de los tres

Patrón Model-View-Controller

Model-View-Controller (MVC)

Aunque aparentemente se trata de un patrón sencillo, es uno de los patrones con más variantes y sobre el que menos consenso hay.

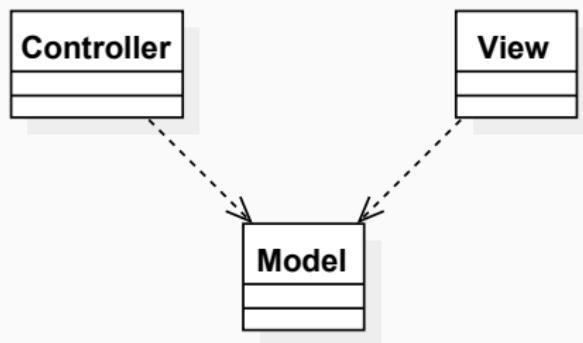
Hoy en día es el **patrón más utilizado en aplicaciones web**, aunque de forma muy distinta a como fue diseñado en sus orígenes.

MVC original

El patrón MVC fue concebido inicialmente para pequeños componentes gráficos en Smalltalk-80.

Cada componente gráfico (cuadro de texto, checkbox, etc.) tiene una vista y un controlador:

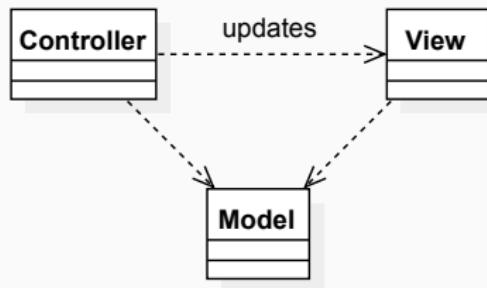
- La vista se encarga de dibujar el componente (había que escribir el código para esto) a partir de los datos del modelo
- El controlador recibe la interacción del usuario y manipula el modelo



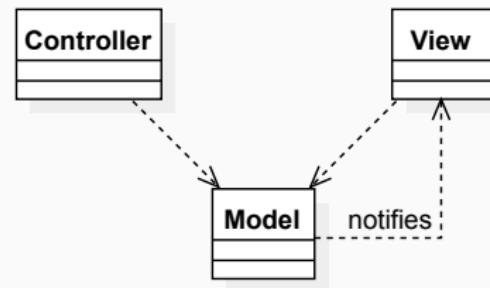
MVC original

Hay dos posibilidades para implementar la forma en que se actualiza la vista:

- El controlador actualiza la vista después de manipular el modelo
- El modelo notifica a la vista para que se actualice (**modelo activo**)



Modelo pasivo

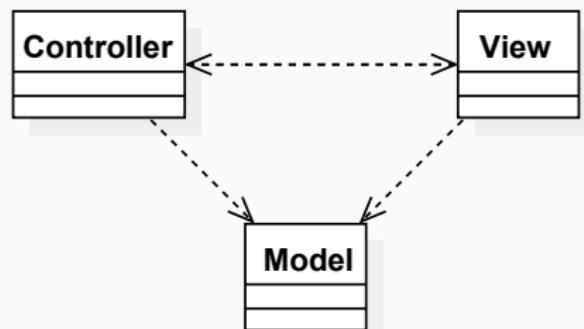


Modelo activo

MVC en aplicaciones web

En aplicaciones web simples (sin clientes enriquecidos con JavaScript) la comunicación entre objetos es distinta:

- La vista gestiona la interacción con el usuario y notifica al controlador
- El controlador manipula el modelo y decide qué vista mostrar a continuación

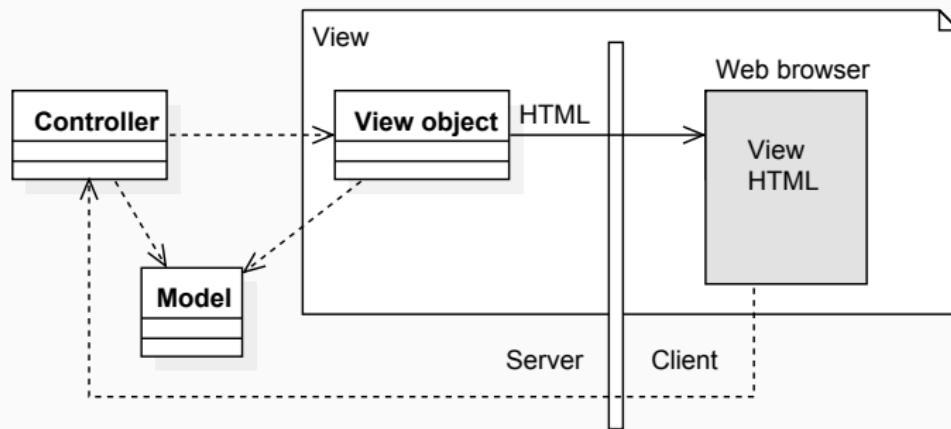


MVC en aplicaciones web

Esta forma de comunicarse es necesaria porque la aplicación está dividida físicamente entre el cliente y el servidor.

La vista tiene dos partes:

- Un objeto (dinámico) que se instancia en la capa de presentación del servidor para generar el HTML
- El HTML (estático) mostrado en el navegador del cliente



Tipos de controladores

Hay varias formas de programar los controladores:

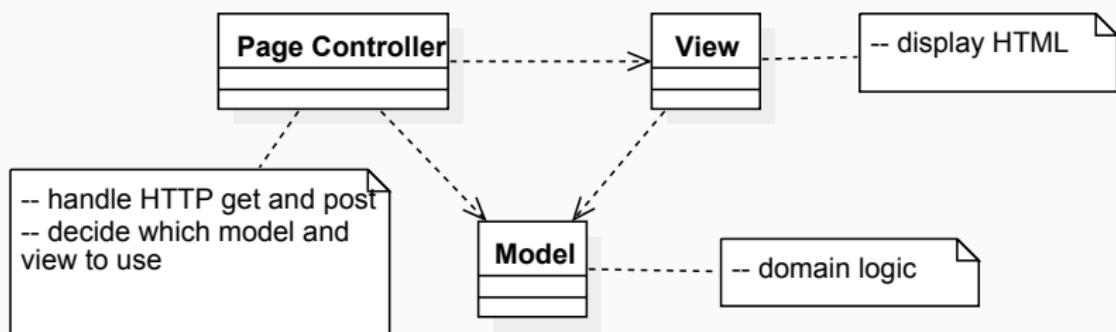
- Page controller
- Front controller

Un **Front Controller** puede colaborar además con un **Application Controller**.

Page Controller

Patrón Page Controller [Fowler, 2003]

Un objeto que gestiona una petición para una página o acción específica en un sitio web.



Page controller

Es el patrón utilizado por ASP.NET (*code-behind*).

Implica la creación de un controlador para cada página lógica de la aplicación.

Responsabilidades:

- Decodificar la URL y extraer los datos de la petición
- Crear e invocar los objetos del modelo
- Seleccionar la vista a mostrar

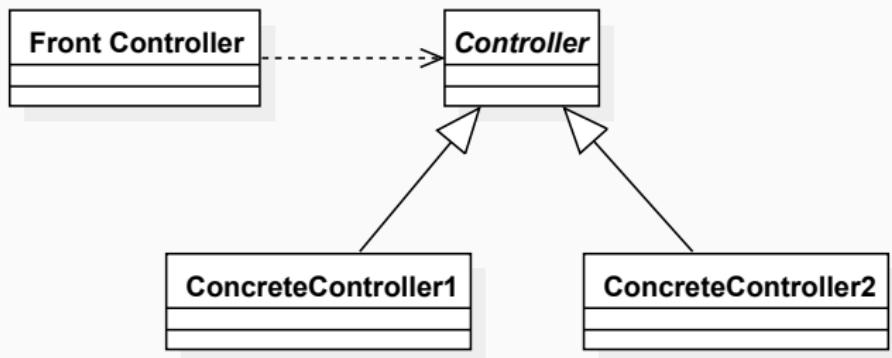
Es manejable cuando la navegación es simple. Para aplicaciones más complejas es muy difícil de gestionar.

Alternativa → Patrón Front Controller

Front Controller

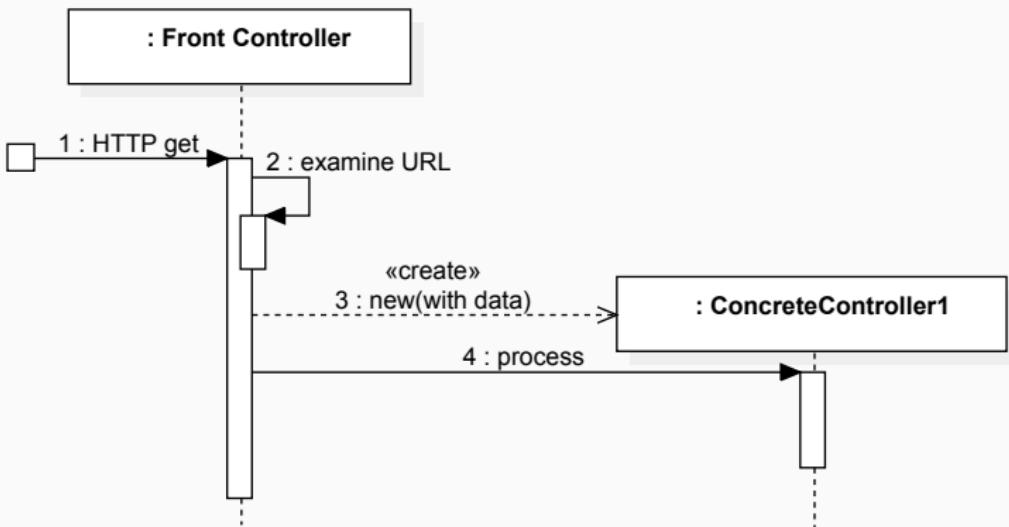
Patrón Front Controller [Fowler, 2003]

Un controlador que gestiona todas las peticiones de un sitio web.



Front Controller

El **Front Controller** realiza el comportamiento común a todas las acciones, y luego delega en controladores específicos para cada acción.



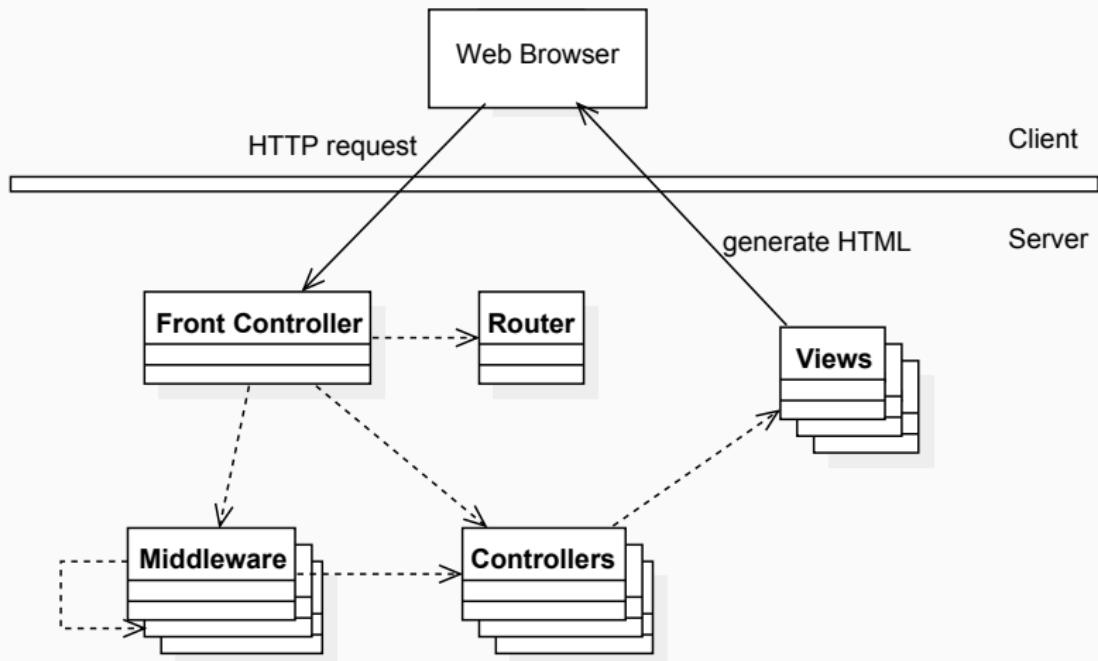
Front Controller

La principal ventaja frente al patrón Page Controller es que ahora un solo objeto controlador gestiona todas las peticiones para evitar duplicación de comportamiento.

En frameworks web modernos, el Front Controller delega algunas de sus responsabilidades en otros objetos:

- Objetos **Middleware** para chequeos de seguridad, internacionalización, etc.
- Un enrutador (**Router**) para seleccionar el controlador específico que corresponde a cada acción
- Si la lógica para gestionar la navegación es compleja se puede usar un **Application Controller**

Front Controller



Estructura de un framework web actual

Middleware

Cada objeto Middleware se especializa en realizar un tipo de comprobación:

- Verificar si el usuario está autentificado
- Comprobar si el usuario tiene permisos para ejecutar la acción
- Limitar el acceso a los recursos por número de accesos o ancho de banda (*throttling*)
- Comprobar y/o modificar los valores de entrada
- ...

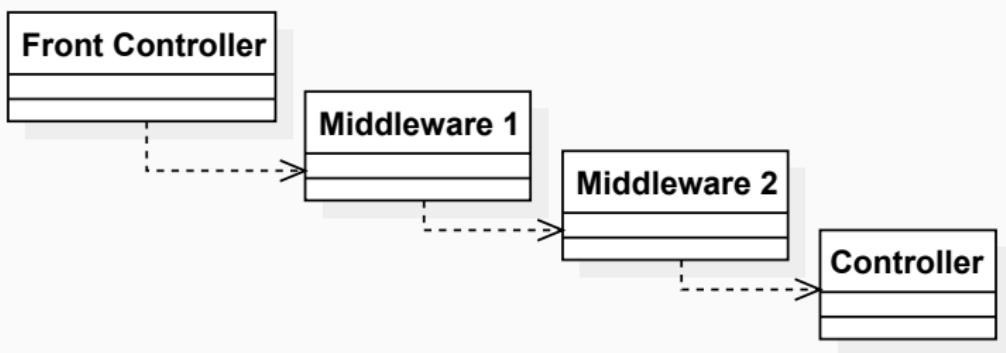
Los frameworks incorporan algunos Middleware por defecto (p.ej. comprobar usuario autentificado).

También se pueden crear objetos Middleware personalizados.

Middleware

Se pueden encadenar distintos objetos Middleware para realizar varias comprobaciones antes de ejecutar el controlador:

- Si las comprobaciones son correctas pasa la ejecución al siguiente Middleware, o finalmente al Controlador
- Si alguna de las condiciones definidas en los Middleware no se cumple se puede detener la ejecución. Por eso se les llama también **Filtros**



Router

El objeto **Router** es el responsable de seleccionar el controlador apropiado para cada petición.

En algunos frameworks como Laravel las rutas se especifican en un archivo de **rutas** que el Router carga al iniciar la aplicación:

```
Route::get('/', 'HomeController@index');
Route::get('post/create', 'PostController@create');
Route::post('post', 'PostController@store');
```

Otros frameworks como ASP.NET MVC están configurados para hacer una correspondencia entre la ruta y el método del controlador a ejecutar. Por ejemplo, la ruta /Product/Edit/3 se correspondería con:

- Controlador = Product
- Método = Edit
- id = 3

Application Controller

En ocasiones un controlador no puede decidir directamente cuál es la siguiente vista a mostrar, ya que puede depender del estado de los modelos o de la ejecución de una regla de negocio.

Ejemplo

Para una compañía aseguradora, después de modificar los datos de un parte tras la actuación de un especialista podrían pasar dos cosas:

- Si el incidente se ha resuelto mostraría la vista para cerrar el parte
- Si es necesaria la actuación de otro especialista se mostraría la vista correspondiente

Application Controller

En estos casos es conveniente mantener un **Application Controller** en una capa separada, p.ej. en la capa de lógica de dominio.

Si el flujo de trabajo es complejo podría ser útil usar una máquina de estados, representada por algún tipo de metadatos.

Patrón Model-View-Presenter

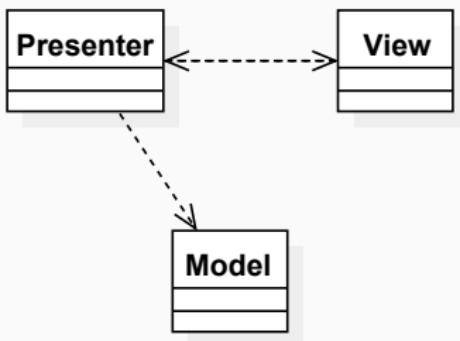
Patrón MVP

El patrón MVC tiene varios inconvenientes:

- La vista está acoplada al modelo, si no tomamos precauciones podemos acabar con vistas que deben hacer un procesamiento complejo para mostrar los datos
- Cada vista tiene su propio controlador, hay poca reutilización de código
- Es difícil probar los controladores porque están acoplados con vistas concretas

Patrón Model-View-Presenter

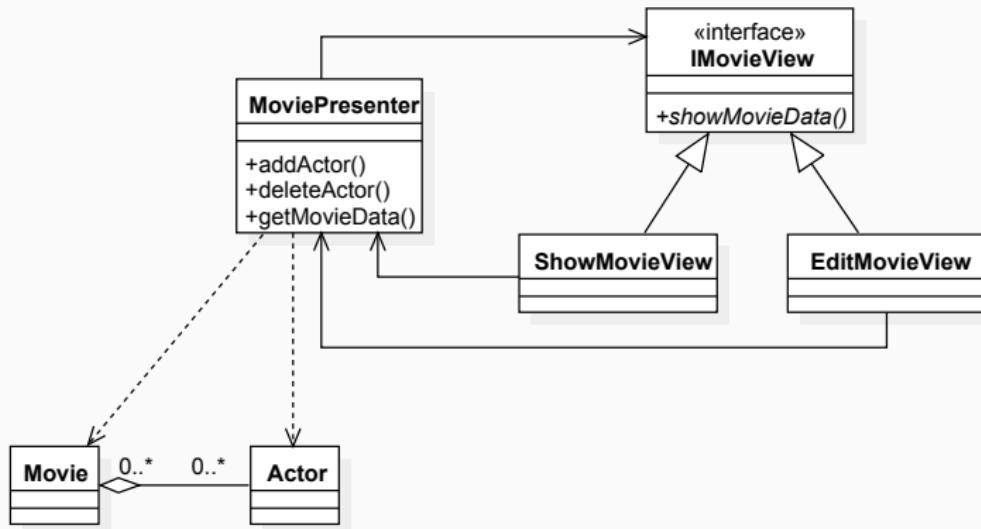
Para solucionar estos problemas, el patrón Model-View-Presenter (MVP) propone el uso de un objeto **Presenter** en lugar del controlador de MVC.



La vista ya no depende del modelo, ahora el objeto Presenter lee el modelo y prepara los datos para pasárselos a la vista.

Patrón Model-View-Presenter

Para facilitar la reutilización de código, los objetos Presenter están desacoplados de las vistas concretas que los usan.



De esta manera también es más fácil probar los Presenter **inyectándoles** vistas mock.

Patrón Model-View-Presenter

Cuando se crea una instancia de la vista, ésta crea a su vez un Presenter para que cargue la información que necesita.

Si la aplicación es distribuida el Presenter realiza la petición en un hilo paralelo, de manera que la vista puede continuar su ejecución sin bloquear el interfaz. Cuando el presenter recibe la información, pasa los datos a la vista para que los muestre.

Cada vez que una vista necesita nueva información se la pide al Presenter siguiendo el mismo procedimiento.

Patrón Model-View-ViewModel

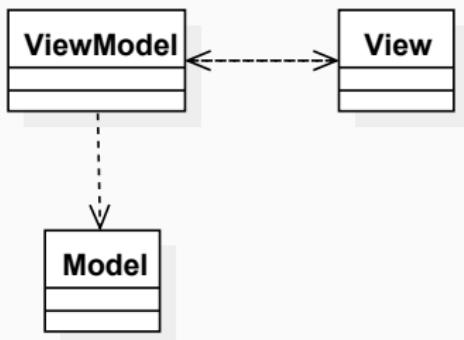
Patrón MVVM

Cuando una aplicación tiene mucha interacción en el interfaz, la complejidad del código puede aumentar demasiado:

- Una sola vista puede necesitar datos de muchos modelos
- Hay que gestionar todos los eventos de los controles gráficos del interfaz y responder adecuadamente a cada acción
- Un cambio en el modelo puede necesitar que se actualicen varias partes de una pantalla

Patrón MVVM

Para este tipo de aplicaciones surgió el patrón Model-View-ViewModel (MVVM).



El objeto **ViewModel** es una representación del modelo en la capa de presentación, que además contiene la lógica necesaria para responder a los eventos del interfaz.

Patrón MVVM

El objeto ViewModel es el responsable de comunicarse con la capa de lógica de negocio cuando se solicita desde la vista.

El verdadero potencial de este patrón está en el uso de **Data Binding**, creando un enlace entre los controles del interfaz y los objetos ViewModel:

- Cuando el usuario actualiza el campo de un formulario, se actualiza automáticamente la propiedad asociada del ViewModel
- Cuando el ViewModel se actualiza como resultado de una llamada a la lógica de negocio, se actualizan automáticamente los controles asociados en el interfaz

Esto permite construir interfaces con mucho menos código.

Aplicaciones web RIA

Rich Internet Application (RIA)

Una Aplicación Rica (o enriquecida) de Internet es una aplicación web que tiene un interfaz con características similares a las aplicaciones de escritorio.

Se pueden usar distintas tecnologías para implementarlas:
JavaScript, Java Applets, Flash, Silverlight, ...

Las más extendidas actualmente son las aplicaciones de tipo
Single Page Application (SPA), y usan JavaScript y AJAX en el cliente.

Aplicaciones web RIA

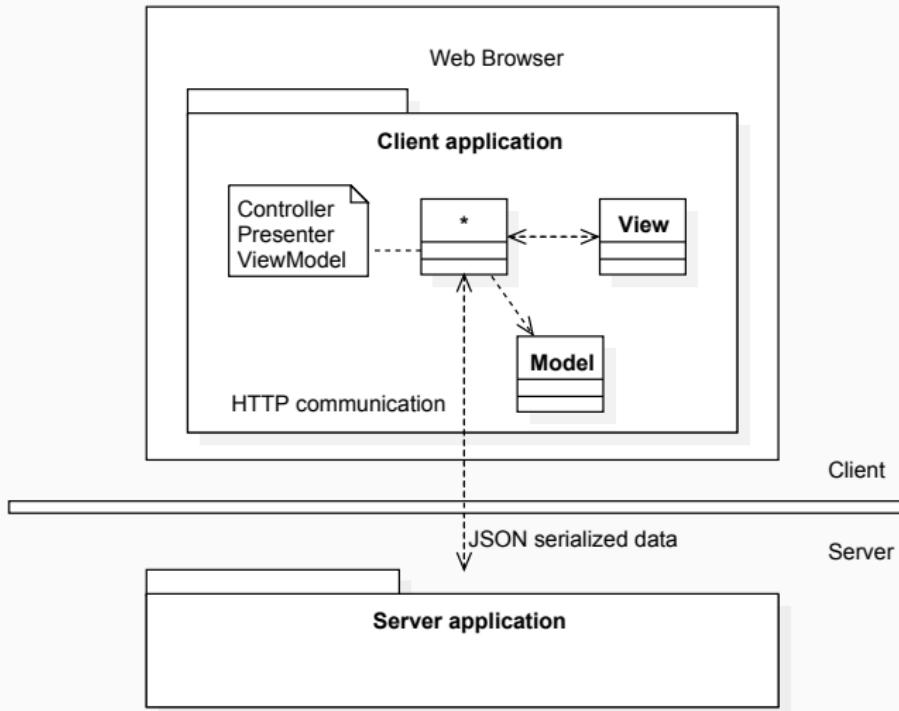
Funcionamiento de una aplicación SPA:

- El cliente carga un HTML mínimo y los scripts con la lógica de la aplicación cliente
- El interfaz se construye dinámicamente, generando el HTML necesario a partir de los datos proporcionados por el servidor
- El cliente realiza peticiones AJAX al servidor, la página no se recarga
- Los datos se transmiten serializados en formato JSON

Independientemente de la arquitectura usada en el servidor, estas **aplicaciones cliente necesitan estructurar su código.**

Para esto existen frameworks que favorecen el uso de los distintos **patrones MV***.

Aplicaciones web RIA



Estructura típica de una aplicación SPA

¿Preguntas?

Referencias i

-  Fowler, M. (2003).
Patterns of Enterprise Application Architecture.
Addison-Wesley Professional.
-  Osmani, A. (2015).
Learning JavaScript Design Patterns.
O'Reilly Media.
[https://addyosmani.com/resources/essentialjsdesignpatterns/book/.](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

Diseño responsive

Diseño de Sistemas Software
Curso 2020/2021

Carlos Pérez Sancho



Universitat d'Alacant
Universidad de Alicante

Departament de Llenguatges i Sistemes Informàtics
Departamento de Lenguajes y Sistemas Informáticos

Contenidos

1. Introducción
2. Diseño responsive
3. Bootstrap 4
4. Integración Bootstrap/Laravel

Introducción

Diseño de interfaces

Cuando se diseña un interfaz hay que tener en cuenta

- El tipo de público al que va dirigido
- Los dispositivos en los que se va a visualizar la aplicación

El público objetivo determina el estilo visual y el vocabulario empleado en el interfaz

Los dispositivos objetivo condicionan la forma en que se muestra la información en pantalla

Diseño de interfaces

Los usuarios están acostumbrados a un aspecto visual y un comportamiento característicos (**patrones**) de cada tipo de aplicación

Familias de aplicaciones: aplicaciones de gestión, ofimática, redes sociales, ...

El aspecto de la aplicación debe ser consistente en todos los dispositivos, pero al mismo tiempo debe adaptarse a las características de cada uno

- Tamaño de pantalla
- Introducción de datos

Diseño de interfaces

Actualmente lo más común es crear aplicaciones multiplataforma

- Herramientas para generar aplicaciones nativas en distintos dispositivos: Xamarin, Universal Windows Platform (UWP)
- Aplicaciones web que se ejecutan en navegadores

Las tecnologías web son omnipresentes

- Navegadores
- Webviews en móviles (Apache Cordova, PhoneGap)
- Aplicaciones de escritorio (Electron)
- Aplicaciones Web Progresivas (PWA)

Diseño de interfaces

Independientemente de la tecnología de desarrollo (UWP, web, aplicaciones móviles nativas) no podemos saber en qué tipo de pantalla se va a mostrar la aplicación

Existe la necesidad de crear diseños “fluidos” que se adapten automáticamente al tamaño de la pantalla: Diseño Adaptable o **Responsive Design**

Convergencia en la forma de diseñar interfaces

Diseño web adaptable

Adaptive web design

Técnicas para adaptar el contenido de una página web a algún dispositivo

Responsive Web Design (RWD)

Técnicas para que el contenido de una página web se adapte **automáticamente** al tamaño de pantalla del dispositivo

En español normalmente se usa “Adaptable” como sinónimo de “Responsive”.

¿Por qué es necesario?

Si no hacemos nada, una página web se ve en un móvil igual que en un navegador de escritorio



¿Por qué es necesario?

The screenshot shows the old website's layout. At the top, there is a header with the university's name and some social media links. Below this, a large banner features several links: 'Portal de transparencia', 'datos.ua.es', 'La UA en cifras...', 'Perfil del contratante' (with a cursor icon pointing at it), and 'Boletín Oficial (BOUA)'. The main content area has sections for 'EUA', 'Sociedad Europea de Universidades', 'CRUE', and 'ACQUA'. At the bottom, there is contact information, a map, and links for 'Contactar', 'Reportar' (with a cursor icon pointing at it), and 'Seguir a UA, Universidad'.

The screenshot shows the new website's layout. It features a header with the university's name and social media links. Below this is a large search bar and a menu icon. The main content area has three main sections: 'UA, Universidad Abierta', 'Portal de transparencia' (with a cursor icon pointing at it), and 'datos.ua.es'. Each section includes a brief description and a small icon. At the bottom, there is a link for 'Boletín Oficial (BOUA)'.

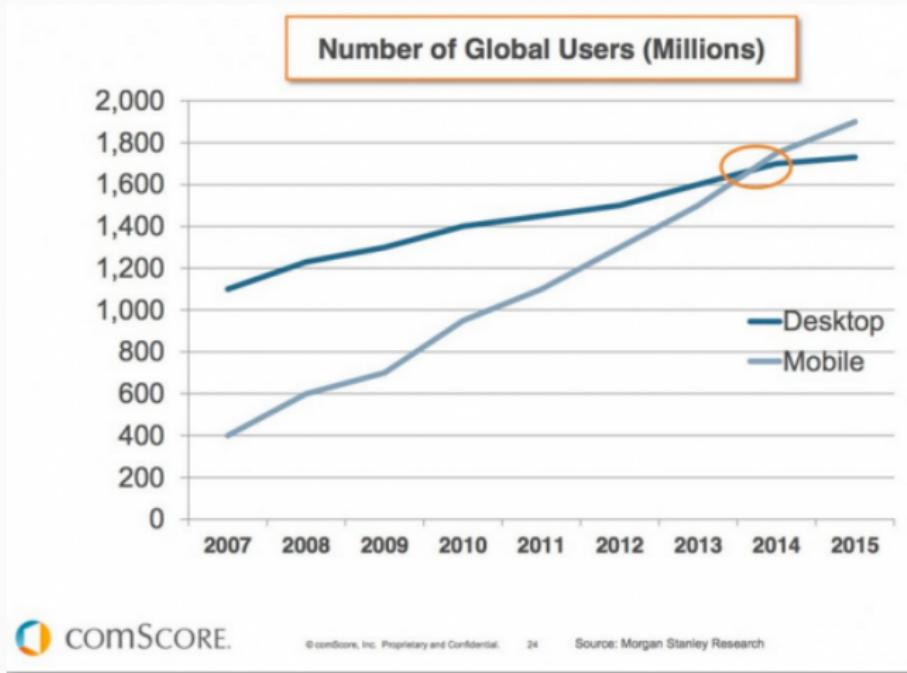
Mejor experiencia de usuario (UX) → disminuye la tasa de rebote

¿Por qué es necesario?

"Por primera vez en España hay más usuarios de internet (76,2 %) que de ordenador (73,3 %). El 77,1 % de los internautas accedieron a internet mediante el teléfono móvil."

Instituto Nacional de Estadística, 2 de octubre de 2014
<http://www.ine.es/prensa/np864.pdf>

¿Por qué es necesario?



<http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics>

¿Por qué es necesario?

Finding more mobile-friendly search results

Thursday, February 26, 2015

Webmaster level: all

When it comes to search on mobile devices, users should get the most relevant and timely results, no matter if the information lives on mobile-friendly web pages or apps. As more people use mobile devices to access the internet, our [algorithms](#) have to adapt to these usage patterns. In the past, we've made updates to ensure a site is [configured properly](#) and [viewable on modern devices](#). We've made it easier for users to [find mobile-friendly web pages](#) and we've introduced [App Indexing](#) to surface useful content from apps. Today, we're announcing two important changes to help users discover more mobile-friendly content:

1. More mobile-friendly websites in search results

Starting April 21, we will be expanding our use of mobile-friendliness as a ranking signal. This change will affect mobile searches in all languages worldwide and will have a significant impact in our search results. Consequently, users will find it easier to get relevant, high quality search results that are optimized for their devices.

<https://webmasters.googleblog.com/2015/02/finding-more-mobile-friendly-search.html>

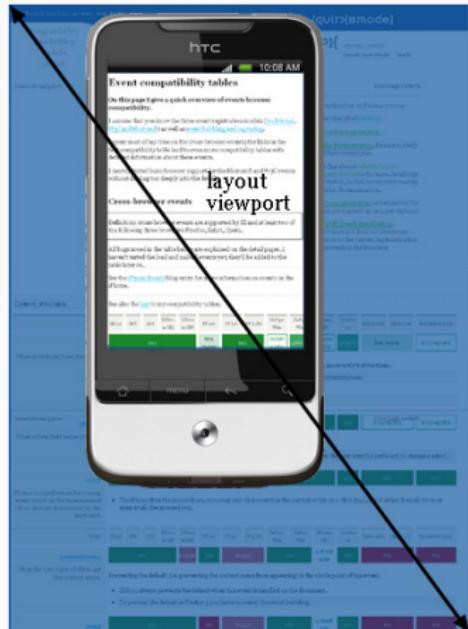
¿Qué vamos a aprender?

- Fundamentos del diseño responsive
- Funcionamiento básico de Bootstrap
- Creación de una plantilla y páginas usando Bootstrap en aplicaciones Laravel

Diseño responsive

Viewport

- Muchas páginas web están diseñadas para escritorio, con un tamaño fijo en píxeles
- Para poder mostrarlas correctamente, los navegadores móviles dibujan la página en un área (**viewport**) de tamaño similar al de una pantalla de escritorio: unos 980px, aunque puede variar

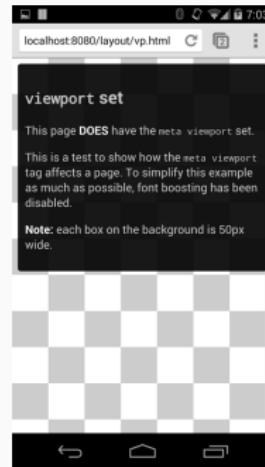


http://www.quirksmode.org/blog/archives/2010/04/a_pixel_is_not.html

Viewport

El tamaño del viewport se puede controlar con la etiqueta <meta>

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```



<https://developers.google.com/web/fundamentals/design-and-ux/responsive/>

Densidades de pantalla

No todos los dispositivos tienen la misma resolución con el mismo tamaño de pantalla.

