PROGRA 102

Paolo Sandoval Noel

Gmail: paolosandovaln@gmail.com

Email: paolo.sandoval@jalasoft.com

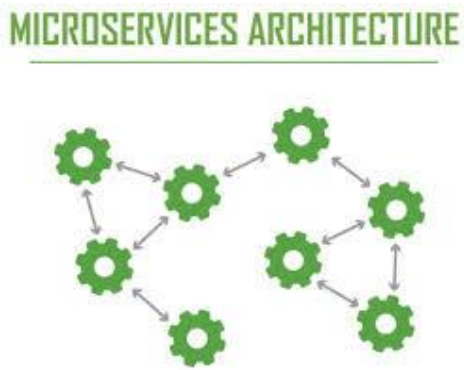
Skype: paolo.sandoval.jala

Github: jpsandovaln

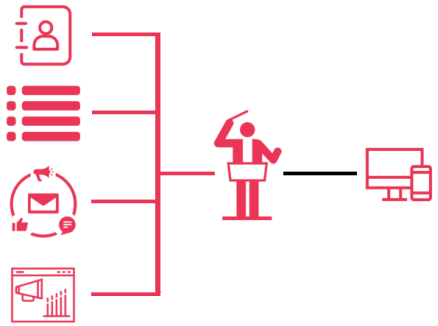

JavaScript


TypeScript

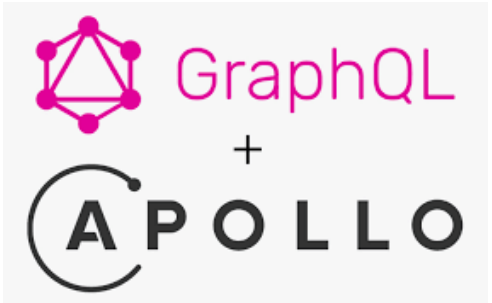

ex
M E R N


MICROSERVICES ARCHITECTURE

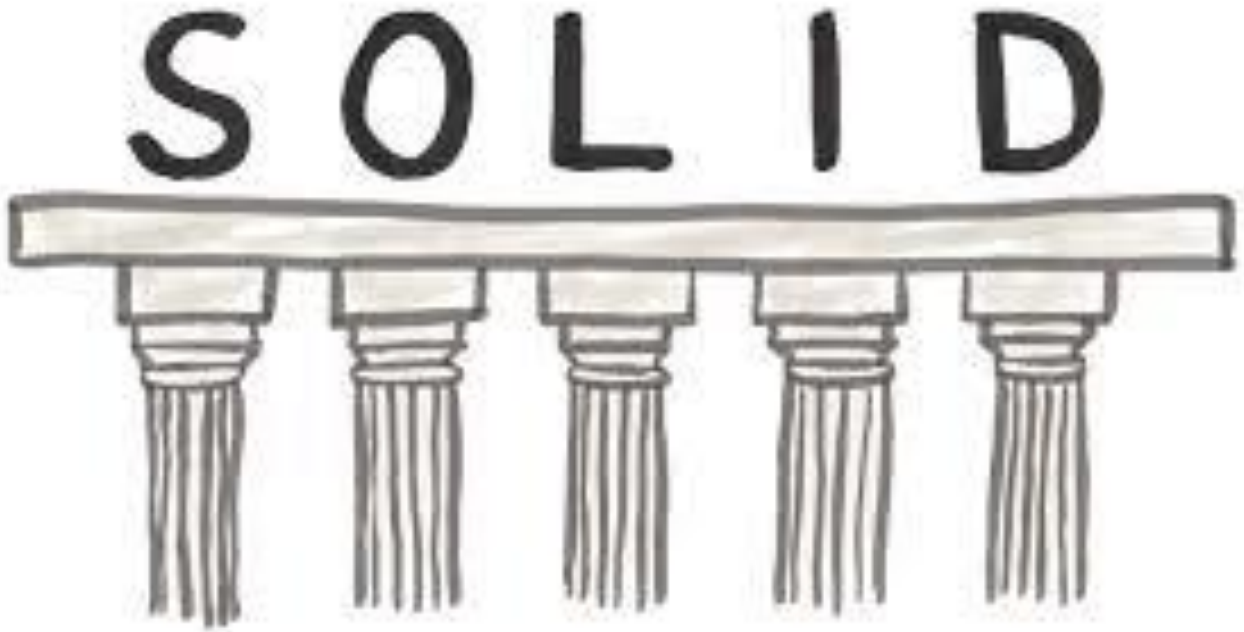

Design
Patterns




docker


GraphQL
+
APOLLO

Principios



S.O.L.I.D

- 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.

S.O.L.I.D

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

S
- **Single responsibility principle** o Principio de responsabilidad única.

O
- **Open/Close principle** o Principio de abierto/cerrado.

L
- **Liskov substitution principle** o Principio de sustitución de Liskov.

I
- **Interface segregation principle** o Principio de segregación de la interfaz.

D
- **Dependency inversión principle** o Principio de inversión de dependencia

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){ ... }
}
```

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")
};
```

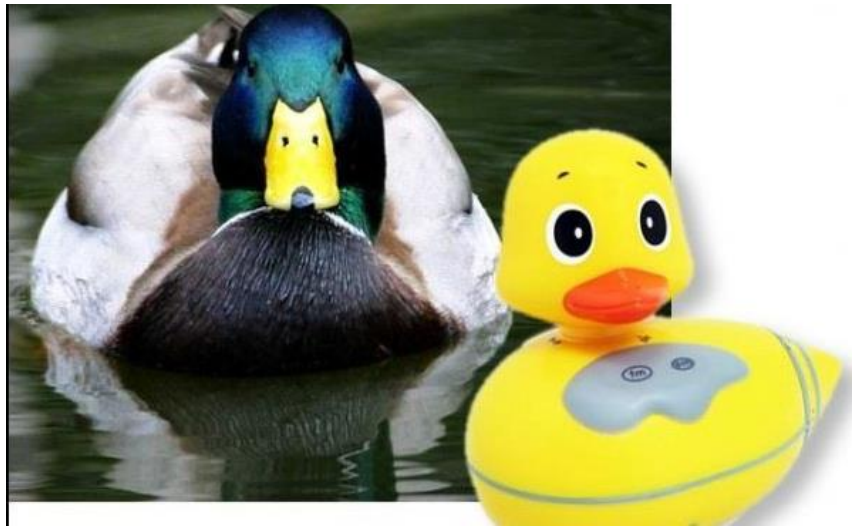
```
public static void imprimirPrecioMedioCoche(Coche[] arrayCoches){
    for (Coche coche : arrayCoches) {
        if(coche.marca.equals("Renault")) System.out.println(18000);
        if(coche.marca.equals("Audi")) System.out.println(25000);
        if(coche.marca.equals("Mercedes")) System.out.println(27000);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

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 substitución 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:

```java
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() {
        //..
    }
}
```

S.O.L.I.D

I: Principio de segregación de interfaz:

```java
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() {
        //...
    }
```

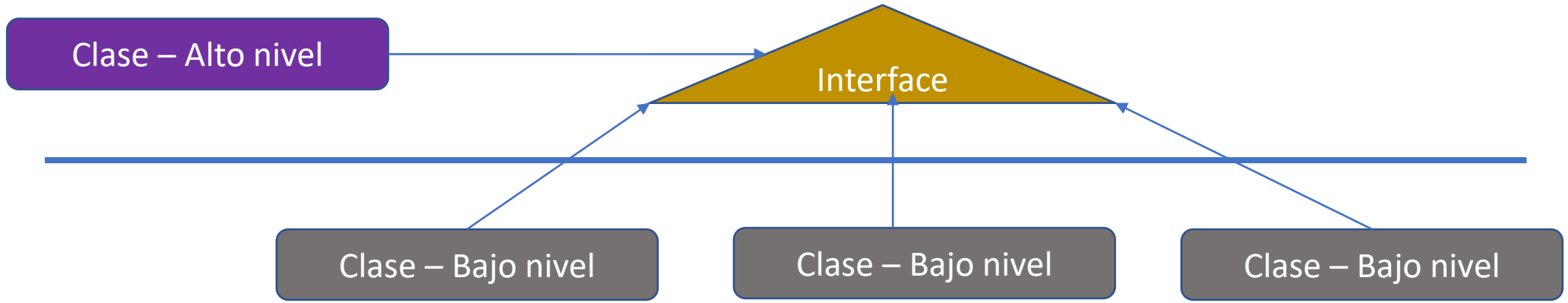
I: Principio de segregación de interfaz:

