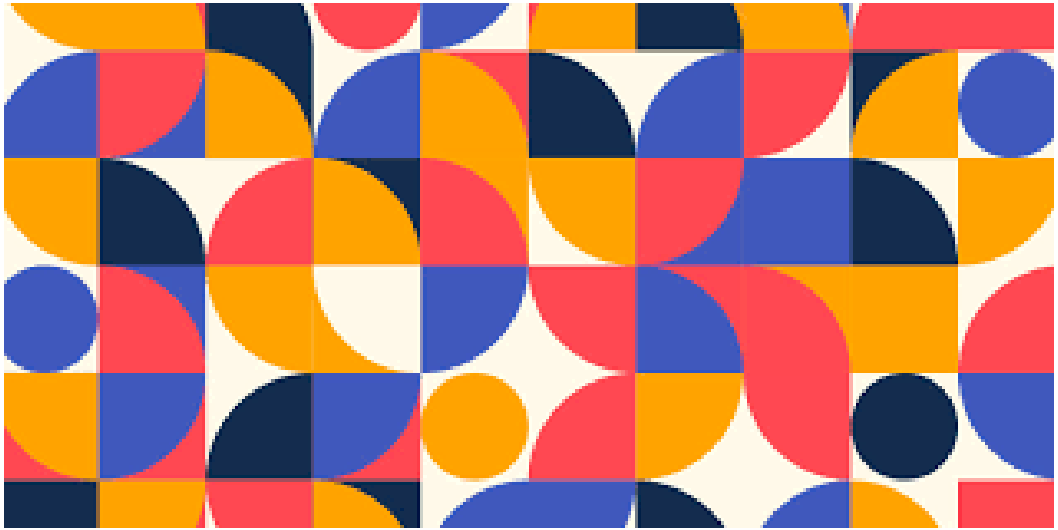


Patrones de Diseño

- más comunes..... -

ÍNDICE DE CONTENIDOS

SINGLETON

- Garantizar 1 única instancia de una clase.
- 1 punto de acceso a esa instancia.
- Método que crea y chequea si ya existe.
- Problema con multi-hilos. (mutex)

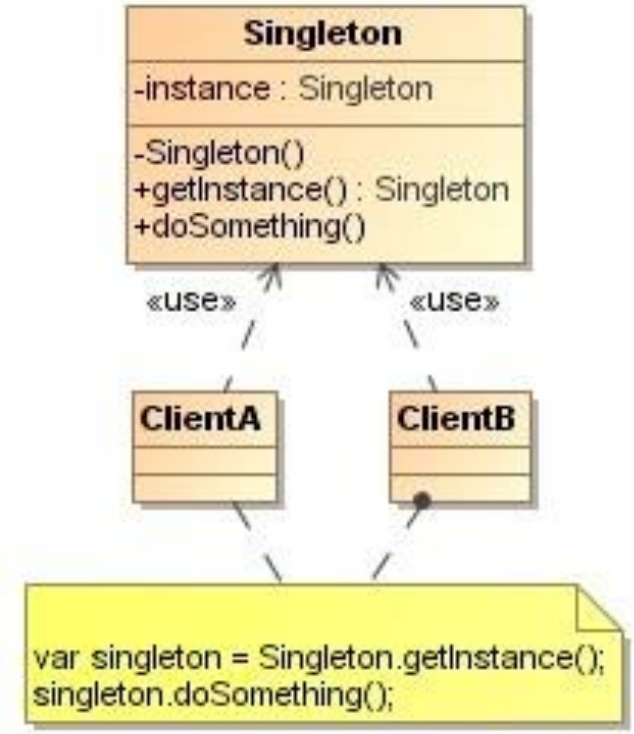
SINGLETON

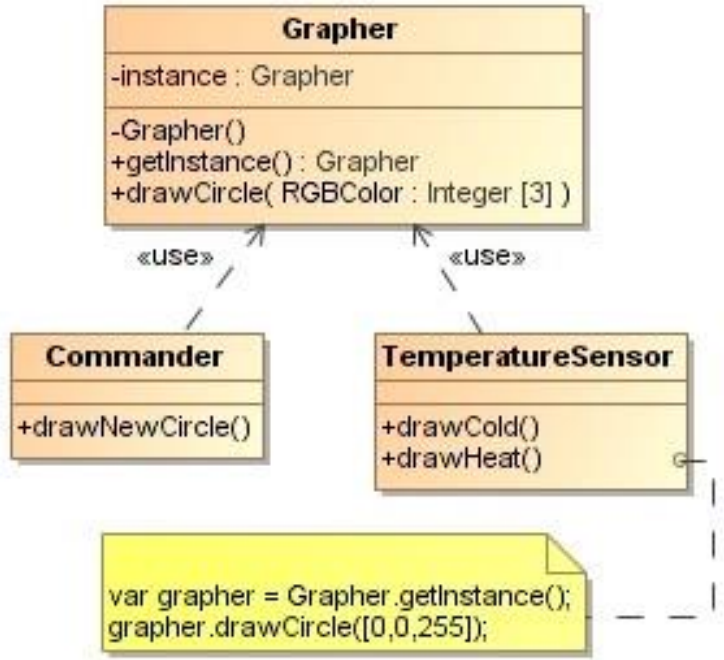
```
public class Singleton {
  private static Singleton instance = null;
  private String nombre;

  // El constructor es privado, no permite que se genere
  // un constructor por defecto.
  private Singleton(String nombre) {
    this.nombre = nombre;
    System.out.println("Mi nombre es " + this.nombre);
  }

  public static Singleton getInstance(String nombre) {
    if (Singleton.instance == null) {
      Singleton.instance = new Singleton(nombre);
    } else {
      System.out.println(
          "No se puede crear el objeto " + nombre +
          " porque ya existe un objeto de la clase Singleton");
    }
    return Singleton.instance;
  }
}
```

SINGLETON

```
public class SingletonSync {
  private static SingletonSync instance = null;

  // El constructor es privado, no permite que se genere
  private SingletonSync() {}

  // Creador sincronizado para protegerse de posibles problemas multi-hilo
  // otra prueba para evitar instanciación múltiple
  public static synchronized void createInstance() {
    if (instance == null) {
      instance = new SingletonSync();
    }
  }

  public static SingletonSync getInstance() {
    if (instance == null) {
      createInstance();
    }
    return instance;
  }
}
```