Densidad de pantalla

Cantidad de píxeles que caben en la pantalla

Normalmente se mide en puntos por pulgada (dots per inch, **dpi**):

- ldpi ($\approx 120 \text{ dpi}$)
- mdpi ($\approx 160 \text{ dpi}$)
- hdpi ($\approx 240 \text{ dpi}$)
- xhdpi ($\approx 320 \text{ dpi}$), xxhdpi ($\approx 480 \text{ dpi}$), xxxhdpi ($\approx 640 \text{ dpi}$), ...

Un píxel no es un píxel

Usando píxeles reales, una página con diseño fijo se vería muy pequeña en dispositivos con densidades altas de pantalla.

Para que las páginas sean legibles, los navegadores móviles aumentan el tamaño de los píxeles CSS.

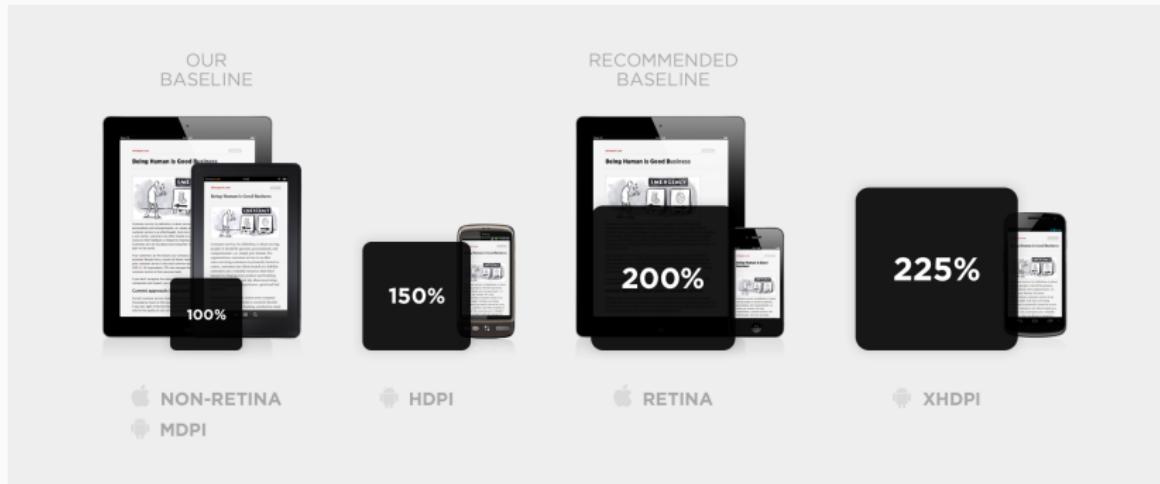


A screenshot of the Viewport Sizes website (viewportsizes.com). The search bar shows "galaxy s3". The results table displays the following information:

Device	Platform	Version	Portrait	Landscape
Samsung Galaxy S3 I9300	Android	4.0.4	360	640

Un píxel no es un píxel

Se usa como referencia el tamaño real de un píxel en dispositivos con densidad mdpi (160dpi)



<http://web.archive.org/web/20161020193410/http://www.teehanlax.com/blog/density-converter/>

Sistemas de rejillas

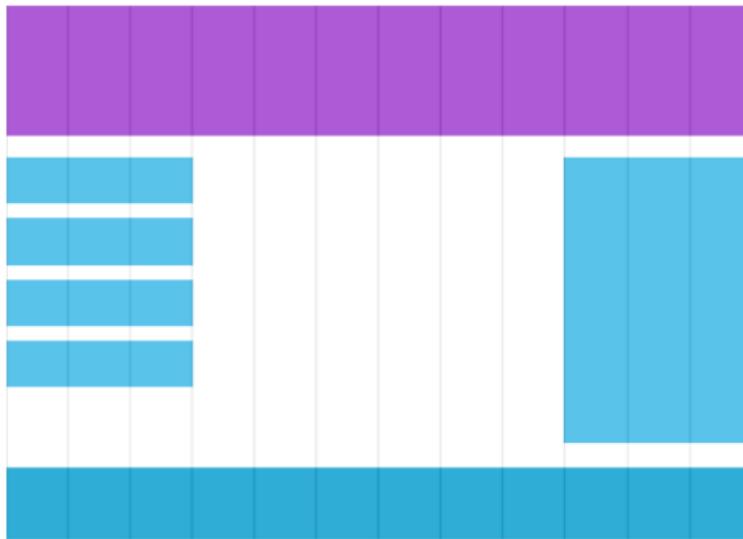
Muchas páginas estructuran el contenido mediante una rejilla (*grid*), que divide la página en columnas.

El uso de rejillas facilita el posicionamiento de los elementos en la pantalla colocando los elementos en celdas.



Sistemas de rejillas

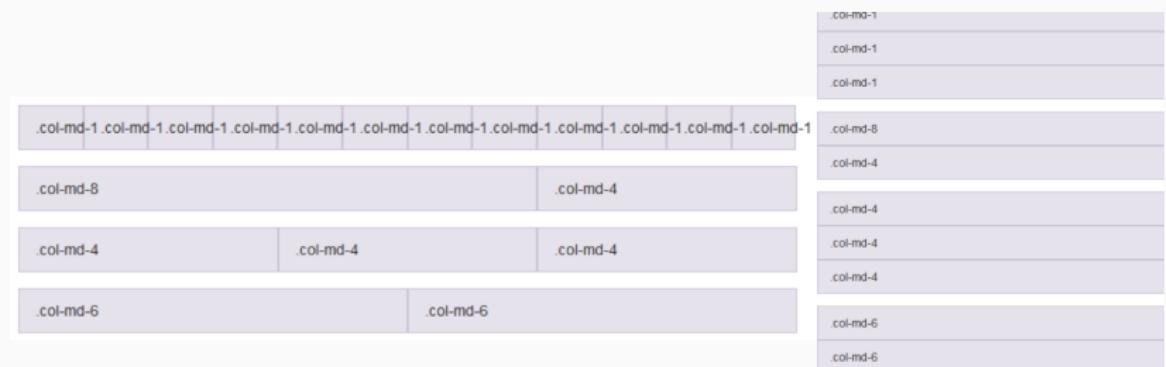
Normalmente se definen rejillas de 12 columnas, que se redimensionan automáticamente al cambiar el tamaño de la pantalla.



http://www.w3schools.com/css/tryresponsive_grid.htm

Fluid grids

Una “rejilla fluida” reestructura el contenido apilando las columnas cuando el tamaño de la pantalla es demasiado pequeño.



Vista en pantallas grandes

Vista en pantallas
pequeñas

<https://getbootstrap.com/docs/4.6/examples/grid/>

Fluid grids

Para decidir cuándo debe cambiarse la disposición del contenido, las rejillas fluidas definen una serie de puntos de interrupción (**breakpoints**)

Se definen breakpoints para varios tamaños de pantalla

	Extra small <576px	Small ≥576px	Medium ≥768px	Large ≥992px	Extra large ≥1200px
.container	100%	540px	720px	960px	1140px
.container-sm	100%	540px	720px	960px	1140px
.container-md	100%	100%	720px	960px	1140px
.container-lg	100%	100%	100%	960px	1140px
.container-xl	100%	100%	100%	100%	1140px
.container-fluid	100%	100%	100%	100%	100%

Breakpoints en Bootstrap 4

Media queries

Media queries: introducidas en CSS3, permiten aplicar un conjunto de reglas sólo si se cumplen unas determinadas condiciones.

Permiten identificar el tipo de dispositivo, el tamaño y la orientación de la pantalla.

```
@media only screen and (max-width: 500px) {  
    body {  
        background-color: lightblue;  
    }  
}
```

http://www.w3schools.com/css/tryit.asp?filename=tryresponsive_mediaquery

Imágenes responsive

- Técnica básica: **max-width**

```
img {  
    max-width: 100%;  
    height: auto;  
}
```

- Distintas imágenes para distintas densidades de pantalla:

```
  
  
This is used as the lx src & by  
browsers that don't support srcset  
  
Image url Pixel density of screen
```

- Adaptar el nivel de detalle al tipo de pantalla



<https://jakearchibald.com/2015/anatomy-of-responsive-images/>

Bootstrap 4

Bootstrap 4

Bootstrap es un framework para el diseño de interfaces con HTML y CSS

The screenshot shows the official Bootstrap website. At the top, there's a purple header bar with the Bootstrap logo (a stylized 'B' icon), navigation links for Home, Docs, Examples, Icons, Themes, and Blog, and social media icons for GitHub, Twitter, and others. A prominent 'Download' button is also in the header. Below the header, the main title 'Build fast, responsive sites with Bootstrap' is displayed in large, bold, black font. To the right of the title is a graphic featuring a large purple hexagon containing a white letter 'B', overlaid on a background of blurred wireframe web interface designs. Below the title, a descriptive paragraph explains that Bootstrap is a mobile-first toolkit with a responsive grid system, Sass variables, mixins, and JavaScript plugins. At the bottom of the main content area are two buttons: 'Get started' (purple) and 'Download' (white). At the very bottom of the page, there's a footer note: 'Currently v5.0.0-beta2 · [v4.6.x docs](#) · [All releases](#)'.

<https://getbootstrap.com/>

Bootstrap 4

Bootstrap proporciona

- Una rejilla fluida para posicionar el contenido
- Herramientas para dar formato a distintos tipos de contenido: tablas, imágenes, código fuente, ...
- Una colección de componentes con estilos predefinidos: botones, menús, formularios, ...
- Utilidades para modificar el aspecto de los elementos anteriores: bordes, márgenes, alineación, ...

Todo esto mediante el uso de clases CSS para no tener que escribir nuestras propias reglas

Uso de Bootstrap

El primer paso es copiar la plantilla de la página de Bootstrap

The screenshot shows the Bootstrap documentation website at version v4.6. The main content area displays the 'Starter template' code example. The code includes an HTML5 doctype, meta tags for character encoding and viewport, Bootstrap CSS and JS imports, and a simple title and body. A 'Copy' button is located next to the code. To the right, a sidebar lists navigation links such as 'Quick start', 'CSS', 'JS', 'Bundle', 'Separate Components', 'Starter template', 'Important global', 'HTML5 doctype', 'Responsive meta tag', 'Box-sizing', 'Reboot', 'Community', 'CSPs and embedded SVGs'. At the bottom of the content area, there's a note about visiting the 'Layout docs' or 'examples' for more information.

```
<!doctype html>
<html lang="en">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=1">
    <!-- Bootstrap CSS -->
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.0/dist/css/bootstrap.min.css" />

    <title>Hello, world!</title>
  </head>
  <body>
    <h1>Hello, world!</h1>

    <!-- Optional JavaScript; choose one of the two! -->

    <!-- Option 1: jQuery and Bootstrap Bundle (includes Popper) -->
    <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js" integrity="sha384-DfXdz2htPH0lsSSs5nCTpuj/zy4C+OGpamoFVy30+DofGrTbR
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@4.6.0/dist/js/bootstrap.bundle.min.js" integrity="sha384-pwWZdHmQy+P3uM9HvOepeRmJ+iA6A9q9YDQgkN4oZqBxGhH" />
    <!-- Option 2: Separate Popper and Bootstrap JS -->
    <!--
    <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js" integrity="sha384-DfXdz2htPH0lsSSs5nCTpuj/zy4C+OGpamoFVy30+DofGrTbR
    <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.1/dist/umd/popper.min.js" integrity="sha384-9/reFTGAW83EW2RDu2S2UpWVXVEIYFmS1PbRrO7mgtKd4" />
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@4.6.0/dist/js/bootstrap.min.js" integrity="sha384-pwWZdHmQy+P3uM9HvOepeRmJ+iA6A9q9YDQgkN4oZqBxGhH" />
    -->
  </body>
</html>
```

That's all you need for overall page requirements. Visit the [Layout docs](#) or our official [examples](#) to start laying out your site's content and components.

<https://getbootstrap.com/docs/4.6/getting-started/introduction/#starter-template>

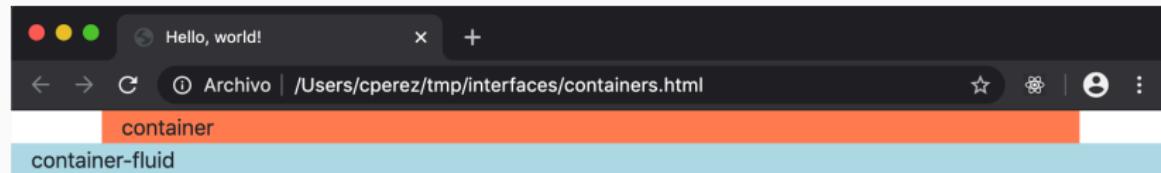
Contenedores

Los contenedores son los elementos más básicos de Bootstrap

```
<body>
  <div class="container">
    <h1>Hello world!</h1>
  </div>
</body>
```

El contenedor determina el ancho máximo del contenido en la ventana

- **container** tiene el tamaño del breakpoint inferior más cercano
- **container-fluid** ocupa siempre todo el ancho disponible



Grid

Las rejillas deben estar obligatoriamente dentro de un contenedor

El contenido se organiza en filas y columnas, usando etiquetas div y las clases proporcionadas por Bootstrap

Si no se especifica el tamaño de las columnas, se reparten el tamaño disponible a partes iguales

One of three columns

One of three columns

One of three columns

```
<div class="container">
  <div class="row">
    <div class="col-sm">
      One of three columns
    </div>
    <div class="col-sm">
      One of three columns
    </div>
    <div class="col-sm">
      One of three columns
    </div>
  </div>
</div>
```

Copy

Grid

Hay 12 espacios disponibles en cada fila para repartir entre sus columnas

col-sm-8

col-sm-4

col-sm

col-sm

col-sm

Copy

```
<div class="container">
  <div class="row">
    <div class="col-sm-8">col-sm-8</div>
    <div class="col-sm-4">col-sm-4</div>
  </div>
  <div class="row">
    <div class="col-sm">col-sm</div>
    <div class="col-sm">col-sm</div>
    <div class="col-sm">col-sm</div>
  </div>
</div>
```

Grid

Layout básico de una página

```
<div class="container">
  <div class="row">
    <div class="col-sm">Header</div>
  </div>
  <div class="row">
    <div class="col-sm-8">Body</div>
    <div class="col-sm-4">Sidebar</div>
  </div>
  <div class="row">
    <div class="col-sm">Footer</div>
  </div>
</div>
```

Se pueden anidar rejillas para crear estructuras más complejas

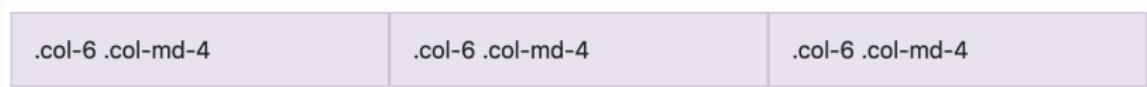
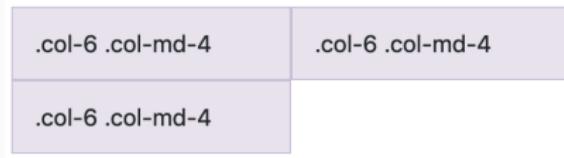
Responsive grid

Las columnas pueden tener varias clases para definir su comportamiento en distintos tamaños de pantalla

	Extra small ≤576px	Small ≥576px	Medium ≥768px	Large ≥992px	Extra large ≥1200px
Max container width	None (auto)	540px	720px	960px	1140px
Class prefix	.col-	.col-sm-	.col-md-	.col-lg-	.col-xl-

Responsive grid

```
<!-- Columns start at 50% wide on mobile  
     and bump up to 33.3% wide on desktop -->  
<div class="row">  
    <div class="col-6 col-md-4">.col-6 .col-md-4</div>  
    <div class="col-6 col-md-4">.col-6 .col-md-4</div>  
    <div class="col-6 col-md-4">.col-6 .col-md-4</div>  
</div>
```



Componentes

Para crear componentes podemos usar los ejemplos de Bootstrap como punto de partida

The screenshot shows the Bootstrap Documentation website. At the top, there's a navigation bar with links for Home, Documentation (which is active), Examples, Icons, Themes, Expo, and Blog. To the right of the navigation is a version indicator (v4.4) and a Download button. Below the navigation is a search bar labeled "Search...". On the left, there's a sidebar with a navigation menu. The "Components" section is expanded, showing sub-links for Alerts, Badge, Breadcrumb, Buttons, Button group, Card, Carousel, Collapse, Dropdowns, Forms, Input group, Jumbotron, List group, Media object, Modal, and Navs. The main content area has a title "Examples" and a sub-section "Buttons". It shows a row of colored buttons: Primary (blue), Secondary (gray), Success (green), Danger (red), Warning (yellow), Info (teal), Light (light blue), Dark (black), and Link (blue). Below the buttons is a code snippet demonstrating their HTML structure. A "Copy" button is located to the right of the code. At the bottom, there's a section titled "Conveying meaning to assistive technologies" with a note about using color for meaning and ensuring it's understandable for screen readers.

Home Documentation Examples Icons Themes Expo Blog

v4.4 Download

Search...

Getting started
Layout
Content

Components

Alerts
Badge
Breadcrumb
Buttons
Button group
Card
Carousel
Collapse
Dropdowns
Forms
Input group
Jumbotron
List group
Media object
Modal
Navs

Examples

Bootstrap includes several predefined button styles, each serving its own semantic purpose, with a few extras thrown in for more control.

Primary Secondary Success Danger Warning Info Light Dark Link

```
<button type="button" class="btn btn-primary">Primary</button>
<button type="button" class="btn btn-secondary">Secondary</button>
<button type="button" class="btn btn-success">Success</button>
<button type="button" class="btn btn-danger">Danger</button>
<button type="button" class="btn btn-warning">Warning</button>
<button type="button" class="btn btn-info">Info</button>
<button type="button" class="btn btn-light">Light</button>
<button type="button" class="btn btn-dark">Dark</button>
<button type="button" class="btn btn-link">Link</button>
```

Copy

Conveying meaning to assistive technologies

Using color to add meaning only provides a visual indication, which will not be conveyed to users of assistive technologies – such as screen readers. Ensure that information denoted by the color is either obvious from the content itself (e.g. the visible text), or is included through alternative means, such as additional text hidden with the `.sr-only` class.

Examples
Disable text wrapping
Button tags
Outline buttons
Sizes
Active state
Disabled state
Button plugin
Toggle states
Checkbox and radio buttons
Methods

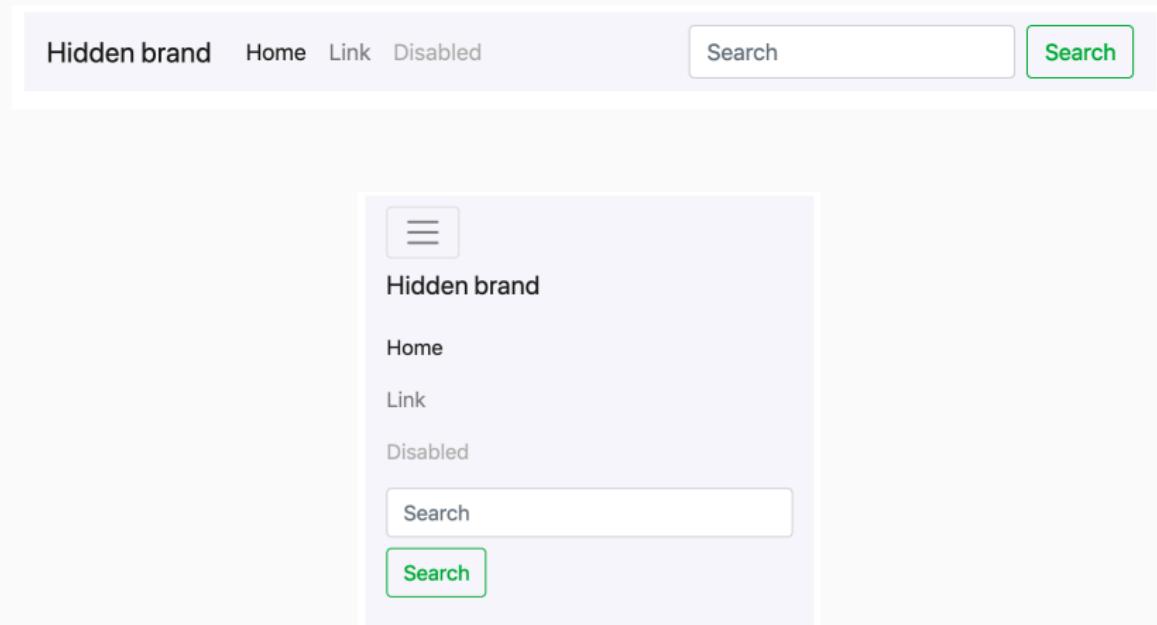
Componentes

Componentes más frecuentes

- Botones (buttons)
- Menús de navegación (NavBar)
- Formularios (Forms)
- Paginación (Pagination)

NavBar

Permite crear menús que se contraen en dispositivos móviles



Forms

Los formularios se adaptan al tamaño de la pantalla y tienen un estilo más agradable

Email address

We'll never share your email with anyone else.

Password

Check me out

Submit

Forms

Los campos de tipo input (donde el usuario escribe) se pueden complementar con 2 atributos de HTML5

- **placeholder**: texto que se muestra al usuario como pista
- **type**: text, password, email, number, ...

Especificar el tipo permite abrir un teclado especializado en dispositivos móviles

https://www.w3schools.com/html/html_form_input_types.asp

```
<input type="email" placeholder="name@example.com" ...>
```

Email address

name@example.com

Integración Bootstrap/Laravel

Código de ejemplo

Repositorio GitHub

<https://github.com/cperezs/laravel-bootstrap>

Rutas

routes/web.php

```
Route::get('/products', 'ProductController@list');
Route::get('/products/{id}', 'ProductController@details');
```

- **/products** muestra todos los productos
- **/products/{id}** muestra los detalles de un producto, el valor del parámetro id se pasa al método del controlador

Las dos rutas enlazan con métodos del controlador

ProductController

Controlador

app/Http/Controllers/ProductController.php

```
class ProductController extends Controller
{
    public function list() {
        $products = Product::all();
        return view('products.list')->with('products', $products);
    }

    public function details($id) {
        $product = Product::findOrFail($id);
        return view('products.details')->with('product', $product);
    }
}
```

Cada método realiza una consulta y pasa el resultado a la vista correspondiente

Por simplificar, no se ha hecho uso de ningún patrón de lógica de negocio en el ejemplo

views/layout.blade.php

```
<!doctype html>
<html lang="en">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

    <!-- Bootstrap CSS -->
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.0/dist/css/bootstrap.m

      <title>App name - @yield('title')</title>
    </head>
    <body>
      <div class="container">
        |   @yield('content')
      </div>

      <!-- Optional JavaScript -->
      <!-- jQuery first, then Popper.js, then Bootstrap JS -->
      <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js" integrity="sha384-DfXdz2htPH0ls
      <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.1/dist/umd/popper.min.js" integrity=
      <script src="https://cdn.jsdelivr.net/npm/bootstrap@4.6.0/dist/js/bootstrap.min.js" integrity=
    </body>
  </html>
```

Vista detalle

views/products/details.blade.php

```
@extends('layout')

@section('title', 'Product details')

@section('content')
<h1>{{ $product->name }}</h1>
<p><strong>Price:</strong> {{ $product->price }}</p>
<p>{{ $product->description }}</p>
<p><a href="{{ action('ProductController@list') }}">Go back</a></p>
@endsection
```

Vista lista productos

views/products/list.blade.php

```
@extends('layout')

@section('title', 'Product list')

@section('content')
<h1>Products</h1>
<table class="table table-striped">
    <thead>
        <tr>
            <th scope="col">Id</th>
            <th scope="col">Name</th>
            <th scope="col">Price</th>
        </tr>
    </thead>
    <tbody>
        @foreach ($products as $product)
        <tr>
            <th scope="row">{{ $product->id }}</th>
            <td><a href="{{ action('ProductController@details', $product->id) }}">{{ $product->name }}</a></td>
            <td>{{ $product->price }}</td>
        </tr>
        @endforeach
    </tbody>
</table>
@endsection
```

Vista lista productos

Crea una tabla usando los estilos de Bootstrap

En cada fila se crea un enlace para ver los detalles del producto

```
'  
@foreach ($products as $product)  
<tr>  
    <th scope="row">{{ $product->id }}</th>  
    <td><a href="{{ action('ProductController@details', $product->id) }}">{{ $product->name }}</a></td>  
    <td>{{ $product->price }}</td>  
</tr>  
@endforeach
```

¿Preguntas?

Otros patrones arquitecturales

Diseño de Sistemas Software

Curso 2020/2021

Carlos Pérez Sancho



Universitat d'Alacant
Universidad de Alicante

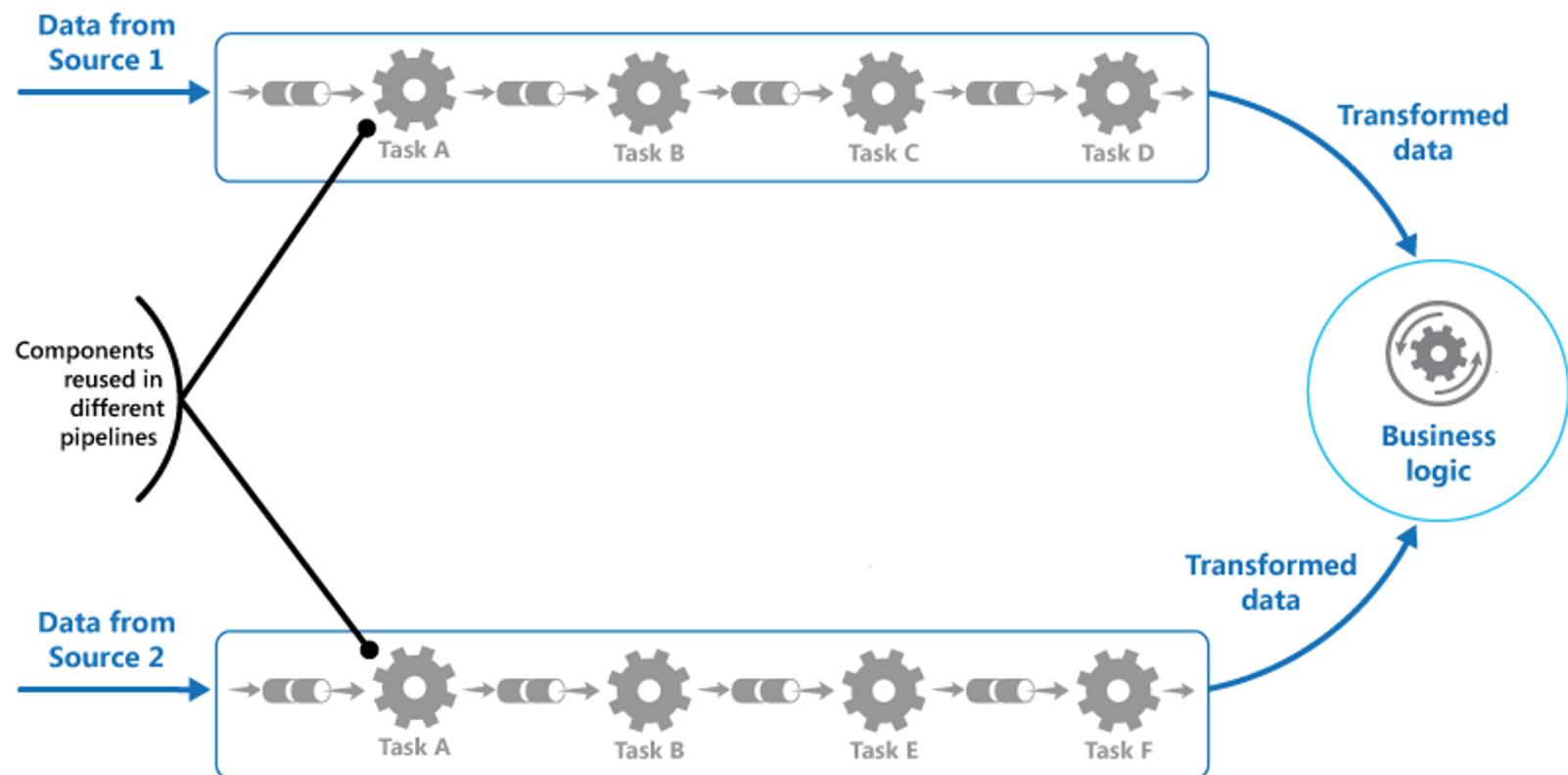
Departament de Llenguatges i Sistemes Informàtics
Departamento de Lenguajes y Sistemas Informáticos

Arquitectura tuberías y filtros

Patrones arquitecturales

Arquitectura tuberías y filtros

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



Arquitectura tuberías y filtros

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

Arquitectura tuberías y filtros

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

Arquitectura tuberías y filtros

- Ejemplo: Apertium
<https://www.apertium.org>
- Desarrollo de aplicaciones derivadas mediante la reutilización de módulos (interNOSTRUM → Apertium)



Arquitectura tuberías y filtros

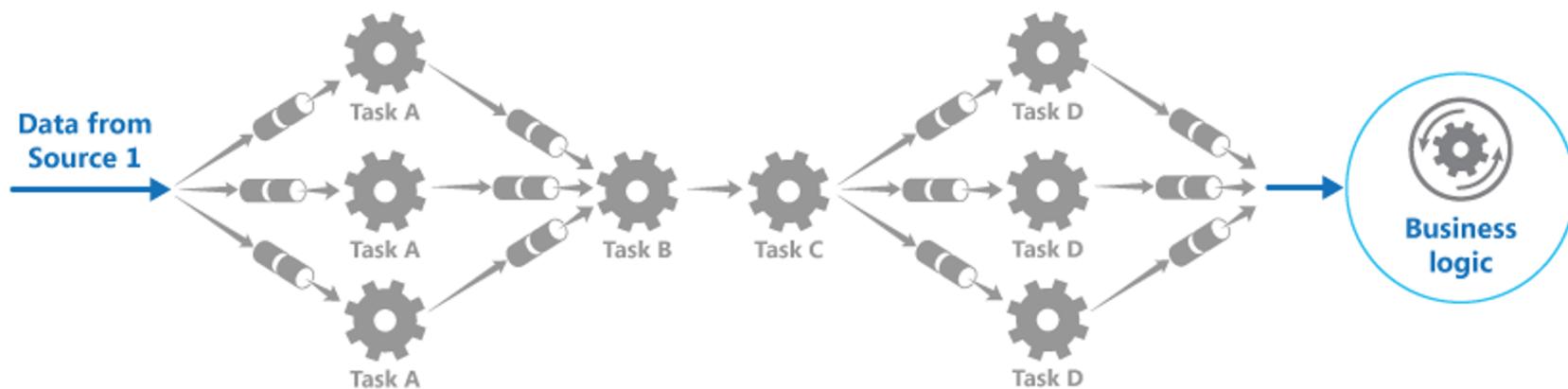
- Ventajas
 - Bajo acoplamiento entre componentes
 - Se pueden probar los componentes individualmente
 - Facilidad para añadir componentes nuevos o reordenar los existentes
 - Se puede hacer cambios en un componente de forma ágil
 - Los filtros actúan como “cajas negras”
 - Reusabilidad

