

# Software Design introduction and Clean code

---



**Paolo Sandoval Noel**  
**Gmail:** [paolosandovaln@gmail.com](mailto:paolosandovaln@gmail.com)  
**Email:** [paolo.sandoval@jalasoft.com](mailto:paolo.sandoval@jalasoft.com)  
**Skype:** paolo.sandoval.jala  
**Github:** jpsandovaln

## AGENDA

### SOLID

- **Single Responsabilitiy principle** – Principio de responsabilidad única.
- **Open Close Principle** – Principio de Abierto / Cerrado.
- **The Liskov Substitution Principle** – Principio de sustitución de Liskov.
- **Dependency Inversión Principle** – Principio de inversión de dependencia.

### Testing

- TDD
- Patterns
- AAA – Arrange, Act, Assert – Organizar / Inicializa, Actuar, Confirmar / Comprobar
- Object Mock

### Static code analysis (Checkstyle + PMD + Findbugs)

### Sistemas extensibles y seguros con genéricos

- Tipos genéricos
- Restricciones de tipo
- Inferencia de tipos
- Varianza, Covarianza y Contra varianza
- Diferencia entre `<T>` y `<?>`

### Procesamiento de archivos

- **Reading from files** – leyendo de archivos
- **Writing to files** – Escribir en archivos
- **Data parsing** – Análisis de datos

### Características de java9 – Java10 – Java11

- **Java9** – Jshell
- **Java9** – Colecciones, factory methods
- **Java10** – inferencia de tipos para variables
- **Java11** – nueva sintaxis en expresiones lambda

## AGENDA

### Wrappers

#### Mejorar mantenimiento de sus estrategias de código

- Patrón estrategia
- Patrón comando
- Eliminar jerarquías de clase: Composición
- Template vs Estrategia
- Patrón plantilla – template
- Inyección de dependencias

### Creación de objetos de encapsulación

- Objectos Caching
- Singleton
- Factory method
- Factory object

### Iterables e iteraciones

- Iteradores
- Patrón iterador
- Iteración internas
- Iteraciones externas
- Paton visitor

### PROYECTO DE CLASE

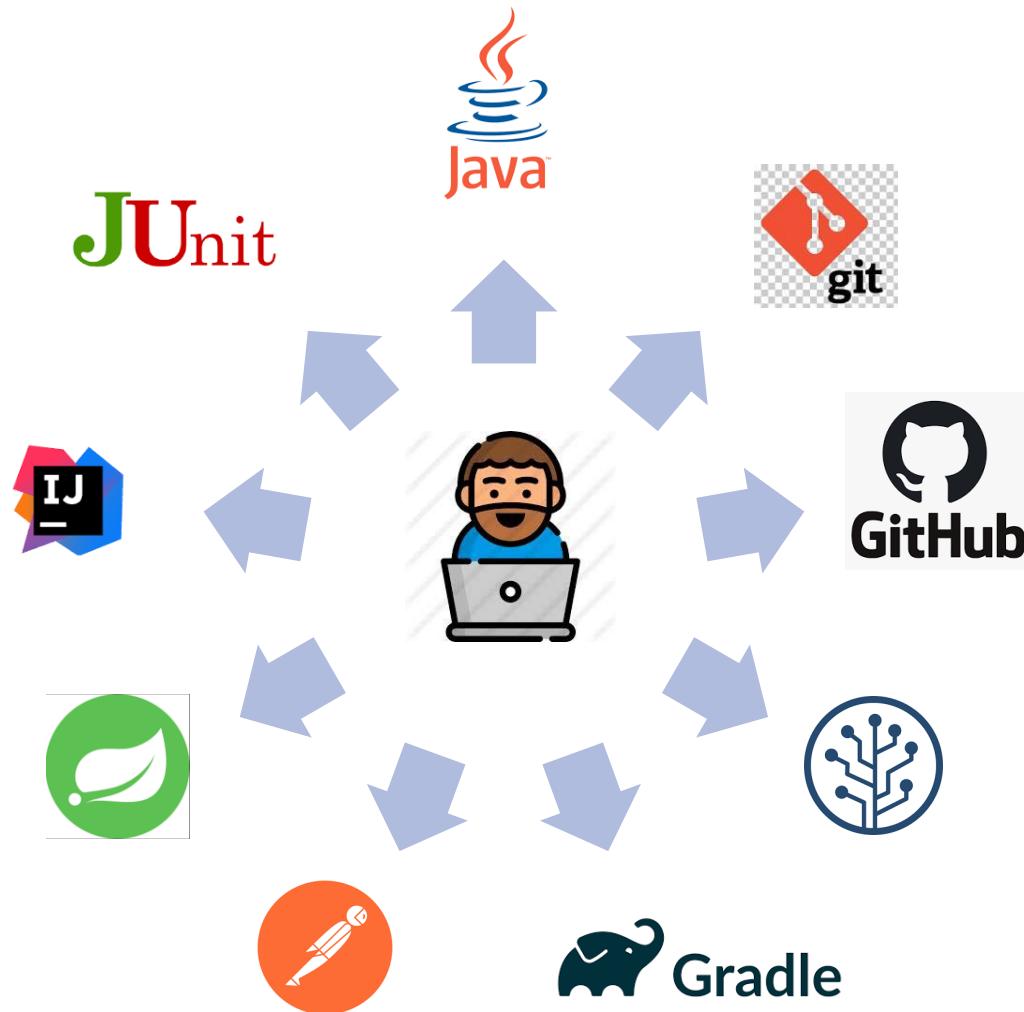
Implementar una aplicación según se vaya avanzando con los temas establecidos para este curso.

### PROYECTO FINAL

Implementar un proyecto donde los participantes puedan aplicar lo que aprendieron (patrones de diseño).

## Tecnologías y Herramientas

- Java
- Git
- GitHub
- SourceTree
- Gradle
- Postman
- Spring
- IntelliJ Idea
- Junit



SourceTree <https://www.sourcetreeapp.com/>

IntelliJ Idea Community <https://www.jetbrains.com/es-es/idea/download/#section=windows>

Postman <https://www.postman.com/downloads/>

- Asistencia 20%
- Prácticas 30%
- Proyecto 50%

**Nota de aprobación 70%**

# Práctica 1

---

Realice un diagrama de clases (solo cabeceras) que sirva como modelo para la realización del proyecto.  
(buscar una determinado objeto dentro de un video – machine learning)

**Objetivo:** medir el nivel de abstracción del problemas, manejo de SOLID y  
utilización de patrones de diseño que conoce.

Video Path:

Browse

Word:

Neural network Model

VGG16

Percentage

0

Search

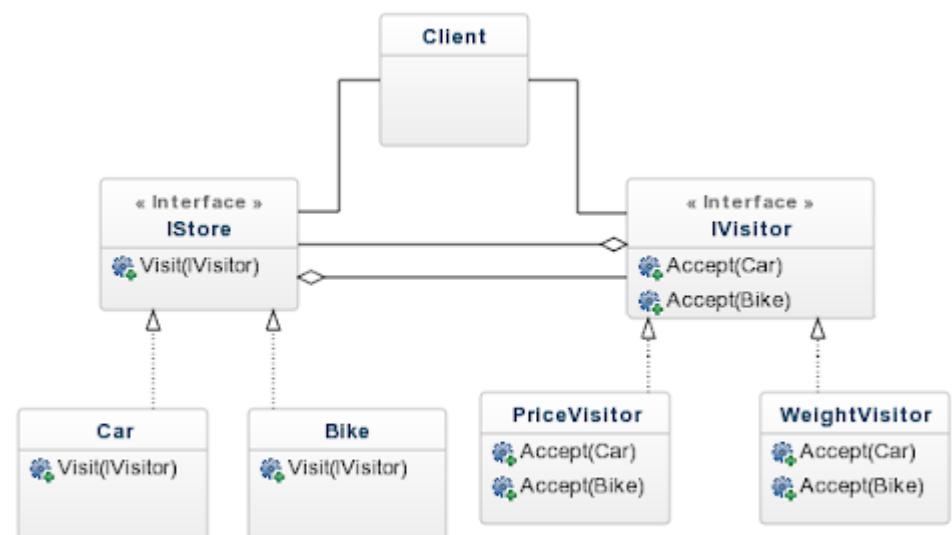
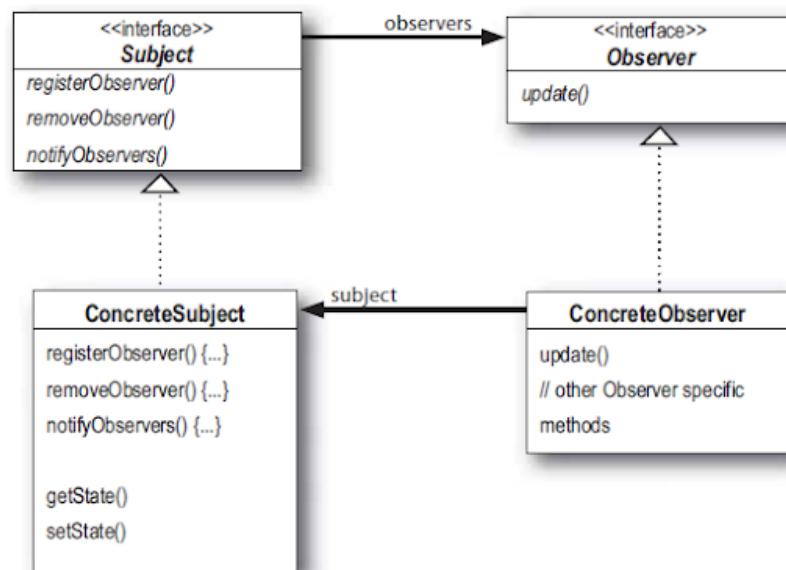
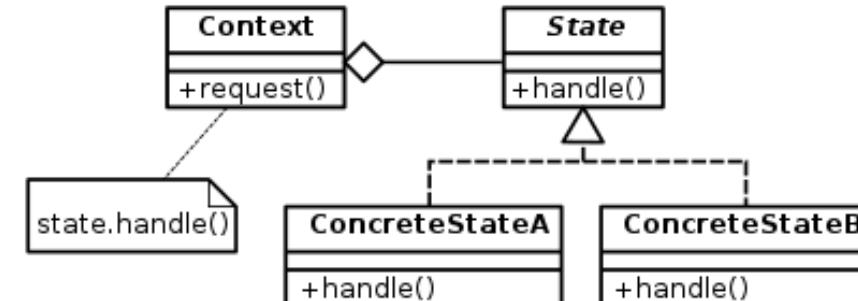
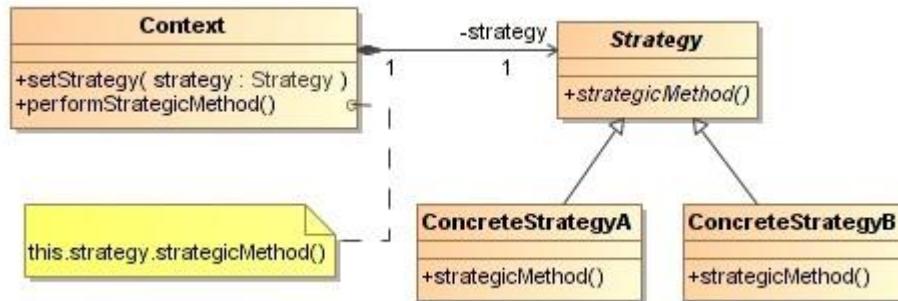
Algorithm	Word	Percentage	Second	Time

Activar Windows

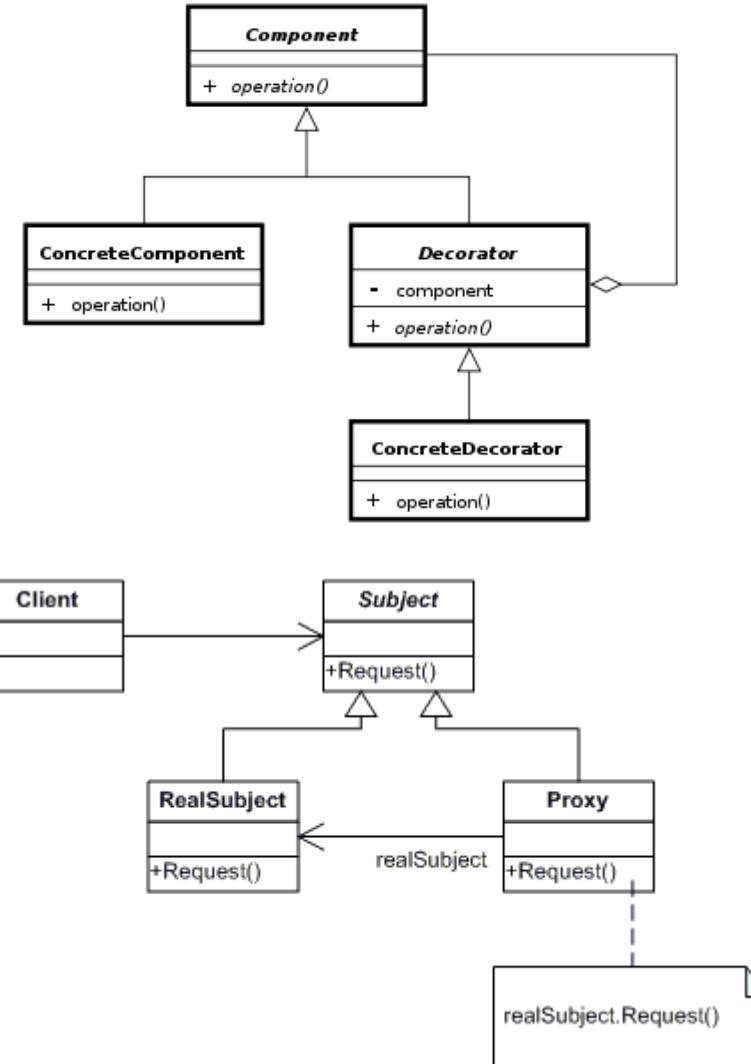
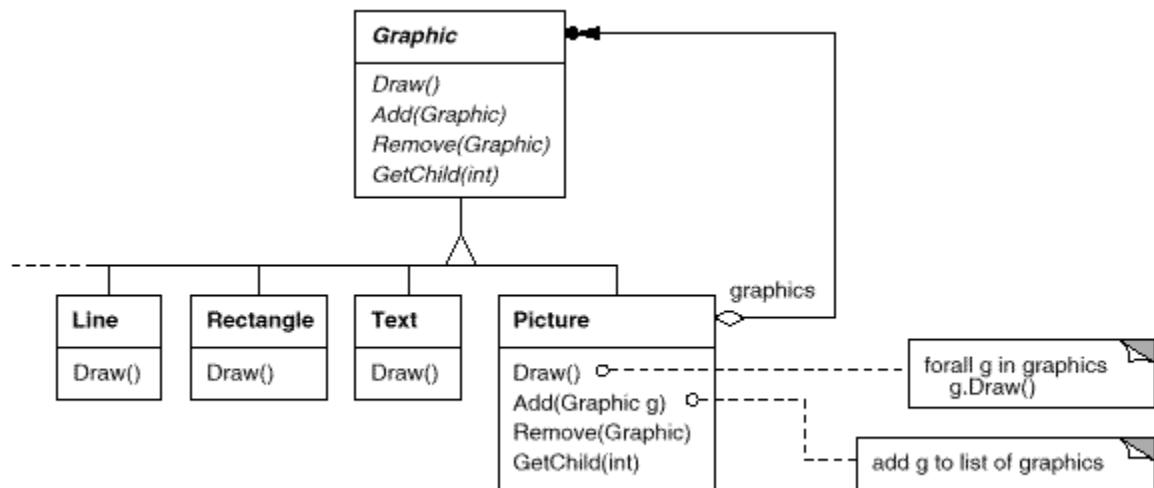
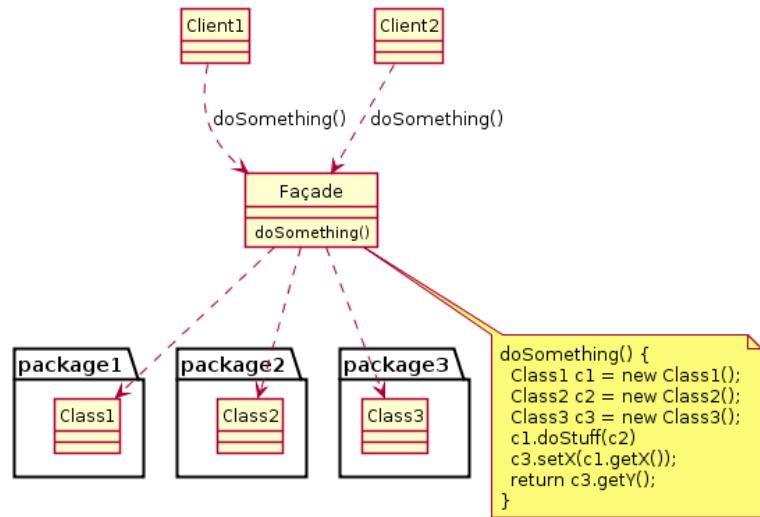
Ve a Configuración para activar Windows.

Show Image

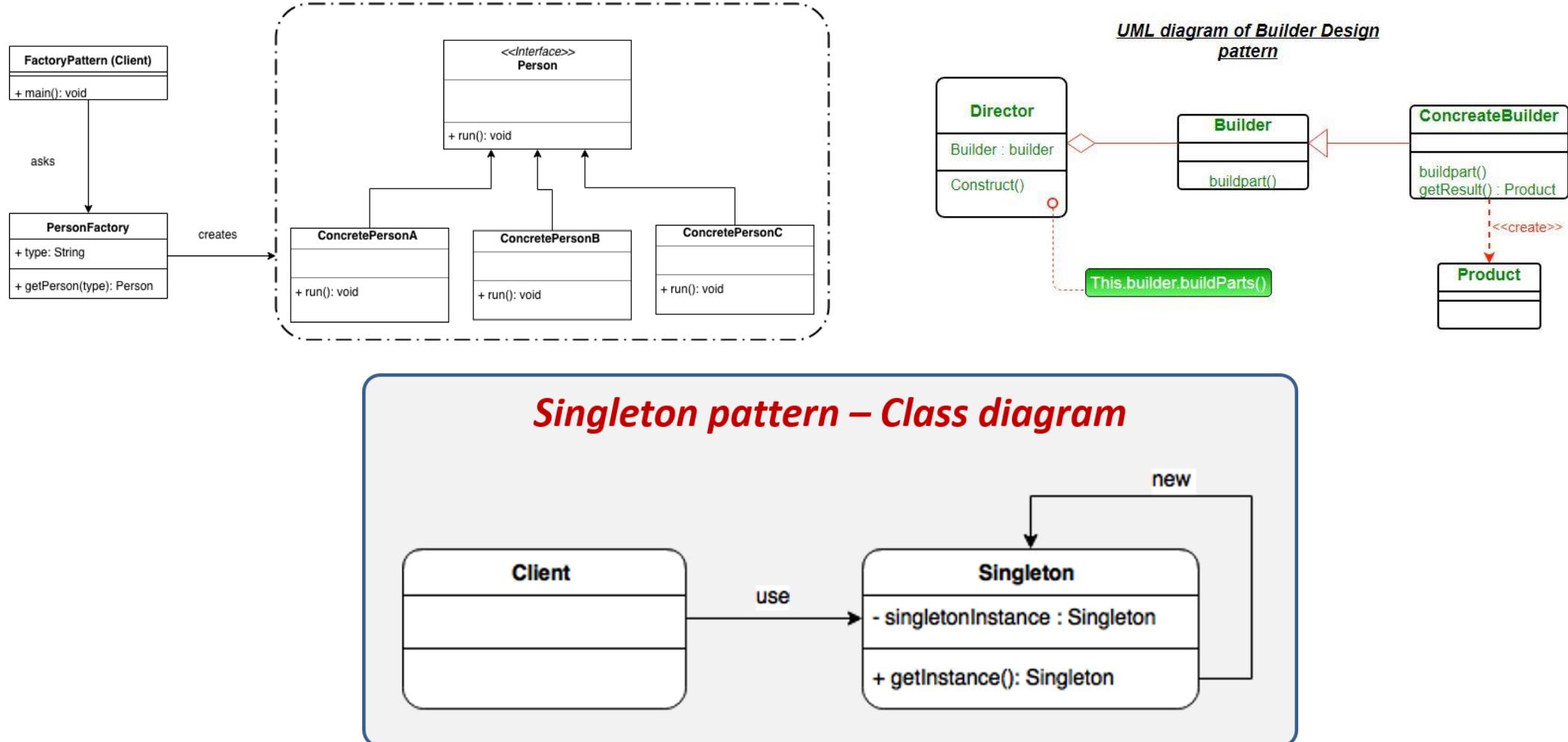
# Patrones



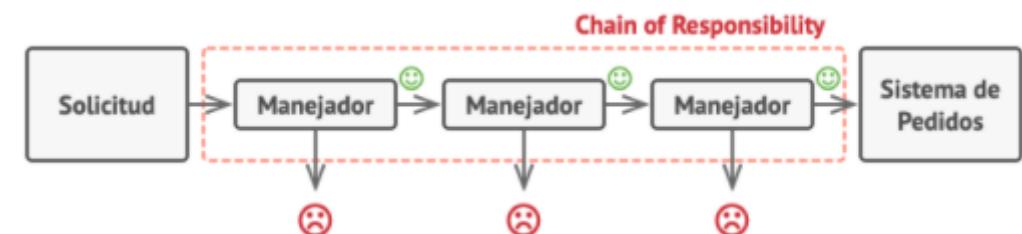
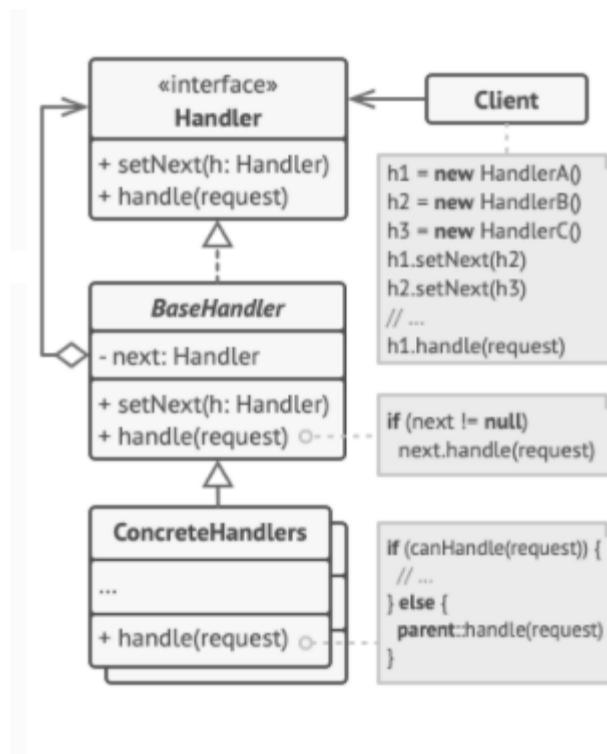
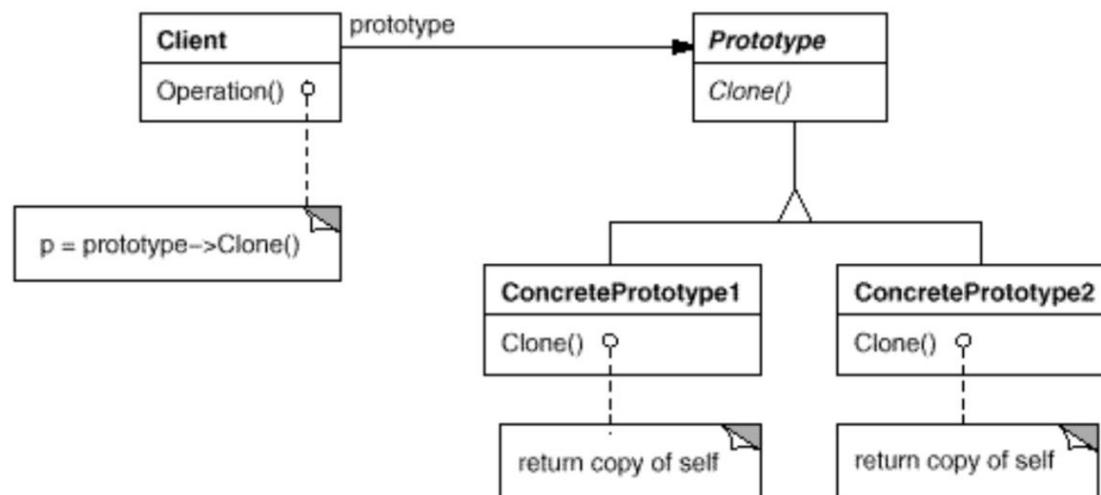
# Patrones



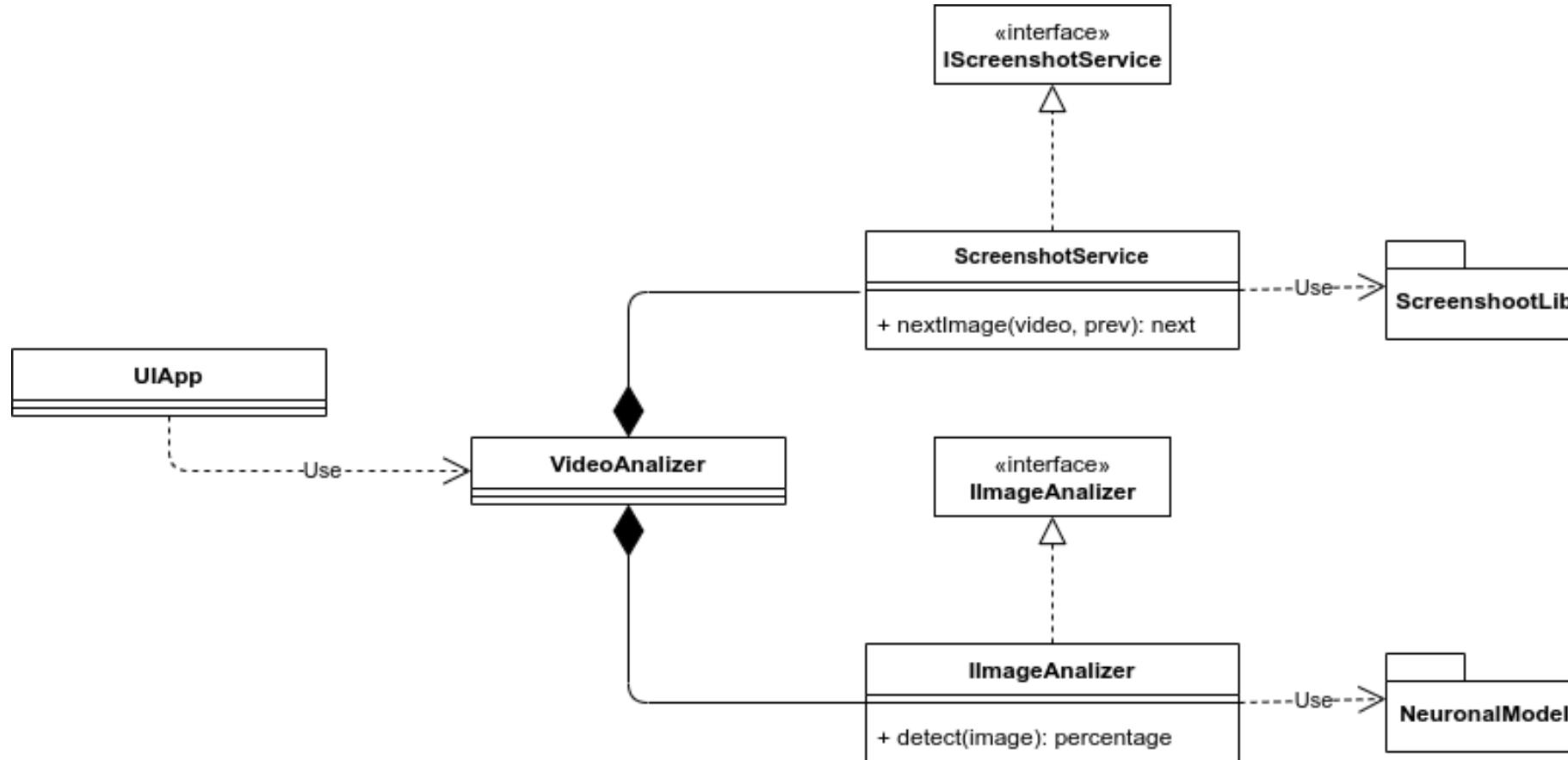
# Patrones



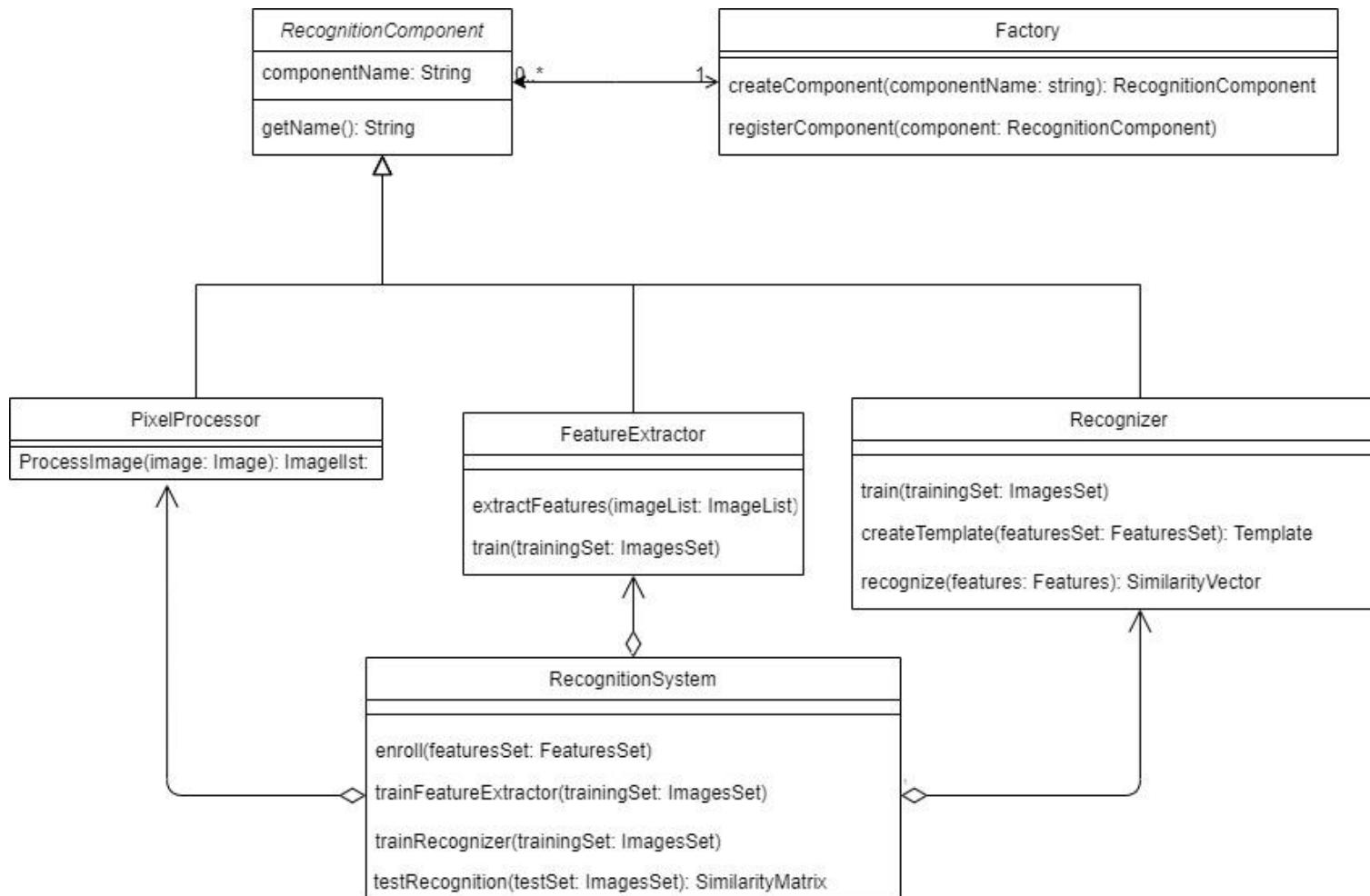
# Patrones



# Práctica 1 - Entregadas

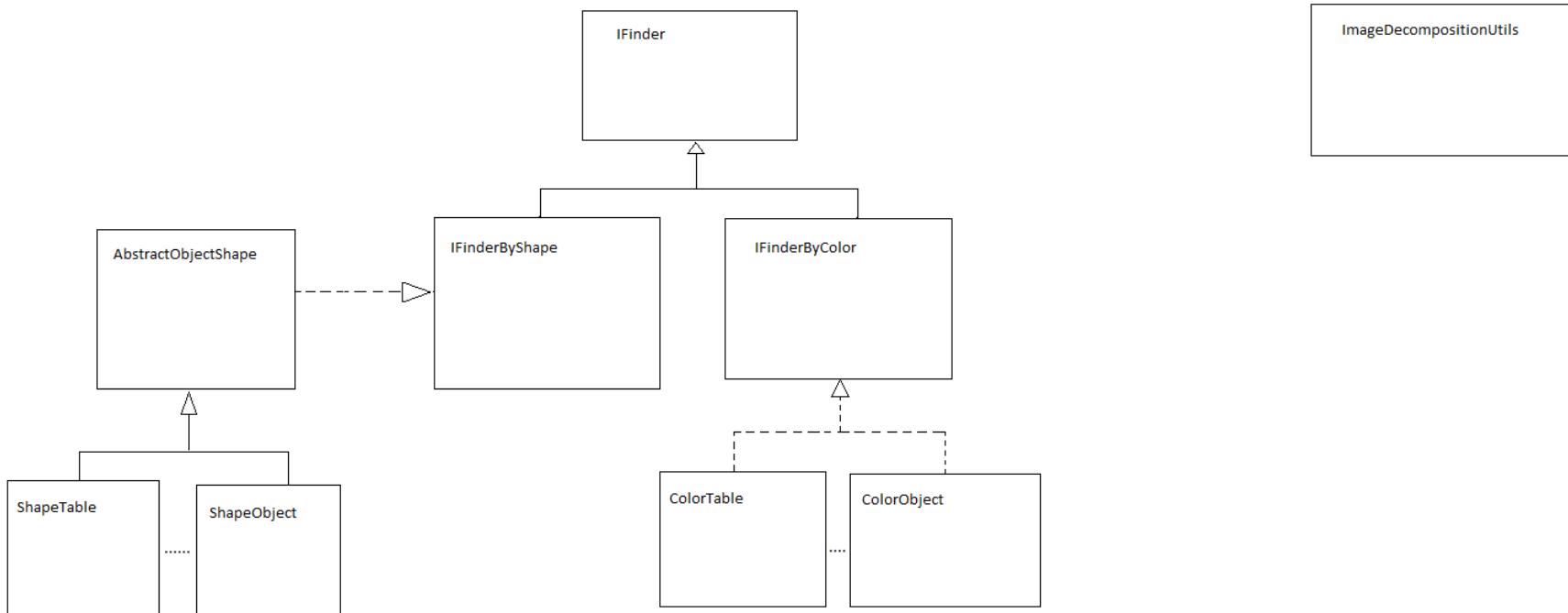


# Práctica 1



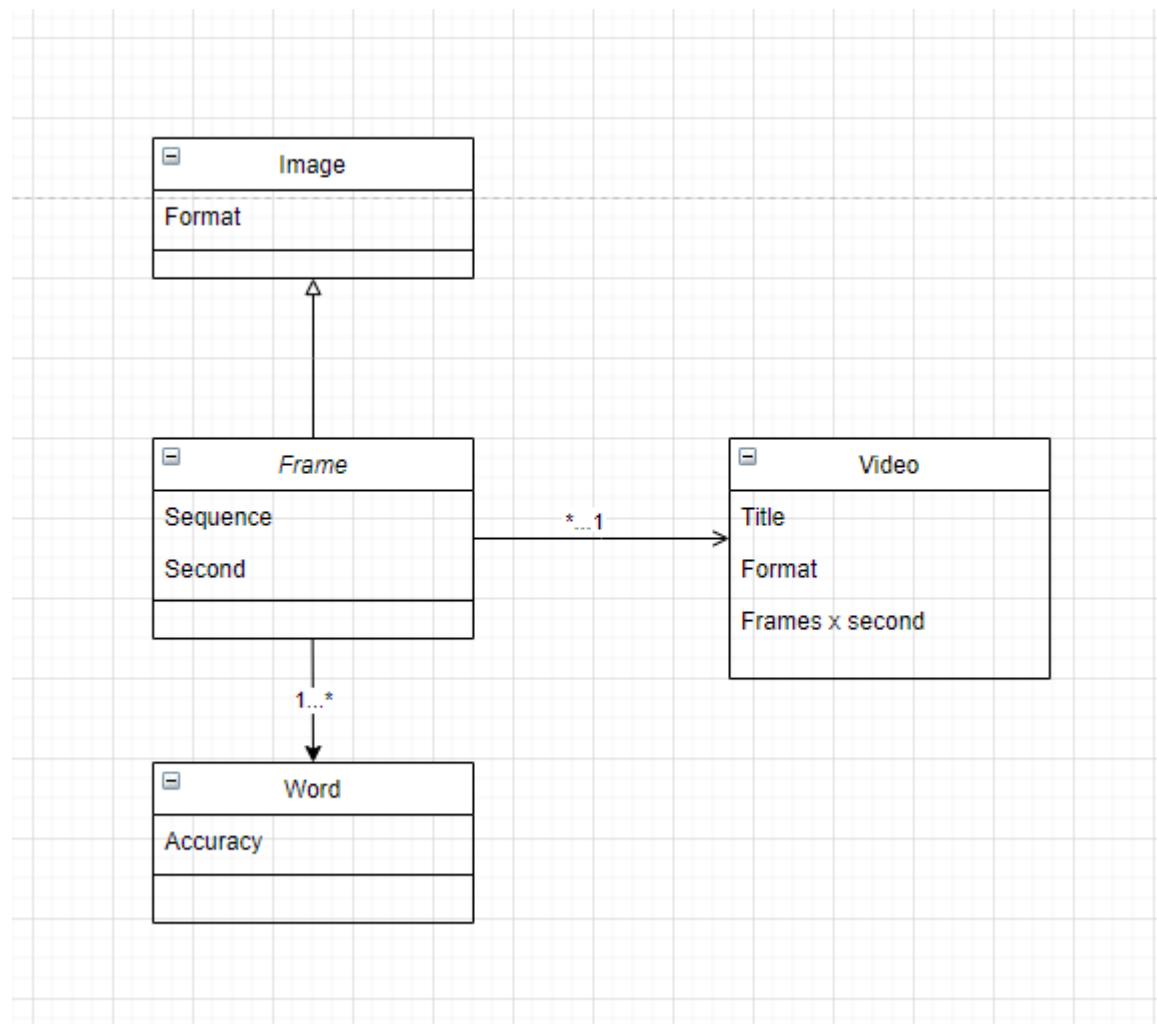
# Práctica 1

---



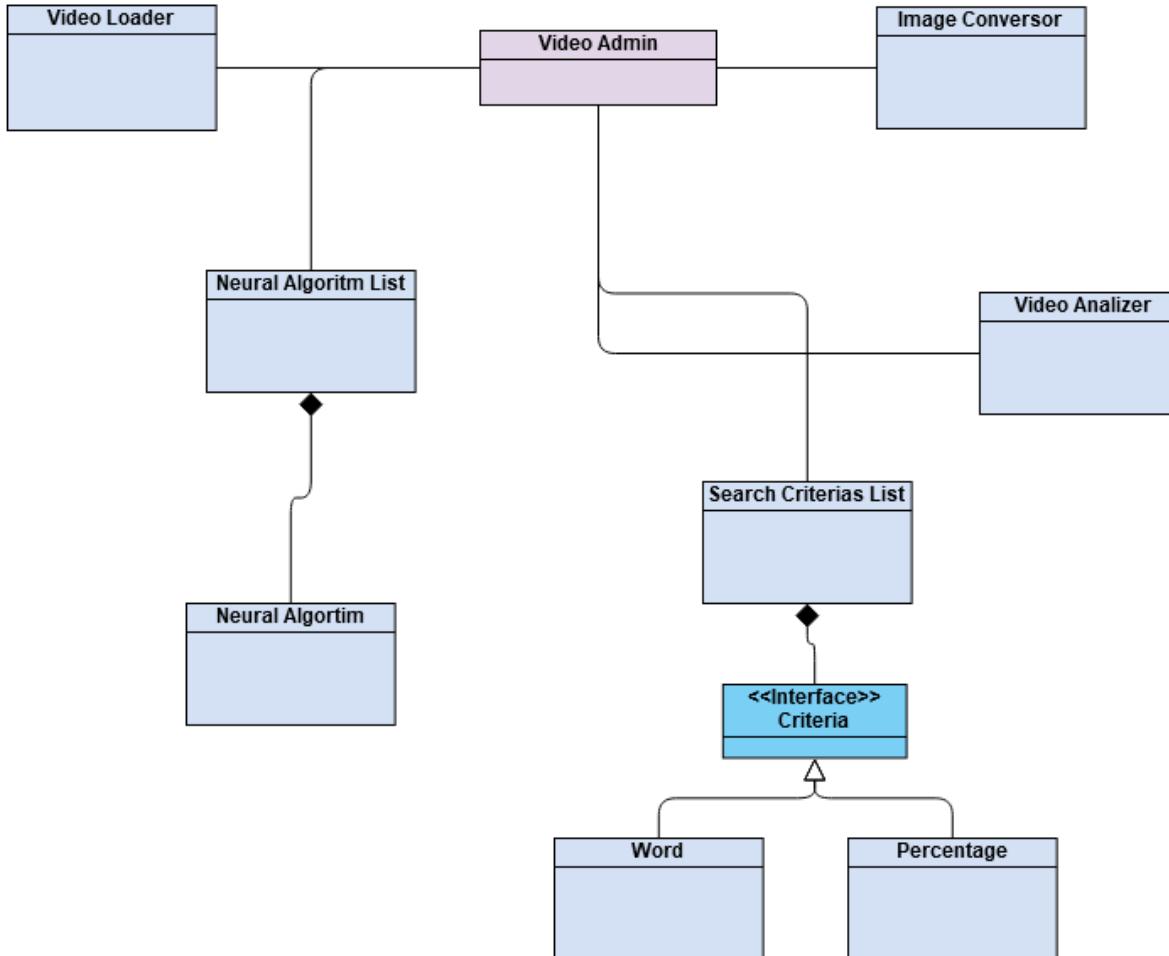
# Práctica 1

---

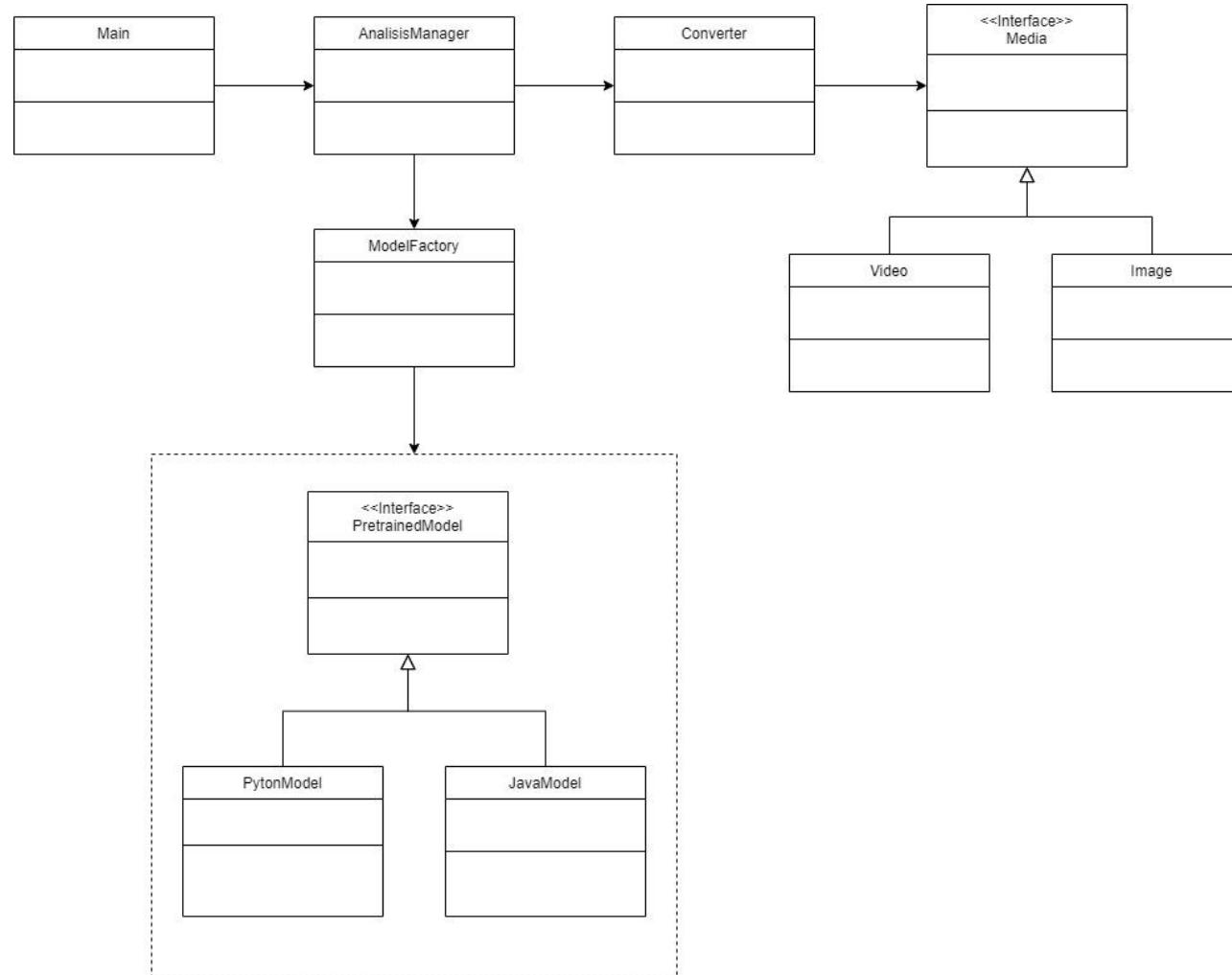


# Práctica 1

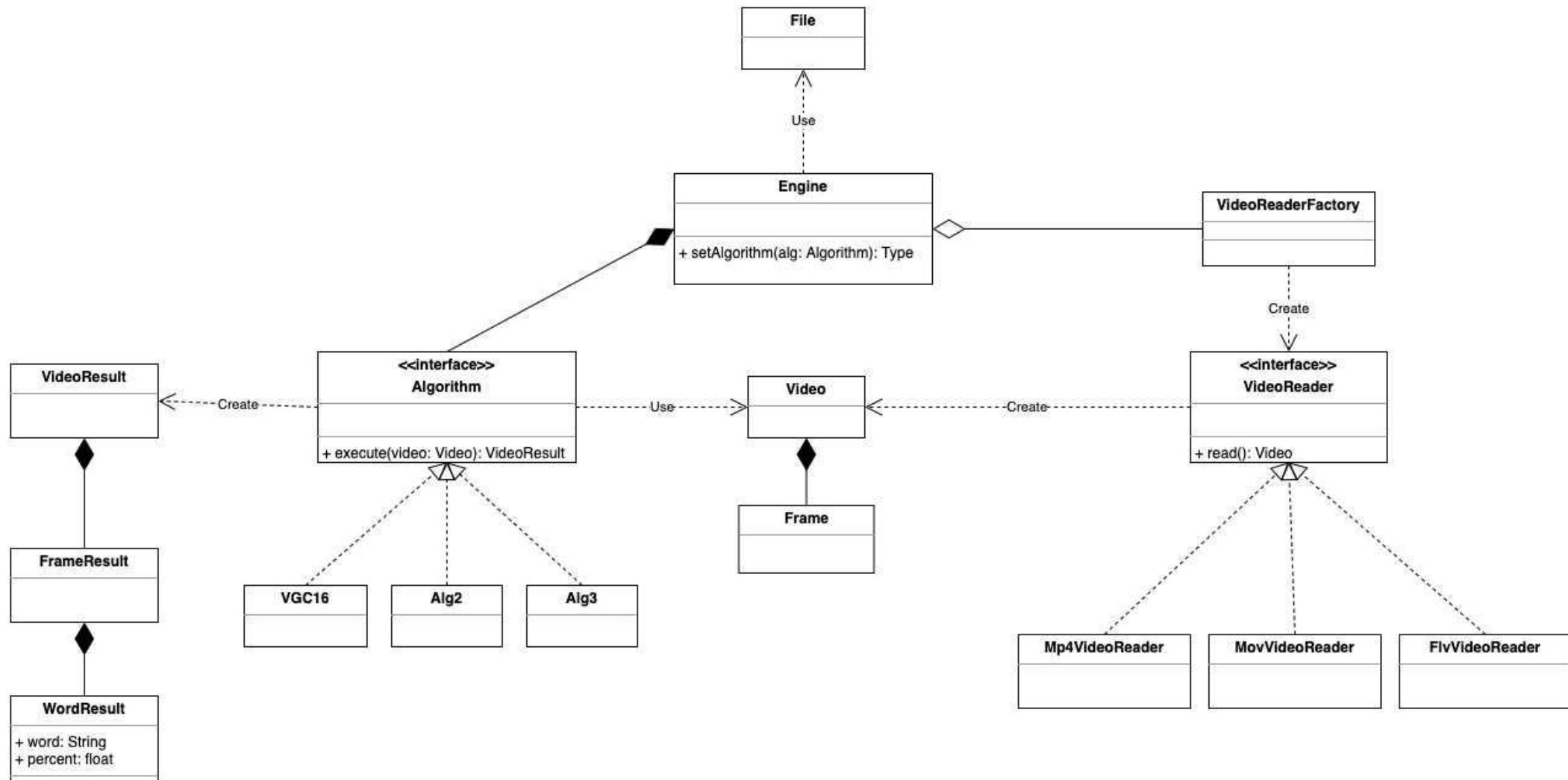
---



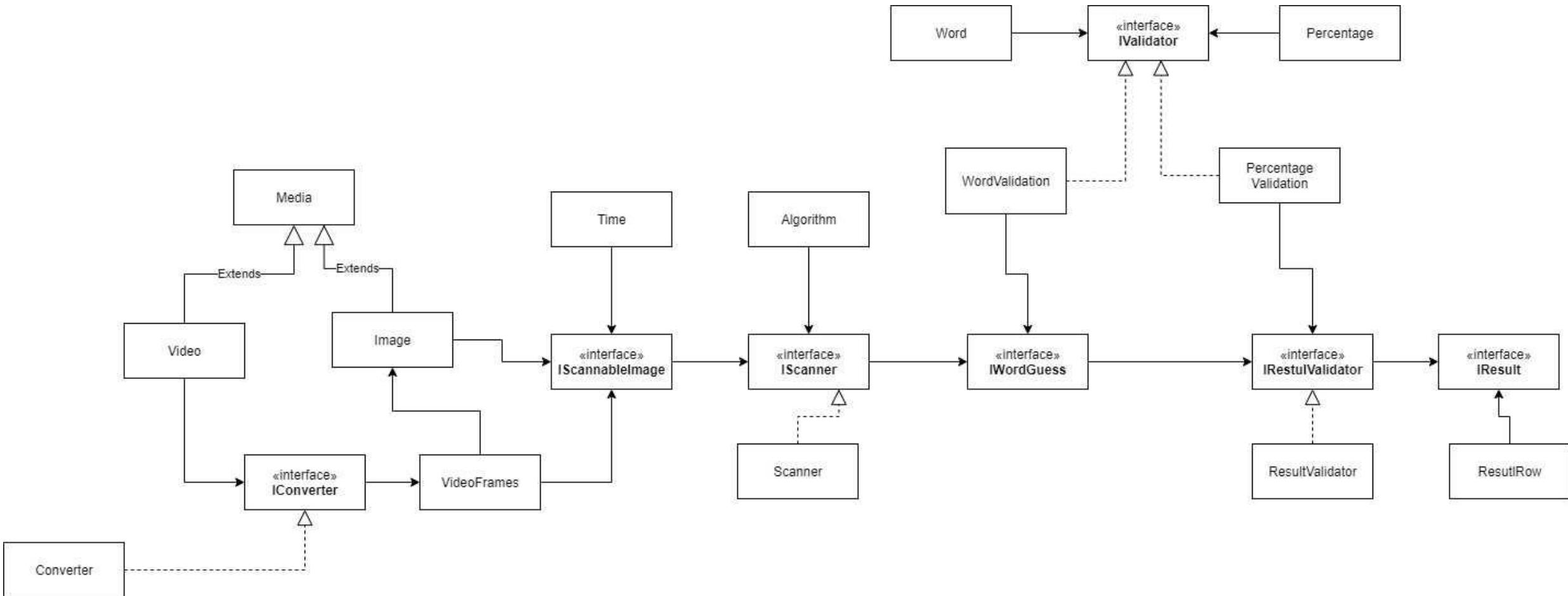
# Práctica 1



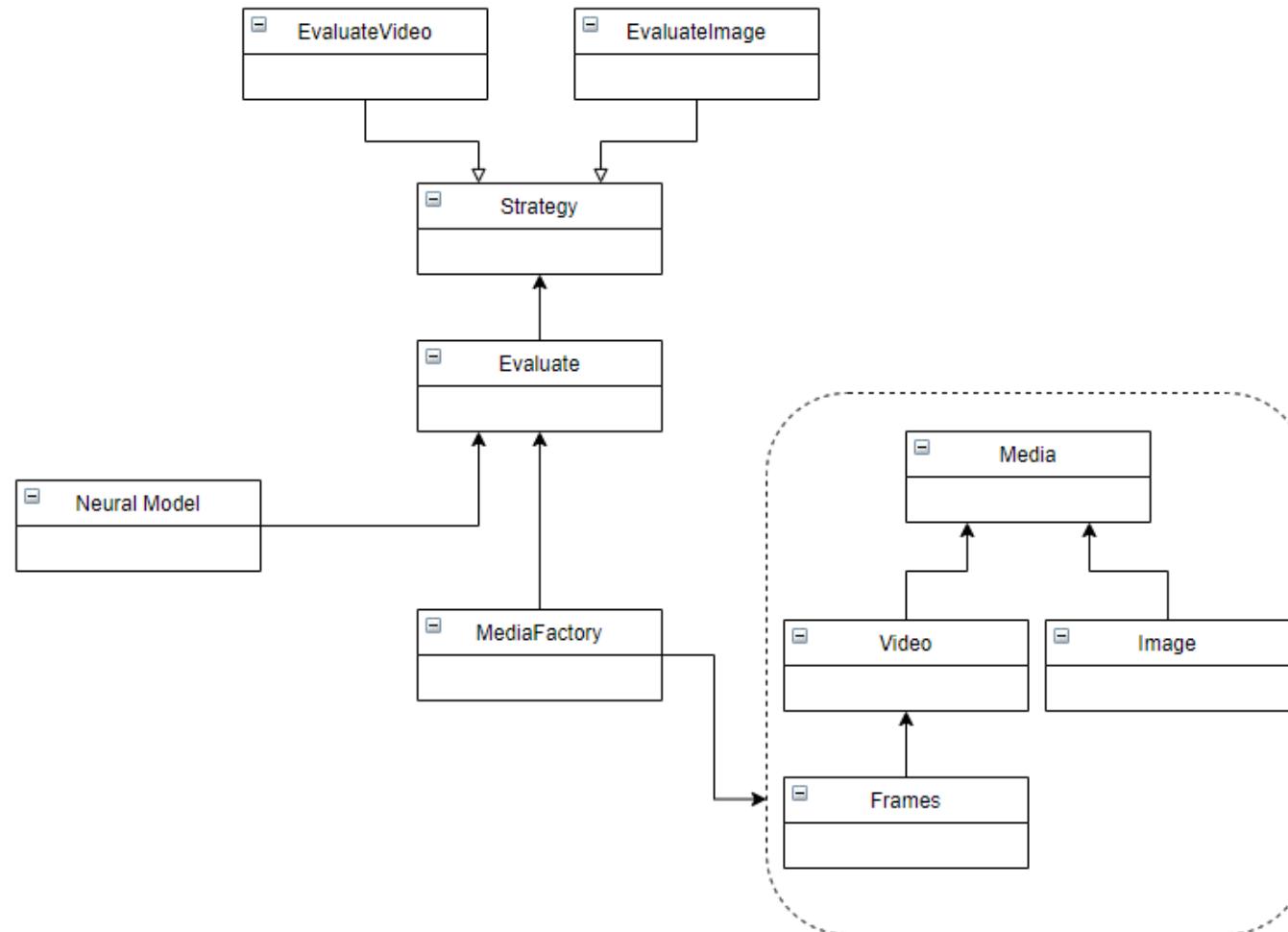
# Práctica 1



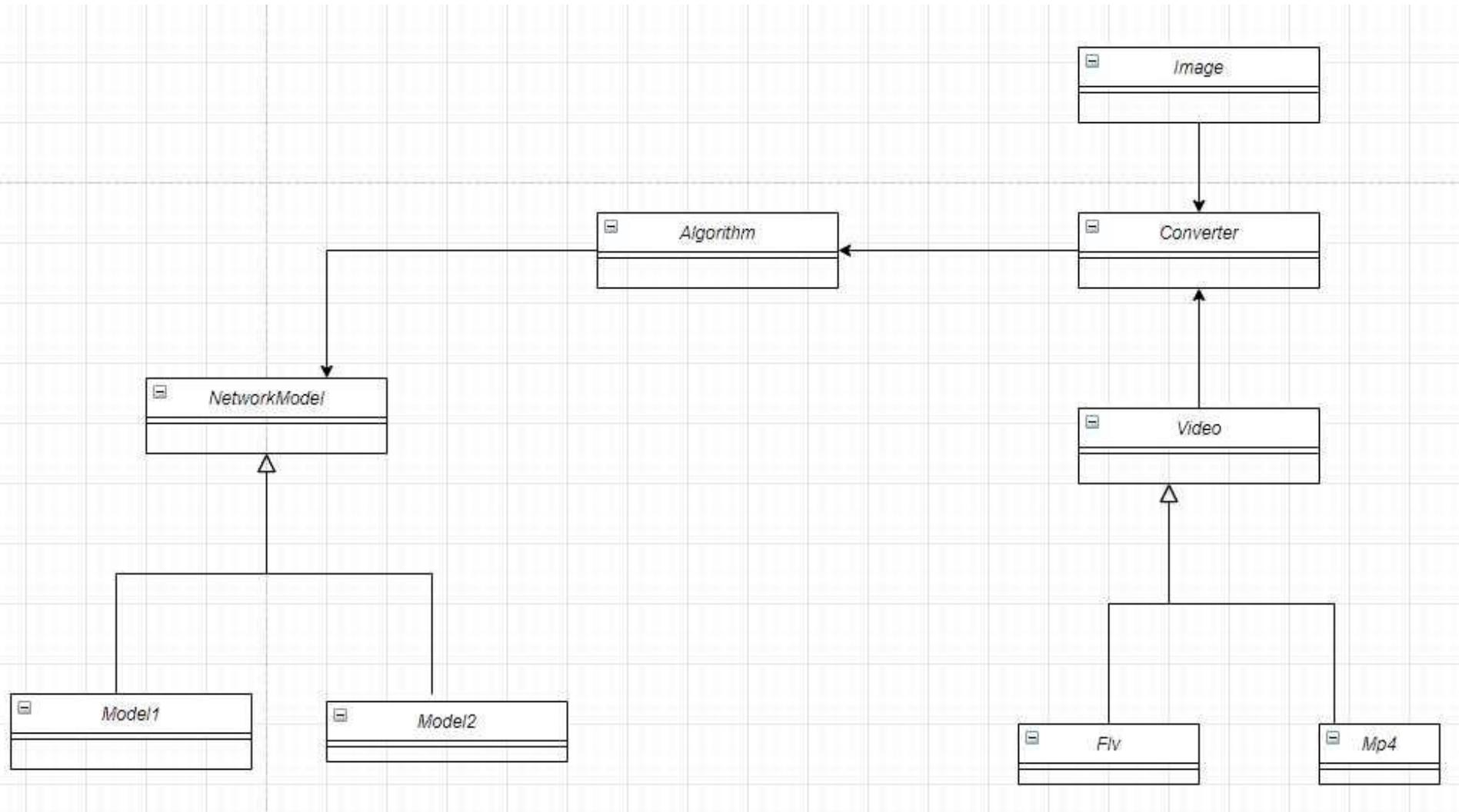
# Práctica 1



# Práctica 1



# Práctica 1



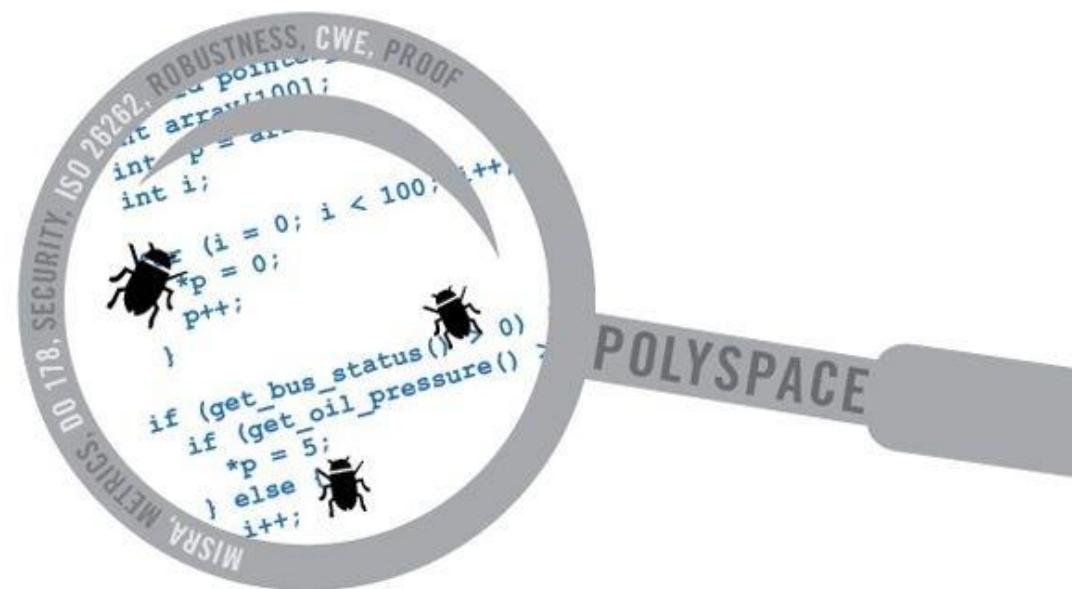
# Análisis de código estático

---



## ¿Qué es el análisis de código estático?

El análisis de código estático significa que no hay necesidad de ejecutar el código probado, y la corrección del programa se verifica analizando o verificando la sintaxis, estructura, proceso, interfaz, etc. del programa fuente, y descubriendo los errores y defectos del código oculto, como la falta de coincidencia y la ambigüedad de los parámetros. Las declaraciones anidadas, recursividad incorrecta, cálculos ilegales, posibles referencias de puntero nulo, etc.