SINGLETON


Singleton
-instance : Singleton
-Singleton()
+getInstance() : Singleton
+doSomething()
«use»
«use»
ClientA
ClientB
var singleton = Singleton.getInstance();
singleton.doSomething();


Grapher
-instance : Grapher
-Grapher()
+getInstance() : Grapher
+drawCircle(RGBColor : Integer [3])
«use»
«use»
Commander
+drawNewCircle()
TemperatureSensor
+drawCold()
+drawHeat()
var grapher = Grapher.getInstance();
grapher.drawCircle([0,0,255]);

DAO

- Patrón **estructural**.
- Pretende **separar las capas** de aplicación y persistencia (BD normalmente).
- Para eso se crea un **API que esconde** las complejidades de las operaciones **CRUD** (**C**reate, **R**ead, **U**pdate y **D**elete)
- En Java DAOs básicos hasta frameworks JPA, EJB, Hibernate….

DAO

Ventajas:

- Separación de Clases de Negocio y Persistencia. (Cambios aislados)
- Encapsulación de los detalles de almacenamiento. (No tengo porqué conocer los detalles de la persistencia.

Desventajas:

- Esconde el coste de acceso (varias operaciones cuando solo necesito 1)
- Repito código para CRUD de diferentes DAOs.

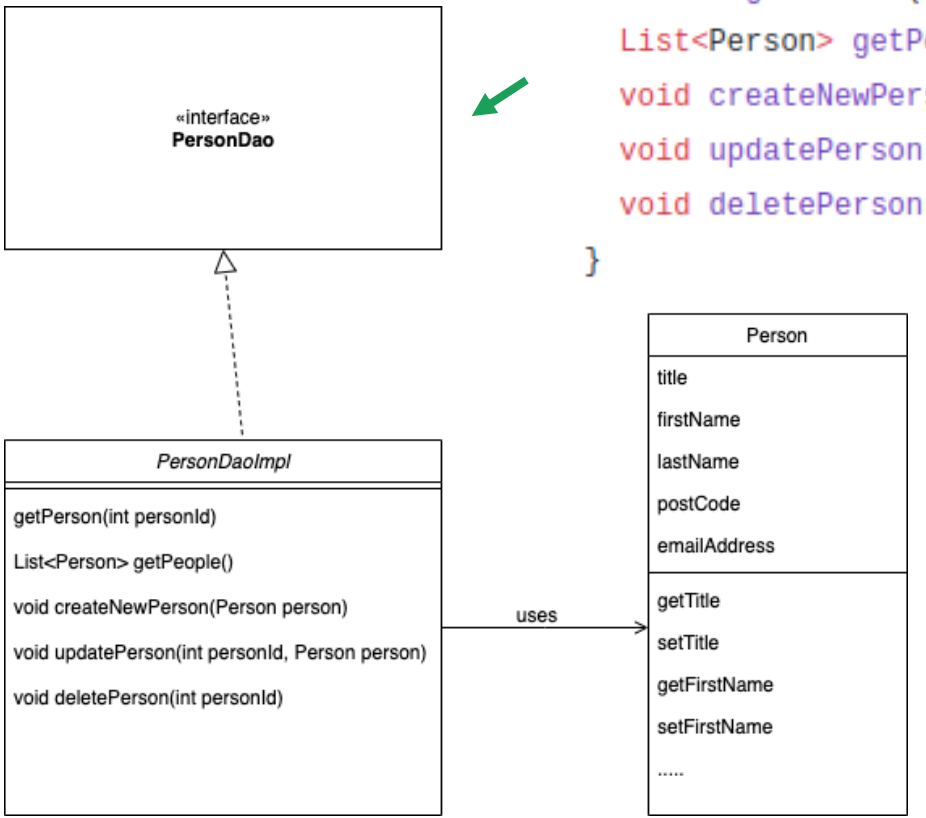
DAO

Usos:

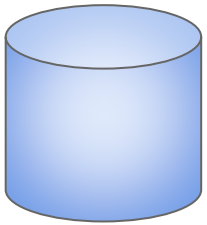
- Abstraerme de las operaciones con la persistencia.
- Quiero guardar mis objetos de negocio/aplicación en dos clientes de BD diferentes.

DAO

```java
public interface PersonDao {
  Person getPerson(int personId);
  List<Person> getPeople();
  void createNewPerson(Person person);
  void updatePerson(int personId, Person person);
  void deletePerson(int personId);
}
```



Accede a BBD. Puede haber distintas Impl.



FACTORY

- Patrón **creacional**.
- Para crear objetos sin saber exactamente la clase del objeto a crear.
- A veces la creación de objetos es compleja.
- La creación de objetos se hace a través una clase (**Factory**).

FACTORY

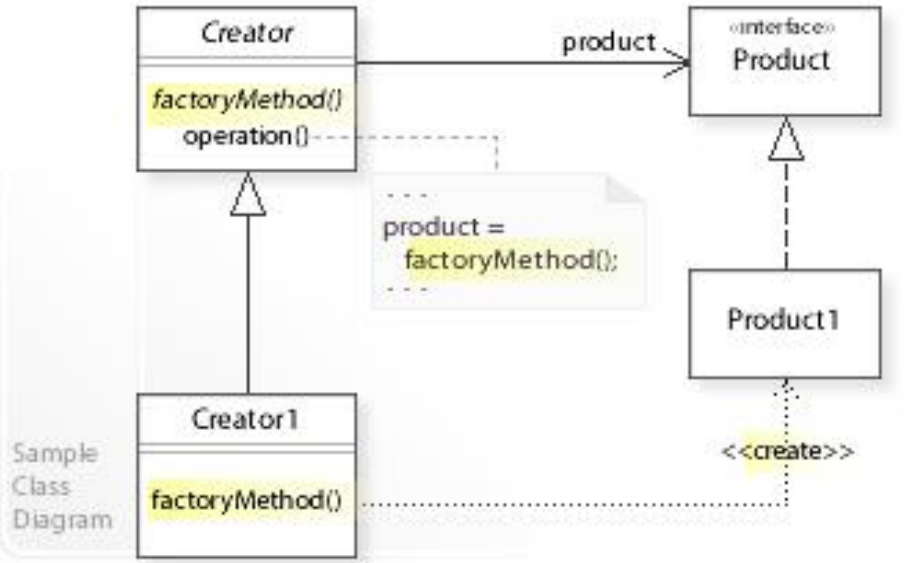
Problema que resuelve:

- ¿Cómo puede ser creado un objeto para que las clases redefinan qué objeto instanciar?
- ¿Cómo puede una clase trasladar la instanciación a sus subclases?