Arquitectura tuberías y filtros

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

Arquitectura tuberías y filtros

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

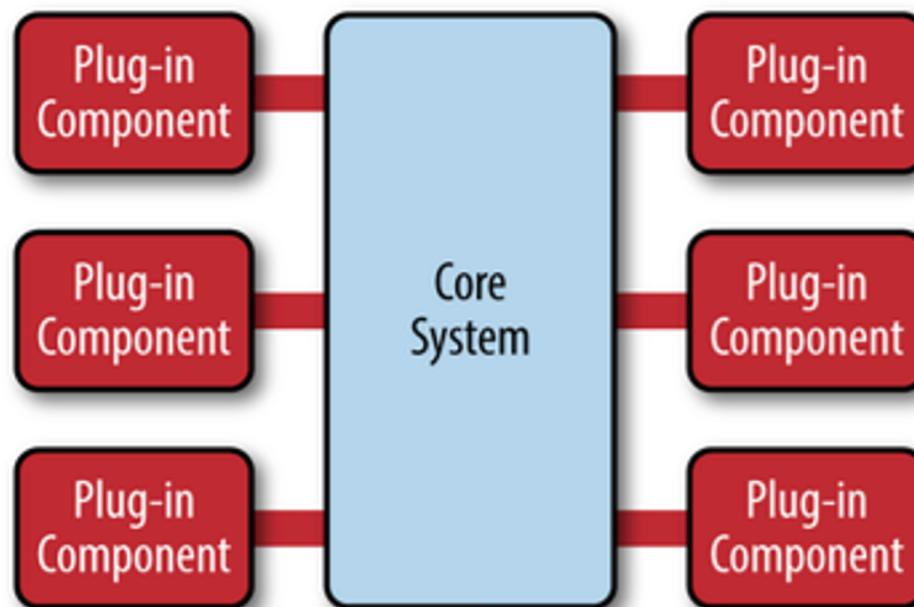


Arquitectura microkernel

Patrones arquitecturales

Arquitectura microkernel

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

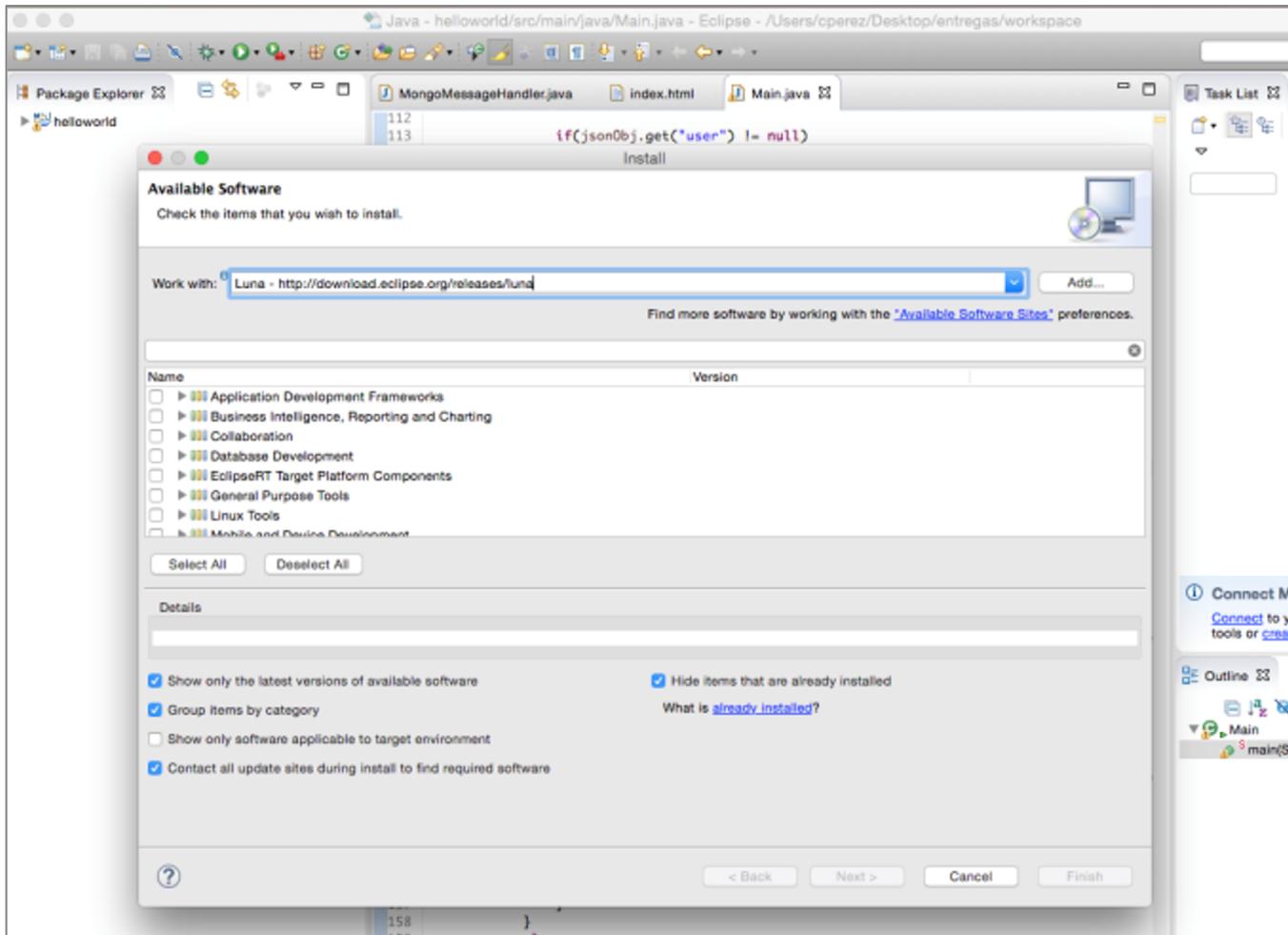


Arquitectura microkernel

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

Arquitectura microkernel

Ejemplo: Eclipse IDE



Arquitectura microkernel

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

Arquitectura orientada a eventos

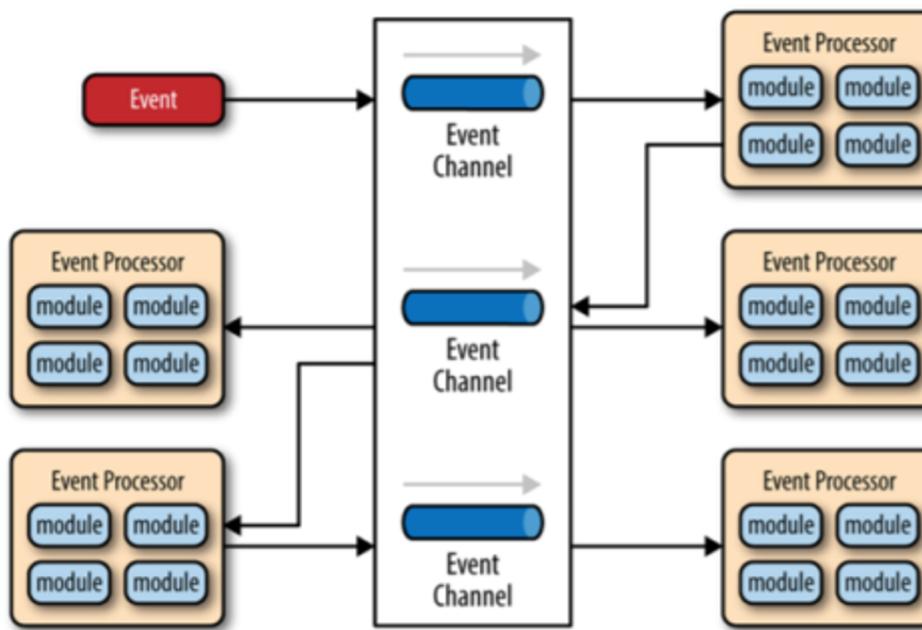
Patrones arquitecturales

Arquitectura orientada a eventos

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

Arquitectura orientada a eventos

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



Arquitectura orientada a eventos

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

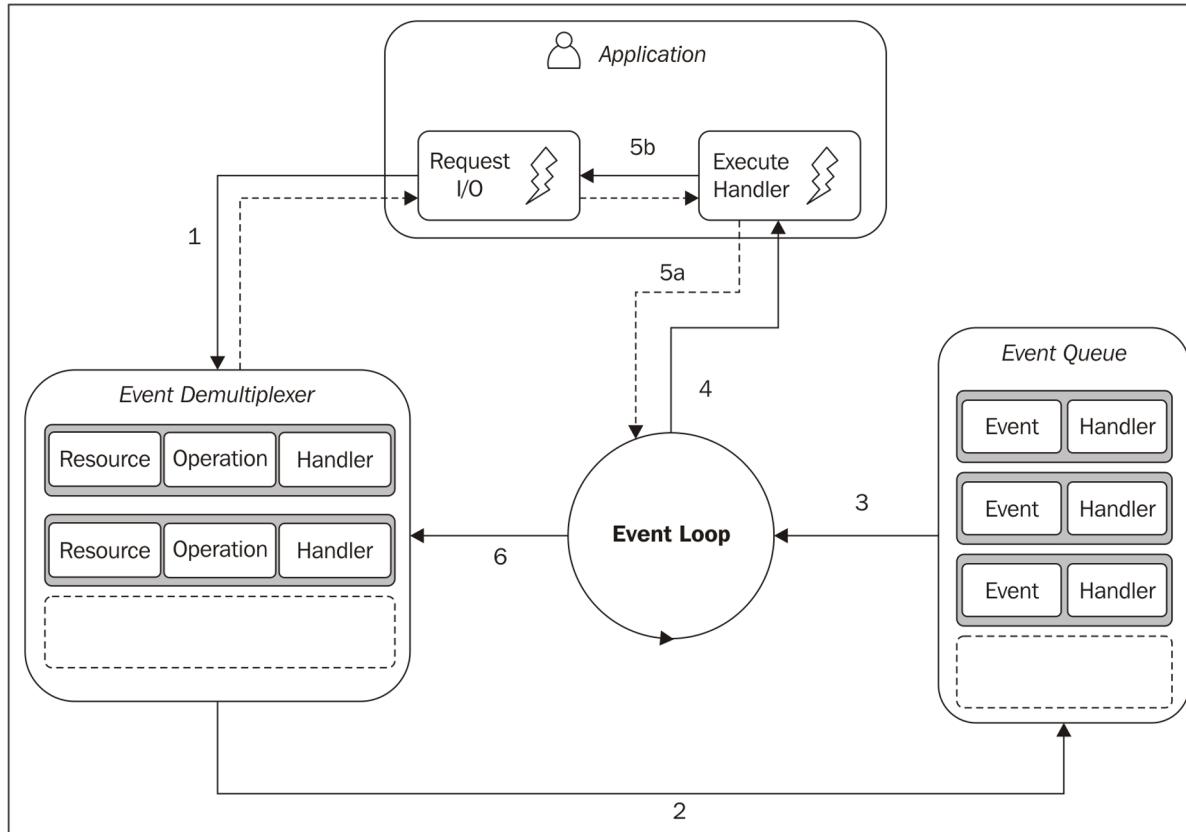
Arquitectura orientada a eventos

Ejemplo: un cliente de una aseguradora cambia de domicilio



Arquitectura orientada a eventos

Ejemplo: Node.js



Arquitectura orientada a eventos

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

Arquitectura orientada a eventos

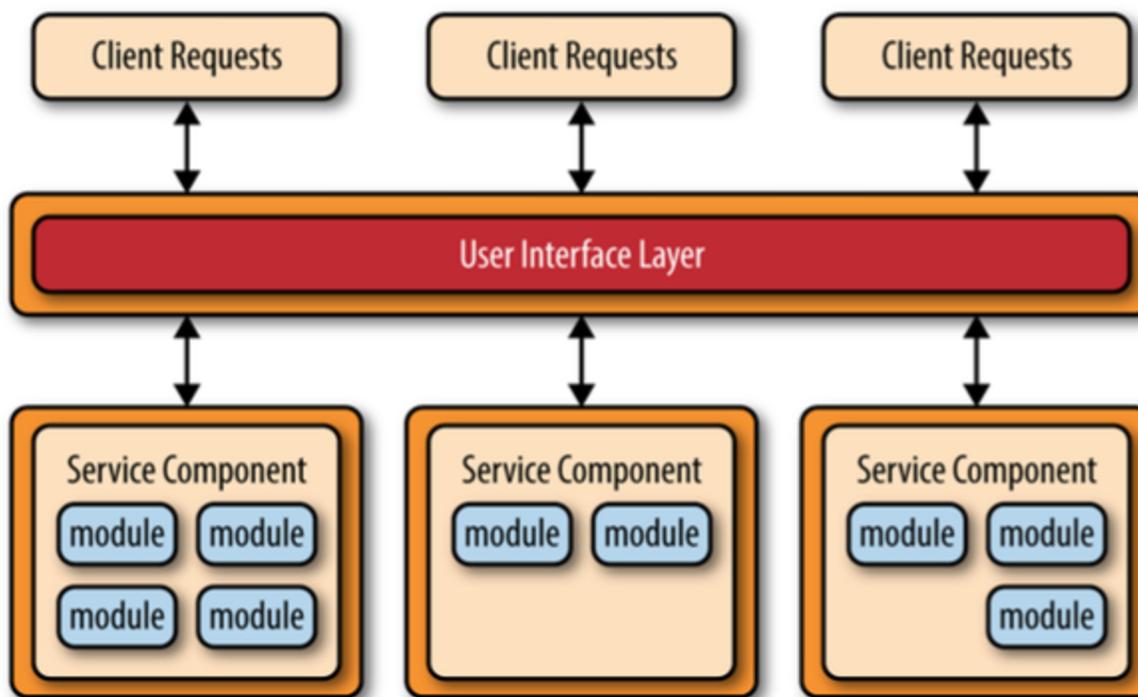
- Inconvenientes
 - Requiere herramientas especializadas para gestionar eventos
 - Por su naturaleza asíncrona puede dar lugar a condiciones de carrera

Arquitectura de microservicios

Patrones arquitecturales

Arquitectura de microservicios

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



Arquitectura de microservicios

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

Arquitectura de microservicios

- Ejemplo de interfaz REST

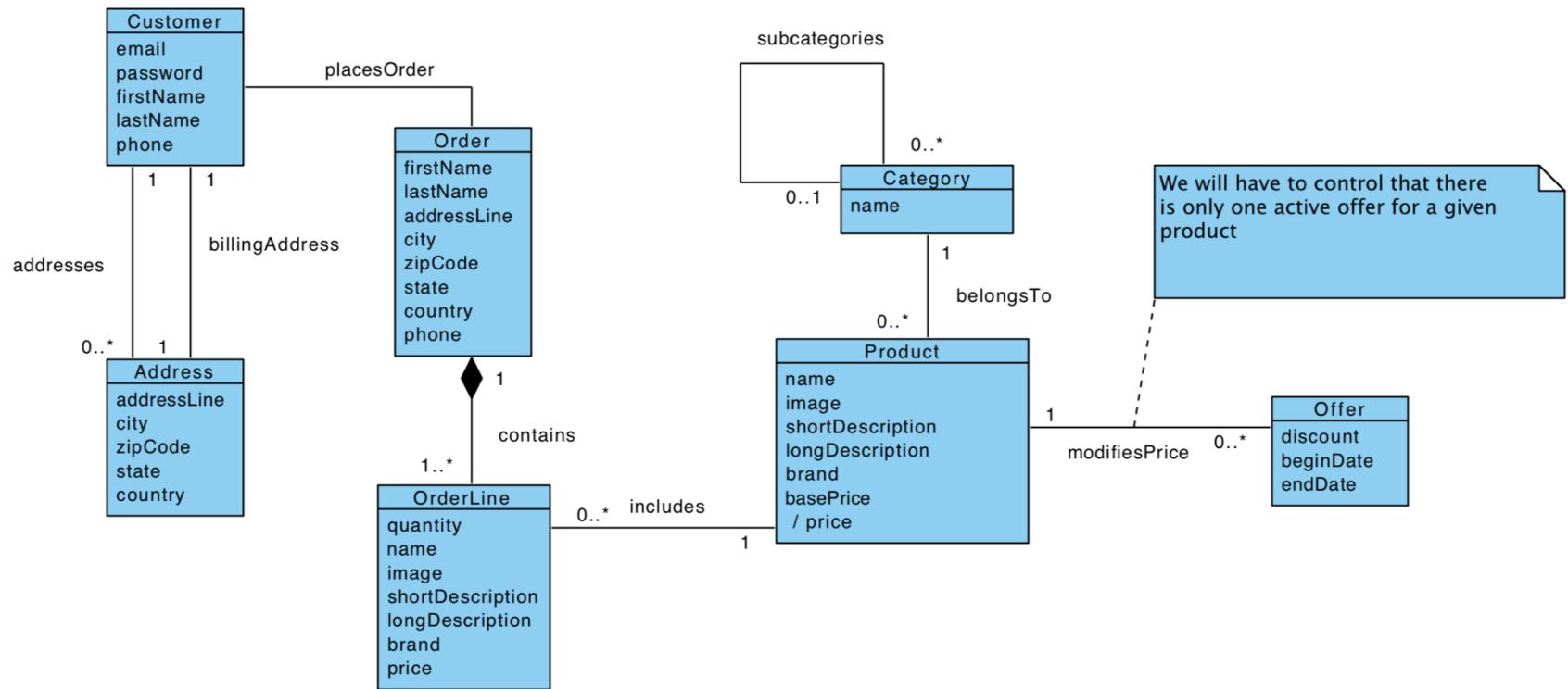
GET /products	Obtener todos
POST /products	Crear un producto
GET /products/{id}	Obtener un producto
PUT /products/{id}	Modificar un producto
DELETE /products/{id}	Borrar un producto

Arquitectura de microservicios

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

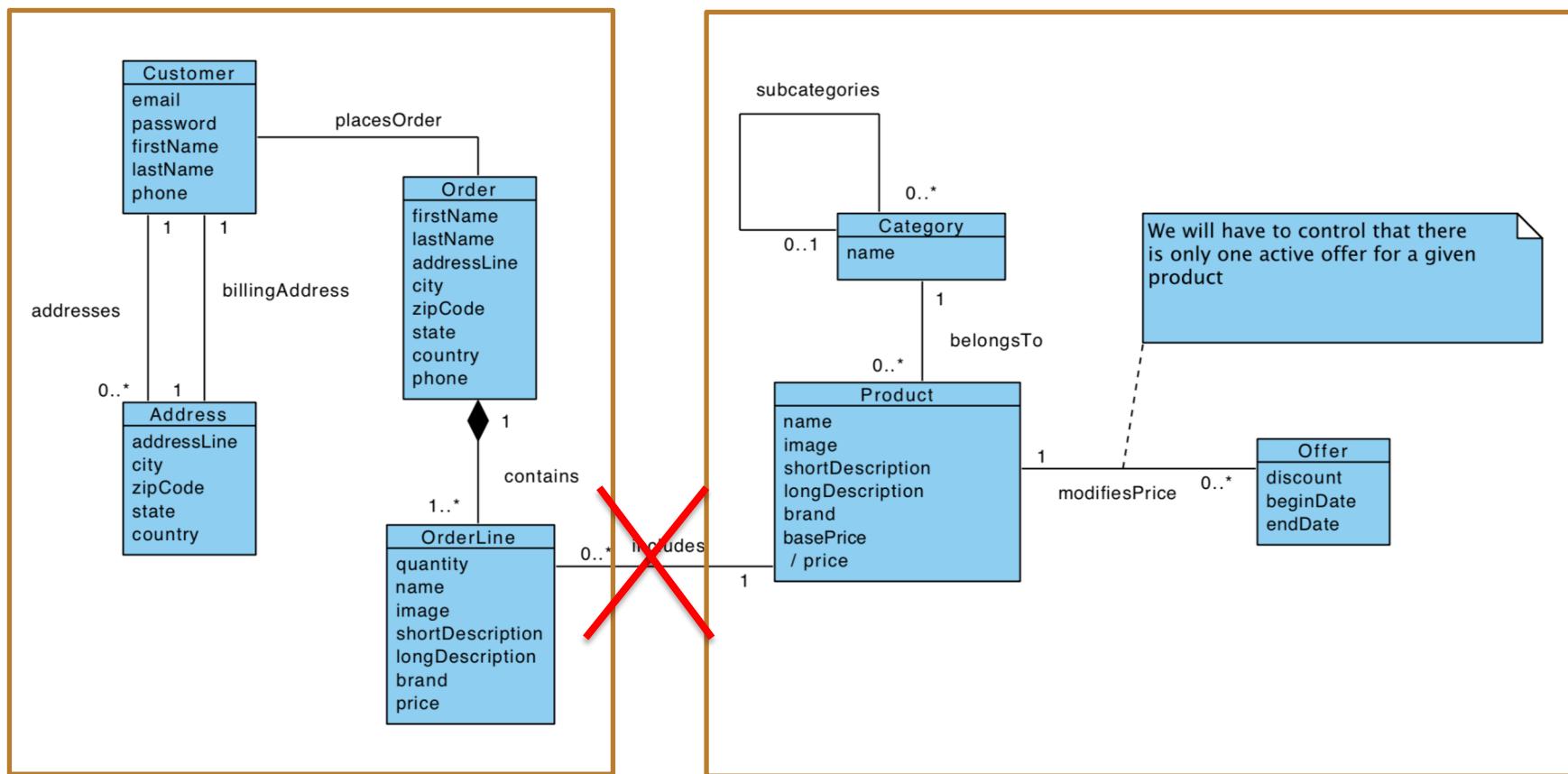
Arquitectura de microservicios

- Ejemplo



Arquitectura de microservicios

- Dividimos en dos servicios

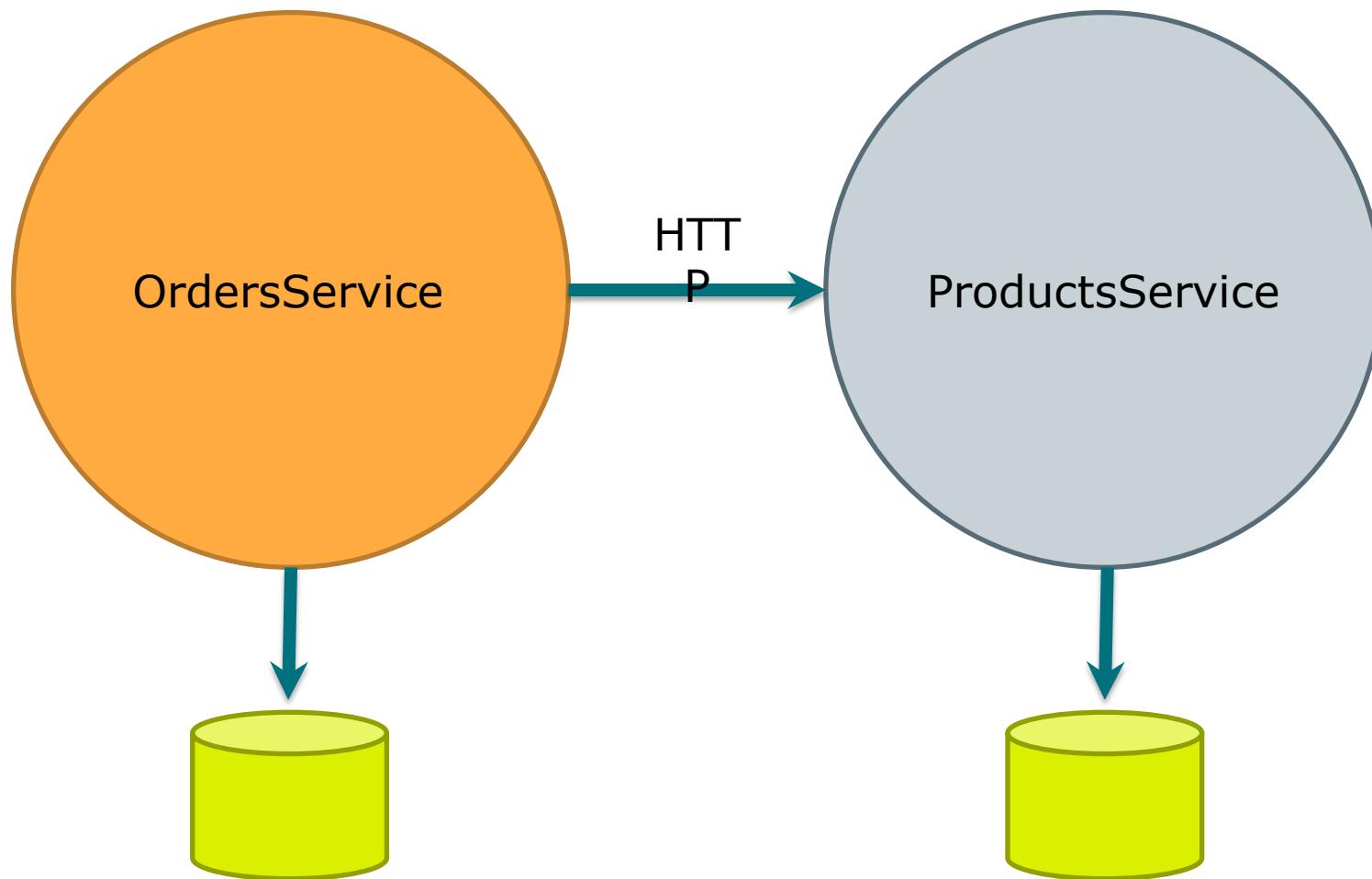


OrdersService

ProductsService

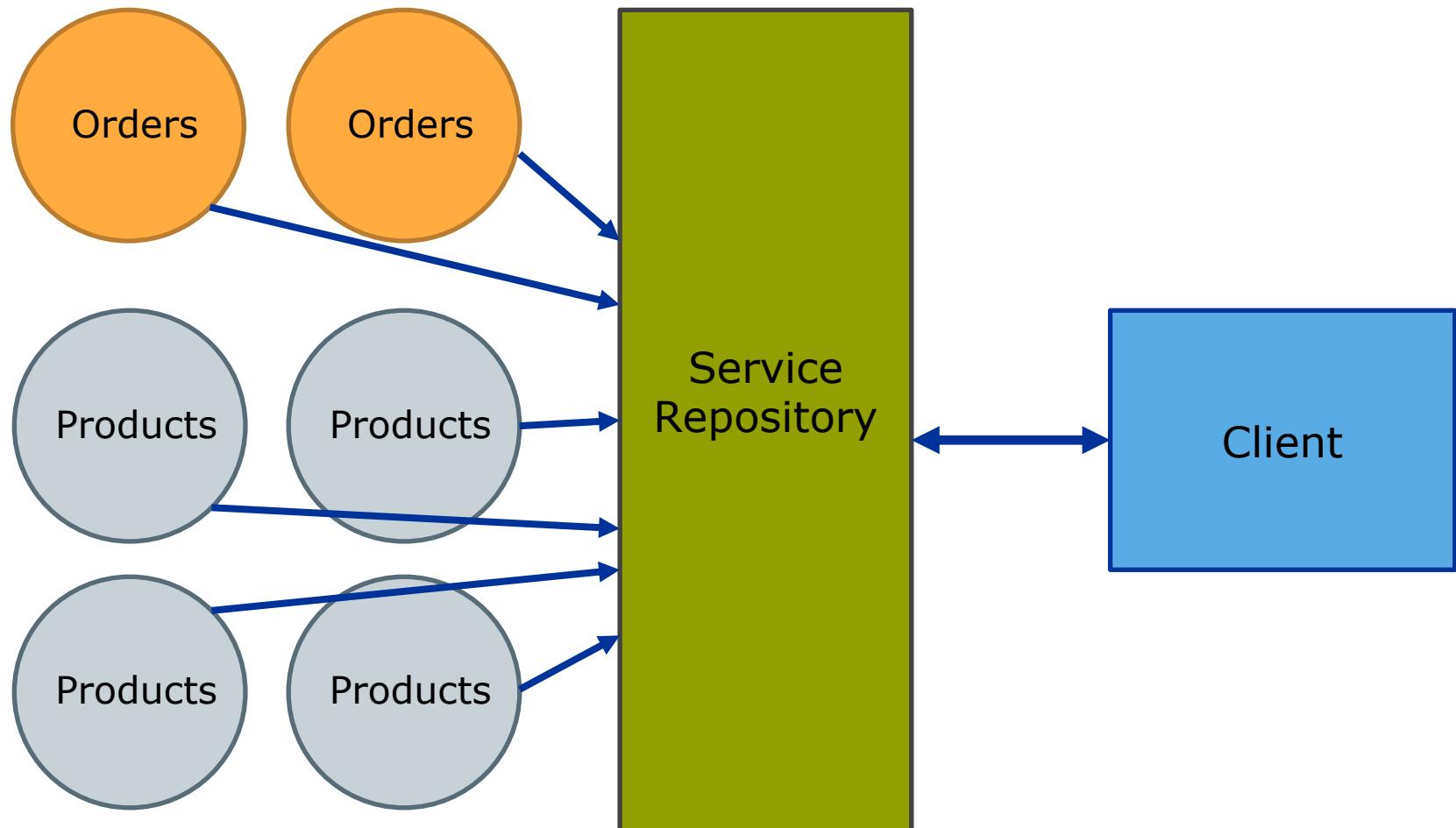
Arquitectura de microservicios

- Dividimos en dos servicios



Arquitectura de microservicios

- Los servicios pueden escalar individualmente



Arquitectura de microservicios

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

Patrones de distribución

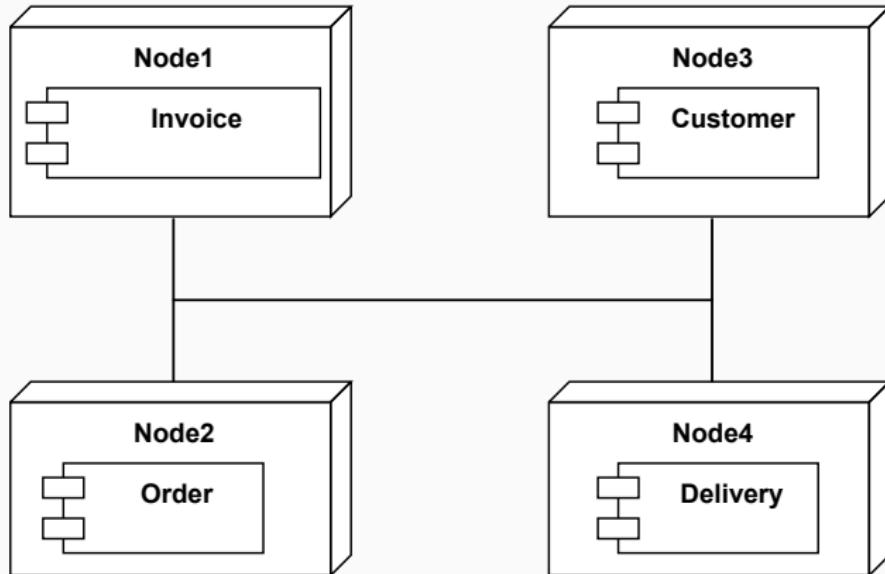
Patrones arquitecturales

Introducción

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

Introducción

¿Cuál es el problema de este diseño?



Aplicación distribuida con diferentes componentes en distintas máquinas
[Fowler, 2003]

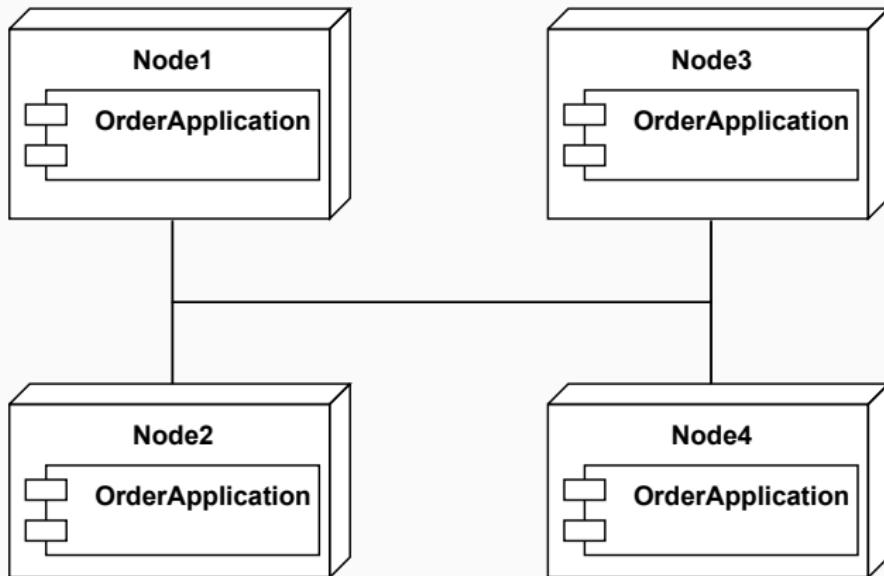
Introducción

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

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

Clustering

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



[Fowler, 2003]

Primera ley del diseño de objetos distribuidos:
“¡No distribuyas los objetos!”

Estrategias de distribución

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

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

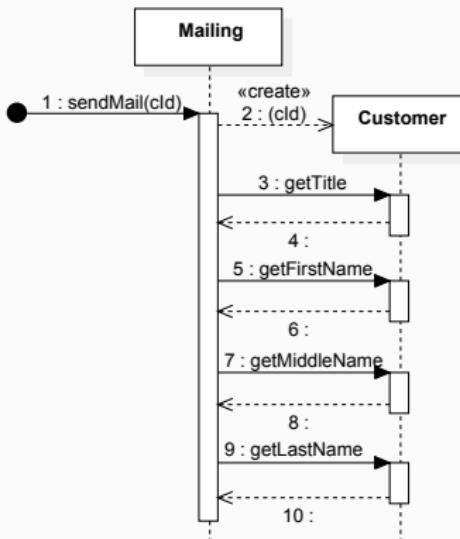
Estrategias de distribución

Qué hacer cuando los objetos deben estar distribuidos:

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

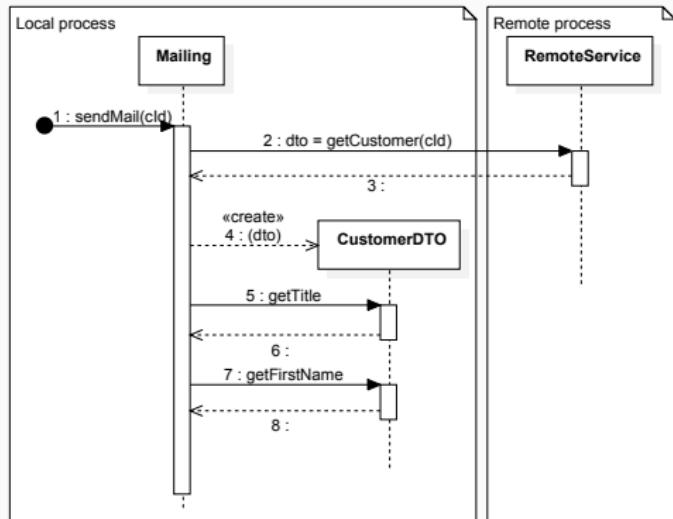
Estrategias de distribución

INTERFAZ DE GRANO FINO



El cliente solicita los datos uno a uno.