```java
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() {
        //...
```

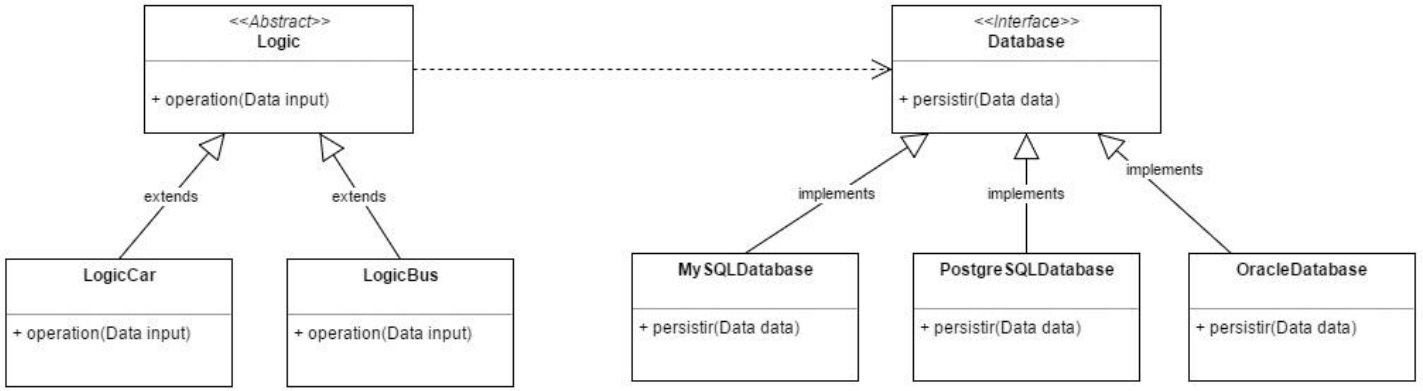
D: Principio de inversión de dependencias:

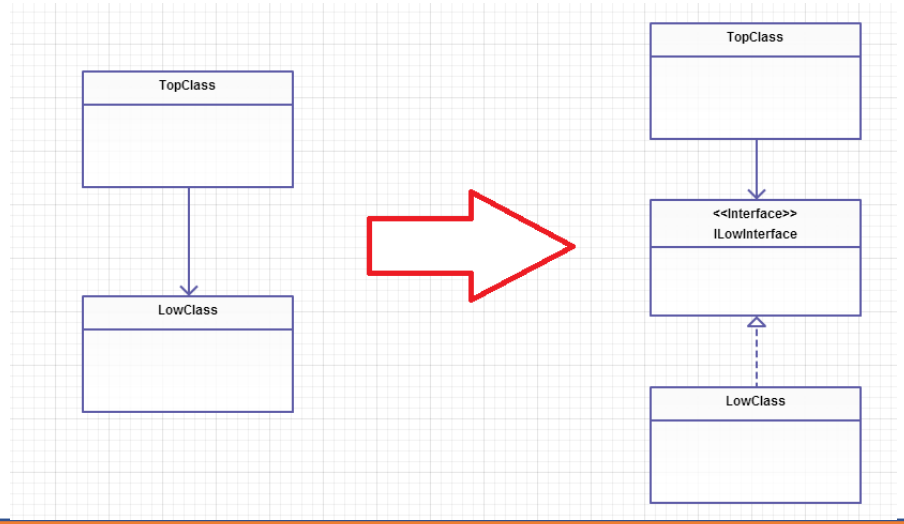
Establece que las dependencias deben estar en las abstracciones, no en las concreciones. Es decir:

- 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:





S.O.L.I.D

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

Patrones de Diseño

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:

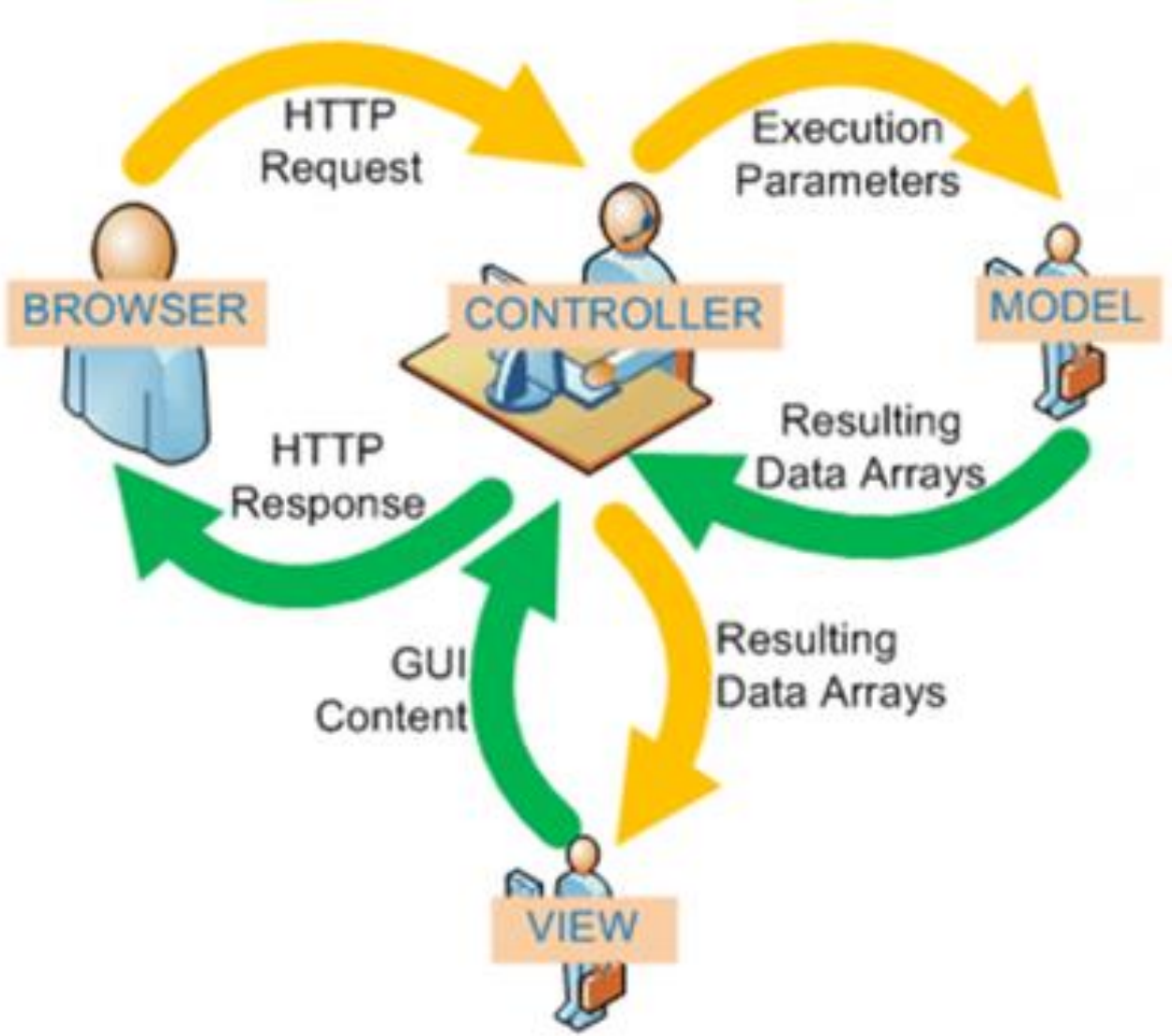
- **Patrones de arquitectura:** Aquellos que expresan un esquema organizativo estructural fundamental para sistemas de software.
- **Patrones de diseño**: Aquellos que expresan esquemas para definir estructuras de diseño (o sus relaciones) con las que construir sistemas de software.