## Ventajas de las herramientas de análisis de código estático

1. Ayuda a localizar rápidamente el código que oculta errores y defectos.
2. Ayuda a centrarse más en analizar y resolver defectos de diseño de código.
3. Reducir significativamente el tiempo dedicado a la inspección de código línea por línea.



PMD (Programming Mistake Detector) es un analizador de código fuente estático de código abierto que informa sobre los problemas encontrados dentro del código fuente de una aplicación.

1. PMD incluye conjunto de reglas por defecto y admite la capacidad de escribir reglas personalizadas.
2. Analiza archivos en los lenguajes: JavaScript, java, PLSQL, XML, etc.



PMD puede detectar fallas o posibles fallas en el código fuente, tales como:

- **Errores posibles :** bloques vacíos de sentencias try / catch/finally/switch
- **Código muerto:** variables locales, parámetros y métodos privados no usados.
- Declaraciones if / while vacías.
- Expresiones demasiado complicadas: declaraciones if innecesarias, bucles for que podrían ser bucles while.
- **Código subóptimo:** uso de String / StringBuffer inútil.
- Clases con medidas de complejidad ciclomática alta.
- **Código duplicado:** el código copiado/pegado puede significar errores copiados/pegados y reduce la capacidad de mantenimiento.

## Checkstyle

Es una herramienta de análisis de código estático que se utiliza en el desarrollo de software para comprobar si el código fuente de Java cumple con las reglas de codificación.

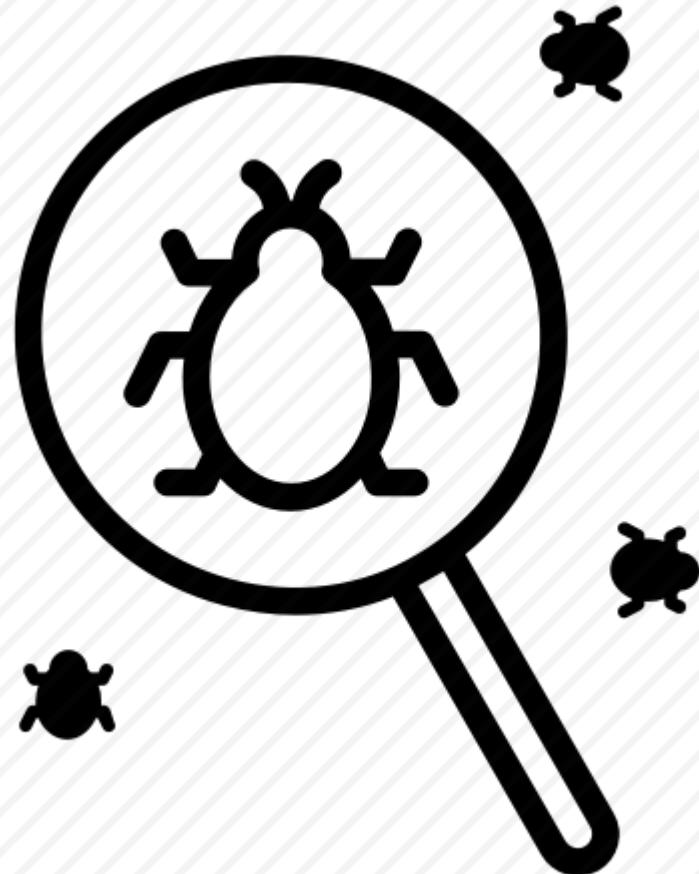
- Comentarios de Javadoc para clases, atributos y métodos;
- Convenciones de nomenclatura de atributos y métodos;
- El número de parámetros de función;
- Longitudes de línea;
- La presencia de encabezados obligatorios;
- El uso de importaciones y modificadores de alcance;
- Los espacios entre algunos personajes ;
- Las prácticas de construcción de clases;
- Múltiples medidas de complejidad .



checkstyle-rules

## Findbugs

Es una herramientas de análisis de código estático que no se centra en el estilo o el formato, se centra en encontrar defectos reales o problemas potenciales de rendimiento. Compara el código de bytes con un conjunto de patrones de defectos para encontrar posibles problemas.



## Problemas que detecta:

- **Corrección (Correctness):** Los problemas de esta clasificación pueden provocar bug, como conversión de tipo incorrecto, etc.
- **Contra ejemplos de mejores prácticas (Bad practice):** El código de esta categoría viola los estándares reconocidos de mejores prácticas.
- **Corrección de subprocesos múltiples (Multithreaded correctness):** Se centra en los problemas de sincronización y subprocesos múltiples.
- **Performance:** Posibles problemas de rendimiento.
- **La seguridad(Security):** Relacionado con la seguridad.

## ¿Cuáles son las diferencias entre PMD y FindBugs?

**PMD** funciona en el código fuente y, por lo tanto, encuentra problemas como: violación de las convenciones de nomenclatura, falta de llaves, comprobación de nulos fuera de lugar

**FindBugs** trabaja en bytecode. Aquí hay algunos problemas que FindBugs encuentra que PMD no:

- el método equals () falla en los subtipos,
- el método de clonación puede devolver nulo,
- la comparación de referencia de los valores booleanos,
- la conversión imposible

## ¿Cuáles son las diferencias entre PMD y CheckStyle?

**Checkstyle** le ayudará durante su programación comprobando su estilo de codificación, es decir, llaves, nombres, etc. Cosas simples pero muy numerosas,

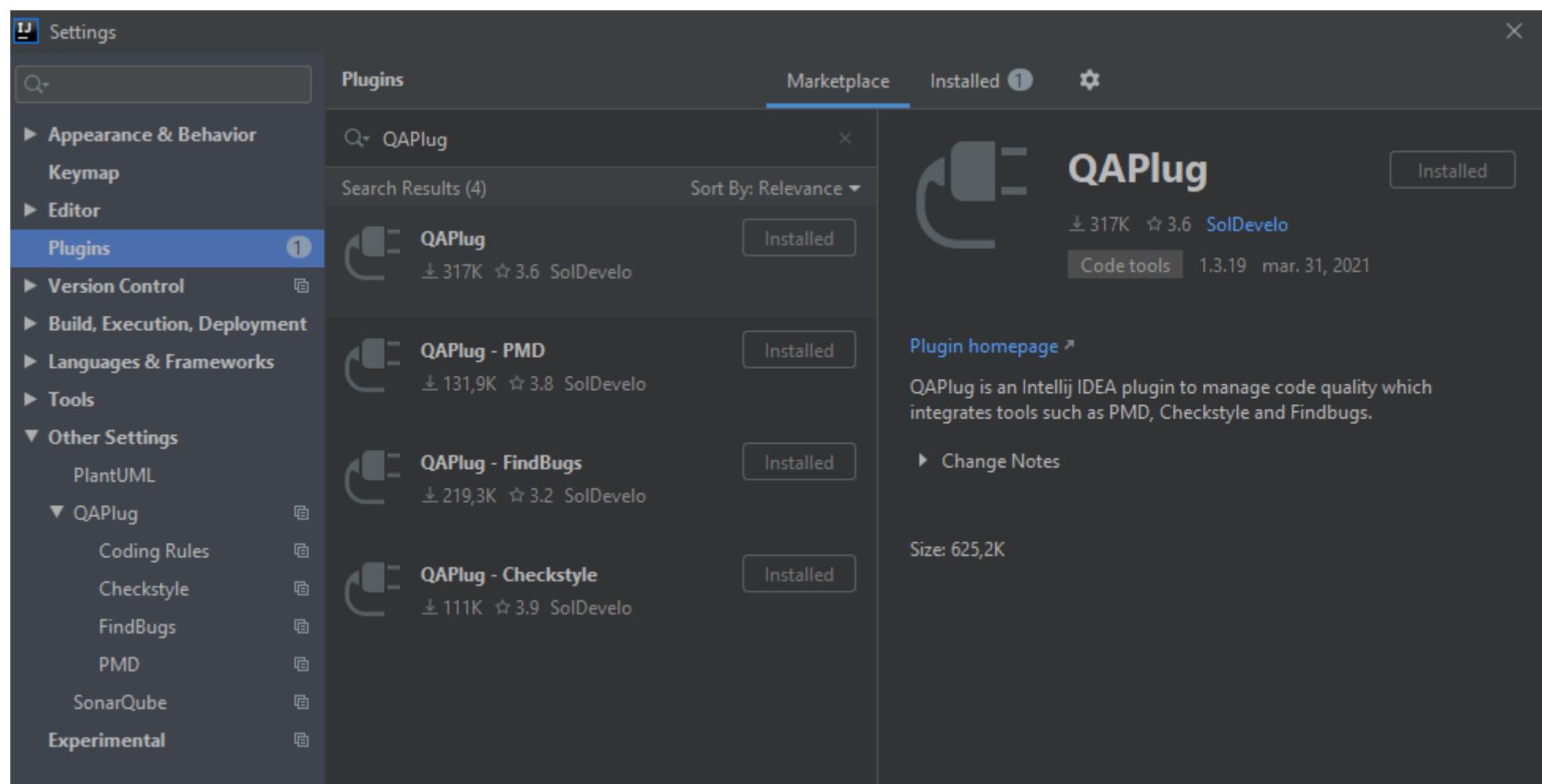
- ¿Existe javadoc en los métodos públicos? ¿El proyecto sigue las convenciones de nomenclatura de Sun? ¿El código está escrito con un formato coherente?

**PMD** lo ayudará a verificar reglas más complicadas, como durante el diseño de sus clases, o para problemas más especiales, como implementar correctamente la función de clonación. Simplemente, PMD verificará su estilo de programación:

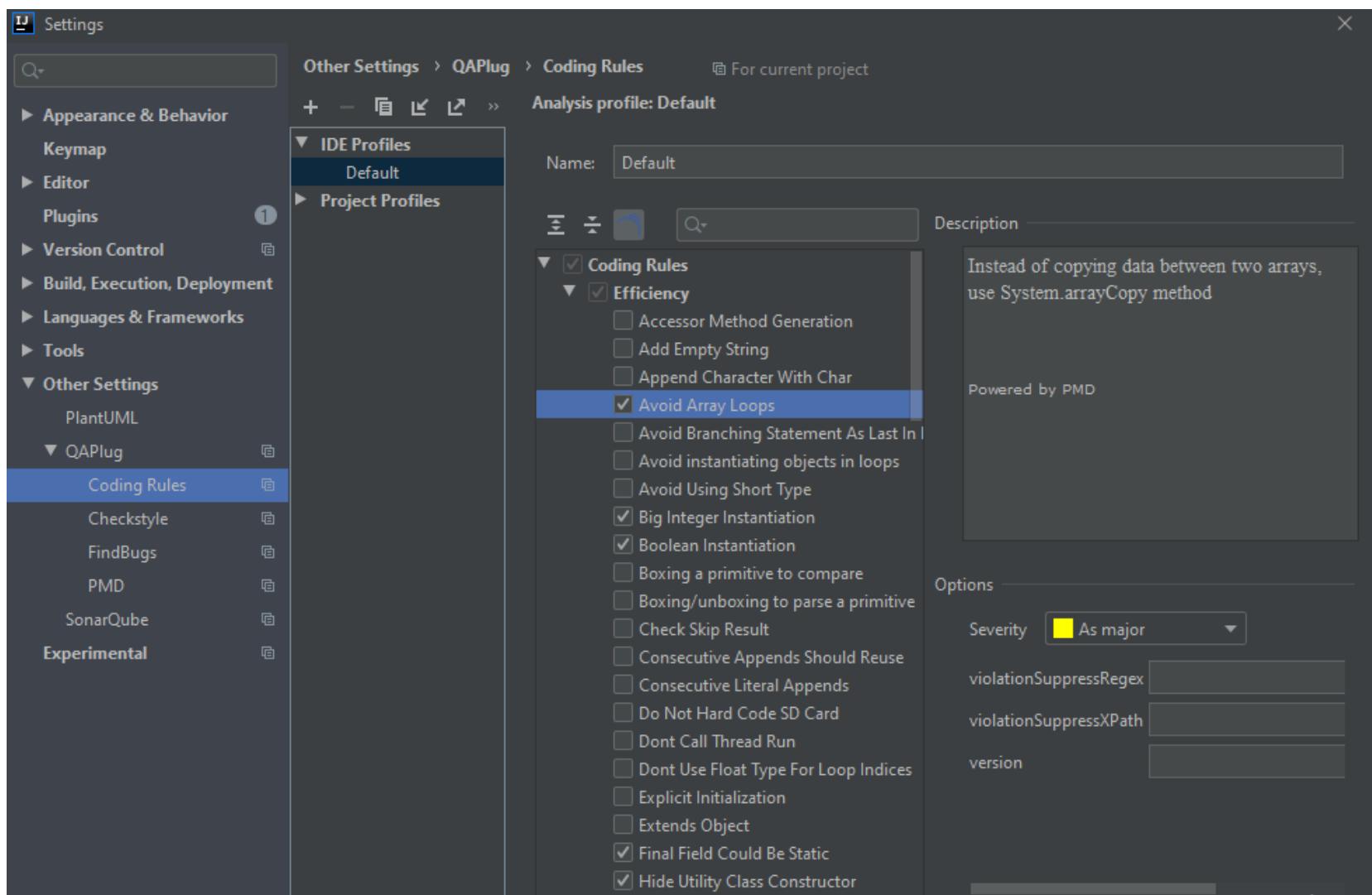
- Atrapando una excepción sin hacer nada, Tener código muerto, Demasiados métodos complejos,

**Uso directo de implementaciones en lugar de interfaces**

## Instalación



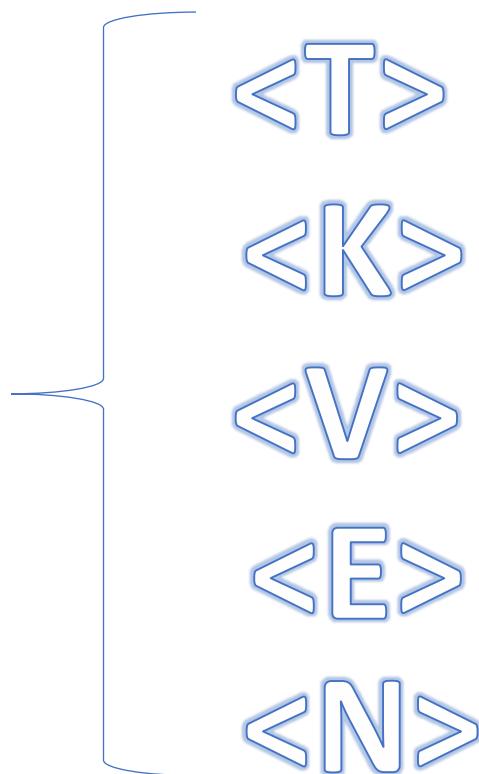
## Instalación



# Genéricos

---

generics



## Genéricos

En su esencia, el término genéricos significa **tipos parametrizados**. Los tipos parametrizados son importantes porque le permiten **crear clases, interfaces y métodos en los que el tipo de datos sobre los que operan se especifica como parámetro**. Una clase, interfaz o método que funciona con un tipo de parámetro se denomina genérico, como una **clase genérica o método genérico**.

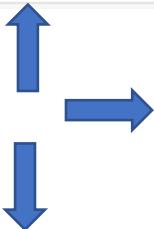


## Genéricos

Los *generics* permiten usar tipos para parametrizar las clases, interfaces y métodos al definirlas. Los **beneficios** son:

- Comprobación de tipos más fuerte en tiempo de compilación.
- Eliminación de *casts* aumentando la legibilidad del código.
- Posibilidad de implementar algoritmos genéricos, con tipado seguro.

```
1 public class Box<T> {  
2  
3     private T t;  
4  
5     public T get() { return t; }  
6     public void set(T t) { this.t = t; }  
7 }
```



```
1 public interface Pair<K, V> {  
2     public K getKey();  
3     public V getValue();  
4 }
```

```
1 public class OrderedPair<K, V> implements Pair<K, V> {  
2  
3     private K key;  
4     private V value;  
5  
6     public OrderedPair(K key, V value) {  
7         this.key = key;  
8         this.value = value;  
9     }  
10  
11    public K getKey() { return key; }  
12    public V getValue() { return value; }  
13 }
```

## Genéricos

Incluso utilizando *generics*, a veces se presentan ciertas restricciones respecto de los tipos, teniendo que usar - en determinados casos- tipos concretos. Por ejemplo, en el caso de que una función aplique sólo a números, se utilizan los **tipos límites o bounded types**.

Con estos tipos, es posible limitar a los generics haciendo que se extiendan de una clase y en consecuencia, que su límite sea el de ese tipo.

Además, los *bounds*, en *generics*, pueden ser múltiples.

```
class Bound<T extends A>
{
    private T objRef;

    public Bound(T obj){
        this.objRef = obj;
    }

    public void doRunTest(){
        this.objRef.displayClass();
    }
}
```

Existen una serie de **convenciones para nombrar a los genéricos:**

**E – Element** (usado bastante por Java Collections Framework)

**K – Key** (Llave, usado en mapas)

**N – Number** (para números)

**T – Type** (Representa un tipo, es decir, una clase)

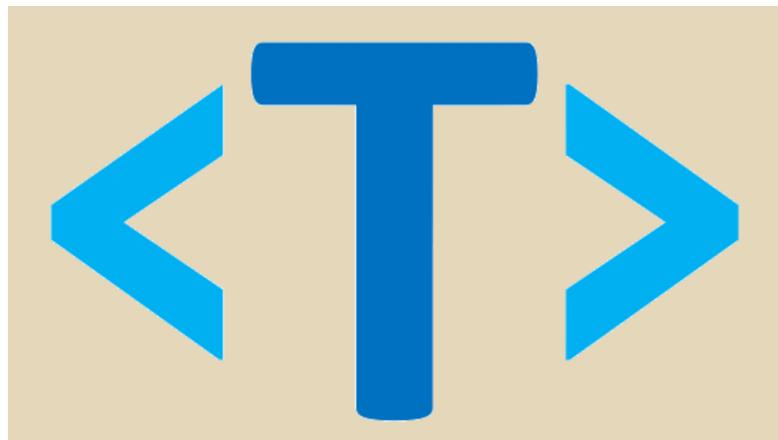
**V – Value** (representa el valor, también se usa en mapas)

**S,U,V etc.** – usado para representar otros tipos.

## Restricciones de Genéricos en Java

Hay algunas restricciones que debe tener en cuenta al usar genéricos.

Implican la creación de objetos de un parámetro de tipo, miembros estáticos, excepciones y matrices.



### 1 . Los genéricos no pueden ser instanciados

No es posible crear una instancia de un parámetro de tipo. Por ejemplo:

```
1. //No se puede crear una instancia de T
2. class Gen<T>{
3.     T ob;
4.     Gen() {
5.         ob= new T(); //Illegal;
6.     }
7. }
```

### 2 . Restricciones de miembros estaticos

Ningún miembro estático puede usar un parámetro de tipo declarado por la clase envolvente. Por ejemplo, ambos miembros estáticos de esta clase son ilegales:

```
1. class Incorrecto<T>{
2.     //Incorrecto, no hay variables estáticas de tipo T.
3.     static T ob;
4.     //Incorrecto, no hay métodos estáticos de tipo T.
5.     static T getOb(){
6.         return ob;
7.     }
8. }
```

### 3 . Restricciones de arrays genéricos

Hay dos restricciones genéricas importantes que se aplican a los arrays.

- Primero, no puede instanciar una array cuyo tipo de elemento es un parámetro de tipo.
- En segundo lugar, no puede crear un array de referencias genéricas específicas de tipo. El siguiente programa corto muestra ambas situaciones:

```
1. //Genéricos y Arrays
2.
3. class Gen <T extends Number>{
4.     T ob;
5.     T vals[]; //Ok
6.
7.     Gen(T o,T[] nums){
8.         ob=o;
9.
10.        //Esta declaración es ilegal
11.        // vals=new T; //No se puede crear un array de T
12.
13.        //Esta declaración es Ok
14.        vals=nums; //OK para asignar referencia al array existente
15.    }
16.
17.
18. class GenArrays {
19.     public static void main(String[] args) {
20.         Integer n[]={1,2,3,4,5};
21.         Gen<Integer> iOb=new Gen<Integer>(50,n);
22.
23.         //no puede crear un array o referencias genéricas específicas de tipo
24.         //Gen<Integer> gens[]=new Gen<Integer>; //Error
25.
26.         //Esto sí está Ok
27.         Gen<?> gens[]=new Gen<?>; //Ok
28.     }
29. }
```

### 4 . Restricciones de Excepción genérica

Una clase genérica no puede extender **Throwable**. Esto significa que no puede crear clases de excepciones genéricas.

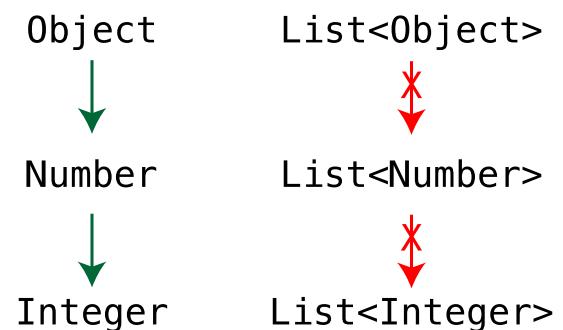
Java ofrece la posibilidad de utilizar el tipo comodín (?) que nos ayuda en el trato con los tipos genéricos y que significa, ni más ni menos, «cualquier tipo de objeto».

Este símbolo puede utilizarse tanto con la palabra extends como con la palabra super, para limitar el rango de objetos aceptados a un subtipo concreto, o a un supertipo respectivamente.

## <?> Wildcards in Collections

```
package java.util;

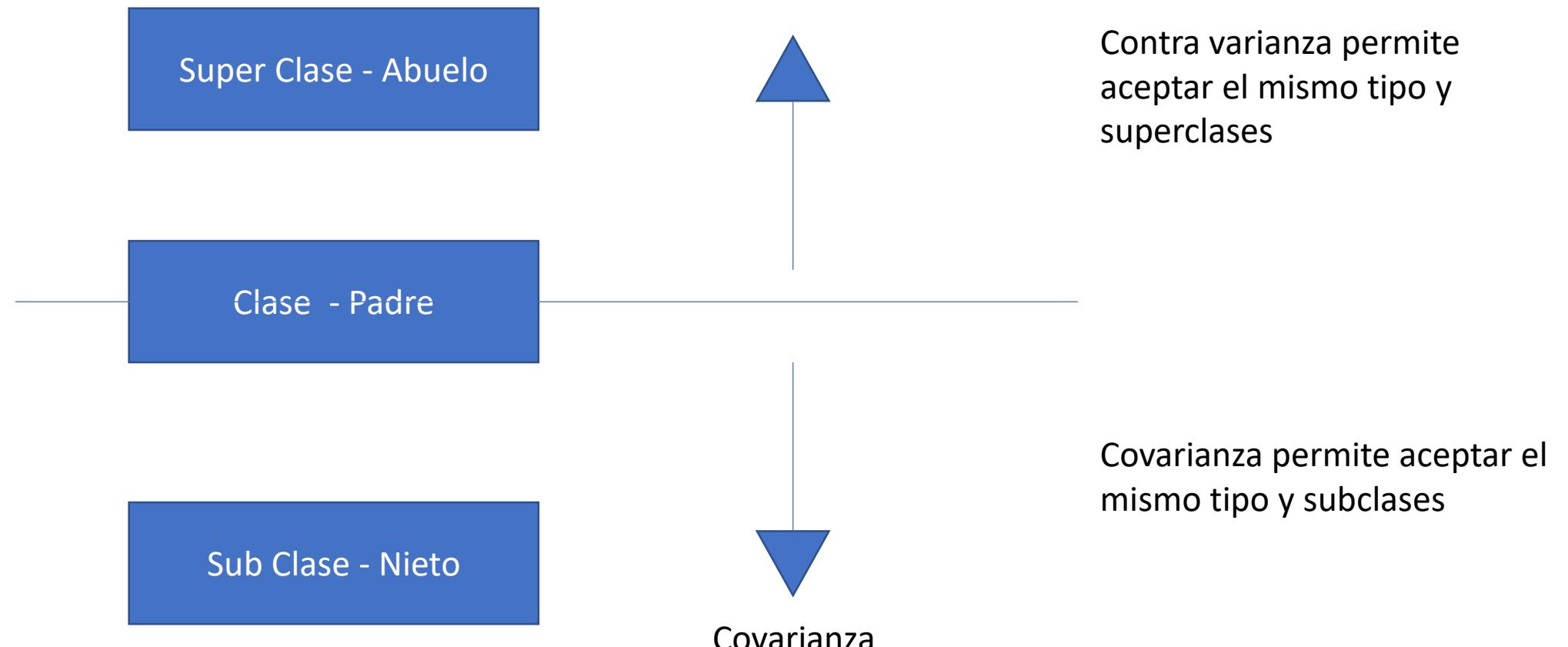
public interface Collection<E> {
    boolean containsAll(Collection<?> c);
    boolean removeAll(Collection<?> c);
    ...
}
```



## Diferencias entre T y ?

1. T es un parámetro de tipo y ? Es un comodín
2. T se utiliza como un parámetro de tipo al definir una clase genérica. T será reemplazado por un tipo concreto al crear una instancia de la clase genérica. Por otro lado, utilizamos ? cuando queremos referirnos a un desconocido escriba el argumento.
3. T en la parte superior de la clase, o método si define un método genérico. Puedes usar ? en todos lados.
4. cada uso de T se asigna al mismo tipo (en la misma clase). Cada uso de ? Se pueden mapear a un tipo diferente.
5. puede agregar objetos a una colección de tipo T. No se puede agregar objeto a una colección de tipo ? (ya que no sabes su tipo).

## Covarianza, Contra varianza e Invarianza



# Práctica 2

---

Convertir la Siguiente lista enlazada en Genérico

## Práctica 2

---

```
1 package generics;  
2  
3 public class Node {  
4     private String data;  
5     private Node next;  
6  
7     public Node(String data) {  
8         this.data = data;  
9     }  
10  
11    public String getData() {  
12        return data;  
13    }  
14  
15    public void setData(String data) {  
16        this.data = data;  
17    }  
18  
19    public Node getNext() {  
20        return next;  
21    }  
22  
23    public void setNext(Node next) {  
24        this.next = next;  
25    }  
26}
```

```
1  package generics;
2  public class MyCustomList {
3      private Node first;
4      private Node last;
5      public MyCustomList() {
6          this.first = null;
7          this.last = null;
8      }
9      public void add(String data) {
10         Node newNode = new Node(data);
11         newNode.setNext(null);
12         if(this.first == null) {
13             this.first = newNode;
14         } else {
15             this.last.setNext(newNode);
16         }
17         last = newNode;
18     }
19     public void print() {
20         Node node = this.first;
21         while (node != null) {
22             System.out.println(node.getData());
23             node = node.getNext();
24         }
25     }
26     public void addLast(String data) {
27         this.add(data);
28     }
29     public void addFirst(String data) {
30         Node newNode = new Node(data);
31         if (this.first == null) {
32             this.first = newNode;
33             this.last = newNode;
34         } else {
35             newNode.setNext(this.first);
36             this.first = newNode;
37         }
38     }
39 }
```

```
1 import generics.MyCustomList;
2 public class Main {
3     public static void main(String[] args) {
4         System.out.println("Hi Software Design - Exercises");
5         MyCustomList list1 = new MyCustomList();
6         list1.add("a");
7         list1.add("b");
8         list1.addFirst( data: "c");
9         list1.addLast( data: "d");
10        list1.addFirst( data: "e");
11        list1.print();
12
13        MyCustomList<Integer> list2 = new MyCustomList<>();
14        list2.add(1);
15        list2.addFirst( data: 2);
16        list2.addLast( data: 0);
17        list2.print();
18
19        MyCustomList<Boolean> list3 = new MyCustomList<>();
20        list3.add(true);
21        list3.addFirst( data: false);
22        list3.addLast( data: false);
23        list3.print();
24
25        MyCustomList<String> list4 = new MyCustomList<>();
26        list4.add("x");
27        list4.addFirst( data: "y");
28        list4.addLast( data: "z");
29        list4.print();
30    }
31 }
```

# Práctica 2

---

```
1 package generics;  
2  
3 public class Node<T> {  
4     private T data;  
5     private Node<T> next;  
6  
7     public Node(T data) {  
8         this.data = data;  
9     }  
10  
11    public T getData() {  
12        return data;  
13    }  
14  
15    public void setData(T data) {  
16        this.data = data;  
17    }  
18  
19    public Node<T> getNext() {  
20        return next;  
21    }  
22  
23    public void setNext(Node<T> next) {  
24        this.next = next;  
25    }  
26}
```

```

1 package generics;
2
3 public class MyCustomList<T> {
4     private Node first;
5     private Node last;
6
7     public MyCustomList() {
8         this.first = null;
9         this.last = null;
10    }
11
12    public void add(T data){
13        Node newNode = new Node(data);
14        newNode.setNext(null);
15        if (this.first == null){
16            this.first = newNode;
17        }
18        else {
19            this.last.setNext(newNode);
20        }
21        last= newNode;
22    }
23
24    public void print(){
25        Node node = this.first;
26        while (node != null){
27            System.out.println(node.getData());
28            node = node.getNext();
29        }
30    }
31
32    public void addLast(T data){
33        this.add(data);
34    }
35
36    public void addFirst(T data){
37        Node newNode = new Node(data);
38        if (this.first == null){
39            this.first = newNode;
40            this.last = newNode;
41        }
42        else{
43            newNode.setNext(this.first);
44            this.first = newNode;
45        }
46    }
47 }

```

---

```

1 import generics.MyCustomList;
2
3 public class Main {
4     public static void main(String[] args) {
5         System.out.println("Hi Software Design - Exercises");
6         MyCustomList list1 = new MyCustomList();
7         list1.add("a");
8         list1.add("b");
9         list1.addFirst("c");
10        list1.addLast("d");
11        list1.addFirst("e");
12        list1.print();
13
14        MyCustomList<Integer> list2 = new MyCustomList<>();
15        list2.add(1);
16        list2.addFirst(2);
17        list2.addLast(0);
18        list2.print();
19
20        MyCustomList<Boolean> list3 = new MyCustomList<>();
21        list3.add(true);
22        list3.addFirst(false);
23        list3.addLast(false);
24        list3.print();
25
26        MyCustomList<String> list4 = new MyCustomList<>();
27        list4.add("x");
28        list4.addFirst("y");
29        list4.addLast("z");
30        list4.print();
31    }
32

```

# Práctica 2

---

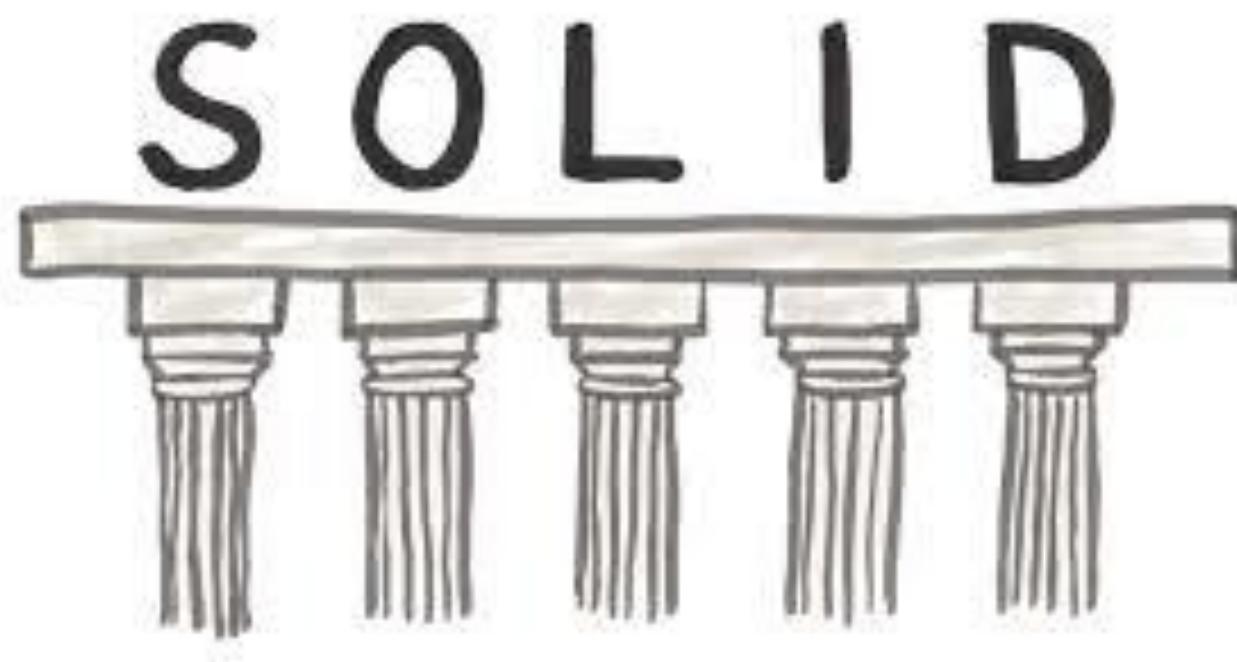
```
3  public class MyCustomList<E> {  
4      private Node first;  
5      private Node last;  
6  
7      public MyCustomList() {  
8          this.first = null;  
9          this.last = null;  
10     }  
11  
12     public void add(E data) {  
13         Node newNode = new Node(data);  
14         newNode.setNext(null);  
15         if (this.first == null) {  
16             this.first = newNode;  
17         } else {  
18             this.last.setNext(newNode);  
19         }  
20         this.last = newNode;  
21     }  
22  
23     public void print() {  
24         this.print("\n");  
25     }  
26  
27     public void print(String separator) {  
28         Node node = this.first;  
29         while (node != null) {  
30             String calculatedSeparator = node.getNext() != null ? separator : "";  
31             System.out.print(node.getData() + calculatedSeparator);  
32  
33             node = node.getNext();  
34         }  
35     }  
36  
37     public void addLast(E data) {  
38         this.add(data);  
39     }
```

```
1  package generics;  
2  
3  public class Node<E> {  
4      private E data;  
5      private Node next;  
6  
7      public Node(E data) {  
8          this.data = data;  
9      }  
10  
11     public E getData() {  
12         return data;  
13     }  
14  
15     public void setData(E data) {  
16         this.data = data;  
17     }  
18  
19     public Node getNext() {  
20         return next;  
21     }  
22  
23     public void setNext(Node next) {  
24         this.next = next;  
25     }  
26 }
```

---

# Principios

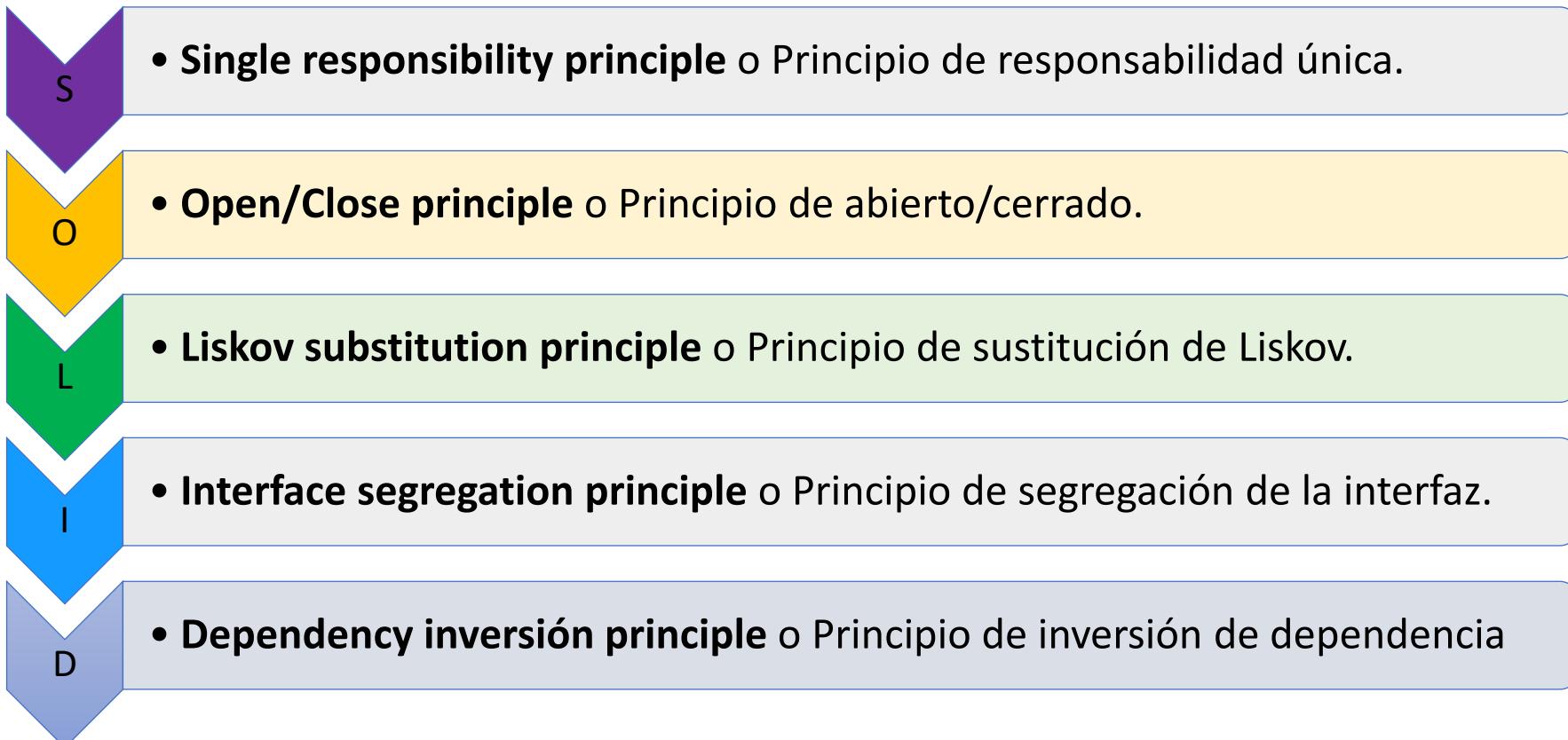
---



- Como sabemos Programación Orientada a Objetos nos permite **agrupar entidades con funcionalidades parecidas o relacionadas entre sí**, pero esto no implica que los programas no se **vuelvan confusos o difíciles de mantener**.
- Muchos programas acaban volviéndose un monstruo al que se va alimentando según se añaden nuevas funcionalidades, se realiza mantenimiento, etc.
- Viendo este problema, **Robert C. Martin** estableció cinco directrices o principios para facilitarnos a los desarrolladores la labor de crear programas legibles y mantenibles.



Estos principios se llamaron **S.O.L.I.D.** por sus siglas en inglés:



**S: Principio de responsabilidad única:**

Como su propio nombre indica, establece que una clase, componente o microservicio debe ser responsable de una sola cosa (el tan aclamado término “decoupled” en inglés). Si una clase tiene varias responsabilidades, esto implica que el cambio en una responsabilidad provocará la modificación en otra responsabilidad.



**S: Principio de responsabilidad única:**

```
class Coche {  
    String marca;  
  
    Coche(String marca){ this.marca = marca; }  
  
    String getMarcaCoche(){ return marca; }  
  
    void guardarCocheDB(Coche coche){ ... }  
}
```



```
class Coche {  
    String marca;  
  
    Coche(String marca){ this.marca = marca; }  
  
    String getMarcaCoche(){ return marca; }  
}  
  
class CocheDB{  
    void guardarCocheDB(Coche coche){ ... }  
    void eliminarCocheDB(Coche coche){ ... }  
}
```

S.O.L.I.D

## O: Principio abierto/cerrado:

Establece que las entidades software (clases, módulos y funciones) deberían estar abiertos para su extensión, pero cerrados para su modificación.



**O: Principio abierto/cerrado:**

```
class Coche {  
    String marca;  
  
    Coche(String marca){ this.marca = marca; }  
  
    String getMarcaCoche(){ return marca; }  
}
```

Si quisiéramos iterar a través de una lista de coches e imprimir sus precios por pantalla:

```
public static void main(String[] args) {  
    Coche[] arrayCoches = {  
        new Coche("Renault"),  
        new Coche("Audi")  
    };  
    imprimirPrecioMedioCoche(arrayCoches);  
}  
  
public static void imprimirPrecioMedioCoche(Coche[] arrayCoches){  
    for (Coche coche : arrayCoches) {  
        if(coche.marca.equals("Renault")) System.out.println(18000);  
        if(coche.marca.equals("Audi")) System.out.println(25000);  
    }  
}
```

## O: Principio abierto/cerrado:

Esto no cumpliría el principio abierto/cerrado, ya que si decidimos añadir un nuevo coche de otra marca  
También tendríamos que modificar el método que hemos creado anteriormente:

```
Coche[] arrayCoches = {  
    new Coche("Renault"),  
    new Coche("Audi"),  
    new Coche("Mercedes")  
};
```

```
public static void imprimirPrecioMedioCoche(Coche[] arrayCoches){  
    for (Coche coche : arrayCoches) {  
        if(coche.marca.equals("Renault")) System.out.println(18000);  
        if(coche.marca.equals("Audi")) System.out.println(25000);  
        if(coche.marca.equals("Mercedes")) System.out.println(27000);  
    }  
}
```

**O: Principio abierto/cerrado:**

```
abstract class Coche {
    // ...
    abstract int precioMedioCoche();
}

class Renault extends Coche {
    @Override
    int precioMedioCoche() { return 18000; }
}

class Audi extends Coche {
    @Override
    int precioMedioCoche() { return 25000; }
}

class Mercedes extends Coche {
    @Override
    int precioMedioCoche() { return 27000; }
}

public static void main(String[] args) {

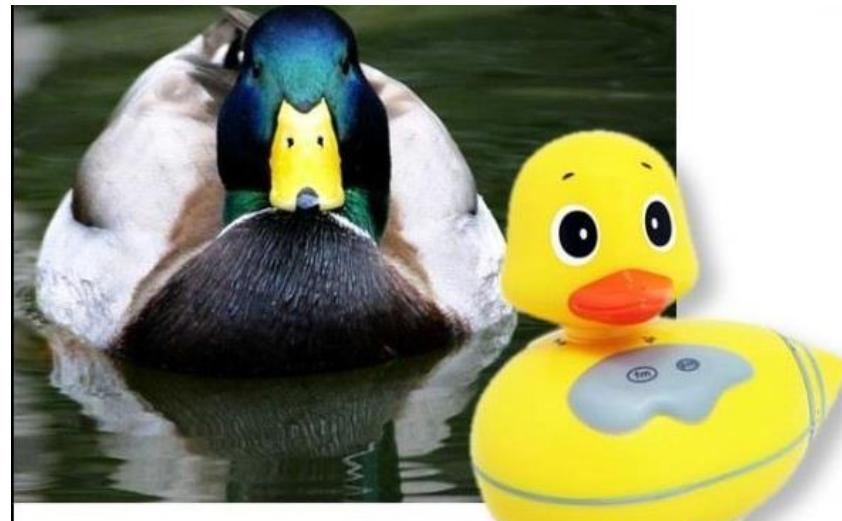
    Coche[] arrayCoches = {
        new Renault(),
        new Audi(),
        new Mercedes()
    };

    imprimirPrecioMedioCoche(arrayCoches);
}

public static void imprimirPrecioMedioCoche(Coche[] arrayCoches){
    for (Coche coche : arrayCoches) {
        System.out.println(coche.precioMedioCoche());
    }
}
```

**L: Principio de substitución de Liskov:**

Declara que una subclase debe ser sustituible por su superclase, y si al hacer esto, el programa falla, estaremos violando este principio. Cumpliendo con este principio se confirmará que nuestro programa tiene una jerarquía de clases fácil de entender y un código reutilizable.



si parece un pato, flota como un pato, pero necesita baterías - probablemente tiene la abstracción equivocada.

**L: Principio de substitución de Liskov:**

```
// ...
Coche[] arrayCoches = {
    new Renault(),
    new Audi(),
    new Mercedes(),
    new Ford()
};

public static void imprimirNumAsientos(Coche[] arrayCoches){
    for (Coche coche : arrayCoches) {
        if(coche instanceof Renault)
            System.out.println(numAsientosRenault(coche));
        if(coche instanceof Audi)
            System.out.println(numAsientosAudi(coche));
        if(coche instanceof Mercedes)
            System.out.println(numAsientosMercedes(coche));
        if(coche instanceof Ford)
            System.out.println(numAsientosFord(coche));
    }
}
imprimirNumAsientos(arrayCoches);
```

## L: Principio de sustitución de Liskov:

```
public static void imprimirNumAsientos(Coche[] arrayCoches){  
    for (Coche coche : arrayCoches) {  
        System.out.println(coche.numAsientos());  
    }  
  
    imprimirNumAsientos(arrayCoches);
```

```
abstract class Coche {  
  
    // ...  
    abstract int numAsientos();  
}
```

```
class Renault extends Coche {  
  
    // ...  
    @Override  
    int numAsientos() {  
        return 5;  
    }  
}  
// ...
```

## I: Principio de segregación de interfaz:

Este principio establece que los clientes no deberían verse forzados a depender de interfaces que no usan.

Dicho de otra manera, cuando un cliente depende de una clase que implementa una interfaz cuya funcionalidad este cliente no usa, pero que otros clientes sí usan, este cliente estará siendo afectado por los cambios que fuercen otros clientes en dicha interfaz.



## I: Principio de segregación de interfaz:

```
interface IAve {
    void volar();
    void comer();
}

class Loro implements IAve{
    @Override
    public void volar() {
        //...
    }

    @Override
    public void comer() {
        //..
    }
}

class Tucan implements IAve{
    @Override
    public void volar() {
        //...
    }

    @Override
    public void comer() {
        //..
    }
}
```

## I: Principio de segregación de interfaz:

```
interface IAve {  
    void volar();  
    void comer();  
    void nadar();  
}  
  
class Loro implements IAve{  
  
    @Override  
    public void volar() {  
        //...  
    }  
  
    @Override  
    public void comer() {  
        //...  
    }  
  
    @Override  
    public void nadar() {  
        //...  
    }  
}  
  
class Pinguino implements IAve{  
  
    @Override  
    public void volar() {  
        //...  
    }  
  
    @Override  
    public void comer() {  
        //...  
    }  
  
    @Override  
    public void nadar() {  
        //...  
    }  
}
```

**I: Principio de segregación de interfaz:**

```
interface IAve {
    void comer();
}

interface IAveVoladora {
    void volar();
}

interface IAveNadadora {
    void nadar();
}

class Loro implements IAve, IAveVoladora{

    @Override
    public void volar() {
        //...
    }

    @Override
    public void comer() {
        //...
    }
}

class Pinguino implements IAve, IAveNadadora{

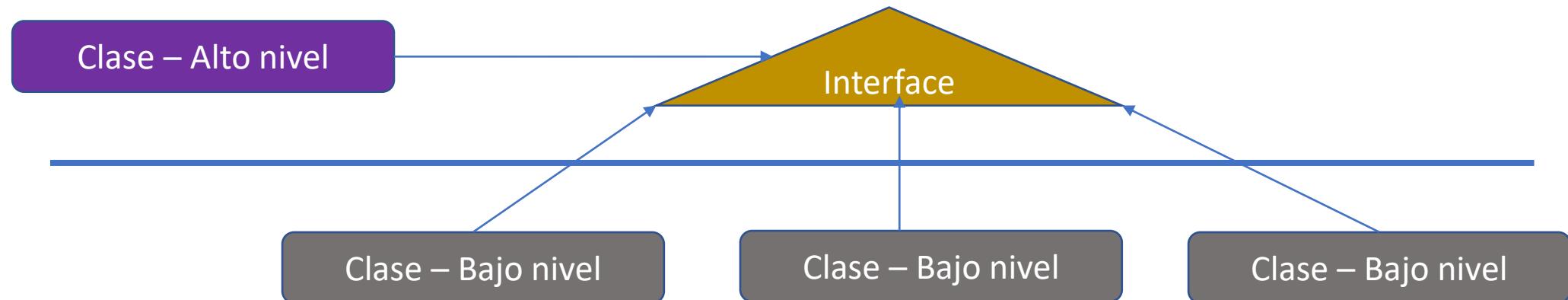
    @Override
    public void nadar() {
        //...
    }

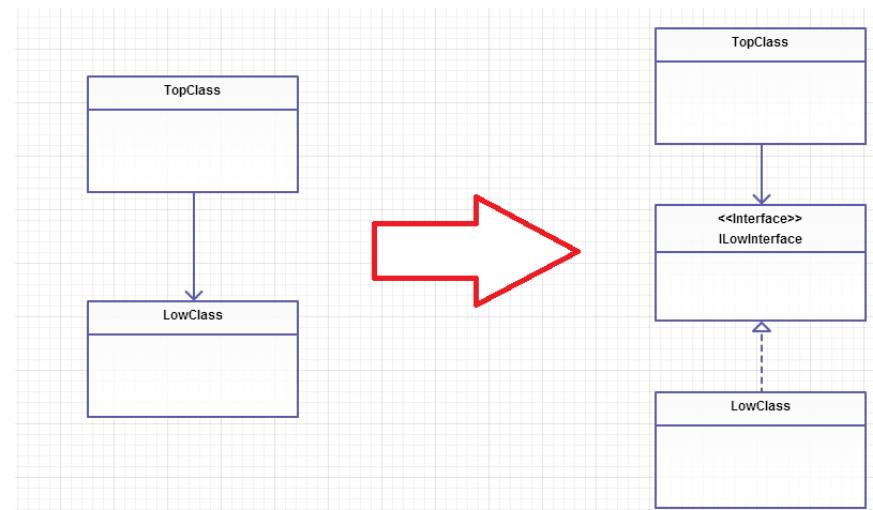
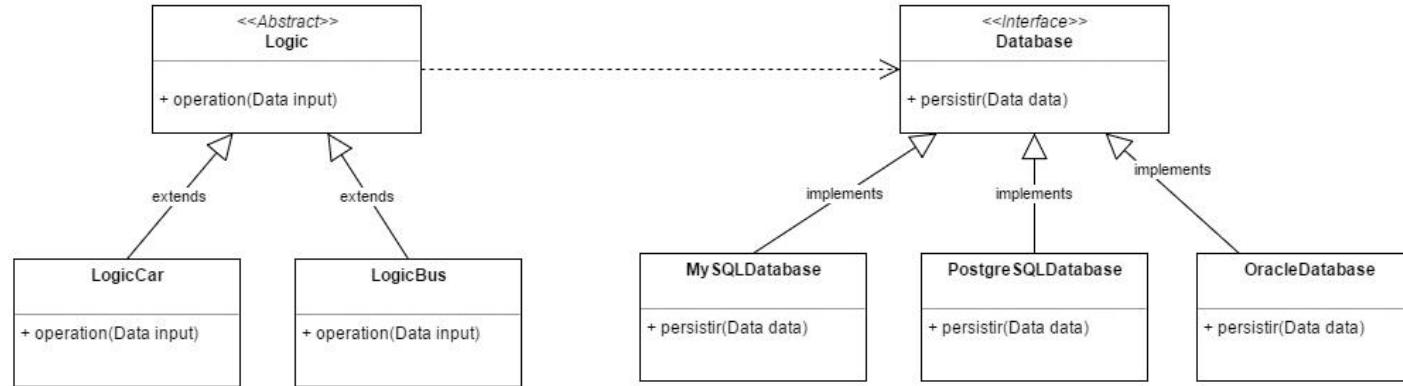
    @Override
    public void comer() {
        //...
    }
}
```

#### D: Principio de inversión de dependencias:

Establece que las dependencias deben estar en las abstracciones, no en las concreciones. Es decir:

- Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.
- Las abstracciones no deberían depender de detalles. Los detalles deberían depender de abstracciones.