INTERFAZ DE GRANO GRUESO



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

Estrategias de distribución

Patrones de distribución involucrados:

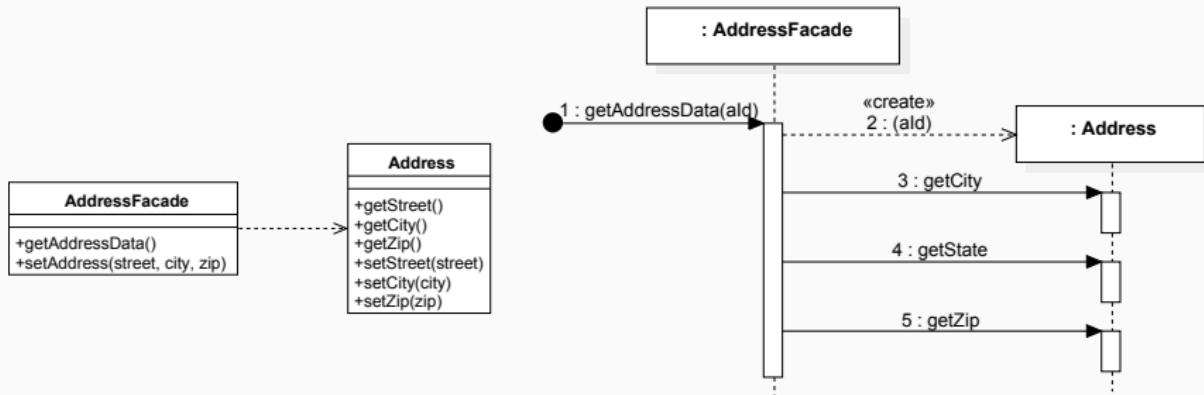
- Remote Facade
- Data Transfer Object

Patrón Remote Facade

Patrón Remote Facade

Remote Facade

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



[Fowler, 2003]

Patrón Remote Facade

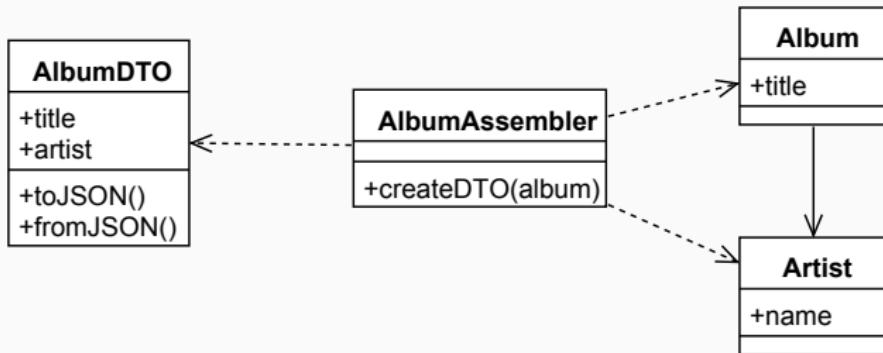
- Una **Fachada Remota** reemplaza todos los métodos get() y set() con un único método para acceder a todas las propiedades del objeto.
- Provee un interfaz de grano grueso sin modificar los objetos de dominio.
- **¡Las Fachadas Remotas no contienen lógica de negocio!**
- Se puede usar una única Fachada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

Patrón Data Transfer Object

Patrón Data Transfer Object

Data Transfer Object

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



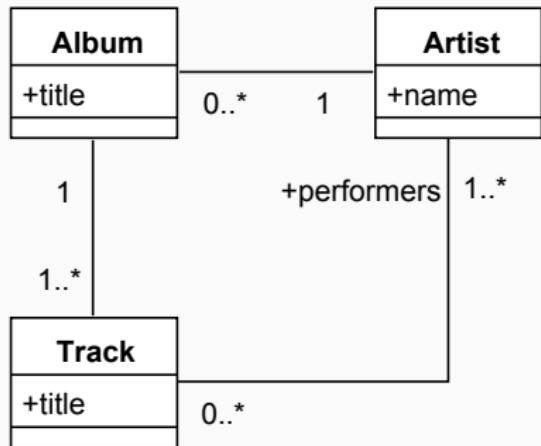
[Fowler, 2003]

Patrón Data Transfer Object

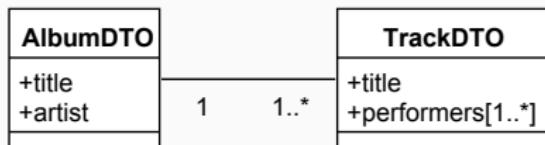
- **Contiene múltiples datos** para transportarlos en una única llamada remota.
- Debe ser **serializable** para poder transferir los datos sobre la conexión:
 - **Formato binario:** es más compacto pero más sensible a errores (p.ej. clientes no actualizados).
 - **Formato textual:** (p.ej. JSON) necesita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

Patrón Data Transfer Object

Los objetos DTO normalmente contienen datos de varios objetos de dominio



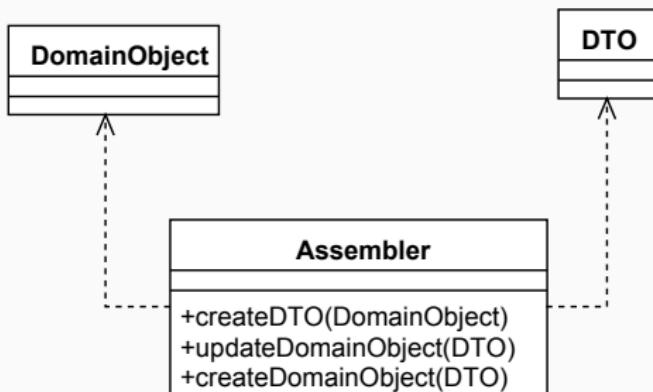
Modelo de dominio



Objetos DTO con información agregada

Patrón Data Transfer Object

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



¿Preguntas?

Bibliografía

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

Diseño detallado y patrones GRASP

Diseño de Sistemas Software
Curso 2020/2021

Carlos Pérez Sancho



Universitat d'Alacant
Universidad de Alicante

Departament de Llenguatges i Sistemes Informàtics
Departamento de Lenguajes y Sistemas Informáticos

Objetivos del tema

- Comprender el uso del diagrama de clases para reflejar conceptos a distintos niveles de abstracción
- Ser capaz de realizar un diseño software que sea respetuoso con los principios de asignación de responsabilidades de GRASP

Perspectiva de análisis
vs.
Perspectiva de diseño

Diseño detallado

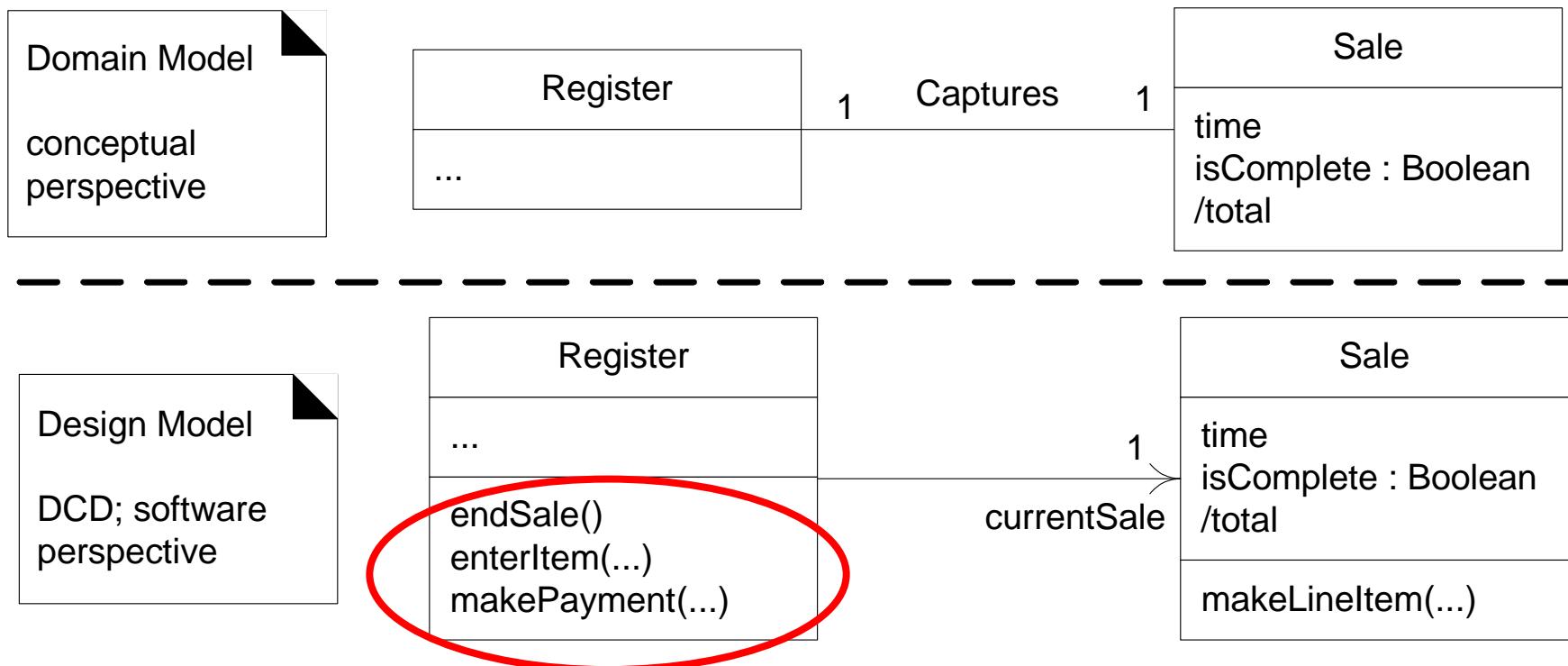
Introducción: usos del diagrama de clases

- UML incluye los diagramas de clases para ilustrar clases, interfaces y sus asociaciones.
- Éstos se utilizan para el modelado estático de objetos.
- Los diagramas de clases se pueden usar desde dos perspectivas:
 - Perspectiva conceptual → MODELO DE DOMINIO
 - Perspectiva de implementación → modelado de la CAPA DE DOMINIO

Introducción: diseño de objetos

- Los diagramas de diseño de clases se derivan del modelo de dominio, añadiendo los métodos y secuencias de mensajes necesarios para satisfacer los requisitos.
- Por lo tanto tenemos que:
 - Decidir qué operaciones hay que asignar a qué clases
 - Cómo los objetos deberían interactuar para dar respuesta a los casos de uso
- El artefacto más importante del flujo de trabajo de diseño es el Modelo de Diseño, que incluye el diagrama de clases software (no conceptuales) y diagramas de interacción.

Introducción: diseño de objetos



Objetivo: acercar el diseño a la implementación

Pasando del análisis al diseño

- Los diagramas de diseño de clases normalmente contienen nuevas clases que no están presentes en el modelo de dominio
 - Clases de utilidad: clases reutilizables
 - Librerías: clases del sistema
 - Interfaces: definiendo comportamientos abstractos
 - Clases de ayuda: aparecen por la descomposición de clases grandes

Introducción: Responsabilidades

- UML define una responsabilidad como “un contrato u obligación de una clase”
- Las responsabilidades se asignan a las clases durante el diseño de objetos
 - Hacer:
 - Hacer algo él mismo (p.ej. crear un objeto, realizar un cálculo)
 - Iniciar la acción en otros objetos
 - Controlar y coordinar actividades en otros objetos
 - Saber o Conocer:
 - Conocer sus datos privados (encapsulados)
 - Conocer los objetos con los que se relaciona
 - Conocer las cosas que puede derivar o calcular

Patrones GRASP

Diseño detallado

GRASP: Introducción

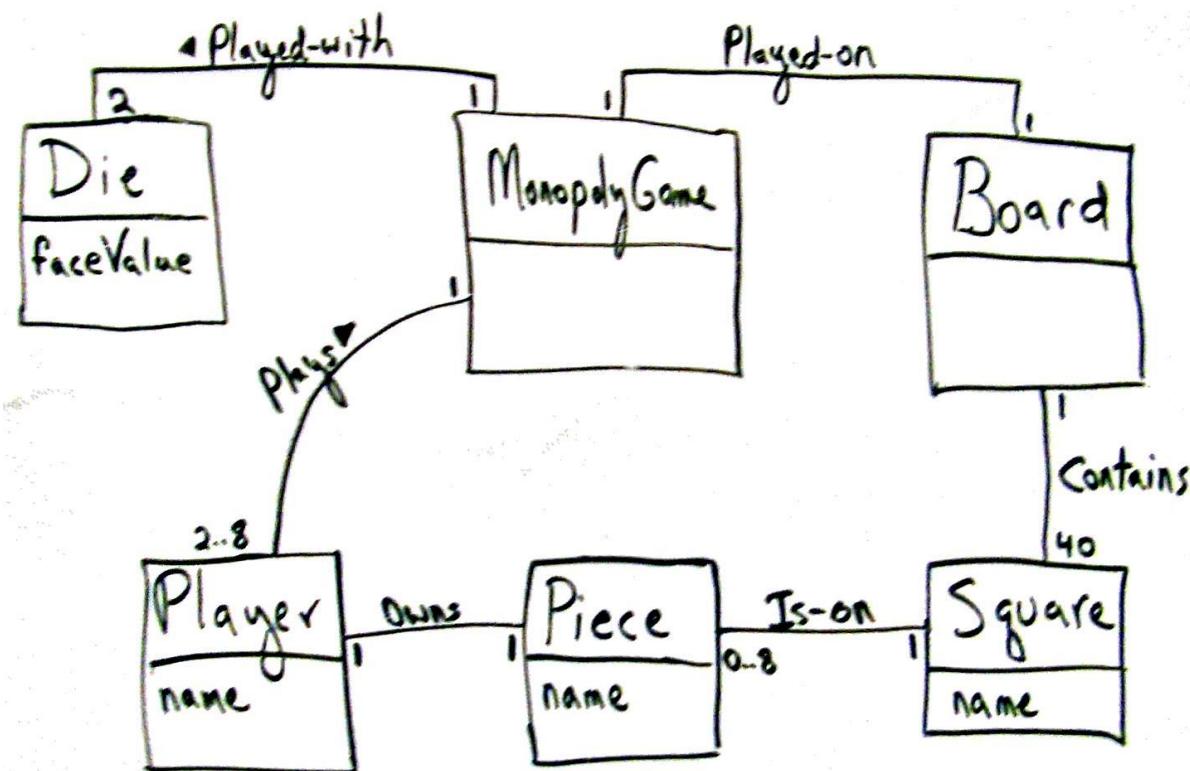
- GRASP es un acrónimo para General Responsibility Assignment Software Patterns
- Describen 9 principios fundamentales del diseño de objetos y de la asignación de responsabilidades, expresado en términos de patrones
- Se dividen en dos grupos:

BÁSICOS	AVANZADOS
<ul style="list-style-type: none">• Creador• Experto (en Información)• Bajo Acoplamiento• Controlador• Alta Cohesión	<ul style="list-style-type: none">• Polimorfismo• Fabricación Pura• Indirección• Protección de Variaciones

GRASP: introducción

- Para ilustrar los patrones vamos a suponer que queremos modelar un monopoly
- Dibujad su modelo de dominio (perspectiva conceptual)
 - Nota: asume dos dados
 - Nota: comienza modelando el tablero, las casillas, y el acto de hacer una tirada y mover las piezas por parte de un jugador

GRASP: introducción



Protección de variaciones

Patrones GRASP

GRASP: Protección de variaciones

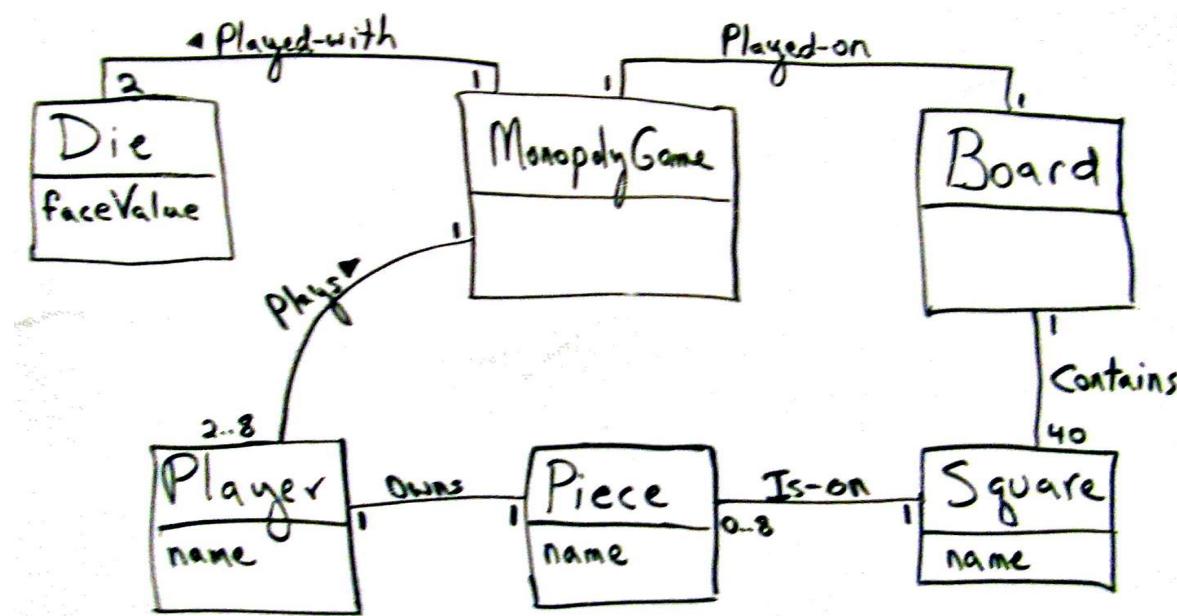
- Problema:
 - ¿Cómo diseñar objetos, subsistemas y sistemas de tal manera que las variaciones o inestabilidad en estos elementos no tenga un impacto indeseable sobre otros elementos?
- Solución:
 - Identifique los puntos de variación o inestabilidad; asigne responsabilidades para crear una interfaz estable a su alrededor.
 - Nota: el término interfaz se usa en su sentido más amplio, refiriéndose a los métodos que expone una clase; no implica el uso de interfaces de lenguajes de programación.

Bajo acoplamiento

Patrones GRASP

GRASP: Bajo acoplamiento

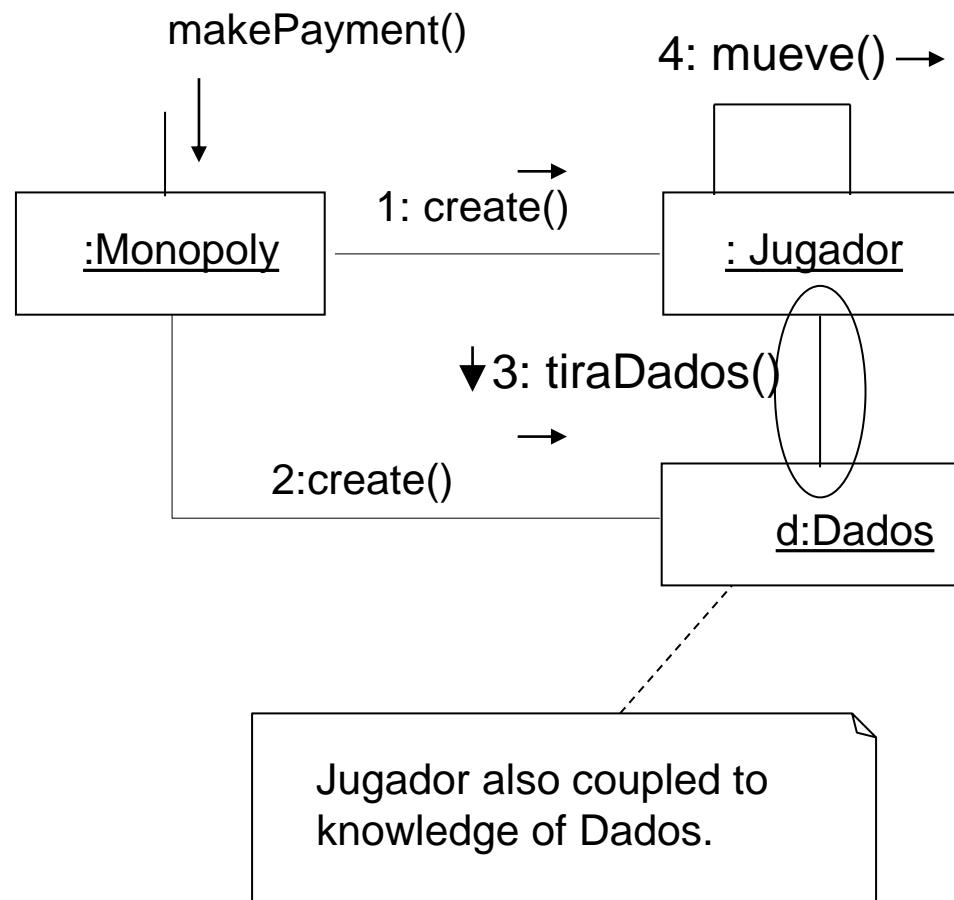
- Asume que necesitamos modelar el acto de tirar los dos dados de un jugador en el monopoly. ¿A quién asignamos la responsabilidad?



- Parece que Player es el mejor candidato pero, ¿qué problema tiene esta solución?

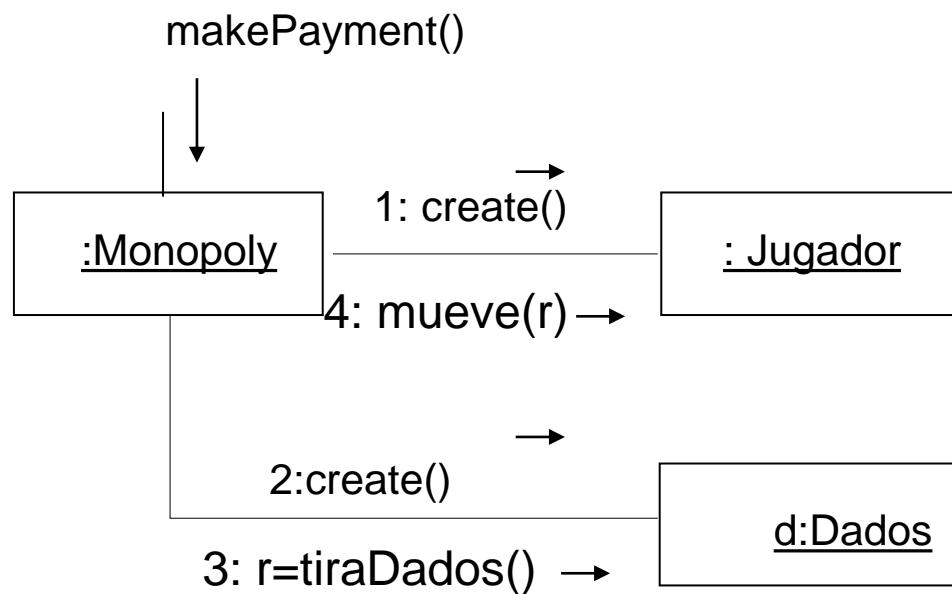
GRASP: Bajo acoplamiento

- Si el jugador tira los dados, es necesario acoplar al Jugador con conocimiento sobre los Dados (acoplamiento que antes no existía)



GRASP: Bajo acoplamiento

- Una solución alternativa es que sea el propio Juego el que tire los dados y envíe el valor que ha salido al Jugador.
- Se disminuye el acoplamiento entre Jugador y Dados y, desde el punto de vista del “acoplamiento” es un mejor diseño.



GRASP: Bajo acoplamiento

- Problema: ¿Cómo soportar una baja dependencia, un bajo impacto de cambios en el sistema y un mayor reuso?
- Solución: Asigna una responsabilidad de manera que el acoplamiento permanezca bajo

GRASP: Bajo acoplamiento

- Acoplamiento: medida que indica cómo de fuertemente un elemento está conectado a, tiene conocimiento de, o depende de otros elementos.
 - Una clase con un alto acoplamiento depende de muchas otras clases (librerías, herramientas, etc.)

GRASP: Bajo acoplamiento

- Problemas del alto acoplamiento:
 - Cambios en clases relacionadas fuerzan cambios en la clase afectada por el alto acoplamiento.
 - La clase afectada por el alto acoplamiento es más difícil de entender por sí sola: necesita entender otras clases.
 - La clase afectada por el alto acoplamiento es más difícil de reutilizar, porque requiere la presencia adicional de las clases de las que depende.

GRASP: Bajo acoplamiento

- Beneficios
 - Las clases con bajo acoplamiento normalmente...
 - No les afectan los cambios en otros componentes
 - Son sencillas de comprender de forma aislada
 - Es fácil reutilizarlas
- Contraindicaciones
 - El alto acoplamiento con elementos estables es raramente un problema, ya raramente habrá cambios que puedan afectar a estas clases, por lo que no merece la pena el esfuerzo de evitar este acoplamiento.

GRASP: Bajo acoplamiento

- Tipos de acoplamiento. El acoplamiento dentro de los diagramas de clases puede ocurrir por varios motivos:
 - Definición de Atributos: X tiene un atributo que se refiere a una instancia Y
 - Definición de Interfaces de Métodos: p.ej. un parámetro o una variable local de tipo Y se encuentra en un método de X
 - Definición de Subclases: X es una subclase de Y
 - Definición de Tipos: X implementa la interfaz Y

¿Cuál es mejor?

GRASP: Bajo acoplamiento

- Tipos de acoplamiento. El acoplamiento dentro de los diagramas de clases puede ocurrir por varios motivos:
 - Definición de Tipos: X implementa la interfaz Y
 - Definición de Interfaces de Métodos: p.ej. un parámetro o una variable local de tipo Y se encuentra en un método de X
 - Definición de Atributos: X tiene un atributo que se refiere a una instancia Y
 - Definición de Subclases: X es una subclase de Y

Ordenados de menor a mayor acoplamiento

Alta cohesión

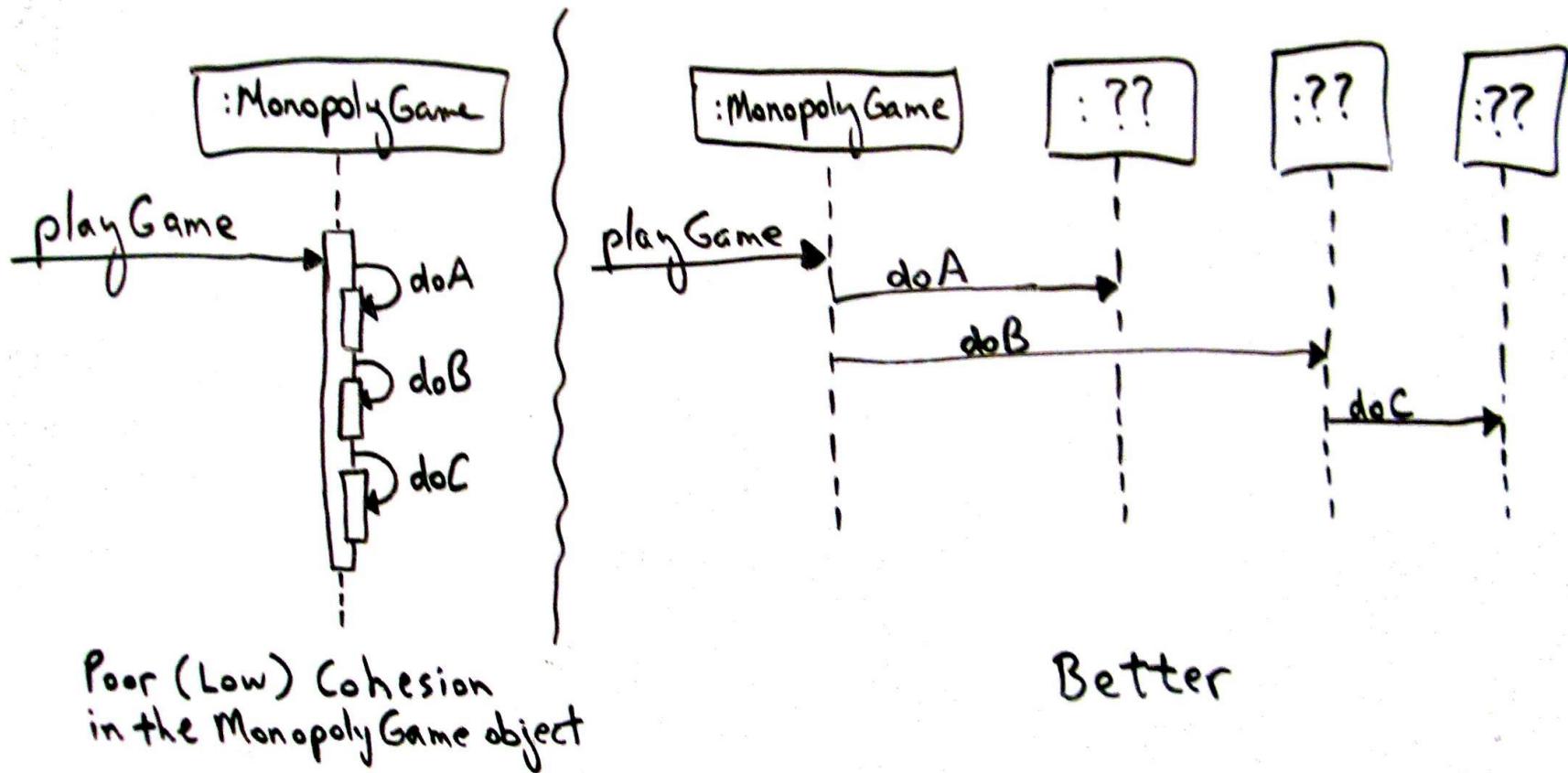
Patrones GRASP

GRASP: Alta cohesión

- ¿Cómo reescribiríais el siguiente código del monopoly?

```
Monopoly::PlayGame () {  
    turno = random(numeroJugadores);  
    Dado dados[2] = new Dado[2];  
    int puntuacion = 0;  
    for (i=0; i<2; i++)  
        puntuacion += dados[i].tirarDado();  
    Casilla cAct = getCasillaActual(jugadores[turno]);  
    Casilla cNueva = cAct + puntuacion;  
    colocaEnCasilla(jugadores[turno], cNueva)  
    ...  
    turno = (turno+1) mod numeroJugadores;  
    ...  
}
```

GRASP: Alta cohesión



GRASP: Alta cohesión

- Cohesión: medida de cómo se fuertemente se relacionan y focalizan las responsabilidades de un elemento. Los elementos pueden ser clases, subsistemas, etc.
- Una clase con baja cohesión hace muchas actividades poco relacionadas o realiza demasiado trabajo
- Problemas causados por un diseño con baja cohesión:
 - Difíciles de entender
 - Difíciles de usar
 - Difíciles de mantener
 - Delicados: fácilmente afectables por el cambio

GRASP: Alta cohesión

- Problema: ¿Cómo mantener la complejidad manejable?
- Solución: Asigna las responsabilidades de manera que la cohesión permanezca alta.
 - Clases con baja cohesión a menudo representan abstracciones demasiado elevadas, o han asumido responsabilidades que deberían haber sido asignadas a otros objetos.

GRASP: Alta cohesión

- Beneficios
 - Aumenta la claridad y comprensibilidad del diseño.
 - Se simplifican el mantenimiento y las mejoras.
- Contraindicaciones: en ocasiones se puede aceptar una menor cohesión:
 - Agrupar responsabilidades o código en una única clase o componente puede simplificar el mantenimiento por una persona (p.ej. Un experto en bases de datos)
 - Objetos distribuidos entre servidores. Debido al sobrecoste y las implicaciones en el rendimiento, a veces es deseable crear objetos menos cohesivos que provean un interfaz para numerosas operaciones.

GRASP: cohesión y acoplamiento

- Una mala cohesión normalmente implica un mal acoplamiento, y viceversa
- Una clase con demasiadas responsabilidades (baja cohesión), normalmente se relaciona con demasiadas clases (alto acoplamiento)

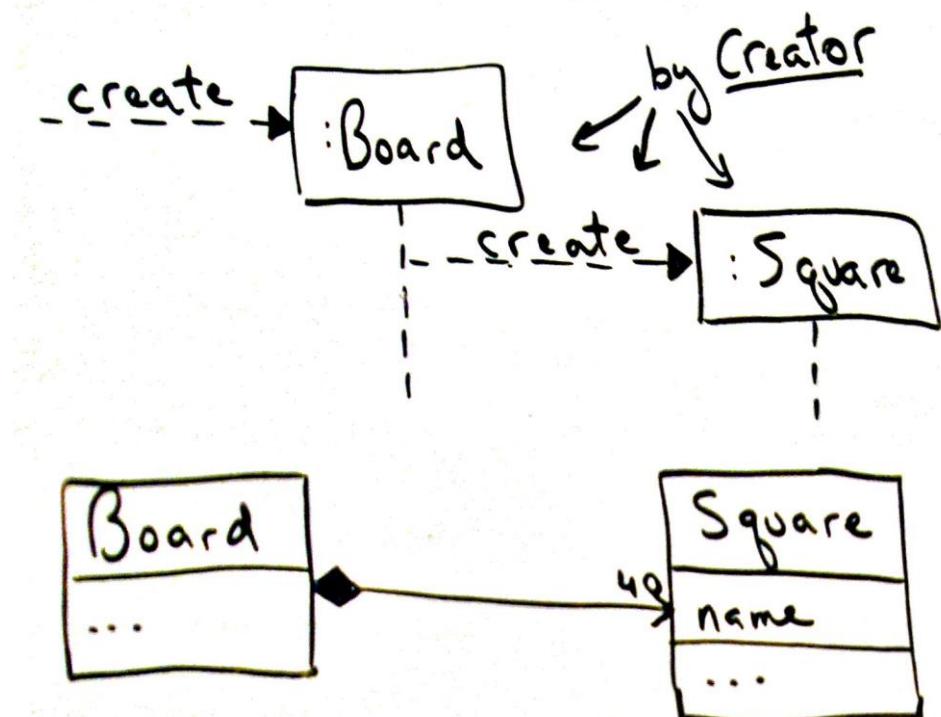
Creador

Patrones GRASP

GRASP: Creador

- En el Monopoly, ¿quién debería ser el responsable de crear Casillas?

- Puesto que un Tablero contiene casillas, por el patrón **Creator** éste debería ser el que las creara
- Además, una casilla sólo está en un tablero, por lo que la relación es de composición



GRASP: Creador

- Problema: ¿Quién debería ser el responsable de crear una nueva instancia de alguna clase?
- Solución: Asigna a la clase B la responsabilidad de crear una instancia de la clase A si una o más de las siguientes afirmaciones es cierta:
 - B agrega objetos de tipo A.
 - B contiene objetos de tipo A.
 - B graba objetos de tipo A.
 - B tiene datos inicializadores que serán pasados a A cuando sea necesario crear un objeto de tipo A (por tanto B es un Experto con respecto a la creación de A).