- **Dialectos**: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

Patrones de arquitectura:

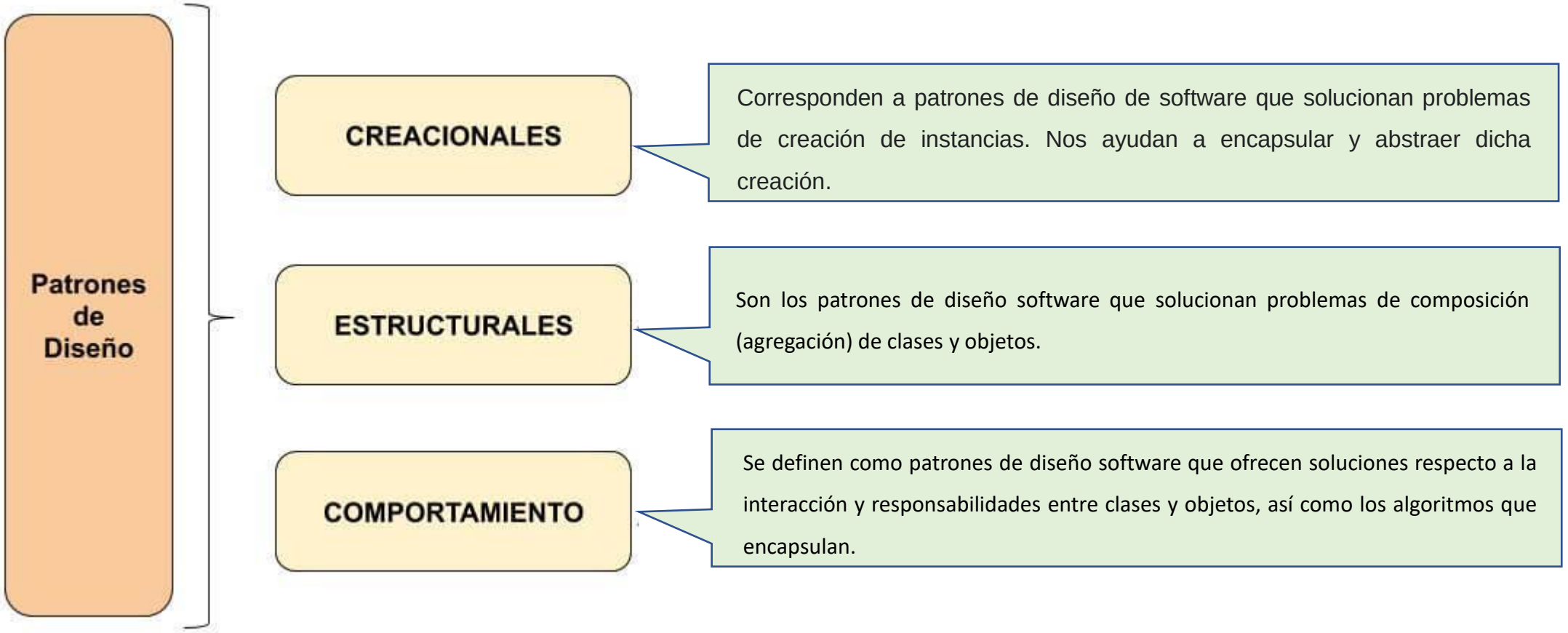
Patrones de Arquitectura

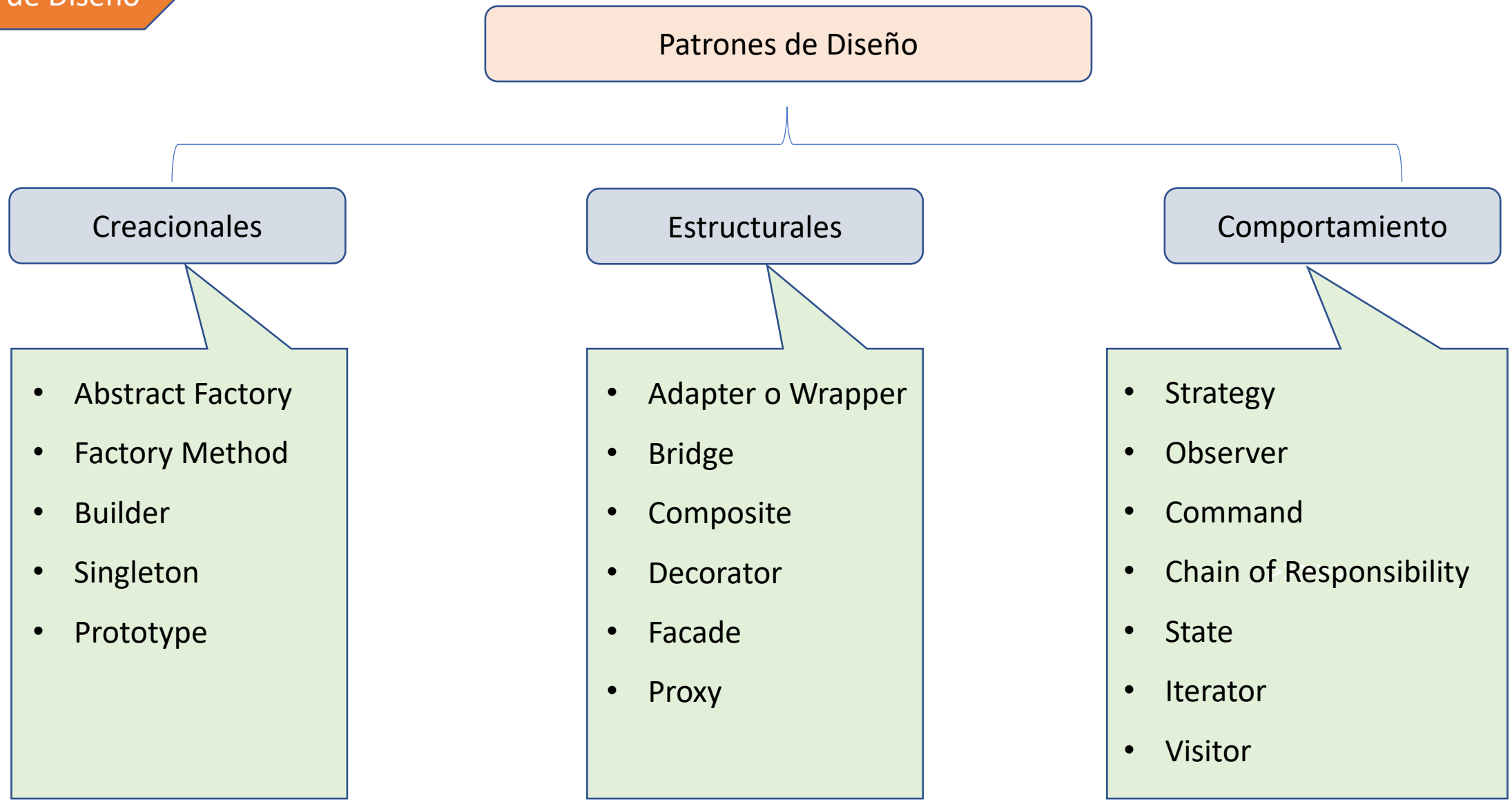
- Programación por capas.
- Tres niveles.
- Arquitectura de microservicios.
- Arquitectura de microkernel.
- Pipeline.
- Invocación implícita.
- Arquitectura orientada a servicios
- **Modelo vista controlador**
- Peer to peer.

Modelo Vista Controlador:

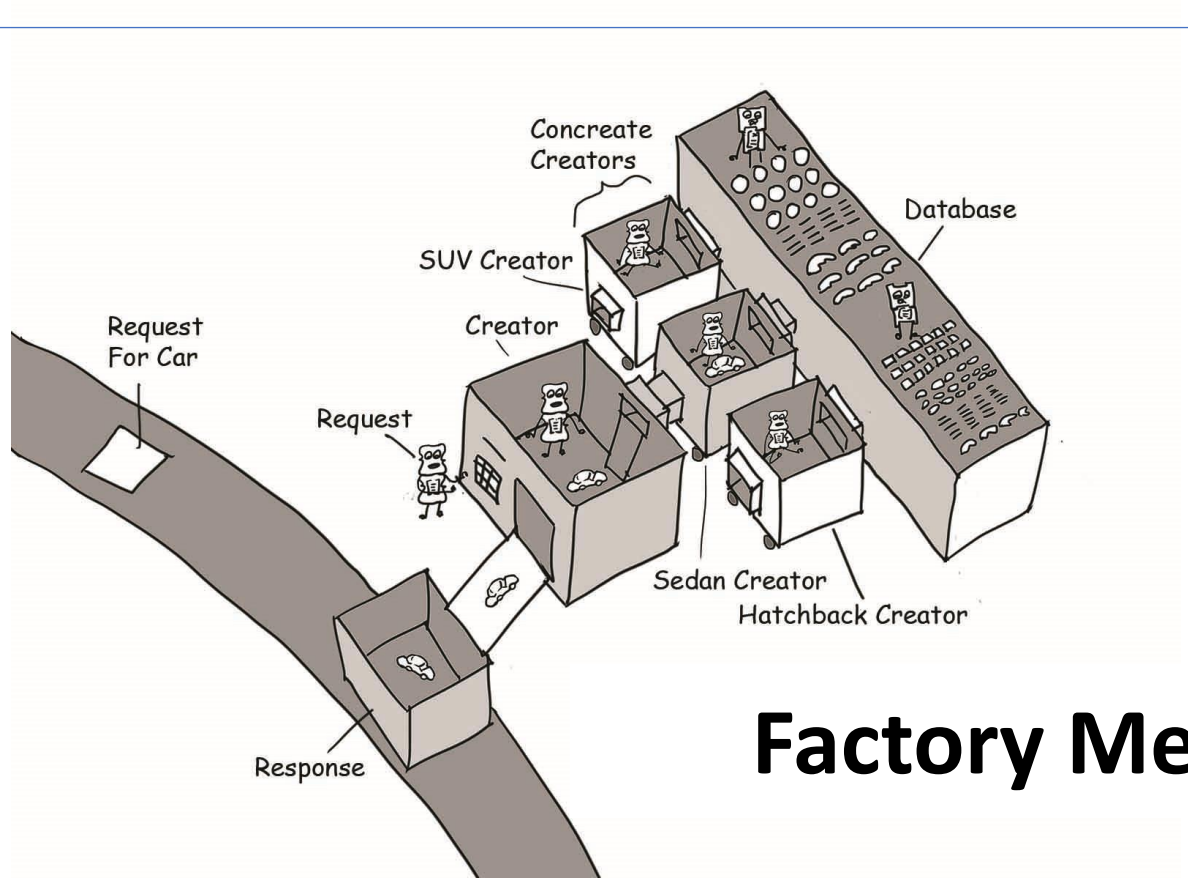


Tipos de Patrones de Diseño:




Patrones de Diseño
Creacionales
Estructurales
Comportamiento
Abstract Factory
Factory Method
Builder
Singleton
Prototype
Adapter o Wrapper
Bridge
Composite
Decorator
Facade
Proxy
Strategy
Observer
Command
Chain of Responsibility
State