**D: Principio de inversión de dependencias:**

## VENTAJAS

Mantenimiento del código más fácil y rápido

Permite añadir nuevas funcionalidades de forma más sencilla

Favorece una mayor reusabilidad y calidad del código, así como la encapsulación



# Crear proyecto spring

## Contenido del archivo build.gradle

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:2.1.6.RELEASE")
    }
}

plugins {
    id 'java'
}

apply plugin: 'org.springframework.boot'
apply plugin: 'io.spring.dependency-management'

group 'org.example'
version '1.0-SNAPSHOT'

sourceCompatibility = 1.8

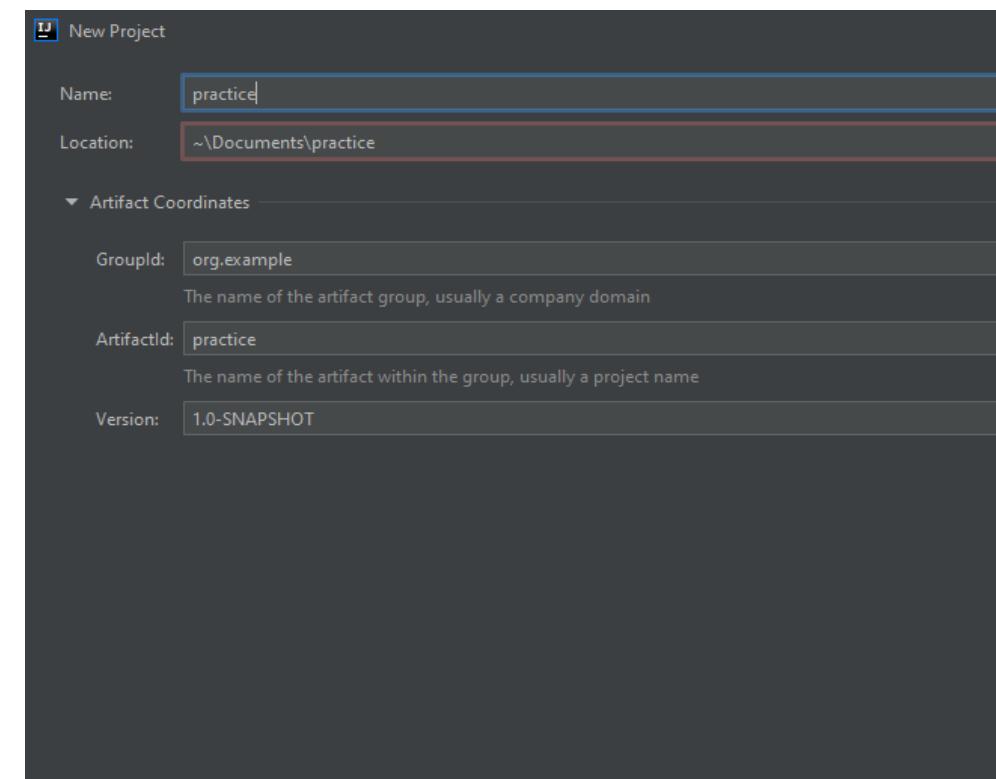
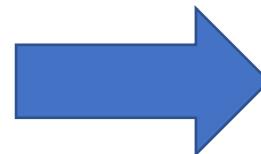
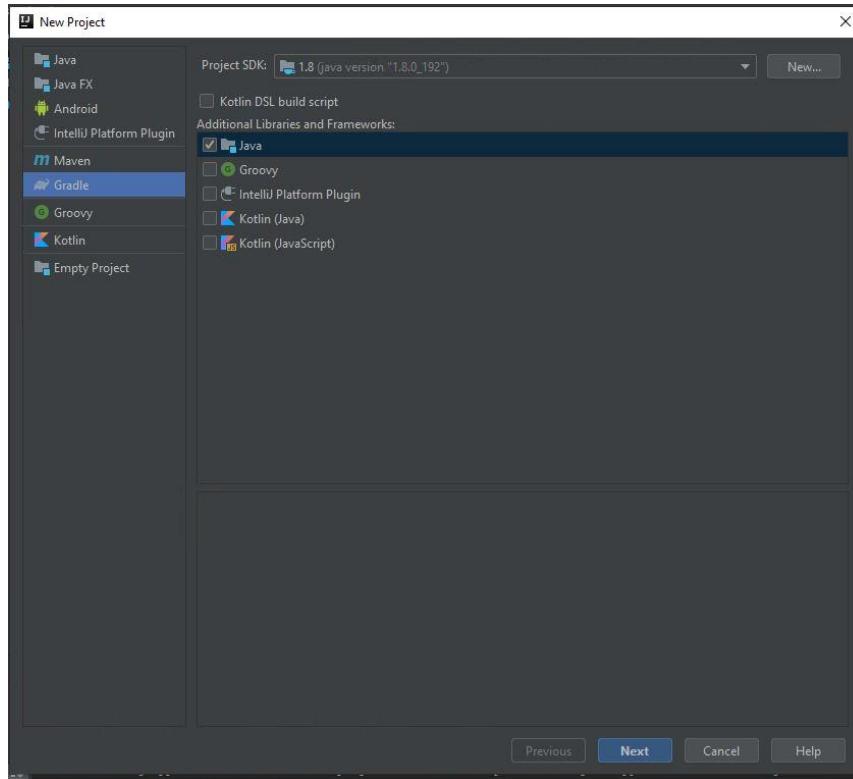
repositories {
    mavenCentral()
}

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.12'
    compile("org.springframework.boot:spring-boot-starter-web")
}
```

# Proyecto Spring

## Pasos:

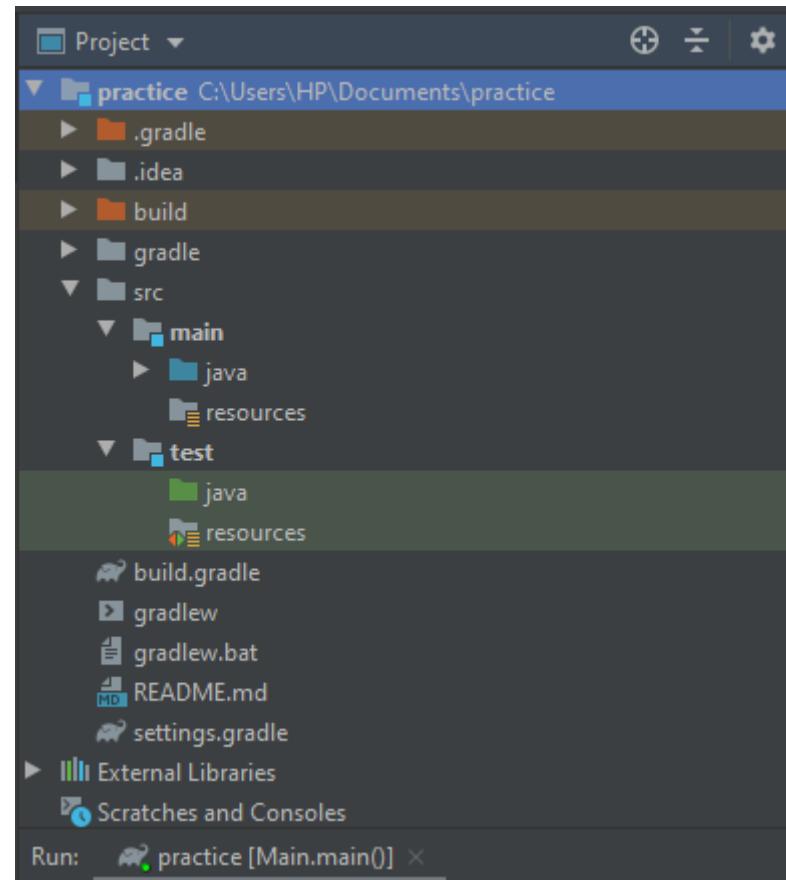
1. Abrir IntelliJ Idea
2. Ir a File->New->Project



# Proyecto Spring

## Pasos:

3. Verifique que el proyecto se haya creado con la siguiente estructura y que el build se haya ejecutado sin errores.

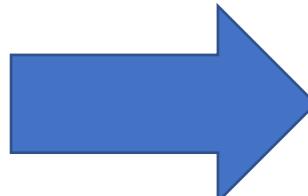


# Proyecto Spring

## Pasos:

4. Modificar el archivo build.gradle para incluir las dependencias de Spring

```
1  plugins {
2      id 'java'
3  }
4
5  group 'WebService'
6  version '1.0-SNAPSHOT'
7
8  sourceCompatibility = 1.8
9
10 repositories {
11     mavenCentral()
12 }
13
14 > dependencies {
15     testCompile group: 'junit', name: 'junit', version: '4.12'
16 }
17
```

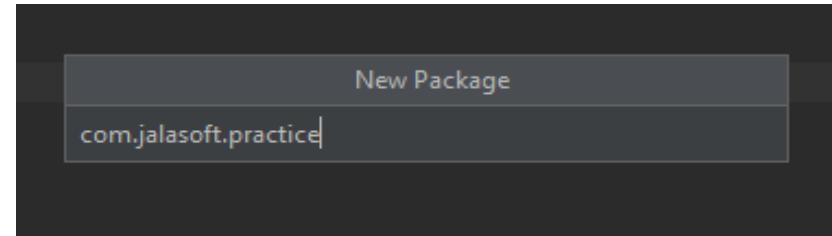


```
1  buildscript {
2      repositories {
3          mavenCentral()
4      }
5      dependencies {
6          classpath("org.springframework.boot:spring-boot-gradle-plugin:2.1.6.RELEASE")
7      }
8  }
9
10 repositories {
11     mavenCentral()
12 }
13
14 > dependencies {
15     compile("org.springframework.boot:spring-boot-starter-web")
16     testCompile group: 'junit', name: 'junit', version: '4.12'
17 }
```

# Proyecto Spring

## Pasos:

5. Crear paquetes
6. Crear la clase Main y agregar las anotaciones de Spring



The screenshot shows an IDE interface with the following details:

- Project View:** Shows the project structure under 'practice': .gradle, .idea, build, gradle, inputFiles, src (expanded), main (expanded), java (expanded), com.jalasoft.practice (expanded), controller (expanded), HelloController.java (selected), model, Main (selected), resources, test.
- Main.java Content:**

```
1 package com.jalasoft.practice;
2
3 import org.springframework.boot.autoconfigure.SpringBootApplication;
4 import org.springframework.boot.SpringApplication;
5
6 @SpringBootApplication
7 public class Main {
8     public static void main(String[] args) {
9         System.out.println("Hello!!!");
10        SpringApplication.run(Main.class, args);
11    }
12 }
13 |
```
- Other Files:** README.md, Main.java, HelloController.java, build.gradle are listed in the tabs at the top.

# Proyecto Spring

## Pasos:

8. Verificar desde postman si el servicio web esta corriendo.

The screenshot shows the Postman interface with the following details:

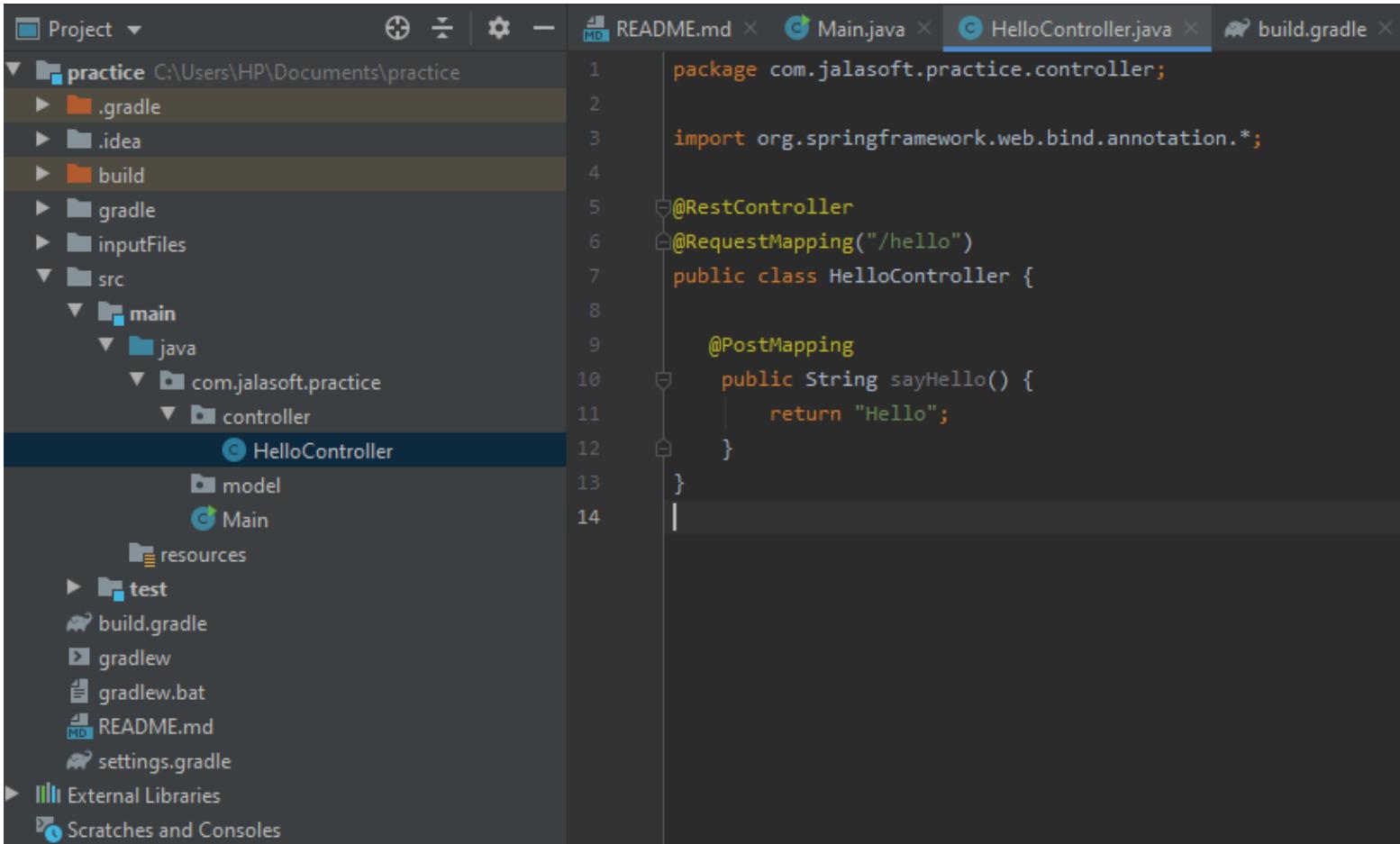
- Request URL:** GET localhost:8080
- Environment:** No Environment
- Method:** GET
- URL:** localhost:8080
- Buttons:** Send, Save
- Params Tab:** Active. Shows a table for "Query Params" with one entry: Key (Value) and Value (Description).
- Body Tab:** Active. Shows the response body in JSON format:

```
1 {
2     "timestamp": "2019-09-16T01:09:40.719+0000",
3     "status": 404,
4     "error": "Not Found",
5     "message": "No message available",
6     "path": "/"
7 }
```

# Proyecto Spring

## Pasos:

9. Crear el primer endpoint (localhost:8080/hello), para ellos necesitamos adicionar una nueva clase controlador y realizar el mapeo.



The screenshot shows a Java development environment with the following details:

- Project View:** The project is named "practice" located at "C:\Users\HP\Documents\practice". It contains a ".gradle" folder, an ".idea" folder, a "build" folder, a "gradle" folder, an "inputFiles" folder, and a "src" folder. The "src" folder contains a "main" folder with a "java" subfolder. Inside "java", there is a package "com.jalasoft.practice" which contains a "controller" folder. Within "controller", there is a file named "HelloController.java". Other files in "main/java" include "Main.java" and "model.java". There are also "resources" and "test" folders.
- Code Editor:** The "HelloController.java" file is open in the editor. The code defines a controller class named "HelloController" with a single endpoint mapping to "/hello".
- Toolbars and Status:** The top bar includes standard icons for file operations like new, open, save, and close. The bottom status bar shows "Scratches and Consoles".

```
package com.jalasoft.practice.controller;  
import org.springframework.web.bind.annotation.*;  
  
@RestController  
@RequestMapping("/hello")  
public class HelloController {  
  
    @PostMapping  
    public String sayHello() {  
        return "Hello";  
    }  
}
```

# Proyecto Spring

## Pasos:

10. Verificar desde postman si el servicio web esta corriendo (utilice el nuevo endpoint).

The screenshot shows a Postman interface with the following details:

- Method:** POST
- URL:** localhost:8080/hello
- Headers:** (7) (highlighted)
- Body:** (highlighted)
- Params:** none
- Authorization:** None
- Body Content:** Form-data (selected)

KEY	VALUE	DESCRIPTION
Key	Value	Description
- Status:** 200 OK
- Time:** 157 ms
- Size:** 120 B
- Buttons:** Send, Save, Pretty, Raw, Preview, Visualize

## Proyecto Spring

### Pasos:

10. Modificar el endpoint (localhost:8080/hello) para que reciba parámetros del tipo cadena

```
package com.jalasoft.practice.controller;

import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/hello")
public class HelloController {

    @PostMapping
    public String sayHello(@RequestParam(value="name") String name,
                          @RequestParam(value="lastName") String lname) {
        return "Hello " + name + " " + lname;
    }
}
```

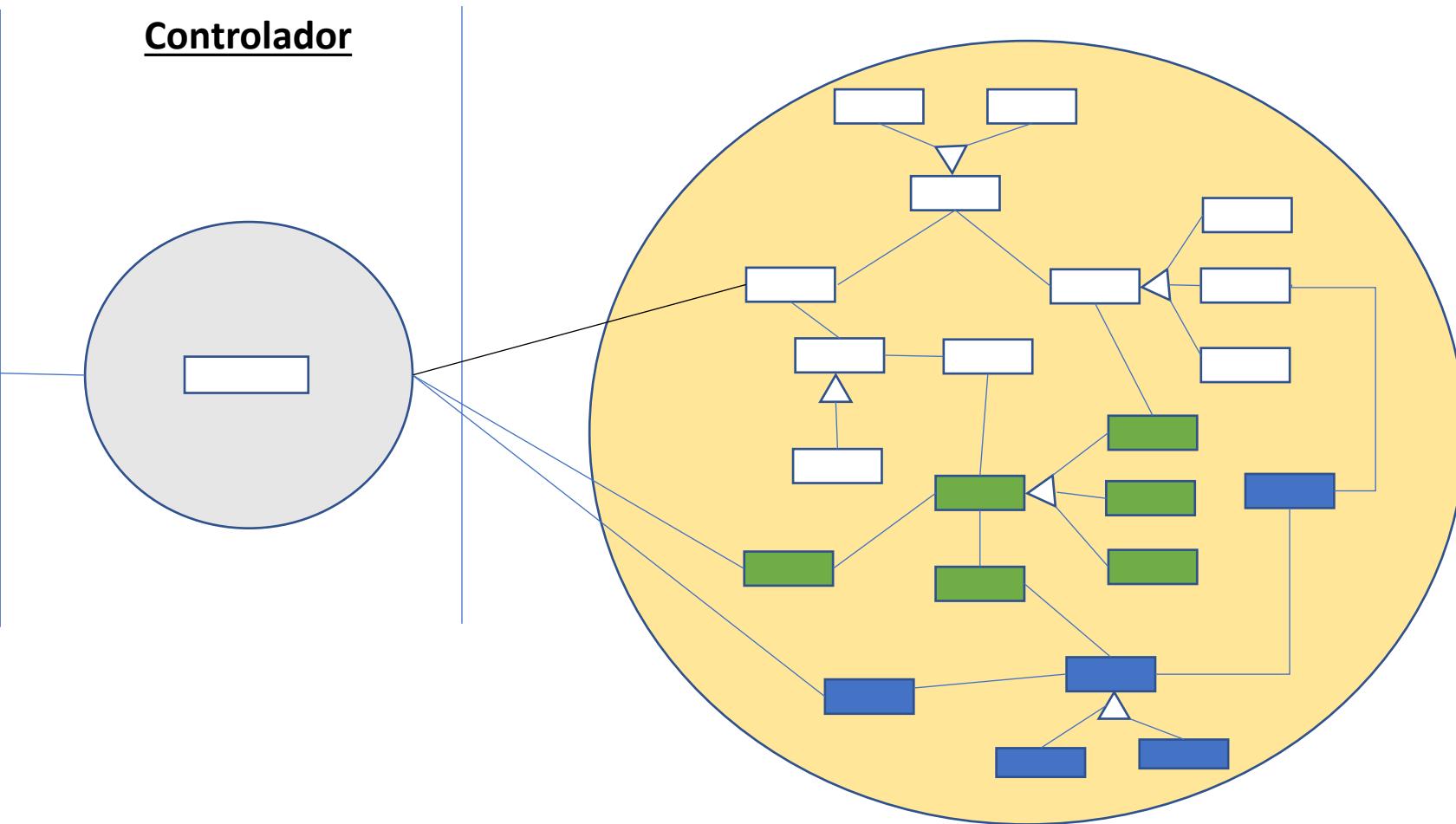
# Proyecto Spring

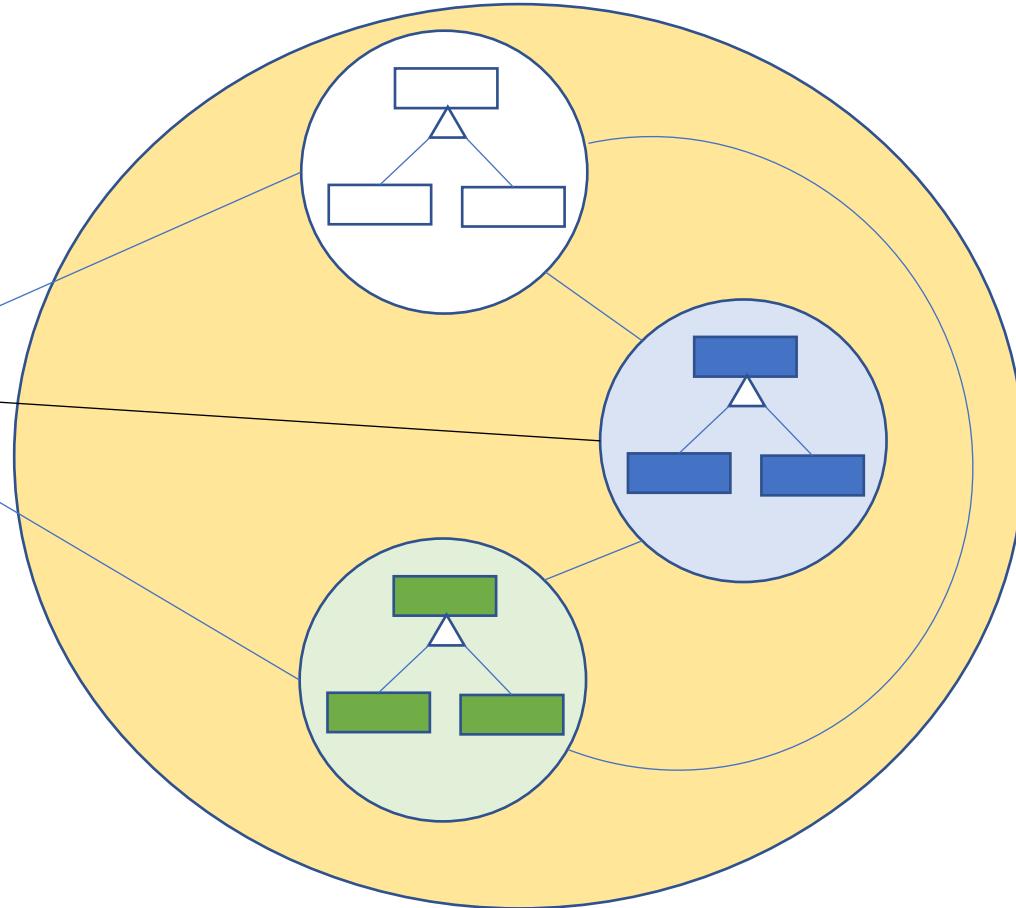
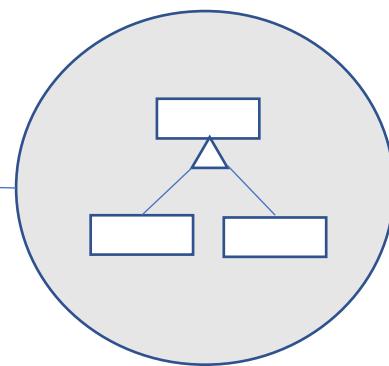
## Pasos:

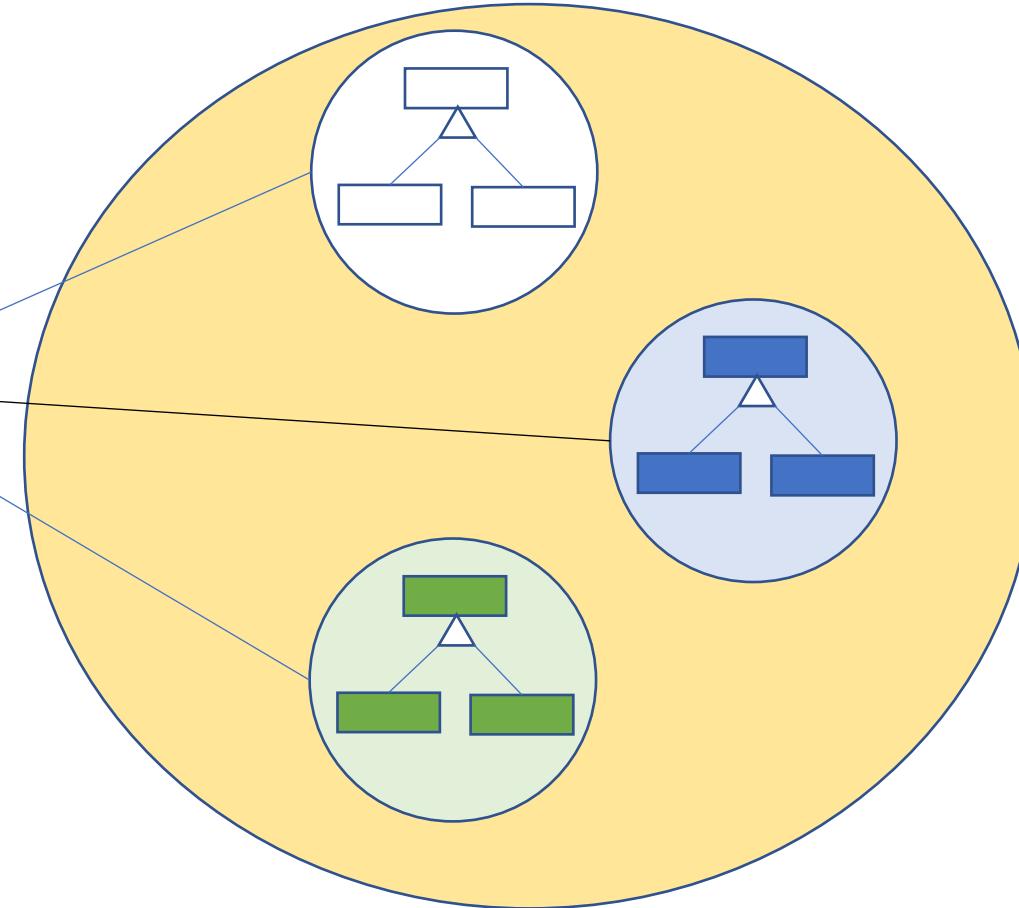
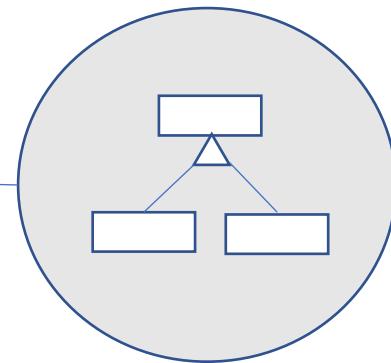
11. Verificar desde postman si el servicio web esta corriendo.

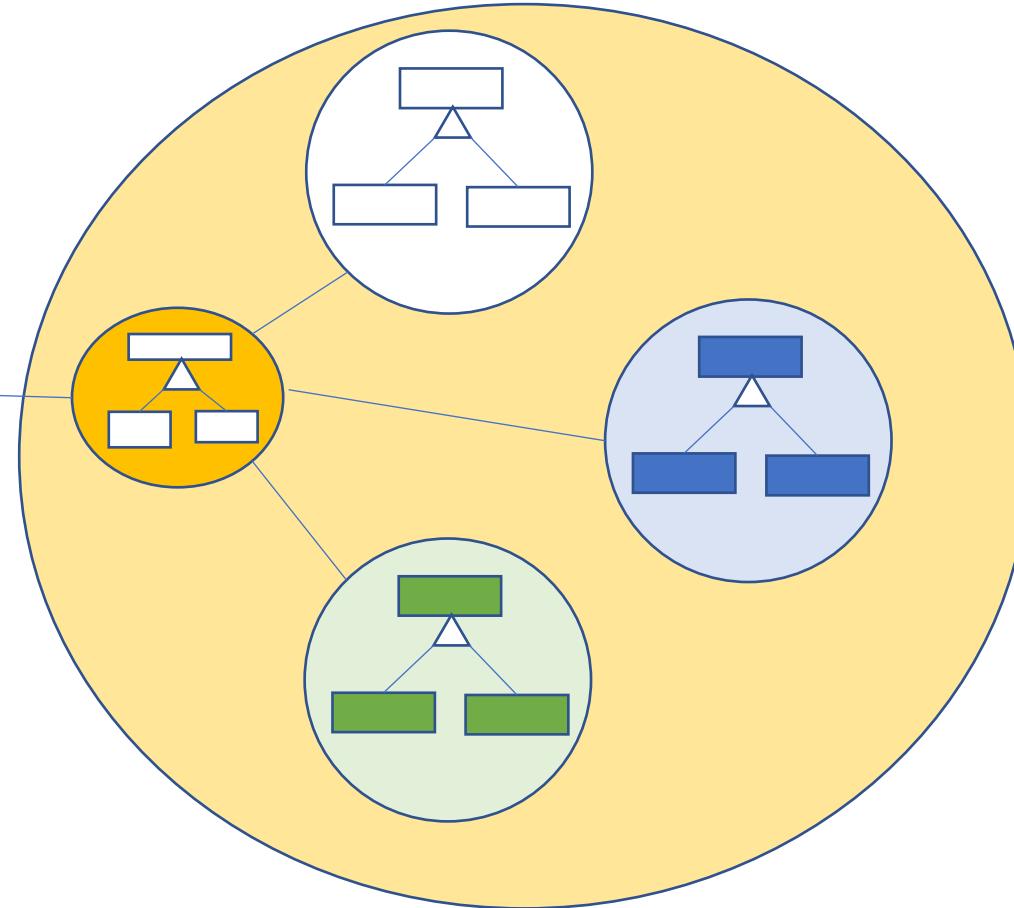
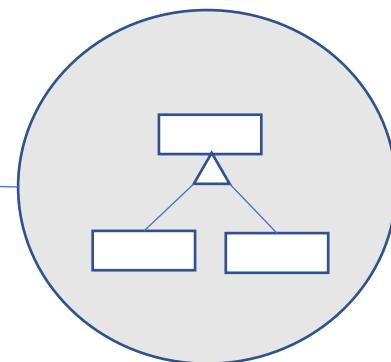
The screenshot shows a Postman interface with the following details:

- Method:** POST
- URL:** localhost:8080/hello
- Body (form-data):**
  - name: Juan
  - lastName: Perez
- Status:** 200 OK
- Time:** 98 ms
- Size:** 132 B

ModeloVistaControlador

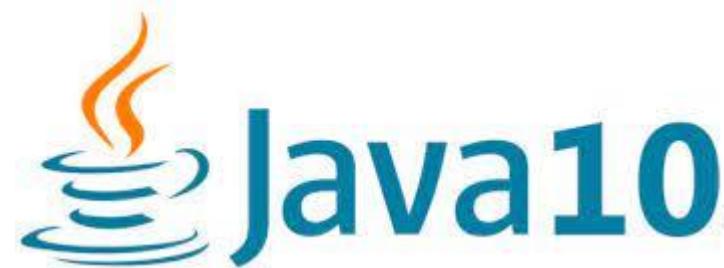
ModeloVistaControlador

ModeloVistaControlador

**Modelo****Vista****Controlador**

# Características de java9 – Java10 – Java11

---

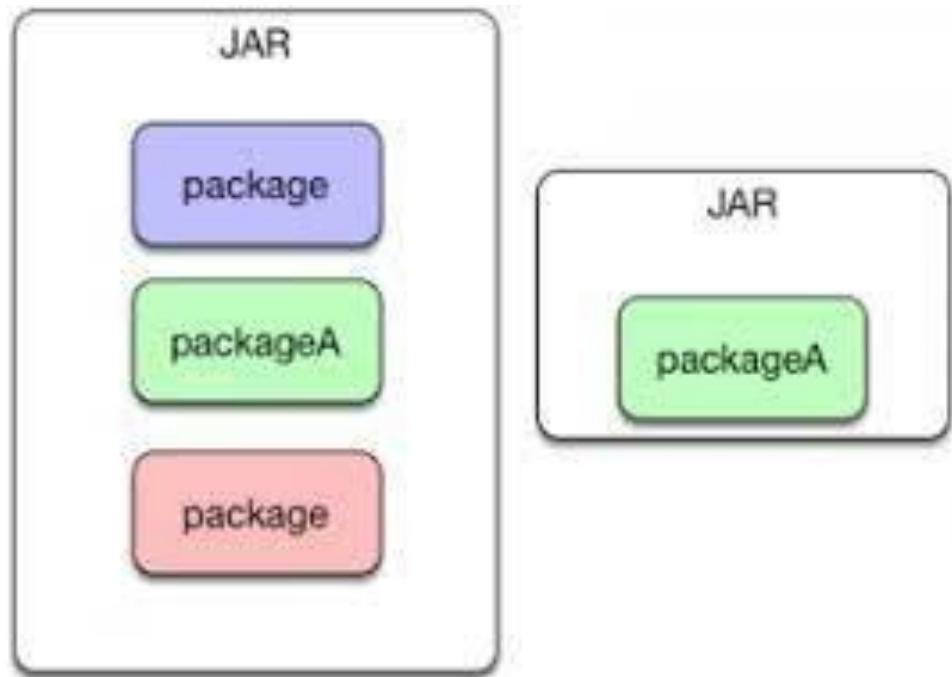


## Características - PROGRAMACION MODULAR

1. Métodos factoría para colecciones
2. Mejoras en la clase Optional
3. Mejoras en la API de streams
4. REPL con jshell
5. Jlink para generar runtimes mínimos
6. Concurrencia
7. Variables Handles
8. Actualizaciones en la API para procesos
9. StackWalker
11. Strings compactos
12. Recolector de basura G1 por defecto
13. Identificador para variables
14. Métodos privados en interfaces
15. Mejor try-with-resource
16. Javadoc
17. Archivos Jar multiversion
18. Nuevo modelo de publicacion

## MODULARIZACION

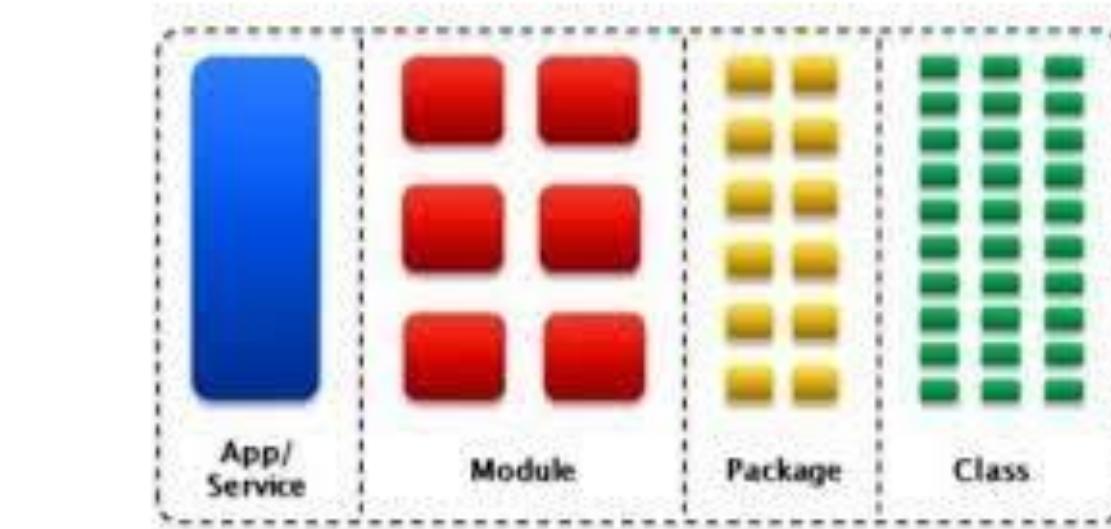
Los módulos van a mejorar una de las deficiencias existentes en la visibilidad de las clases entre paquetes. Los módulos de Java proporcionan una mayor encapsulación de las clases contenidas en un paquete y las librerías. Esta encapsulación evita que una aplicación u otra librería haga uso y dependa de clases y paquetes de los que no debería lo que mejora la compatibilidad con versiones futuras.



Los módulos proporcionan:

- Encapsulación fuerte:
- Interfaces bien definidas
- Dependencias explícitas

## Modularity in Java



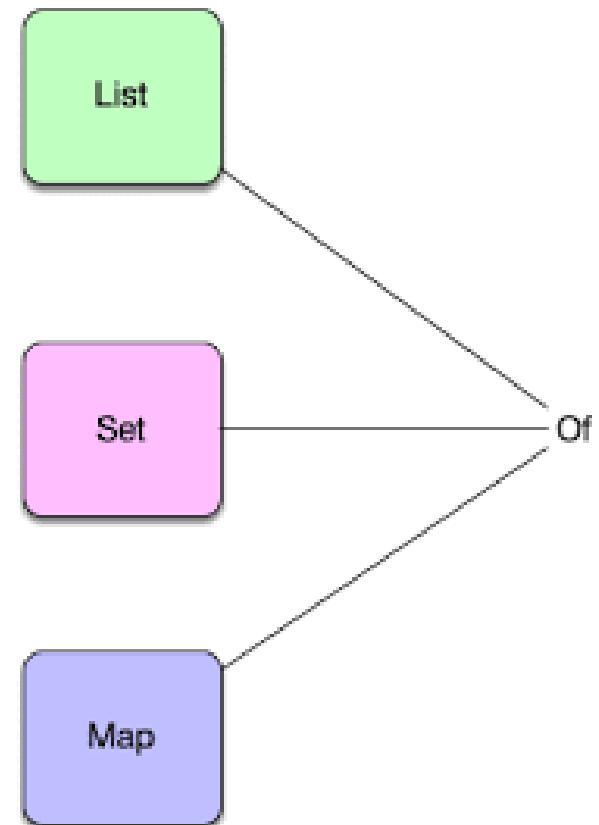
### **Los beneficios son:**

- Configuración confiable: el sistema de módulos comprueba si una combinación de módulos satisface todas las dependencias antes de compilar o ejecutar una aplicación.
- Encapsulación fuerte: se evitan dependencias sobre detalles internos de implementación.
- Desarrollo escalable: se crean límites entre el equipo que desarrolla un módulo y el que lo usa.
- Optimización: dado que el sistema de módulos sabe que módulos necesita cada uno solo se consideran los necesarios mejorándose tiempos de inicio y memoria consumida.
- Seguridad: la encapsulación y optimización limita la superficie de ataque.

## Métodos de factoría para colecciones

Aún Java no incorpora en el lenguaje una forma de definir como literales elementos tan comunes como listas, conjuntos o mapas. Como alternativa se proporcionan métodos factoría estáticos para crear este tipo de estructuras de datos usando métodos por defecto en sus respectivas interfaces. Además, estos métodos crean colecciones inmutables.

A parte de definir este tipo de colecciones de una forma mucho más sencilla que hasta Java 8, las colecciones además son significativamente más eficientes.



## Mejoras en el API de streams

- Los nuevos métodos de los streams dropWhile(), takeWhile() permiten descartar o tomar elementos del stream mientras se comprueba una condición.
- El método ofNullable() devuelve un stream de un elemento o vacío dependiendo de si el objeto es null o no.
- Los métodos iterate() permiten generar una secuencia de valores similar a un bucle for.



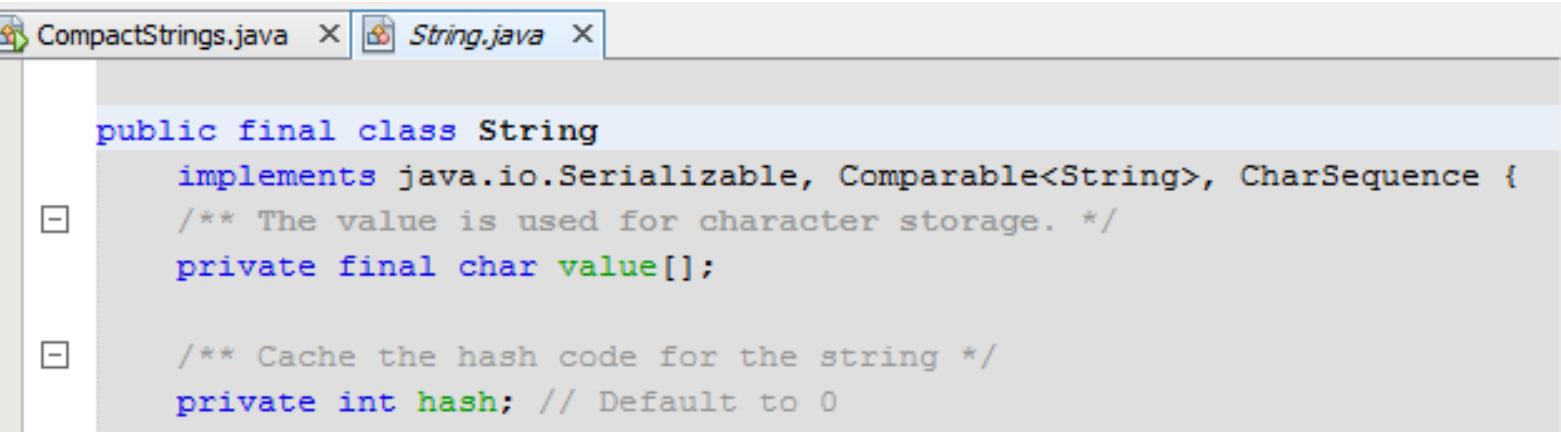
## STREAM

tabla donde se muestran algunos métodos de la interfaz Stream junto a una posible equivalencia en sql.

SQL	Interfaz Stream
from	stream()
select	map()
where	filter() (antes de un collecting)
order by	sorted()
distinct	distinct()
having	filter() (después de un collecting)
join	flatMap()
union	concat().distinct()
offset	skip()
limit	limit()
group by	collect(groupingBy())
count	count()

## Strings compactos

En Java , String es el tipo fundamental a la hora de trabajar con cadenas de caracteres. Este usa internamente un vector de tipo **char** para almacenar los caracteres, lo cual implica que cada uno de ellos ocupa **2 bytes**. De esta forma se facilita la codificación Unicode de 16 bits:



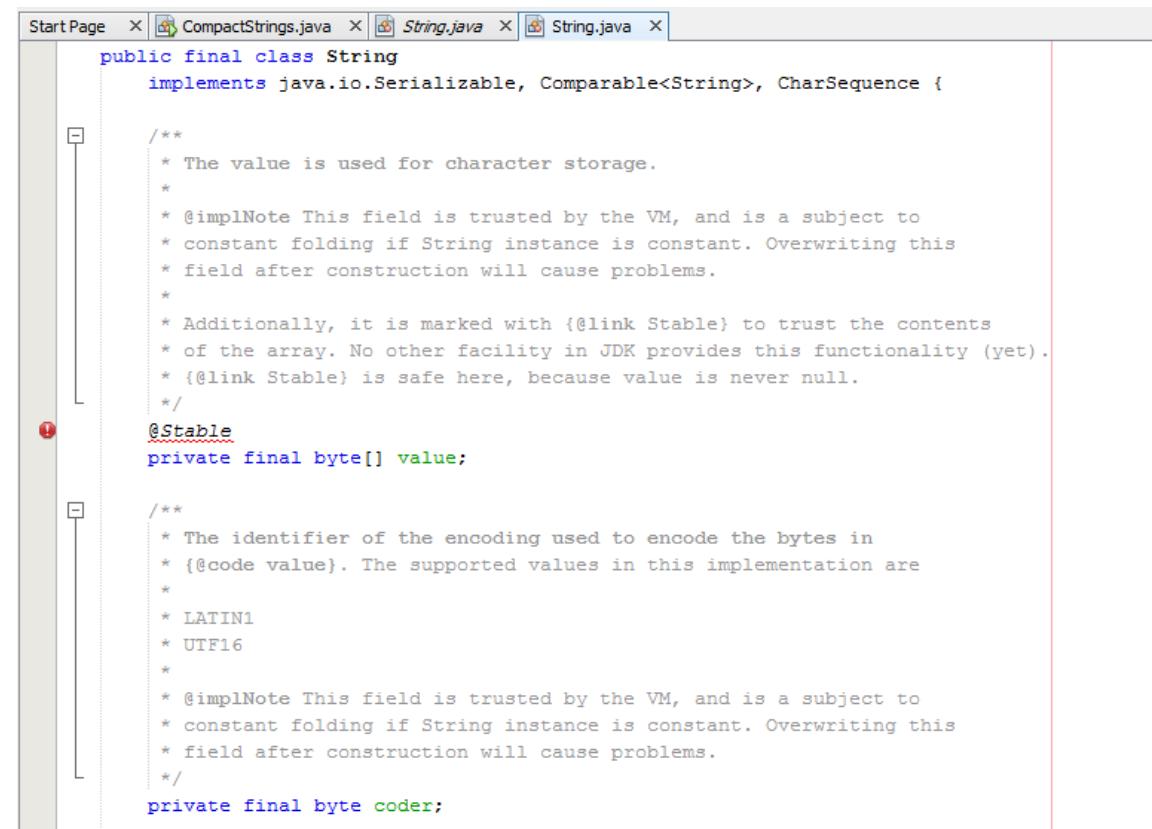
The screenshot shows a Java code editor with two tabs at the top: "CompactStrings.java" and "String.java". The "String.java" tab is active, displaying the following code:

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /* The value is used for character storage. */
    private final char value[];

    /* Cache the hash code for the string */
    private int hash; // Default to 0
```

## Strings compactos

Una de las novedades introducidas en Java 9 afecta a la representación del tipo String que ha cambiado el tipo char por byte para los elementos de la cadena. La clase String cuenta con un atributo adicional, llamado coder, que permite a los métodos que operan sobre cadenas saber si están utilizando uno o dos bytes por carácter. De esta forma se sigue dando soporte a alfabetos extendidos, al tiempo que se reduce la ocupación en memoria y mejora la eficiencia cuando se opera con alfabetos simplificados:



The screenshot shows a Java code editor with the file `String.java` open. The code defines the `String` class, which implements `Serializable`, `Comparable<String>`, and `CharSequence`. It contains two fields: `value` (a `byte[]`) and `coder` (also a `byte`). Both fields are annotated with `@implNote` and `@stable`. The `value` field is described as being used for character storage and subject to constant folding if the string is constant. The `coder` field is described as identifying the encoding used to encode the bytes in `value`.

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {

    /**
     * The value is used for character storage.
     *
     * @implNote This field is trusted by the VM, and is a subject to
     * constant folding if String instance is constant. Overwriting this
     * field after construction will cause problems.
     *
     * Additionally, it is marked with {@link Stable} to trust the contents
     * of the array. No other facility in JDK provides this functionality (yet).
     * {@link Stable} is safe here, because value is never null.
     */
    @Stable
    private final byte[] value;

    /**
     * The identifier of the encoding used to encode the bytes in
     * {@code value}. The supported values in this implementation are
     *
     * @code LATIN1
     * @code UTF16
     *
     * @implNote This field is trusted by the VM, and is a subject to
     * constant folding if String instance is constant. Overwriting this
     * field after construction will cause problems.
     */
    private final byte coder;
}
```

## Características

1. Inferencia de tipos en variables locales
2. Consolidar todos los repositorios del JDK en uno solo
3. Una interfaz limpia y unificada para el GarbageCollector
4. Thread Local-Handshakes
5. Extensiones de etiquetas de idioma Unicode adicionales
6. Versionamiento de releases basados en la fecha

## Inferencia de tipos en variables locales

La nueva inferencia de tipos introduce la palabra reservada **var** para declarar variables locales e inferir su tipo.

```
var list = new ArrayList<String>(); // infiere ArrayList<String>
var stream = list.stream();        // infiere Stream<String>
```

En resumen esto solo funciona en un ámbito local y con variables inicializadas.

## Inferencia de tipos en variables locales

Existe un numero de operaciones riesgosas en las cuales se podría caer con esto de la inferenciaa de tipos.

- 1. Variables sin inicializar:** Es una sintaxis válida, sin embargo, java no sabrá qué hacer con ella dado que no puede inferir el tipo de algo que no tiene un valor aún.
- 2. Expresiones lambda sin tipo objetivo:** Por un principio similar al anterior a una función lambda se le puede inferir el tipo siempre y cuando esta diga que tipo es el objetivo.
- 3. Null no es inferible:** no podemos inferir que tipo es null.
- 4. Los arreglos necesitan un tipo explícito:** Cuando definimos arrays en java 10, no podemos inferir el tipo si lo hacemos de la forma abreviada. Es decir la inferencia no sería type-safe en este caso.

```
var x;
```

```
var f = () -> {};  
var m = this::l;
```

```
var g = null;
```

```
var k = { 1 , 2 };
```

## Características

1. Nueva sintaxis para parámetros Lambda
2. Recolectores de basura
3. Inicie programas de código fuente de un solo archivo
4. Actualizaciones de seguridad
5. Mejoras de JVM
6. Actualizaciones de API de archivos

## Sintaxis de variables locales para parámetros en lambdas

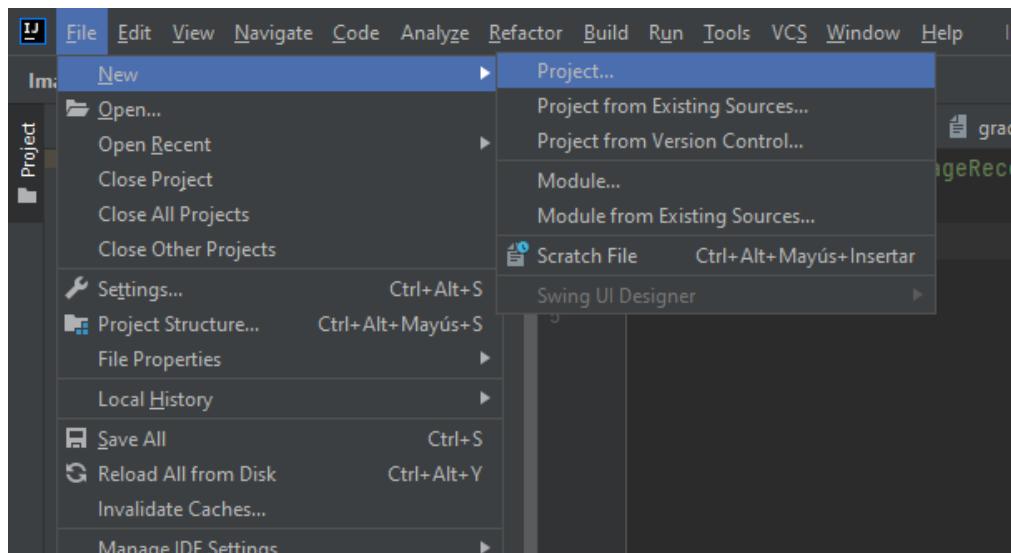
Ahora los parámetros de una lambda pueden declararse con **var** con inferencia de tipos. Esto proporciona uniformidad en el lenguaje al declarar los parámetros permite usar anotaciones en los parámetros de la función lambda como @NotNull. Esta funcionalidad tiene algunas restricciones. No se puede mezclar el uso y no uso de var y no se puede mezclar el uso de var y tipos en lambdas explícitas. Son consideradas ilegales por el compilador y producirá un error en tiempo de compilación.