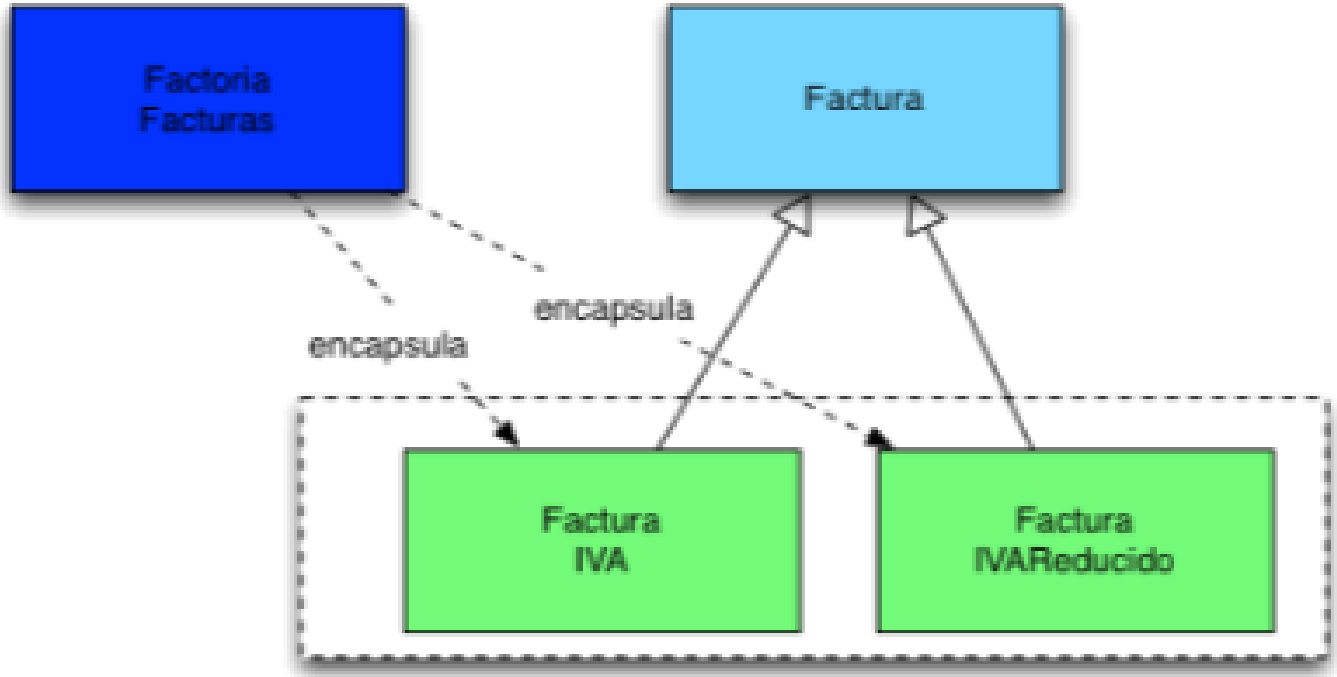
Para afrontar esto:

- Define un método **Factory** para crear un objeto
- Se crea el objeto llamando a ese método **Factory**

FACTORY


Creator
factoryMethod()
operation()
product
«interface»
Product
product =
factoryMethod();
Product1
Creator1
factoryMethod()
<<create>>
Sample
Class
Diagram

FACTORY


Factoria
Facturas
Factura
encapsula
encapsula
Factura
IVA
Factura
IVAReducido

FACTORY

```java
public abstract class Factura {
    private int id;
    private double importe;

    public int getId() {
        return id;
    }

    public void setId(int id){
        this.id = id;
    }

    public double getImporte() {
        return importe;
    }

    public void setImporte(double importe){
        this.importe = importe;
    }

    public abstract double getImporteIva();
}
```

```java
public class FacturaIva extends Factura {
    @Override
    public double getImporteIva() {
        return getImporte() * 1.21;
    }
}
```

```java
public class FacturaIvaReducido extends Factura {
    @Override
    public double getImporteIva() {
        return getImporte() * 1.10;
    }
}
```

FACTORY

```
public class FactoriaFacturas {
    public static Factura getFactura(String tipo) {
        if (tipo.equals("iva")) {
            return new FacturaIva();
        } else if (tipo.equals("ivaReducido")) {
            return new FacturaIvaReducido();
        } else {
            return null;
        }
    }
}
```

```
public class Principal {
    public static void main(String[] args) {
        Factura f = FactoriaFacturas.getFactura("iva");
        f.setId(1);
        f.setImporte(100);
        System.out.println("El importe con IVA es: " + f.getImporteIva());
    }
}
```

ABSTRACT FACTORY

- Factory de Factories.....

MVC

> **MVC** (Modelo-Vista-Controlador) es un patrón en el diseño de software comúnmente utilizado para implementar interfaces de usuario, datos y lógica de control. Enfatiza una separación entre la lógica de negocios y su visualización. Esta "separación de preocupaciones" proporciona una mejor división del trabajo y una mejora de mantenimiento.