Iterator
Visitor

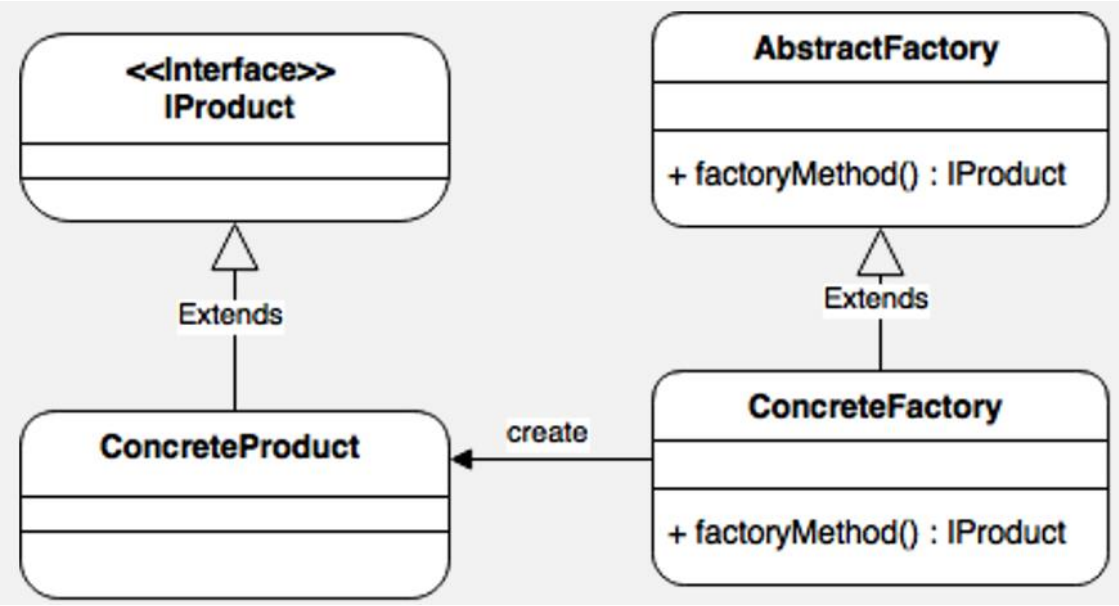
Patrones



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.

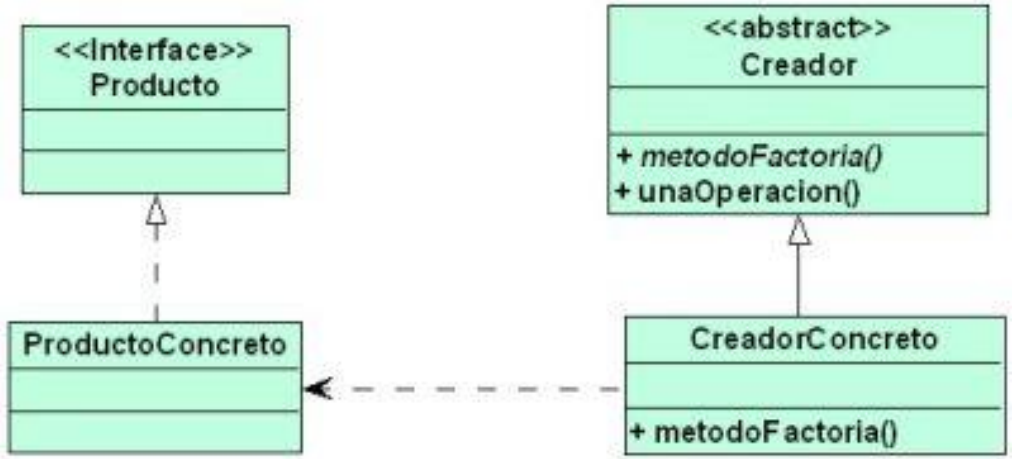


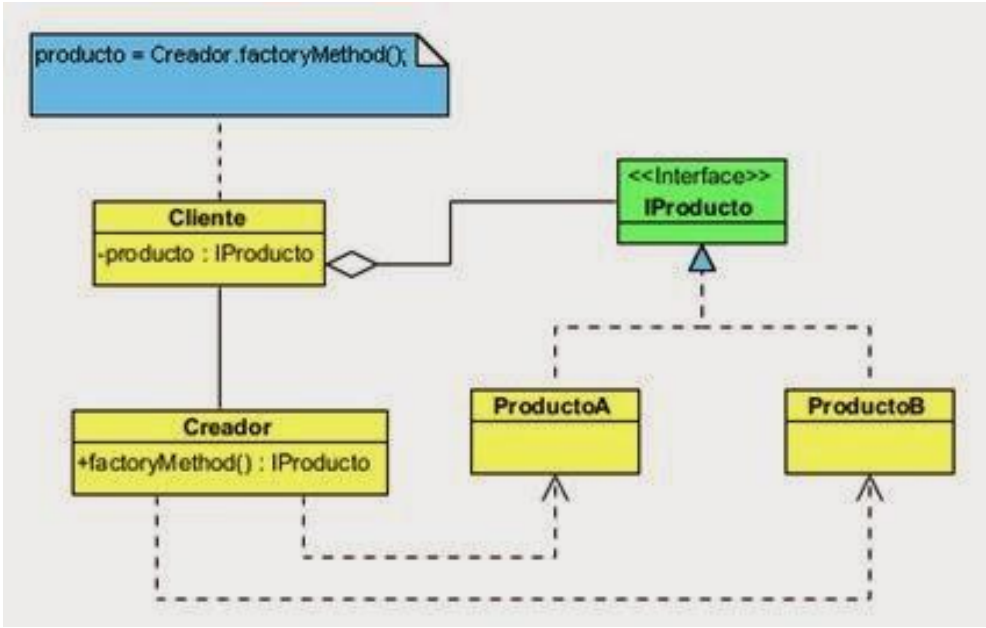
Patrón de creación – Factory method

Participantes:

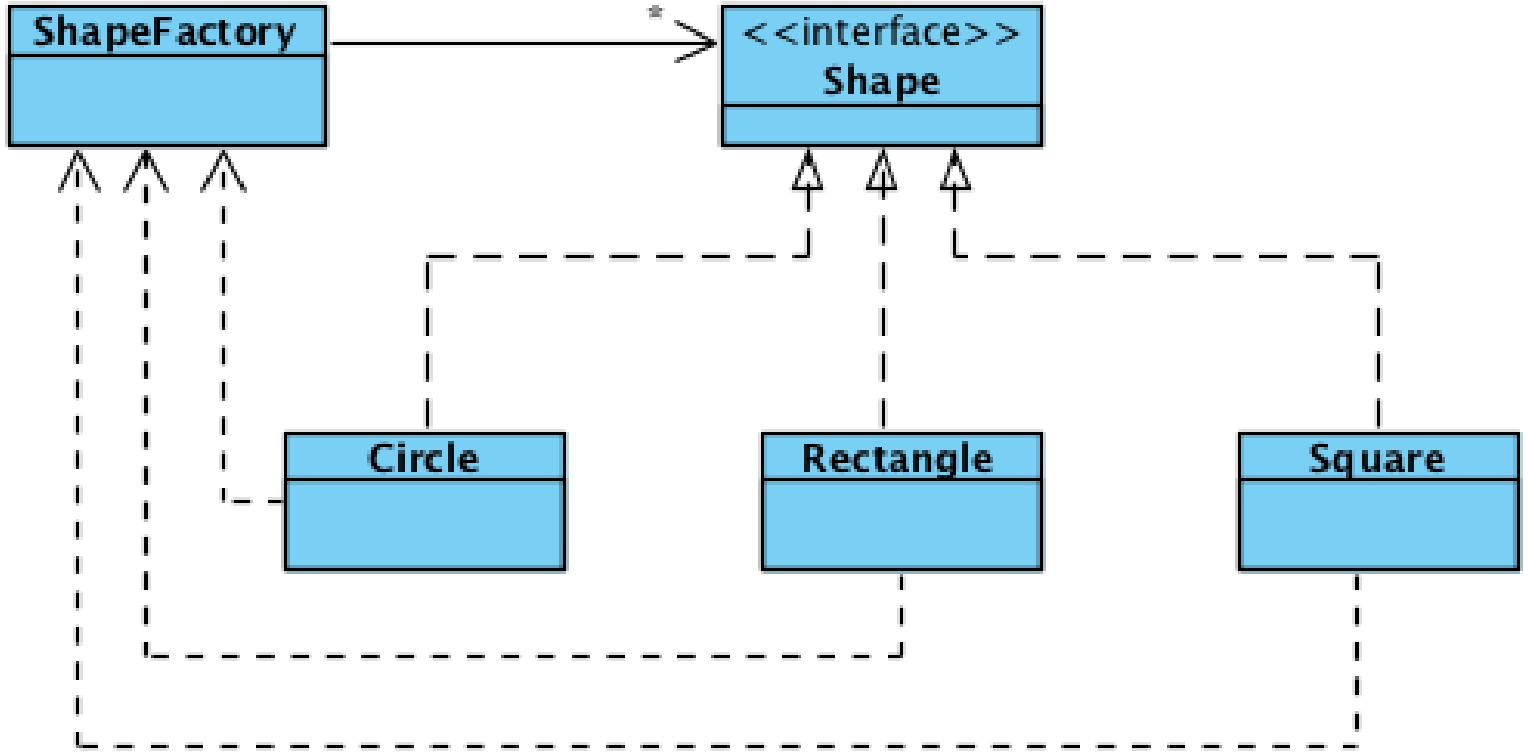
- **Producto:** Define la interfaz de los objetos que crea el método de fabricación.
- **ProductoConcreto:** Implementa la interfaz Producto.
- **Creador:** Declara el método de fabricación, el cual devuelve un objeto del tipo Producto. También puede definir una implementación predeterminada del método 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

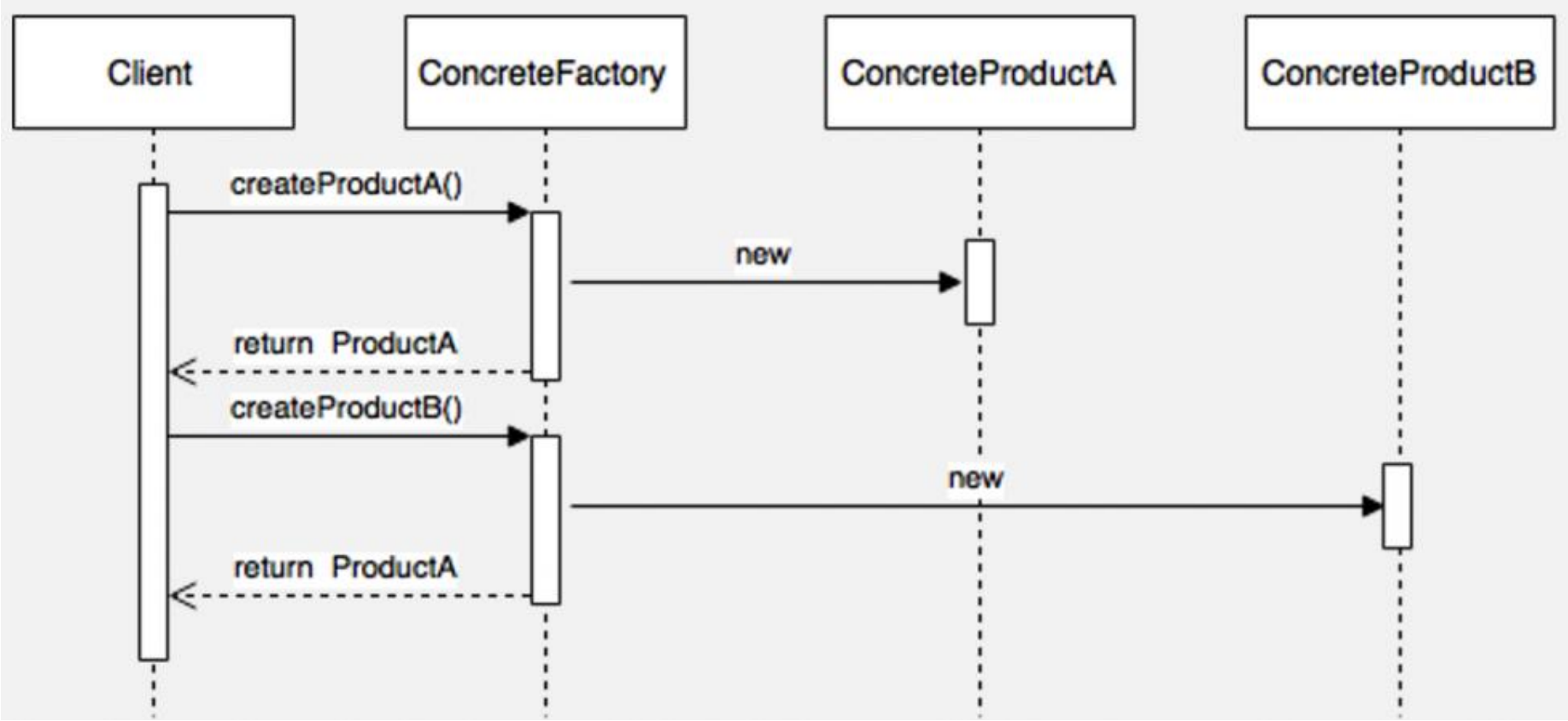




Patrón de creación – Factory method



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