```
1 (x, y) -> x.process(y)
2
3 (var x, var y) -> x.process(y)
4
5 (@NotNull var x, @NotNull var y) -> x.process(y)
6
7 (var x, y) -> x.process(y) // no se puede mezclar var y no var
8 (var x, int y) -> x.process(y) // no se puede mezclar var y tipos en lambdas explícitas
```

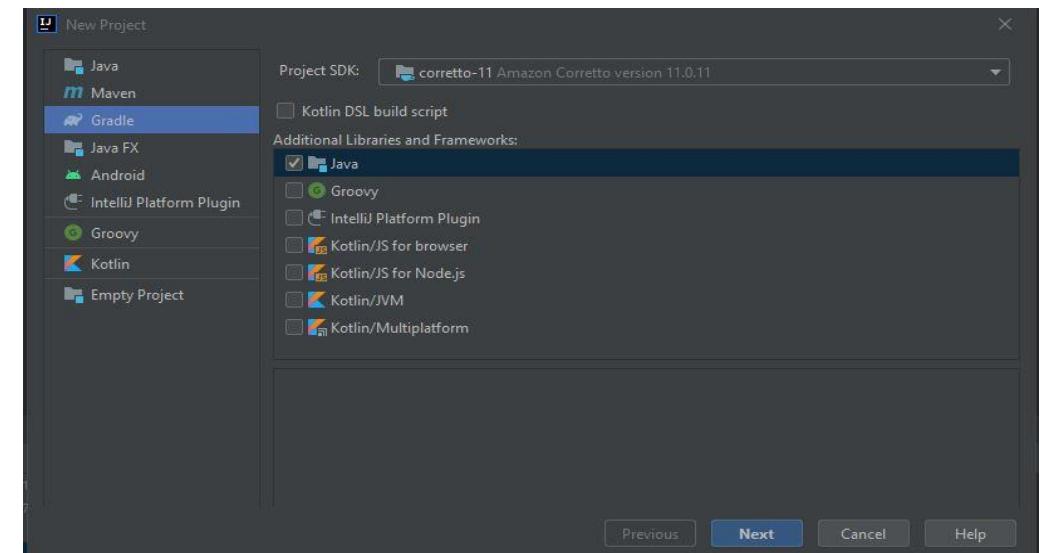
# **CREAR PROYECTO MULTI MODULO USANDO INTELLIJ IDEA Y JAVA 9**

# MODULARIZACION

1 Crear un proyecto abriendo intelliJ idea y seleccionando  
File -> New -> Project

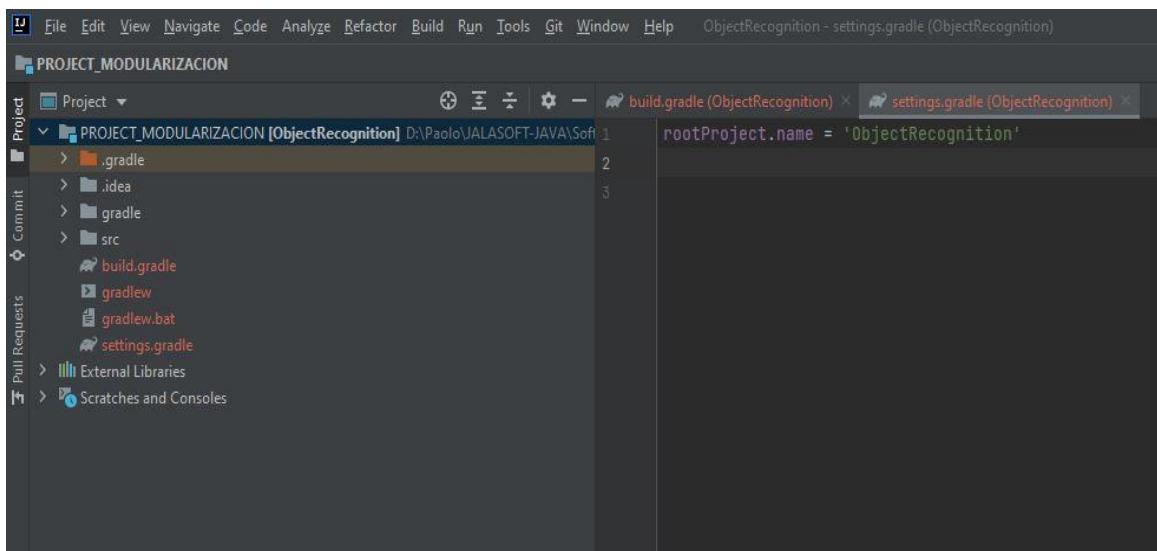


2 Verifique que gradle y java estén seleccionados

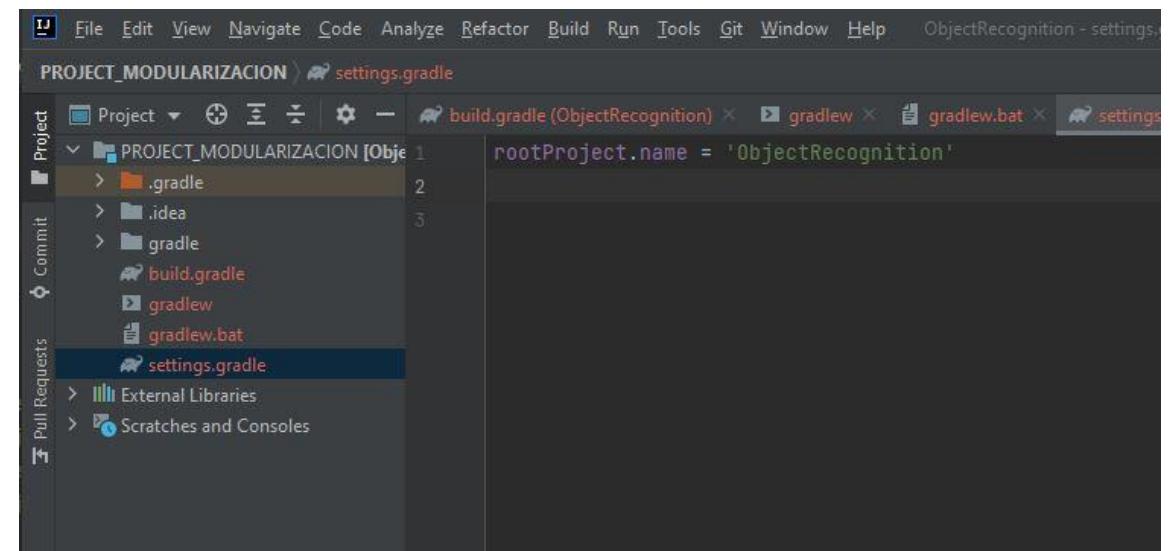


# MODULARIZACION

3 Una vez creado el proyecto tendrá la siguiente estructura.

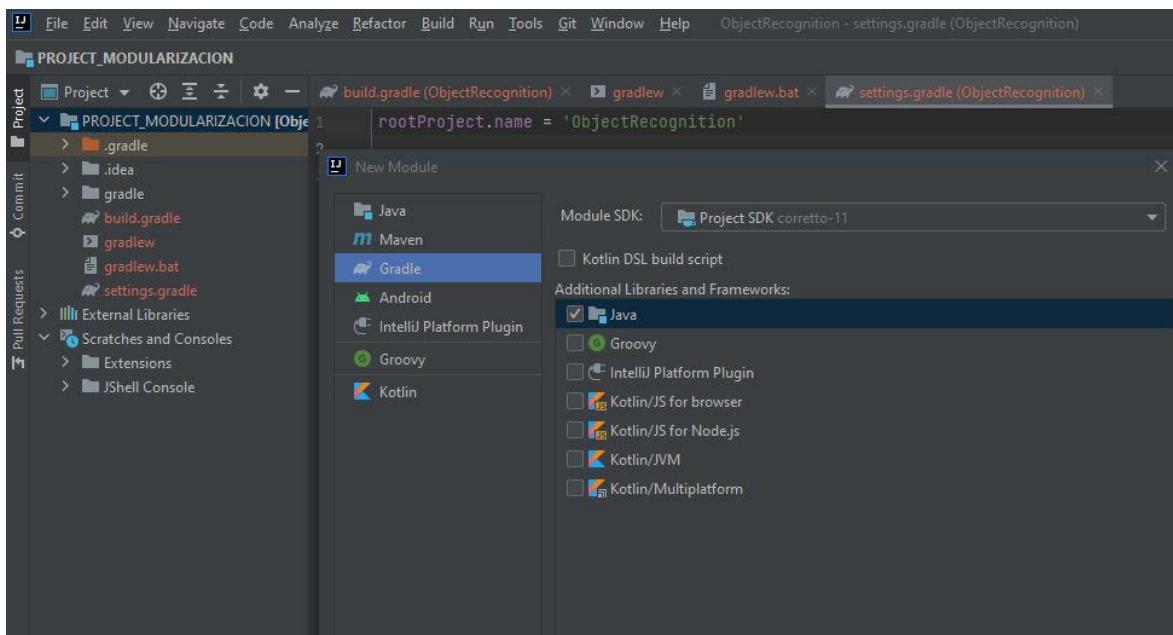


4 Remover la carpeta **src** que fue creada por defecto

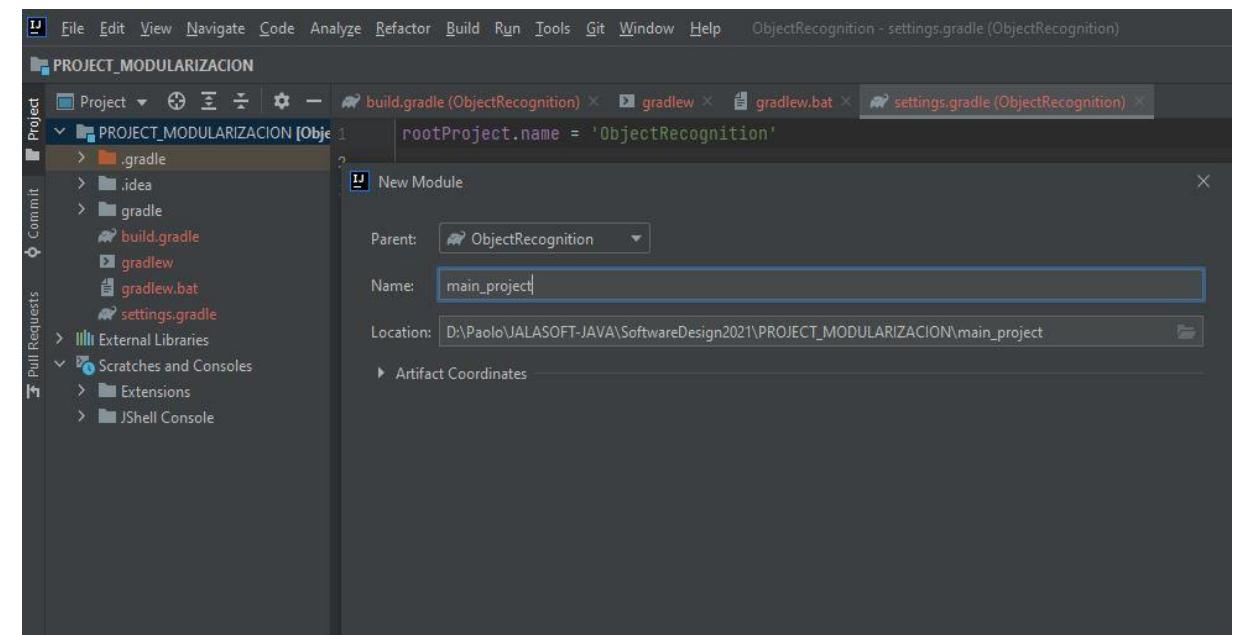


# MODULARIZACION

5 Crea modulo: File -> New -> Module...

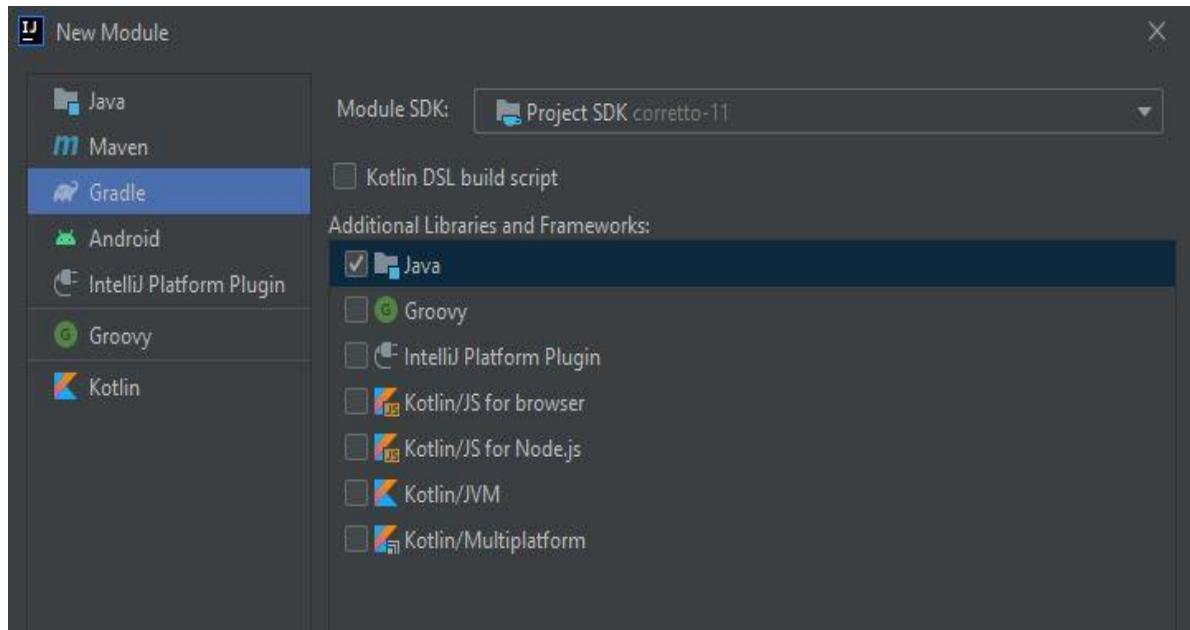


6 Crear modulo con el nombre de: main\_project

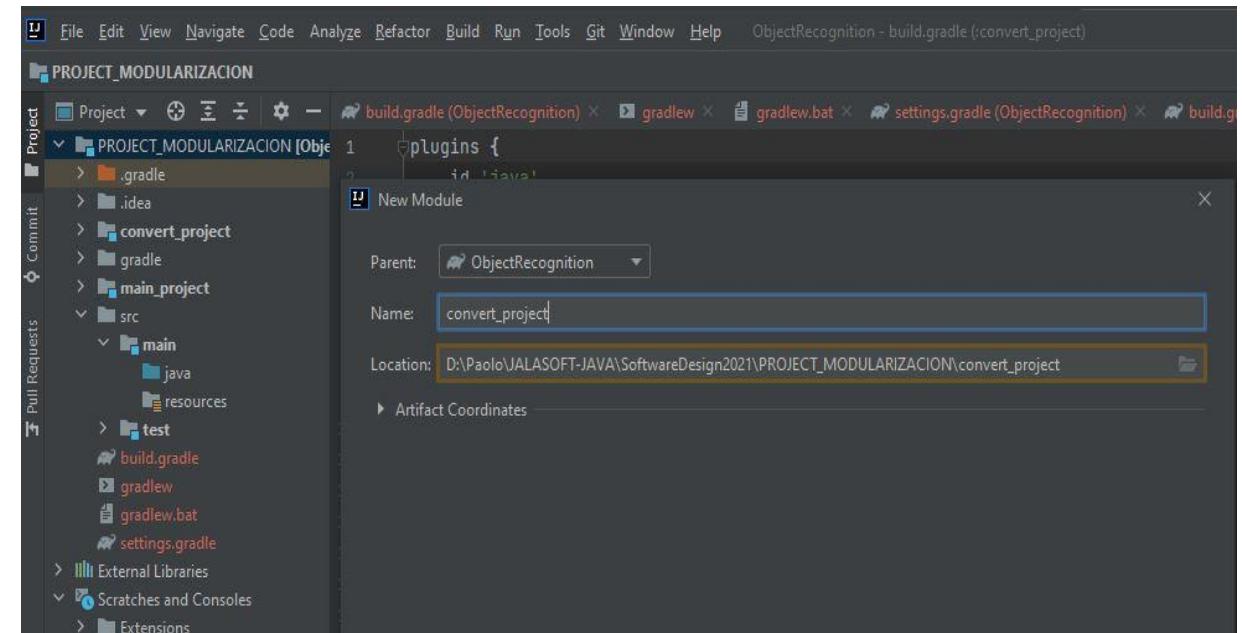


# MODULARIZACION

7 Crea otro verificando que este seleccionado gradle y java

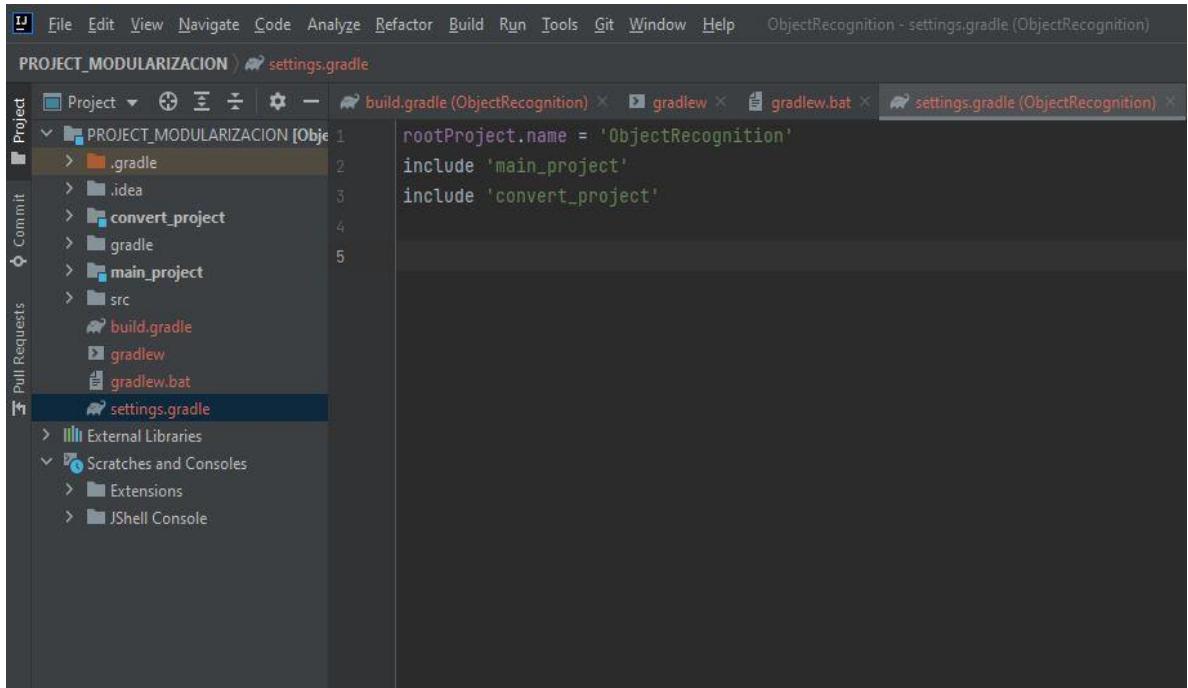


8 Crear modulo con el nombre de: convert\_project

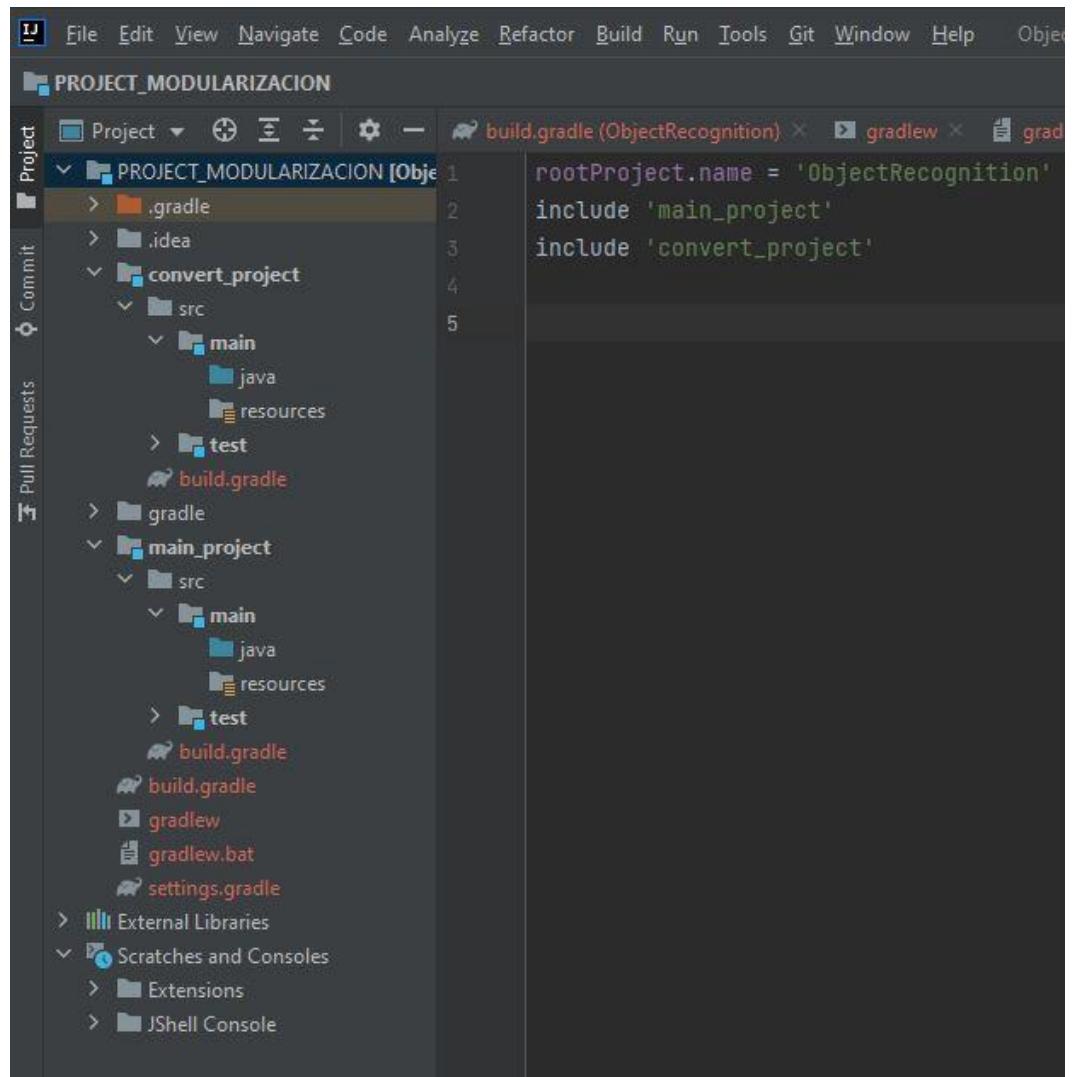


## MODULARIZACION

9 Verificar que en archivo settings.gradle esta incluyendo los 2 módulos creados

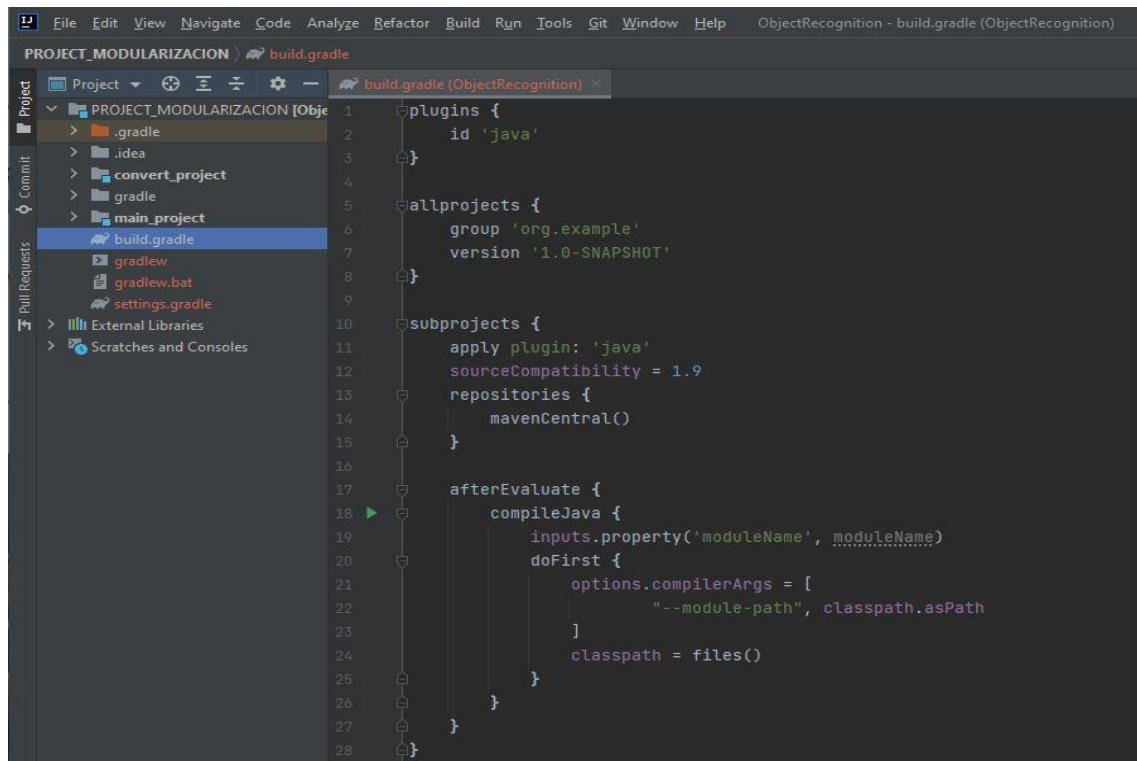


10 Verifique que el proyecto ahora contiene la siguiente estructura



## MODULARIZACION

11 Modifique el archivo build.gradle que se encuentra en la raíz del proyecto para que contenga lo siguiente

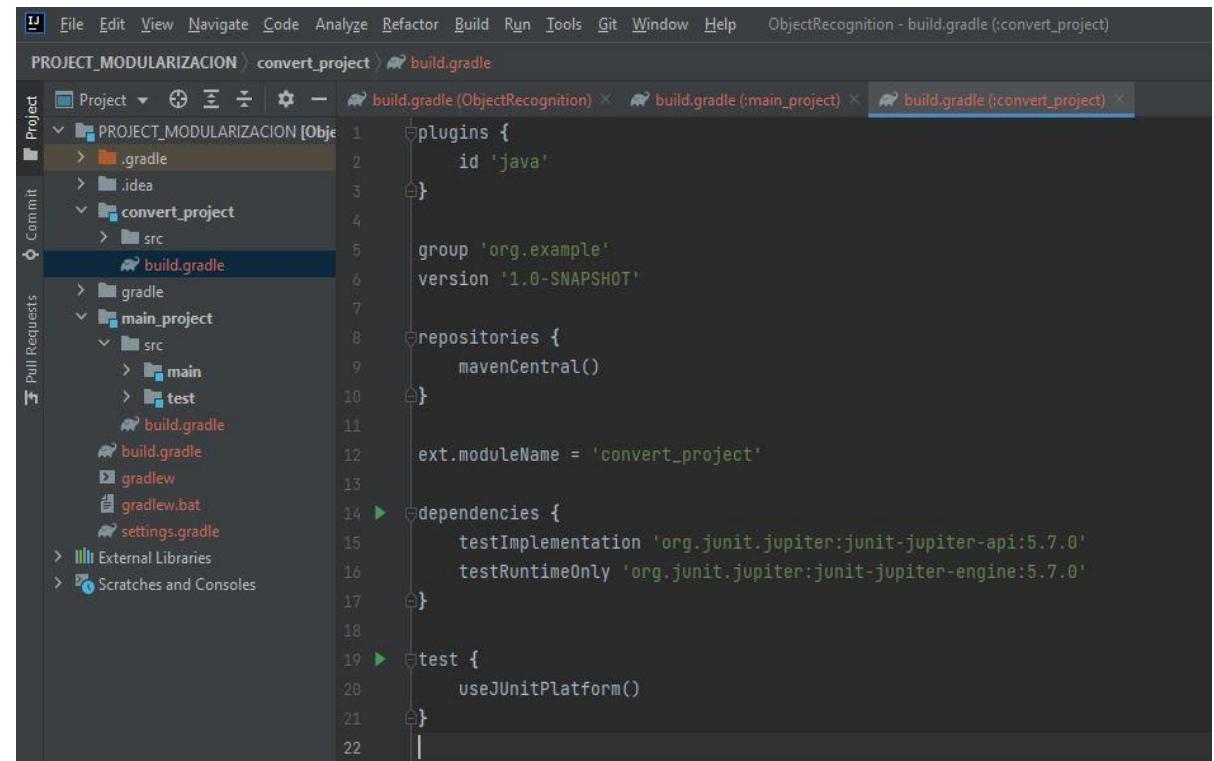


```
PROJECT_MODULARIZACION > build.gradle (ObjectRecognition)
File Edit View Navigate Code Analyze Refactor Build Run Tools Git Window Help ObjectRecognition - build.gradle (ObjectRecognition)

Project build.gradle (ObjectRecognition) ×
  PROJECT_MODULARIZACION [Objetos]
    .gradle
    .idea
    convert_project
    gradle
    settings.gradle
  main_project
    build.gradle
    gradlew
    gradlew.bat
    settings.gradle
  External Libraries
  Scratches and Consoles

build.gradle (ObjectRecognition) ×
1 plugins {
2     id 'java'
3 }
4
5 allprojects {
6     group 'org.example'
7     version '1.0-SNAPSHOT'
8 }
9
10 subprojects {
11     apply plugin: 'java'
12     sourceCompatibility = 1.9
13     repositories {
14         mavenCentral()
15     }
16
17     afterEvaluate {
18         compileJava {
19             inputs.property('moduleName', moduleName)
20             doFirst {
21                 options.compilerArgs = [
22                     '--module-path', classpath.asPath
23                 ]
24                 classpath = files()
25             }
26         }
27     }
28 }
```

12 Modifique el archivo build.gradle que se encuentra dentro del modulo convert\_project para que contenga lo siguiente



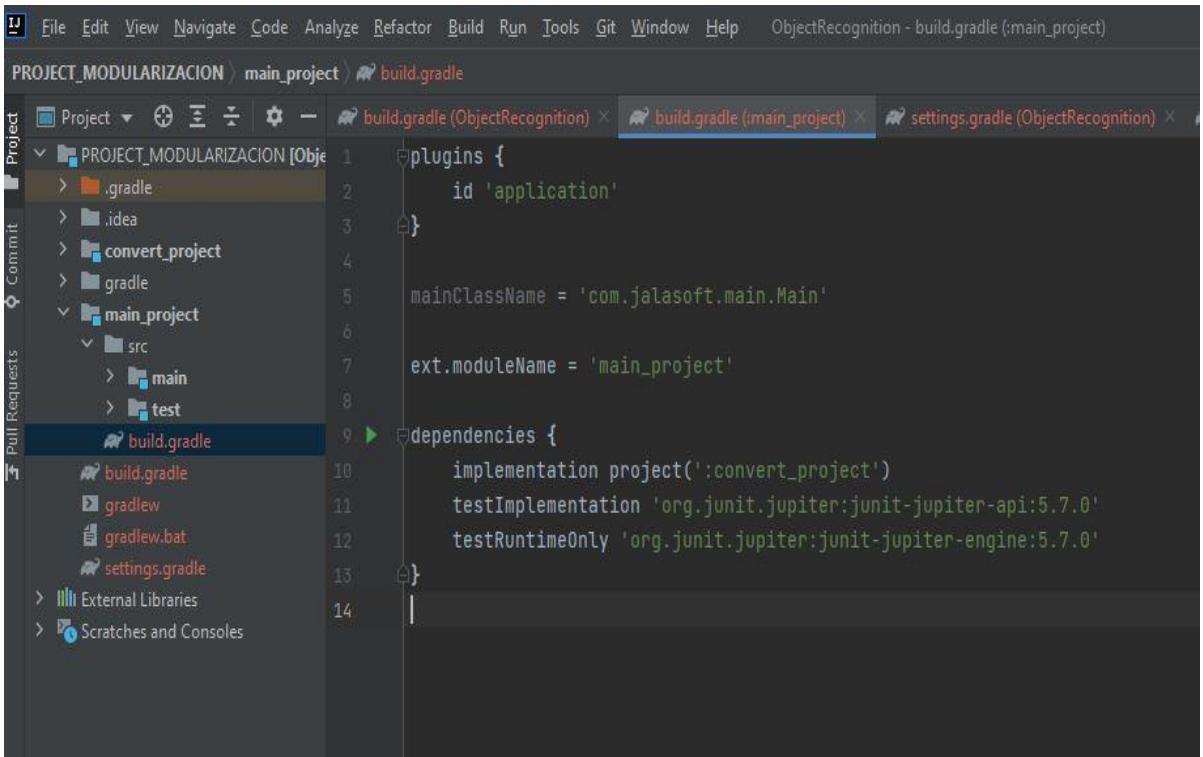
```
PROJECT_MODULARIZACION > convert_project > build.gradle (ObjectRecognition) ×
File Edit View Navigate Code Analyze Refactor Build Run Tools Git Window Help ObjectRecognition - build.gradle (:convert_project)

Project build.gradle (ObjectRecognition) ×
  build.gradle (main_project) ×
  build.gradle (:convert_project) ×
  PROJECT_MODULARIZACION [Objetos]
    .gradle
    .idea
    convert_project
      src
        build.gradle
    main_project
      src
        main
        test
        build.gradle
      build.gradle
      gradlew
      gradlew.bat
      settings.gradle
    External Libraries
    Scratches and Consoles

build.gradle (:convert_project) ×
1 plugins {
2     id 'java'
3 }
4
5 group 'org.example'
6 version '1.0-SNAPSHOT'
7
8 repositories {
9     mavenCentral()
10 }
11
12 ext.moduleName = 'convert_project'
13
14 dependencies {
15     testImplementation 'org.junit.jupiter:junit-jupiter-api:5.7.0'
16     testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.7.0'
17 }
18
19 test {
20     useJUnitPlatform()
21 }
22
```

## MODULARIZACION

13 Modifique el archivo build.gradle que se encuentra en el modulo main\_Project para que contenga lo siguiente



```
File Edit View Navigate Code Analyze Refactor Build Run Tools Git Window Help ObjectRecognition - build.gradle (:main_project)

PROJECT_MODULARIZACION > main_project > build.gradle

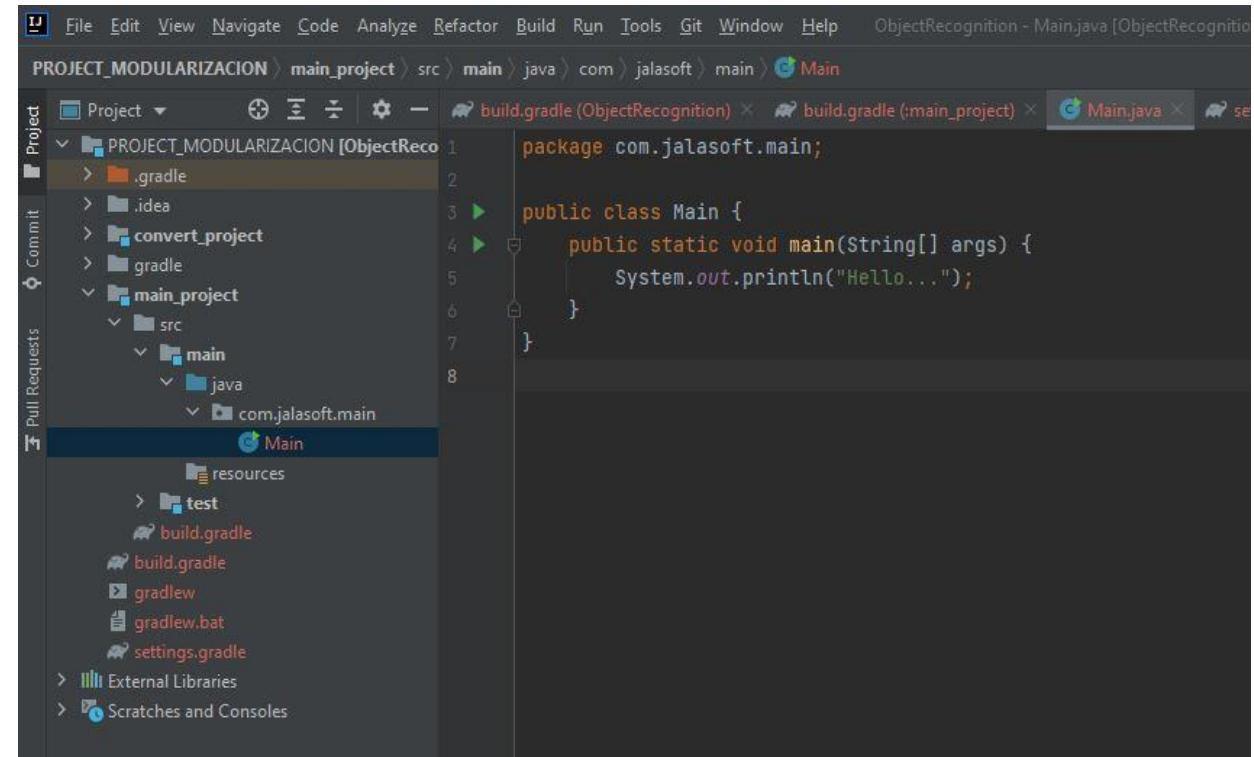
Project Commit Pull Requests

PROJECT_MODULARIZACION [ObjectRecognition]
  > .gradle
  > .idea
  > convert_project
  > gradle
  > main_project
    > src
      > main
      > test
        > build.gradle
        > build.gradle
        > gradlew
        > gradlew.bat
        > settings.gradle
  > External Libraries
  > Scratches and Consoles

build.gradle (ObjectRecognition) > build.gradle (main_project) > settings.gradle (ObjectRecognition)

1 plugins {
2     id 'application'
3
4     mainClassName = 'com.jalasoft.main.Main'
5
6     ext.moduleName = 'main_project'
7
8     dependencies {
9         implementation project(':convert_project')
10        testImplementation 'org.junit.jupiter:junit-jupiter-api:5.7.0'
11        testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.7.0'
12
13    }
14 }
```

14 Agregue la clase principal en el modulo main\_Project con la siguiente información.



```
File Edit View Navigate Code Analyze Refactor Build Run Tools Git Window Help ObjectRecognition - Main.java [ObjectRecognition]

PROJECT_MODULARIZACION > main_project > src > main > java > com > jalasoft > main > Main.java

Project Commit Pull Requests

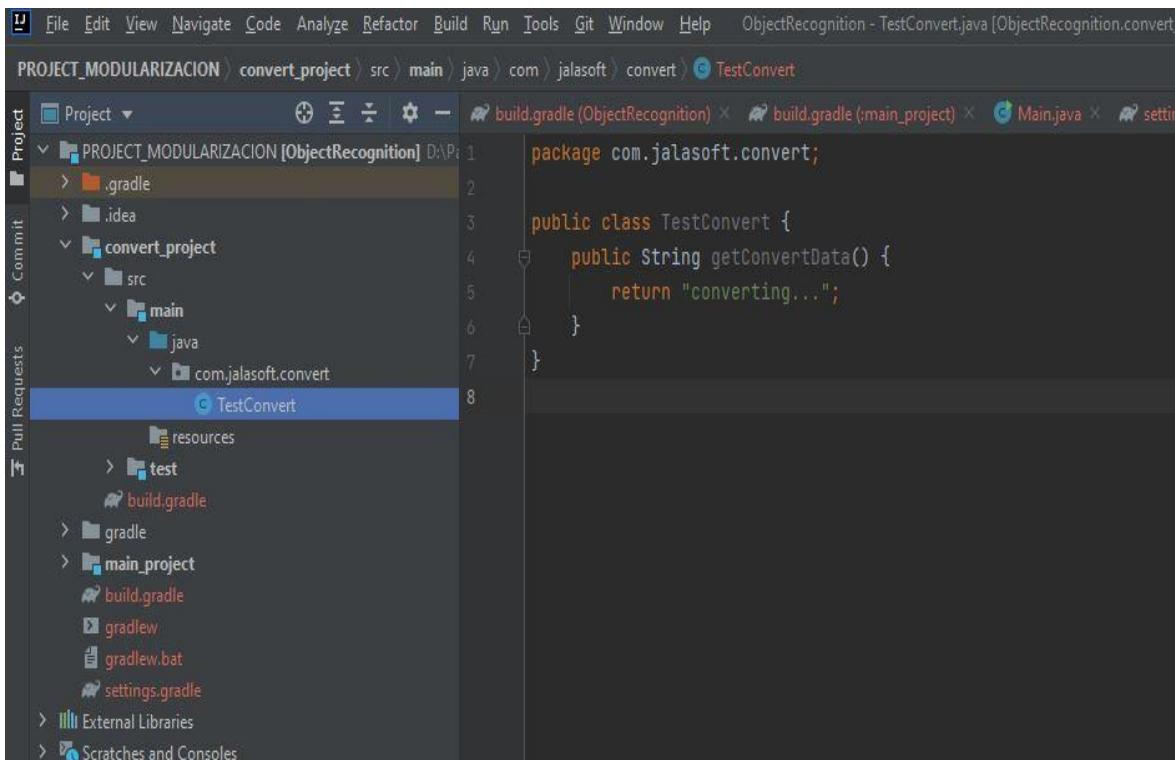
PROJECT_MODULARIZACION [ObjectRecognition]
  > .gradle
  > .idea
  > convert_project
  > gradle
  > main_project
    > src
      > main
        > java
          > com.jalasoft.main
            > Main.java
            > resources
              > test
                > build.gradle
                > build.gradle
                > gradlew
                > gradlew.bat
                > settings.gradle
  > External Libraries
  > Scratches and Consoles

Main.java

1 package com.jalasoft.main;
2
3 public class Main {
4     public static void main(String[] args) {
5         System.out.println("Hello...");
6     }
7 }
8 
```

## MODULARIZACION

15 Agregar el archivo TestConvert dentro del modulo convert\_Project con la siguiente informacion

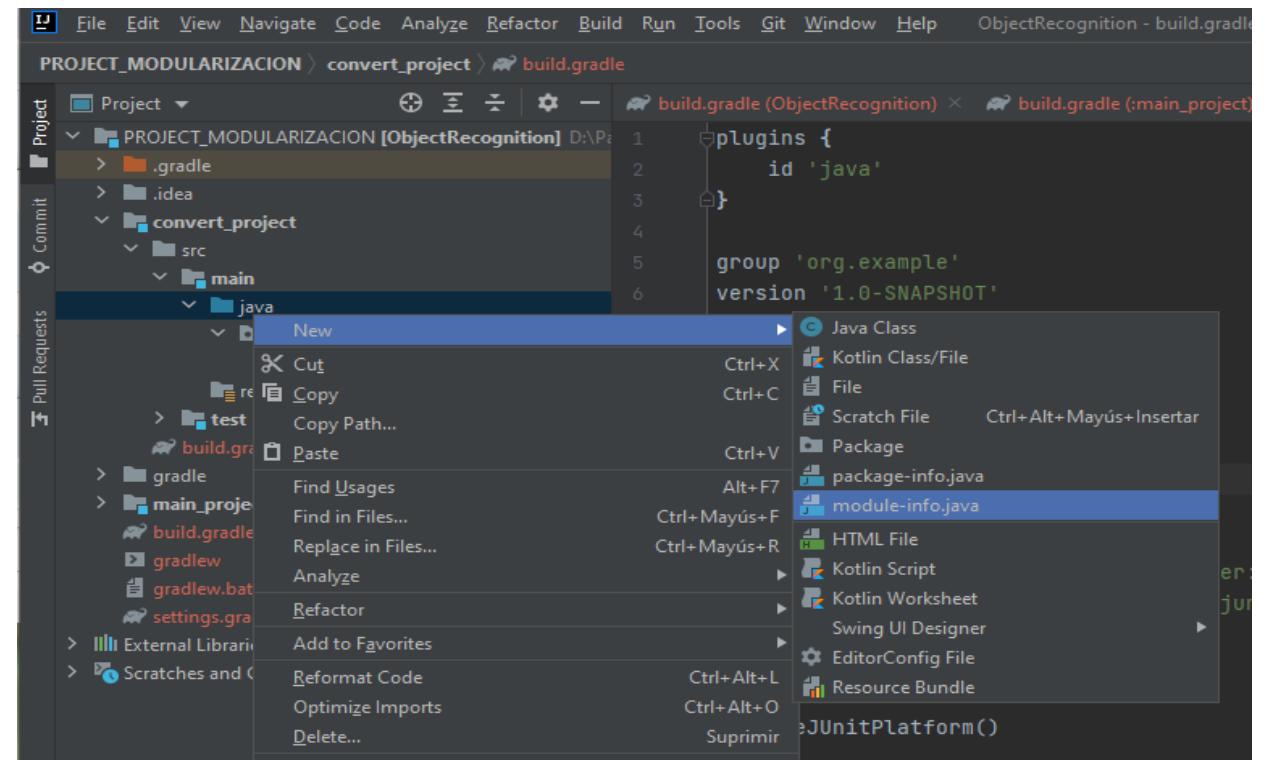


The screenshot shows the IntelliJ IDEA interface with the project navigation bar at the top. Below it, the project structure is displayed. A file named 'TestConvert.java' is selected in the 'convert\_project' module's 'src/main/java/com/jalasoft/convert' package. The code within the file is:

```
package com.jalasoft.convert;

public class TestConvert {
    public String getConvertData() {
        return "converting...";
    }
}
```

16 Agregar el archivo descriptor module-info.java dentro del proyecto convert\_Project en la ruta: convert\_Project -> src -> main -> java



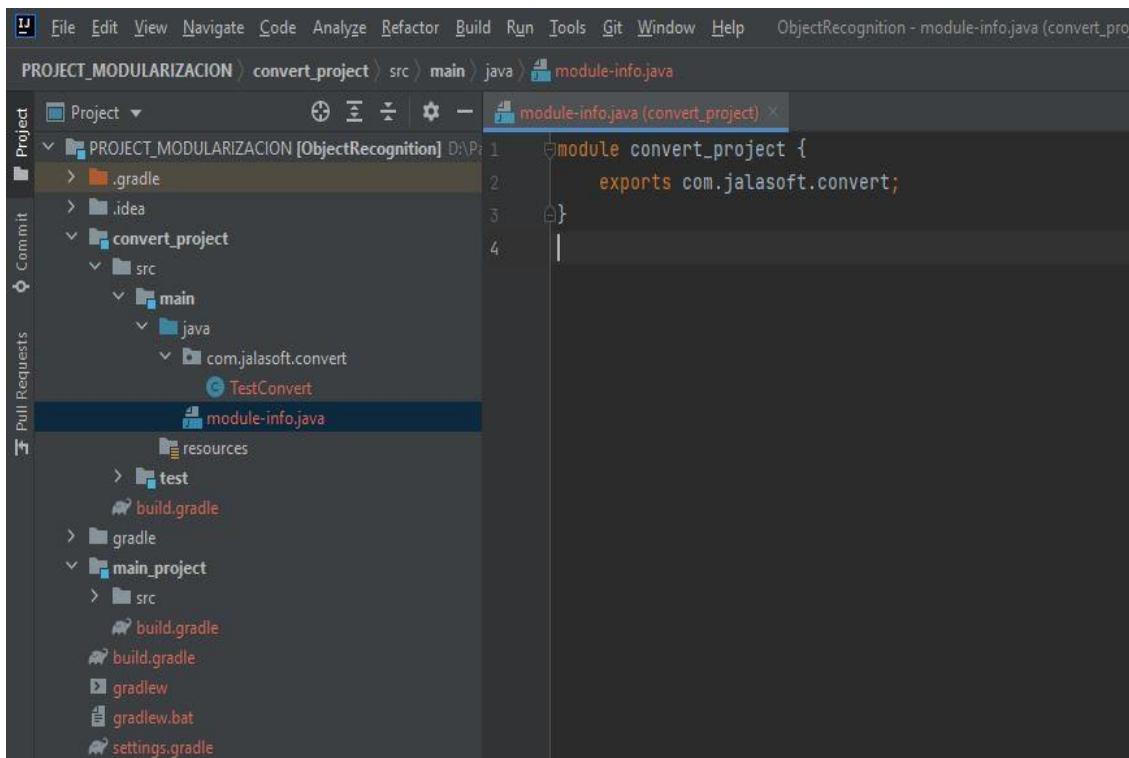
The screenshot shows the IntelliJ IDEA interface with the project navigation bar at the top. Below it, the project structure is displayed. A context menu is open over the 'java' directory in the 'convert\_project' module's 'src/main' path. The 'New' option is selected, and a submenu is open, showing various options. The 'module-info.java' option is highlighted.

```
plugins {
    id 'java'
}

group 'org.example'
version '1.0-SNAPSHOT'
```

# MODULARIZACION

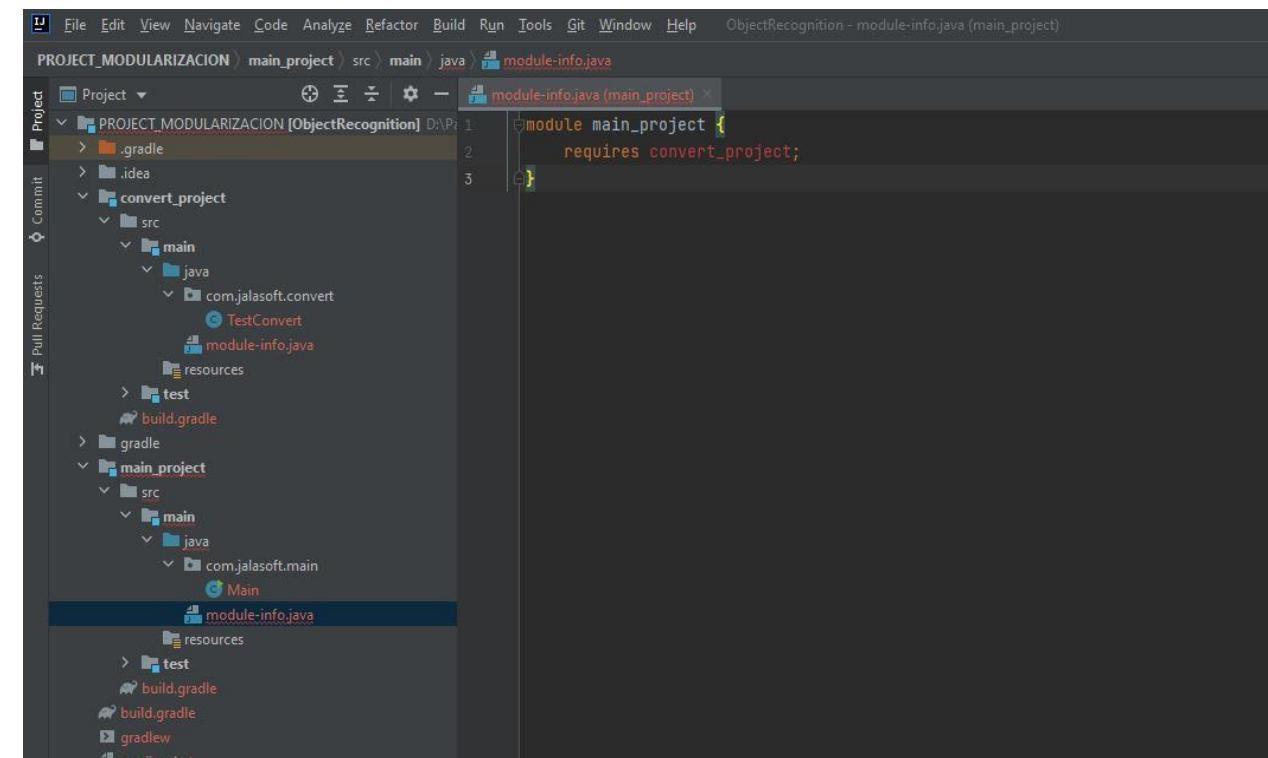
17 dentro del descriptor para el modulo convert\_Project exportar el paquete com.jalasoft.convert



The screenshot shows the Android Studio interface with the project 'PROJECT\_MODULARIZACION' selected. In the left sidebar, under 'convert\_project', there is a 'src' folder containing a 'main' folder with a 'java' folder. Inside 'java' is a package named 'com.jalasoft.convert' which contains a class 'TestConvert'. Below 'java' is another 'module-info.java' file. The code in 'module-info.java' is:

```
module convert_project {  
    exports com.jalasoft.convert;  
}
```

17 Crear un descriptor dentro del modulo main\_Project y agregar el siguiente contenido



The screenshot shows the Android Studio interface with the project 'PROJECT\_MODULARIZACION' selected. In the left sidebar, under 'main\_project', there is a 'src' folder containing a 'main' folder with a 'java' folder. Inside 'java' is a package named 'com.jalasoft.main' which contains a class 'Main'. Below 'java' is another 'module-info.java' file. The code in 'module-info.java' is:

```
module main_project {  
    requires convert_project;  
}
```

# MODULARIZACION

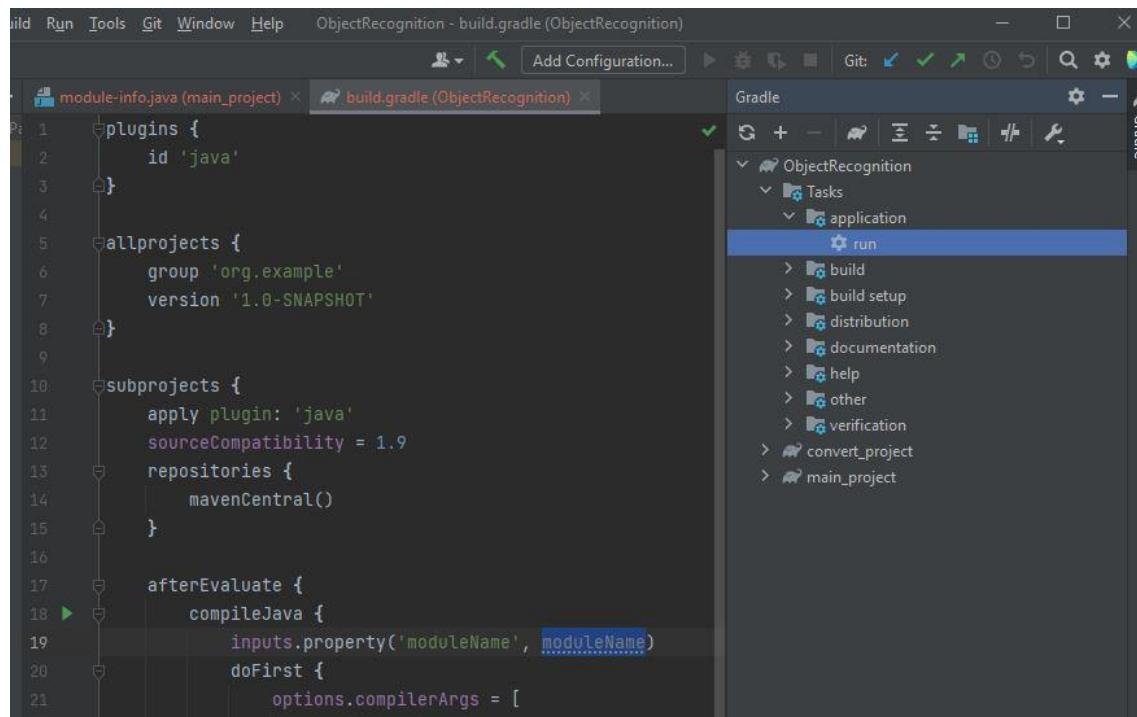
19 Verificar dentro de Tasks se encuentra la opción application caso contrario seleccione el botón de refresh

The screenshot shows an IDE interface with the following details:

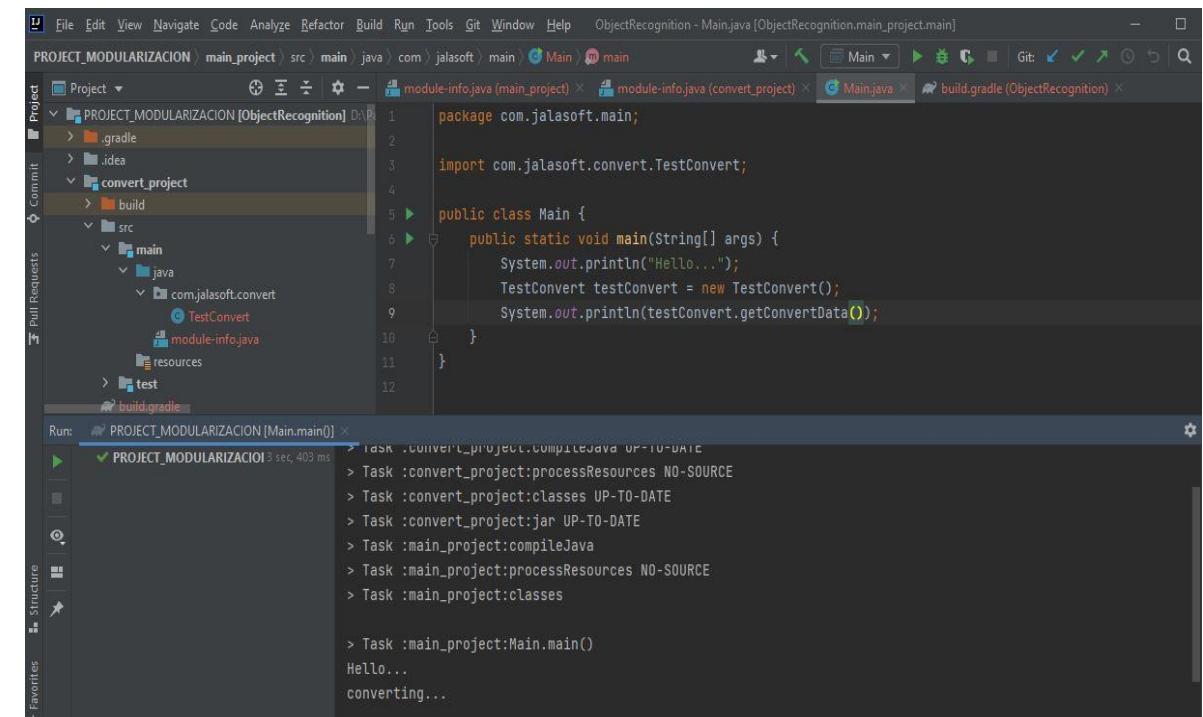
- Project View:** On the left, it shows the project structure under "PROJECT\_MODULARIZACION [ObjectRecognition]". It includes a ".gradle" folder, an ".idea" folder, a "convert\_project" module containing "src/main/java/com.jalasoft.convert/TestConvert/module-info.java" and "resources", and a "main\_project" module containing "src/main/java/com.jalasoft.main/Main/module-info.java" and "resources". Both modules have "build.gradle" files.
- Code Editor:** The main editor window displays the content of the "build.gradle" file for the "ObjectRecognition" module. The code defines a Java plugin, sets up a group and version for all projects, and configures repositories and compiler arguments for the main project's Java source code.
- Gradle Task List:** On the right, the "Gradle" tool window is open, showing the "Tasks" section for the "ObjectRecognition" module. The tasks listed include "application", "build", "build setup", "distribution", "documentation", "help", "other", and "verification".

# MODULARIZACION

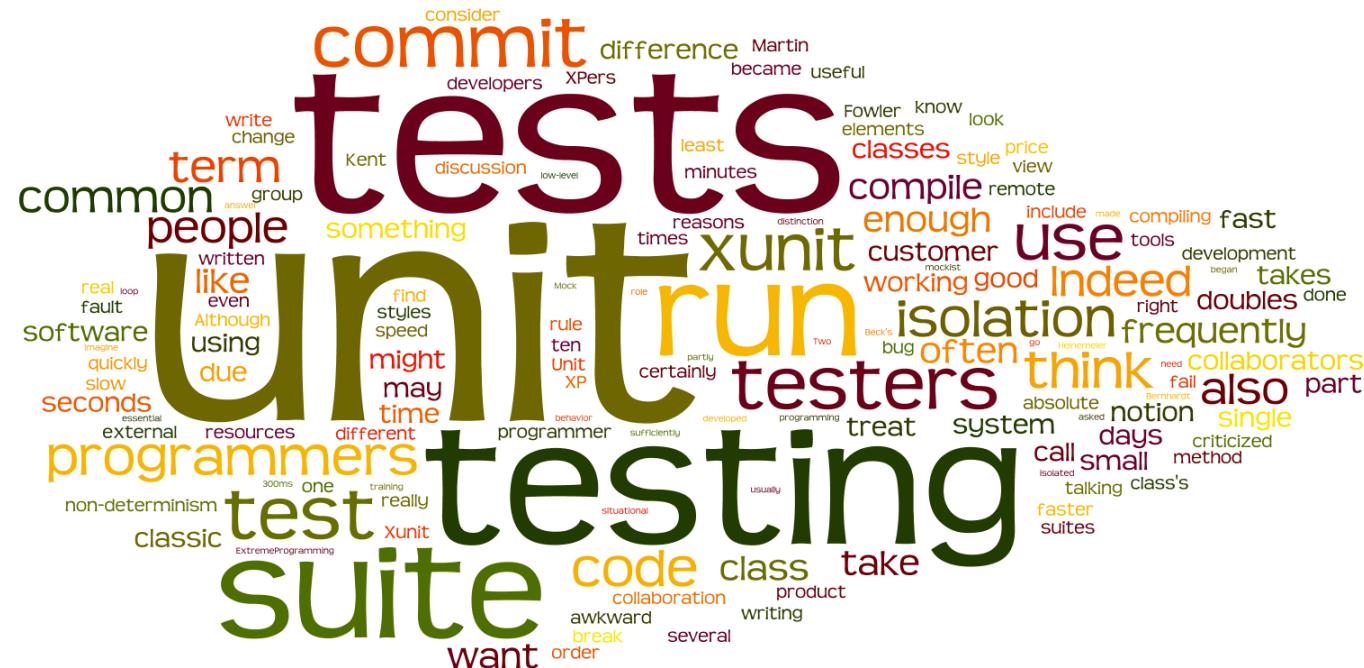
20 ejecutar la opción run dentro de task -> application



21 En el archivo Main usar las clases creadas en el modulo convert\_project

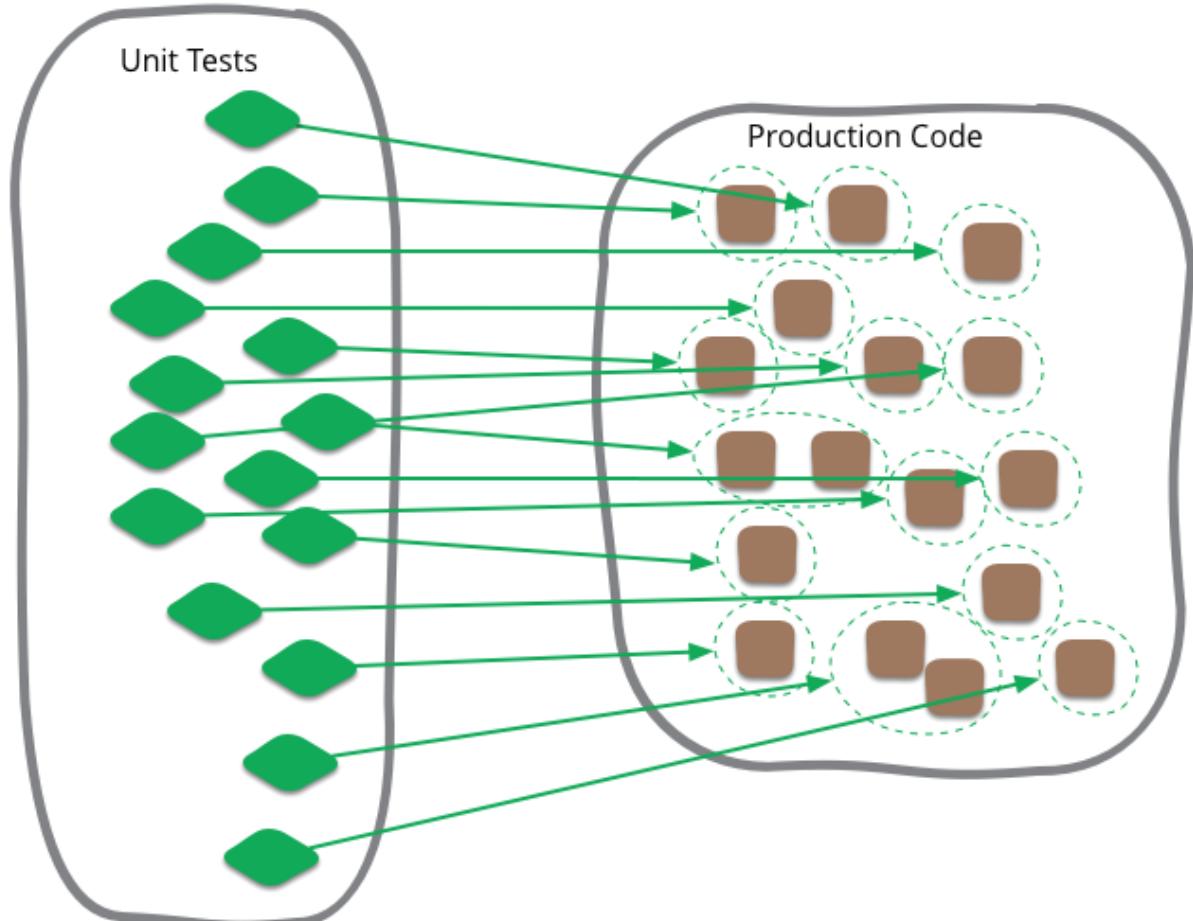


# Pruebas unitarias



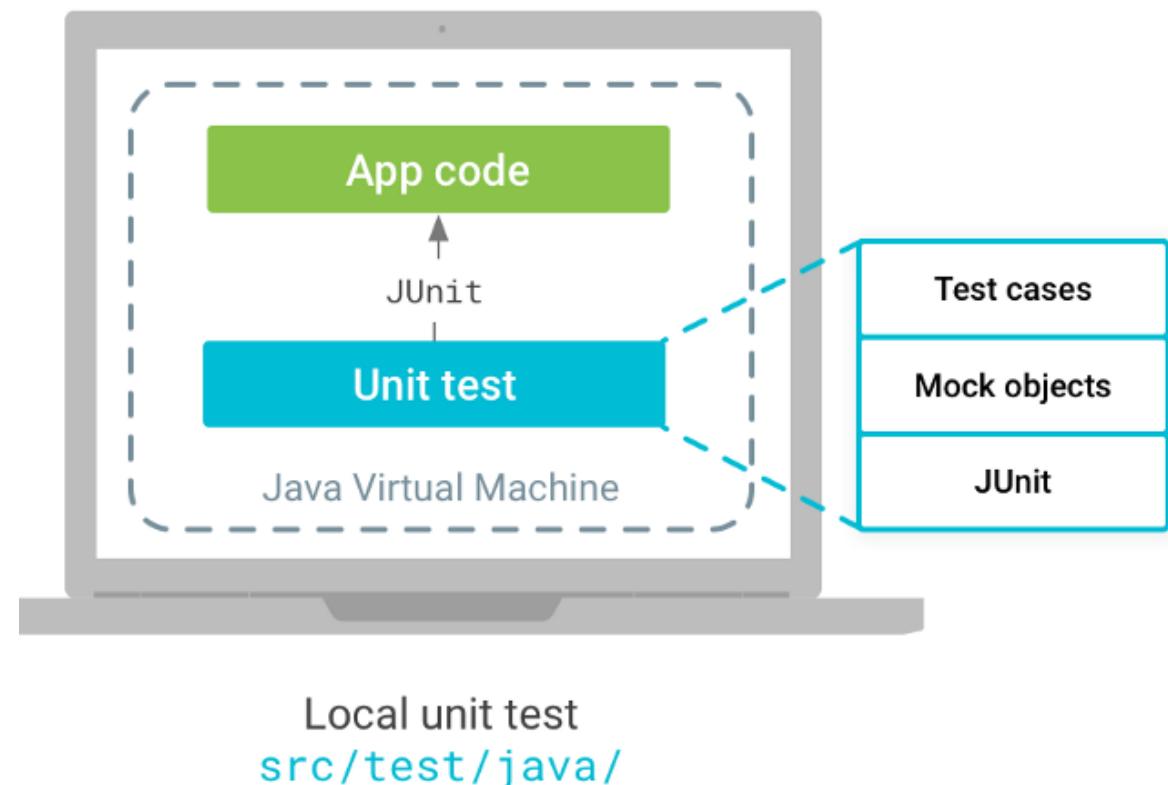
### ¿Qué es una prueba unitaria?

- Es una prueba vinculada al código.
- Una clase cuyo métodos ejercitan los métodos de otra.
- Estas clases se integran en un proyecto diferente llamado pruebas(test).



### ¿Por qué probar las aplicaciones?

- Las pruebas no son el remedio de los errores, pero incrementan la calidad.
- Una prueba es un testigo de comprobación.
- Mucho más útiles según evoluciona el software.
- Debemos complementar con buenas prácticas y prácticas adquiridas



JUnit es un framework Java para implementar test en Java. Se basa en **anotaciones**:

- **@Test**: indica que el método que la contiene es un test: expected y timeout.
- **@Before**: ejecuta el método que la contiene justo antes de cada test.
- **@After**: ejecuta el método que la contiene justo después de cada test.
- **@BeforeClass**: ejecuta el método (estático) que la contiene justo antes del **primer test**.
- **@AfterClass**: ejecuta el método (estático) que la contiene justo después del **último test**.
- **@Ignore**: evita la ejecución del tests. No es muy recomendable su uso porque puede ocultar test fallidos. Si dudamos si el test debe estar o no, quizás borrarlo es la mejor de las decisiones.

Las **condiciones de aceptación** del test se implementa con los asserts. Los más comunes son los siguientes:

- **assertTrue/assertFalse** (condición a testear): Comprueba que la condición es cierta o falsa.
- **assertEquals/assertNotEquals** (valor esperado, valor obtenido). Es importante el orden de los valores esperado y obtenido.
- **assertNull/assertNotNull (object)**: Comprueba que el objeto obtenido es nulo o no.
- **assertSame/assertNotSame(object1, object2)**: Comprueba si dos objetos son iguales o no.
- **fail()**: Fuerza que el test termine con fallo. Se puede indicar un mensaje.

## Estructura de las pruebas unitarias

Por norma general, los unit test deberían seguir una estructura AAA, son las siglas de:

- 1. Arrange (Organizar):** En esta parte de la prueba, debes establecer las condiciones iniciales para poder realizarla, así como el resultado que esperas obtener. Y esto significa por ejemplo, declarar las variables y crear las instancias de los objetos.
- 2. Act (Accionar):** Es la parte de ejecución, tanto del fragmento de código de la prueba, como del fragmento de código a testear.
- 3. Assert (Comprobar):** Por último, se realiza la comprobación para verificar que el resultado obtenido, coincide con el esperado.