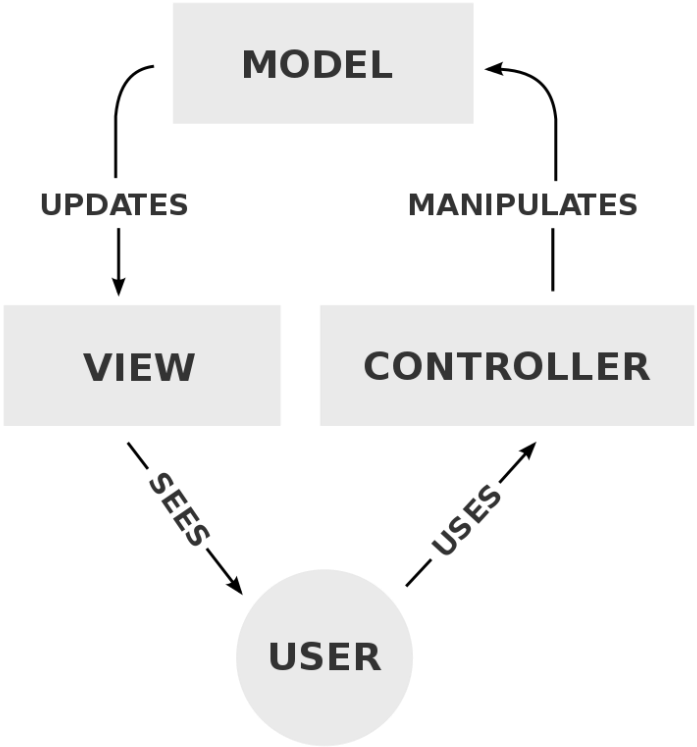
https://developer.mozilla.org/es/docs/Glossary/MVC

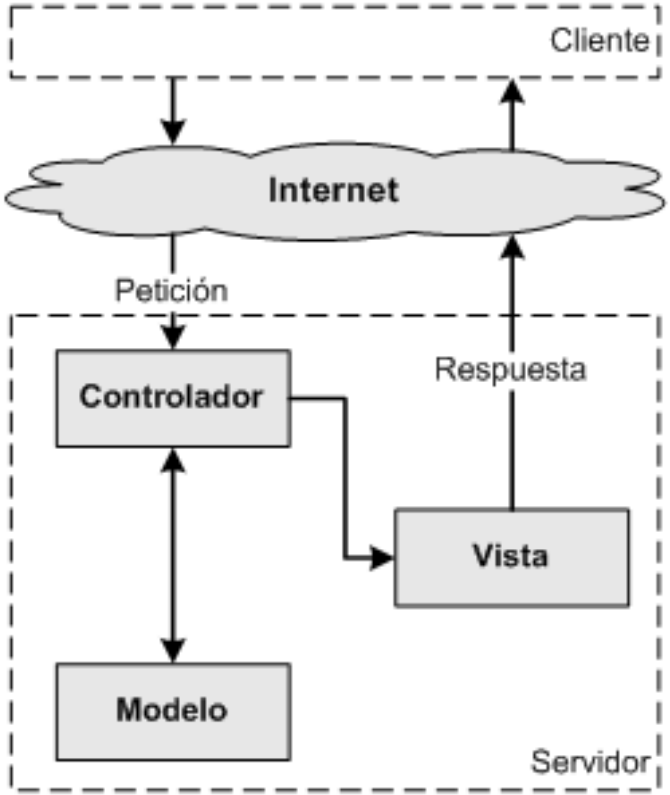
MVC

> Algunos otros patrones de diseño se basan en MVC, como MVVM (Modelo-Vista-modelo de vista), MVP (Modelo-Vista-Presentador) y MVW (Modelo-Vista-Whatever)..

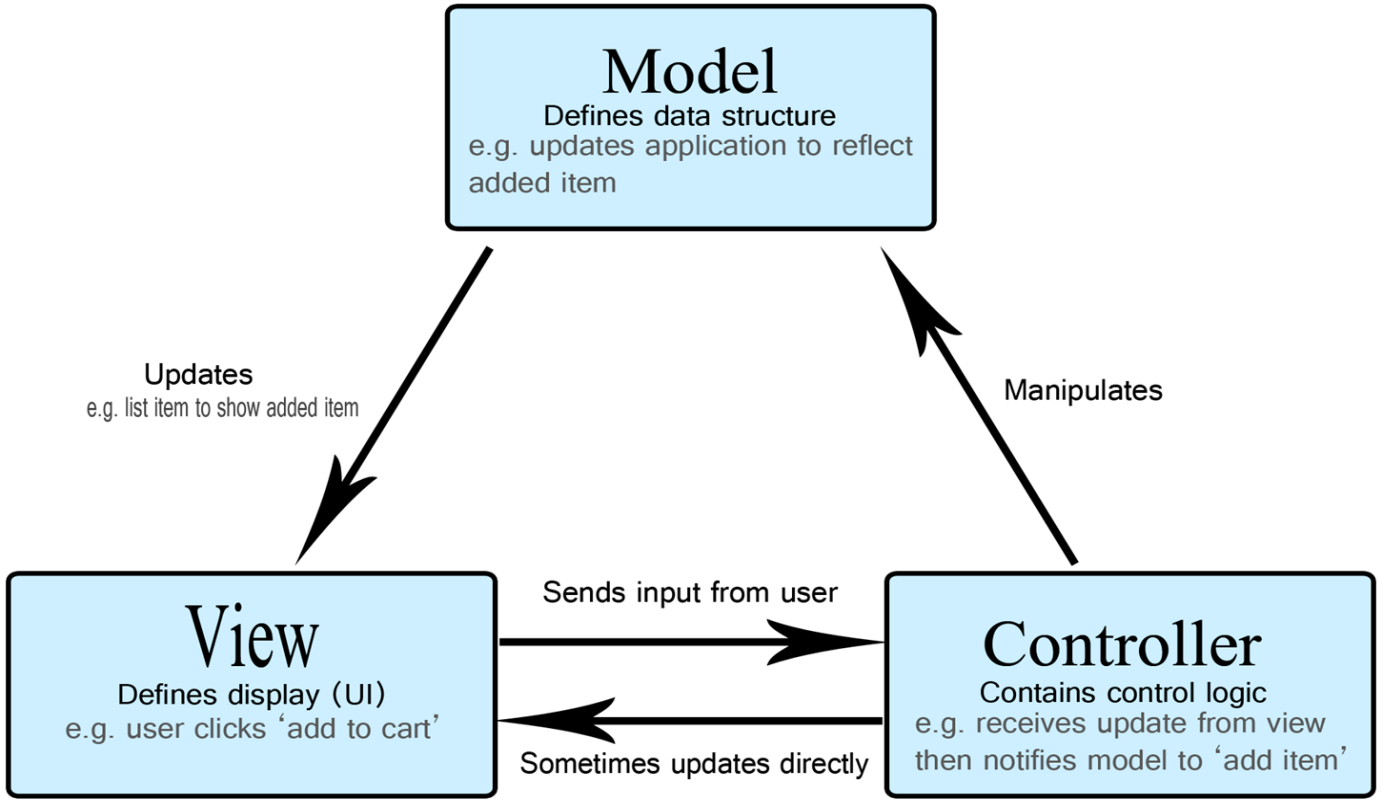
https://developer.mozilla.org/es/docs/Glossary/MVC