```java
package FactoryMethod2;

public class ArchivoVideo implements IArchivo
{
    public ArchivoVideo() {
    }

    // ----------------------------

    @Override
    public void reproducir() {
        System.out.println("Reproduciendo archivo de vídeo...");
    }
}
```

```java
package FactoryMethod1;

public abstract class CreadorAbstracto
{
    public static final int AUDIO = 1;
    public static final int VIDEO = 2;

    // --------------------------------

    public abstract IArchivo crear(int tipo);
}
```

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.  }
```

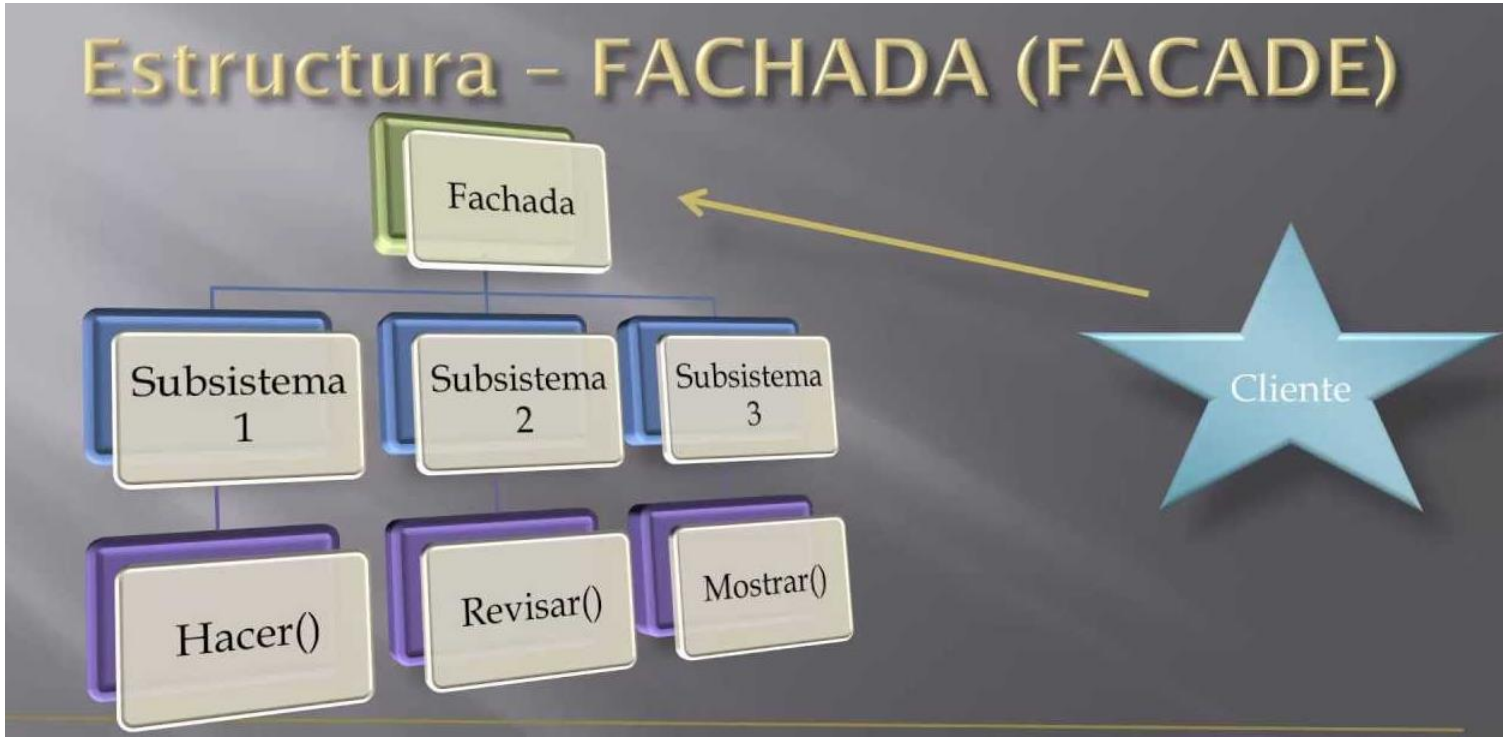
```
01.  package FactoryMethod2;
02.
03.  public class Creador
04.  {
05.      public static final int AUDIO = 1;
06.      public static final int VIDEO = 2;
07.
08.      // --------------------------------------
09.
10.      public Creador() {
11.      }
12.
13.      // --------------------------------------
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.  }
```

Patrón de creación – Factory method

```
interface Creator {
  factoryMethod(param: 1 | 2 | 3): Product;
}

class ConcreteCreator implements Creator {
  factoryMethod(param: 1 | 2 | 3): Product {
    const dictionary = {
      1: ConcreteProduct1,
      2: ConcreteProduct2,
      3: ConcreteProduct3
    }
    const ConcreteProduct = dictionary[param];
    return new ConcreteProduct();
  }
}

export class Client {
  createProduct(): Product {
    const creator = new ConcreteCreator();
    return creator.factoryMethod(1);
  }
}
```


```
interface Creator {
  factoryMethod(param: 1 | 2 | 3): Product;
}

class ConcreteCreator implements Creator {
  factoryMethod(param: 1 | 2 | 3): Product {
    const dictionary = {
      1: ConcreteProduct1,
      2: ConcreteProduct2,
      3: ConcreteProduct3
    }
    const ConcreteProduct = dictionary[param];
    return new ConcreteProduct();
  }
}

export class Client {
  createProduct(): Product {
    const creator = new ConcreteCreator();
    return creator.factoryMethod(1);
  }
}
```

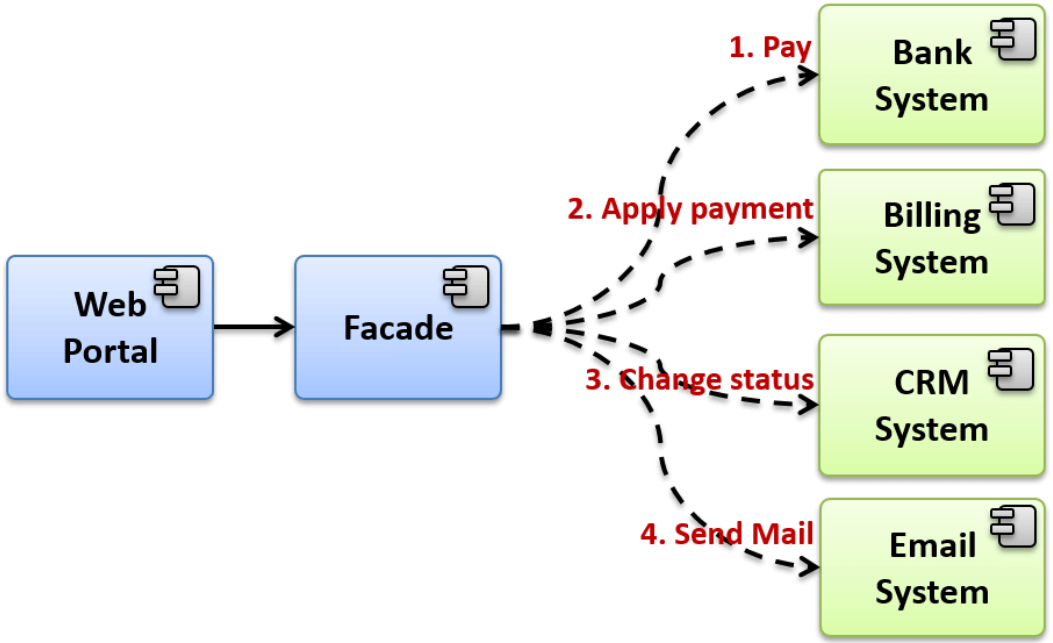
Patrones


Estructura – FACHADA (FACADE)
Fachada
Subsistema 1
Subsistema 2
Subsistema 3
Cliente
Hacer()
Revisar()
Mostrar()

Patrón Estructural – Fachada

El patrón de diseño Facade simplifica la complejidad de un sistema mediante una interfaz mas 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.

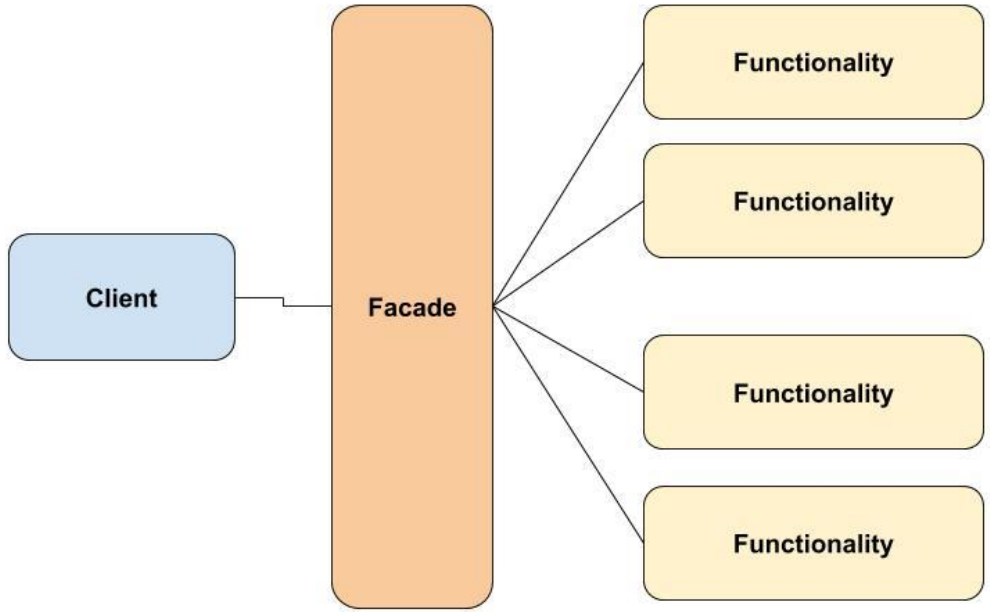
Se trata de un patrón de diseño bastante útil en vista de que también desacopla los sistemas .



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.