GRASP: Creador

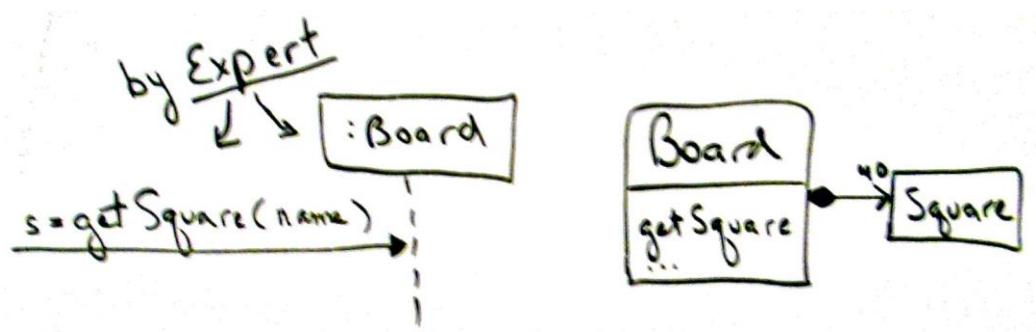
- Beneficios
 - Promueve el bajo acoplamiento, lo que implica menos dependencias (mejor mantenimiento) y mayores oportunidades de reutilización.
- Contraindicaciones
 - A veces la creación de un objeto requiere cierta complejidad, como por ejemplo el uso de instancias recicladas para aumentar el rendimiento o la creación condicional de una instancia perteneciente a una familia de clases. En estos casos es preferible delegar la creación a una clase auxiliar, mediante el uso de los patrones Concrete Factory o Abstract Factory.

Experto en información

Patrones GRASP

GRASP: Experto en información

- Necesitamos ser capaces de referenciar una casilla particular, dado su nombre (la calle que representa). ¿A quién le asignamos la responsabilidad?
- El patrón Information Expert nos indica que debemos asignar esa responsabilidad al objeto que conoce la información necesaria: los nombres de todas las casillas. Éste objeto es el objeto TABLERO



¿Por qué no poner el método `getSquare(name)` como
estático en la clase `Square`?

GRASP: Experto (en información)

- Problema: ¿cuál es el principio general de asignación de responsabilidades a objetos?
- Solución: Asigna cada responsabilidad al experto de información: la clase que tiene (la mayor parte de) la información necesaria para cubrir la responsabilidad.

GRASP: Experto (en información)

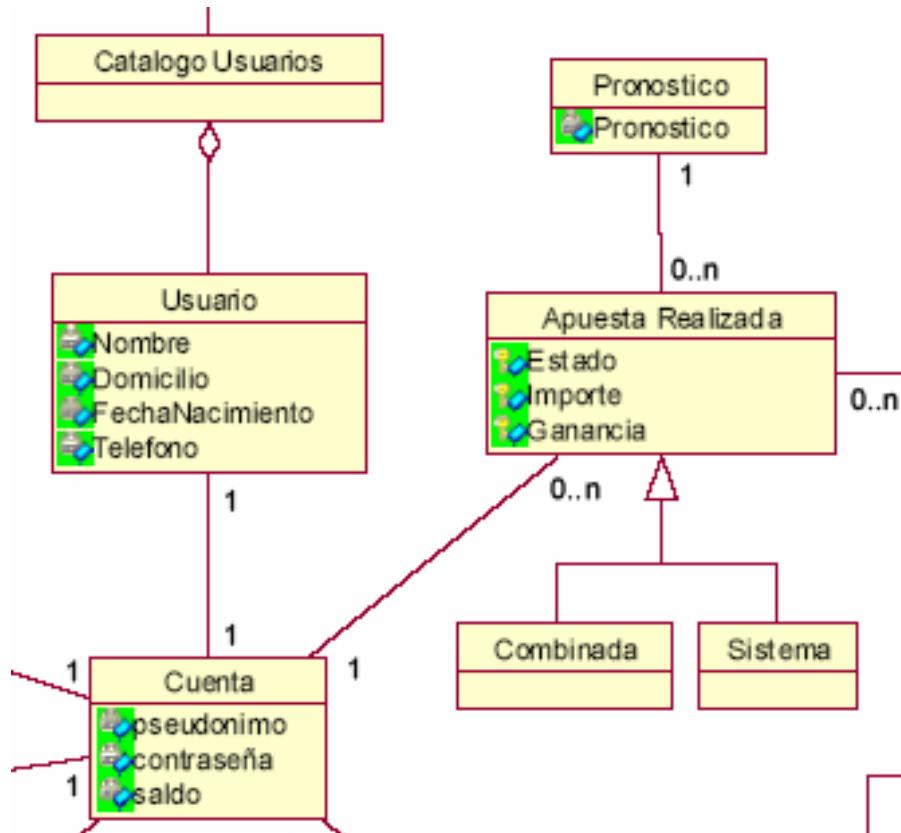
¡ATENCIÓN!

A veces hay que aplicar el Information Expert en cascada.

Ejemplo: Casa de apuestas

Suponed que queremos asignar la responsabilidad de calcular la ganancia/pérdida neta de un usuario en el siguiente diagrama.

Dibujad el diagrama de secuencia e indicad los cambios en el diagrama de clases.



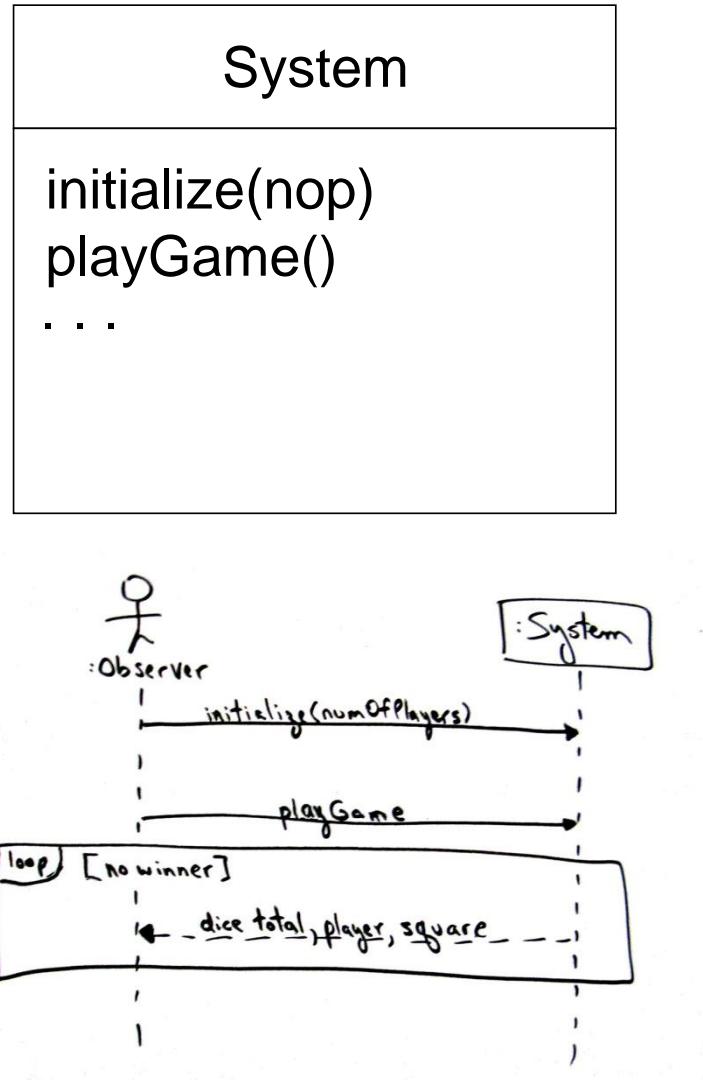
GRASP: Experto (en información)

- Beneficios
 - Se respeta la encapsulación de información, ya que los objetos usan su propia información para completar las tareas. Esto implica normalmente un bajo acoplamiento.
 - El comportamiento se distribuye entre las clases que contienen la información necesaria, consiguiendo clases más ligeras.
- Contraindicaciones
 - ¿Quién debería ser el responsable de almacenar una apuesta en la base de datos? Añadir esta responsabilidad a la clase Apuesta aumentaría sus responsabilidades añadiendo lógica de acceso a datos, disminuyendo así su cohesión.

Controlador

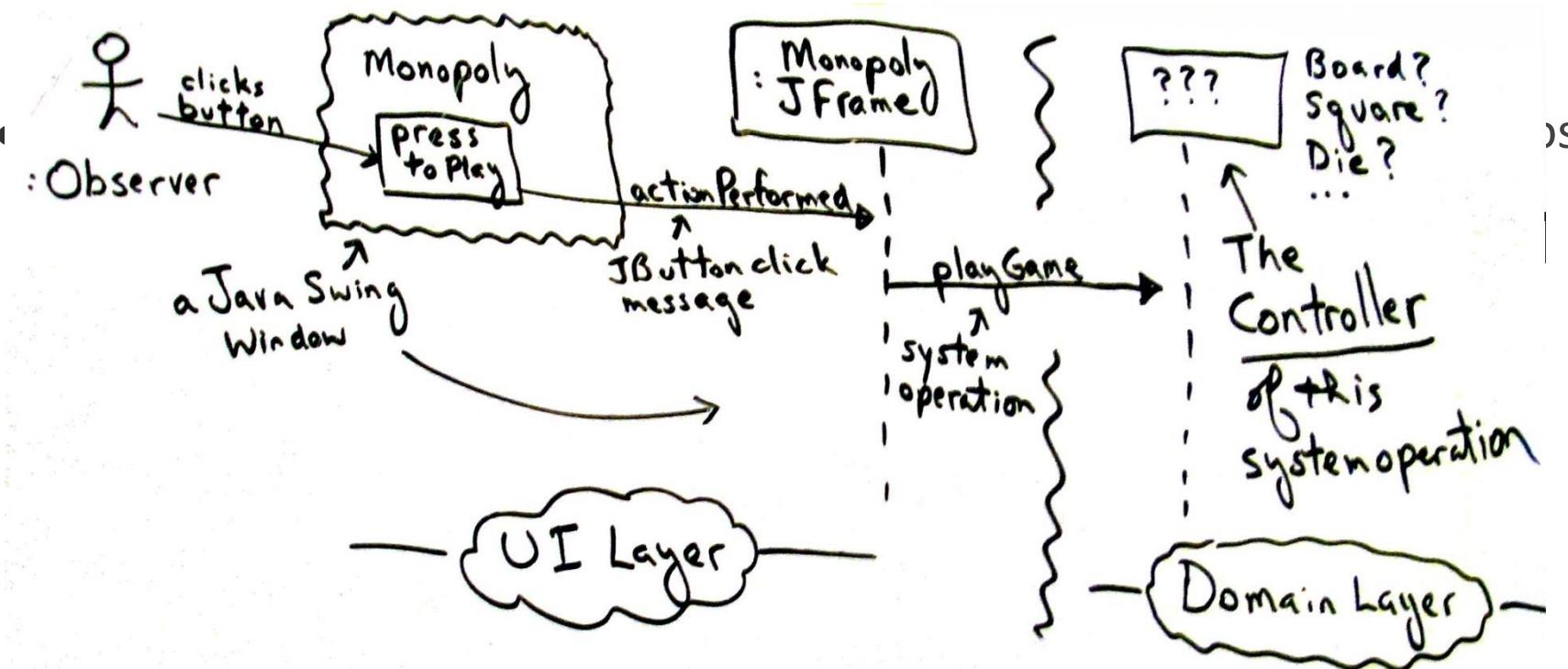
Patrones GRASP

GRASP: Controlador



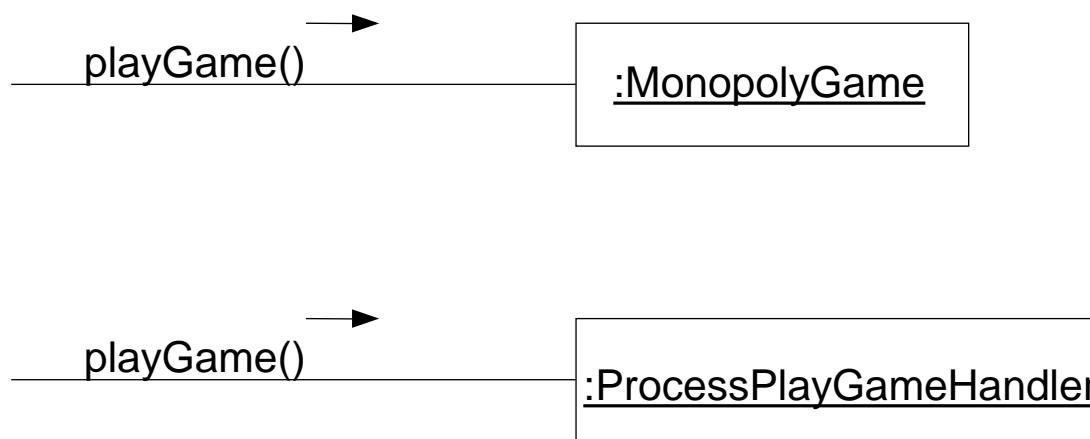
- En el Monopoly puede haber múltiples operaciones del sistema, que en un principio se podrían asignar a una clase **System**.
- Sin embargo esto no significa que finalmente deba existir una clase software **System** que satisfaga este requisito; es mejor asignar las responsabilidades a uno o más **Controllers**.

GRASP: Controlador

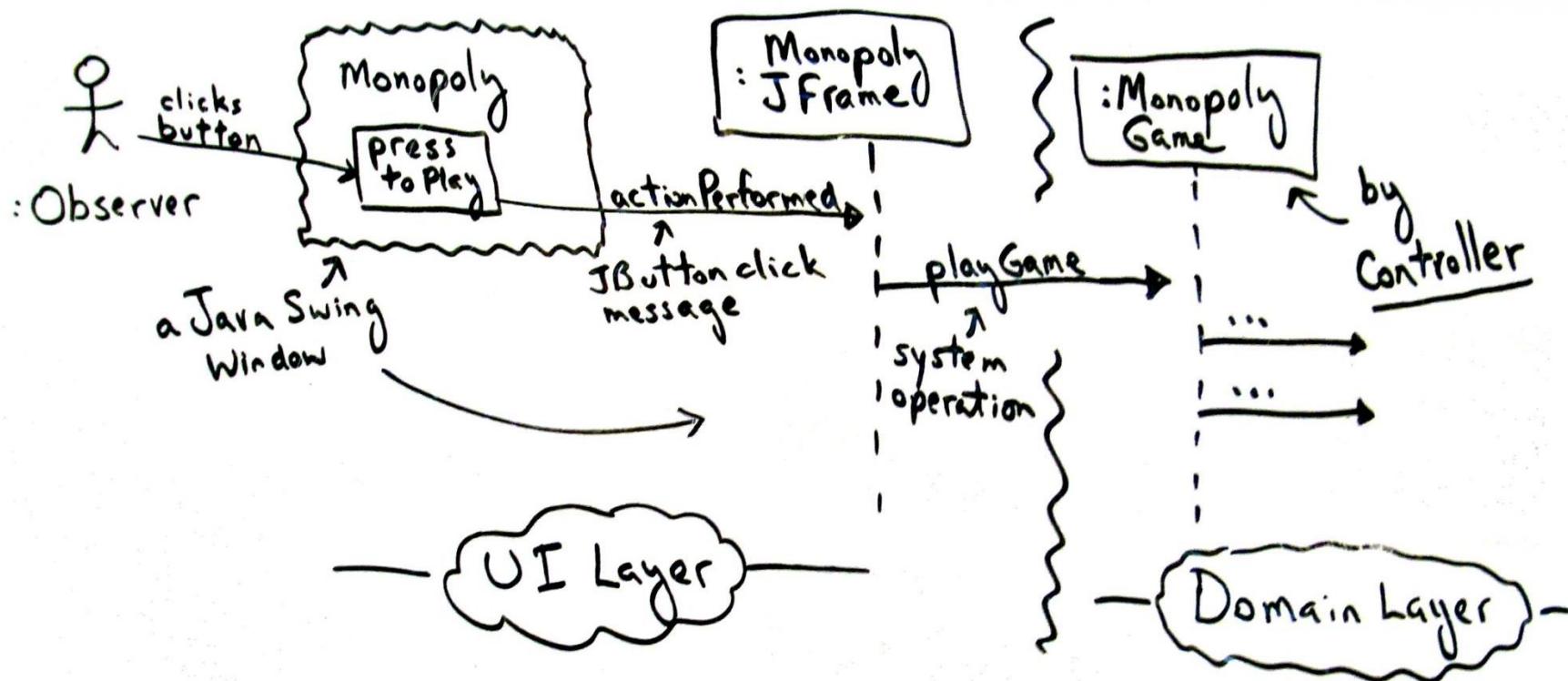


GRASP: Controlador

- Existen dos posibilidades:
 - MonopolyGame
 - ProcessInitializeHandler (esta solución requiere que haya otros controladores como ProcessPlayGameHandler, etc.).



GRASP: Controlador



GRASP: Controlador

- Problema: ¿Quién debería ser el responsable de manejar un evento de entrada al sistema?
 - ¿Quién es el primer objeto de la capa de dominio que recibe los mensajes de la interfaz?
- Solución: Asigna la responsabilidad de recibir o manejar un evento del sistema a una clase que represente una de estas dos opciones:
 - El sistema completo (Control ‘fachada’).
 - Un escenario de un Caso de Uso (estandariza nomenclatura: ControladorRealizarCompra, CoordinadorRealizarCompra, SesionRealizarCompra, ControladorSesionRealizarCompra.)

GRASP: Controlador

- El Controlador del que habla este patrón NO es el controlador del patrón MVC
- Normalmente ventanas, applets, etc. reciben eventos mediante sus propios controladores de interfaz, y los DELEGAN al tipo de controlador del que hablamos aquí.

Polimorfismo

Patrones GRASP

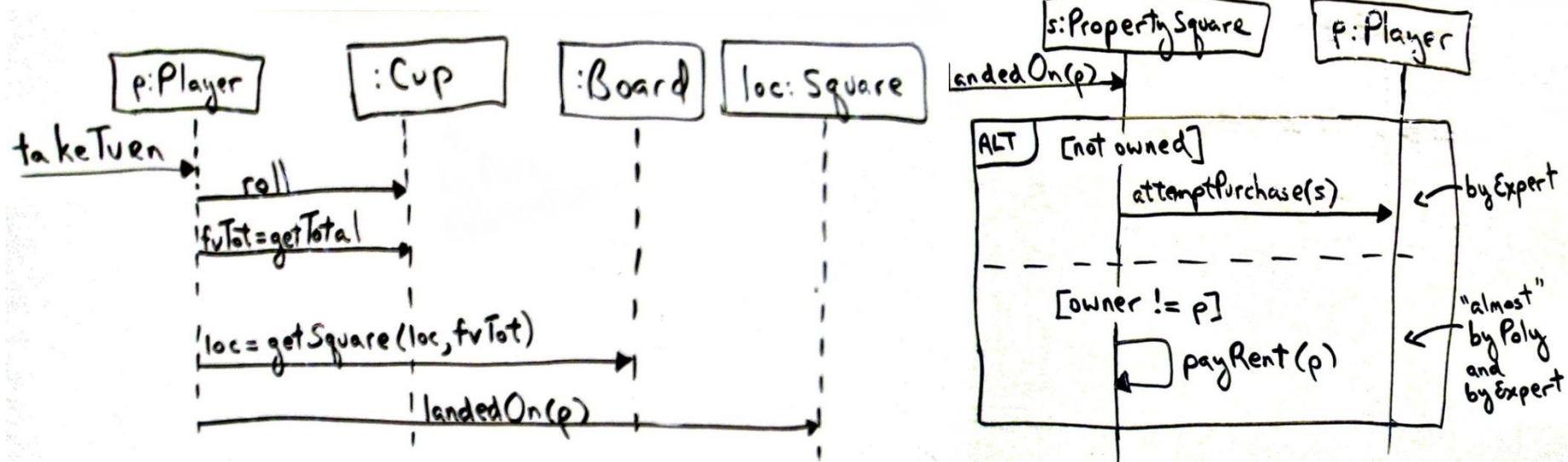
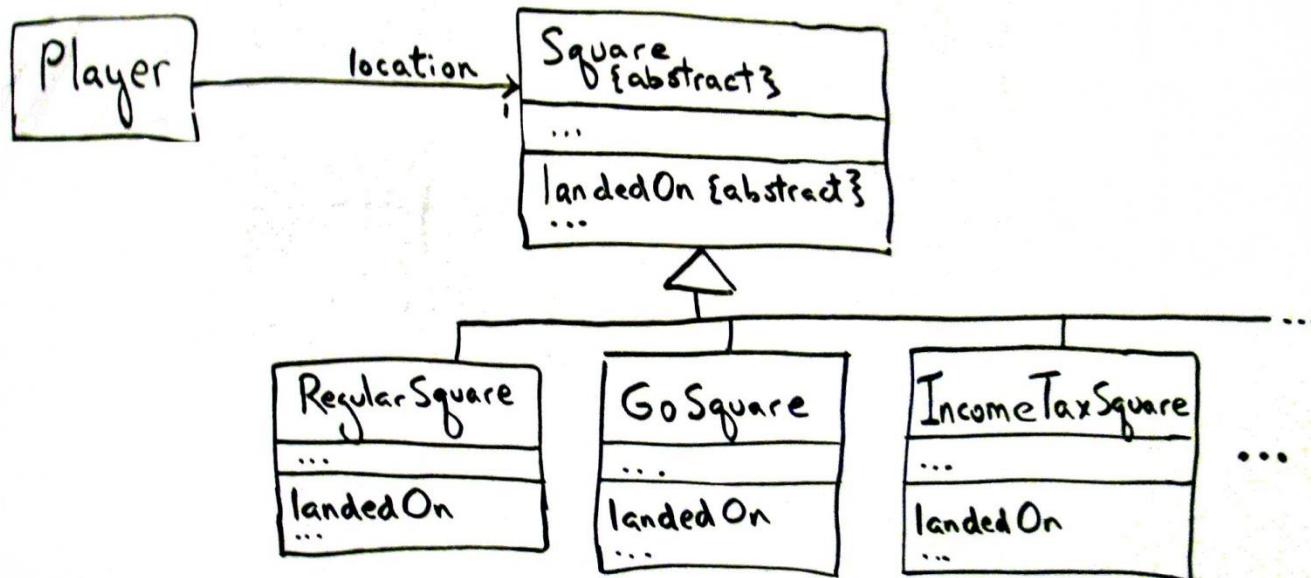
GRASP: Polimorfismo

- Vamos ahora a incluir el concepto de tipos de casillas en el Monopoly.
- Distintos tipos de casillas. En función del tipo de casilla, el comportamiento del método caerEn() varía:
 - Casillas de suerte: coger una carta de la suerte
 - Casillas de propiedades: comprar o pagar...
 - Casilla de Vaya a la Cárcel: ir a la cárcel
 - Casilla de Tasas: pagar tasas
 - ...
- ¿A quién asigno la responsabilidad caerEn()? ¿Cómo lo implemento?

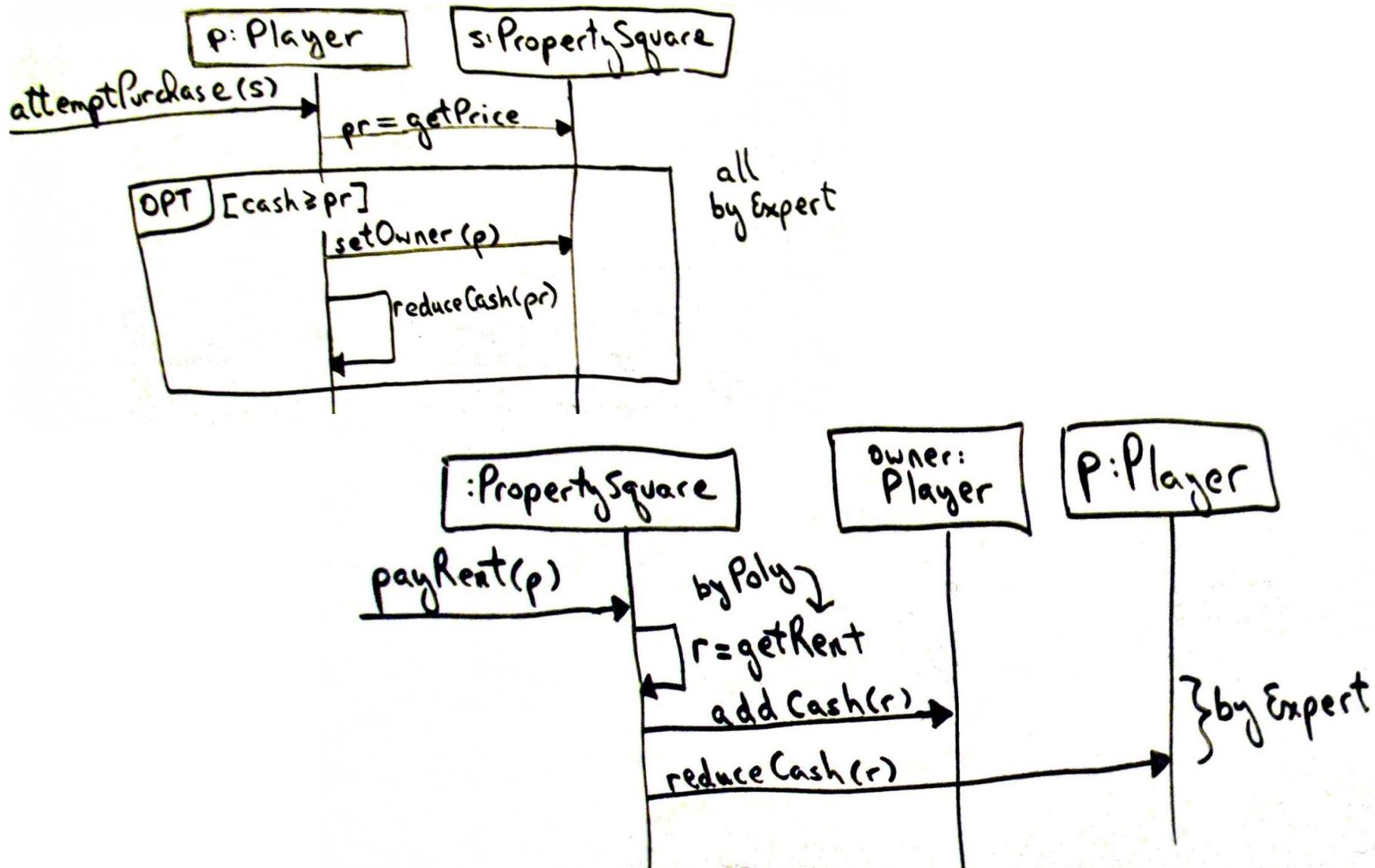
GRASP: Polimorfismo

- Problemas:
 - ¿Cómo manejar alternativas basadas en un tipo sin usar sentencias condicionales if-then o switch que requerirían modificación en el código?
- Solución:
 - Cuando alternativas o comportamientos relacionados varían por el tipo (clase), asigna la responsabilidad del comportamiento usando “operaciones polimórficas” a los tipos para los cuales el comportamiento varía.
 - Corolario: No preguntes por el tipo del objeto usando lógica condicional

GRASP: Polimorfismo



GRASP: Polimorfismo



GRASP: Polimorfismo

- Beneficios
 - Es fácil extender el sistema con nuevas variaciones.
 - Se pueden introducir nuevas implementaciones sin afectar a las clases cliente.
- Contraindicaciones
 - No es raro dedicar demasiado tiempo a la realización de diseños preparados para cambios poco probables, mediante el uso de herencia y polimorfismo.

Fabricación pura

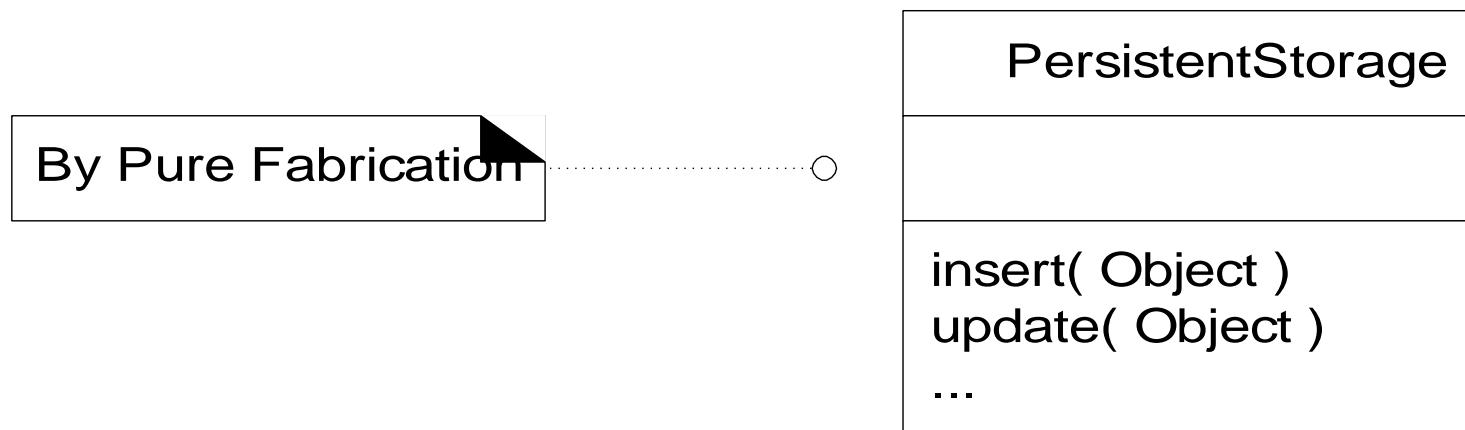
Patrones GRASP

GRASP: Fabricación pura

- Es necesario guardar instancias del Monopoly en una base de datos relacional. ¿Quién debería tener esa responsabilidad?
- Por Expert la clase Monopoly debería tener esta responsabilidad, sin embargo:
 - La tarea requiere un número importante de operaciones de base de datos, ninguna relacionada con el concepto de Monopoly, por lo que Monopoly resultaría incohesiva.
 - Monopoly quedaría acoplado con la interface de la base de datos (ej.- JDBC en Java, ODBC en Microsoft) por lo que el acoplamiento aumenta
 - Guardar objetos en una base de datos relacional es una tarea muy general para la cual se requiere que múltiples clases le den soporte. Colocar éstas en Monopoly sugiere pobre reuso o gran cantidad de duplicación en otras clases que hacen lo mismo.

GRASP: Fabricación pura

- Solución: Crear una clase (PersistentStorage) que sea responsable de guardar objetos en algún tipo de almacenamiento persistente (tal como una base de datos relacional).



GRASP: Fabricación pura

- Problemas resueltos:
 - La clase Monopoly continua bien definida, con alta cohesión y bajo acoplamiento.
 - La clase PersistentStorage es, en sí misma, relativamente cohesiva, tiene un único propósito de almacenar o insertar objetos en un medio de almacenamiento persistente.
 - La clase PersistentStorage es un objeto genérico y reusable.

GRASP: Fabricación pura

- Problema: ¿Qué objeto debería tener la responsabilidad, cuando no se desean violar los principios de “Alta Cohesión” y “Bajo Acoplamiento” o algún otro objetivo, pero las soluciones que sugiere Experto en información (por ejemplo) no son apropiadas?
- Solución: Asigne un conjunto “altamente cohesivo” de responsabilidades a una clase artificial conveniente que no represente un concepto del dominio del problema, algo producto de la “imaginación” para soportar “alta cohesión”, “bajo acoplamiento” y reuso.

GRASP: Fabricación pura

- En sentido amplio, los objetos pueden dividirse en dos grupos:
 - Aquellos diseñados por/mediante descomposición representacional.
(Ej.- Monopoly representa el concepto “partida”)
 - Aquellos diseñados por/mediante descomposición conductual. Este es el caso más común para objetos Fabricación pura.

Indirección

Patrones GRASP

GRASP: Indirección

- ¿Cómo podemos desacoplar el juego del hecho de que se juega con 2 dados?
- Solución: Cubilete
 - El objeto Cubilete es un ejemplo de indirección: un elemento que no existía en el juego real pero que introducimos para aislarlos del número de dados que usa el juego.
- ¿Cuándo hemos visto esto antes?

GRASP: Indirección

- Problema:
 - ¿Dónde asignar una responsabilidad para evitar acoplamiento directo entre dos o más cosas? ¿Cómo desacoplar objetos de tal manera que el bajo acoplamiento se soporte y el reuso potencial se mantenga alto?
- Solución:
 - Asignad la responsabilidad a un objeto intermedio que medie entre otros componentes o servicios, de tal manera que los objetos no estén directamente acoplados. El objeto intermedio crea una indirección entre los componentes.

¿Preguntas?

Bibliografía

- Cachero, C. Patrones GRASP. Apuntes de Ingeniería del Software I
- Larman, C. (2004). Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition. Addison Wesley Professional

[Leer en Safari Books Online](#)

Principios generales de diseño

Diseño de Sistemas Software
Curso 2020/2021

Carlos Pérez Sancho



Universitat d'Alacant
Universidad de Alicante

Departament de Llenguatges i Sistemes Informàtics
Departamento de Lenguajes y Sistemas Informáticos

Contenidos

1. Composición sobre herencia
2. Principios SOLID
3. Inyección de dependencias

Composición sobre herencia

Composición sobre herencia

La herencia supone un acoplamiento muy fuerte que puede ocasionar problemas:

- Las relaciones “**es un ...**” pueden dejar de cumplirse al incorporar nuevos cambios
- Heredar de una clase para adquirir un comportamiento nos impide heredar de otra
- Los cambios de comportamiento en tiempo de ejecución son difíciles
- **La herencia es difícil de implementar cuando los modelos persisten en una base de datos relacional**

Composición sobre herencia

Síntomas de que una herencia puede no ser adecuada:

- Numerosos métodos heredados que no se usan
- Demasiados métodos sobrescritos
- Demasiados niveles de herencia

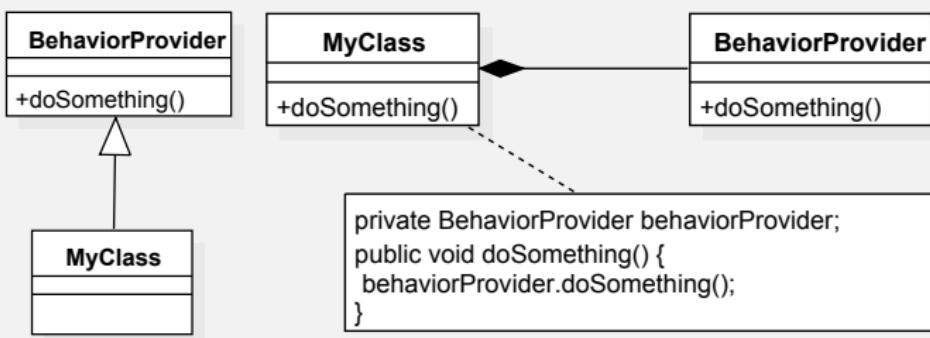
En general, la **herencia** sólo es imprescindible cuando necesitamos el **polimorfismo**.