MVC

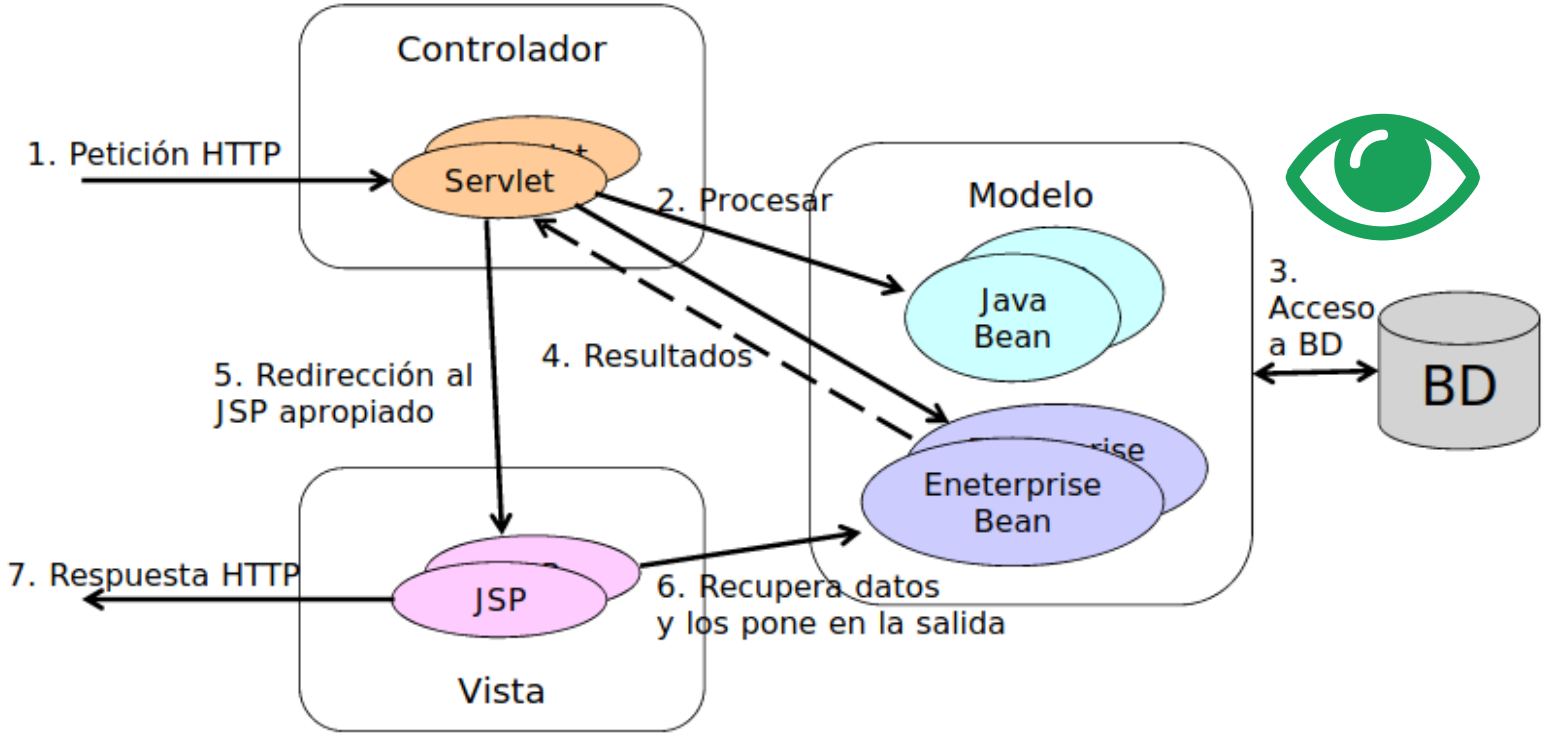

MODEL
UPDATES
MANIPULATES
VIEW
CONTROLLER
SEES
USES
USER


Cliente
Internet
Petición
Respuesta
Controlador
Vista
Modelo
Servidor

MVC


Model
Defines data structure
e.g. updates application to reflect added item
Updates
e.g. list item to show added item
Manipulates
View
Defines display (UI)
e.g. user clicks 'add to cart'
Sends input from user
Sometimes updates directly
Controller
Contains control logic
e.g. receives update from view then notifies model to 'add item'

Arquitectura MVC en Java



DI

- Un objeto (**client**) recibe otros objetos de los que depende (**service**). Ni construye ni encuentra....puede que no sepa cómo crearlos.
- Un objeto **Injector** es el que pasa esas dependencias.
- Frameworks lo propocionan.

DI

- Usado por ejemplo , en la configuración a partir de ficheros de configuración

DI

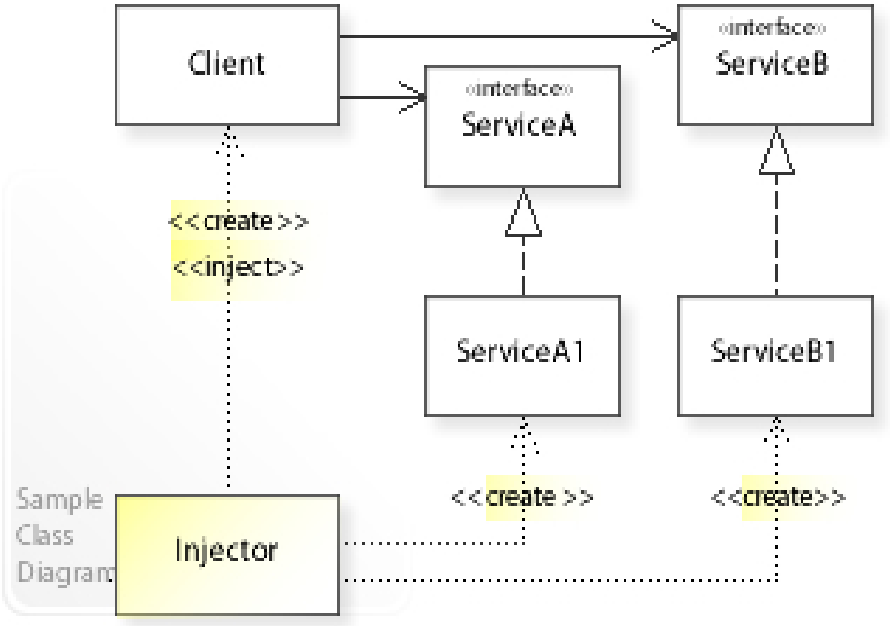
Ventajas:

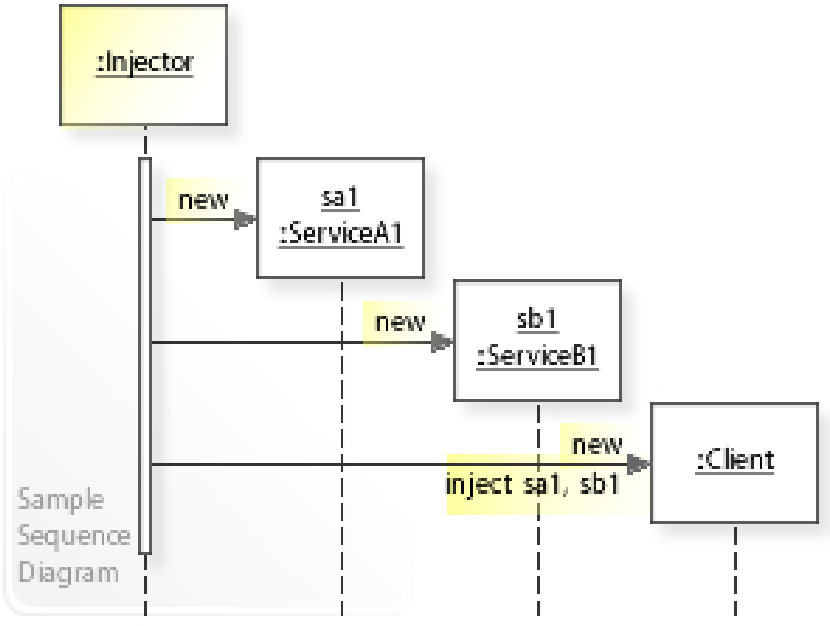
- Desacopla clases y sus dependencias.
- Código más usable, mantenible y flexible.

Desventajas:

- Difícil de detectar errores ya que separa la construcción del comportamiento.
- Requiere más esfuerzo de desarrollo.
- Dependes del framework y puede dificultar automatizaciones del IED

DI


Client
«interface»
ServiceB
«interface»
ServiceA
<<create>>
<<inject>>
ServiceA1
ServiceB1
<<create>>
<<create>>
Injector
Sample
Class
Diagram


:Injector
new
sa1
:ServiceA1
new
sb1
:ServiceB1
new
inject sa1, sb1
:Client
Sample
Sequence
Diagram

DI

- El Injector crea objetos del ServiceA1 y del ServiceB1.
- El Injector crea el objeto Client.
- El Injector inyecta ServiceA1 y ServiceB1 en Client.

Tipos de Inyección de Dependencias

Constructor

Las dependencias se pasan en el constructor.

Setter

Las dependencias se pasan en un método setter.

Interface Injection

Existe un interfaz que inyecta las dependencias a cualquier cliente que se le pase.

DI

Sin Dependencia:

```
public class Client {
    // Internal reference to the service used by this client
    private Service service = null;

    // Constructor
    public Client() {
        // Specify a specific implementation in the constructor instead of using dependency inje
        this.service = new Service();
    }

    // Method within this client that uses the services
    public String greet() {
        return "Hello " + service.getName();
    }
}
```

Constructor Injection:

¿ Y si cambia?

```
public class Client {
    // Internal reference to the service used by this client
    private Service service = null;
    private Service otherService = null;

    // Constructor
    public Client(Service service, Service otherService) {
    if (service == null) {
      throw new IllegalArgumentException("Service must not be null");
        }
    if (otherService == null) {
      throw new IllegalArgumentException("OtherService must not be null");
        }
        this.service = service;
        this.otherService = otherService;
    }
}
```

Setter Injection:

```java
// Set the service to be used by this client
public void setService(Service service) {
    this.service = service;
}

// Set the otherService
public void setOtherService(Service otherService) {
    this.otherService = otherService;
}

// Check the service references of  this client
private void validateState() {
    if (service == null) {
        throw new IllegalStateException("service must be set");
    }
    if (otherService == null) {
        throw new IllegalStateException("otherService must be set");
    }
}

public void doSomething() {
    // Check the state of this client
    validateState();

    // Delegate to the services
    String tmp = service.getName() + " " + otherService.getName();
    System.out.println("I am " + tmp);
}
```

¿ Cuando se usa el servicio?

Interface Injection:

```java
public interface Service {
  String getName();
}


public class ServiceFoo implements Service {
  private final String name;

  public ServiceFoo(String name) {
    this.name = name;
  }

  public void doFooThings() {
    System.out.println("Haciendo cosas de foo");
  }

  @Override
  public String getName() {
    return name;
  }
}
```

```java
public class ServiceInjector {
  public Client getClient() {
    Service service = new ServiceFoo("foo");
    return new Client(service);
  }
}
```

```java
public class Main {
  public static void main(String[] args) {
    ServiceInjector injector = new ServiceInjector();
    Client client = injector.getClient();
    client.greet();
  }
}
```

Las dependencias ignoran a los clientes pero puede tener más.

El método injector se proporciona a través de un interface.