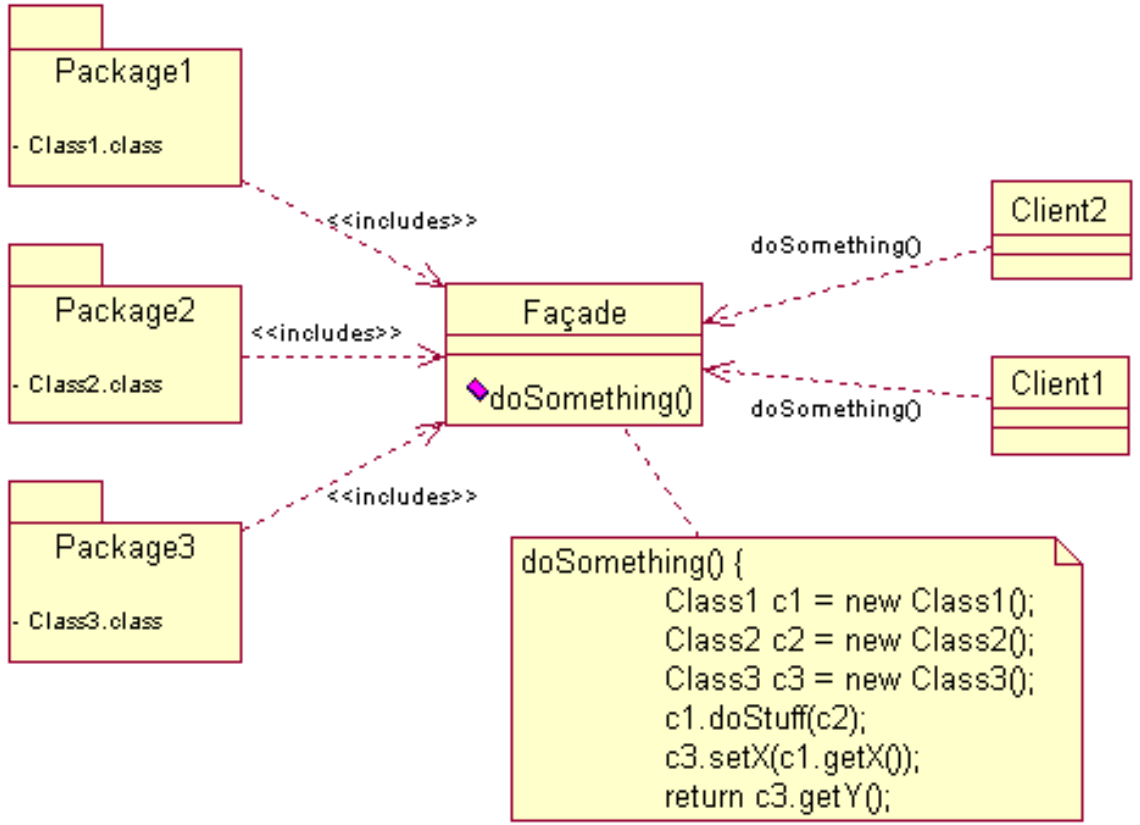
Patrón Estructural – Fachada

Ventajas e inconvenientes

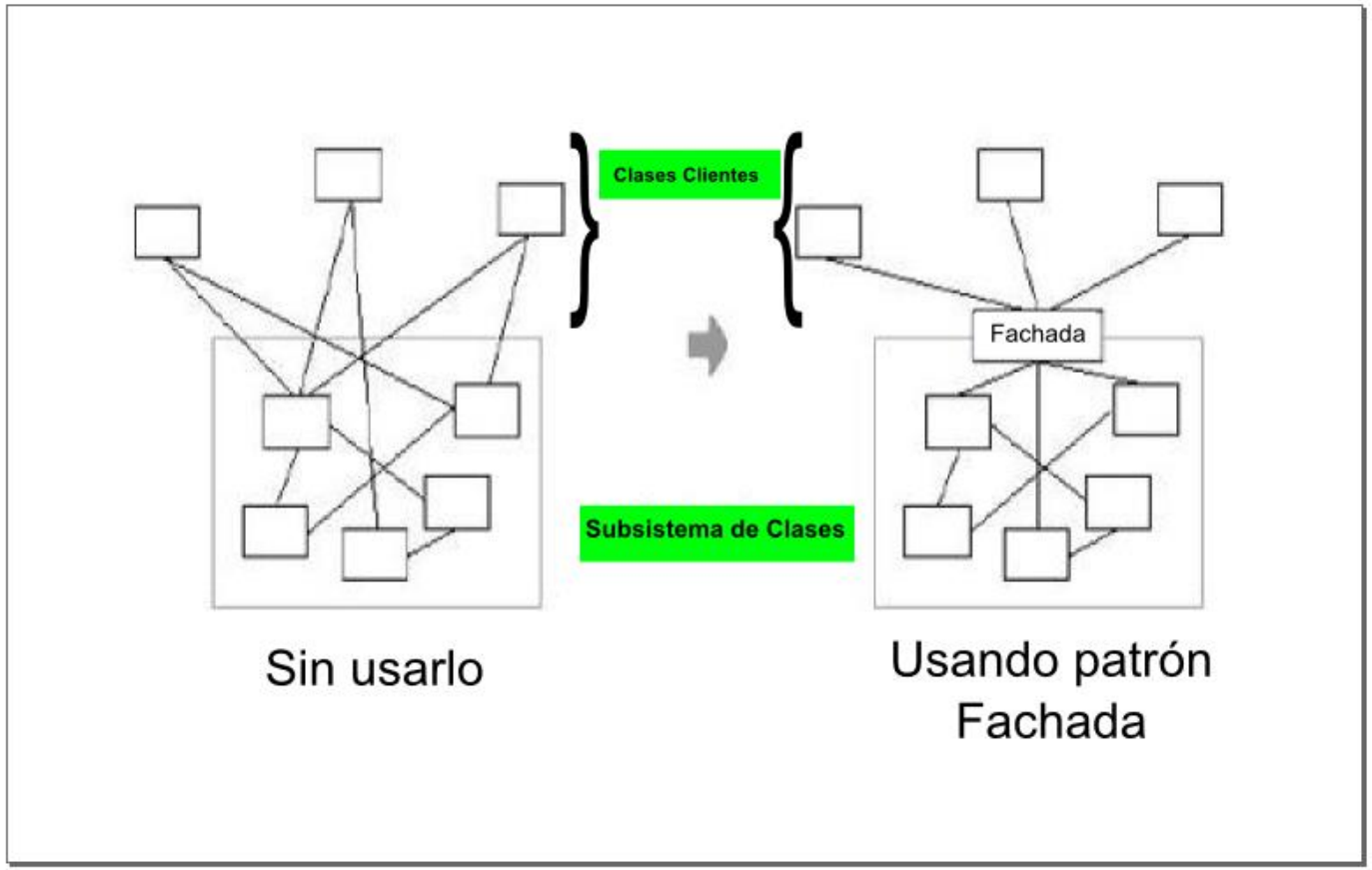
- La principal ventaja del patrón fachada consiste en que para modificar las clases de los subsistemas, sólo hay que realizar cambios en la interfaz/fachada, y los clientes pueden permanecer ajenos a ello. Además, y como se mencionó anteriormente, los clientes no necesitan conocer las clases que hay 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.