Aún así, en algunos casos se puede evitar la herencia simplificando el problema, a costa de sacrificar otros aspectos del diseño.

Composición sobre herencia

Otra forma de “heredar” un comportamiento es usando la composición, **delegando** la implementación de la responsabilidad a otra clase.

Si estamos diseñando el modelo de dominio, la nueva clase no tiene por qué persistir en base de datos.

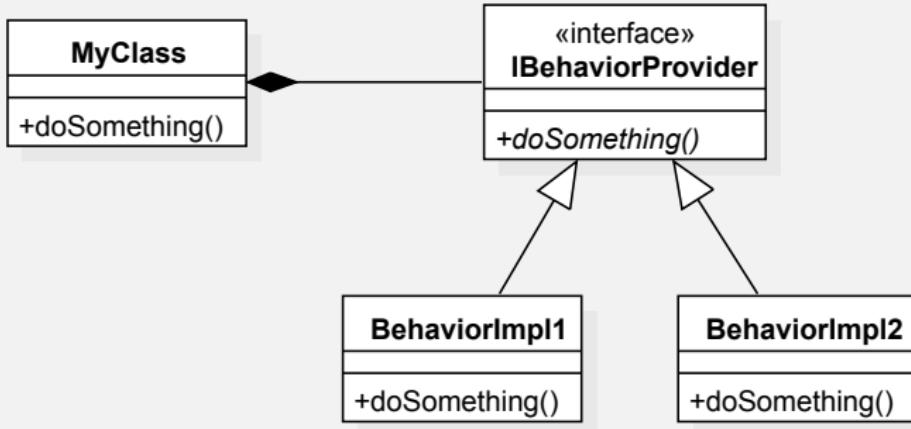


Herencia

Mismo resultado usando la composición

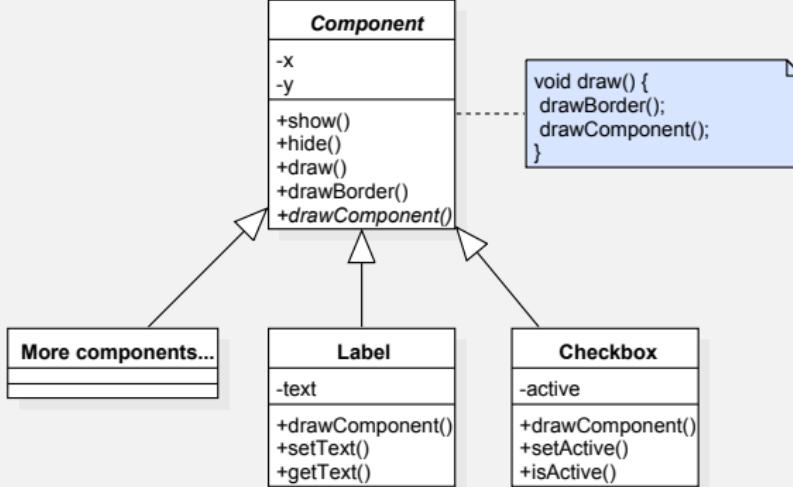
Composición sobre herencia

La composición ofrece mayor flexibilidad en tiempo de ejecución, ya que permite cambiar fácilmente el comportamiento de una clase.



Widgets 1.3.3

Supuesto: añadimos la funcionalidad `drawBorder()` a la clase base `Component`.



¿Problemas?

Widgets 1.3.3

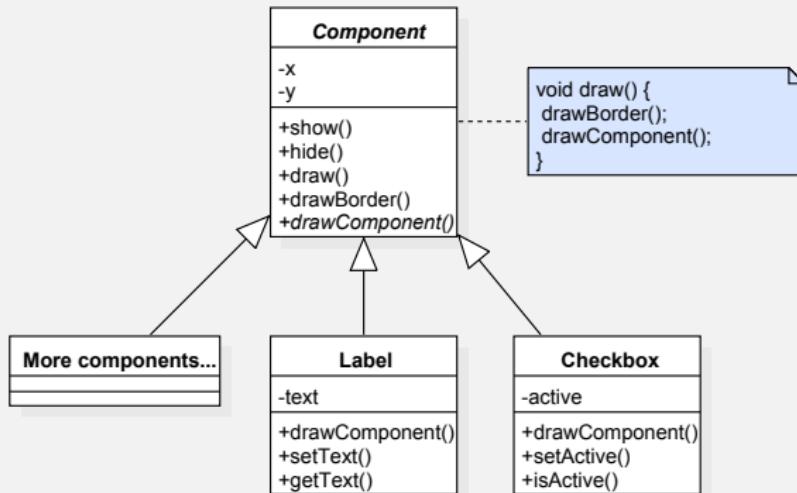
Problemas de este diseño

- Baja la cohesión de la clase Componente
- Dificulta el mantenimiento, p.ej. si queremos añadir nuevos tipos de borde

Veamos qué podría pasar si continuamos con este diseño...

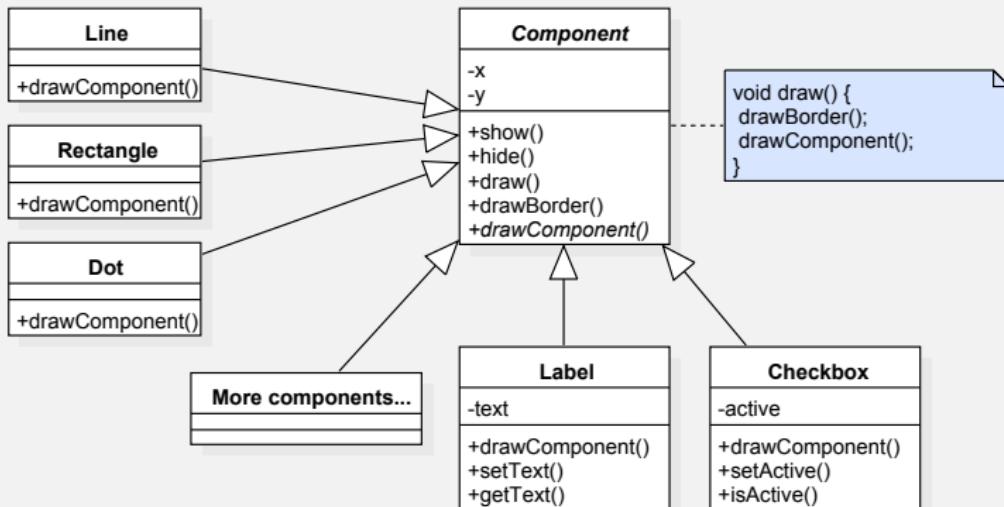
Widgets 1.3.4

Supuesto: para enriquecer aún más el interfaz, nos piden añadir líneas, puntos y algunas formas geométricas para aumentar las posibilidades de diseño gráfico.



¿Cómo modificamos el diseño?

Widgets 1.3.4



¿Qué problemas tiene este diseño?

Widgets 1.3.4

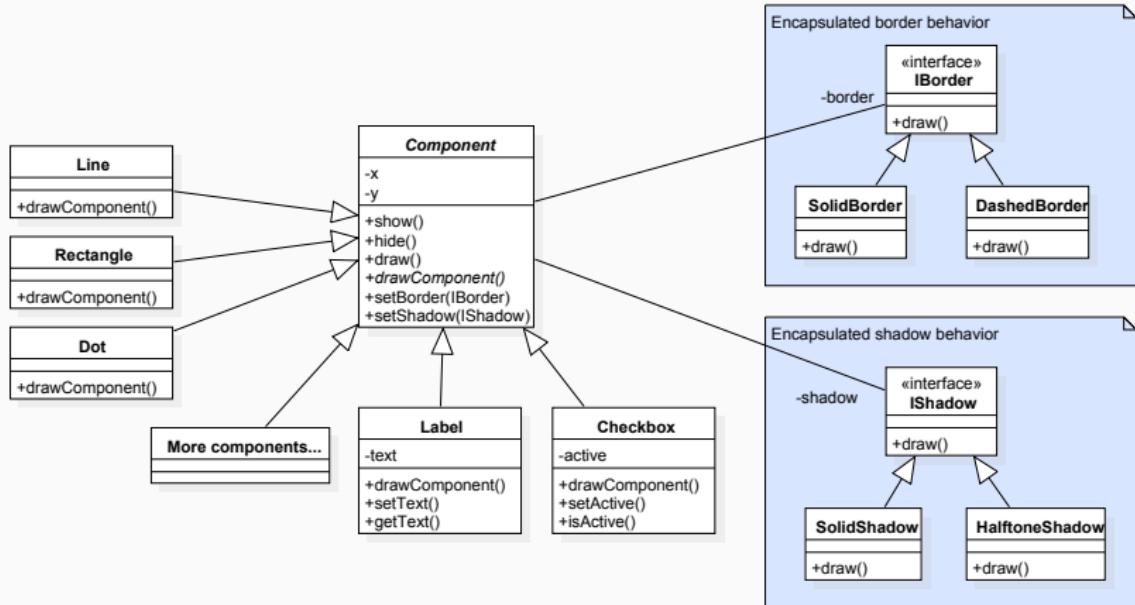
Problemas

- La herencia provoca un comportamiento no deseado: los nuevos componentes también tienen bordes.
- Es difícil de extender para añadir distintos tipos de border.

Solución

Podemos solucionar este problema usando la composición en lugar de la herencia.

Widgets 2.0



Ejemplo con Java Swing:

<https://docs.oracle.com/javase/tutorial/uiswing/components/border.html>

Principios SOLID

Principios SOLID

5 principios fundamentales de diseño [Martin, 2000]

- Principio de responsabilidad única (**S**ingle-responsibility)
- Principio abierto/cerrado (**O**pen/Closed)
- Principio de sustitución de Liskov (**L**iskov substitution)
- Principio de segregación de interfaces (**I**nterface segregation)
- Principio de inversión de dependencias (**D**evelopment dependency inversion)

Supuesto: implementación de la clase Fibonacci del ejercicio de la primera sesión de prácticas.

```
class Fibonacci {
    private ArrayList<Integer> series
        = new ArrayList<Integer>() {{ add(0); add(1); }};

    public int fibonacci(int n) {
        int len = series.size();
        if (n > len) {
            for (; len<n; len++)
                series.add(series.get(len-1) + series.get(len-2));
            return series.get(len-1);
        }
        else
            return series.get(n-1);
    }

    public void printSeries() {
        System.out.println(Arrays.toString(series.toArray()));
    }
}
```

¿Hay algún problema en este diseño?

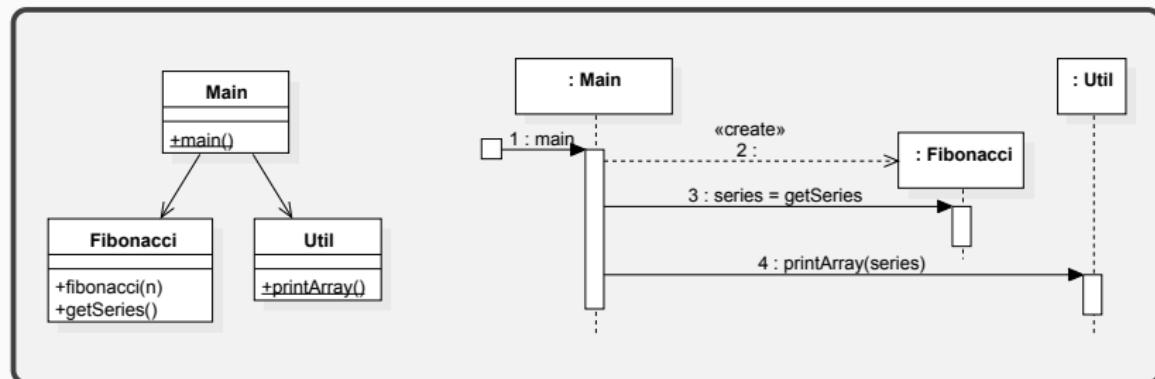
Principio de responsabilidad única

Principio de responsabilidad única

“Una clase sólo debería tener un motivo para cambiar.”

En el ejemplo, la clase Fibonacci no debería ser la encargada de imprimir la secuencia.

Possible solución:



Supuesto: queremos imprimir un array usando distintos formatos (p.ej. CSV y JSON).

```
class Util {  
    public void printArray(String format) {  
        if (format.equals("csv")) {  
            // Print as CSV  
        }  
        else if (format.equals("json")) {  
            // Print as JSON  
        }  
    }  
}
```

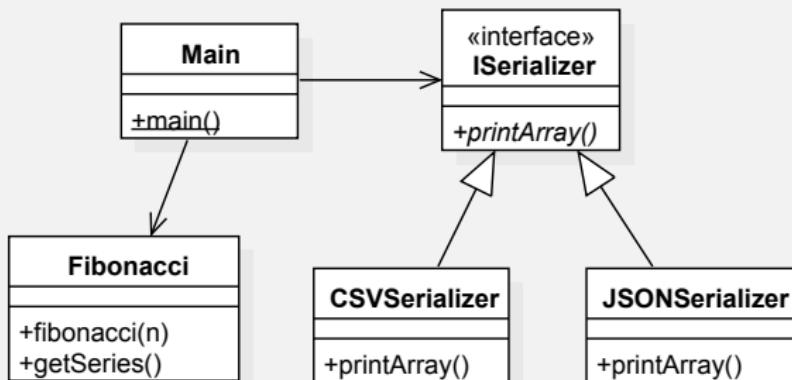
¿Hay algún problema en este diseño?

Principio abierto/cerrado

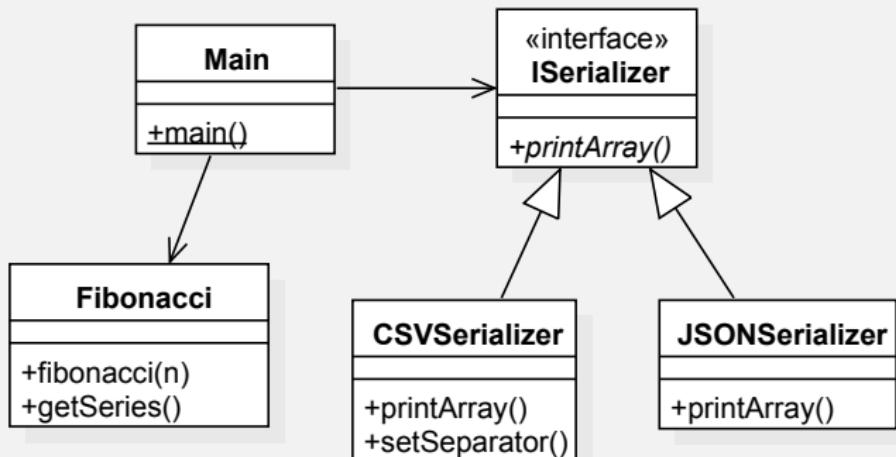
Principio abierto/cerrado

“Las entidades de software (clases, módulos, funciones, etc.) deberían estar abiertas para la extensión, pero cerradas a la modificación.”

La forma tradicional de conseguirlo es mediante la herencia o implementación de interfaces.



Supuesto: añadimos la posibilidad de especificar el tipo de separador para archivos CSV.



¿Hay algún problema en este diseño?

Principio de sustitución de Liskov

```
class Main {  
    public static void main(String args[]) {  
        String format = args[1];  
        ISerializer out = null;  
        if (format.equals("csv")) {  
            out = new CSVSerializer();  
            ((CSVSerializer) out).setSeparator(":");  
        }  
        else if (format.equals("json"))  
            out = new JSONSerializer();  
  
        Fibonacci fib = new Fibonacci();  
        // Computes the 10 first elements in the series  
        fib.fibonacci(10);  
  
        // Prints the array  
        out.printArray(fib.getSeries());  
    }  
}
```

¿Cuál es el problema?

Principio de sustitución de Liskov

Principio de sustitución de Liskov

“Si S es un subtipo de T, entonces los objetos de tipo T en un programa deberían poder ser sustituidos por objetos de tipo S.”

Al usar las subclases no deberíamos tener que conocer los detalles específicos de cada una de ellas, ni usarlas de forma distinta.

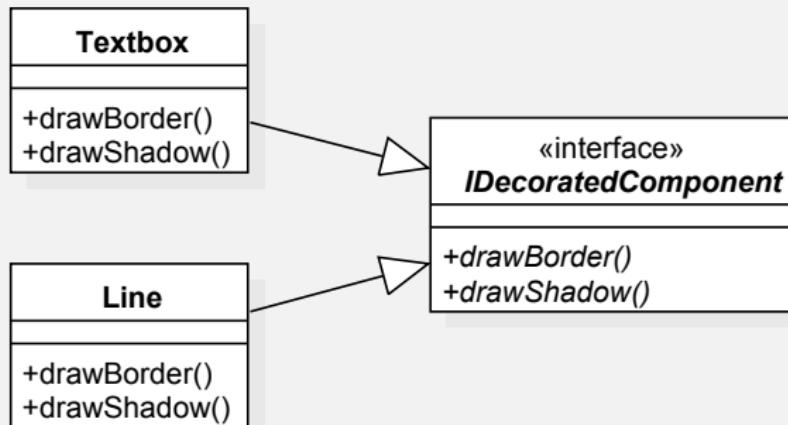
Principio de sustitución de Liskov

```
class Main {
    public static void main(String args[]) {
        String format = args[1];
        ISerializer out = null;
        if (format.equals("csv"))
            out = new CSVSerializer(":");
        else if (format.equals("json"))
            out = new JSONSerializer();

        Fibonacci fib = new Fibonacci();
        // Computes the 10 first elements in the series
        fib.fibonacci(10);

        // Prints the array
        out.printArray(fib.getSeries());
    }
}
```

Supuesto: agrupamos las funcionalidades para dibujar bordes y sombras en un único interfaz **IDecoratedComponent**.

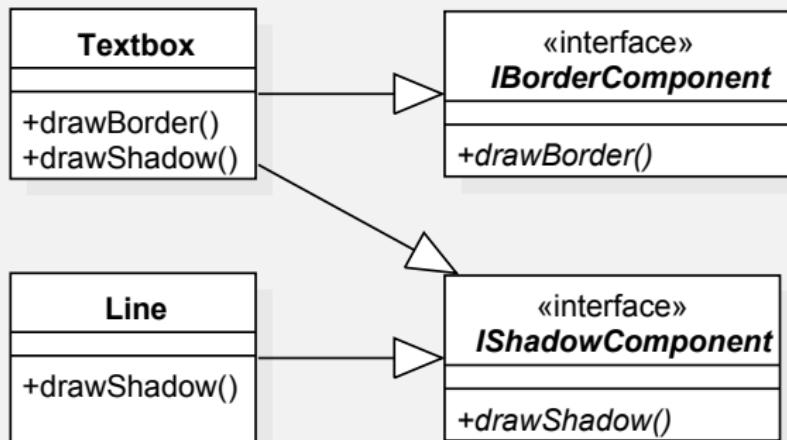


¿Hay algún problema en este diseño?

Principio de segregación de interfaces

Principio de segregación de interfaces

“Ninguna clase debería depender de métodos que no usa.”



Supuesto: asumamos que queremos que la clase Fibonacci mantenga el método printArray.

En su código instanciamos la clase necesaria para serializar dependiendo del formato de salida.

```
class Fibonacci {  
    public void printArray(String format) {  
        ISerializer out = null;  
        if (format.equals("csv"))  
            out = new CSVSerializer();  
        else if (format.equals("json"))  
            out = new JSONSerializer();  
  
        // Prints the array  
        out.printArray(series.toArray());  
    }  
}
```

¿Problemas?

Principio de inversión de dependencias



DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

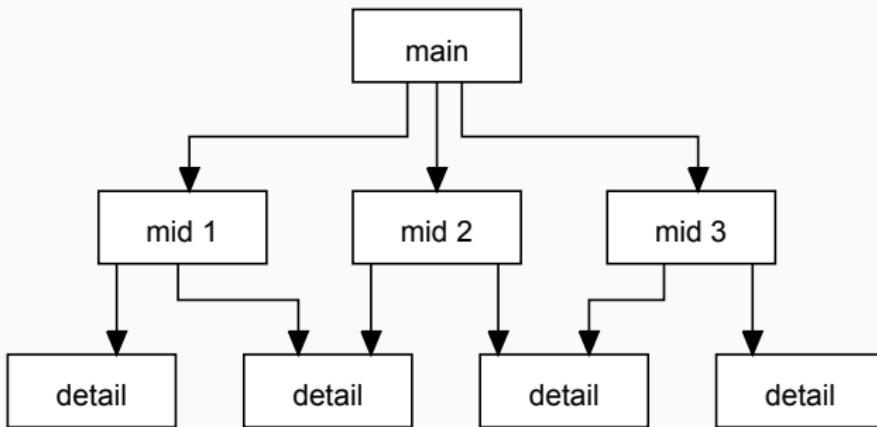
Principio de inversión de dependencias

Principio de inversión de dependencias

- “Módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.”
- “Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.”

Principio de inversión de dependencias

En sistemas procedurales los módulos de alto nivel dependen de módulos de más bajo nivel.

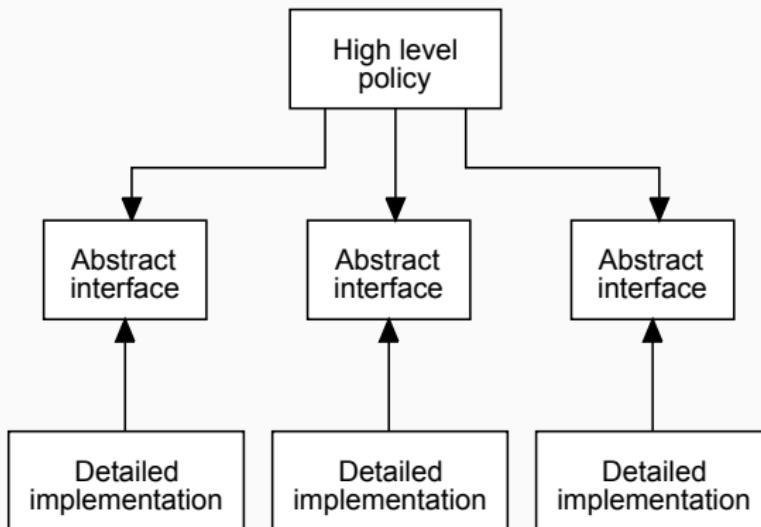


[Martin, 2000]

En sistemas orientados a objetos, los módulos de bajo nivel (implementaciones concretas) se consideran volátiles, ya que es más probable que cambien que las abstracciones.

Principio de inversión de dependencias

En la medida de lo posible se evitan estas dependencias invirtiéndolas, de manera que módulos de alto y bajo nivel dependen de abstracciones.



[Martin, 2000]

Principio de inversión de dependencias

Es preferible el acoplamiento con el interfaz, antes que con las clases que lo implementan.

Solución usando Factoría:

```
class SerializerFactory {  
    public static ISerializer createSerializer(String format) {  
        if (format.equals("csv"))  
            return new CSVSerializer();  
        else if (format.equals("json"))  
            return new JSONSerializer();  
    }  
}  
  
class Fibonacci {  
    public void printArray(String format) {  
        ISerializer out = SerializerFactory.createSerializer(format);  
        out.printArray(series.toArray());  
    }  
}
```

Principio de inversión de dependencias

Solución usando inyección de dependencias:

```
class Fibonacci {
    public void printArray(I Serializer serializer) {
        serializer.printArray(this.series.toArray());
    }
}

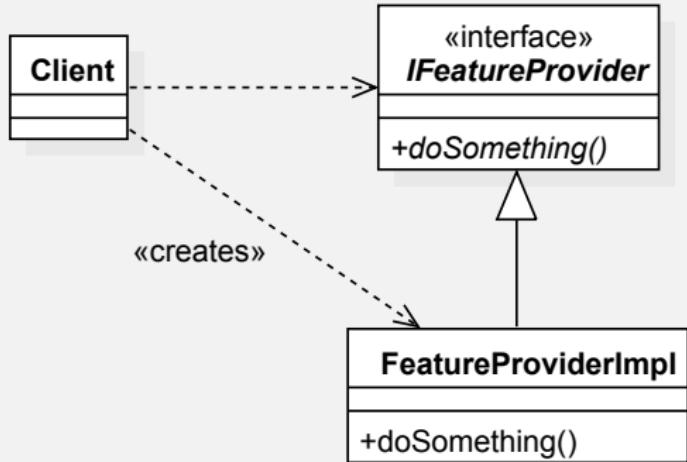
class Main {
    public static void main(String args[]) {
        I Serializer serializer = // Initialize depending on args
        Fibonacci fib = new Fibonacci();

        // do stuff...

        fib.printArray(serializer);
    }
}
```

Inyección de dependencias

Inyección de dependencias



Problema

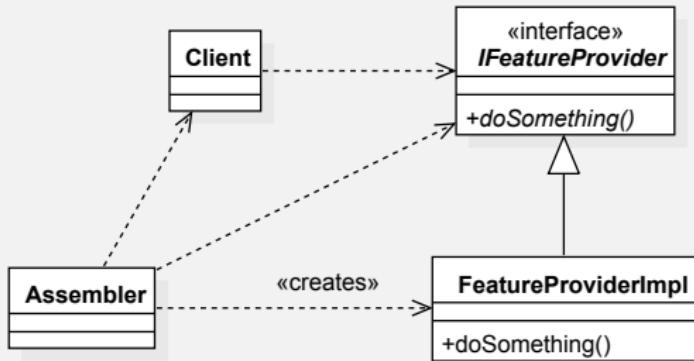
Cuando una clase A necesita una funcionalidad de otra clase B, y A crea una instancia de B, se establece una dependencia en tiempo de compilación que no se puede cambiar en tiempo de ejecución.

Inyección de dependencias

Solución

En lugar de crear la instancia directamente, la clase recibe una instancia que implementa el interfaz de la funcionalidad que necesita. Esta técnica se conoce como **Inyección de Dependencias**. [Fowler, 2004]

Inyección de dependencias



La inyección de dependencias es una forma de inversión de control:
un objeto distinto se encarga de crear la instancia de una
implementación concreta y proporcionársela al objeto que la usará.

Inyección de dependencias

Formas de inyección de dependencias:

- Inyección en el constructor
- Inyección con método *setter*
- Uso de un proveedor de servicios (*Service Locator*)

Inyección en el constructor

La clase cliente recibe una instancia del servicio en su constructor:

```
class Client {
    private IFeatureProvider provider = null;

    public Client(IFeatureProvider provider) {
        this.provider = provider;
    }
}
```

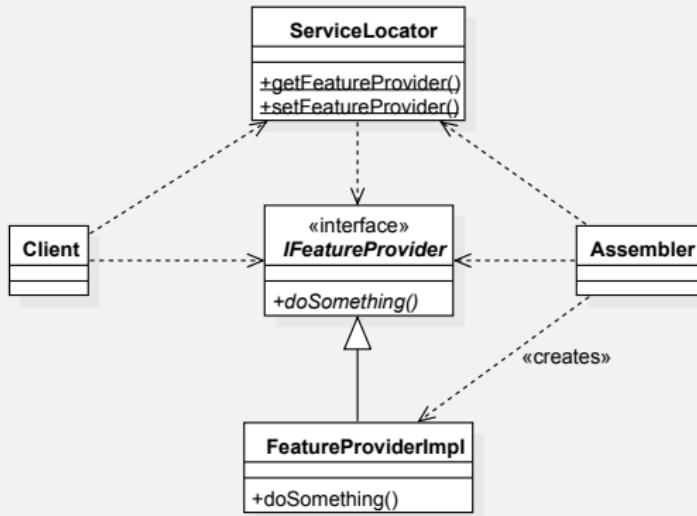
Inyección con método setter

La clase cliente recibe una instancia del servicio mediante un método setter:

```
class Client {
    private IFeatureProvider provider = null;

    public void setProvider(IFeatureProvider provider) {
        this.provider = provider;
    }
}
```

Uso de un proveedor de servicios



El objeto cliente obtiene los servicios de un *ServiceLocator*.

Uso de un proveedor de servicios

```
class Assembler {
    public void init() {
        IFeatureProvider provider = new FeatureProviderImpl();
        ServiceLocator.setFeatureProvider(provider);
    }
}

class Client {
    public void method() {
        IFeatureProvider provider =
            ServiceLocator.getFeatureProvider();
    }
}
```

Al seguir usando un objeto *Assembler* para crear las instancias tenemos más flexibilidad:

- Podemos probar el *ServiceLocator* independientemente
- Podemos inyectarle clases *mock* o *stub* para probar el cliente

Inyección de dependencias: comparativa

Inyección vs. *Service Locator*

- La inyección mediante constructor o método *setter* permite identificar mejor las dependencias, con un *Service Locator* hay que buscar sus llamadas en el código
- El *Service Locator* centraliza la inyección de dependencias en un único objeto, son más fáciles de gestionar

Inyección de dependencias: comparativa

Constructor vs. método *setter*

- La inyección mediante constructor permite crear objetos válidos desde la inicialización y proteger campos que no deben cambiar
- El método *setter* permite cambiar el comportamiento del objeto una vez inicializado
- **Lo importante es mantener la consistencia**
- Se pueden proporcionar los dos mecanismos, no son excluyentes

Inyección de dependencias en Laravel

Laravel realiza la inyección de dependencias automáticamente en algunos tipos de objetos, como p.ej. en los controladores, usando un contenedor de dependencias (*Service Container*):

<https://laravel.com/docs/6.x/container>

Inyección de dependencias en Laravel

El *Service Container* se encarga de resolver las dependencias en los constructores:

```
class UserController extends Controller
{
    protected $users;

    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }

    public function show($id)
    {
        $user = $this->users->find($id);

        return view('user.profile', ['user' => $user]);
    }
}
```

Inyección de dependencias en Laravel

Ejemplo usando interfaces con distintas implementaciones:

```
class TestController extends Controller
{
    protected $serializer;

    // Service Container will provide the $serializer instance
    public function __construct(I Serializer $serializer) {
        $this->serializer = $serializer;
    }
}
```

Código de ejemplo:

<https://github.com/cperezs/dss-dependency-injection>

Inyección de dependencias en Laravel

Para poder resolver las dependencias hay que registrarlas mediante *ServiceProviders*:

```
class SerializerServiceProvider extends ServiceProvider
{
    public function boot()
    {
        //
    }

    public function register()
    {
        $this->app->bind(
            'App\ISerializer',
            'App\CommaSerializer'
        );
    }
}
```

Inyección de dependencias en Laravel

Los *Service Providers* deben registrarse en el fichero `config/app.php`:

```
'providers' => [
    // ...
    App\Providers\SerializerServiceProvider::class,
],
```

<https://laravel.com/docs/6.x/providers>

Inyección de dependencias en Laravel

Al hacer pruebas automáticas se pueden injectar objetos Mock en lugar de los objetos reales:

<https://laravel.com/docs/6.x/mocking#mocking-objects>

¿Preguntas?

Referencias i

-  Fowler, M. (2004).
Inversion of Control Containers and the Dependency Injection pattern.
<https://martinfowler.com/articles/injection.html>.
[Online; accessed on January 2018].
-  Martin, R. C. (2000).
Design Principles and Design Patterns.
https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf.
[Online; accessed on January 2018].

1. Composite
2. Command
3. Singleton
4. Observer
5. Abstract Factory
6. State
7. Adapter
8. Decorator
9. Builder
10. Facade
11. Strategy
12. Factory method
13. Template method
14. Proxy

Ejercicios sobre patrones GOF

Diseño de Sistemas Software
Curso 2020/2021

Carlos Pérez Sancho



Universitat d'Alacant
Universidad de Alicante

Departament de Llenguatges i Sistemes Informàtics
Departamento de Lenguajes y Sistemas Informáticos

Ejercicio 1

Enunciado

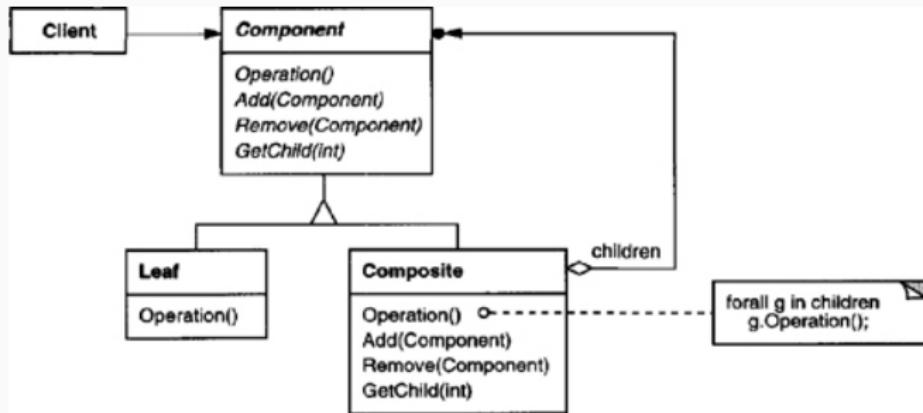
Decathlon nos ha pedido que le creamos un módulo para el manejo de la configuración de bicicletas a medida. Para ello, la tienda nos ha proporcionado la siguiente lista de partes, que podría irse especializando con nuevas **partes, simples o agregadas**: rueda, horquilla, eje, radio, tuercas del radio de las ruedas, cámara, tubo, llanta, armazón, manillar, ..., cada una con su precio. En un principio supondremos que Decathlon trabaja con un solo modelo de cada parte. Decathlon quiere que su software pueda llamar a nuestro módulo sin tener que preocuparse de si está tratando con una bicicleta completa o con partes aisladas.