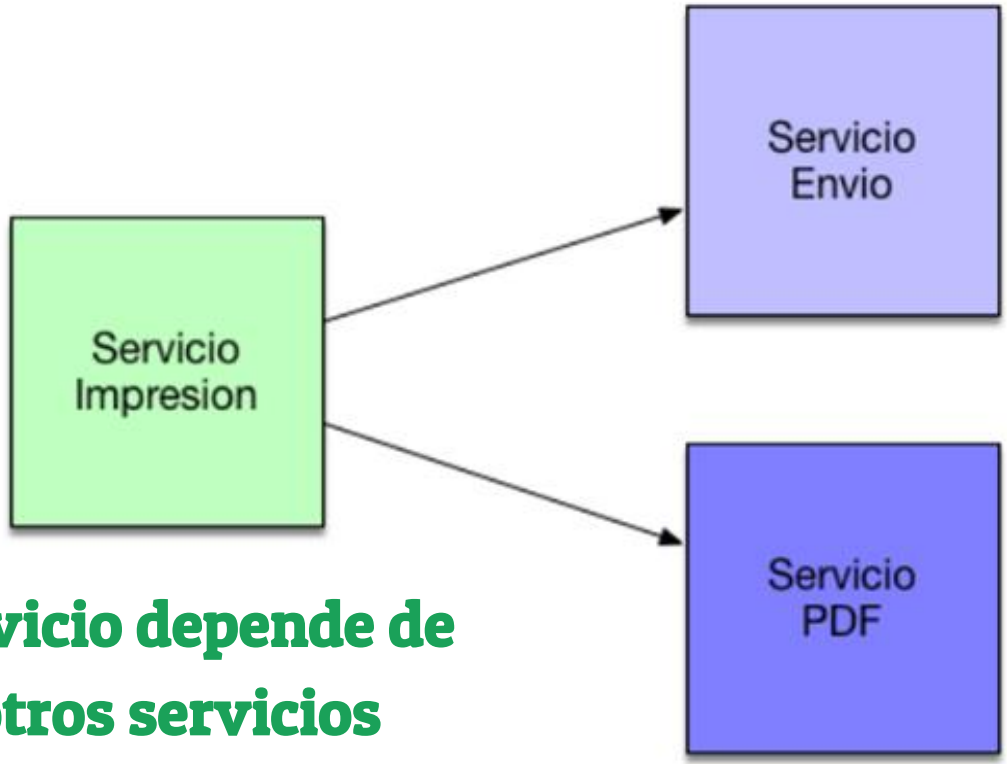
DI

```java
public class ServicioImpresion {
  public void imprimir() {
    System.out.println("enviando el documento a imprimir");
    System.out.println("imprimiendo el documento en formato pdf");
  }
}
```

Sin inyección de dependencias

```java
public class Main {
  public static void main(String[] args) {
    ServicioImpresion miServicio = new ServicioImpresion();
    miServicio.imprimir();
  }
}
```

DI



Servicio depende de otros servicios

División de responsabilidades.

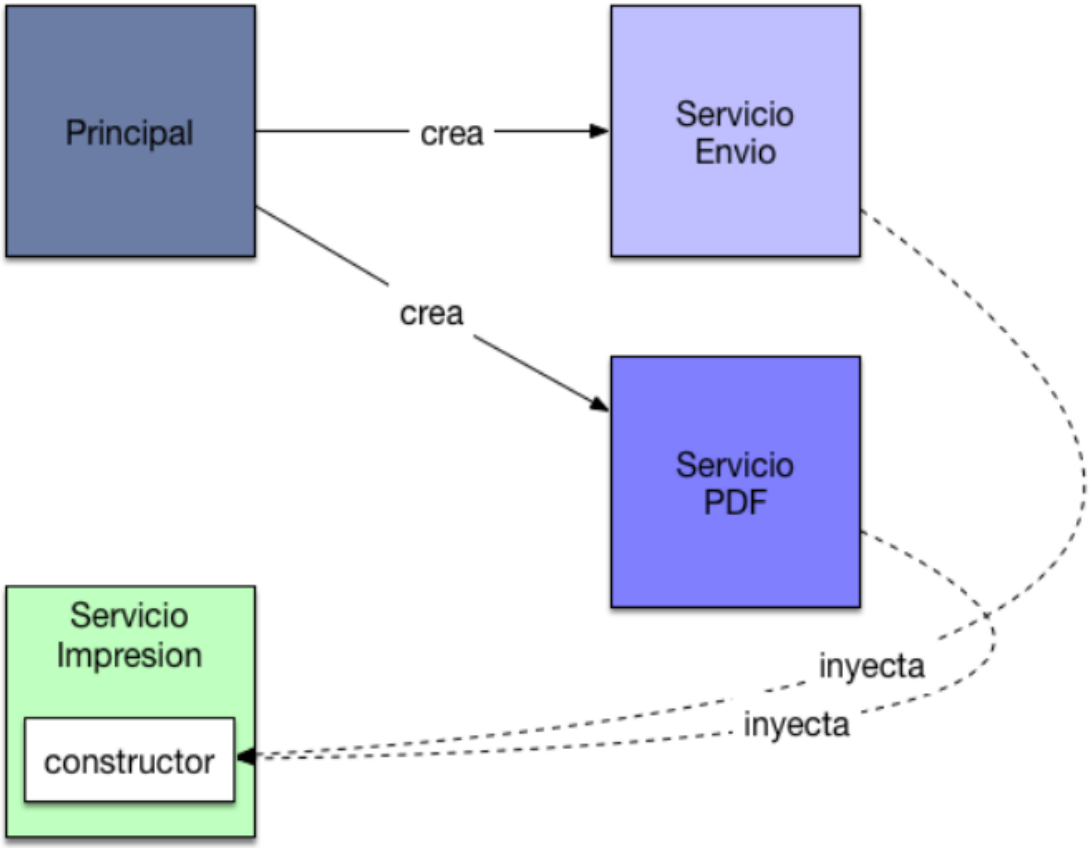
DI

```
public class ServicioImpresion {
  ServicioEnvio servicioA;
  ServicioPDF servicioB;

  public ServicioImpresion() {
    this.servicioA = new ServicioEnvio();
    this.servicioB = new ServicioPDF();
  }

  public void imprimir() {
    servicioA.enviar();
    servicioB.pdf();
  }
}
```

DI

```
public class ServicioEnvio {
  public void enviar() {
    System.out.println("enviando el documento a imprimir");
  }
}


public class ServicioPDF {
  public void pdf() {
    System.out.println("imprimiendo el documento en formato pdf");
  }
}
```