```
public void Test()
{
    // Arrange
    var p1 = new Person();
    var p2 = new Person();
    Session.Save(p1);
    Session.Save(p2);

    // Act
    var result = new PersonQuery().GetAll();
    var firstPerson = result[0];
    var secondPerson = result[1];

    // Assert
    Assert.AreEqual(p1.Id, firstPerson.Id);
    Assert.AreEqual(p2.Id, secondPerson.Id);
}
```

## **buenas practicas para las pruebas unitarias**

1. Una prueba unitaria no debe probar más que solo una cosa.
2. Ejecución rápida.
3. Resultado consistente (confiables).
4. Pruebas unitarias aisladas. la prueba unitaria no puede depender de otras pruebas.
5. Si falla debe ser fácilmente reconocible el fallo.
6. Repetible. Poder ejecutar las veces que fuese necesario y no requerir alguna configuración adicional.
7. Ejecutable por cualquier persona.
8. Veracidad de la prueba. El test debería asegurar que podría fallar si se cambia la lógica del objeto de prueba

## Patrones de diseño para unit test

1. **ObjectMother:** Es una clase que nos proporciona instancias “**predefinidas**” de las clases que usamos con cierta frecuencia, una especie de factoría de la combinación de valores que conforman un sujeto típico. Esta técnica es especialmente útil cuando existen combinaciones de datos concretas que se usan en muchos tests.
2. **Builder:** Un Builder nos permite construir un objeto y ponerlo en el estado que necesitemos, obviando los detalles que no son necesarios para el consumidor mediante la asignación de unos valores por defecto cualquiera. Es decir, si tengo una clase Dog y estoy testando cálculos basados en su edad, es irrelevante cual sea el nombre del perro o su peso, con lo cual puedo utilizar valores por defecto para los datos que no me importan, y ni siquiera necesito saber qué valore son esos.
3. **Creation method:** método encargado de crear lo necesarios para la prueba unitaria.

## 1. ObjectMother

```
public class MoneyShould {  
    @Test  
    public void return_display_name_as_amount_concat_with_currency_symbol() {  
        Currency EUR = Currency.getInstance("EUR");  
  
        Money money = new Money(new BigDecimal("100.00"), EUR);  
  
        assertThat(money.getDisplayName(), is("100.00 €"));  
    }  
}
```

```
public class Currencies{  
    public static Currency EURO() {  
        return Currency.getInstance("EUR");  
    }  
  
    public static Currency DOLLAR() {  
        return Currency.getInstance("USD");  
    }  
}  
  
public class MoneyShould {  
    @Test  
    public void return_display_name_as_amount_concat_with_currency_symbol() {  
        Money money = new Money(new BigDecimal("100.00"), Currencies.EURO());  
  
        assertThat(money.getDisplayName(), is("100.00 €"));  
    }  
}
```

## 2. Builder

```
public class MoneyBuilder {  
  
    private String amount;  
    private Currency currency;  
  
    public MoneyBuilder withAmount(String amount){  
        this.amount = amount;  
        return this;  
    }  
  
    public MoneyBuilder withCurrency(Currency currency){  
        this.currency = currency;  
        return this;  
    }  
  
    public Money build(){  
        return new Money(new BigDecimal(amount),currency);  
    }  
}
```

```
public class MoneyShould {  
    @Test  
    public void return_display_name_as_amount_concat_with_currency_symbol() {  
  
        Money money = new MoneyBuilder()  
            .withAmount("100.00")  
            .withCurrency(Currencies.EURO())  
            .build();  
  
        assertThat(money.getDisplayName(), is("100.00 €"));  
    }  
}
```

### 3. Creation method

```
public class ProductShould {

    @Test
    public void is_on_sale_if_its_not_free_and_is_marked_as_on_sale() throws FreeProductOnSaleException {
        Product product = givenANonFreeProduct();

        product.markOnSale();

        assertThat(product.isOnSale(), is(true));
    }

    private Product givenANonFreeProduct() {
        return new ProductBuilder()
            .withAmount("100.00")
            .withCurrency(Currencies.EURO())
            .build();
    }
}
```

## Code coverage

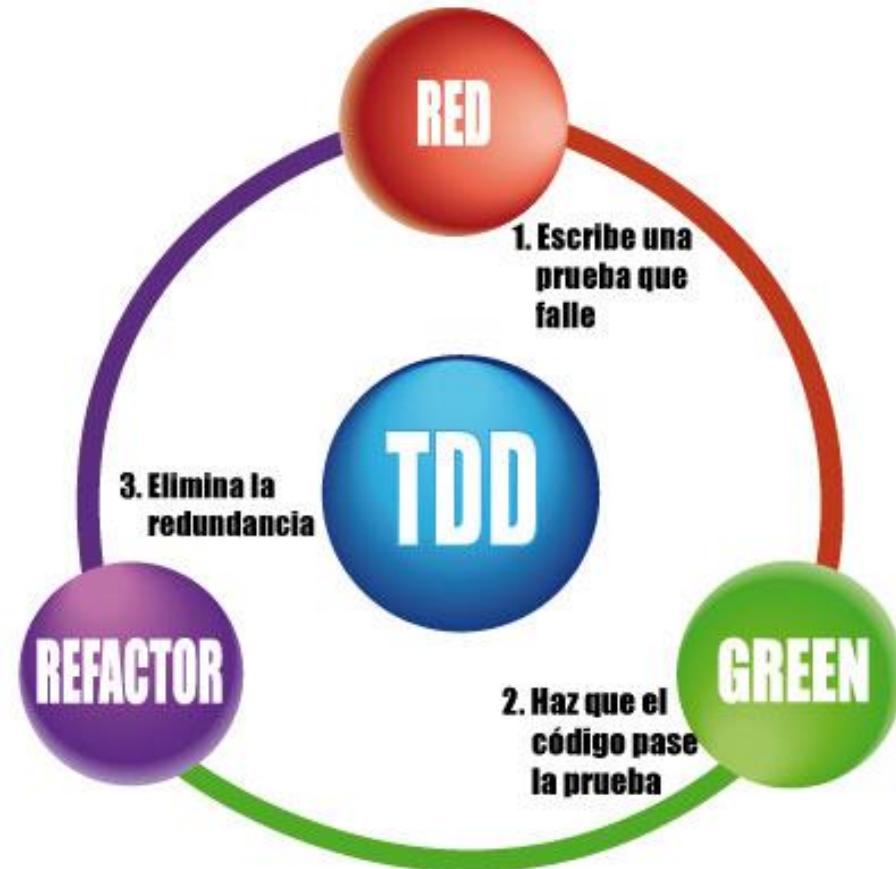
### Code Coverage:

El code coverage es una métrica utilizada en el desarrollo de software que determina el número de líneas de código que fueron ejecutadas durante nuestros test unitarios. Utilizando code coverage podemos determinar si nuestras aplicaciones se prueban de forma correcta y que porcentaje de escenarios no cuenta con pruebas automatizadas.



## ¿Qué es TDD?

- Test-driven development (TDD), Proceso iterativo en el cual el desarrollo esta guiado por test.
- metodología en la que primero se hace un test y luego el código necesario para que el test pase. No se hace nada de código si no falla algún test, por lo que primero debemos hacer el test que falle.



## Los tres pasos de TDD

En TDD deben seguirse estos tres pasos y en este orden:

1. Hacer un test automático de prueba, ejecutarlo y ver que falla.
2. Hacer el código mínimo imprescindible para que el test que acabamos de escribir pase.
- Arreglar el código, sobre todo, para evitar cosas duplicadas en el mismo. Comprobar que los test sigue pasando correctamente después de haber arreglado el código.

Estos tres pasos deben repetirse una y otra vez hasta que la aplicación esté terminada.

## 1. Hacer el test automático.

Lo primero, hacer un test. Ejemplo:

```
public void testSuma() {  
    assertEquals(5, Matematicas.suma(2,3));  
}
```

Lo primero que deberíamos ver es que el test ni siquiera compila. No existe la clase Matematicas y, por supuesto, no tiene el método suma(). Y esta es la primera ventaja de TDD. Nos ha obligado a pensar exactamente qué queremos que haga nuestra aplicación desde fuera de ella. Necesitamos una clase que tenga un método suma() con dos parámetros que nos devuelva el resultado de la suma.

## 2. Hacer el código mínimo imprescindible para que el test pase.

Para que el test pase, lo primero que hay que hacer es la clase Matematicas y ponerle el método suma(), que debe devolver algo, cualquier entero, para que al menos compile.

```
public class Matematicas {  
    public static int suma (int a, int b) {  
        return 0;  
    }  
}
```

Ya compila el test y la clase, pero si ejecutamos el test, fallará, ya que el método devuelve 0 y no 5. Corregimos la clase de la forma más inmediata posible para que pase el test. Y lo más inmediato es hacer que devuelva 5

```
public class Matematicas {  
    public static int suma (int a, int b) {  
        return 5;  
    }  
}
```

### 3. Rehacer el código

El tercer paso es arreglar el código. Debemos arreglar sobre todo duplicidades. A veces, estas duplicidades son evidentes (por ejemplo, código repetido), pero otras, como en este caso, no son tan evidentes.

```
public class Matematicas {  
    public static int suma (int a, int b) {  
        return a+b;  
    }  
}
```

Bueno, este caso es demasiado simple y en un caso real no haríamos tantos pasos para algo tan evidente, bastaría hacer el test y poner directamente la implementación buena.

## Ideas sobre cómo llevar TDD a la práctica

- 1. ¿Cómo de grandes deben ser cada iteración?:** La solución depende de nosotros, de nuestra experiencia y de nuestra capacidad para programar. Los test que hagamos no deben ser muy triviales, de forma que no nos eternicemos haciendo test tontos que se resuelven en cuestión de segundos. Tampoco deben ser muy complejos
- 2. No dejarse llevar mientras resolvemos un test:** Una vez hecho el test y viendo que falla, debemos hacer el código mínimo necesario para que eso funcione. Es normal, y cualquier programador con experiencia me dará la razón, que mientras estamos codificando nos demos cuenta de posibles fallos, mejoras o necesidades en otras partes del código relacionadas con lo que estamos haciendo y que vayamos corriendo a hacerlas. Pues bien, eso es justo lo que NO debemos hacer.
- 3. Dejar que TDD nos lleve al diseño:** sólo cuando un test nos lo requiera y la solución más simple para ese test sea dividir la clase en dos se debe realizar esto caso contrario se debe mantener con una sola clase.

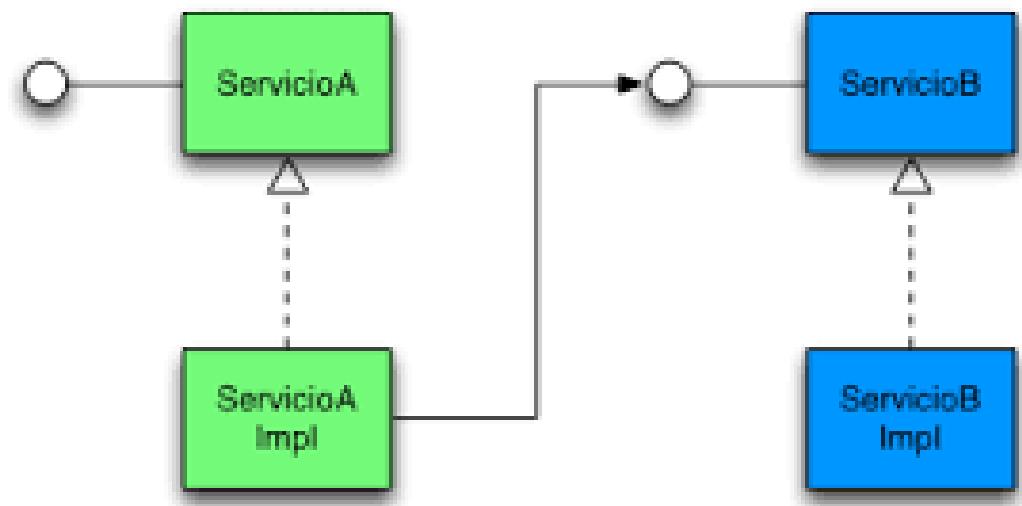
## Prácticas a evitar, o “antipatrones”

1. **Dependencias con el sistema**, ya que obstaculizará la escalabilidad del proyecto.
2. **Dependencias entre los casos de prueba**, que reducirían la modularidad y la efectividad de dichas pruebas al repetirse cuando el proyecto evolucione
3. **Comprobar detalles de implementación**, pues las pruebas deberían centrarse en funcionalidades requeridas.
4. Pruebas que tarden mucho en finalizar.
5. Comprobaciones demasiado minuciosas de detalles poco relevantes..

## JUnit y Mockito

Cuando hablamos de **unit tests** debemos hablar de **“Mocking”**, este es uno de los skills principales que debemos tener a la hora de hacer tests en nuestras aplicaciones.

Al momento de desarrollar aplicaciones las dividimos en capas, web, negocio y datos, entonces al escribir tests unitarios en una capa, no queremos preocuparnos por otra, así que la simulamos, a este proceso se le conoce como Mocking.



## Mockito - ejemplo

```
/**  
 * @author raidentrance  
 *  
 */  
public interface DataService {  
    int[] getListOfNumbers();  
}
```

```
/**  
 * @author raidentrance  
 *  
 */  
public interface CalculatorService {  
    double calculateAverage();  
}
```

```
import com.raidentrance.services.CalculatorService;  
import com.raidentrance.services.DataService;  
  
/**  
 * @author raidentrance  
 *  
 */  
public class CalculatorServiceImpl implements CalculatorService {  
    private DataService dataService;  
  
    @Override  
    public double calculateAverage() {  
        int[] numbers = dataService.getListOfNumbers();  
        double avg = 0;  
        for (int i : numbers) {  
            avg += i;  
        }  
        return (numbers.length > 0) ? avg / numbers.length : 0;  
    }  
  
    public void setDataService(DataService dataService) {  
        this.dataService = dataService;  
    }  
}
```

## Mockito - ejemplo

```
import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.when;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.MockitoJUnitRunner;

import com.raidentrance.services.DataService;
import com.raidentrance.services.impl.CalculatorServiceImpl;

/**
 * @author raidentrance
 */
@RunWith(MockitoJUnitRunner.class)
public class CalculatorServiceTest {
    @InjectMocks
    private CalculatorServiceImpl calculatorService;

    @Mock
    private DataService dataService;

    @Test
    public void testCalculateAvg_simpleInput() {
        when(dataService.getListOfNumbers()).thenReturn(new int[] { 1, 2, 3, 4, 5 });
        assertEquals(3.0, calculatorService.calculateAverage(), .01);
    }

    @Test
    public void testCalculateAvg_emptyInput() {
        when(dataService.getListOfNumbers()).thenReturn(new int[] {});
        assertEquals(0.0, calculatorService.calculateAverage(), .01);
    }

    @Test
    public void testCalculateAvg_singleInput() {
        when(dataService.getListOfNumbers()).thenReturn(new int[] { 1 });
        assertEquals(1.0, calculatorService.calculateAverage(), .01);
    }
}
```

# Patrones de Diseño

---



Se define los patrones como una solución ya probada a un problema en específico.

*“Un design pattern o patrón de diseño consiste en un diagrama de objetos que forma una solución a un problema conocido y frecuente”*

Laurent Debrauwer

## Categoría de Patrones

Según la escala o nivel de abstracción:

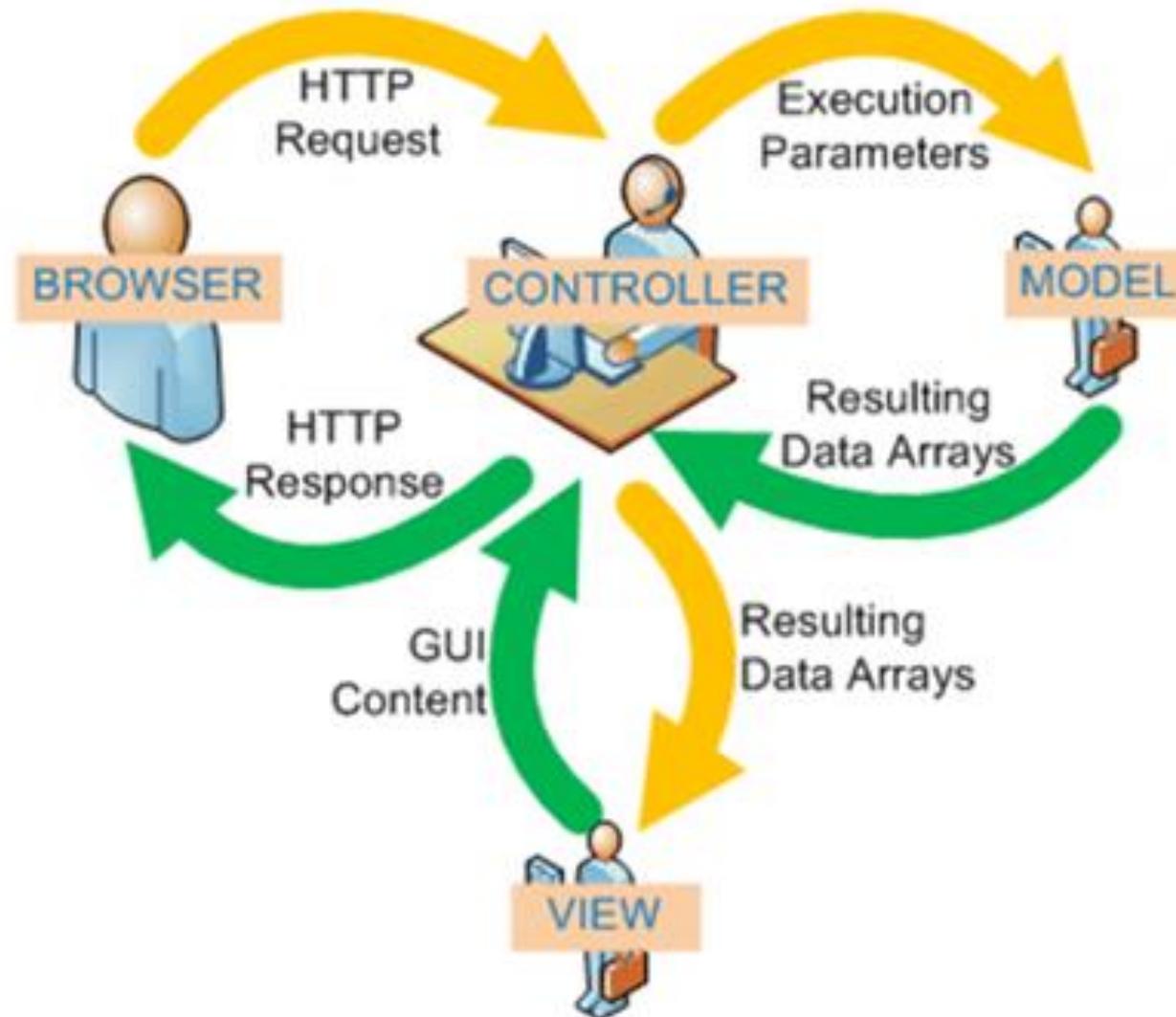
- **Patrones de arquitectura:** Aquellos que expresan un esquema organizativo estructural fundamental para sistemas de software.
- **Patrones de diseño:** Aquellos que expresan esquemas para definir estructuras de diseño (o sus relaciones) con las que construir sistemas de software.
- **Dialectos:** Patrones de bajo nivel específicos para un lenguaje de programación o entorno concreto.

## Patrones de arquitectura:

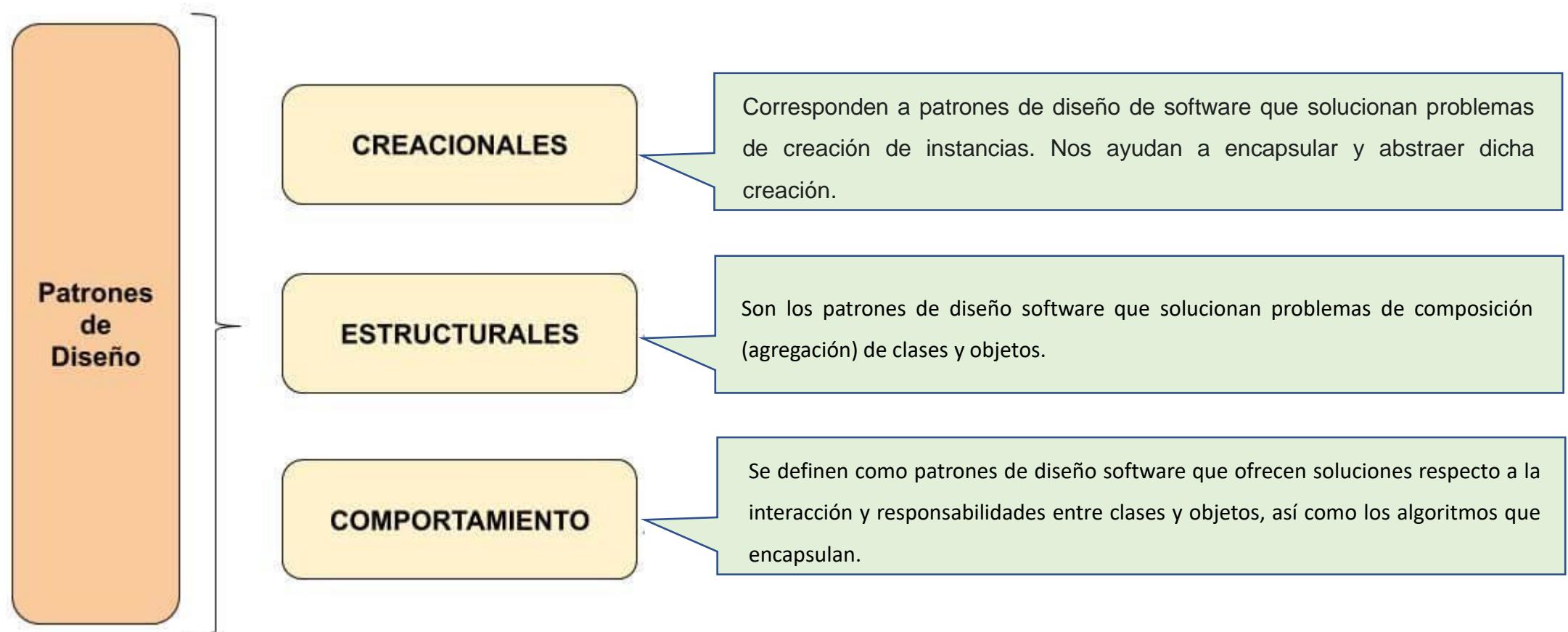
### Patrones de Arquitectura

- Programación por capas.
- Tres niveles.
- Arquitectura de microservicios.
- Arquitectura de microkernel.
- Pipeline.
- Invocación implícita.
- Arquitectura orientada a servicios
- **Modelo vista controlador**
- Peer to peer.

**Modelo Vista Controlador:**



### Tipos de Patrones de Diseño:



## Patrones de Diseño

### Patrones de Diseño

#### Creacionales

- Abstract Factory
- Factory Method
- Builder
- Singleton
- Prototype

#### Estructurales

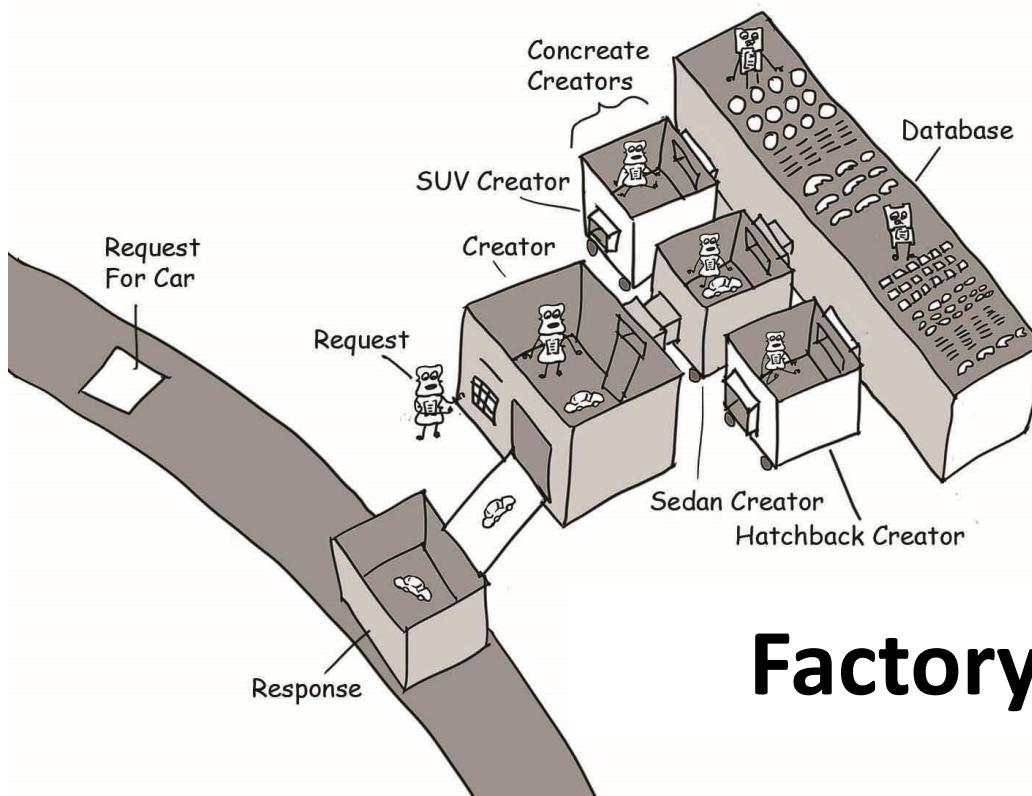
- Adapter o Wrapper
- Bridge
- Composite
- Decorator
- Facade
- Proxy

#### Comportamiento

- Strategy
- Observer
- Command
- Chain of Responsibility
- State
- Iterator
- Visitor

# Patrones

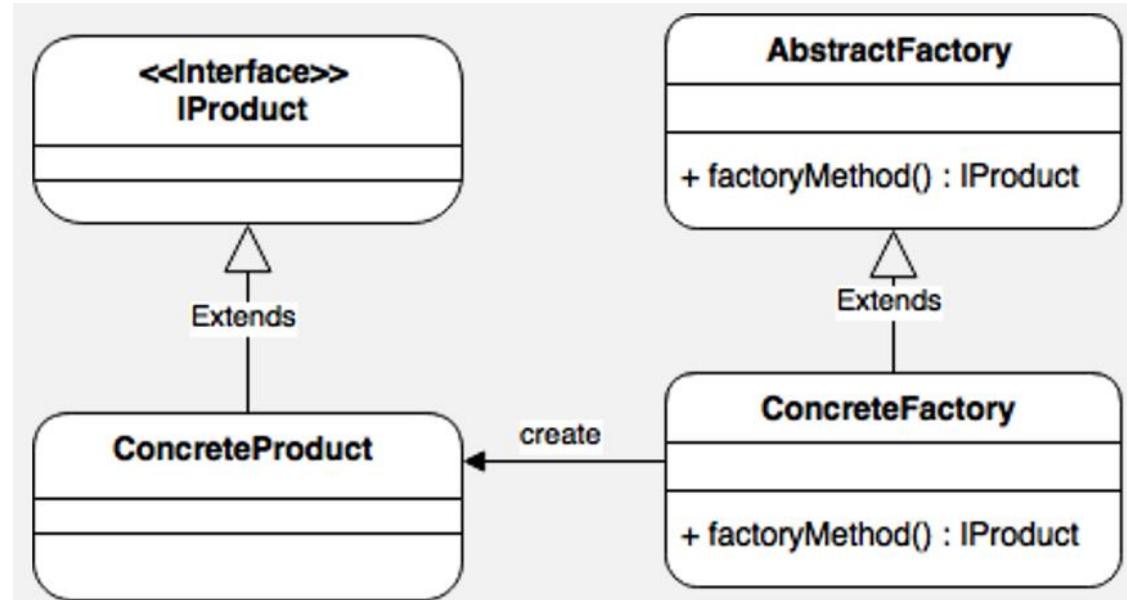
---



## Factory Method

## Patrón de creación – Factory method

- Es un patrón de diseño que define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar.
- Permite que una clase delegue en sus subclases la creación de objetos.
- Permite escribir aplicaciones que son más flexibles respecto de los tipos a utilizar.
- Permite también encapsular el conocimiento referente a la creación de objetos.

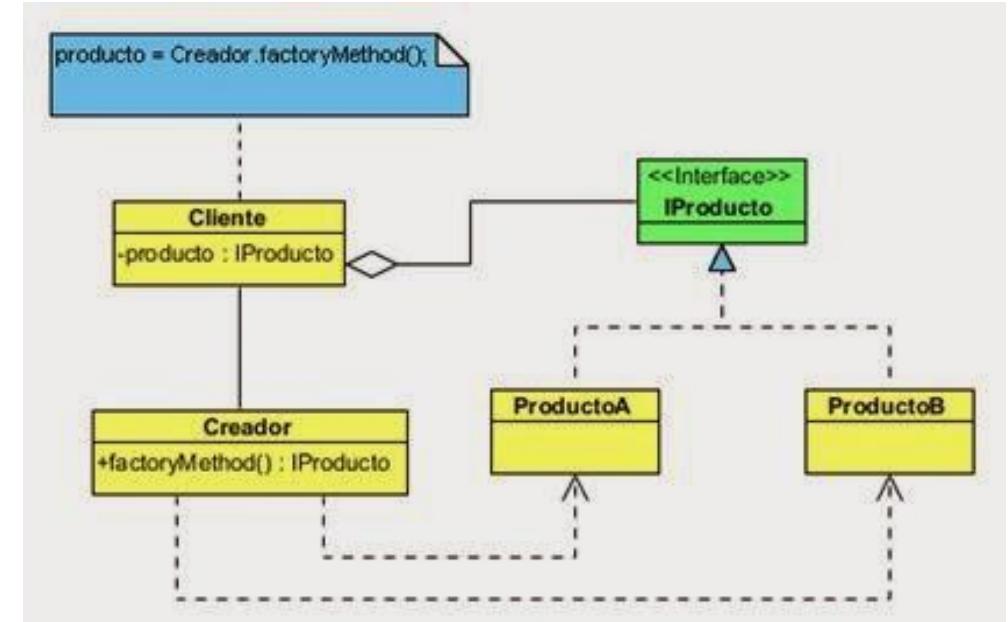
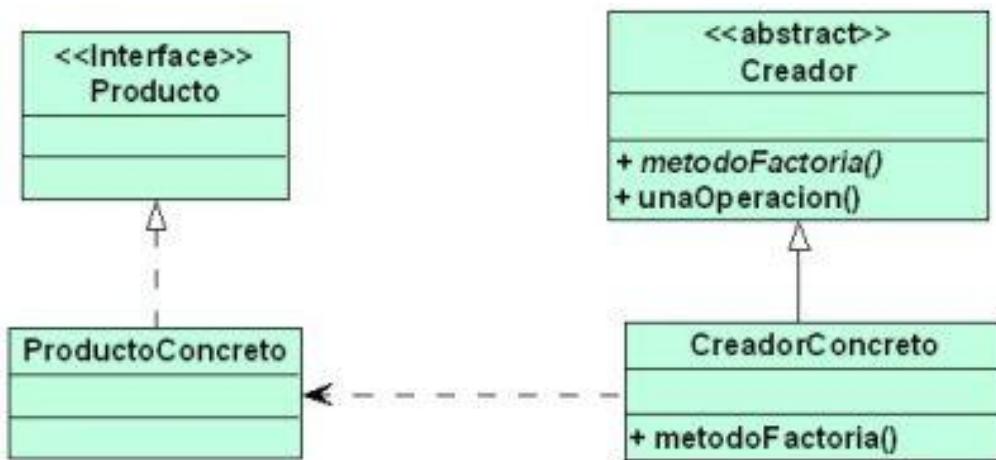


## Patrón de creación – Factory method

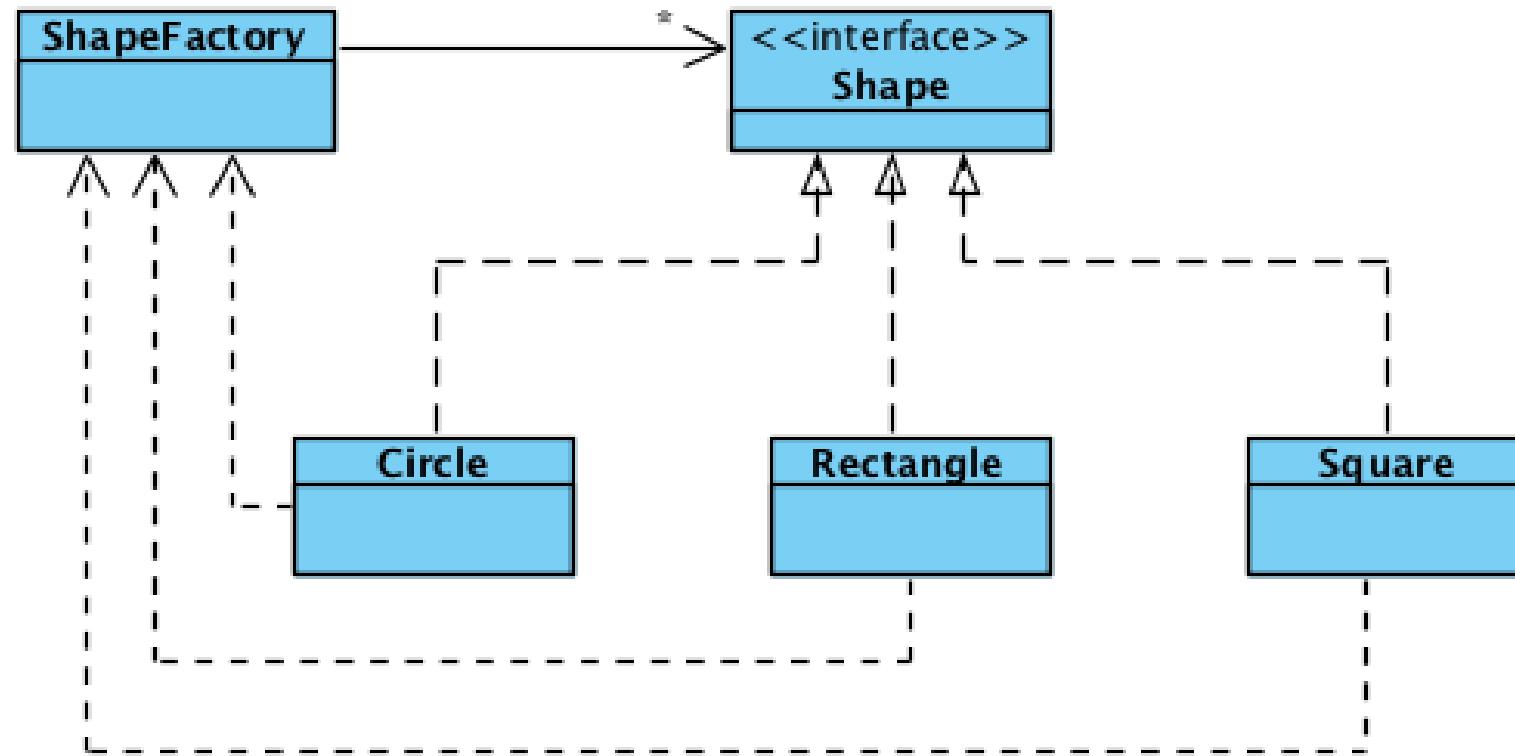
### Participantes:

- **Producto:** Define la interfaz de los objetos que crea el método de fabricación.
- **ProductoConcreto:** Implementa la interfaz Producto.
- **Creador:** Declara el método de fabricación, el cual devuelve un objeto del tipo Producto. También puede definir una implementación predeterminada del método de fabricación que devuelve un objeto ProductoConcreto. Puede llamar al método de fabricación para crear un objeto Producto.
- **CreadorConcreto:** Redefine el método de fabricación para devolver una instancia de ProductoConcreto.

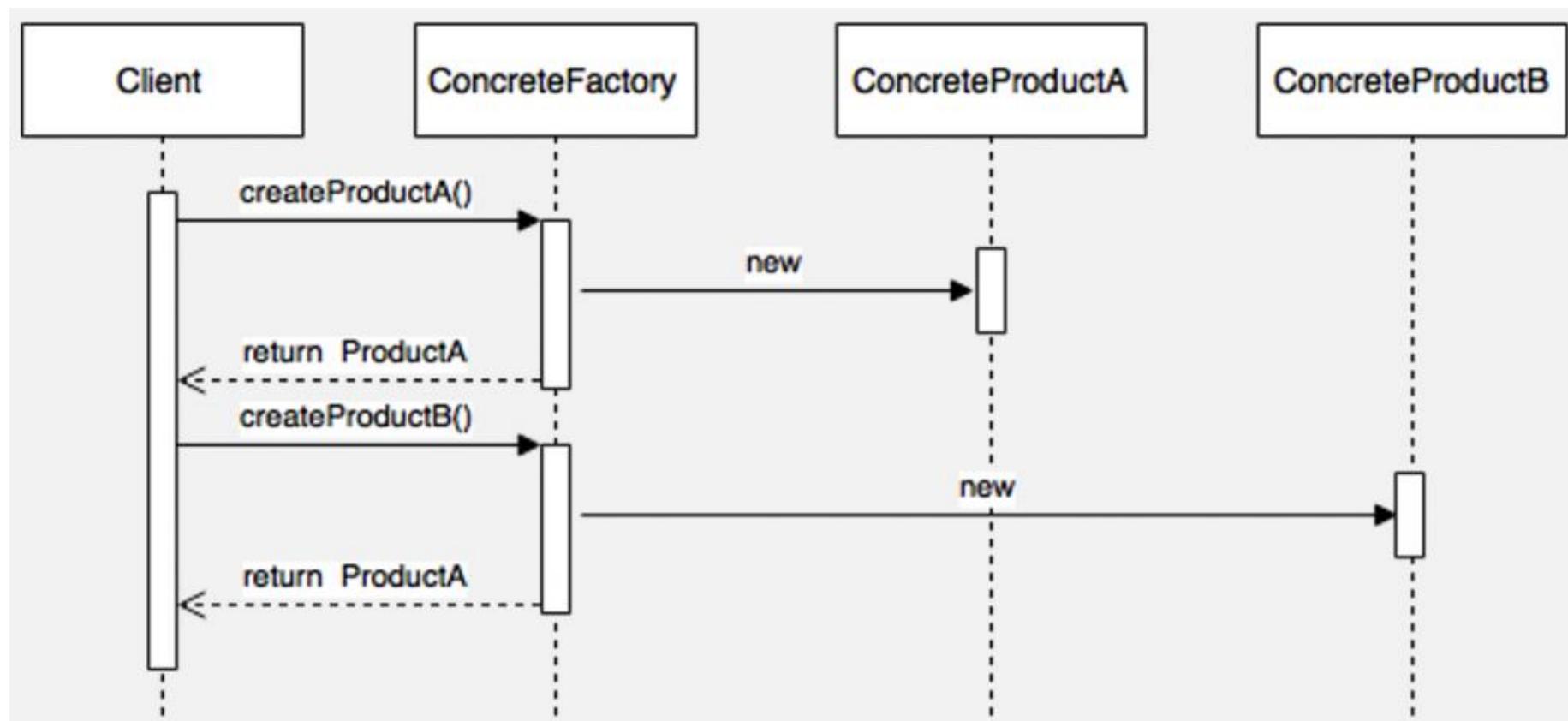
## Patrón de creación – Factory method



## Patrón de creación – Factory method



## Patrón de creación – Factory method



## Patrón de creación – Factory method

### Debe Usarse:

- Cuando una clase no puede adelantar las clases de objetos que debe crear.
- Cuando una clase pretende que sus subclases especifiquen los objetos que ella crea.
- Cuando una clase delega su responsabilidad hacia una de entre varias subclases auxiliares y queremos tener localizada a la subclase delegada.

## Patrón de creación – Factory method

### Ventajas

- **Se gana en flexibilidad**, pues crear los objetos dentro de una clase con un "Método de Fábrica" es siempre más flexible que hacerlo directamente, debido a que se elimina la necesidad de atar nuestra aplicación unas clases de productos concretos.
- **Se facilitan futuras ampliaciones**, puesto que se ofrece las subclases la posibilidad de proporcionar una versión extendida de un objeto, con sólo aplicar en los Productos la misma idea del "Método de Fábrica".

## Patrón de creación – Factory method

```
01. package FactoryMethod1;
02.
03. public class Main
04. {
05.     public static void main(String[] args)
06.     {
07.         CreadorAbstracto creator = new Creador();
08.
09.         IArchivo audio = creator.crear( Creador.AUDIO );
10.         audio.reproducir();
11.
12.         IArchivo video = creator.crear( Creador.VIDEO );
13.         video.reproducir();
14.     }
15. }
```

```
01. package FactoryMethod2;
02.
03. public interface IArchivo
04. {
05.     public void reproducir();
06. }
```

```
01. package FactoryMethod2;
02.
03. public class ArchivoAudio implements IArchivo
04. {
05.     public ArchivoAudio() {
06.     }
07.
08.     // -----
09.
10.     @Override
11.     public void reproducir() {
12.         System.out.println("Reproduciendo archivo de audio...");
13.     }
14. }
```

## Patrón de creación – Factory method

```
01. package FactoryMethod2;  
02.  
03. public class ArchivoVideo implements IArchivo  
04. {  
05.     public ArchivoVideo() {  
06.     }  
07.     // -----  
08.     // -----  
09.  
10.    @Override  
11.    public void reproducir() {  
12.        System.out.println("Reproduciendo archivo de video...");  
13.    }  
14. }
```

```
01. package FactoryMethod1;  
02.  
03. public abstract class CreadorAbstracto  
04. {  
05.     public static final int AUDIO = 1;  
06.     public static final int VIDEO = 2;  
07.     // -----  
08.     // -----  
09.  
10.    public abstract IArchivo crear(int tipo);  
11. }
```

## Patrón de creación – Factory method

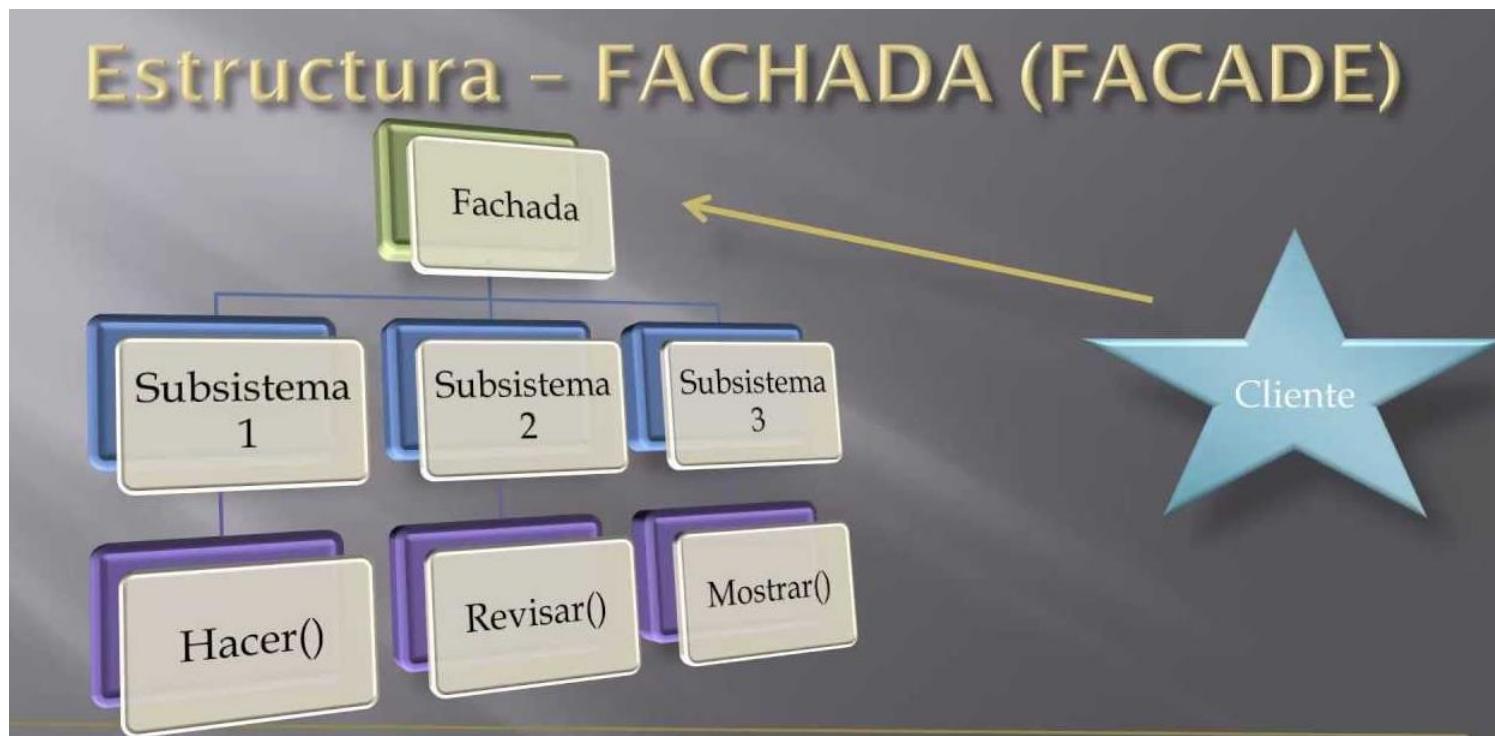
```
01. package FactoryMethod1;
02.
03. public class Creador extends CreadorAbstracto
04. {
05.     public void Creador() {
06.     }
07.
08.     // -----
09.
10.    @Override
11.    public IArchivo crear(int tipo)
12.    {
13.        IArchivo objeto;
14.
15.        switch( tipo )
16.        {
17.            case AUDIO:
18.                objeto = new ArchivoAudio();
19.                break;
20.            case VIDEO:
21.                objeto = new ArchivoVideo();
22.                break;
23.            default:
24.                objeto = null;
25.        }
26.
27.        return objeto;
28.    }
29. }
```

## Patrón de creación – Factory method

```
01. package FactoryMethod2;
02.
03. public class Main
04. {
05.     public static void main(String[] args)
06.     {
07.         IArchivo video = Creador.getArchivo( Creador.VIDEO );
08.         video.reproducir();
09.
10.        IArchivo audio = Creador.getArchivo( Creador.AUDIO );
11.        audio.reproducir();
12.    }
13. }
```

```
01. package FactoryMethod2;
02.
03. public class Creador
04. {
05.     public static final int AUDIO = 1;
06.     public static final int VIDEO = 2;
07.
08.     // -----
09.
10.     public Creador() {
11.     }
12.
13.     // -----
14.
15.     public static IArchivo getArchivo(int tipo)
16.     {
17.         IArchivo objeto;
18.
19.         switch( tipo )
20.         {
21.             case AUDIO:
22.                 objeto = new ArchivoAudio();
23.                 break;
24.             case VIDEO:
25.                 objeto = new ArchivoVideo();
26.                 break;
27.             default:
28.                 objeto = null;
29.         }
30.
31.         return objeto;
32.     }
33. }
```

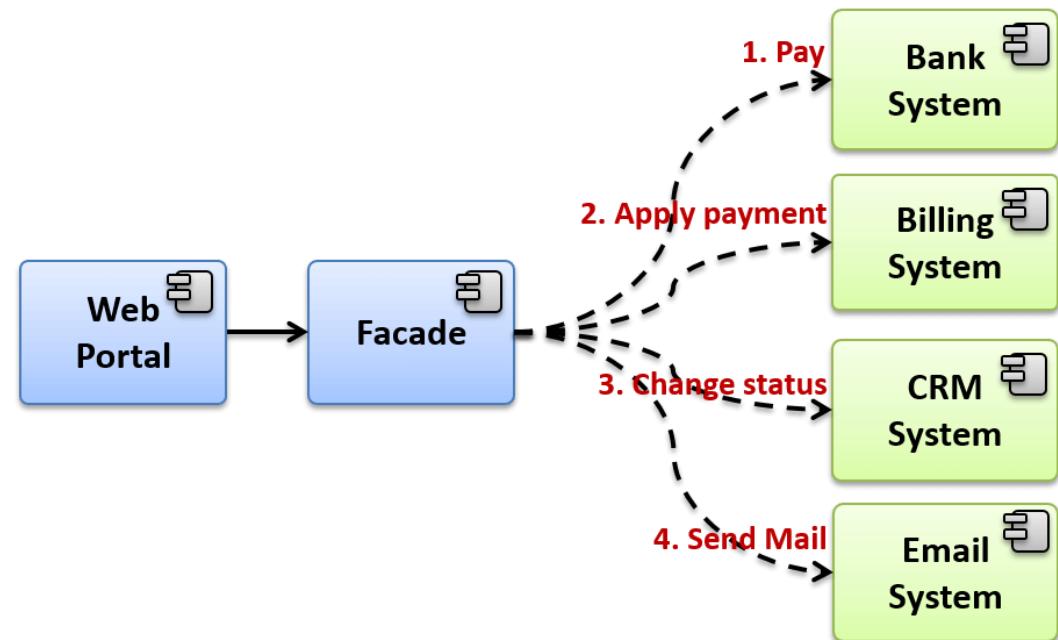
# Patrones



## Patrón Estructural – Fachada

El patrón de diseño Facade simplifica la complejidad de un sistema mediante una interfaz más sencilla. Mejora el acceso a nuestro sistema logrando que otros sistemas o subsistemas usen un punto de acceso en común que reduce la complejidad, minimizando las interacciones y dependencias.

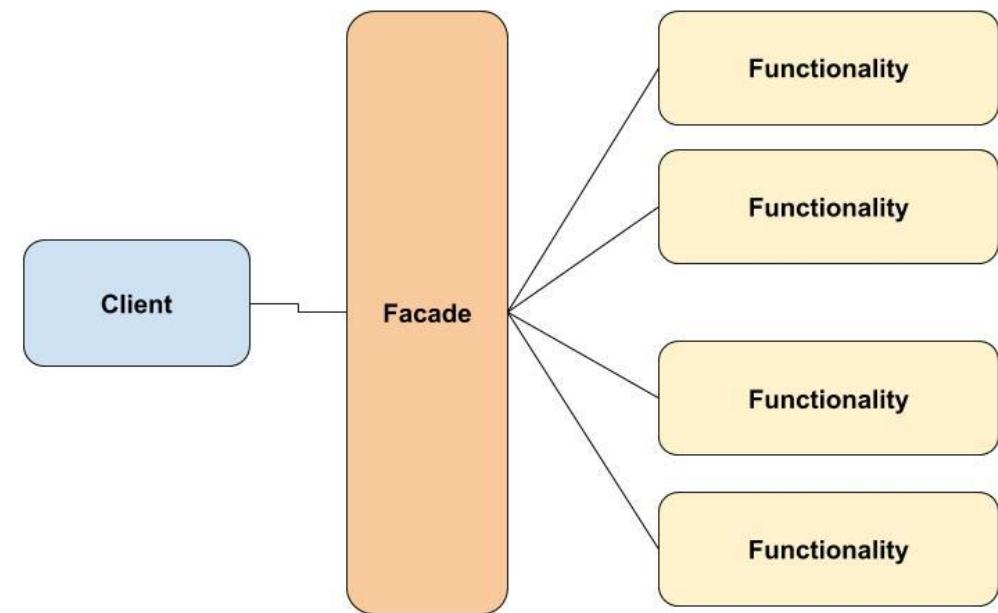
Se trata de un patrón de diseño bastante útil en vista de que también desacopla los sistemas .



## Patrón Estructural – Fachada

Tenemos en este patrón tres partes:

- **El Cliente** que accede al facade.
- **El Facade** que accede al resto de funcionalidades y las unifica o simplifica.
- **El resto de funcionalidades / subsistemas,** que están “atrás” del facade.

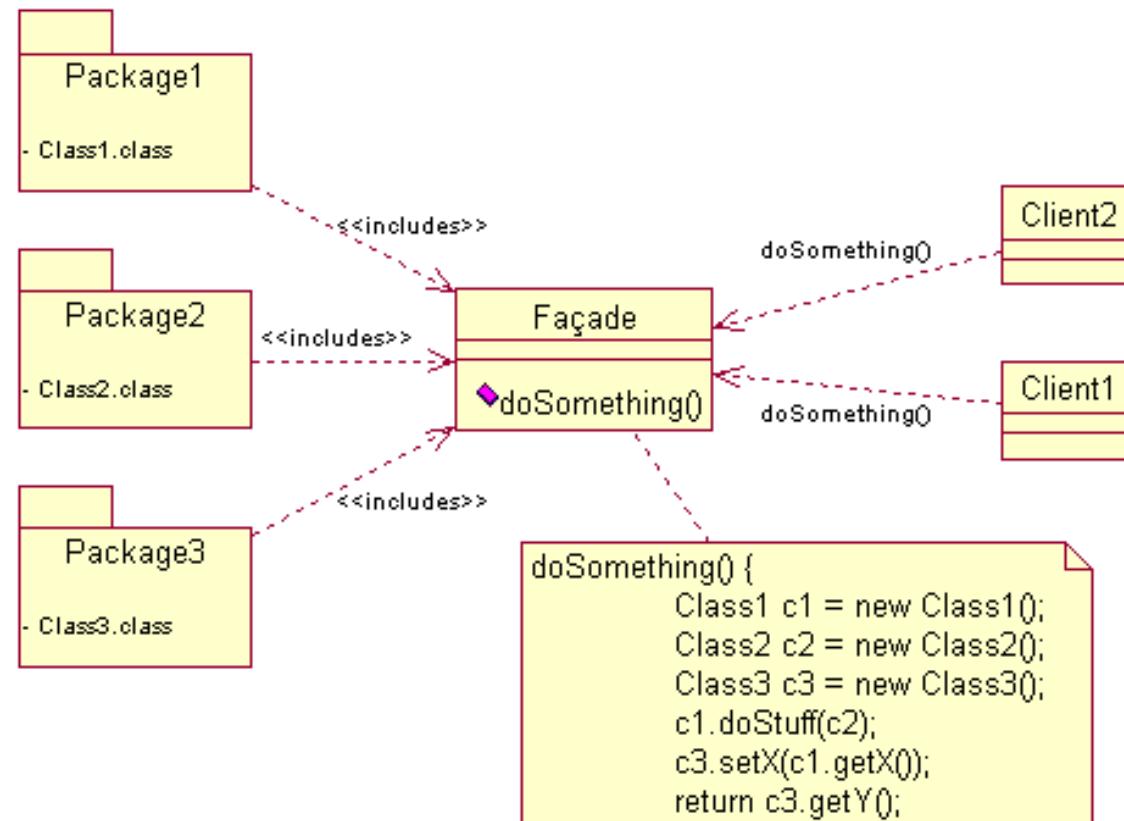


## Patrón Estructural – Fachada

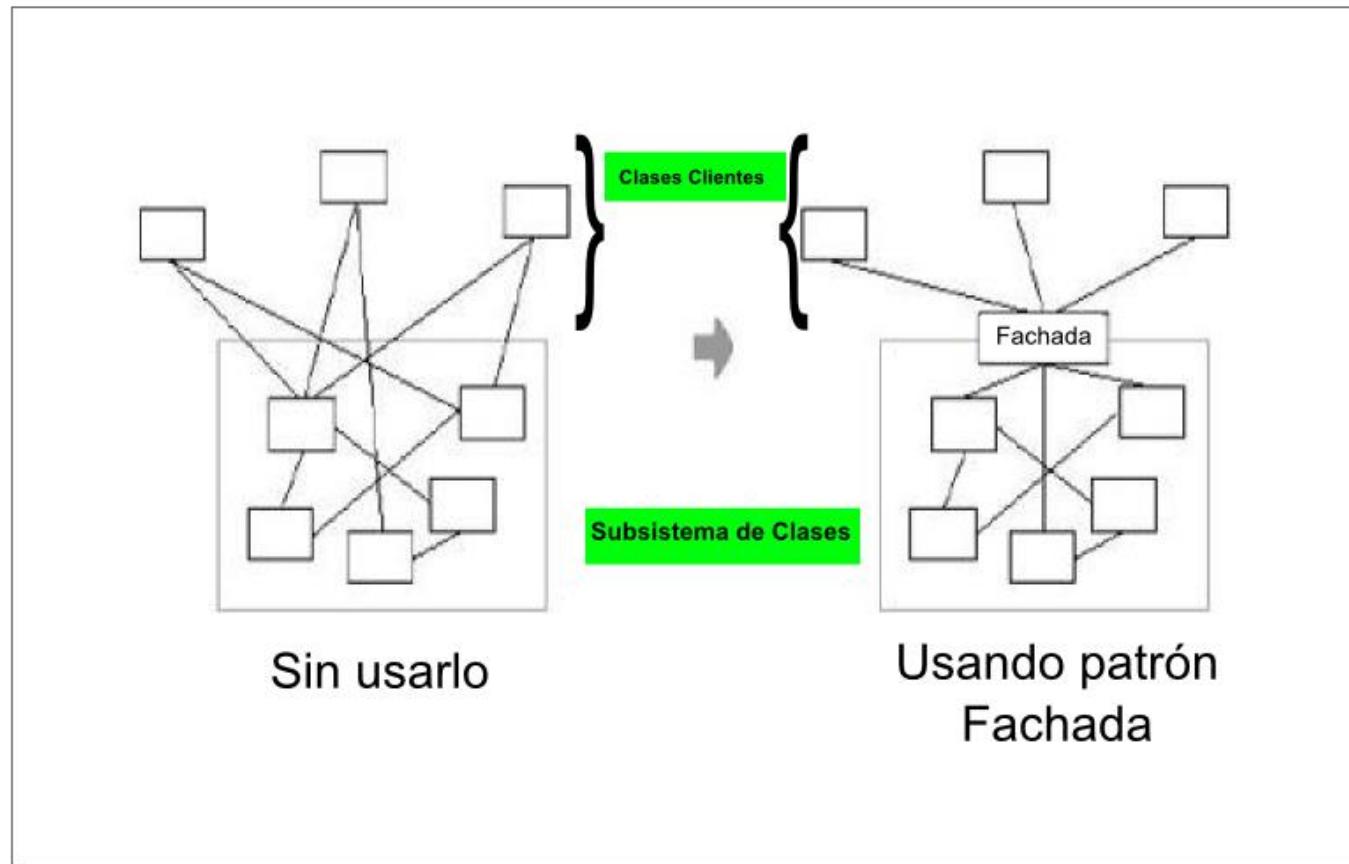
### Ventajas e inconvenientes

- La principal ventaja del patrón fachada consiste en que para modificar las clases de los subsistemas, sólo hay que realizar cambios en la interfaz/fachada, y los clientes pueden permanecer ajenos a ello. Además, y como se mencionó anteriormente, los clientes no necesitan conocer las clases que hay tras dicha interfaz.
- Como inconveniente, si se considera el caso de que varios clientes necesiten acceder a subconjuntos diferentes de la funcionalidad que provee el sistema, podrían acabar usando sólo una pequeña parte de la fachada, por lo que sería conveniente utilizar varias fachadas más específicas en lugar de una única global.

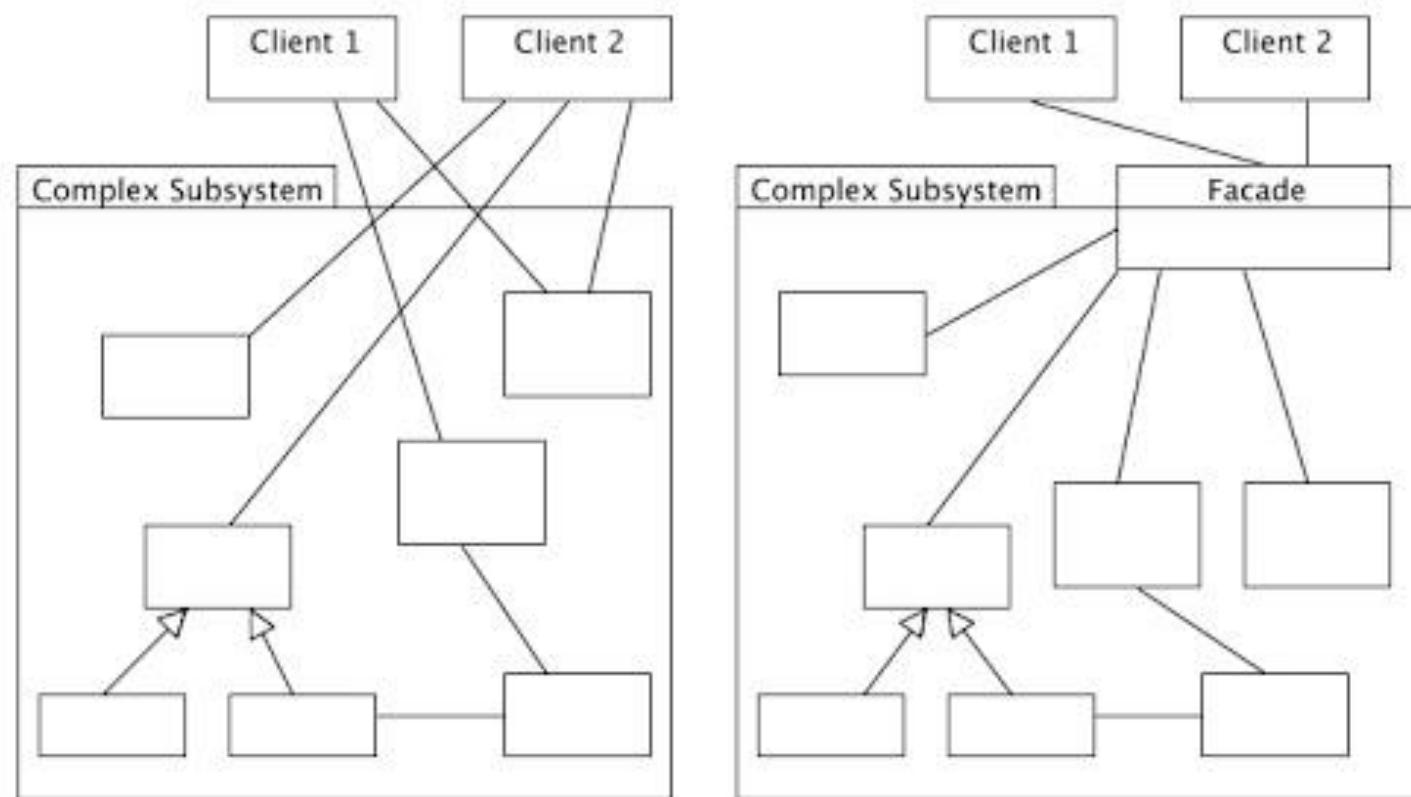
## Patrón Estructural – Fachada



## Patrón Estructural – Fachada



## Patrón Estructural – Fachada



## Patrón Estructural – Fachada