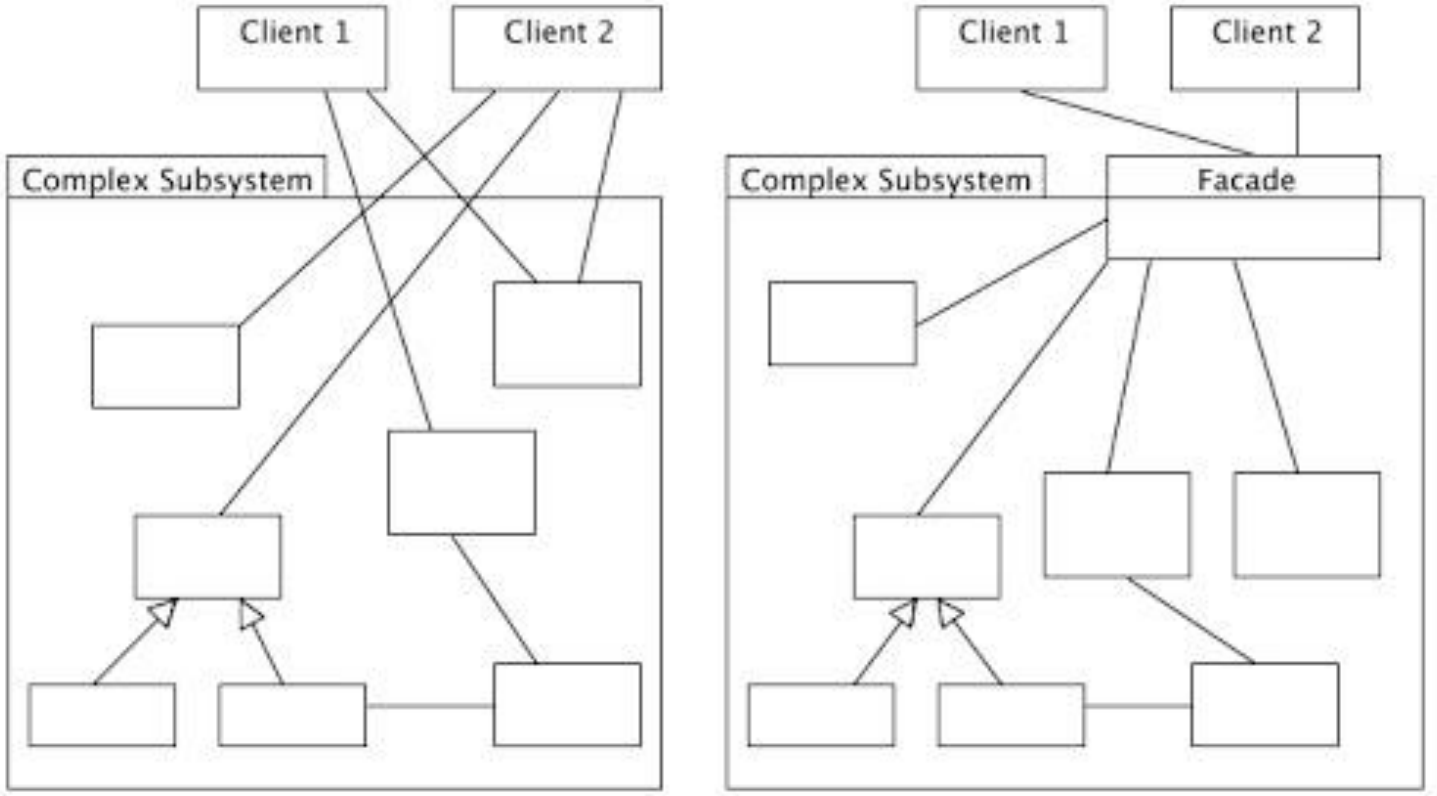
Patrón Estructural – Fachada



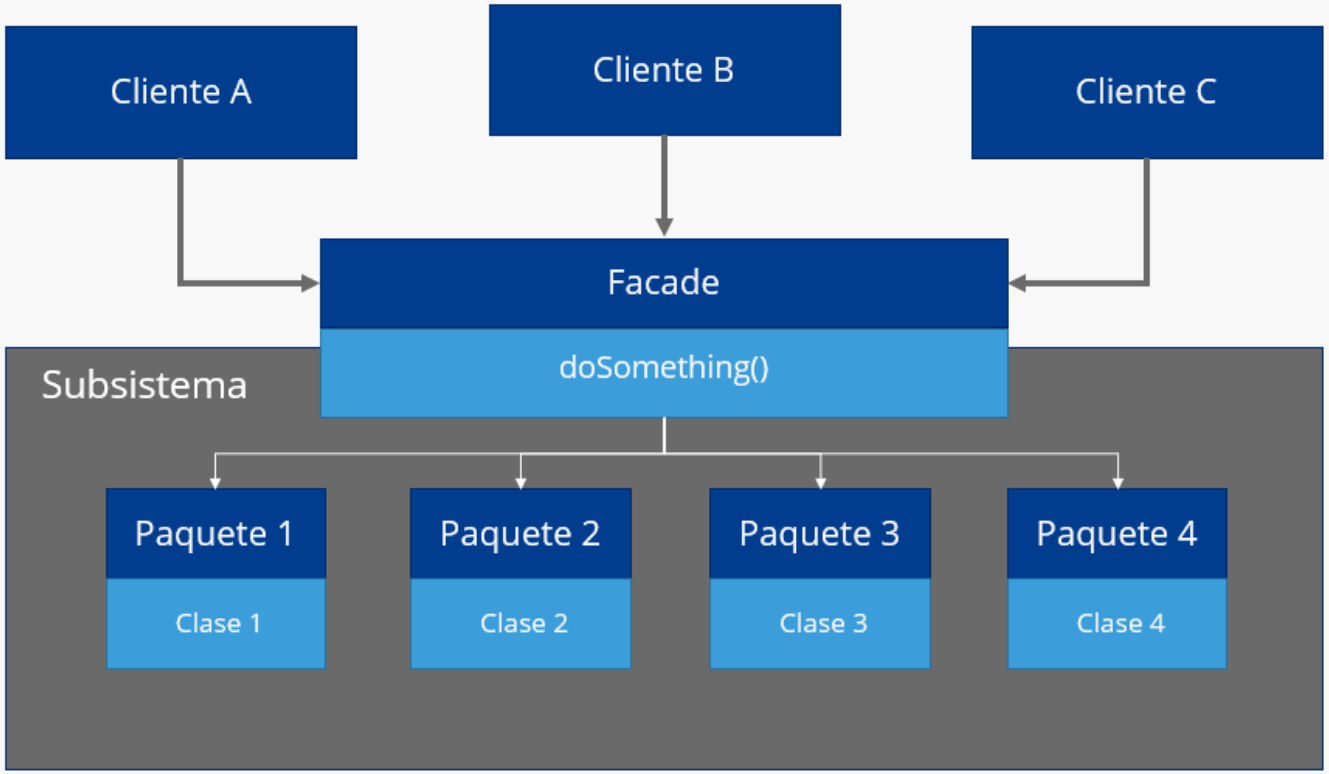
Patrón Estructural – Fachada



Patrón Estructural – Fachada



Patrón Estructural – Fachada



Patrón Estructural – Fachada

```java
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;

	}
```

```java
	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();

	}
```

Patrones

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

```java
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();

	 }
}
```

```java
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();

	}
}
```

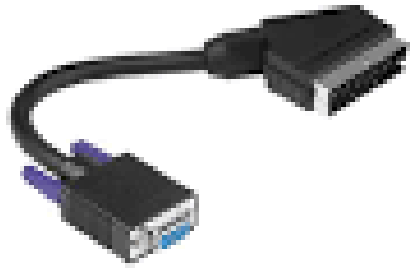
Patrón Estructural – Fachada

```
class Facade {
    protected subsystem1: Subsystem1;

    protected subsystem2: Subsystem2;

    /**
     * Depending on your application's needs, you can provide the Facade with
     * existing subsystem objects or force the Facade to create them on its own.
     */
    constructor(subsystem1: Subsystem1 = null, subsystem2: Subsystem2 = null) {
        this.subsystem1 = subsystem1 || new Subsystem1();
        this.subsystem2 = subsystem2 || new Subsystem2();
    }

    /**
     * The Facade's methods are convenient shortcuts to the sophisticated
     * functionality of the subsystems. However, clients get only to a fraction
     * of a subsystem's capabilities.
     */
    public operation(): string {
        let result = 'Facade initializes subsystems:\n';
        result += this.subsystem1.operation1();
        result += this.subsystem2.operation1();
        result += 'Facade orders subsystems to perform the action:\n';
        result += this.subsystem1.operationN();
        result += this.subsystem2.operationZ();

        return result;
    }
}
```

```
/**
 * The Subsystem can accept requests either from the facade or client directly.
 * In any case, to the Subsystem, the Facade is yet another client, and it's not
 * a part of the Subsystem.
 */
class Subsystem1 {
    public operation1(): string {
        return 'Subsystem1: Ready!\n';
    }

    // ...

    public operationN(): string {
        return 'Subsystem1: Go!\n';
    }
}

/**
 * Some facades can work with multiple subsystems at the same time.
 */
class Subsystem2 {
    public operation1(): string {
        return 'Subsystem2: Get ready!\n';
    }

    // ...

    public operationZ(): string {
        return 'Subsystem2: Fire!';
    }
}
```

```
/**
 * The client code works with complex subsystems through a simple interface
 * provided by the Facade. When a facade manages the lifecycle of the subsystem,
 * the client might not even know about the existence of the subsystem. This
 * approach lets you keep the complexity under control.
 */
function clientCode(facade: Facade) {
    // ...

    console.log(facade.operation());

    // ...
}

/**
 * The client code may have some of the subsystem's objects already created. In
 * this case, it might be worthwhile to initialize the Facade with these objects
 * instead of letting the Facade create new instances.
 */
const subsystem1 = new Subsystem1();
const subsystem2 = new Subsystem2();
const facade = new Facade(subsystem1, subsystem2);
clientCode(facade);
```

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.

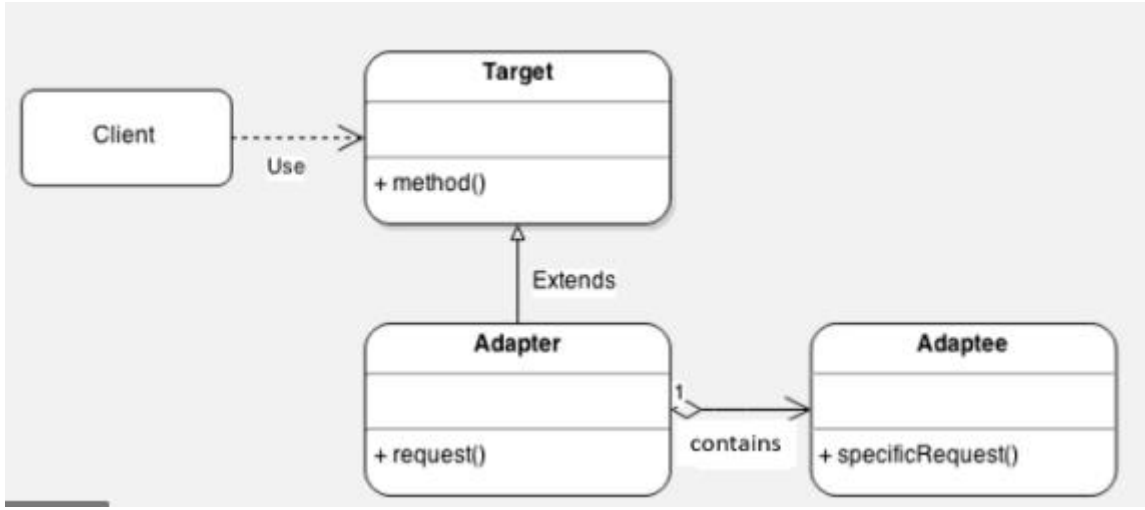
El patrón de diseño Adapter dice en su definición que convierte una interfaz o clase en otra interfaz que el cliente desea.

Patrón Estructural – ADAPTER

Partes del patrón de diseño Adapter

Las partes de patrón Adapter son:

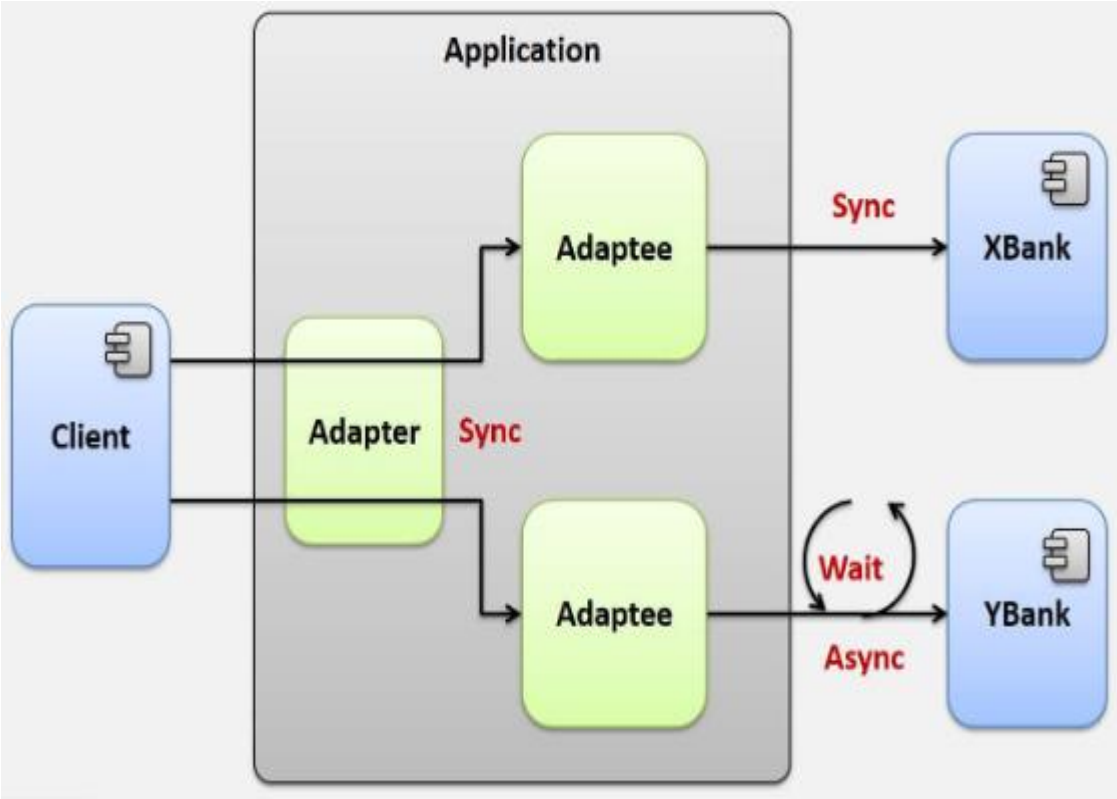
- **Target**: la interfaz que usamos para crear el adapter.
- **Adapter**: es la implementación del target y que se ocupará de realizar la adaptación.
- **Client**: es el que interactúa y usa el adapter.
- **Adaptee**: es la interfaz incompatible que necesitamos adaptar con el adapter.