DI


Principal
crea
Servicio Envio
crea
Servicio PDF
Servicio Impresion
constructor
inyecta
inyecta

Inyección en CONSTRUCTOR

```java
public class ServicioImpresion {
  ServicioEnvio servicioA;
  ServicioPDF servicioB;

  public ServicioImpresion(ServicioEnvio servicioA, ServicioPDF servicioB) {
    this.servicioA = servicioA;
    this.servicioB = servicioB;
  }

  public void imprimir() {
    servicioA.enviar();
    servicioB.pdf();
  }
}
```

```java
public class Main {
  public static void main(String[] args) {
    ServicioImpresion miServicio = new ServicioImpresion(
                                                  new ServicioEnvio(),
                                                  new ServicioPDF());

    miServicio.imprimir();
  }
}
```

DI

```
public class ServicioEnvioAspecto extends ServicioEnvio {
  @Override
  public void enviar() {
    System.out.println("haciendo log del correo que vamos a enviar");
    super.enviar();
  }
}
```

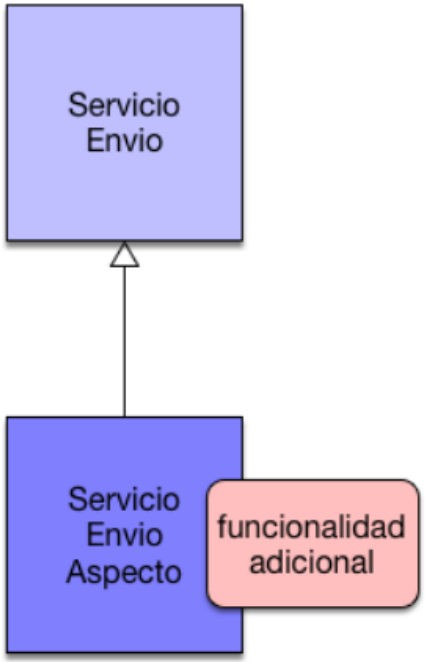
Cambio el comportamiento
extendiendo una de mis clases

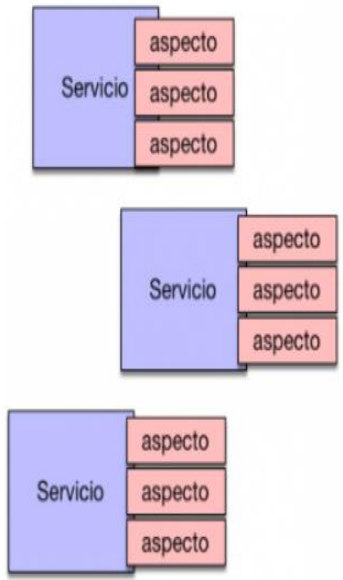
DI

```
public class Main {
  public static void main(String[] args) {
    ServicioImpresion miServicio =
        new ServicioImpresion(new ServicioEnvioAspecto(),
                              new ServicioPDF());
    miServicio.imprimir();
  }
}
```

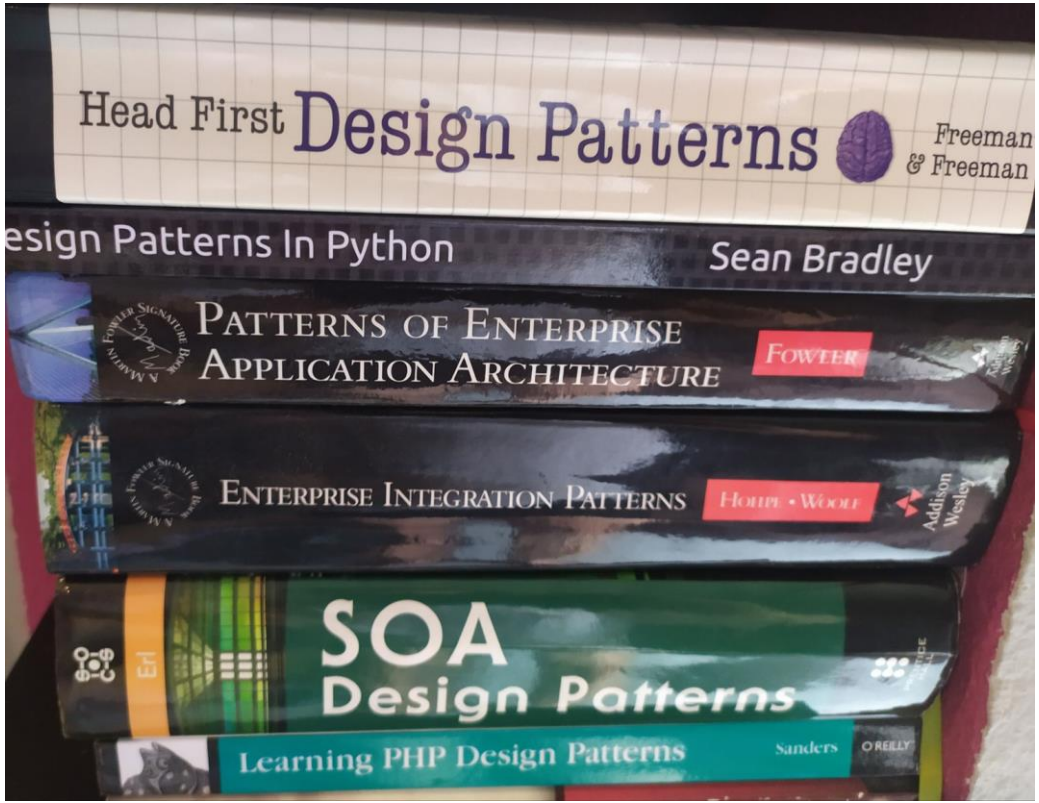
No modifico el servicio.
El Principal inyecta las dependencias.

DI


Servicio
Envio
Servicio
Envio
Aspecto
funcionalidad
adicional


Servicio
aspecto
aspecto
aspecto
Servicio
aspecto
aspecto
aspecto
Servicio
aspecto
aspecto
aspecto

REFERENCIAS



END

jgardur081@g.educaand.es