```
package com.genbetadev;
public class Impresora {
    private String tipoDocumento;
    private String hoja;
    private boolean color;
    private String texto;
    public String getTipoDocumento() {
        return tipoDocumento;
    }
    public void setTipoDocumento(String tipoDocumento) {
        this.tipoDocumento = tipoDocumento;
    }
    public void setHoja(String hoja) {
        this.hoja = hoja;
    }
    public String getHoja() {
        return hoja;
    }
}
```

```
public void setColor(boolean color) {
    this.color = color;
}
public boolean getColor() {
    return color;
}
public void setTexto(String texto) {
    this.texto = texto;
}
public String getTexto() {
    return texto;
}
public void imprimir() {
    impresora.imprimirDocumento();
}
```

## Patrón Estructural – Fachada

```
package com.genbetadev;

public class PrincipalCliente {

    public static void main(String[] args) {
        Impresora i = new Impresora();
        i.setHoja("a4");
        i.setColor(true);
        i.setTipoDocumento("pdf");
        i.setTexto("texto 1");
        i.imprimirDocumento();

        Impresora i2 = new Impresora();
        i2.setHoja("a4");
        i2.setColor(true);
        i2.setTipoDocumento("pdf");
        i2.setTexto("texto 2");
        i2.imprimirDocumento();

        Impresora i3 = new Impresora();
        i3.setHoja("a3");
        i3.setColor(false);
        i3.setTipoDocumento("excel");
        i3.setTexto("texto 3");
        i3.imprimirDocumento();

    }
}
```

## Patrón Estructural – Fachada

```
package com.genbetadev;
public class FachadaImpresoraNormal {
    Impresora impresora;
    public FachadaImpresoraNormal(String texto) {
        super();
        impresora= new Impresora();
        impresora.setColor(true);
        impresora.setHoja("A4");
        impresora.setTipoDocumento("PDF");
        impresora.setTexto(texto);
    }
    public void imprimir() {
        impresora.imprimirDocumento();
    }
}
```

```
package com.genbetadev;
public class PrincipalCliente2 {
    public static void main(String[] args) {
        FachadaImpresoraNormal fachada1= new FachadaImpresoraNormal("texto1");
        fachada1.imprimir();
        FachadaImpresoraNormal fachada2= new FachadaImpresoraNormal("texto2");
        fachada2.imprimir();
        Impresora i3 = new Impresora();
        i3.setHoja("a4");
        i3.setColor(true);
        i3.setTipoDocumento("excel");
        i3.setTexto("texto 3");
        i3.imprimirDocumento();
    }
}
```

# Patrones

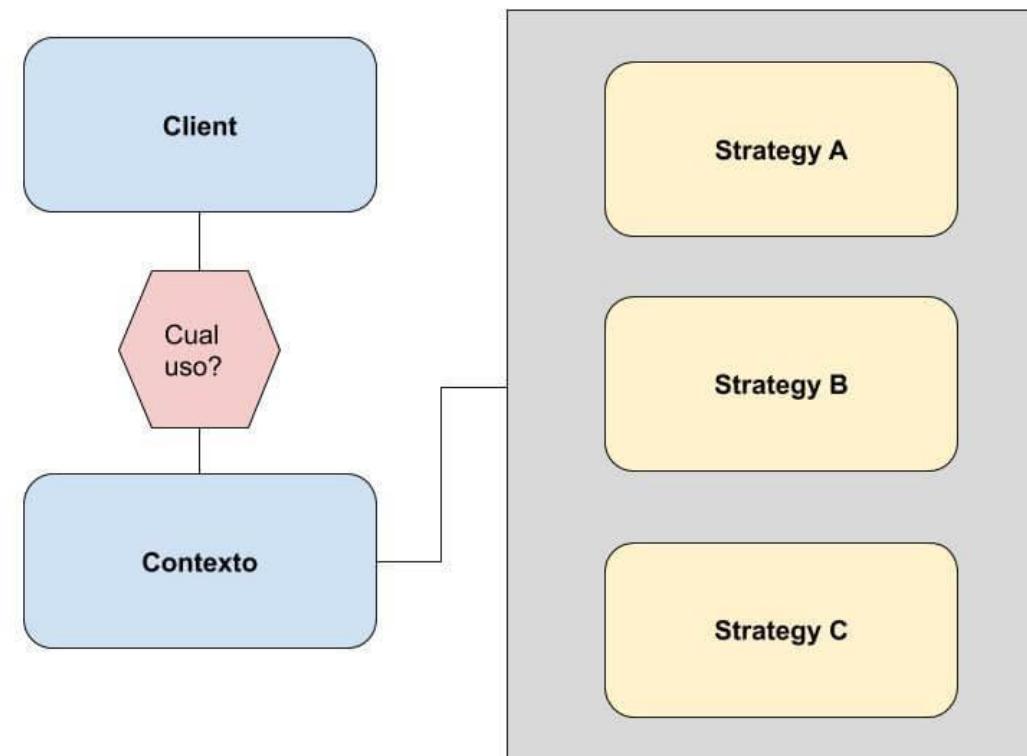
---



## Patrón de comportamiento - STRATEGY

El patrón de diseño Strategy en Java ayuda a definir diferentes comportamientos o funcionalidades que pueden ser cambiadas en tiempo de ejecución.

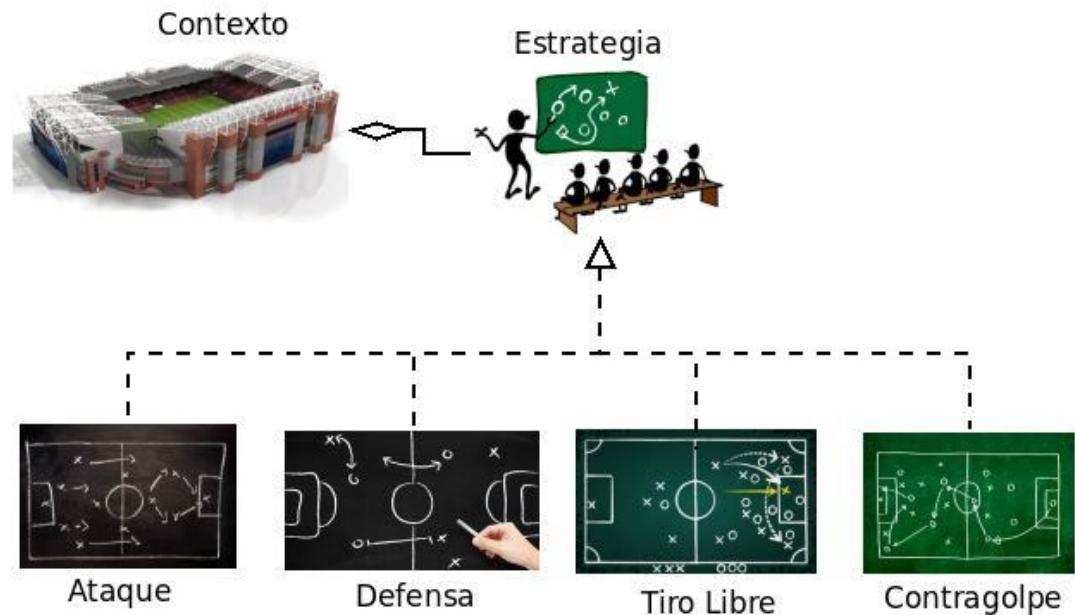
En el patrón Strategy creamos diferentes clases que representan estrategias y que podremos usar según alguna variación o input.



## Patrón de comportamiento – STRATEGY – componentes.

Los componentes del patrón Strategy son:

- **Interfaz Strategy:** es la interfaz que define cómo se conformará el contrato de la estrategia.
- **Clases Concretas Strategy:** son las clases que implementan la interfaz y donde se desarrolla la funcionalidad.
- **Contexto:** donde se establece que estrategia se usará.



## Patrón de comportamiento – STRATEGY

```
1 package patterns.strategy;
2
3 public interface CommissionStrategy {
4     double applyCommission(double amount);
5 }
```

```
1 package patterns.strategy;
2
3 public class FullCommission implements CommissionStrategy {
4     @Override
5     public double applyCommission(double amount) {
6         // do complicated formula of commissions.
7         return amount * 0.50d;
8     }
9 }
```

```
1 package patterns.strategy;
2
3 public class NormalCommission implements CommissionStrategy {
4     @Override
5     public double applyCommission(double amount) {
6         // do complicated formula of commissions.
7         return amount * 0.30;
8     }
9 }
```

```
1 package patterns.strategy;
2
3 public class RegularCommision implements CommissionStrategy {
4     @Override
5     public double applyCommission(double amount) {
6         // do complicated formula of commissions.
7         return amount * 0.10;
8     }
9 }
```

## Patrón de comportamiento – STRATEGY

```
1 package patterns.strategy;  
2  
3 public class Context {  
4  
5     private CommissionStrategy commissionStrategy;  
6  
7     public Context(CommissionStrategy commissionStrategy){  
8         this.commissionStrategy = commissionStrategy;  
9     }  
10  
11    public double executeStrategy(double amount){  
12        return commissionStrategy.applyCommission(amount);  
13    }  
14}
```

```
1 package patterns.strategy;  
2  
3 public interface CommissionStrategy {  
4     double applyCommission(double amount);  
5 }
```

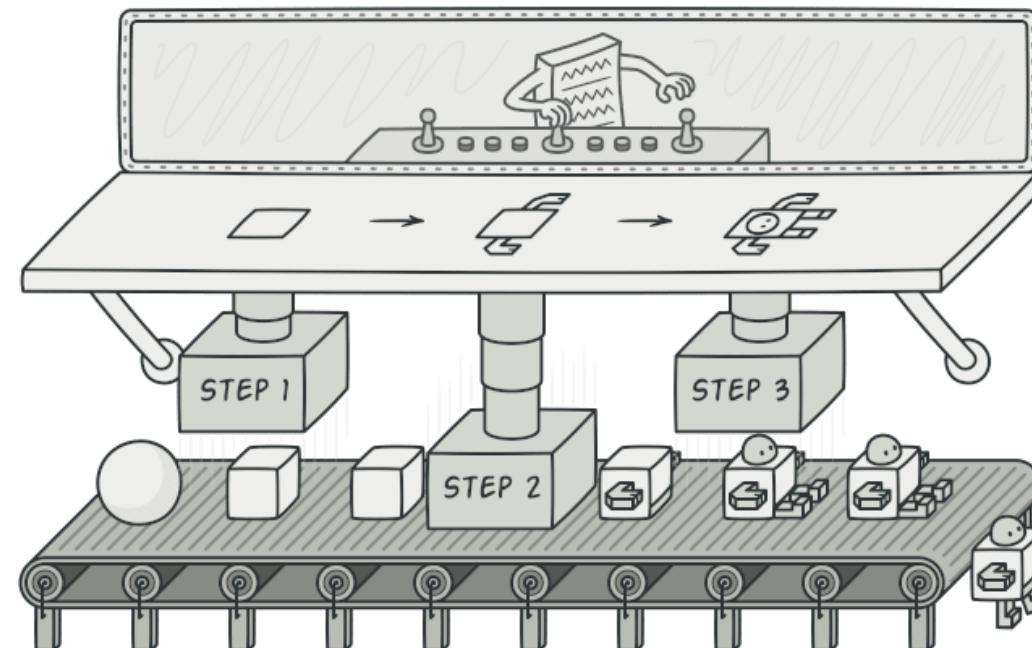
## Patrón de comportamiento – STRATEGY

```
3  public class StrategyPatternExample {  
4  
5      public static void main(String[] args) {  
6  
6          CommissionStrategy commissionStrategy = getStrategy(1000d);  
7          Context context = new Context(commissionStrategy);  
8          System.out.println("Commission for 1000d = " + context.executeStrategy(1000d));  
9  
10         commissionStrategy = getStrategy(500d);  
11         context = new Context(commissionStrategy);  
12         System.out.println("Commission for 500d = " + context.executeStrategy(500d));  
13  
14         commissionStrategy = getStrategy(100d);  
15         context = new Context(commissionStrategy);  
16         System.out.println("Commission for 100d = " + context.executeStrategy(100d));  
17     }  
18  
19     private static CommissionStrategy getStrategy(double amount) {  
20         CommissionStrategy strategy;  
21         if (amount >= 1000d) {  
22             strategy = new FullCommission();  
23         } else if (amount >= 500d && amount <= 999d) {  
24             strategy = new NormalCommission();  
25         } else {  
26             strategy = new RegularCommision();  
27         }  
28         return strategy;  
29     }  
30 }  
31 }
```

```
1  package patterns.strategy;  
2  
3  public class Context {  
4  
5      private CommissionStrategy commissionStrategy;  
6  
7      public Context(CommissionStrategy commissionStrategy){  
8          this.commissionStrategy = commissionStrategy;  
9      }  
10  
11     public double executeStrategy(double amount){  
12         return commissionStrategy.applyCommission(amount);  
13     }  
14 }
```

# Patrones

---

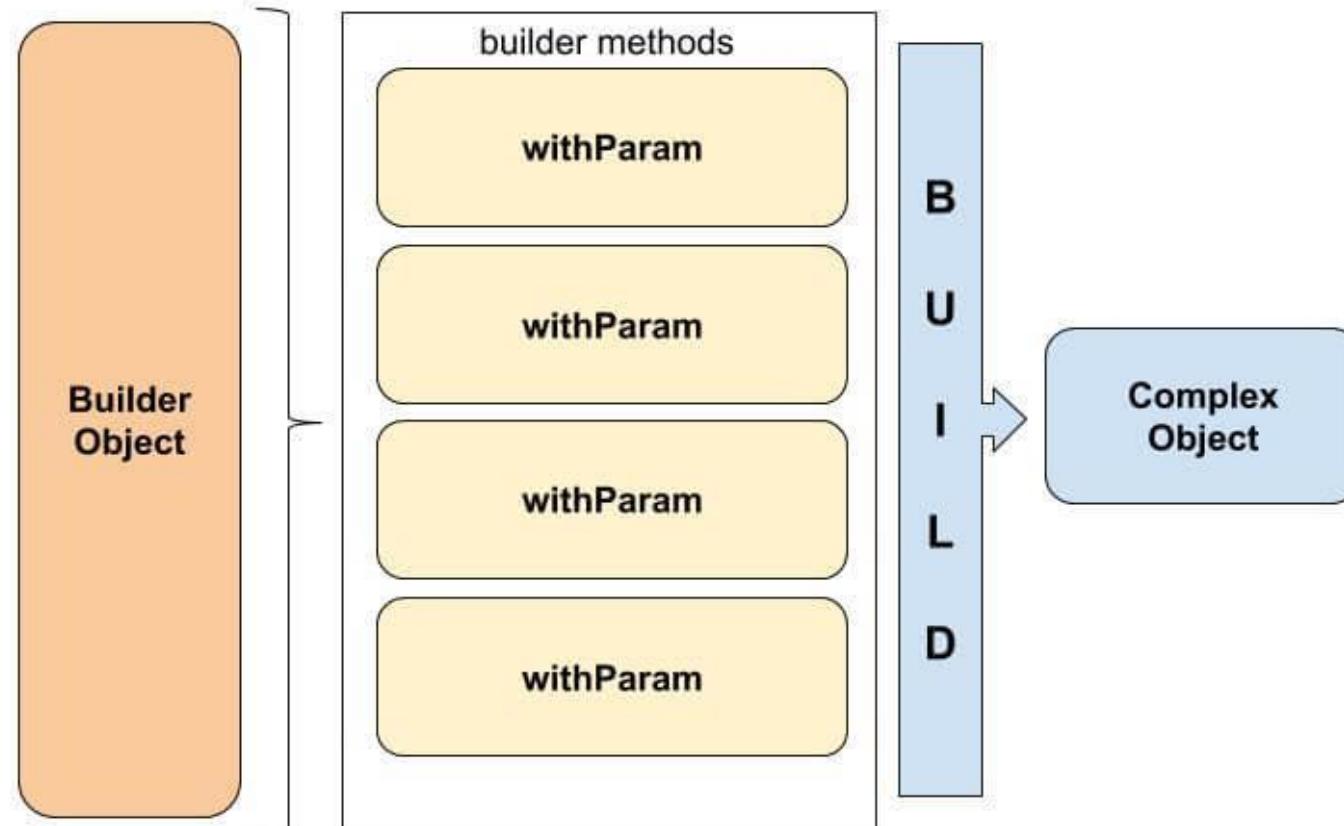


## Builder

## Patrón de creación – Builder

- El patrón de diseño Builder permite crear objetos que habitualmente son complejos, utilizando otro objeto más simple que los construye paso por paso.
- Este patrón Builder se utiliza en situaciones en las que debe construirse un objeto repetidas veces o cuando este objeto tiene gran cantidad de atributos y objetos asociados, y en donde usar constructores para crear el objeto no es una solución cómoda.
- Normalmente resuelve el problema sobre decidir qué constructor utilizar. A menudo las clases tienen muchos constructores y es muy difícil mantenerlos. Es común ver constructores múltiples con distintas combinaciones de parámetros.

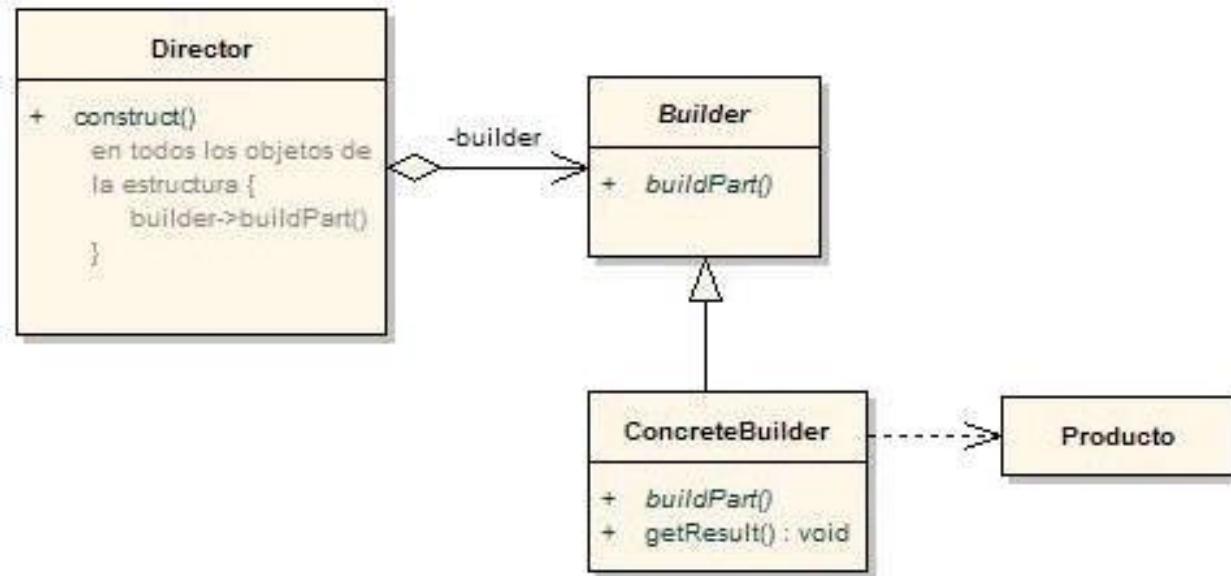
## Patrón de creación – Builder



## Patrón de creación – Builder

### Participantes:

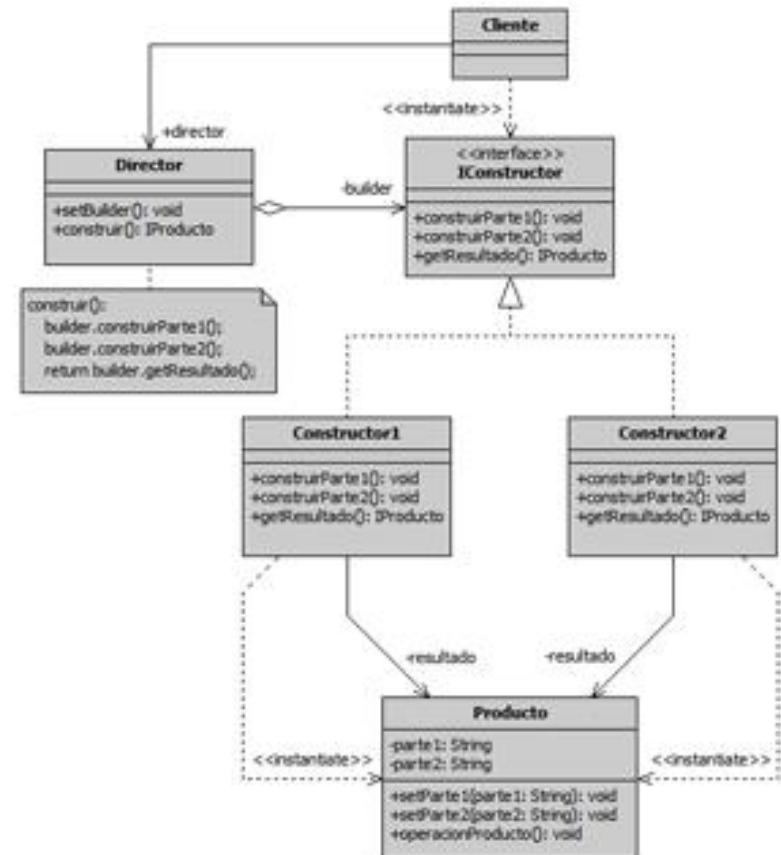
- **Builder:** clase abstracta para crear productos.
- **Concrete Builder:** implementación del Builder construye y reúne las partes necesarias para construir los productos.
- **Director:** construye un objeto usando el patrón Builder.
- **Producto:** El objeto complejo bajo construcción.



## Patrón de creación – Builder

### Ventajas:

- Reduce el acoplamiento.
- Permite variar la representación interna de estructuras complejas.
- Cada ConcreteBuilder tiene el código específico para crear y modificar una estructura interna concreta.
- Distintos Director con distintas utilidades pueden utilizar el mismo ConcreteBuilder.
- Permite un mayor control en el proceso de creación del objeto.
- El Director controla la creación paso a paso, solo cuando el Builder ha terminado de construir el objeto lo recupera el Director.



## Patrón de creación – Builder

```
3  public class BankAccount {  
4  
5      private long accountNumber;  
6      private String owner;  
7      private BankAccountType type;  
8      private double balance;  
9      private double interestRate;  
10  
11     public BankAccount() {  
12     }  
13  
14     public long getAccountNumber() {  
15         return accountNumber;  
16     }  
17  
18     public void setAccountNumber(long accountNumber) {  
19         this.accountNumber = accountNumber;  
20     }  
21  
22     public String getOwner() {  
23         return owner;  
24     }  
25  
26     public void setOwner(String owner) {  
27         this.owner = owner;  
28     }  
29  
30     public BankAccountType getType() {  
31         return type;  
32     }  
33  
34     public void setType(BankAccountType type) {  
35         this.type = type;  
36     }  
37  
38     public double getBalance() {  
39         return balance;  
40     }  
41  
42     public void setBalance(double balance) {  
43         this.balance = balance;  
44     }  
45  
46     public double getInterestRate() {  
47         return interestRate;  
48     }  
49
```

```
1  package patterns.builder;  
2  
3  public interface IBuilder {  
4      BankAccount build();  
5  }  
6  
7  public class BankAccountBuilder implements IBuilder {  
8  
9      private long accountNumber; //This is important, so we will pass it to the constructor.  
10     private String owner;  
11     private BankAccountType type;  
12     private double balance;  
13     private double interestRate;  
14  
15     public BankAccountBuilder(long accountNumber) {  
16         this.accountNumber = accountNumber;  
17     }  
18  
19     public BankAccountBuilder withOwner(String owner){  
20         this.owner = owner;  
21         return this; //By returning the builder each time, we can create a fluent interface.  
22     }  
23  
24     public BankAccountBuilder withType(BankAccountType type){  
25         this.type = type;  
26         return this;  
27     }  
28  
29     public BankAccountBuilder withBalance(double balance){  
30         this.balance = balance;  
31         return this;  
32     }  
33  
34     public BankAccountBuilder withRate(double interestRate){  
35         this.interestRate = interestRate;  
36         return this;  
37     }  
38  
39     @Override  
40     public BankAccount build(){  
41         BankAccount account = new BankAccount();  
42         account.setAccountNumber(this.accountNumber);  
43         account.setOwner(this.owner);  
44         account.setType(this.type);  
45         account.setBalance(this.balance);  
46         account.setInterestRate(this.interestRate);  
47         return account;  
48     }  
49 }
```

```
3  public class BuilderPatternExample {  
4  
5      public static void main(String[] args) {  
6  
7          BankAccountBuilder builder = new BankAccountBuilder(123451);  
8  
9          BankAccount bankAccount = builder.withBalance(1000.20)  
10             .withOwner("Oaken")  
11             .withRate(10.15)  
12             .withType(BankAccountType.PLATINUM)  
13             .build();  
14  
15          System.out.println(bankAccount);  
16      }  
17  }
```

# Patrones

---



## Patrón Adapter

## Patrón Estructural – ADAPTER

El patrón de diseño Adapter te sirve cuando tienes interfaces diferentes o incompatibles entre sí y necesitas que el cliente pueda usar ambas del mismo modo.

El patrón de diseño Adapter dice en su definición que convierte una interfaz o clase en otra interfaz que el cliente desea.

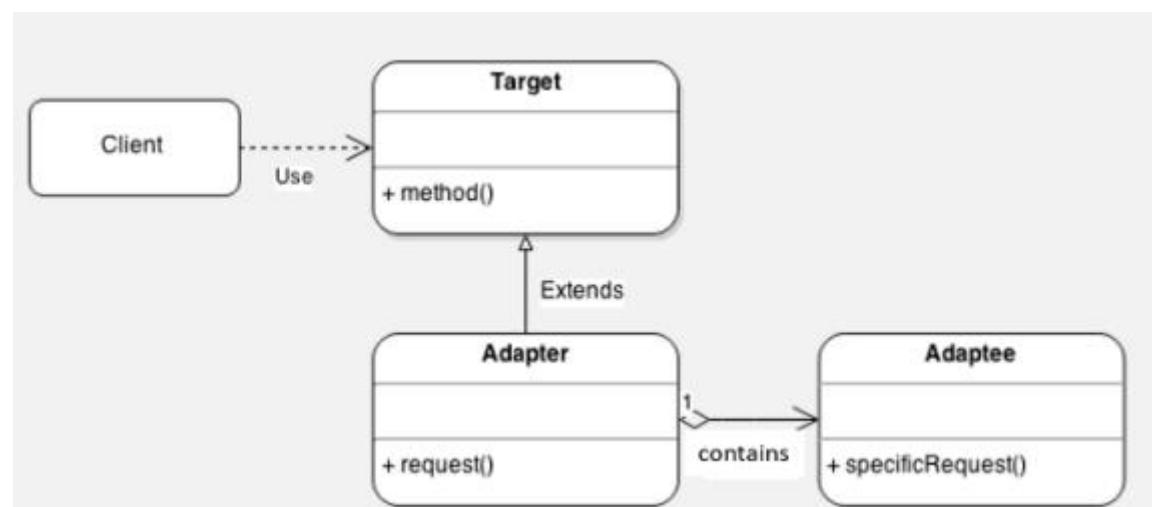


## Patrón Estructural – ADAPTER

### Partes del patrón de diseño Adapter

Las partes de patrón Adapter son:

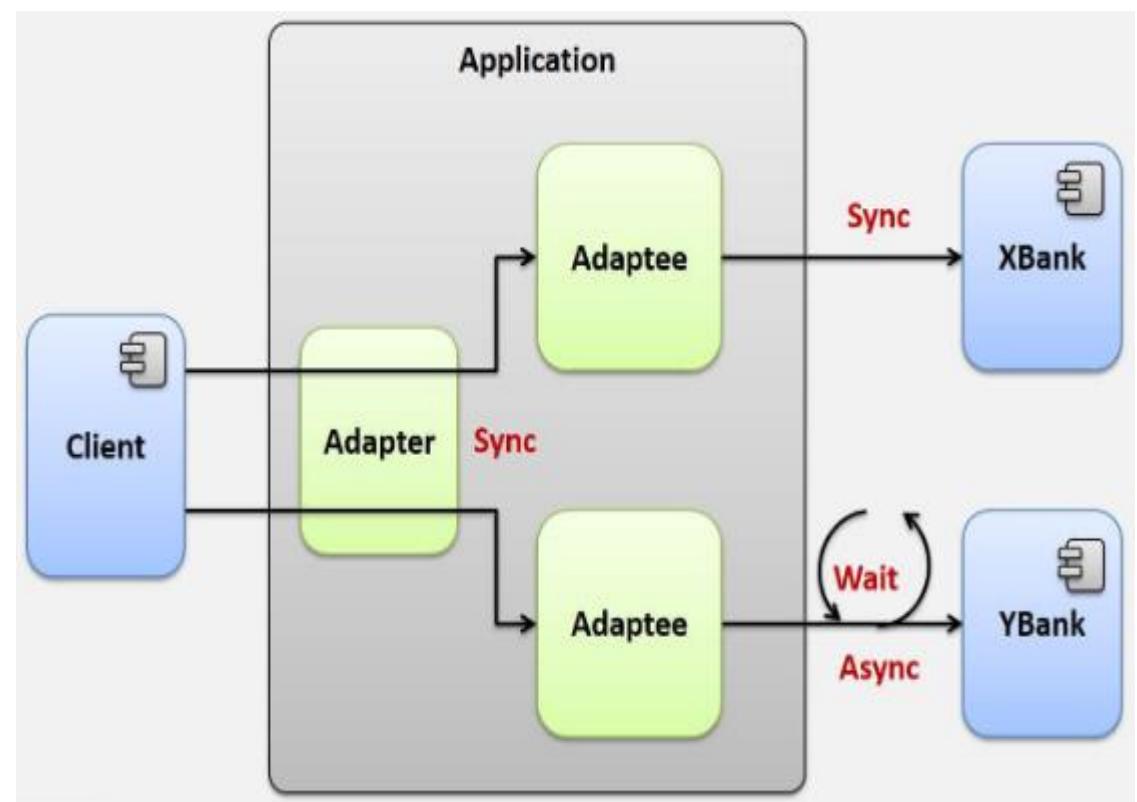
- **Target**: la interfaz que usamos para crear el adapter.
- **Adapter**: es la implementación del target y que se ocupará de realizar la adaptación.
- **Client**: es el que interactúa y usa el adapter.
- **Adaptee**: es la interfaz incompatible que necesitamos adaptar con el adapter.



## Patrón Estructural – ADAPTER

Este patrón se debe utilizar cuando:

- Se quiere utilizar una clase que llame a un método a través de una interface, pero se busca utilizarlo con una clase que no implementa ese interface.
- Se busca determinar dinámicamente qué métodos de otros objetos llama un objeto.
- No se quiere que el objeto llamado tenga conocimientos de la otra clase de objetos.



## Patrón Estructural – ADAPTER

```
public interface IPersonaNueva {  
  
    public String getNombre();  
    public void setNombre(String nombre);  
    public int getEdad();  
    public void setEdad(int edad);  
  
}
```

```
public class PersonaNueva implements IPersonaNueva {  
    private String nombre;  
    private int edad;  
  
    public String getNombre() {  
        return nombre;  
    }  
  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
  
}
```

```
public interface IPersonaVieja {  
  
    public String getNombre();  
    public void setNombre(String nombre);  
    public String getApellido();  
    public void setApellido(String apellido);  
    public Date getFechaDeNacimiento();  
    public void setFechaDeNacimiento(Date fechaDeNacimiento);  
  
}
```

```
public class PersonaVieja implements IPersonaVieja {  
    private String nombre;  
    private String apellido;  
    private Date fechaDeNacimiento;  
  
    public String getNombre() {  
        return nombre;  
    }  
  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
  
}
```

## Patrón Estructural – ADAPTER

```
public class ViejaToNuevaAdapter implements IPersonaNueva {
    private IPersonaVieja vieja;

    public ViejaToNuevaAdapter(IPersonaVieja vieja) {
        this.vieja = vieja;
    }

    public int getEdad() {
        GregorianCalendar c = new GregorianCalendar();
        GregorianCalendar c2 = new GregorianCalendar();
        c2.setTime(vieja.getFechaDeNacimiento());
        return c.get(1) - c2.get(1);
    }

    public String getNombre() {
        return vieja.getNombre() + " " + vieja.getApellido();
    }

    public void setEdad(int edad) {
        GregorianCalendar c = new GregorianCalendar();
        int anioActual = c.get(1);
        c.set(1, anioActual - edad);
        vieja.setFechaDeNacimiento(c.getTime());
    }

    public void setNombre(String nombreCompleto) {
        String[] name = nombreCompleto.split(" ");
        String firstName = name[0];
        String lastName = name[1];
        vieja.setNombre(firstName);
        vieja.setApellido(lastName);
    }
}
```

Δet

```
public static void main(String[] args) {

    PersonaVieja personaVieja = new PersonaVieja();
    personaVieja.setApellido("Perez");
    personaVieja.setNombre("Maxi");
    GregorianCalendar g = new GregorianCalendar();
    g.set(2000, 01, 01);
    // seteamos que nacio en el año 2000
    Date d = g.getTime();
    personaVieja.setFechaDeNacimiento(d);
    // hasta aqui creamos un PersonaVieja como se hacia antes

    // ahora veremos como funciona el adapter

    ViejaToNuevaAdapter personaNueva = new ViejaToNuevaAdapter(personaVieja);

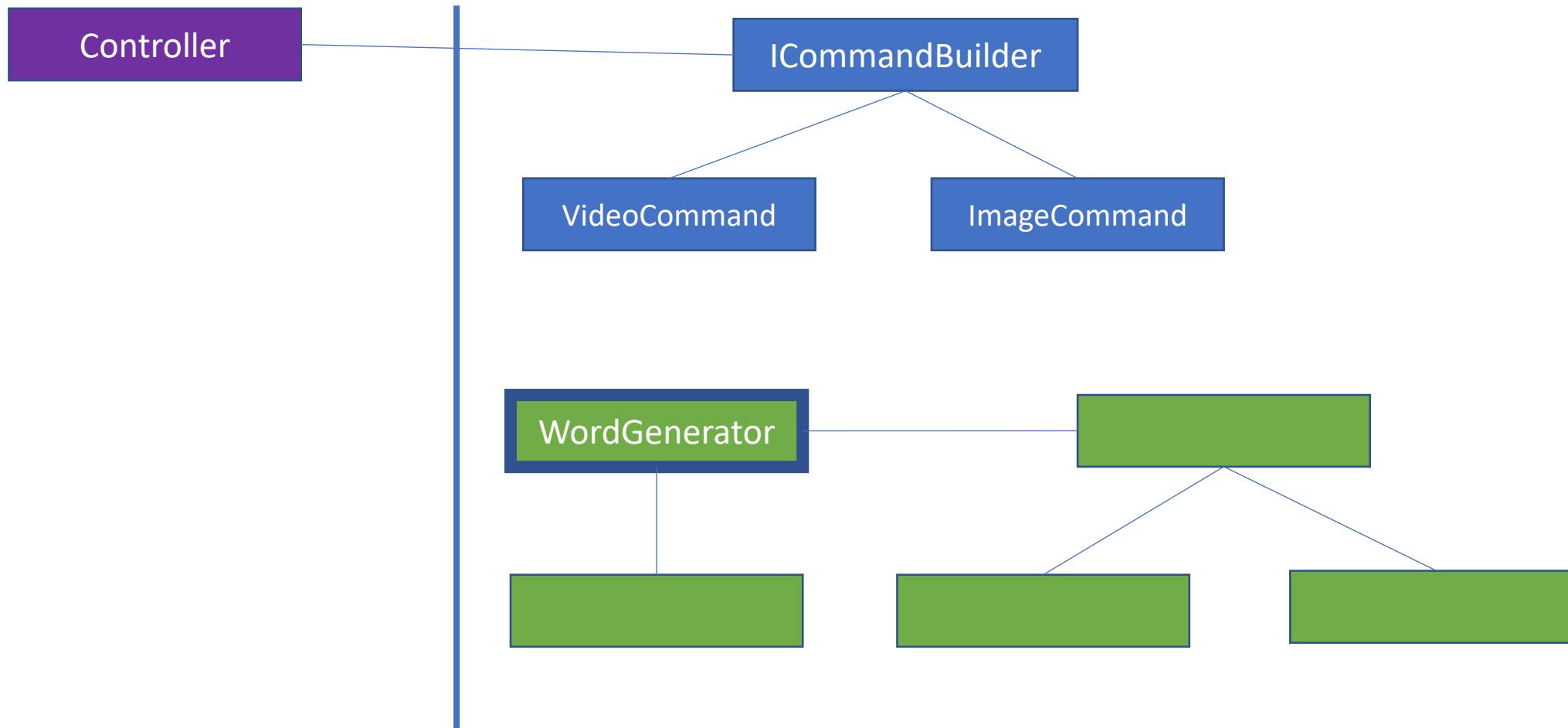
    System.out.println(personaNueva.getEdad());
    System.out.println(personaNueva.getNombre());

    personaNueva.setEdad(10);
    personaNueva.setNombre("Juan Perez");

    System.out.println(personaNueva.getEdad());
    System.out.println(personaNueva.getNombre());
}
```

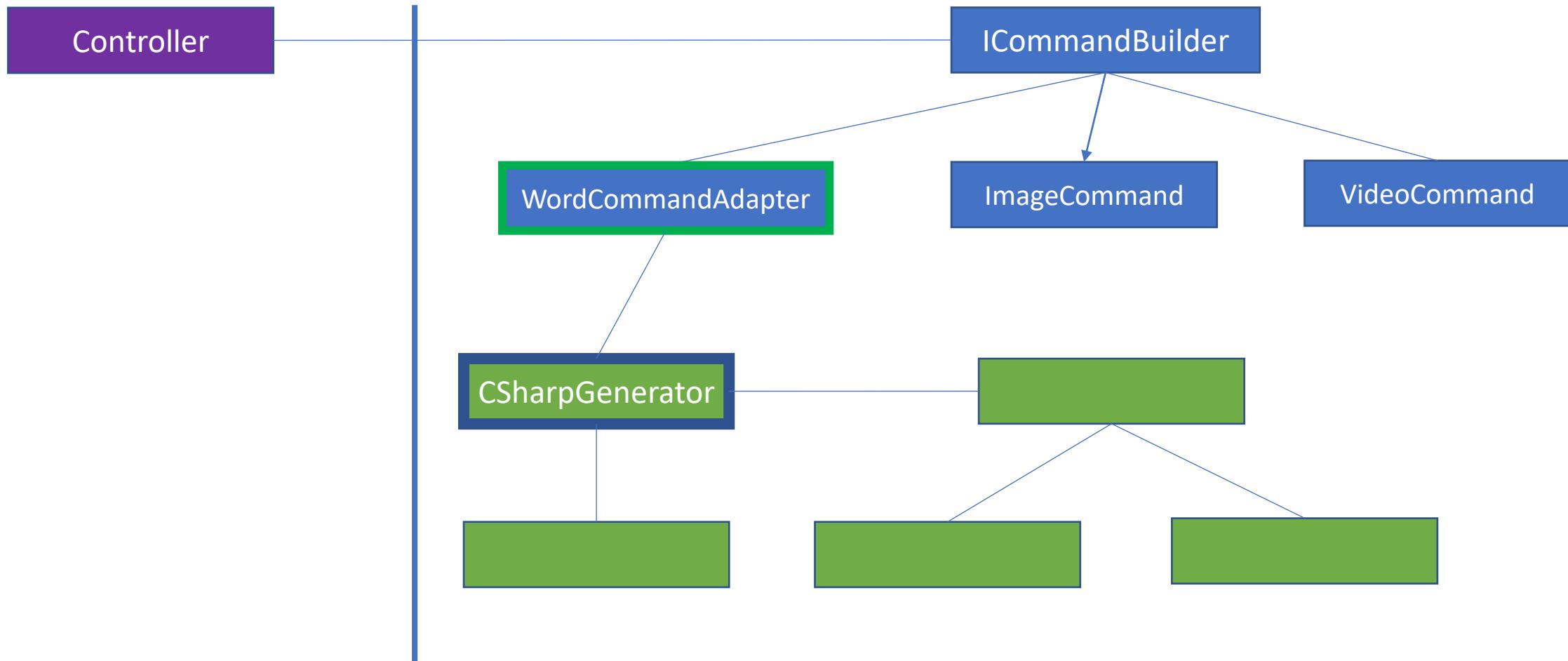
Patrones

## Patrón Estructural – ADAPTER



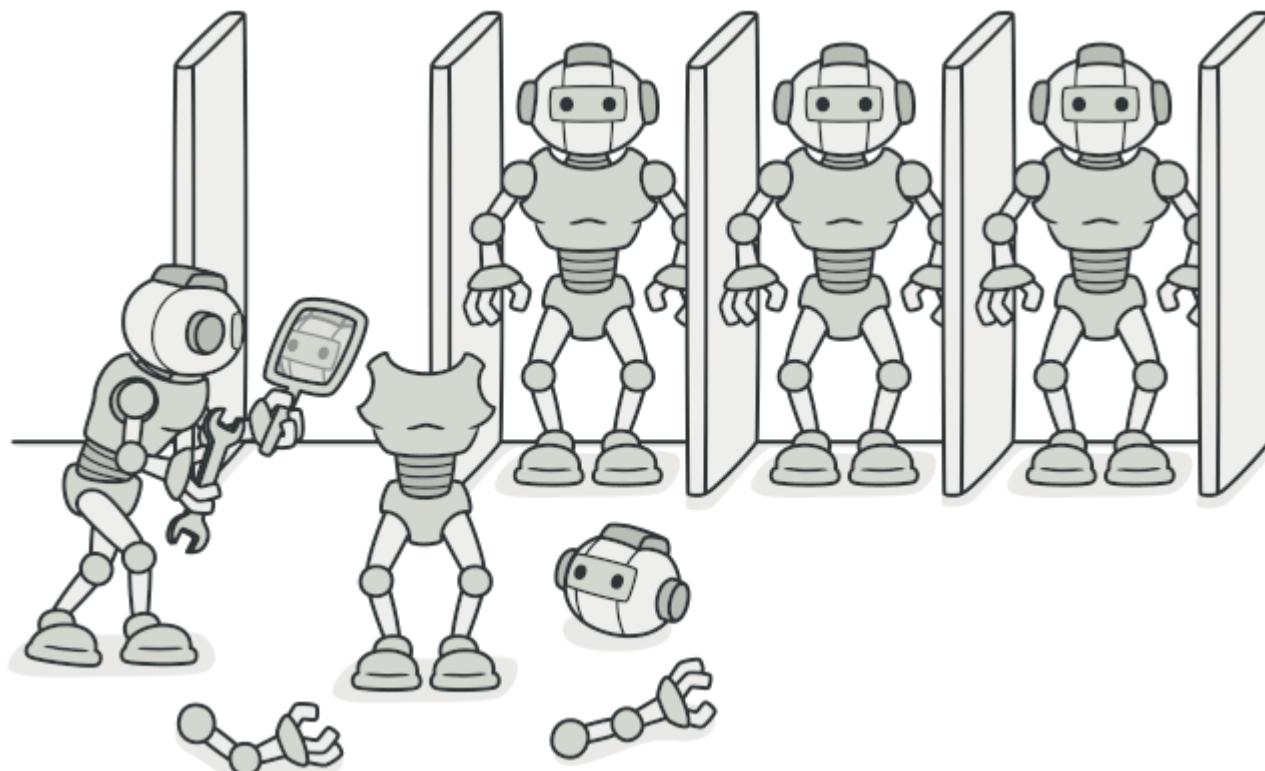
Patrones

## Patrón Estructural – ADAPTER



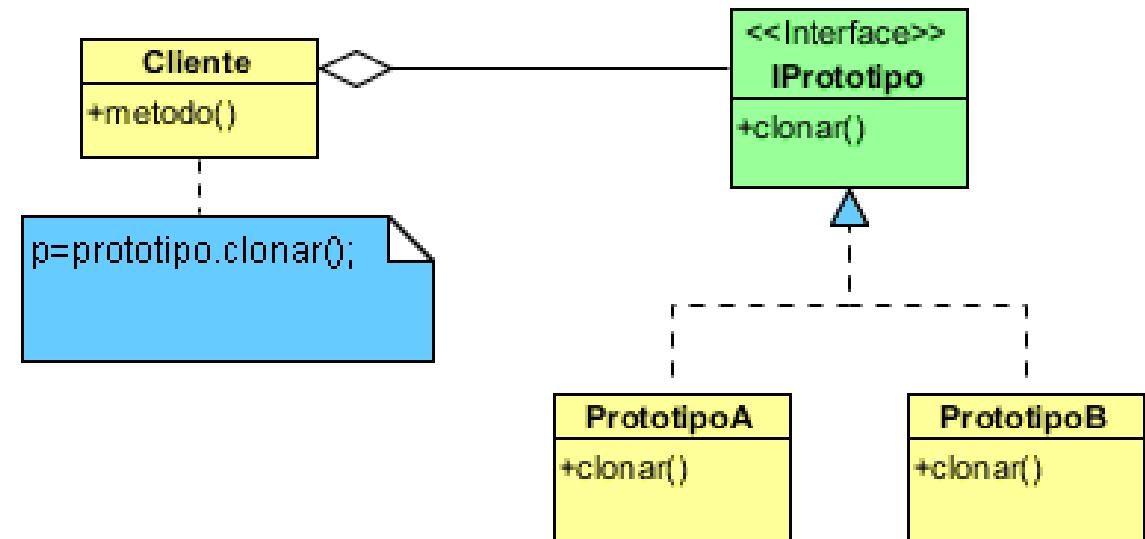
# Prototype

---



## Patrón creacional – Prototipo

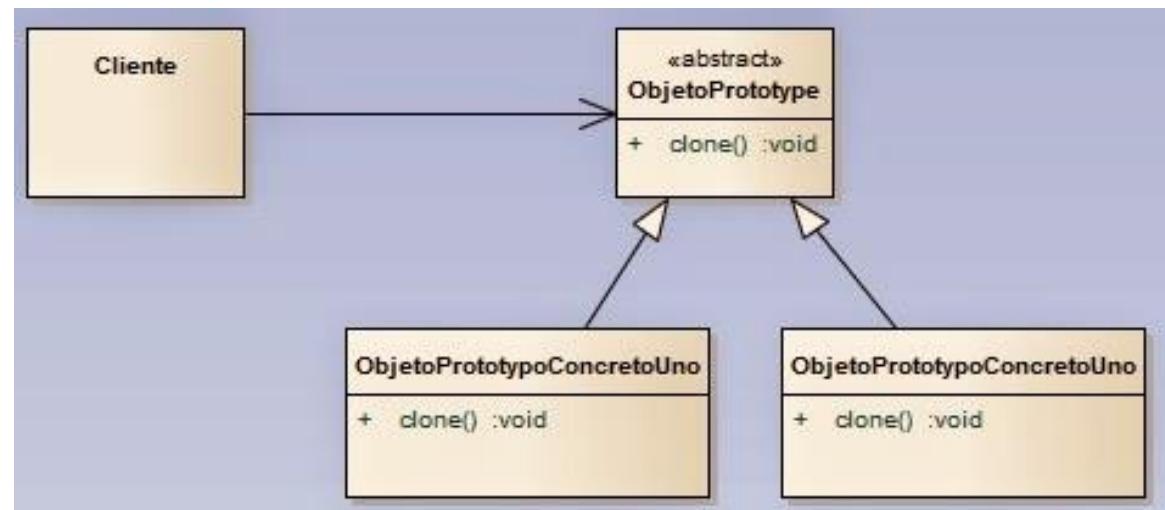
- El patrón de diseño **Prototype** sirve para crear clonaciones de objetos (instancias de clases) a fin de no acarrear todo lo que lleva la creación del mismo desde cero, parámetros, métodos a ejecutar, etc. Hay que tener en cuenta que clonar un objeto es mucho mas rápido que crearlo.



## Patrón creacional – Prototipo

### Participantes:

- **ObjectPrototype:** Declara la interface del objeto que se clona. Suele ser una clase abstracta.
- **ObjectPrototypeConcreto:** Las clases en este papel implementan una operación por medio de la clonación de sí mismo.
- **Cliente:** Crea nuevos objetos solicitando al prototipo que se clone.

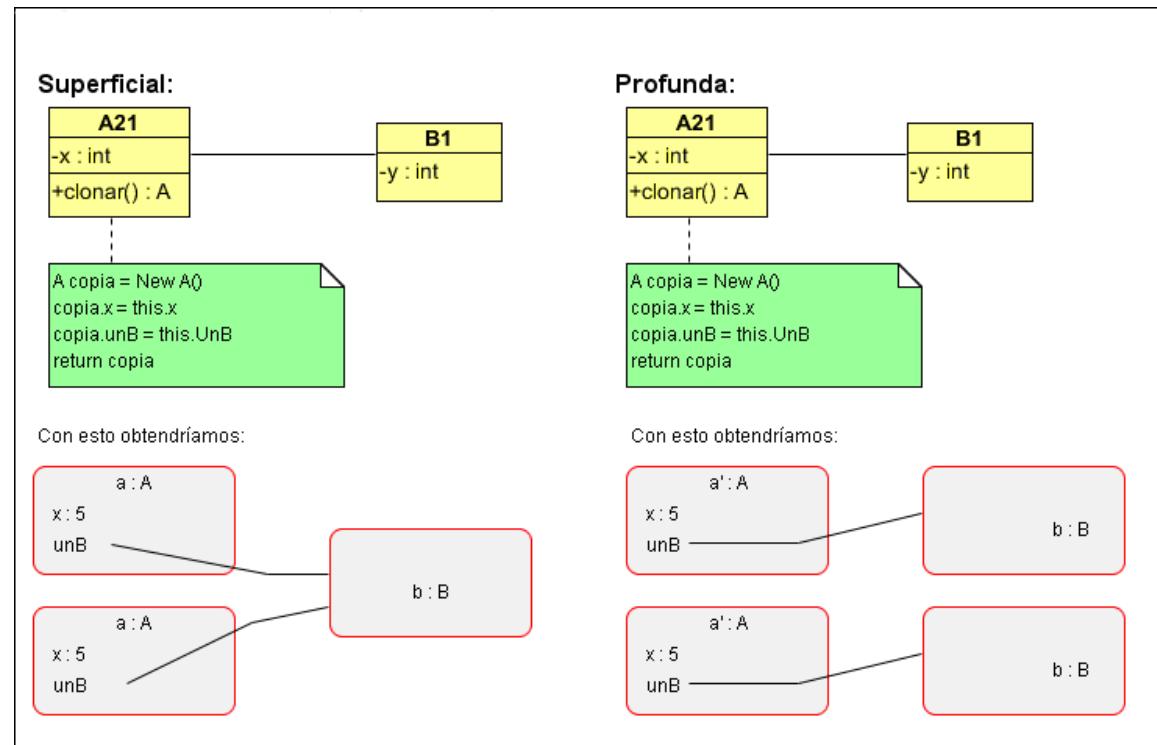


## Clonación profunda

Esta clonación se realiza cuando el objeto que se quiere clonar tiene como atributos otros objetos (1 o más) que se deben clonar de igual manera, para retornar una clonación completa del objeto y no referencias a otros ya existentes.

### Clonación superficial

Por otro lado, esta clonación es aquella que se realiza cuando el objeto que se quiere clonar no posee otros que se deban copiar(tales como Integer, Char, Bool). Se realiza a nivel de bits.



## Patrones

```
package ar.com.patronesdisenio.prototype;  
/**  
 * @author nconde  
 */  
public abstract class Bicicleta implements Cloneable {  
  
    private String color;  
    private String rodado;  
  
    /**  
     * Metodo clonador  
     */  
    public Bicicleta clone() throws CloneNotSupportedException {  
        return (Bicicleta) super.clone();  
    }  
  
    public abstract String verBicleta();  
  
    //Getters and Setters  
    public String getColor() {  
        return color;  
    }  
  
    public void setColor(String color) {  
        this.color = color;  
    }  
  
    public String getRodado() {  
        return rodado;  
    }  
  
    public void setRodado(String rodado) {  
        this.rodado = rodado;  
    }  
}
```

## Patrones

```
package ar.com.patronesdisenio.prototype.impl;

import ar.com.patronesdisenio.prototype.Bicicleta;

/**
 * @author usuario
 */
public class BicicletaModificada extends Bicicleta {

    @Override
    public String verBicleta() {

        return "Este es el color: " + this.getColor() + " El rodado es: " + this.getRodado();
    }

}
```

```
package ar.com.patronesdisenio.prototype.impl;
import ar.com.patronesdisenio.prototype.Bicicleta;

/**
 * @author usuario
 *
 */
public class Cliente {

    /**
     * @param args
     * @throws CloneNotSupportedException
     */
    public static void main(String[] args) throws CloneNotSupportedException {
        Bicicleta bc = new BicicletaModificada();
        bc.setColor("Roja");
        bc.setRodado("22");
        System.out.println(bc.verBicleta());

        Bicicleta bc2 = bc.clone();
        bc2.setColor("Negro");
        bc2.setRodado("30");

        System.out.println(bc2.verBicleta());
    }
}
```

## Patrón creacional – Prototipo

Todas las clases en Java heredan un método de la clase **Object** llamado **clone**.

Un método clone de un objeto retorna una copia de ese objeto. Esto solamente se hace para instancias de clases que dan permiso para ser clonadas. Una clase da permiso para que su instancia sea clonada si, y solo si, ella implementa el interface Cloneable, en el caso que no la clase no tenga permisos para ser clonada lanzara una excepción CloneNotSupportedException.

# Patrones

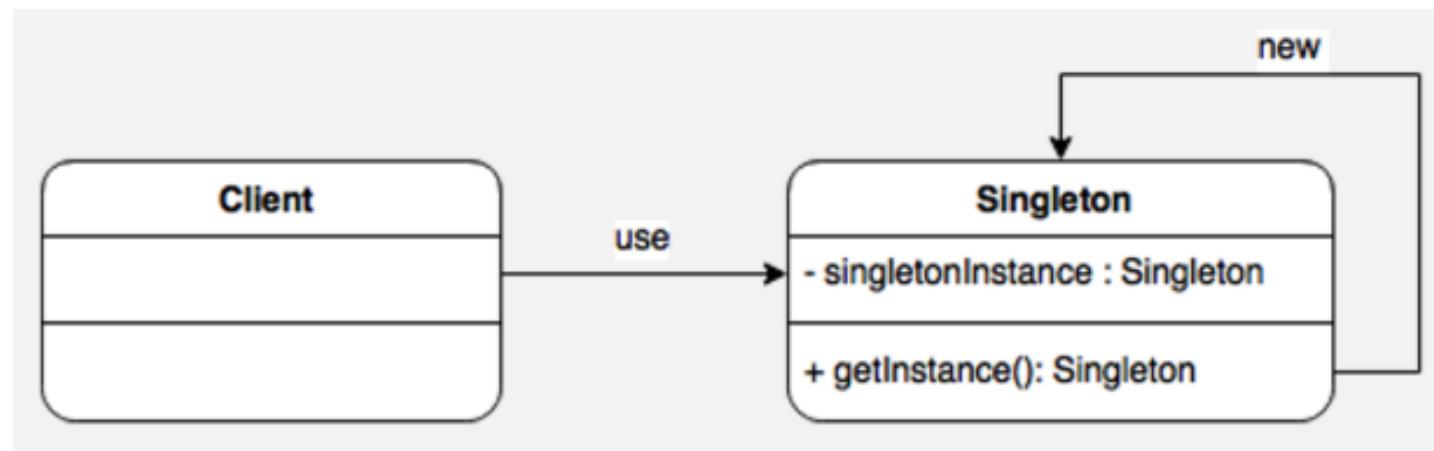
---



## Patrón de creación – SINGLETON

Utilizaremos el patrón **Singleton** cuando por alguna razón necesitemos que exista sólo una instancia (un objeto) de una determinada Clase.

Dicha clase se creará de forma que tenga una **propiedad estática** y un **constructor privado**, así como **un método público estático** que será el encargado de crear la instancia (cuando no exista) y guardar una referencia a la misma en la propiedad estática (devolviendo ésta).



## Patrón de creación – SINGLETON

La implementación **más usual** en JAVA es la siguiente:

```
public class Singleton {  
    private static final Singleton INSTANCE = new Singleton();  
  
    // El constructor privado no permite que se genere un constructor por defecto.  
    // (con mismo modificador de acceso que la definición de la clase)  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        return INSTANCE;  
    }  
}
```

## Patrón de creación – SINGLETON

Un ejemplo **correcto** de inicialización que permite evitar un error común en Java es a través de la sincronización de métodos:

```
public class Singleton {  
    private static Singleton INSTANCE = null;  
  
    // Private constructor suppresses  
    private Singleton(){}
  
  
    // Creador sincronizado para protegerse de posibles problemas multi-hilo  
    // otra prueba para evitar instantiación múltiple  
    private synchronized static void createInstance() {  
        if (INSTANCE == null) {  
            INSTANCE = new Singleton();  
        }  
    }
  
  
    public static Singleton getInstance() {  
        if (INSTANCE == null) createInstance();  
        return INSTANCE;  
    }
}
```

## Patrón de creación – SINGLETON

Si queremos reducir el coste de la sincronización.