De momento nos ha pedido un prototipo donde la única funcionalidad necesaria es `getPrecio()`. Ese precio se compone de un precio de mano de obra (sólo aplicable si el cliente pide una pieza que tiene que ser montada) y un precio base por cada una de las piezas. Plantea un diseño que resuelva este problema.

Análisis del problema

La complejidad del problema consiste en representar dos tipos de piezas (**simples y compuestas**), que deben comportarse de igual manera ante el cliente (**comparten un interfaz común**).

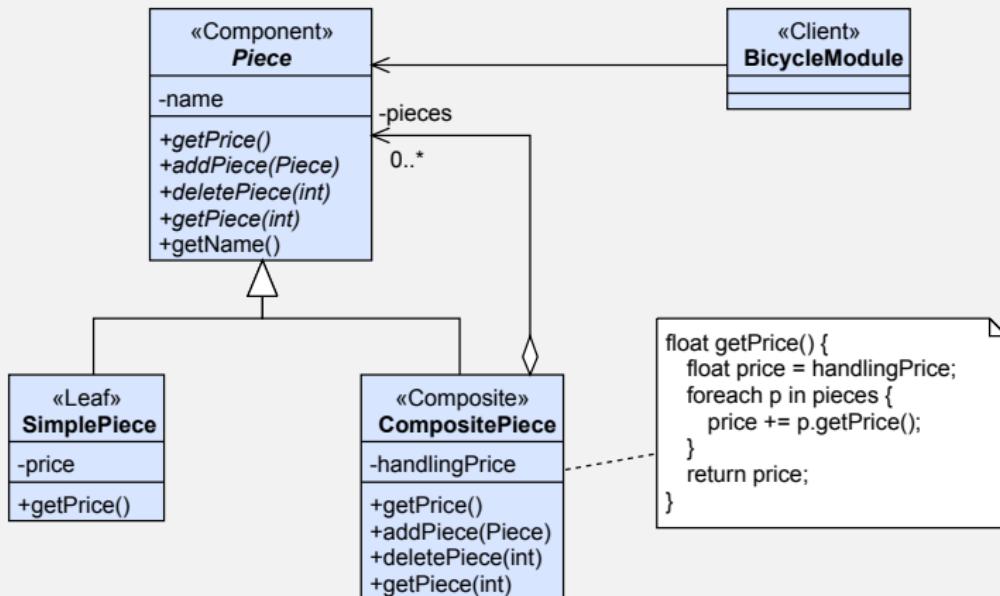
Solución: Patrón Composite (estructural)



[Gamma et al., 1994]

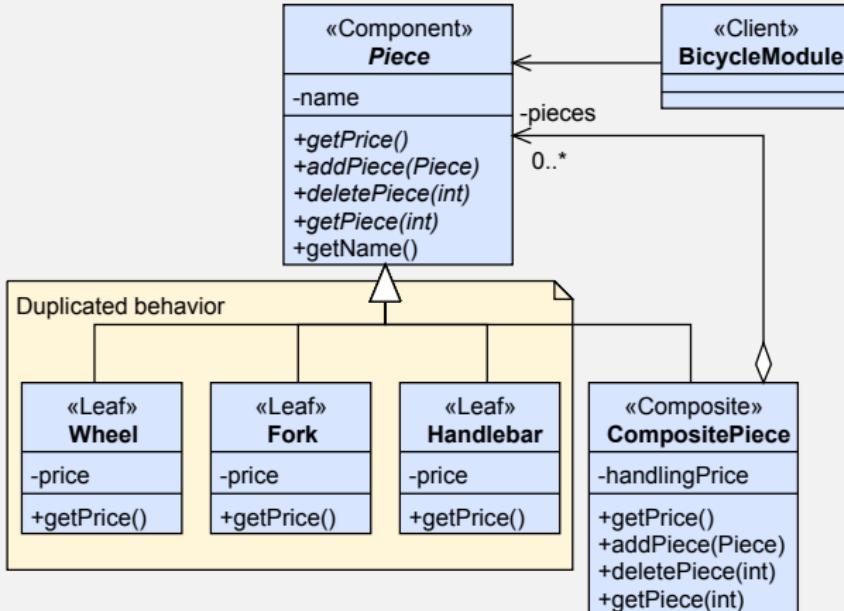
Solución

Patrón Composite



Discusión

Si todas las piezas simples tienen el mismo comportamiento
debería evitarse un diseño como éste:



Discusión

Ventajas:

- El cliente ve todos los objetos como si pertenecieran a la clase Piece y los trata de manera uniforme. Se simplifica el código del cliente.
- Es sencillo añadir nuevos componentes sin tener que modificar el cliente.

Inconvenientes:

- Las restricciones en cuanto a la composición de las piezas compuestas se debe hacer mediante código (en tiempo de ejecución).

Discusión

Detalles de implementación: hay dos alternativas para ubicar los métodos que gestionan los componentes de una pieza compuesta.

- Para que el cliente trate a todos los componentes por igual, deben compartir el mayor número posible de métodos en el interfaz común. De esta manera las piezas simples se comportarían como piezas compuestas que nunca pueden tener componentes → **mayor transparencia**.
- Sin embargo, según el principio de segregación de interfaces, las piezas simples no deberían tener métodos que no necesitan. Los métodos para gestionar los componentes estarían sólo en las piezas compuestas, de manera que el cliente no puede usar por error las piezas simples como si fueran compuestas → **mayor seguridad**.

Ejercicio 2

Enunciado

En un sistema que ofrece servicios a clientes remotos se desea realizar una auditoría del tiempo que tardan los distintos servicios en ejecutarse, a fin de mejorar el rendimiento global del sistema. Para esto se ha implementado una clase UtilTime, que no es abstracta, que incluye un método estático `time()` que mide el tiempo que tarda un método cualquiera en ejecutarse. Parte del código de dicho método sería:

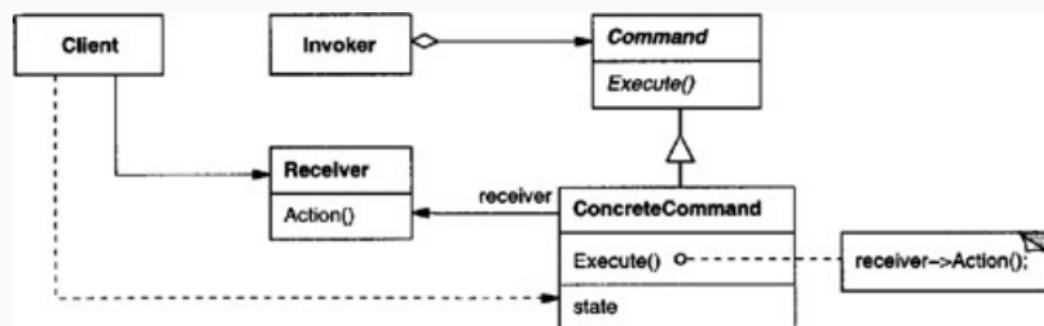
```
public static long time /* parámetros */ {  
    long t1 = System.currentTimeMillis();  
    // falta aquí el código apropiado  
    long t2 = System.currentTimeMillis();  
    return t2 - t1;  
}
```

Diseña una solución basada en alguno de los patrones GOF y completa el método: parámetros y código en la posición del comentario. Escribe un código cliente con un ejemplo de utilización del método `time()`.

Análisis del problema

Para solucionar el problema necesitamos **delegar la ejecución** de los servicios, de manera que ahora será la clase UtilTime la encargada de ejecutarlos. Para esto es necesario **convertir los servicios (operaciones) en objetos**, manteniendo su inicialización donde se hacía originalmente, y añadiéndoles un método que permita ejecutarlos.

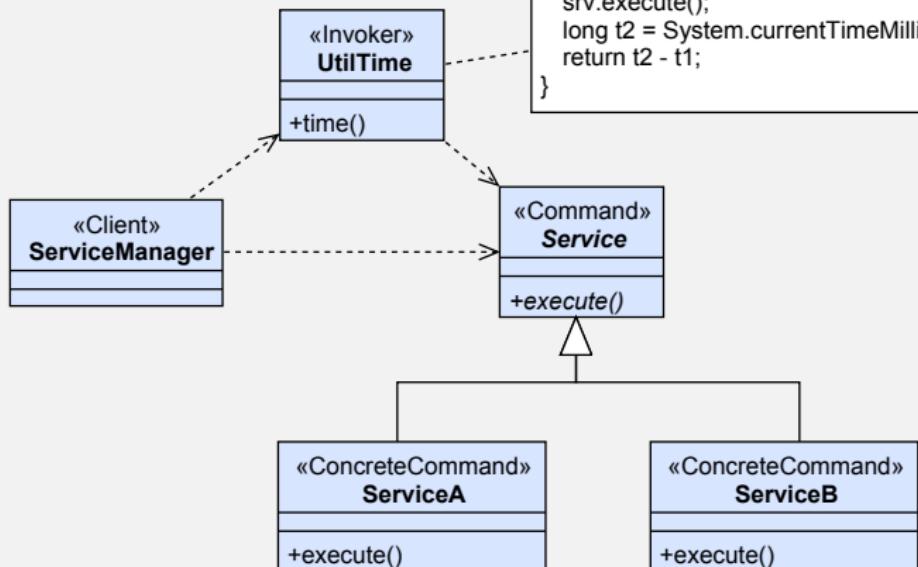
Solución: Patrón Command (de comportamiento)



[Gamma et al., 1994]

Solución

Patrón Command



Solución

```
class ServiceManager {
    public void auditService() {
        Service service = new ServiceA();
        HashMap params = new HashMap();
        params.add("param1", "value1"); // See discussion
        long time = UtilTime.time(service, params);
        // do some stuff
    }
}

class UtilTime {
    public static long time(Service srv, HashMap params) {
        long t1 = System.currentTimeMillis();
        srv.execute(params);
        long t2 = System.currentTimeMillis();
        return t2 - t1;
    }
}
```

Discusión

Detalles de implementación: al compartir todos los servicios un interfaz común se debe proporcionar un mecanismo para poder pasárselos la información necesaria para ejecutarse, ya que pueden necesitar un número distinto de parámetros, o parámetros de distinto tipo.

Una implementación alternativa al uso de mapas sería:

```
class ServiceManager {
    public void auditService() {
        Service service = new ServiceA("value1");
        /* or */
        Service service = new ServiceA();
        service.setParam("param1", "value1");

        long time = UtilTime.time(service);
        // do some stuff
    }
}
```

Discusión

Ventajas:

- Al convertir operaciones en objetos podemos manipularlos con mayor flexibilidad, p.ej. delegando su ejecución o encollarlos para ejecutarlos en orden cuando los recursos son limitados.
- Permite implementar la funcionalidad "Deshacer", si cada comando implementa una operación `unexecute()` y se guarda un histórico de los comandos ejecutados.
- Se pueden definir comandos compuestos (macros) combinando este patrón con el patrón Composite.
- Facilita llevar un registro de las operaciones ejecutadas, añadiendo a cada comando la posibilidad de almacenar información sobre su ejecución.
- Es sencillo añadir nuevos comandos.

Discusión

Inconvenientes:

- Cuando un comando actúa sobre otro objeto (*Receiver*), es difícil decidir qué responsabilidades corresponden a cada uno de ellos. El comando puede actuar como un simple mensaje para indicar al receptor que debe realizar alguna operación, o puede implementar toda la funcionalidad él mismo.
- Para asegurar la consistencia de la aplicación cuando se ofrece la posibilidad de deshacer y rehacer operaciones, se debe almacenar información precisa que permita devolver la aplicación a su estado anterior. Esto puede aumentar significativamente la complejidad.

Ejercicio 3

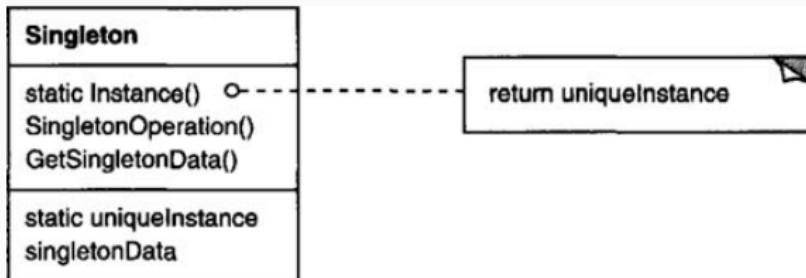
Enunciado

Nos han pedido introducir en la aplicación que estamos diseñando una clase de login que proporcione un **punto de acceso global** a la operación de login y password para todos los componentes de la aplicación. Propón una solución utilizando un patrón GOF.

Análisis del problema

Para proporcionar un punto de acceso global debe haber **una única instancia de la clase**, accesible mediante un método estático. De esta manera todos los componentes controlan el acceso a través de la misma instancia, evitando inconsistencias.

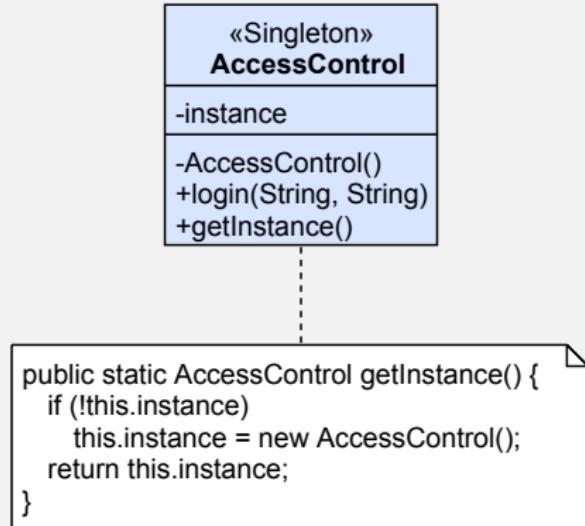
Solución: Patrón Singleton (creacional)



[Gamma et al., 1994]

Solución

Patrón Singleton



Discusión

Ventajas:

- Permite controlar el acceso a la única instancia.
- Se puede modificar fácilmente para permitir un número limitado de instancias reutilizables (*pooling*).
- Más flexible que usar un método estático para implementar la funcionalidad.

Inconvenientes:

- Si se quieren crear subclases de la clase Singleton la solución no es trivial, ya que todas comparten el atributo que almacena la única instancia.

Ejercicio 4

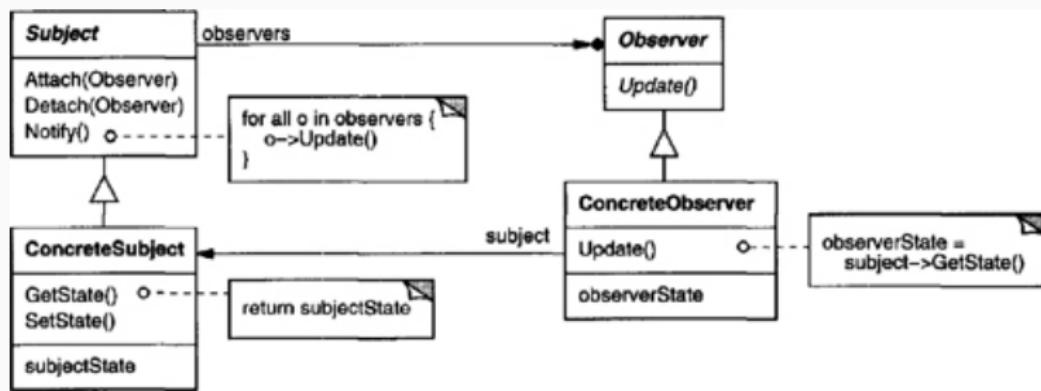
Enunciado

Nos ha contactado la Agencia F de noticias. Esta agencia recoge noticias y comunicados de todo el mundo, y las distribuye a todos sus clientes (periódicos, cadenas de televisión, etc.). Nos han pedido que creamos un framework para que la agencia pueda informar inmediatamente, cuando ocurre cualquier evento, a cualquiera de sus clientes. Estos clientes pueden recibir las noticias de distintas maneras: Email, SMS, etc. Además, nos han pedido que la solución sea lo suficientemente extensible para soportar nuevos tipos de subscripciones (p.ej. recientemente ya hay algún cliente que les ha pedido ser notificado a través de Twitter).

Análisis del problema

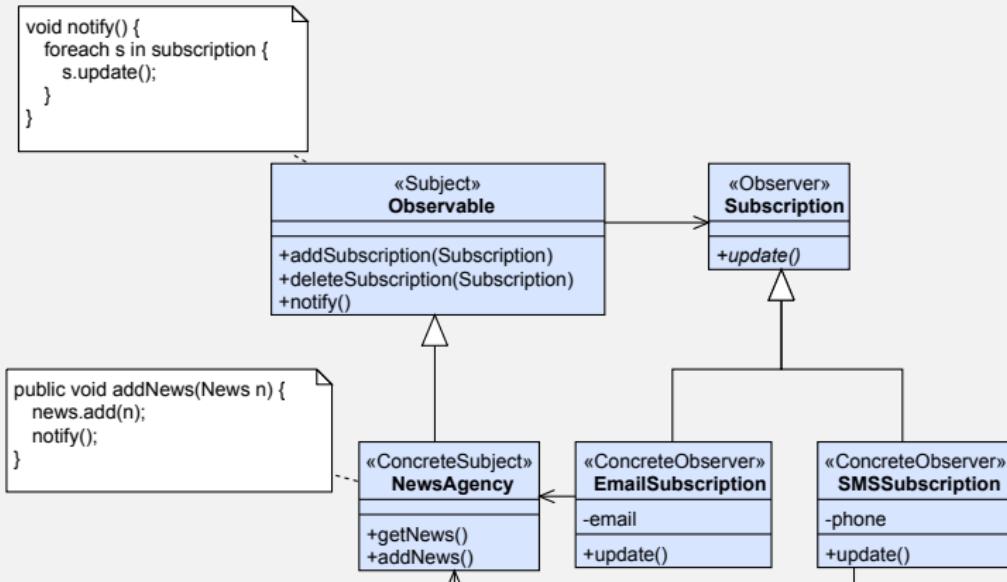
Para este problema necesitamos crear objetos que representan las suscripciones e informen a los clientes cuando haya alguna novedad. Sin embargo, no es viable hacer que estos objetos vigilen activamente el objeto que contiene las noticias, ya que afectaría negativamente al rendimiento. En lugar de esto los objetos “suscripción” **se registrarán para que se les notifique cuando haya alguna novedad.**

Solución: Patrón Observer (de comportamiento)



Solución

Patrón Observer



Discusión

Ventajas:

- El objeto observado no necesita saber a priori cuántos objetos debe notificar ni a qué tipo pertenecen.
- Al estar muy poco acoplados, estos objetos (Object y Observer) pueden estar en capas distintas de la aplicación.

Inconvenientes:

- Cada pequeña actualización en el objeto Subject desencadena una cascada de notificaciones a los observadores. Se puede evitar haciendo que otro objeto Cliente (el encargado de modificar el Subject) dispare las notificaciones después de realizar todas las modificaciones necesarias.

Discusión

Detalles de implementación: existen dos alternativas para que los observadores obtengan la información que les interesa:

- El objeto Subject pasa la información al Observer a través del método update() → **modelo push**. Requiere que el Subject conozca mejor a los Observers, aumenta el acoplamiento.
- El objeto Observer se encarga de recuperar la información que necesita después de ser notificado → **modelo pull**. El Observer debe hacer comprobaciones adicionales para averiguar qué ha cambiado, es más ineficiente.

Ejercicio 5

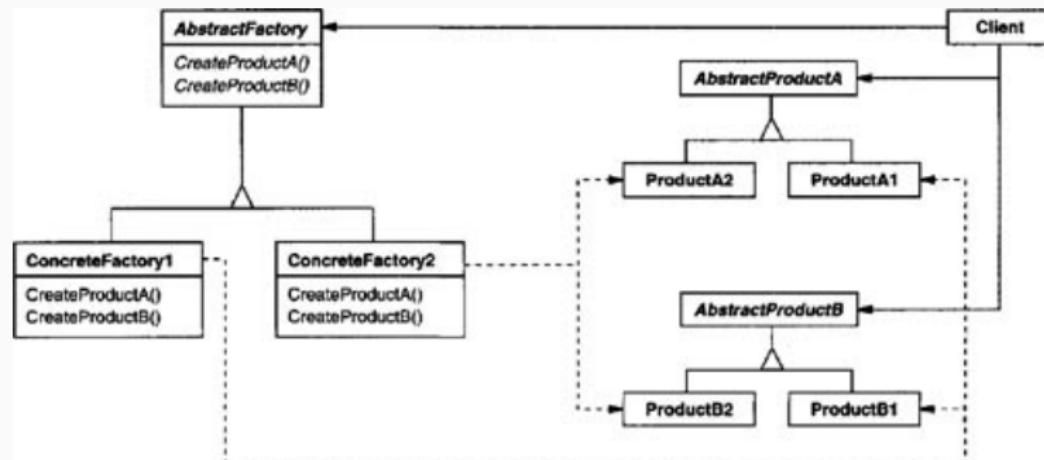
Enunciado

Queremos implementar un manejador de información personal, que controla, entre otras cosas, números de teléfono y direcciones. El alta de números de teléfono sigue una **regla particular**, que **depende de** la región y país a la que pertenezca el número. Sabemos que el número de países manejados por la aplicación va a crecer en un futuro, y queremos proteger nuestro código de esa variación. Propón un diseño que, utilizando un patrón GOF, solucione este problema.

Análisis del problema

Los datos siguen unas reglas de formato que dependen del país. Es conveniente proporcionar un mecanismo que **asegure la consistencia** y permita formatear correctamente los datos, indicando una única vez el país.

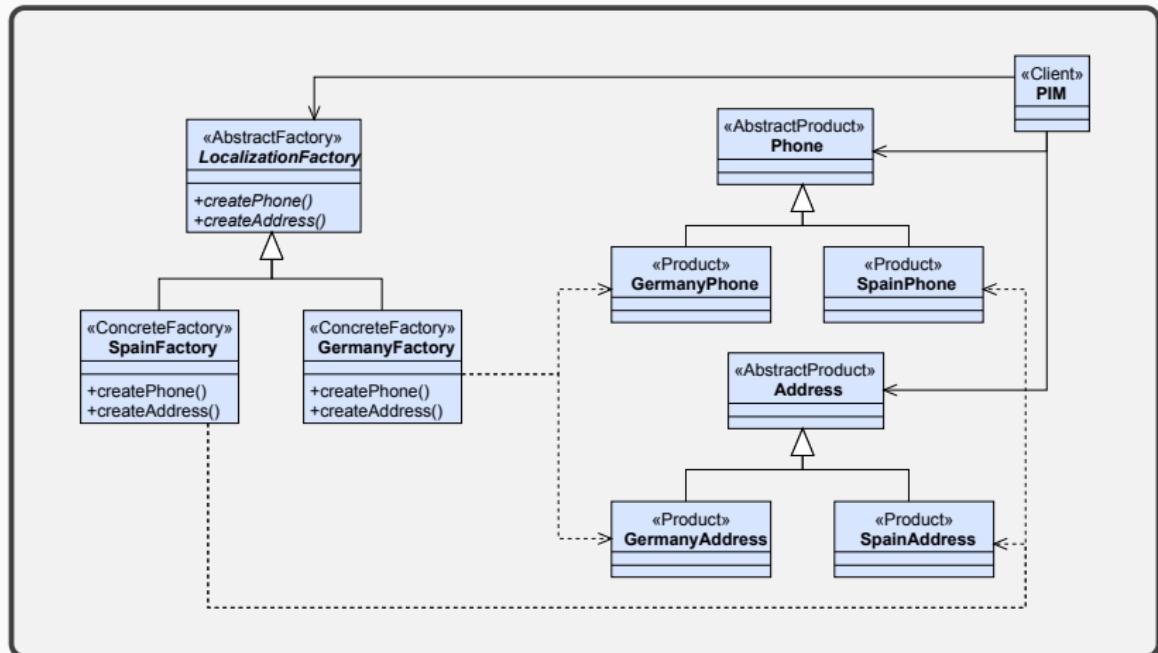
Solución: Patrón Abstract Factory (creacional)



[Gamma et al., 1994]

Solución

Patrón Abstract Factory



Discusión

Ventajas:

- Asegura la consistencia de los productos.
- El cliente sólo trata con clases abstractas, no necesita conocer las clases concretas.
- Es sencillo cambiar familias de productos (países en este ejemplo).

Inconvenientes:

- Involucra un número muy elevado de clases.
- Para añadir nuevos productos debemos modificar los objetos ConcreteFactory, lo que implica modificar también la clase abstracta AbstractFactory.

Ejercicio 6

Enunciado

Nos han pedido implementar un sistema para manejar el envío de artículos a conferencias. Una conferencia tiene distintas fases; por simplicidad vamos a asumir cuatro: **Call4Abstracts**, **Call4Papers**, **Revisión**, **Call4CameraReady**. El diseño inicial contiene tres clases: autor, artículo y conferencia. El sistema debe soportar cinco llamadas de interfaz:

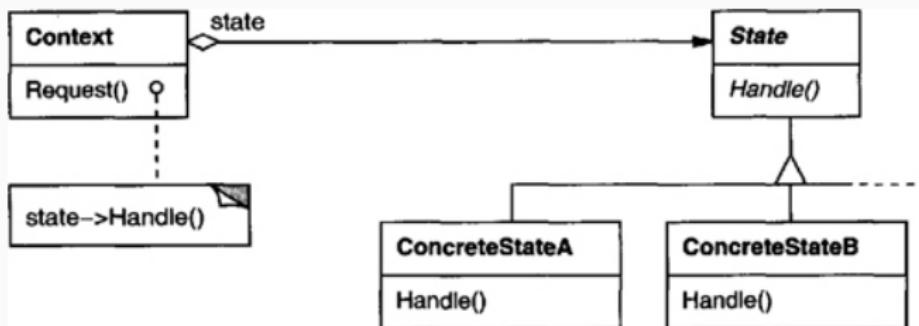
- envíaAbstract()
- envíaArtículo()
- revisaArtículo()
- notificaRevisión()
- envíaCameraReady()

El comportamiento de estas operaciones variará según la fase del proceso de revisión en que nos encontremos. Plantea una solución a este problema.

Análisis del problema

Para cambiar fácilmente el comportamiento de una clase dependiendo de su estado podemos **externalizar su comportamiento** fuera de la clase, proporcionando **distintas implementaciones que representan los distintos estados**.

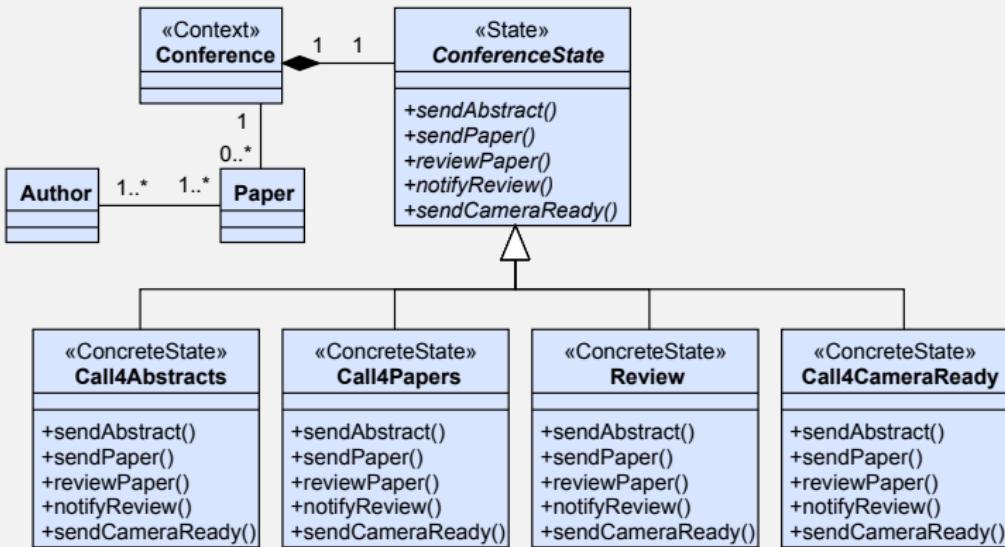
Solución: Patrón State (de comportamiento)



[Gamma et al., 1994]

Solución

Patrón State



Discusión

Ventajas:

- Las operaciones ya no necesitan lógica condicional para comprobar el estado del objeto.
- Toda la lógica asociada a un estado se encuentra en el mismo objeto (mayor cohesión).
- Hace que el cambio de estado sea una operación explícita y atómica, evitando posibles problemas de consistencia interna.

Inconvenientes:

- Todos los estados deben proporcionar todas las operaciones, aunque no las necesiten.

Discusión

Detalles de implementación: los cambios de estado se pueden implementar de dos maneras:

- El objeto principal puede ser el encargado de decidir cuál es el estado siguiente. Permite mantener la gestión de los estados en un punto centralizado.
- Los propios objetos State pueden ser los encargados de proporcionar el siguiente estado, p.ej. mediante un método `nextState()`. Reduce la complejidad del objeto principal pero introduce acoplamiento entre los distintos estados.

Ejercicio 7

Enunciado

Imagine que el profesor Jackson llama al profesor Ernst a media noche porque alguien ha descubierto que **hay un problema** relacionado con el sistema que están desarrollando para la famosa cadena de tiendas Decathlon: éste debe ser capaz de soportar bicicletas que se puedan volver a pintar (para cambiar su color). Los profesores dividen el trabajo: el profesor Jackson escribirá la clase ColorPalette con un método que, dado un nombre como “rojo”, “azul” o “ceniza”, devuelva un array con tres valores RGB, y el profesor Devadas escribirá un código que utilice esta clase. Los profesores hacen esto, pasan los tests al trabajo realizado, se van de fin de semana, y dejan los archivos .class para que los becarios del proyecto los integren.

Enunciado

Estos se dan cuenta de que el profesor Devadas ha escrito un código que depende de:

```
interface ColorPalette {  
    // devuelve valores RGB  
    int[] getColor(String name);  
}
```

Sin embargo, el profesor Jackson implementa una clase que se adhiere a:

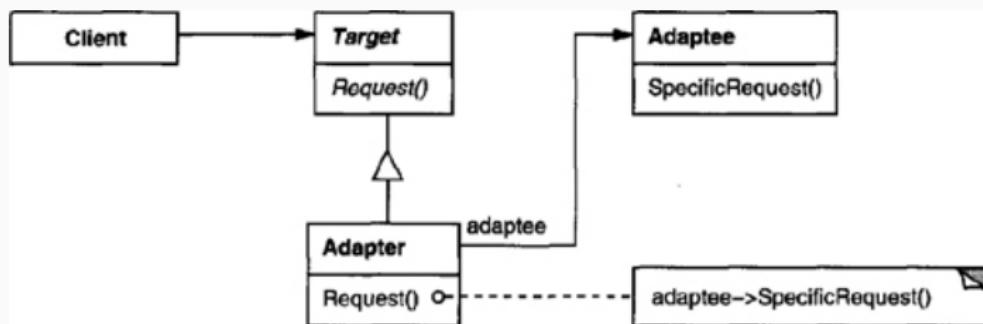
```
interface ColourPalette {  
    // devuelve valores RGB  
    int[] getColores(String name);  
}
```

¿Qué es lo que los becarios tienen que hacer? Ellos **no tienen acceso a la fuente**, y no disponen de tiempo para volver a implementar y volver a pasar las pruebas.

Análisis del problema

Para solucionar el problema hay que proporcionar una clase que implemente el interfaz que espera el cliente, y se encargue de **adaptar este interfaz** al que proporciona la clase que implementa la funcionalidad.

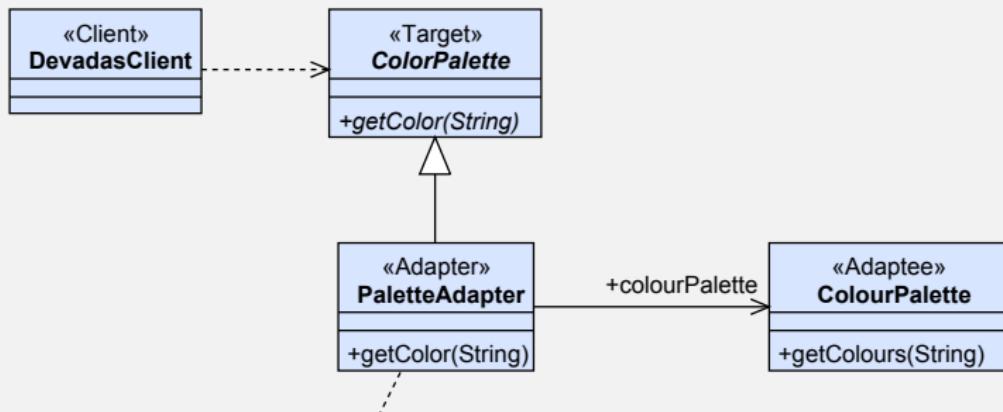
Solución: Patrón Adapter (estructural)



[Gamma et al., 1994]

Solución

Patrón Adapter



```
private ColourPalette colourPalette = new ColourPalette();  
  
public int[] getColor(String name) {  
    return this.colourPalette.getColours(name);  
}
```

Discusión

Ventajas:

- Permite conectar las dos clases sin tener que cambiar su implementación.
- Permite añadir funcionalidad adicional si fuera necesario.

Inconvenientes:

- Al añadir un nivel de indirección puede afectar ligeramente al rendimiento.

¿Preguntas?

Referencias i

-  Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. (1994).
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley Professional.
Safari Books Online.