```
private static void createInstance() {  
    if (INSTANCE == null) {  
        // Solo se accede a la zona sincronizada  
        // cuando la instancia no está creada  
        synchronized(Singleton.class) {  
            // En la zona sincronizada sería necesario volver  
            // a comprobar que no se ha creado la instancia  
            if (INSTANCE == null) {  
                INSTANCE = new Singleton();  
            }  
        }  
    }  
}
```

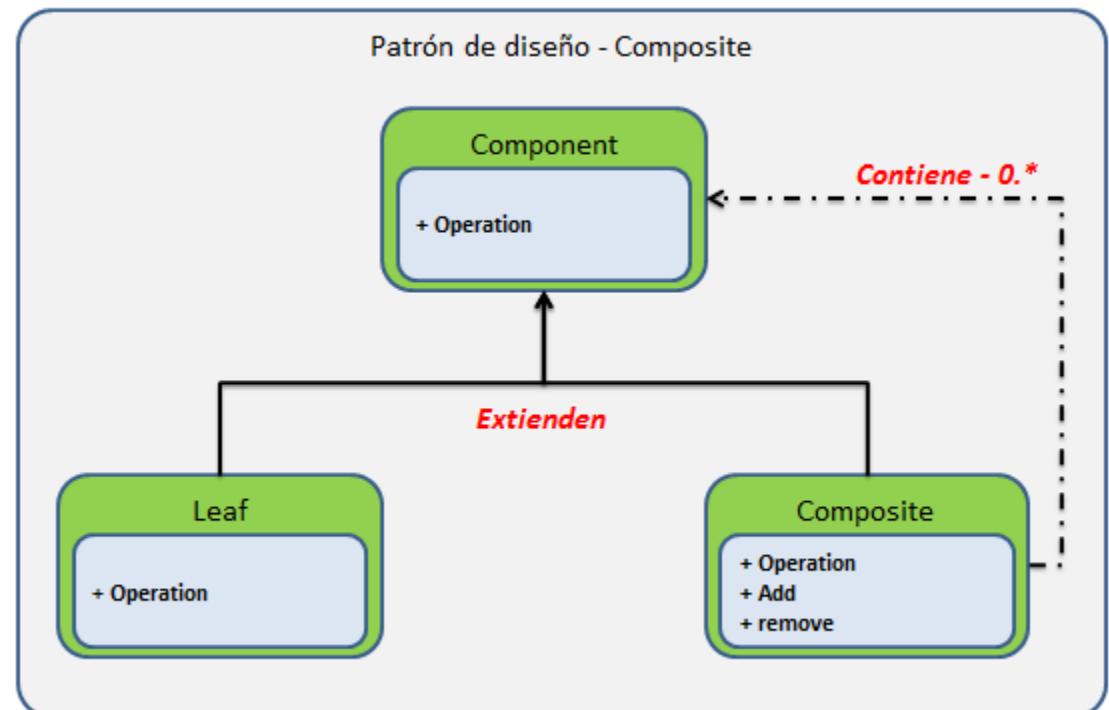
# Patrones

---



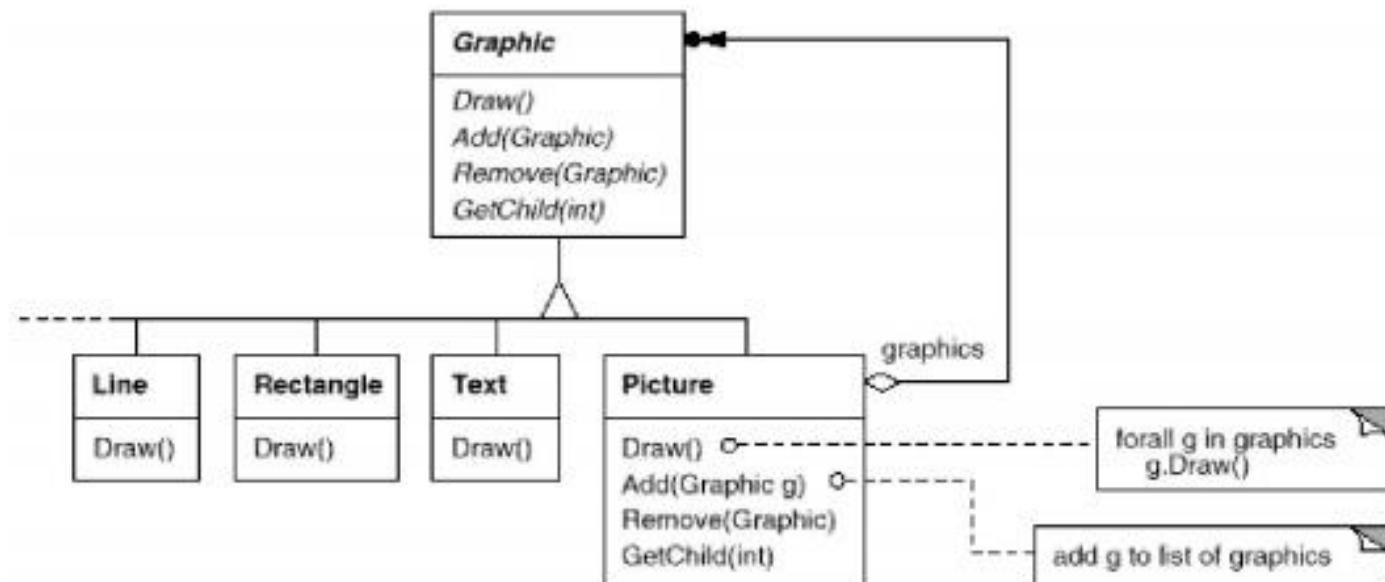
## Patrón estructura – Composite

El patrón de diseño Composite nos sirve para construir estructuras complejas partiendo de otras estructuras mucho más simples, dicho de otra manera, podemos crear estructuras compuestas las cuales están conformadas por otras estructuras más pequeñas.



Para implementar jerarquías parte-todo, el patrón Composite proporciona una interfaz de componentes unificada para los objetos simples, también llamados **objetos leaf** (hoja, en inglés), y los objetos complejos. Los objetos *leaf* simples integran esta interfaz directamente, mientras que los objetos complejos envían **peticiones específicas del cliente automáticamente a la interfaz** y a sus componentes subordinados.

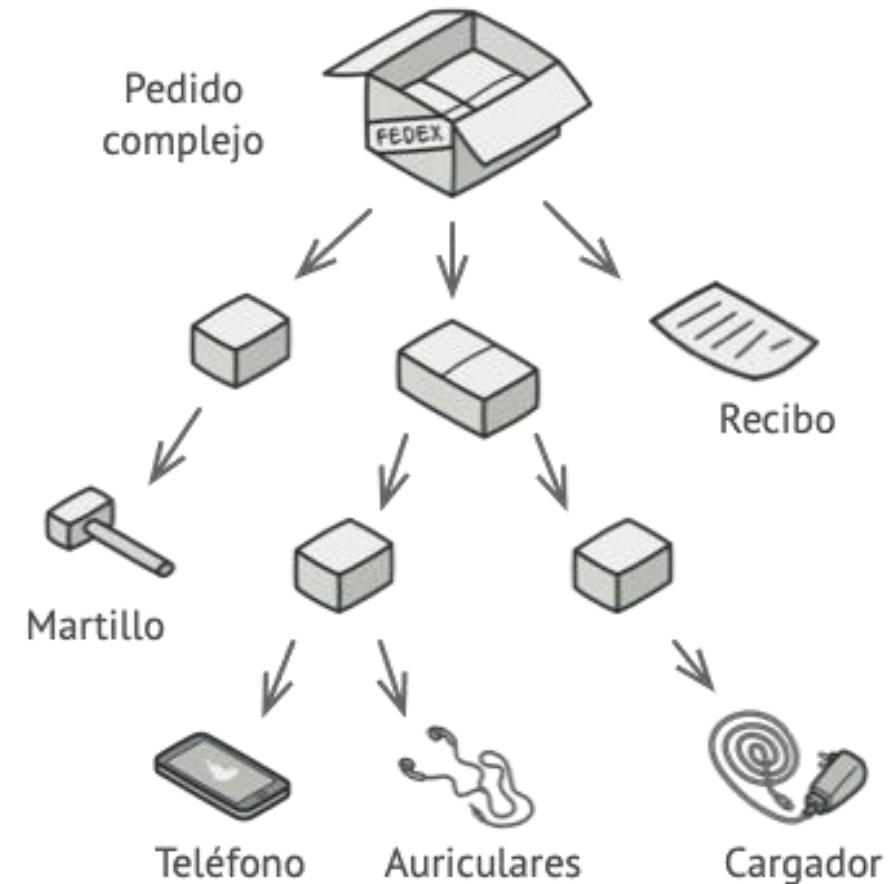
Para el cliente, resulta totalmente indiferente de qué tipo de objeto se trate (parte o todo) ya que solo se comunica con la interfaz.



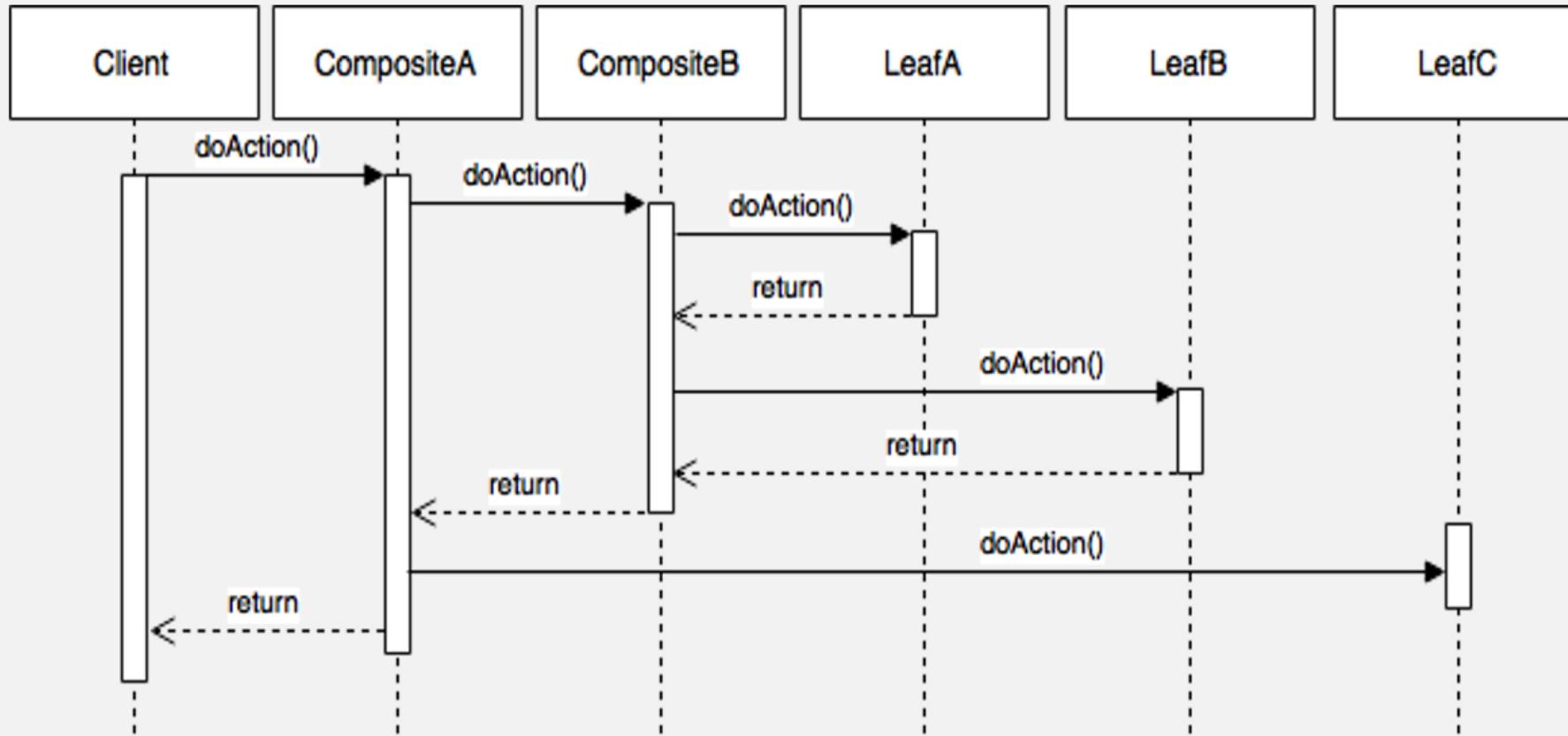
## Aplicabilidad

El patrón Composite se usa cuando:

- Cuando se quieren representar jerarquías de objetos
- Cuando queremos que los clientes puedan ignorar la diferencia entre composiciones de objetos y objetos individuales.



## Composite pattern – Diagram of sequence



## Patrón composición

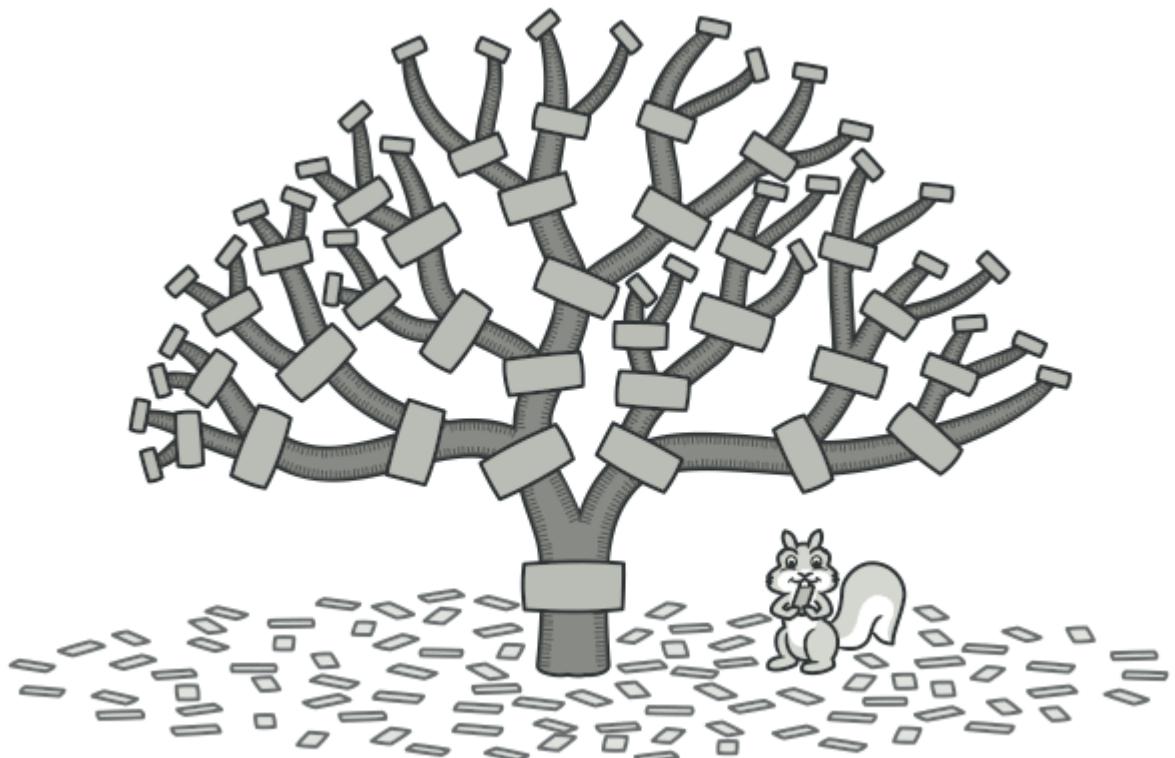
### Ventajas:

- **Define jerarquías de clases que consisten en objetos simples y en composiciones de esos objetos:** Los objetos simples pueden ser compuestos en objetos más complejos que a su vez pueden ser compuestos por otros objetos compuestos y así recursivamente. En cualquier lugar del código del cliente donde se necesite un objeto simple, también se podrá usar un objeto compuesto.
- **Hace al cliente más simple:** Los clientes pueden tratar los objetos simples y compuestos uniformemente. Los clientes normalmente no saben (y no les debería importar) si están tratando con una hoja (Leaf) o con un objeto Composite. Esto simplifica el código del cliente.
- **Hace más fácil añadir nuevos tipos de componentes:** Si se define una nueva clase Leaf o Composite, ésta funcionará automáticamente con la estructura que ya estaba definida y el cliente no tendrá que cambiar.

## Patrón composición

**desventaja:**

**Puede hacer nuestro diseño demasiado general:** La desventaja de hacer más fácil el añadir nuevos componentes es que también hace más difícil restringir los componentes de una composición. A veces queremos que una composición tenga solo un determinado tipo de componentes. Con este patrón tendríamos que hacer las comprobaciones en tiempo de ejecución.



# Patrones

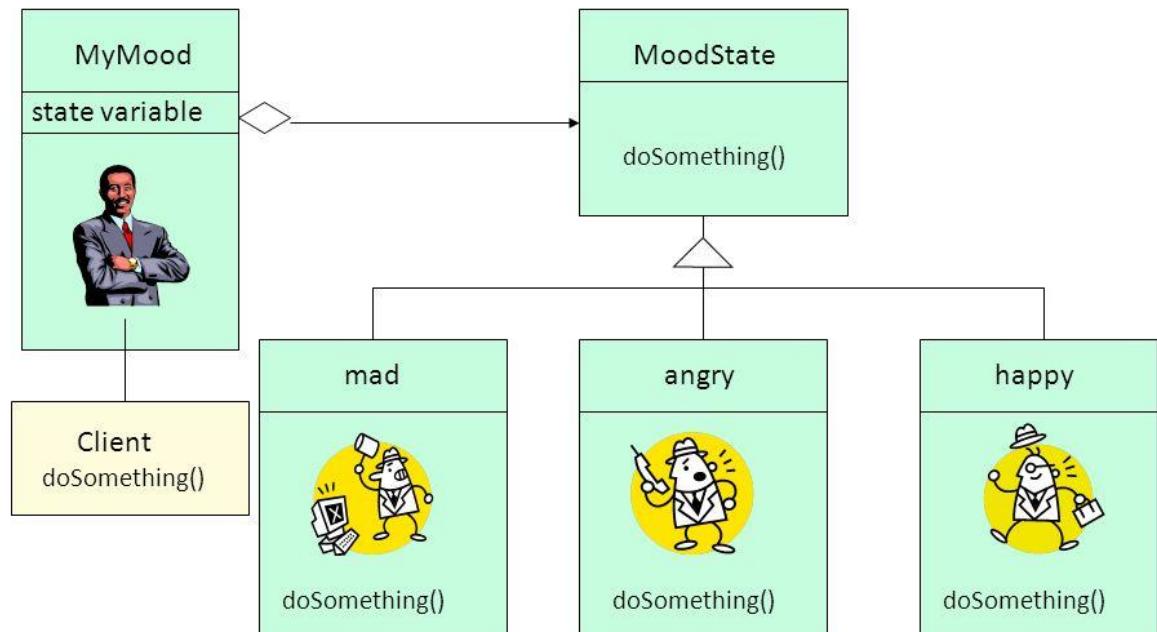
---



## Patrón de comportamiento – state

Propósito:

- Permitir a un objeto modificar su comportamiento cuando su estado interno cambia. El objeto aparecerá para cambiar su clase.



## Patrón de comportamiento – state

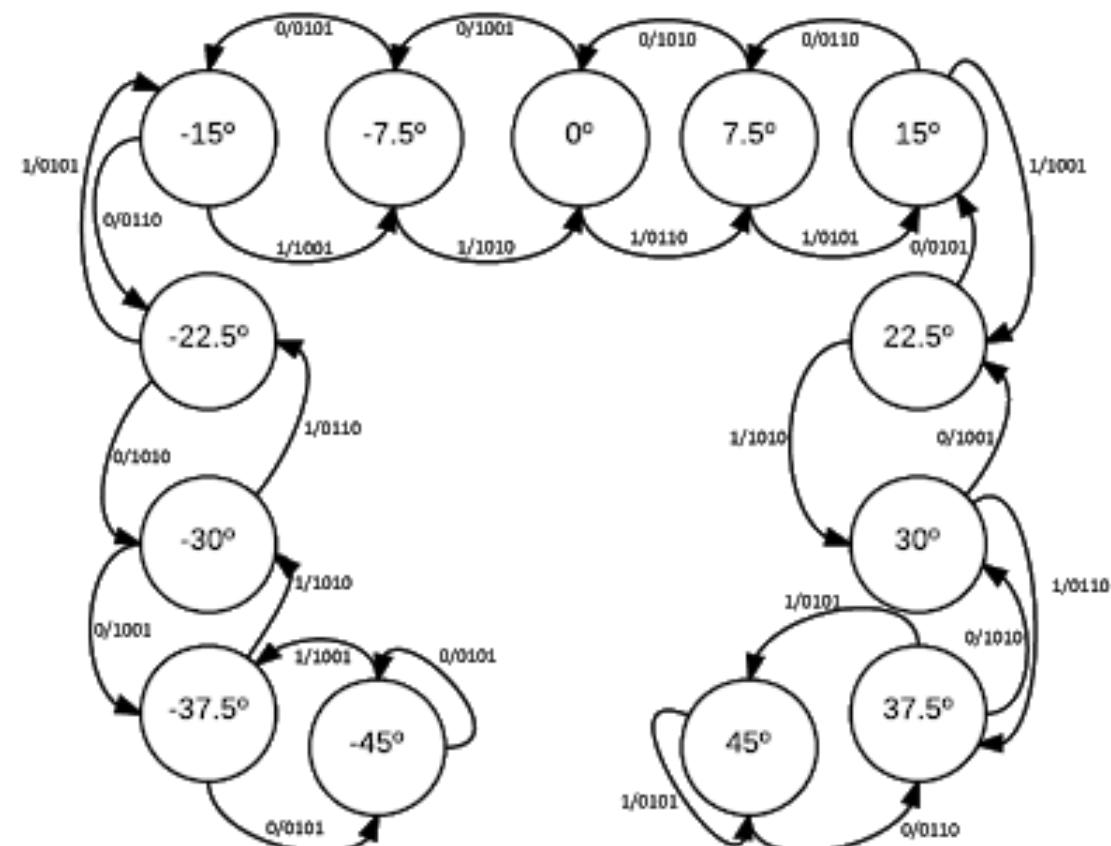
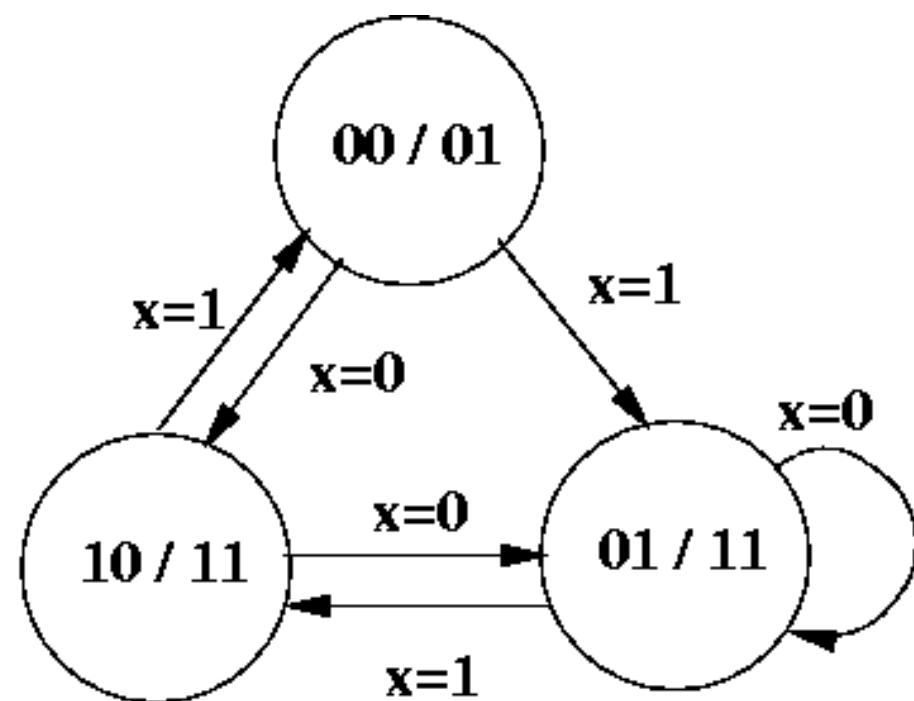
### Aplicabilidad:

El patrón State se usa en cualquiera de estos casos:

- El comportamiento de un objeto depende de su estado y debe cambiar su comportamiento en tiempo de ejecución dependiendo de ese estado.
- Las operaciones tienen muchos condicionales que dependen del estado del objeto. Este estado está representado normalmente por uno o más enumerados. A menudo, varias operaciones contendrán la misma estructura condicional. El patrón State pone cada rama del condicional en una clase separada. Esto permite tratar el estado del objeto que puede variar independientemente de otros objetos.

Más concretamente, el patrón State suele usarse para implementar máquinas de estados

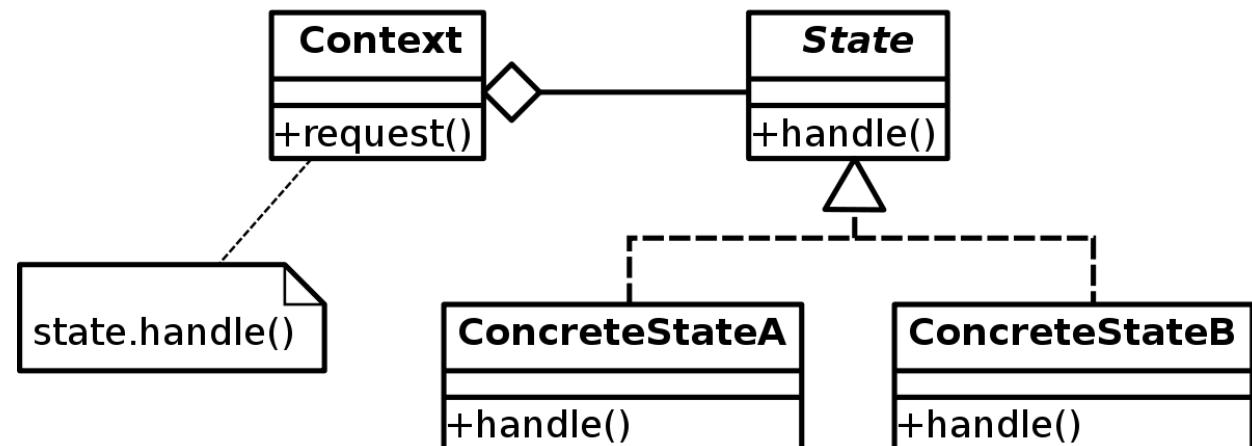
## Patrón de comportamiento – state



## Patrón de comportamiento – state

### Estructura:

La clase **Contexto** define la interfaz de interés para los clientes. La clase **State** define una interfaz para encapsular el comportamiento asociado con un estado particular del contexto. Cada clase **ConcreteState** implementa un comportamiento asociado con un estado el contexto.



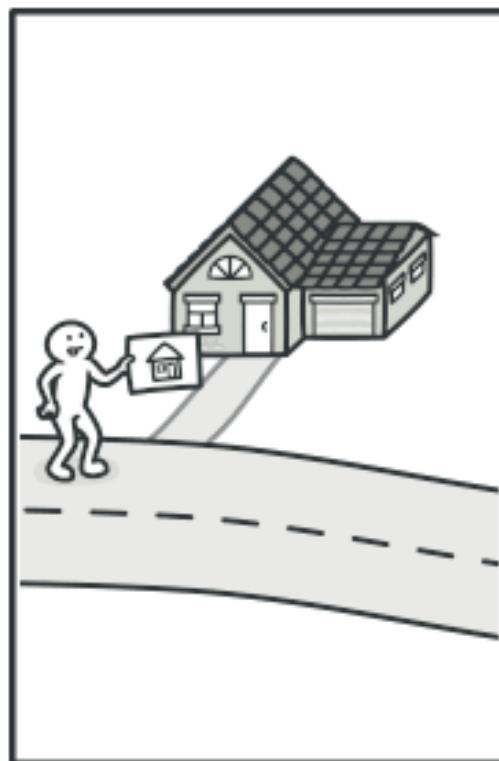
## Patrón de comportamiento – state

### Ventajas:

- **Localiza el comportamiento de un estado específico y divide el comportamiento para diferentes estados:** El patrón State pone todo el comportamiento asociado con un estado particular en un objeto. Debido a que todo el código de un estado específico está en una subclase de State, los nuevos estados y transiciones pueden ser añadidos fácilmente añadiendo nuevas subclases.
- **Hace las transiciones entre estados explícitas:** Cuando un objeto define su estado actual únicamente en términos de datos internos, sus transiciones no tienen una representación explícita. Introducir objetos separados para diferentes estados hace las transiciones más explícitas.
- **Los objetos de los estados pueden ser compartidos:** Si los objetos de los estados no tienen variables instanciadas, el estado que representan esta codificado completamente en su tipo, entonces el contexto puede compartir el objeto del estado.

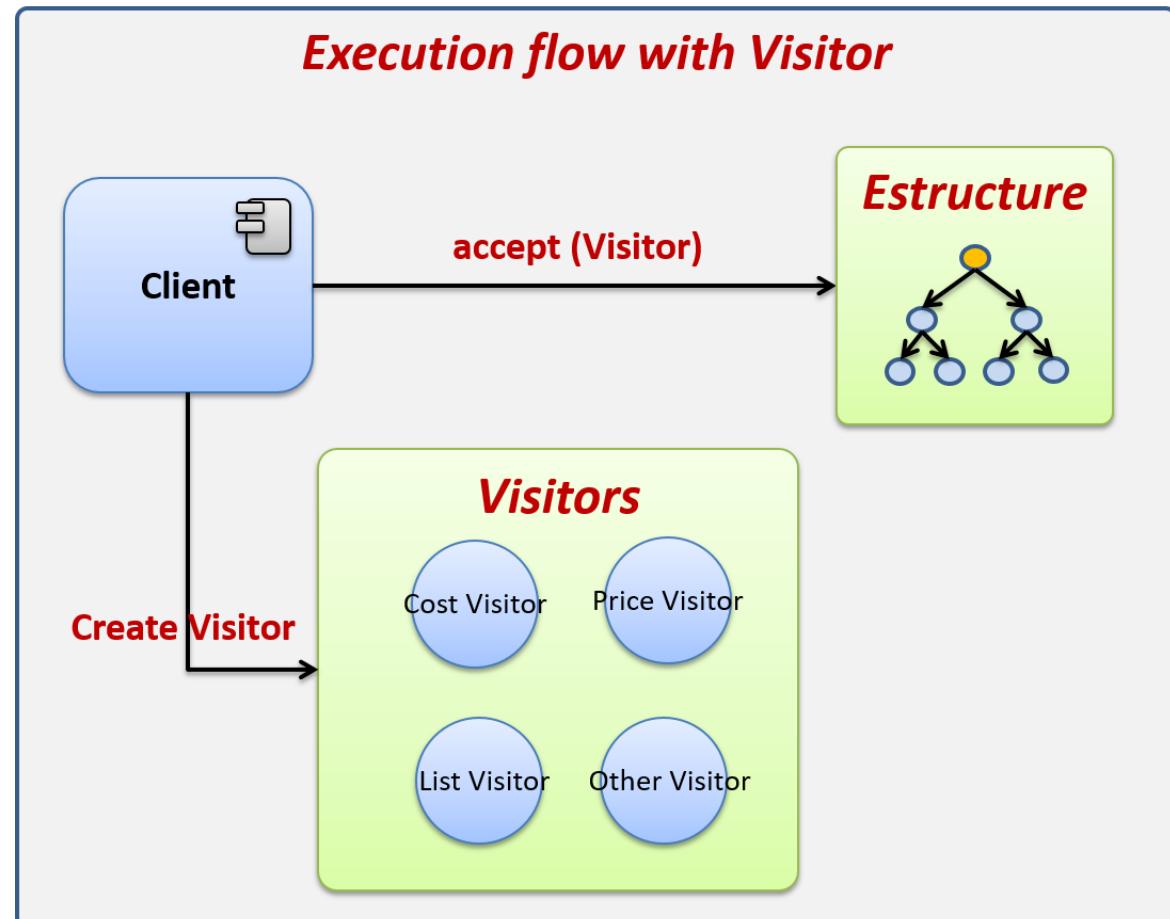
# Patrón Visitor

---



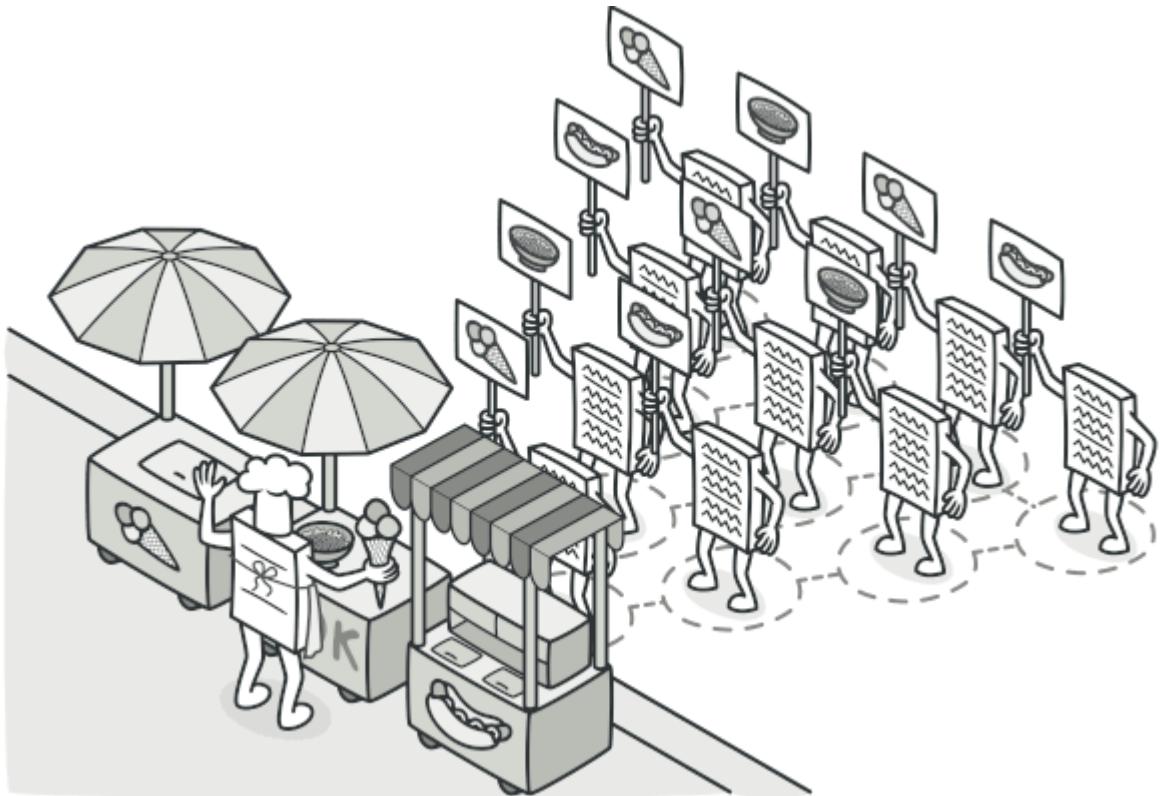
## Patrón de comportamiento – visitor

El visitor pattern es un patrón de solución para separar un algoritmo de la estructura del objeto en el que se ejecuta. Describe una forma de añadir nuevas operaciones a las estructuras de los objetos existentes sin modificar dichas estructuras. Gracias a esta propiedad, el visitor pattern es una manera de implementar el Principio Abierto-Cerrado (OCP). Este principio de desarrollo de software, orientado a objetos, se basa en el hecho de que todas las unidades de software, como los módulos, clases o métodos, están simultáneamente abiertas para extensiones y cerradas para modificaciones.

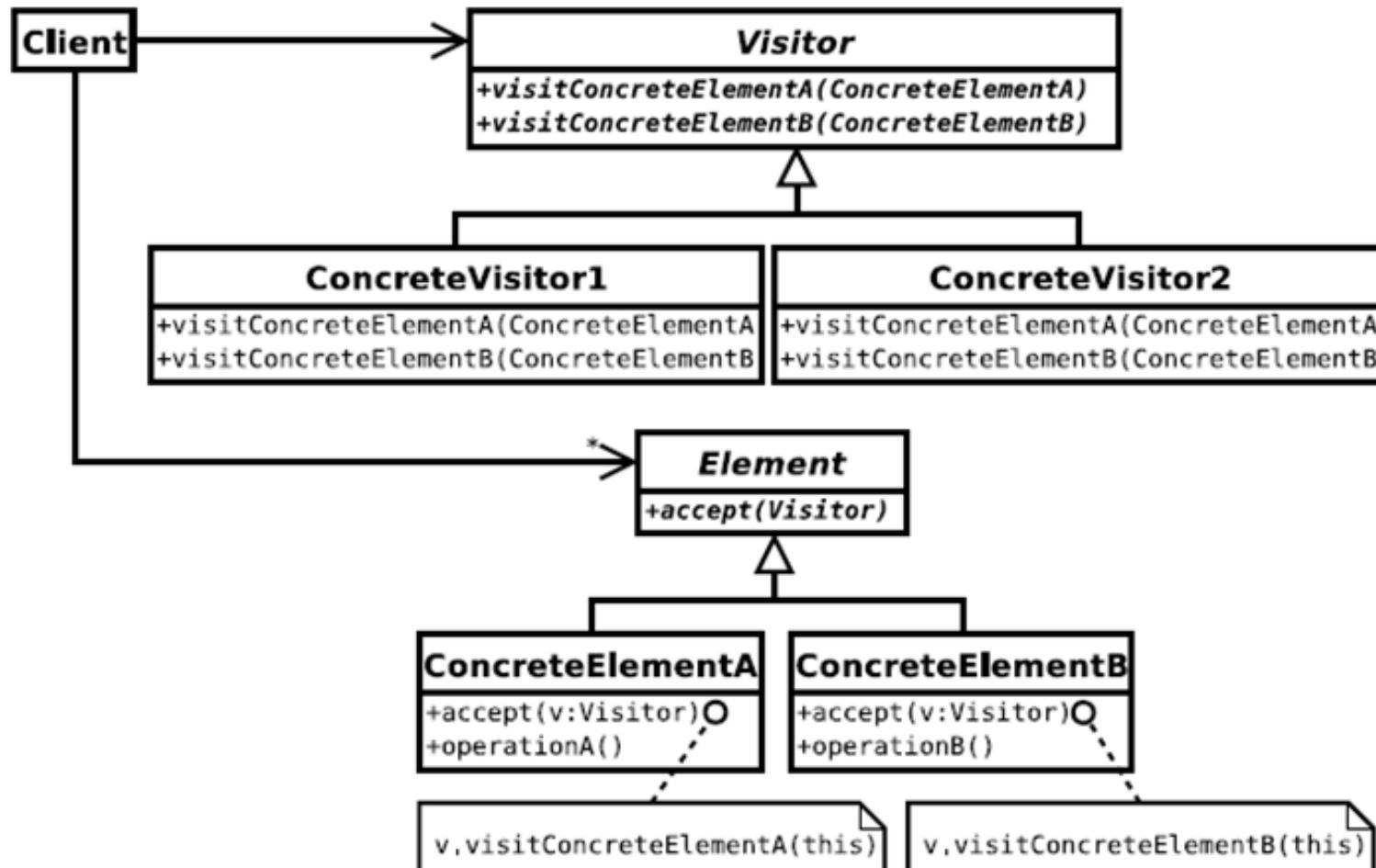


## ¿Para qué sirve el visitor design pattern?

Puesto que la estructura de los objetos está compuesta de muchas clases inconexas y requiere cada vez más operaciones, para los desarrolladores es muy inconveniente implementar una nueva subclase para cada nueva operación. El resultado es un sistema plagado con varias clases de nodos diferentes que no solo es difícil de entender, sino también de mantener y modificar. La instancia esencial del visitor pattern es el Visitor, que permite añadir nuevas funciones virtuales a una familia de clases sin modificarlas.

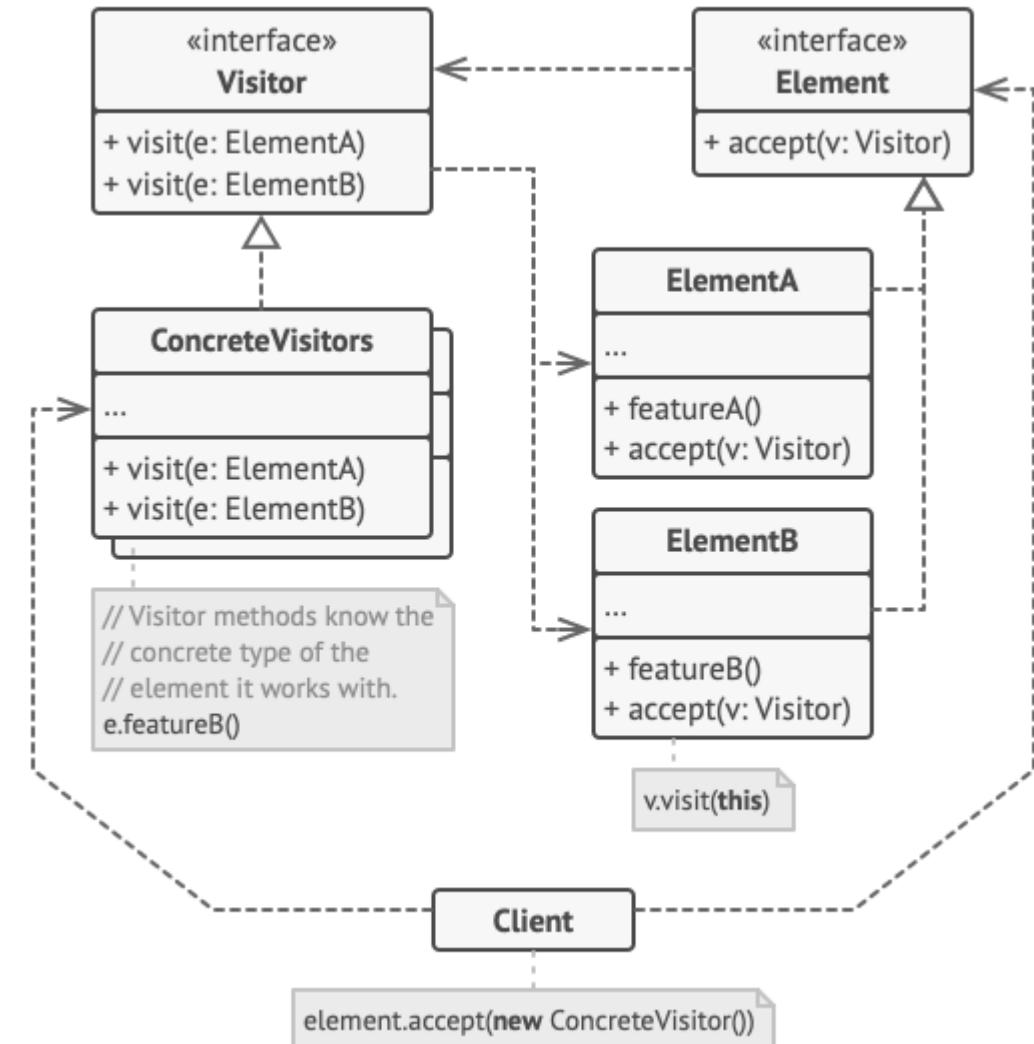


## Estructura

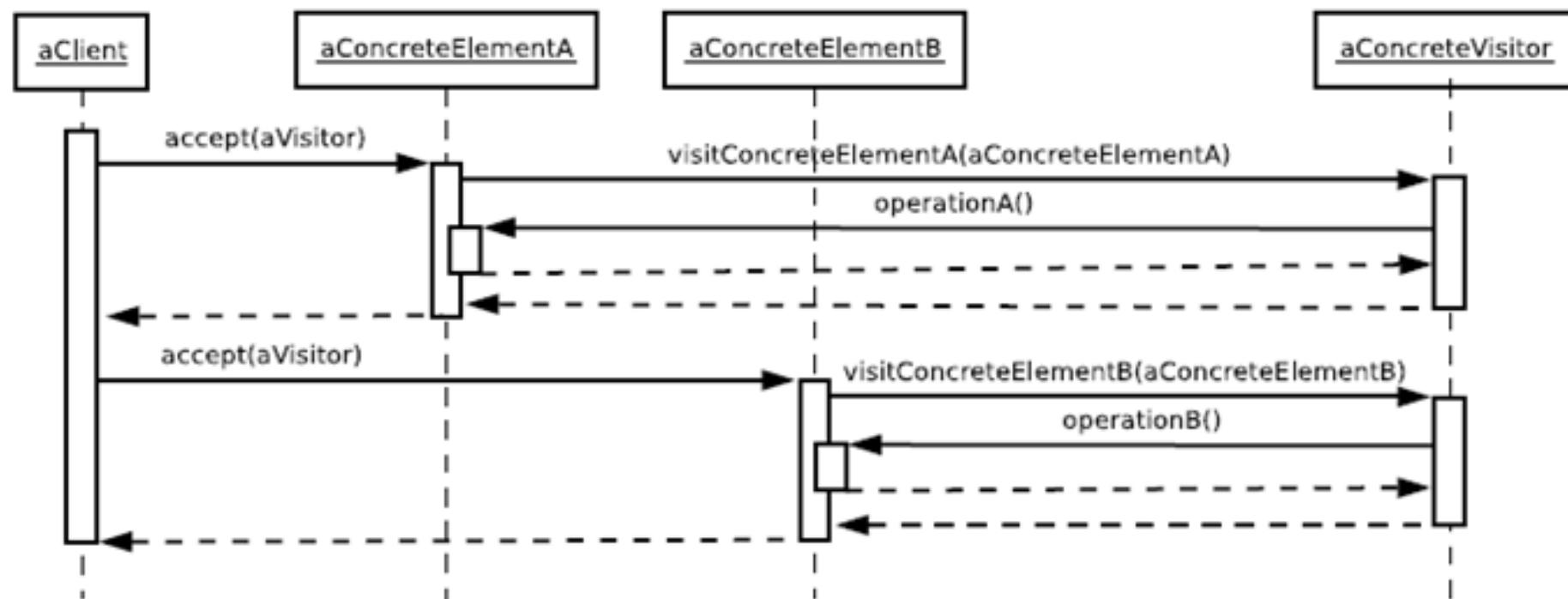
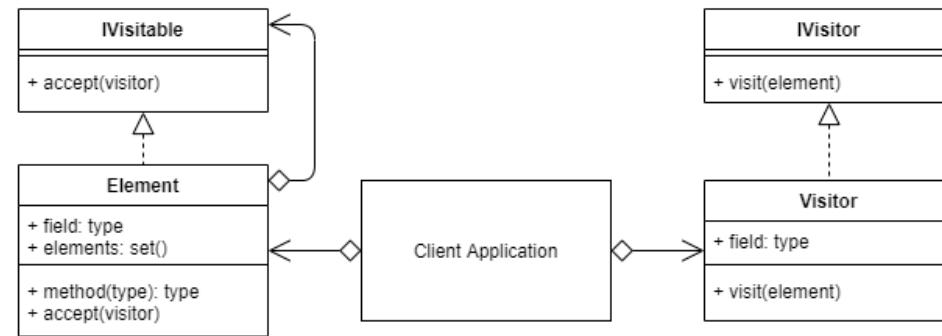


## Actores:

- **Visitante (Visitor):** Declara una operación de visita para cada elemento concreto en la estructura de objetos, que incluye el propio objeto visitado.
  - **Visitante Concreto (ConcreteVisitor):** Implementa las operaciones del visitante y acumula resultados como estado local.
  - **Elemento (Element):** Define una operación “Accept” que toma un visitante como argumento.
  - **Elemento Concreto (ConcreteElement):** Implementa la operación “Accept”.



## Patrones

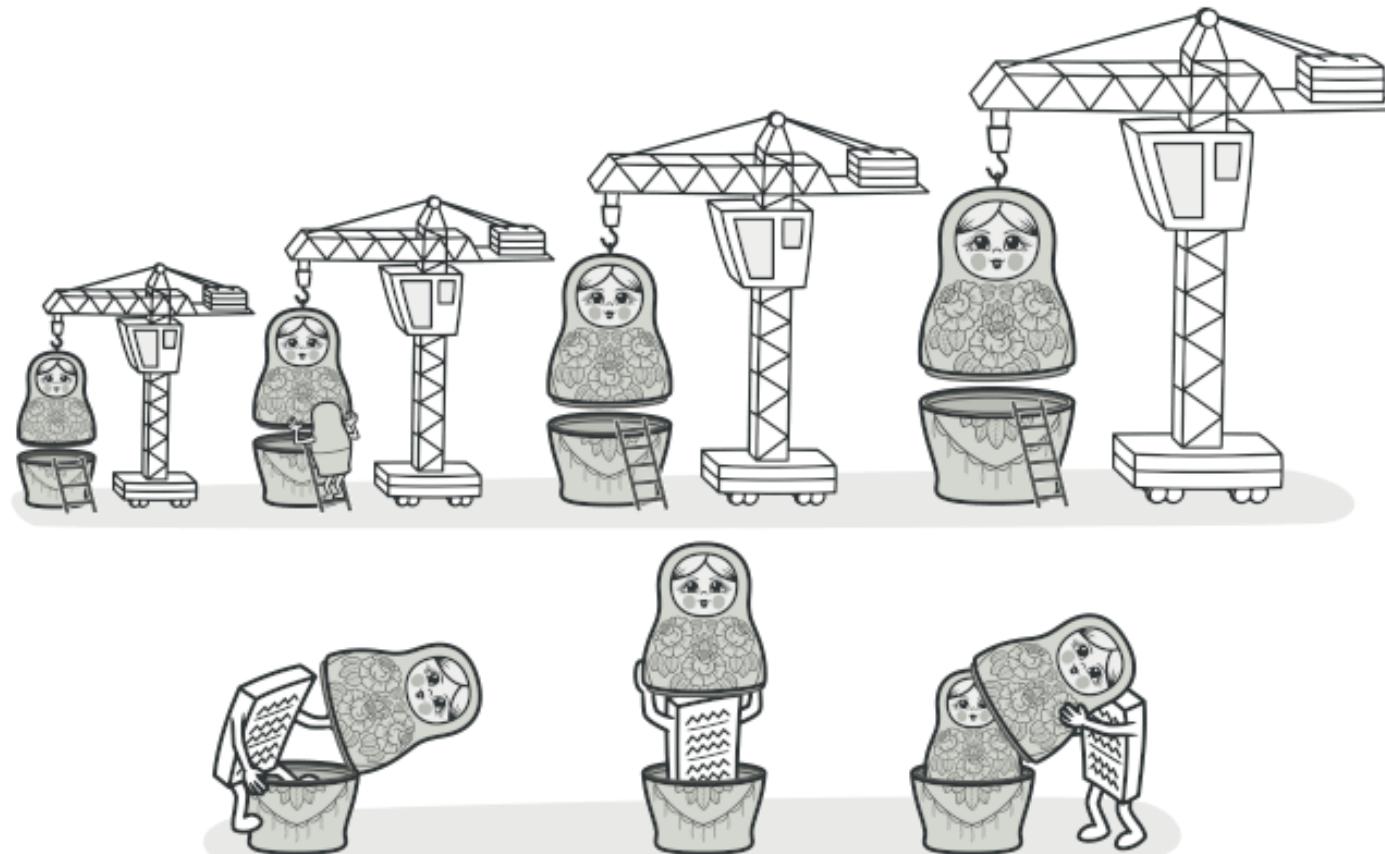


## Consecuencias:

- Es fácil añadir nuevas operaciones a un programa utilizando Visitantes, ya que el visitante contiene el código en lugar de cada una de las clases individuales.
- El patrón Visitante es útil cuando se desea encapsular buscando datos desde un número de instancias de varias clases. Los patrones de diseño sugieren que el visitante puede proporcionar una funcionalidad adicional a una clase sin cambiarla.
- Es difícil añadir nuevas clases de elementos, ya que obliga a cambiar a los visitantes.
- Facilita la acumulación de estado, es decir, acumular resultados

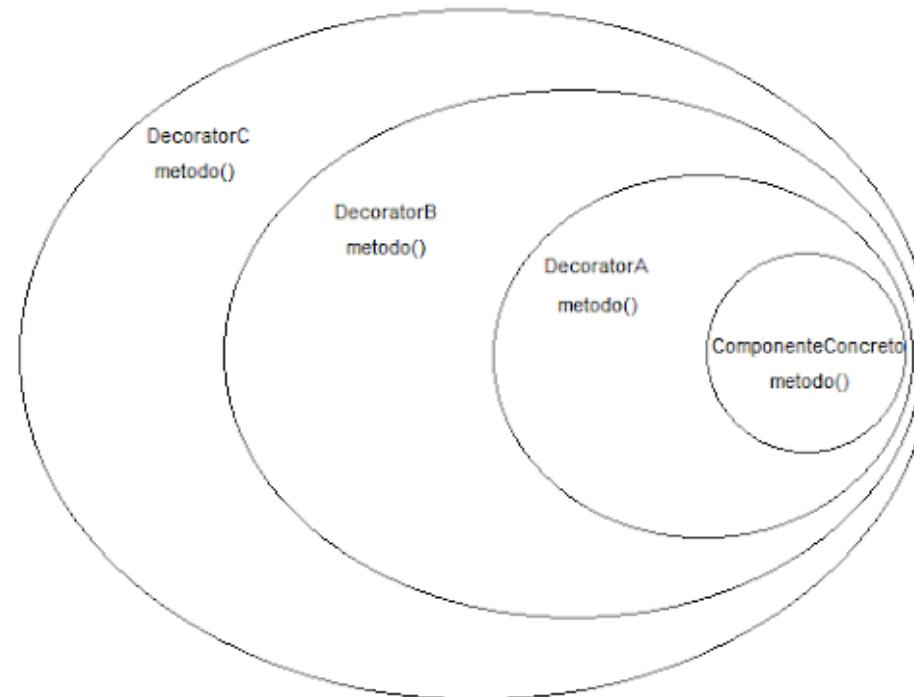
# Patrón Decorador

---



## Patrón de estructura - decorador

Decorator es un patrón de diseño estructural que permite añadir dinámicamente nuevos comportamientos a objetos colocándolos dentro de objetos especiales que los envuelven.



## ¿Cuál es la finalidad del patrón de diseño Decorator?

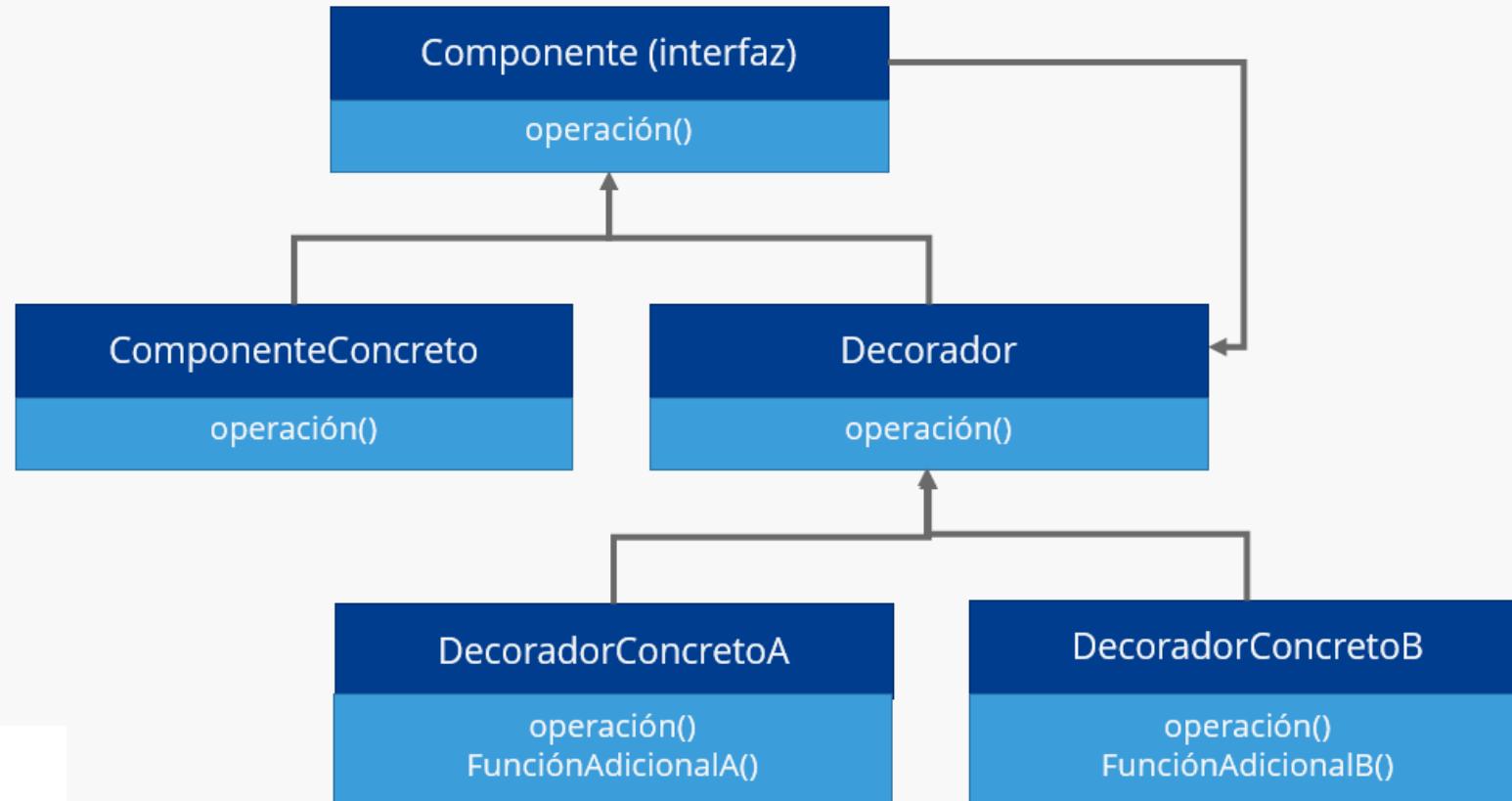
El patrón Decorator tiene como objetivo hacer que los componentes del software orientado a objetos sean más flexibles y fáciles de reutilizar. Con este fin, el enfoque permite a los desarrolladores añadir y eliminar las dependencias de un objeto de manera dinámica y, cuando sea necesario, durante el tiempo de ejecución.

**InputStream**

**InputStreamReader**

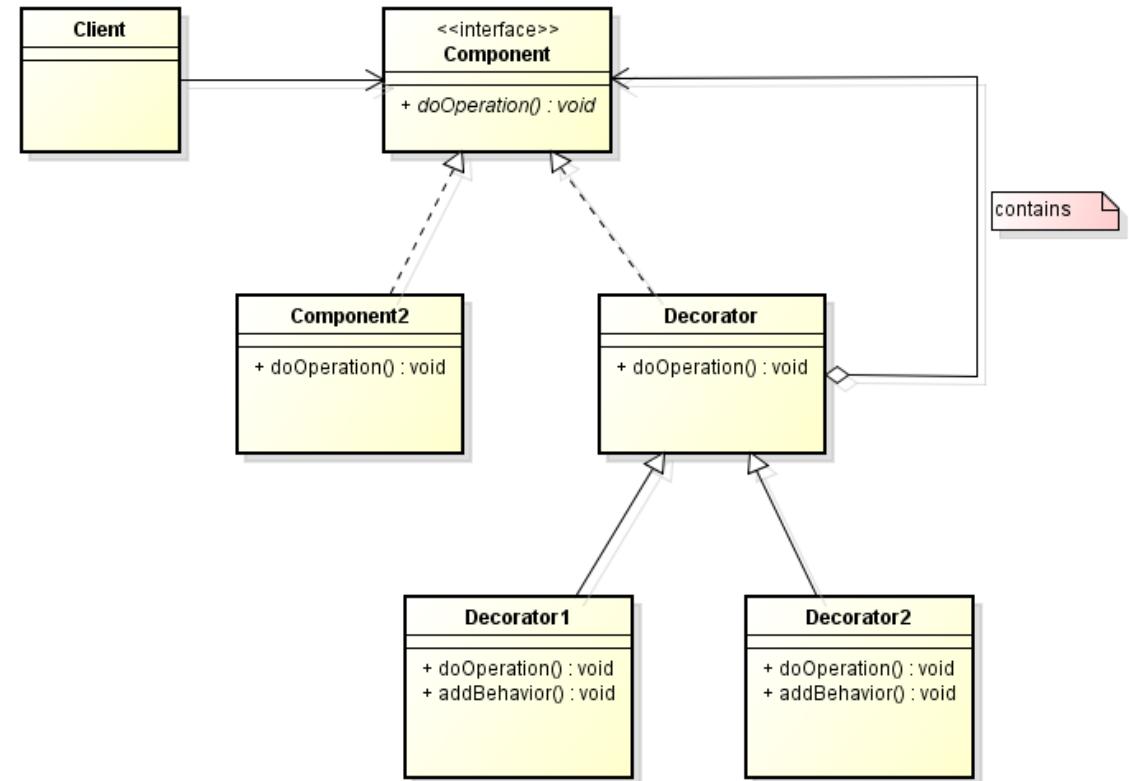
**BufferedReader**

## Diagrama UML: patrón de diseño Decorator

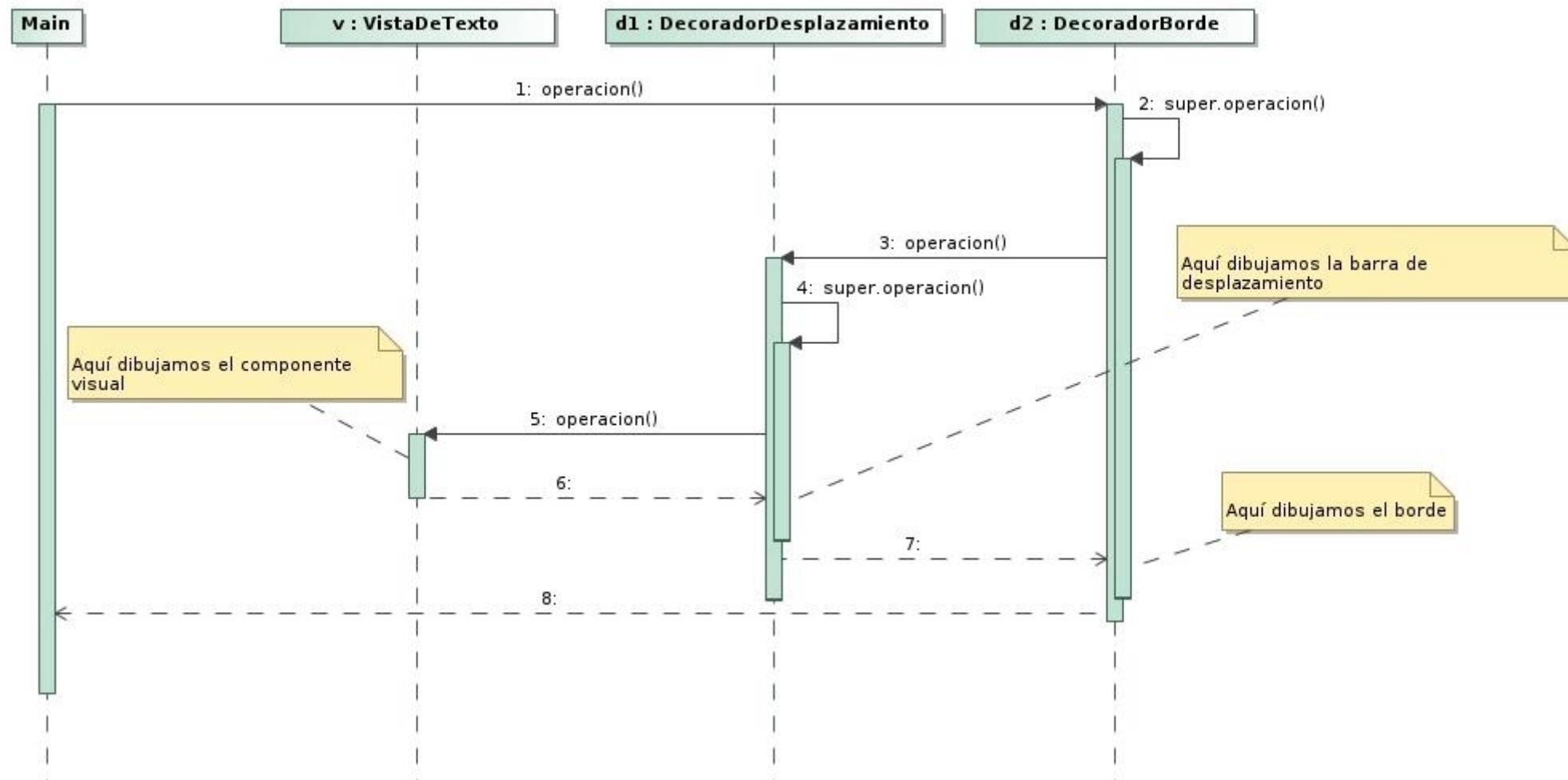


## Participantes

- **Componente:** Define la interfaz para los objetos que pueden tener responsabilidades añadidas.
- **Componente Concreto:** Define un objeto al cual se le pueden agregar responsabilidades adicionales.
- **Decorador:** Mantiene una referencia al componente asociado. Implementa la interfaz de la superclase Componente delegando en el componente asociado.
- **Decorador Concreto:** Añade responsabilidades al componente.



## Patrones



## Consecuencias:

- Más flexible que la herencia. Al utilizar este patrón, se pueden añadir y eliminar responsabilidades en tiempo de ejecución.
- Evita la aparición de clases con muchas responsabilidades en las clases superiores de la jerarquía. Este patrón nos permite ir incorporando de manera incremental responsabilidades.
- Genera gran cantidad de objetos pequeños. El uso de decoradores da como resultado sistemas formados por muchos objetos pequeños y parecidos.
- Puede haber problemas con la identidad de los objetos. Un decorador se comporta como un envoltorio transparente. Pero desde el punto de vista de la identidad de objetos, estos no son idénticos, por lo tanto no deberíamos apoyarnos en la identidad cuando estamos usando decoradores.

# Patrones

---



## Patrón Estructural – Proxy

El patrón Proxy se utiliza como intermediario para acceder a un objeto, permitiendo controlar el acceso a él.

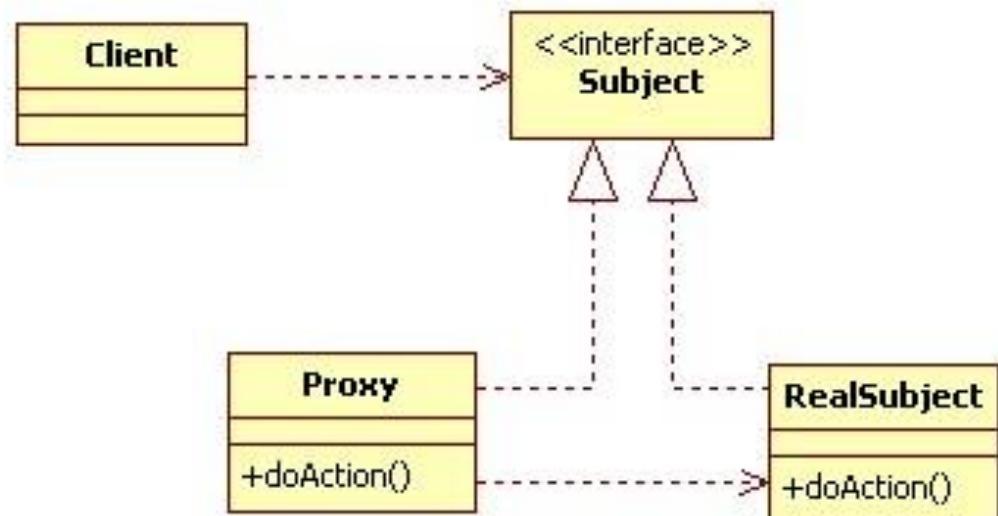
Para ello obliga que las llamadas a un objeto ocurran indirectamente a través de un objeto proxy, que actúa como un sustituto del objeto original, delegando luego las llamadas a los métodos de los objetos respectivos.



## Patrón Estructural – Proxy

### Participantes:

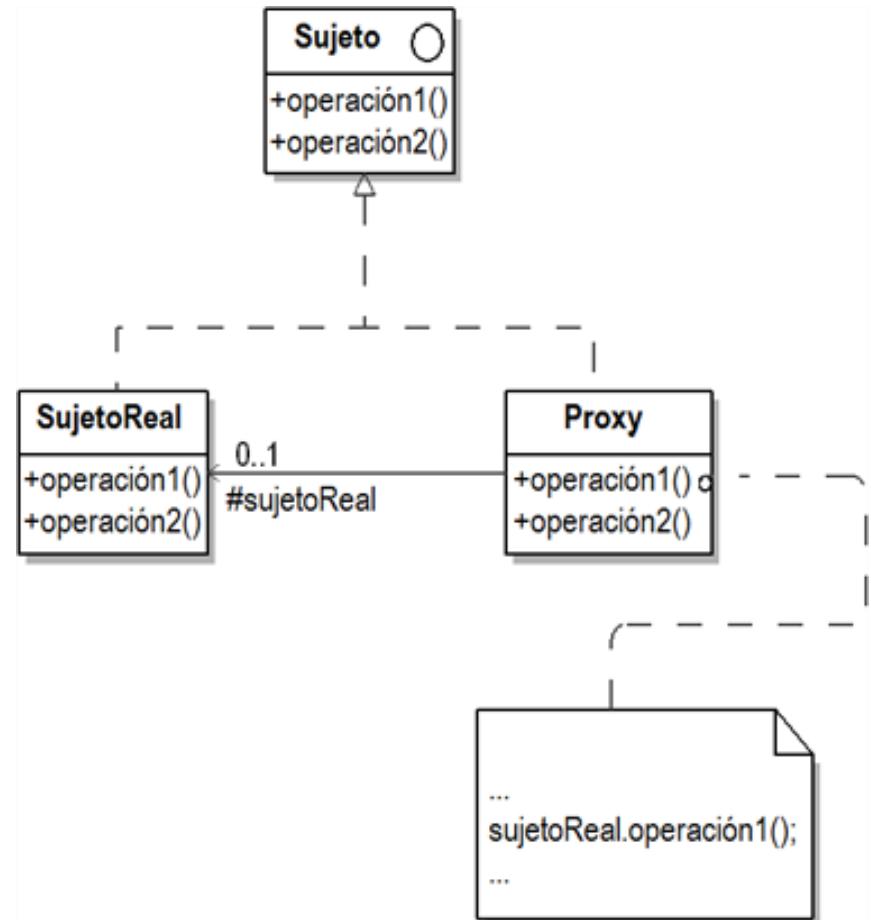
- **Subject:** interfaz o clase abstracta que proporciona un acceso común al objeto real y su representante (proxy).
- **Proxy:** mantiene una referencia al objeto real. Controla la creación y acceso a las operaciones del objeto real.
- **RealSubject:** define el objeto real representado por el Proxy.
- **Client:** solicita el servicio a través del Proxy y es éste quién se comunica con el RealSubject.



## Patrón Estructural – Proxy

### Ventajas y desventajas de este patrón:

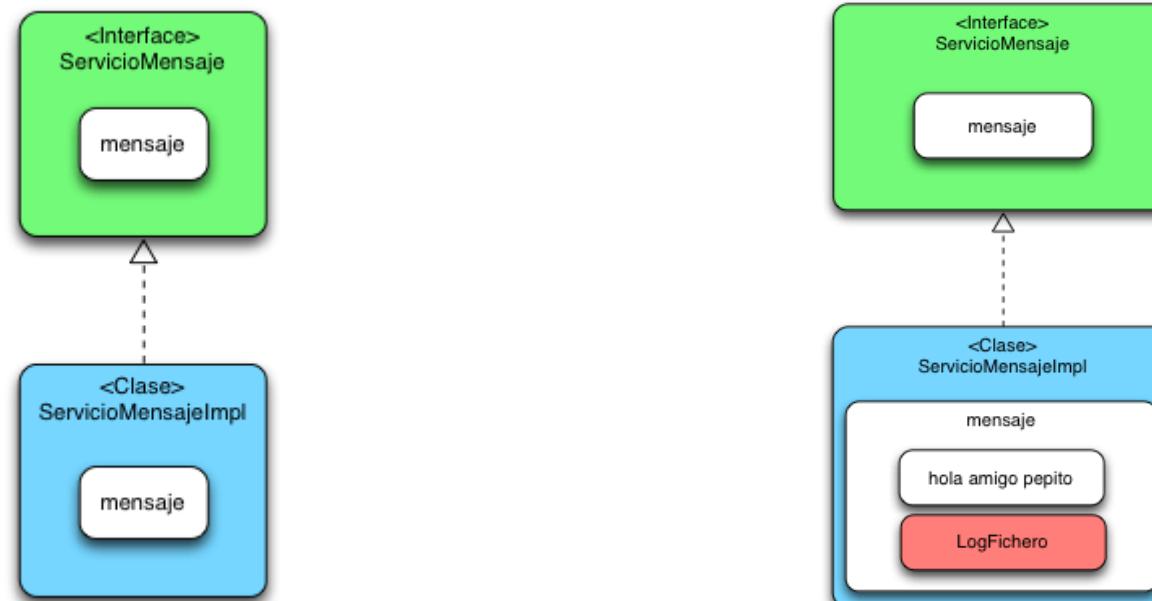
- Podemos modificar funcionalidad sin afectar a los clientes.
- El código se puede volver complicado si es necesario introducir gran cantidad de funcionalidad nueva.
- La respuesta real del servicio puede ser desconocida hasta que realmente se la invoca.



## Patrón Estructural – Proxy

### Ejemplo:

Vamos a suponer que tenemos una clase de servicio con un método que nos devuelve el mensaje “**hola amigo pepito**”, recibiendo como parámetro el nombre que deseemos. Para ello construiremos una interface y una clase que la implemente.



## Patrón Estructural – Proxy

### Ejemplo:

```
1 package com.arquitecturajava.proxy;
2
3 public interface ServicioMensaje {
4     public String mensaje(String persona);
5 }
6
7
8 package com.arquitecturajava.proxy2;
9
10 public class ServicioMensajeImpl implements ServicioMensaje {
11     public String mensaje(String persona)
12     {
13         return "hola amigo "+ persona;
14     }
15 }
```

```
1 package com.arquitecturajava.proxy;
2
3 public class Principal {
4
5     public static void main(String[] args) {
6
7         ServicioMensaje sm= new ServicioMensajeImpl();
8         System.out.println(sm.mensaje("pepito"));
9
10    }
11
12 }
```

## Patrón Estructural – Proxy

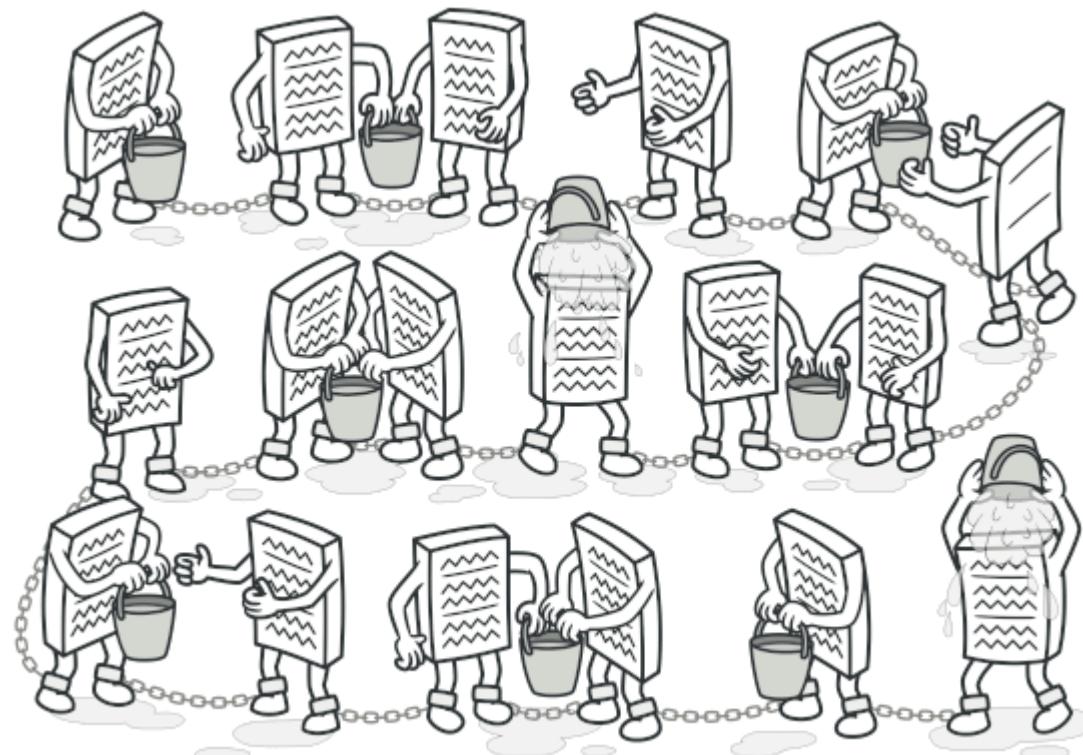
Ejemplo:

```
1 package com.arquitecturajava.proxy2;
2
3 public class ServicioMensajeProxy implements ServicioMensaje {
4     private ServicioMensaje sm;
5
6     @Override
7     public String mensaje(String persona) {
8         System.out.println("log del mensaje para"+ persona);
9         //mensaje delegado
10        return sm.mensaje(persona);
11    }
12    public ServicioMensajeProxy() {
13        super();
14        this.sm = new ServicioMensajeImpl();
15    }
16
17 }
18 }
```

```
1 package com.arquitecturajava.proxy2;
2
3 public class Principal {
4
5     public static void main(String[] args) {
6
7         ServicioMensaje sm= new ServicioMensajeProxy();
8         System.out.println(sm.mensaje("pepito"));
9
10    }
11
12 }
```

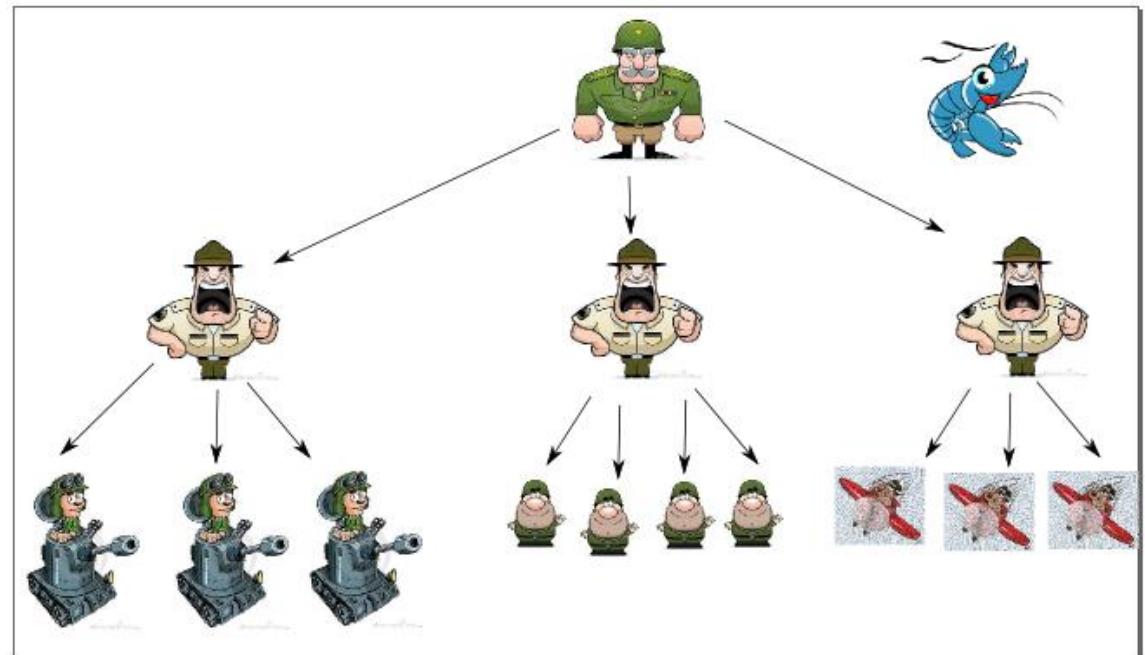
# Cadena de Responsabilidades

---



## Patrón de comportamiento – cadena de responsabilidades

- El patrón de diseño Cadena de responsabilidades permite establecer una cadena de objetos receptores a través de los cuales se pasa una petición formulada por un objeto emisor. Cualquiera de los objetos receptores puede responder a la petición en función de un criterio establecido.

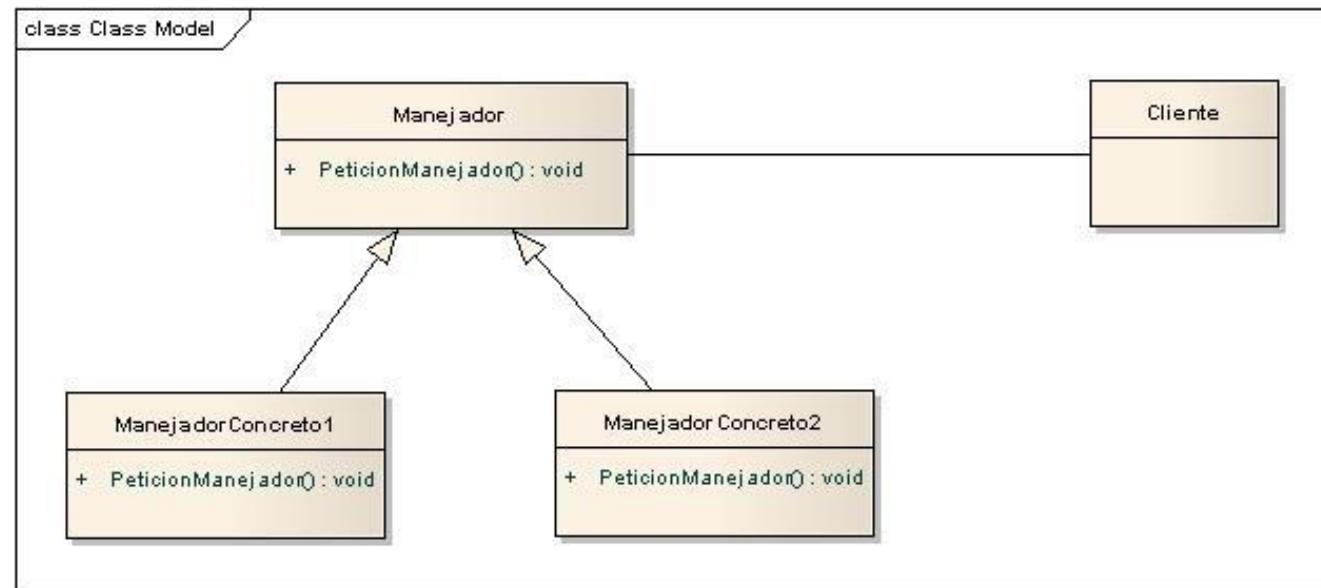


## Participantes

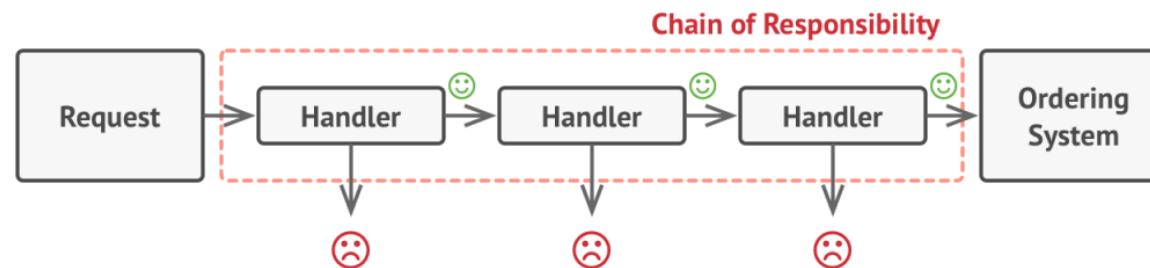
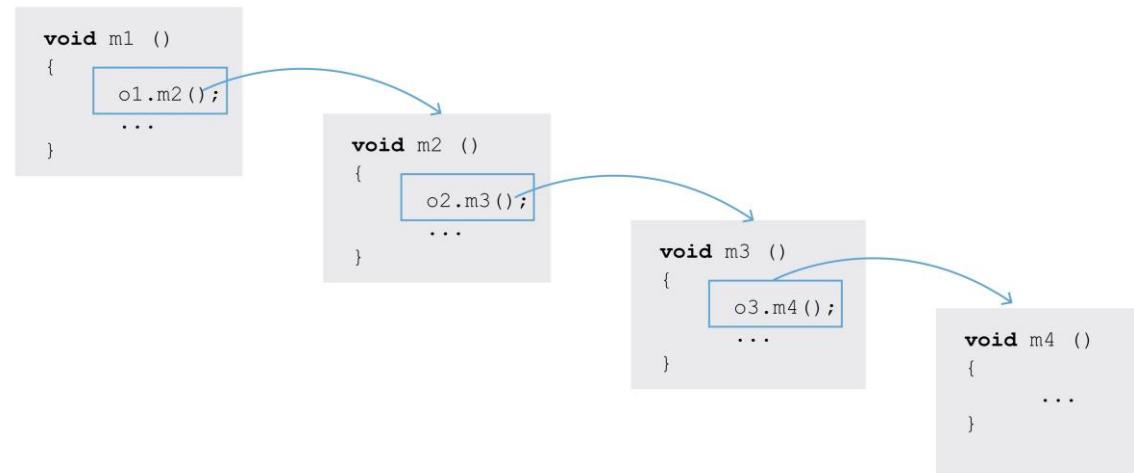
**Cliente:** Inicia la petición que llega a la cadena en busca del responsable.

**Manejador:** Define una interfaz para manejar peticiones.

**ManejadorConcreto:** Define las responsabilidades de cada componente. Si puede manejar una petición, la procesa, en caso contrario busca al siguiente.

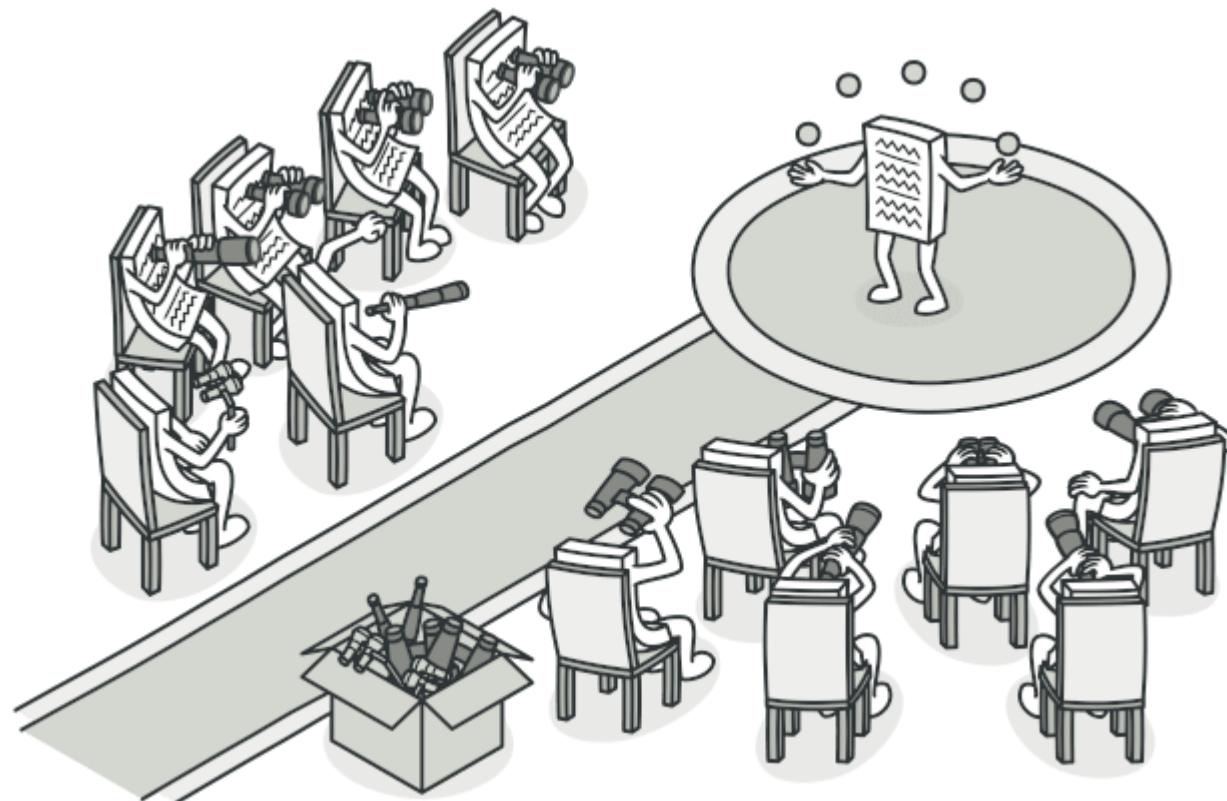


Cuando un Cliente realiza una petición, ésta se propaga por el Manejador hasta que un Manejador Concreto asume la responsabilidad de procesarla.



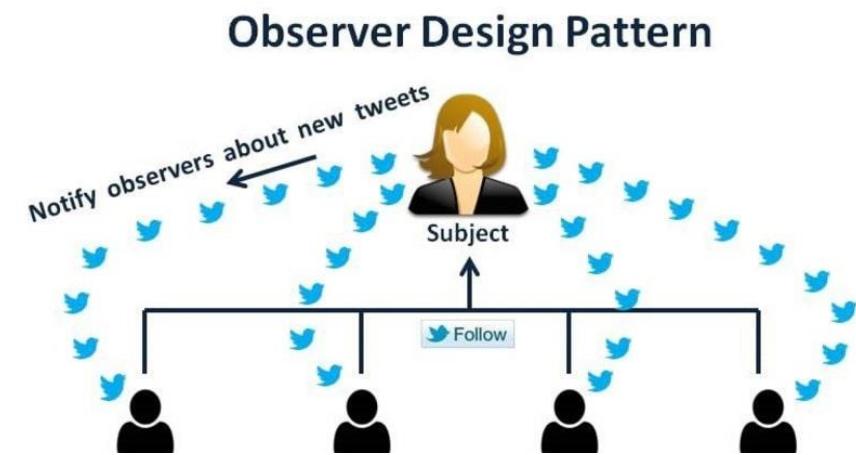
# Patrón Observador

---

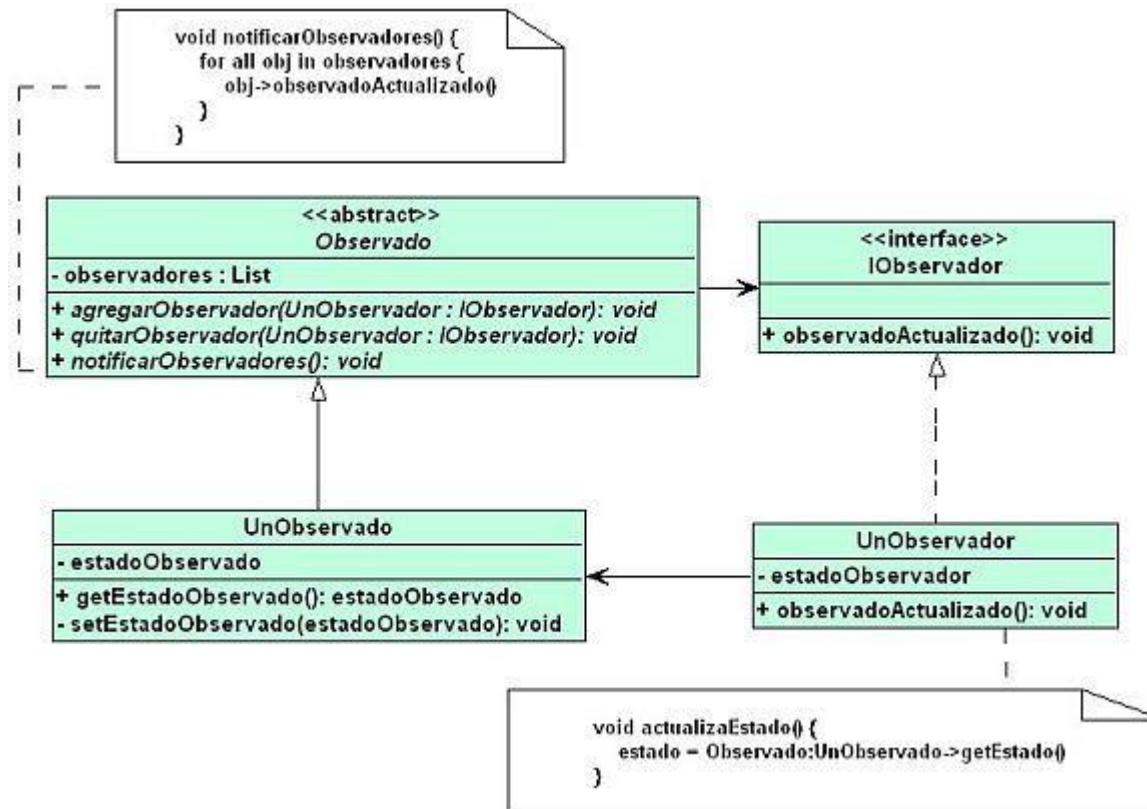


## Patrón observador

- Observer es un patrón de diseño de comportamiento que permite a un objeto notificar a otros objetos sobre cambios en su estado.
- El patrón Observer proporciona una forma de suscribirse y cancelar la suscripción a estos eventos para cualquier objeto que implementa una interfaz suscriptora



El patrón Observer puede ser utilizado cuando hay objetos que dependen de otro, necesitando ser notificados en caso de que se produzca algún cambio en él



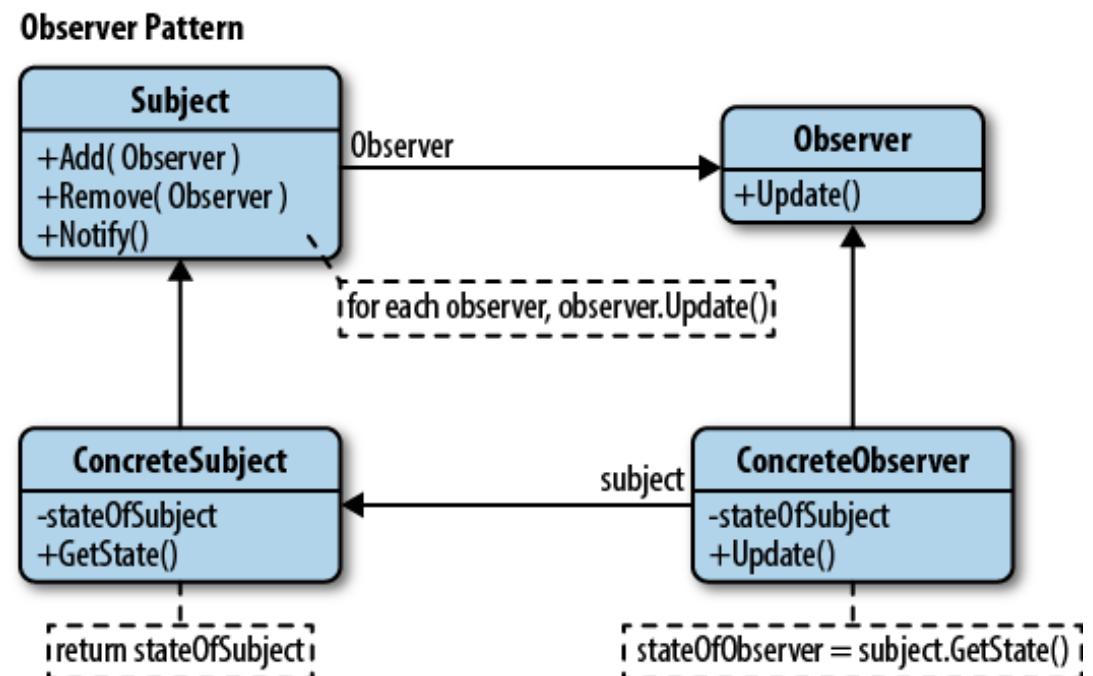
## Participantes

**Sujeto (subject):** El sujeto proporciona una interfaz para agregar (attach) y eliminar (detach) observadores. El Sujeto conoce a todos sus observadores.

**Observador (observer):** Define el método que usa el sujeto para notificar cambios en su estado (update/notify).

**Sujeto concreto (concrete subject):** Mantiene el estado de interés para los observadores concretos y los notifica cuando cambia su estado.

**Observador concreto (concrete observer):** Mantiene una referencia al sujeto concreto e implementa la interfaz de actualización.



```
public class Vuelo {  
    private String codigoDestino;  
    private List<Viajero> viajeros;  
    private String ultimoSuceso;  
  
    public Vuelo(String codigoDestino) {  
        super();  
        this.codigoDestino = codigoDestino;  
        viajeros = new ArrayList<Viajero>();  
        ultimoSuceso = "";  
    }  
    ....
```

```
public class Viajero {  
    private String nombre;  
    private Vuelo vuelo;  
  
    public Viajero(String nombre, Vuelo vuelo) {  
        super();  
        this.nombre = nombre;  
        this.vuelo = vuelo;  
    }  
    ....
```

## Patrones

```
public void suscribirObservador(Viajero viajero) {  
    viajeros.add(viajero);  
}
```

```
public void eliminarObservador(Viajero viajero) {  
    viajeros.remove(viajero);  
}
```

```
public void notificarObservadores() {  
    for(Viajero viajero: viajeros)  
        viajero.notificar();  
}
```

```
public String getUltimoSuceso() {  
    return codigoDestino + ":" + ultimoSuceso;  
}
```

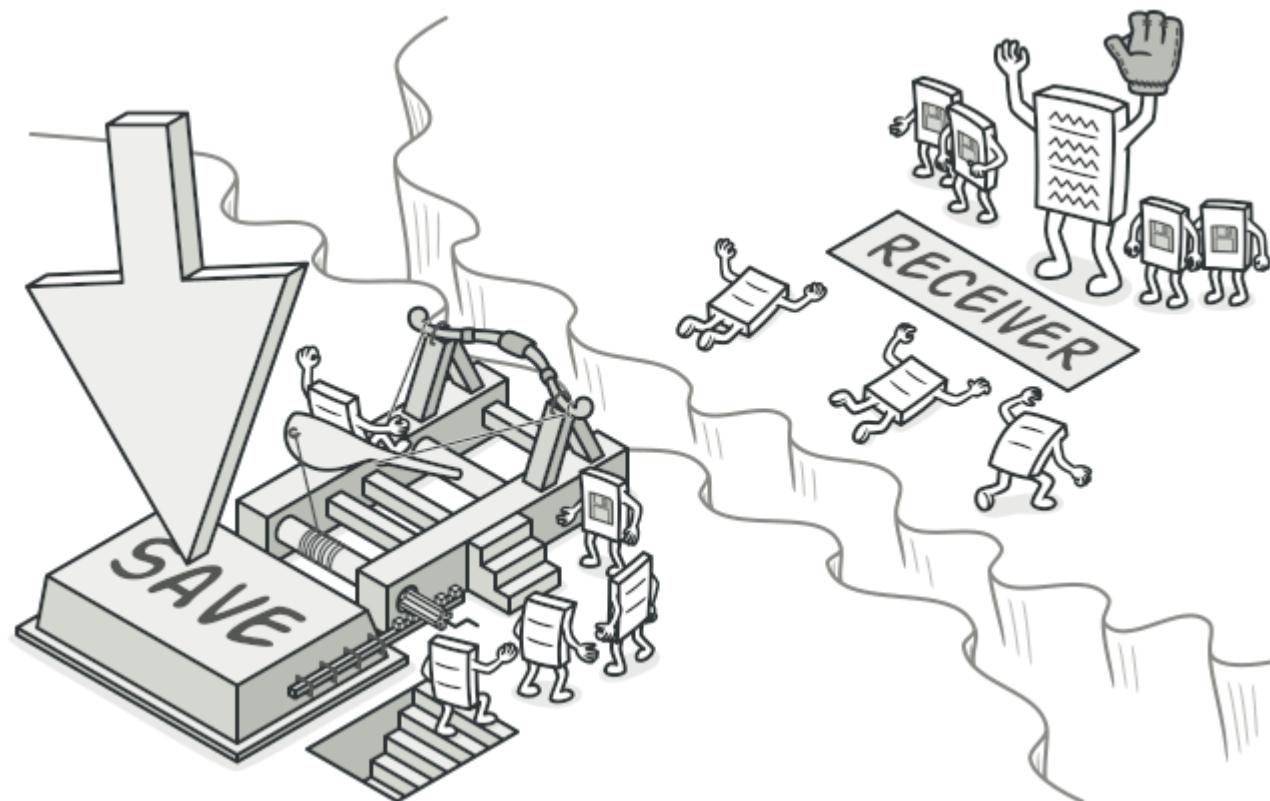
```
public void setUltimoSuceso(String ultimoSuceso) {  
    this.ultimoSuceso = ultimoSuceso;  
    notificarObservadores();  
}
```

```
public void notificar() {  
    System.out.println(nombre + "<-- Notificar: " +  
        vuelo.getUltimoSuceso());  
}
```

```
Vuelo vuelo = new Vuelo("IB123 destino París");  
Viajero oscar = new Viajero("Oscar", vuelo);  
vuelo.suscribirObservador(oscar);  
Viajero pepe = new Viajero("Pepe", vuelo);  
vuelo.suscribirObservador(pepe);  
vuelo.setUltimoSuceso("Llegada del vuelo.");  
vuelo.eliminarObservador(pepe);  
vuelo.setUltimoSuceso("Salida de viajeros");
```

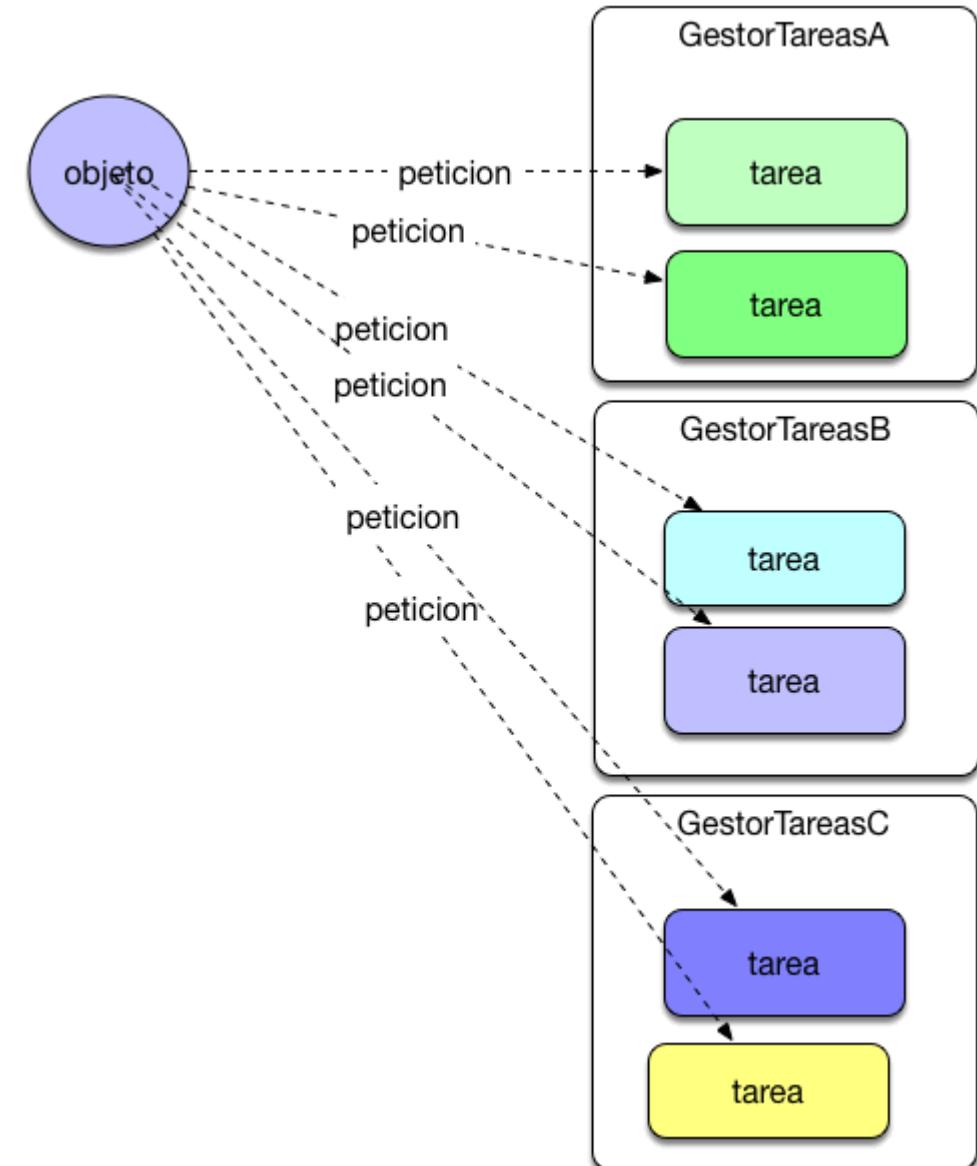
# Patrón Command

---



## Patrones

El patrón de diseño **Command** es muy utilizado cuando se requiere hacer ejecuciones de operaciones sin conocer realmente lo que hacen, estas operaciones son conocidas como comandos y son implementadas como una clase independiente que realiza una acción muy concreta, para lo cual únicamente recibe un conjunto de parámetros para realizar su tarea.



## Participantes

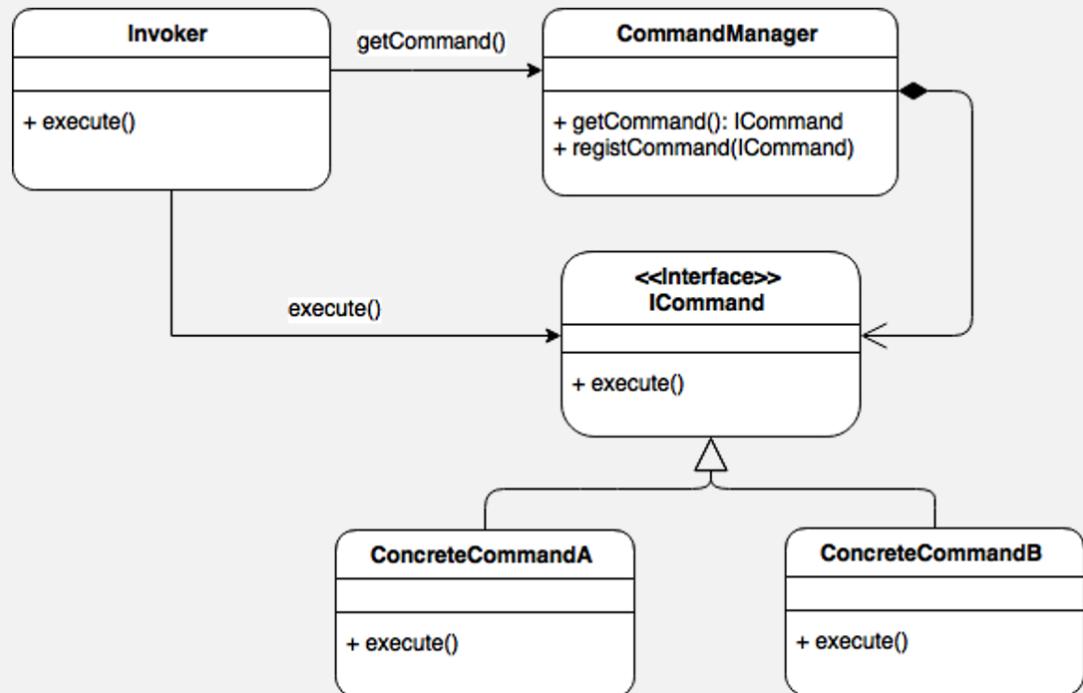
**AbstractCommand:** Clase que ofrece una interfaz para la ejecución de órdenes.

**ConcreteCommand:** Clase que implementa una orden concreta y sus métodos.

**Invoker:** Clase que instancia las órdenes, puede a su vez ejecutarlas inmediatamente.

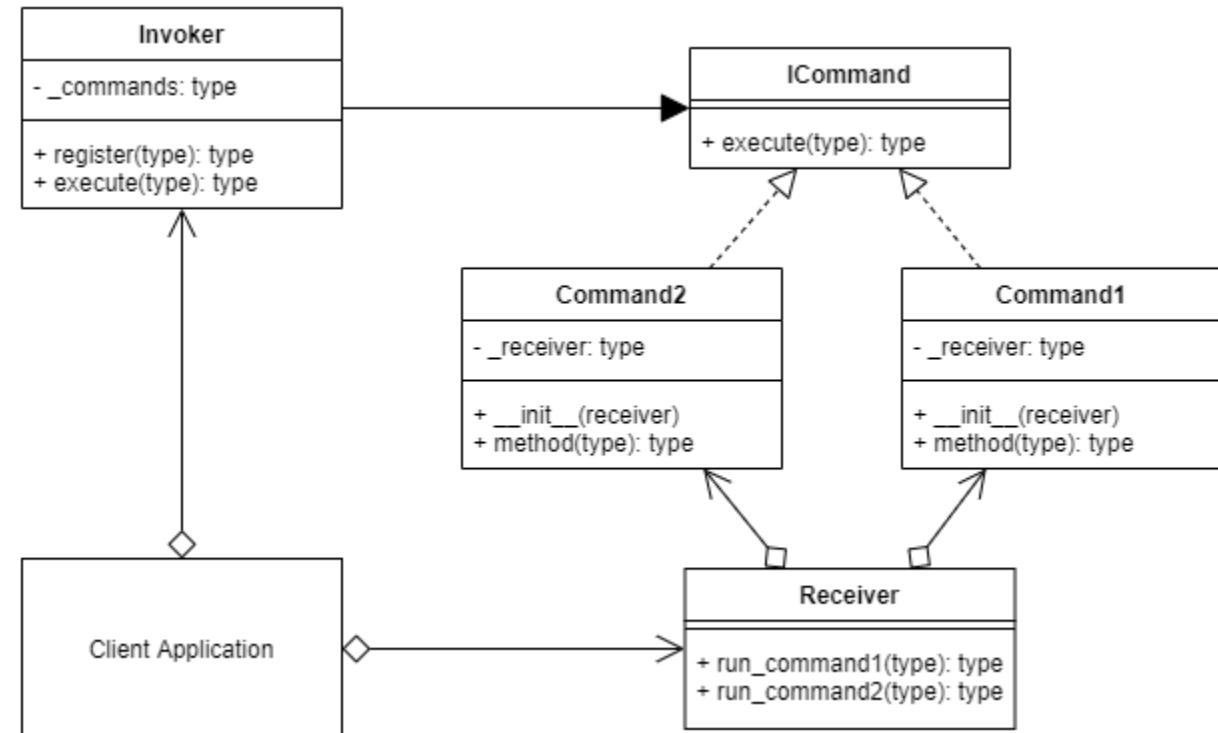
**CommandManager:** Responsable de gestionar una colección de objetos orden creadas por el Invoker.

### Command pattern – Class diagram



## Consecuencias

- Se independiza la parte de la aplicación que invoca las órdenes de la implementación de los mismos.
- Al tratarse las órdenes como objetos, se puede realizar herencia de las mismas, composiciones de órdenes (mediante el patrón Composite).
- Se facilita la ampliación del conjunto de órdenes.

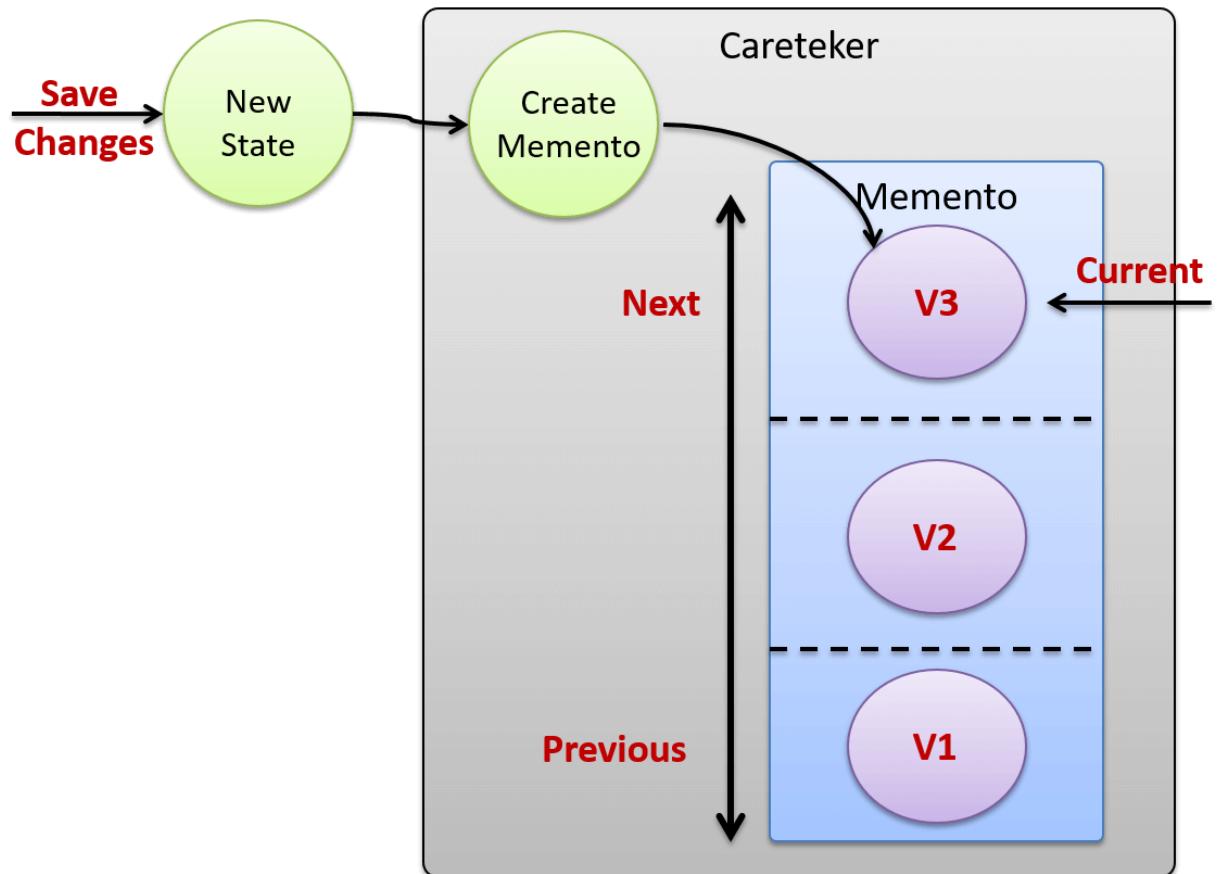


# Patrón Memento

---

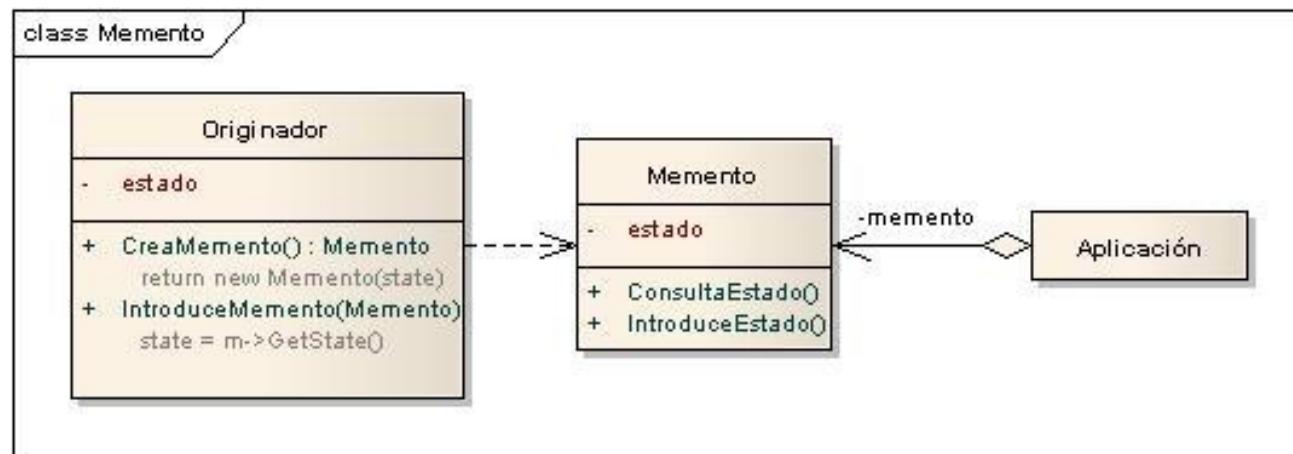


Memento, es un patrón de diseño cuya finalidad es almacenar el estado de un objeto (o del sistema completo) en un momento dado de manera que se pueda restaurar en ese punto de manera sencilla. Para ello se mantiene almacenado el estado del objeto para un instante de tiempo en una clase independiente de aquella a la que pertenece el objeto (pero sin romper la encapsulación), de forma que ese recuerdo permita que el objeto sea modificado y pueda volver a su estado anterior.



## Participantes

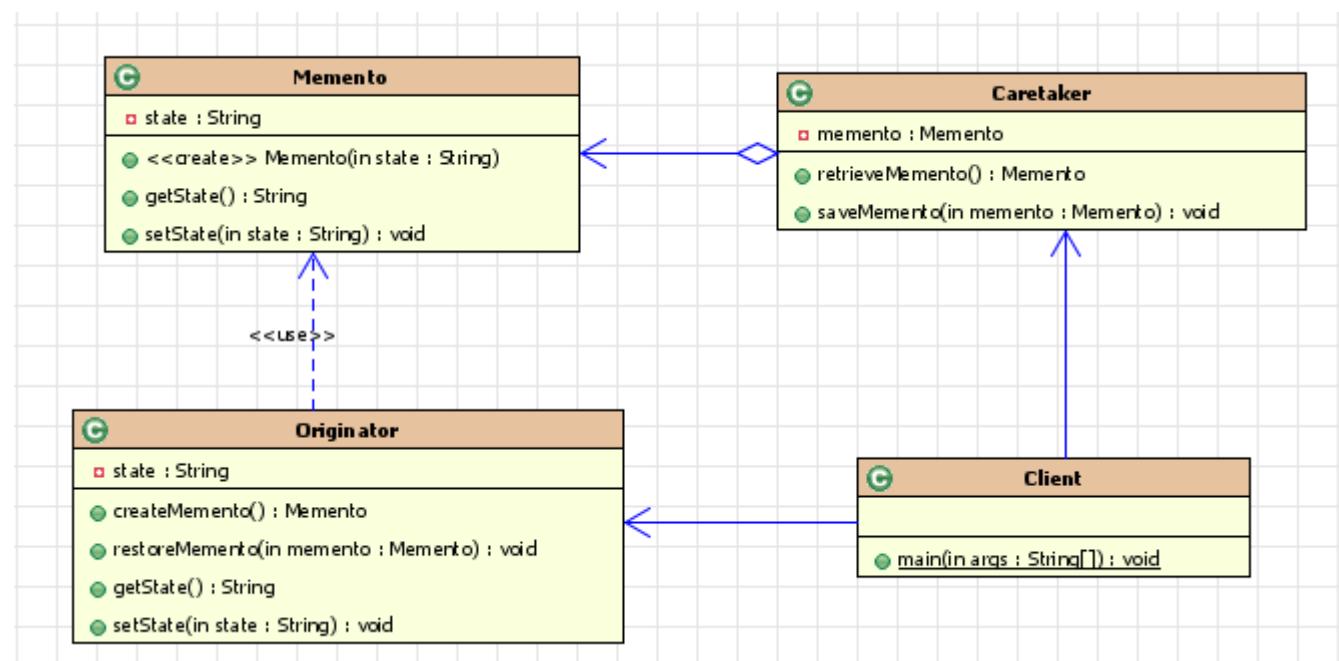
- **Memento:** Almacena el estado de un objeto Originador. Memento almacena todo o parte de Originador. Tiene dos interfaces, una para Aplicación que le permite comunicarse con otros objetos y otra para Originador que le permite almacenar el estado.
- **Originador:** Crea un objeto Memento con una copia de su estado. Usa Memento para restaurar el estado almacenado.
- **Aplicación:** Mantiene a Memento pero no opera con su contenido.



## Implementación:

Se deben seguir las siguientes recomendaciones para la implementación del patrón

- Memento crea una interfaz solo accesible por Originador.
- Almacena los cambios incrementales. Todos los cambios de estado pueden almacenarse en Mementos.



## *Memento pattern – Diagram of sequence*