Ejercicios sobre patrones GOF (2)

Diseño de Sistemas Software
Curso 2020/2021

Carlos Pérez Sancho



Universitat d'Alacant
Universidad de Alicante

Departament de Llenguatges i Sistemes Informàtics
Departamento de Lenguajes y Sistemas Informáticos

Ejercicio 8

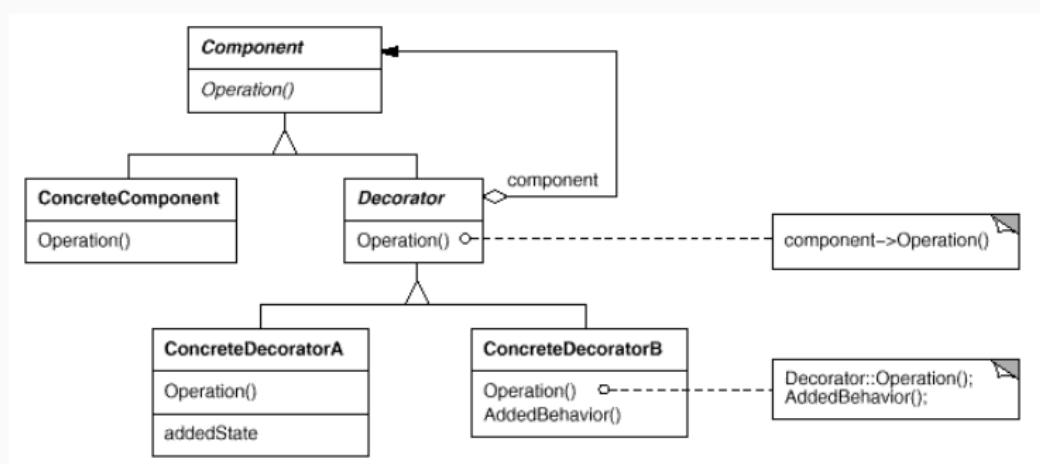
Enunciado

Muy contentos del trabajo realizado con el manejo de las partes de la bicicleta, Decathlon nos ha pedido un **nuevo prototipo**. Esta vez quiere que se **extienda la funcionalidad** de la aplicación para poder trabajar con distintas variaciones de las propiedades de las piezas. Por ejemplo, un manillar tiene un precio base, que puede verse incrementado si queremos algún “extra” (p.ej. pintura metalizada, materiales de gama superior, etc.). Partiendo del diseño anterior, enriquecelo para que soporte esta nueva variación.

Análisis del problema

Para poder calcular el precio teniendo en cuenta los extras y sin modificar el código cliente, necesitamos **añadir nuevas funcionalidades dinámicamente a las piezas**, de manera que se pueda calcular el precio de una pieza **sin modificar su interfaz**.

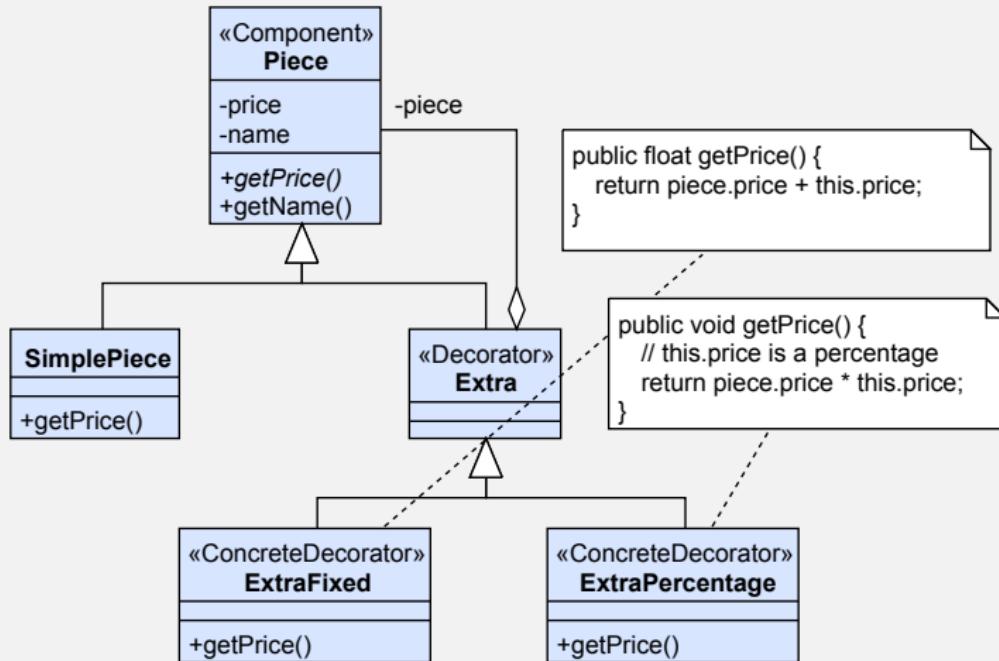
Solución: Patrón Decorator (estructural)



[Gamma et al., 1994]

Solución

Patrón Decorator



Discusión

Ventajas:

- El uso de los decoradores es transparente al cliente, ya que implementan el mismo interfaz que las piezas.
- Se pueden anidar tantos decoradores como sea necesario.
- Se pueden añadir y quitar decoradores en tiempo de ejecución.

Inconvenientes:

- Se crean muchos objetos pequeños, puede dificultar la comprensión del sistema.

Ejercicio 9

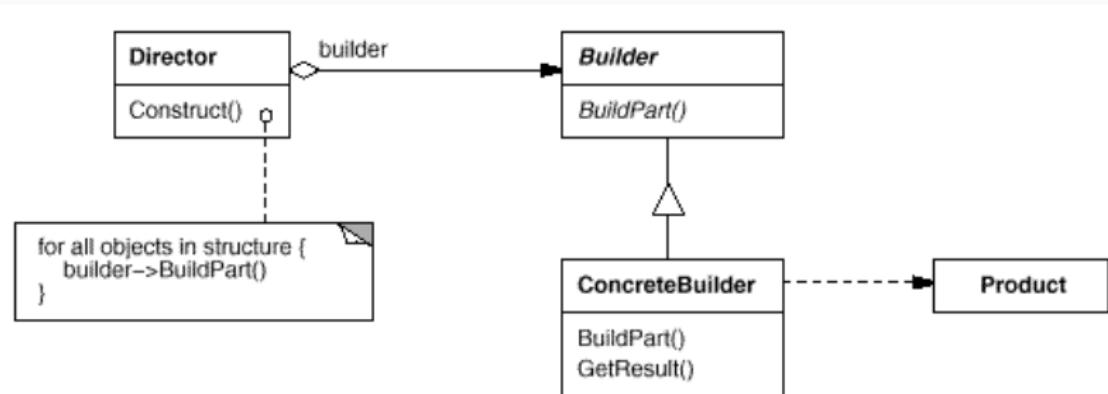
Enunciado

Imaginad que estamos construyendo una aplicación para restaurantes de comidas rápidas que preparan menús infantiles. Para ello hemos creado una clase Menú, de la que hereda Menú infantil. Generalmente estos menús están formados de un plato principal, un acompañamiento, una bebida y un juguete. Si bien el contenido del menú puede variar, el proceso de construcción es siempre el mismo: el cajero indica a los empleados los pasos a seguir. Estos pasos son: preparar un plato principal, preparar un acompañamiento, incluir un juguete y guardarlos en una bolsa. La bebida se sirve en un vaso y queda fuera de la bolsa. Completad el diseño.

Análisis del problema

Queremos **separar el proceso de construcción del menú de la lógica condicional** que hay en cada paso de la construcción. En cada paso hay que comprobar el tipo de menú, así que la solución es usar **objetos constructores para cada tipo de menú** que implementen todos los pasos de la construcción.

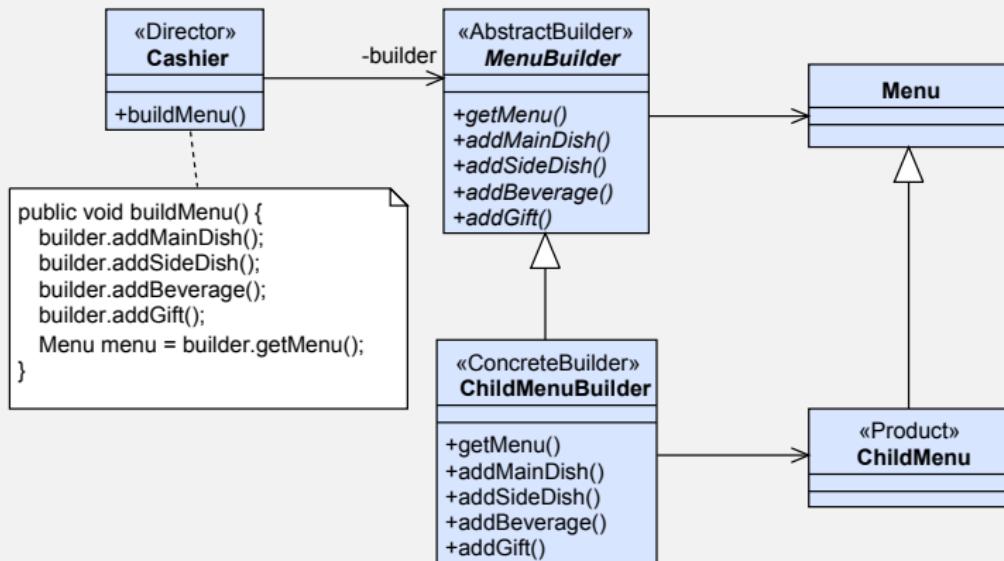
Solución: Patrón Builder (creacional)



[Gamma et al., 1994]

Solución

Patrón Builder



Discusión

Ventajas:

- Permite separar el algoritmo de construcción de los detalles sobre cómo crear las partes del objeto.
- Permite crear distintos tipos de objetos siguiendo el mismo algoritmo.

Inconvenientes:

- Puede haber productos que no necesiten alguna de las partes, en ese caso el ConcreteBuilder debe tener una implementación vacía del método.

Ejercicio 10

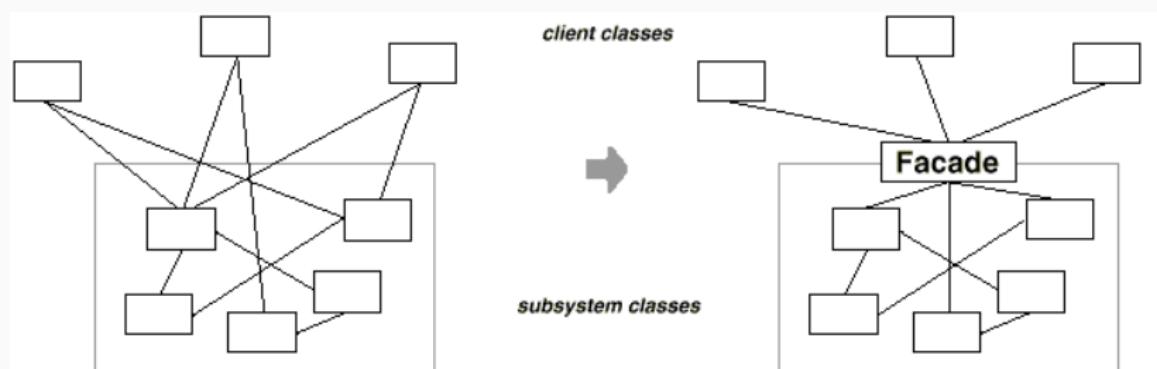
Enunciado

Sea un conjunto de clases que permiten la creación y envío de mensajes de correo electrónico y que entre otras incluye clases que representan el cuerpo del mensaje, los anexos, la cabecera, el mensaje, la firma digital y una clase encargada de enviar el mensaje. El código cliente debe interactuar con instancias de todas estas clases para el manejo de los mensajes, por lo que debe conocer en qué orden se crean esas instancias; cómo colaboran esas instancias para obtener la funcionalidad deseada y cuáles son las relaciones entre las clases. Idea una solución basada en algún patrón tal que se reduzcan las dependencias del código cliente con esas clases y se reduzca la complejidad de dicho código cliente para crear y enviar mensajes. Dibuja el diagrama de clases que refleje la solución e indica qué patrón has utilizado.

Análisis del problema

Queremos **evitar un acoplamiento excesivo** con todas las clases involucradas para el envío de un correo electrónico. Para esto introducimos un nivel de indirección, **reduciendo el acoplamiento a una única clase fachada**.

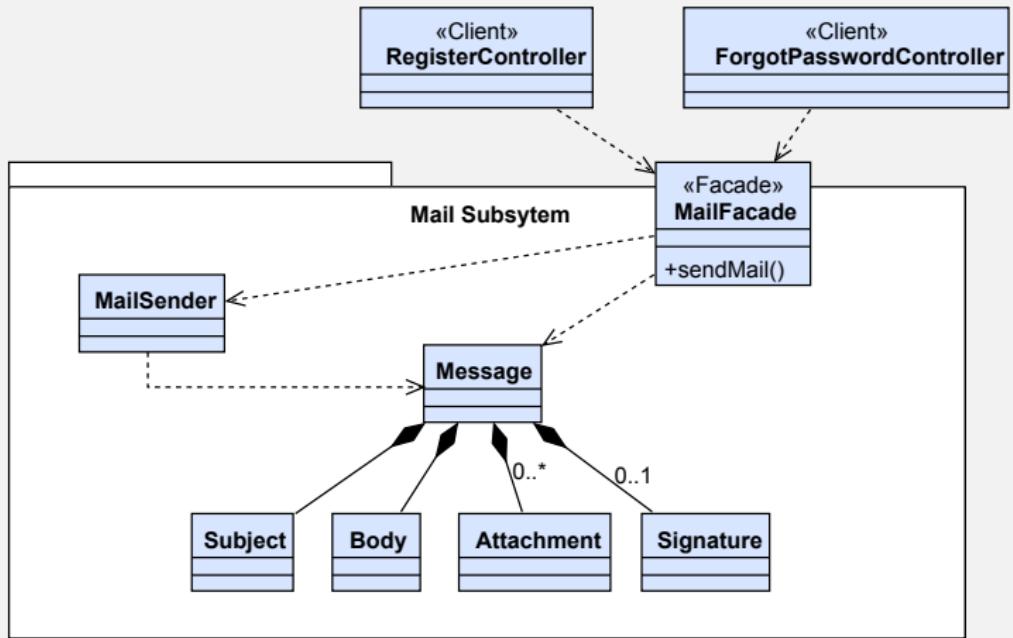
Solución: Patrón Facade (estructural)



[Gamma et al., 1994]

Solución

Patrón Facade



Discusión

Ventajas:

- Proporciona un interfaz fácil de usar para un subsistema complejo.
- Desaparecen las dependencias entre numerosos clientes y las clases del subsistema.
- Permite reducir el acoplamiento entre distintas capas, haciendo que las capas se comuniquen mediante llamadas a sus fachadas.
- Aún así, las clases internas pueden ser utilizadas si es necesario.

Ejercicio 11

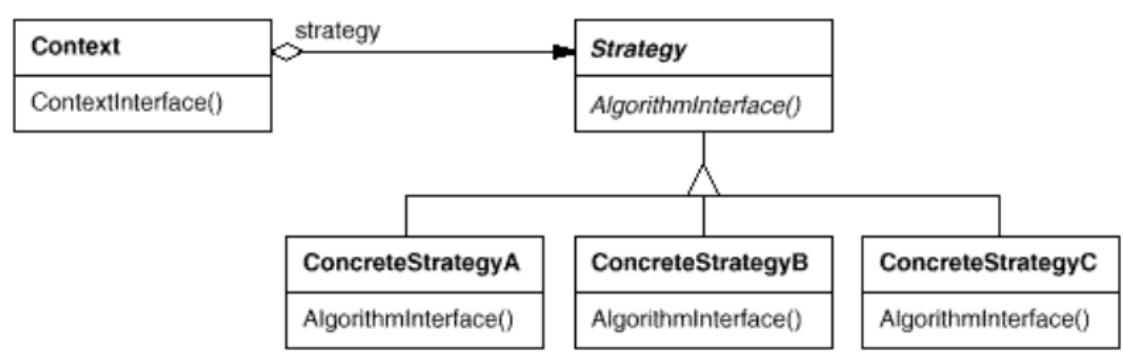
Enunciado

Suponed que estamos desarrollando un juego interactivo basado en la serie Héroes. El juego cuenta con distintos personajes, y cada uno tiene una serie de habilidades especiales: volar, lanzar rayos, leer la mente, regenerarse, teletransportarse, ... Los personajes pueden adquirir/perder poderes de forma dinámica según se desarrolla la trama del juego. Por ejemplo, el padre de Peter y Nathan Petrelli puede desposeer a cualquier personaje de sus poderes. Sylar, por otro lado, va adquiriendo poderes según va analizando a sus víctimas. Además, cualquier personaje puede ver inhabilitados sus poderes cuando están capturados o cuando el Haitiano está cerca o hay un eclipse. Los personajes también ven modificado el comportamiento de sus habilidades cuando están aturdidos.

Análisis del problema

Para poder **intercambiar distintos comportamientos en tiempo de ejecución** es necesario sacar la implementación de la clase, de manera que se ofrecen **distintas implementaciones en clases separadas con el mismo interfaz**.

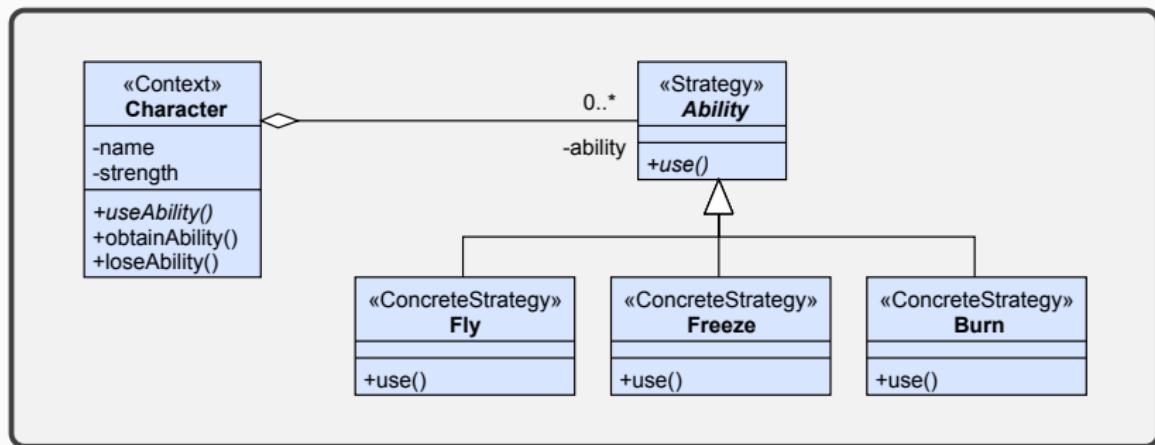
Solución: Patrón Strategy (de comportamiento)



[Gamma et al., 1994]

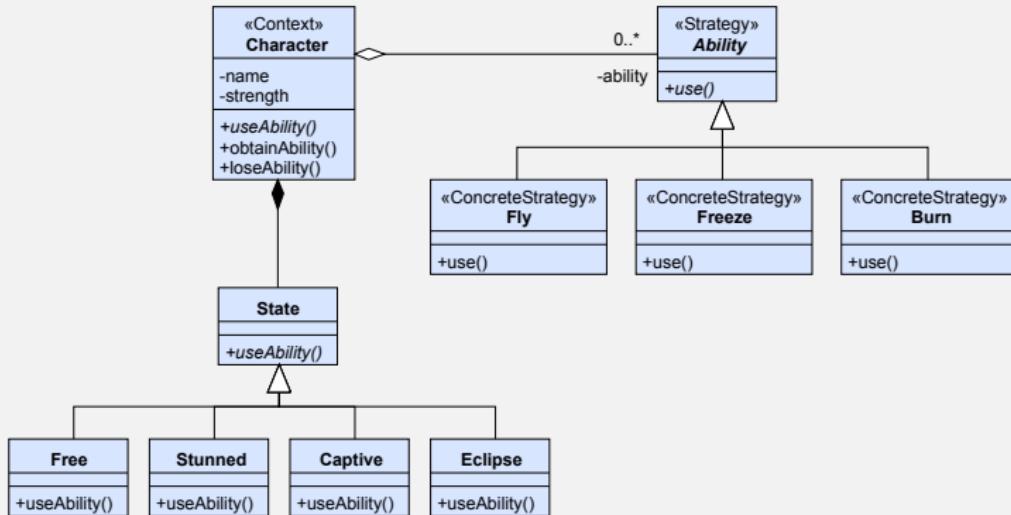
Solución

Patrón Strategy



Solución

Patrón Strategy + State



Discusión

Ventajas:

- Los distintos comportamientos se pueden sustituir en tiempo de ejecución, sustituyendo la instancia que guarda la clase cliente por otra implementación del algoritmo.
- Permite ocultar detalles de implementación del comportamiento que son irrelevantes para el cliente.
- Elimina lógica condicional en el cliente.

Inconvenientes:

- La clase cliente debe conocer todas las subclases que implementan el comportamiento. Se puede resolver usando una clase Factory para crear las instancias.

Ejercicio 12

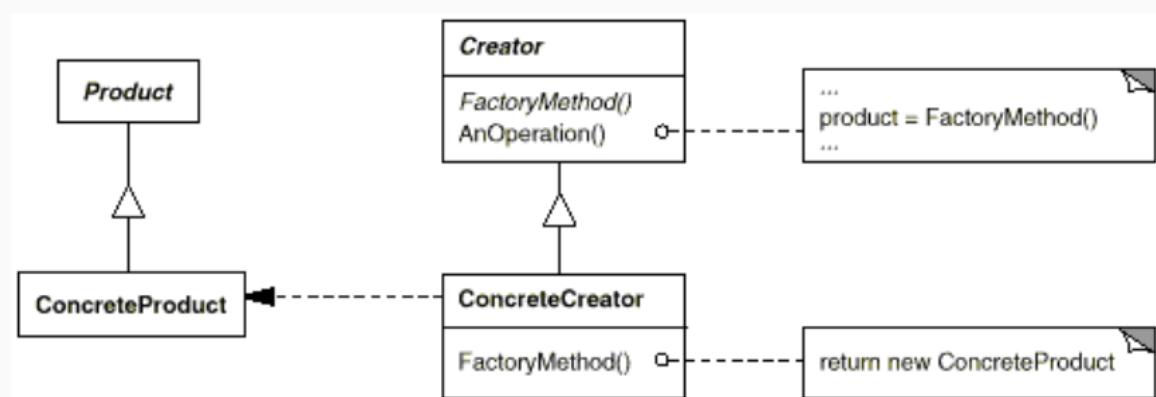
Enunciado

Nos han pedido colaborar en la elaboración de un **framework** para gestionar aplicaciones de manejo de documentos. Este framework debe asegurar, entre otras cosas, el correcto manejo de las peticiones de apertura, creación y salvado de documentos de cualquier tipo, donde el tipo concreto dependerá de la aplicación que extienda nuestro framework. Por ejemplo, una aplicación de dibujo contendrá una clase `AplicaciónDibujo`, y una clase `DocumentoDibujo`.

Análisis del problema

Tenemos una funcionalidad común a todas las aplicaciones (abrir, crear, guardar), que depende de la **creación de objetos que no se conocen a priori**. Se puede implementar esta funcionalidad delegando la creación de estos objetos en **subclases que implementen el método encargado de crearlos**.

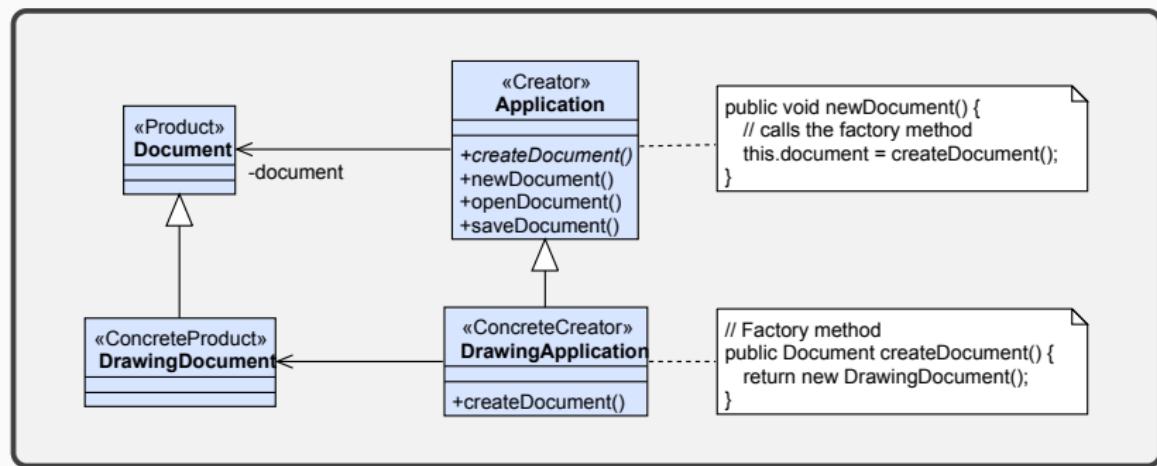
Solución: Patrón Factory Method (creacional)



[Gamma et al., 1994]

Solución

Patrón Factory Method



Discusión

Ventajas:

- Permite implementar una funcionalidad que depende de objetos que no se conocen.

Inconvenientes:

- No se puede aplicar si los distintos objetos a crear son muy diferentes y no se puede extraer un interfaz común.

Ejercicio 13

Enunciado

Nos han pedido implementar una aplicación para una agencia de viajes. La agencia maneja distintos tipos de paquetes; todos ellos comparten un **protocolo común**, que incluye lo siguiente:

- Todos los turistas son transportados al lugar de destino por el medio de transporte seleccionado
- Cada día tienen una actividad programada
- Todos los turistas vuelven (si quieren)

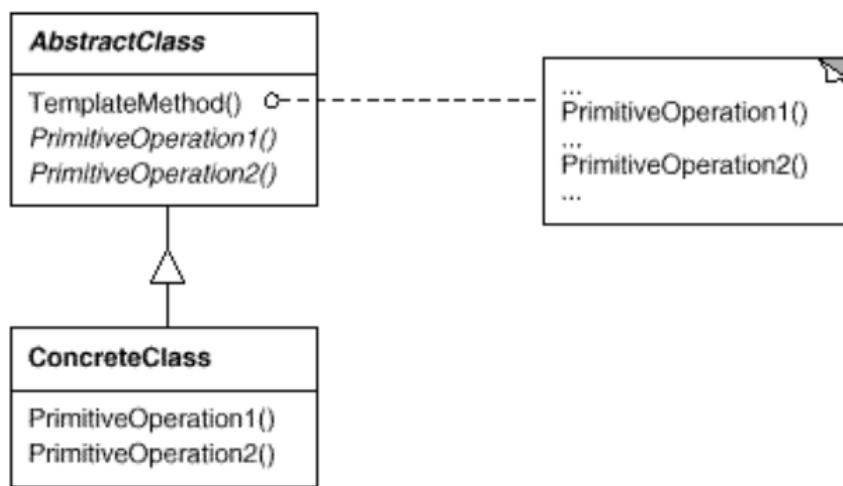
Sin embargo, según el paquete elegido, hay **ciertas variaciones**:

- Los turistas pueden ser llevados en transfer desde el hotel al lugar donde cojan su transporte o ir por su cuenta.
- Las actividades programadas pueden estar incluidas o no en el precio del paquete
- ...

Análisis del problema

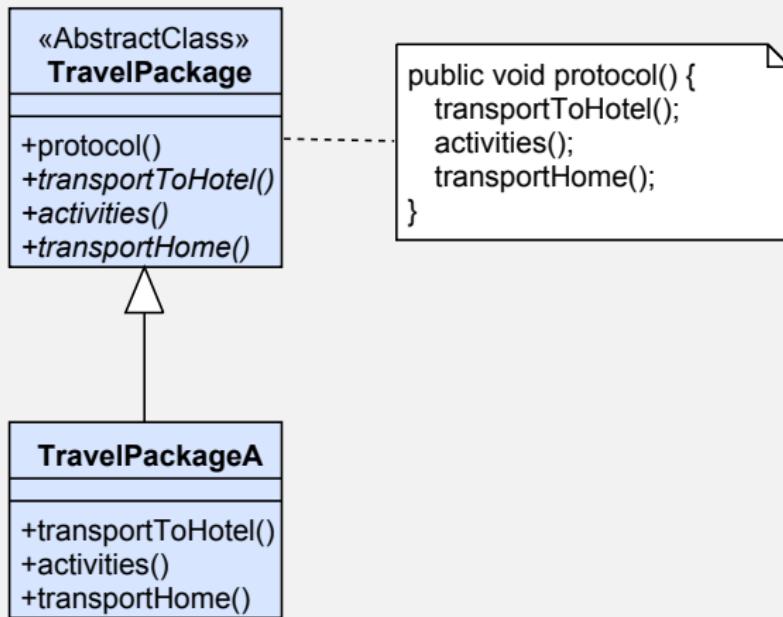
Tenemos un **protocolo común, cuyos pasos se pueden implementar de distintas maneras**. La implementación de los pasos se hará en clases hijas que heredan el comportamiento común e implementan de forma diferente cada uno de los pasos.

Solución: Patrón Template Method (de comportamiento)



Solución

Patrón Template Method



Discusión

Ventajas:

- Permite implementar las partes invariantes de un algoritmo, delegando en las subclases la implementación de los detalles.
- Proporciona un mecanismo de inversión de control, muy usado en frameworks (p.ej. controladores y middleware en Laravel).

Detalles de implementación:

- No se debería permitir sobreescibir el comportamiento del template method.

Ejercicio 14

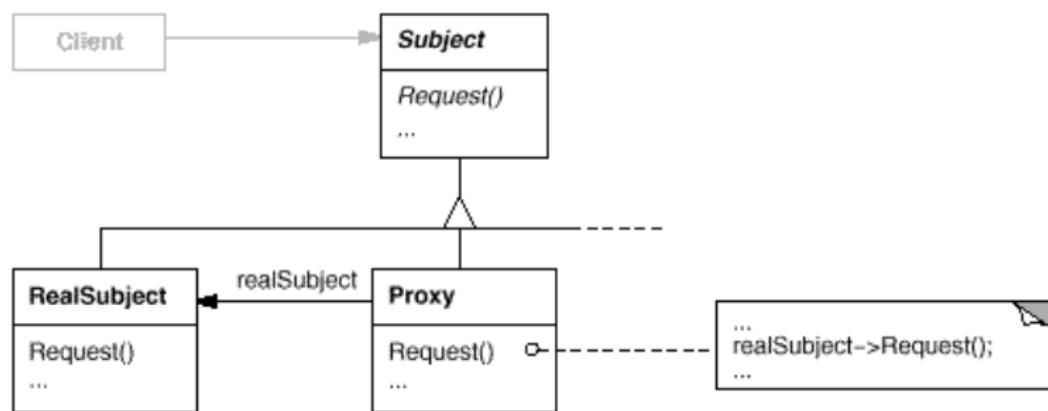
Enunciado

Se desea almacenar en una base de datos todas las transacciones realizadas por un terminal punto de venta (*point of sale*, POS). El terminal puede trabajar en modo conectado o no conectado, dependiendo de si la conexión a internet está en funcionamiento en el momento de almacenar los datos. Como es muy importante que no se pierda ninguna transacción, en el caso de que se esté trabajando en modo sin conexión, los datos deben almacenarse en el disco hasta que la conexión se restablezca.

Análisis del problema

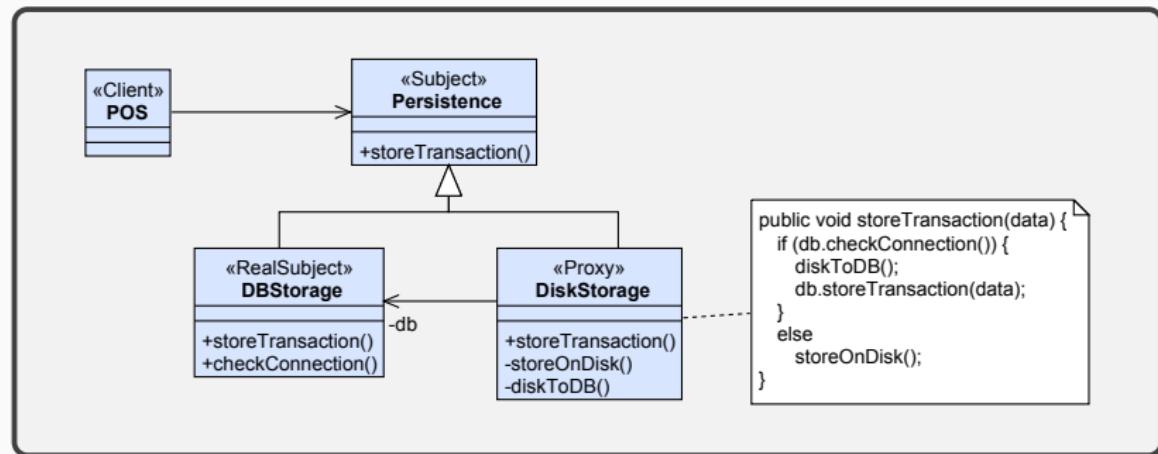
Para no aumentar la complejidad del cliente podemos situar un **objeto intermediario** entre éste y el objeto que se comunica con la base de datos, **imitando su interfaz**. Este nuevo objeto se encargará de gestionar los datos en caso de que falle la conexión para que no se pierdan transacciones.

Solución: Patrón Proxy (estructural)



Solución

Patrón Proxy



Discusión

Combinación de los patrones Decorator y Adapter.

Tipos de proxies:

- **Proxy virtual:** sustituyen objetos “caros” hasta que son necesarios. P.ej. para implementar la estrategia Lazy Loading.
- **Proxy remoto:** representa un objeto ubicado en un servidor remoto y se encarga de comunicarse con él.
- **Proxy de protección:** permite controlar el acceso a un objeto.
- **Smart reference:** permite realizar operaciones adicionales cuando se accede a un objeto. Por ejemplo:
 - Registrar todos los accesos al objeto real.
 - Liberar el objeto real cuando ya no hay objetos referenciándolo (*smart pointers*).
 - Bloquear el objeto real para evitar acceso concurrente.

¿Preguntas?

Referencias i

-  Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. (1994).
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley Professional.
Safari Books Online.