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 que 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

```java
public interface IPersonaNueva {

    public String getNombre() ;
    public void setNombre(String nombre) ;
    public int getEdad() ;
    public void setEdad(int edad);

}
```

```java
public class PersonaNueva implements IPersonaNueva {
    private String nombre;
    private int edad;

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
```

```java
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);

}
```

```java
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

```java
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);
    }
```

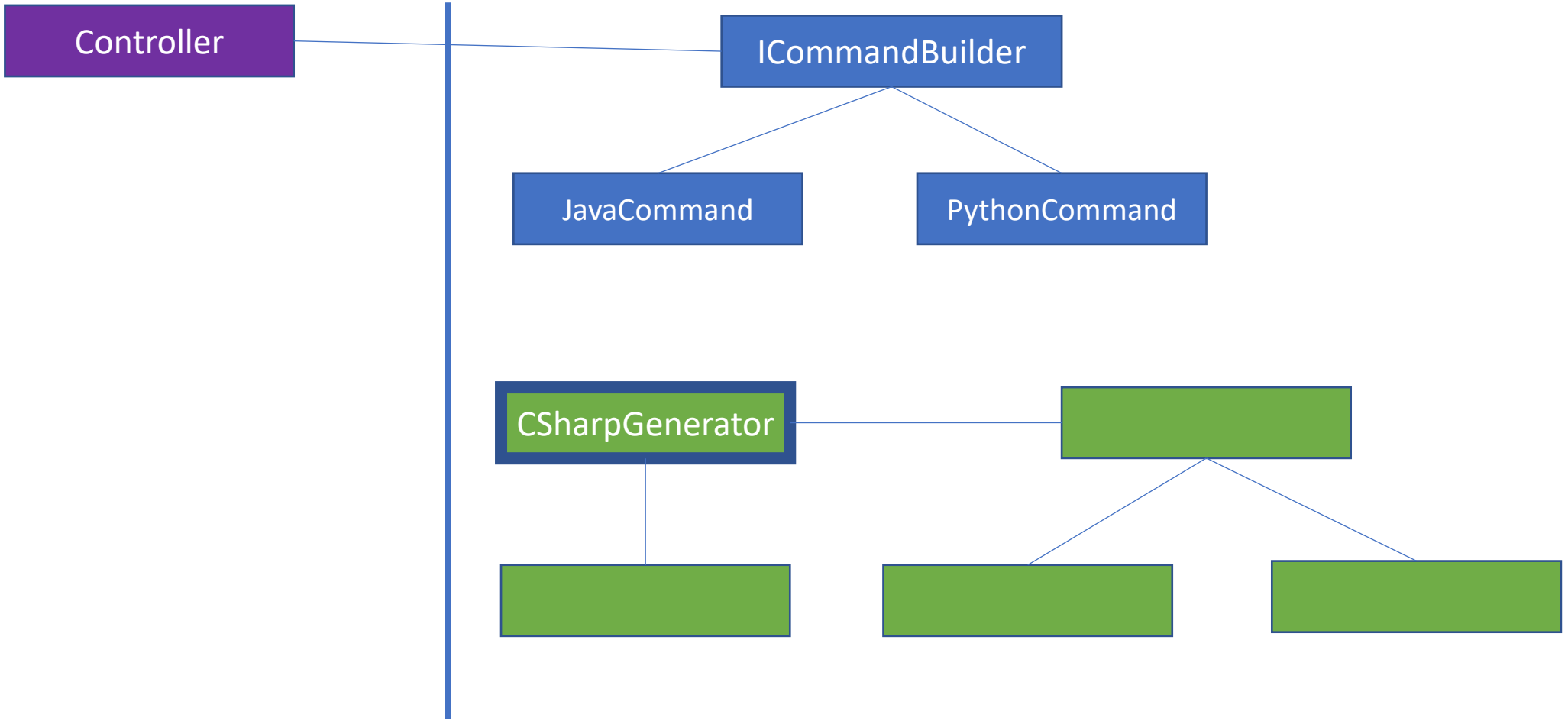
```java
public static void main(String[] args) {

    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());

}
```

Patrón Estructural – ADAPTER

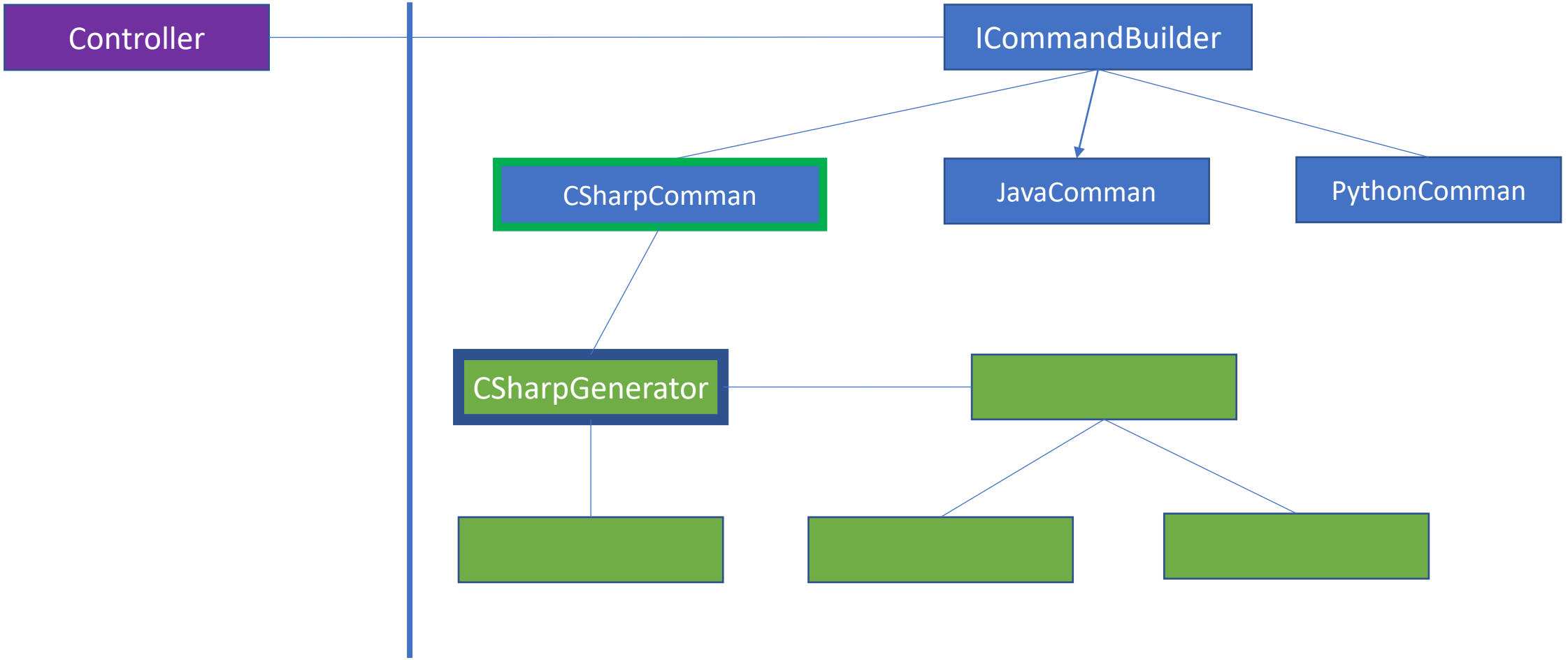
```
/**
 * The Target defines the domain-specific interface used by the client code.
 */
class Target {
    public request(): string {
        return 'Target: The default target\'s behavior.';
    }
}

/**
 * The Adaptee contains some useful behavior, but its interface is incompatible
 * with the existing client code. The Adaptee needs some adaptation before the
 * client code can use it.
 */
class Adaptee {
    public specificRequest(): string {
        return '.eetpadA eht fo roivaheb laicep5';
    }
}

/**
 * The Adapter makes the Adaptee's interface compatible with the Target's
 * interface.
 */
class Adapter extends Target {
    private adaptee: Adaptee;

    constructor(adaptee: Adaptee) {
        super();
        this.adaptee = adaptee;
    }

    public request(): string {
        const result = this.adaptee.specificRequest().split('').reverse().join('');
        return `Adapter: (TRANSLATED) ${result}`;
    }
}
```

```
/**
 * The client code supports all classes that follow the Target interface.
 */
function clientCode(target: Target) {
    console.log(target.request());
}

console.log('Client: I can work just fine with the Target objects:');
const target = new Target();
clientCode(target);

console.log('');

const adaptee = new Adaptee();
console.log('Client: The Adaptee class has a weird interface. See, I don\'t understand i
console.log(`Adaptee: ${adaptee.specificRequest()}`);

console.log('');

console.log('Client: But I can work with it via the Adapter:');
const adapter = new Adapter(adaptee);
clientCode(adapter);
```

Patrón Estructural – ADAPTER



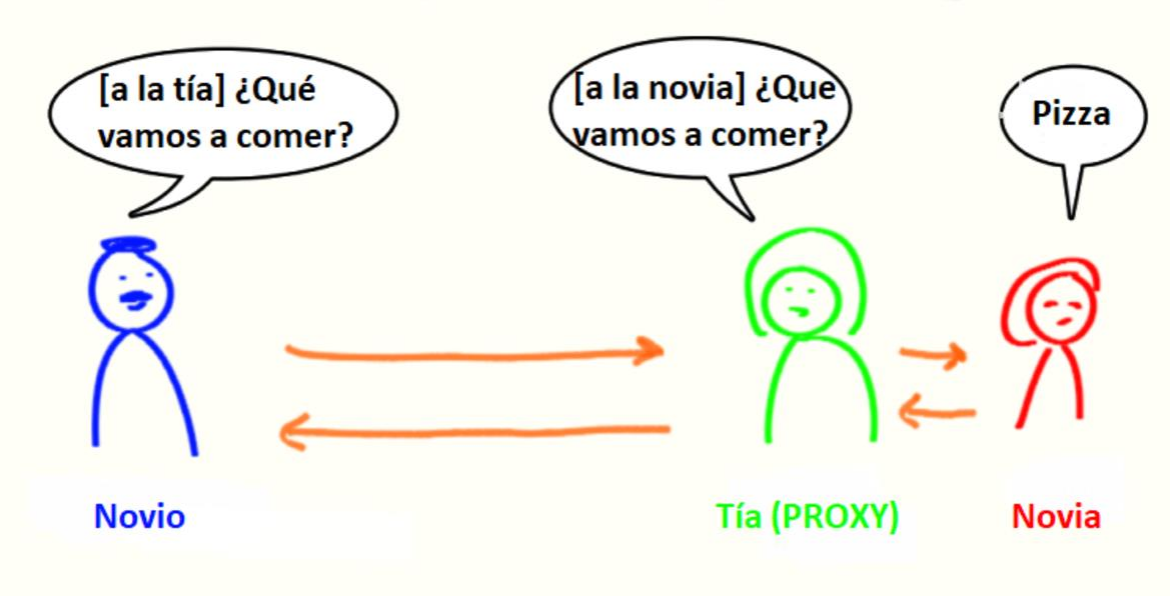
Patrón Estructural – ADAPTER



Patrones

Patrón Estructural – Proxy

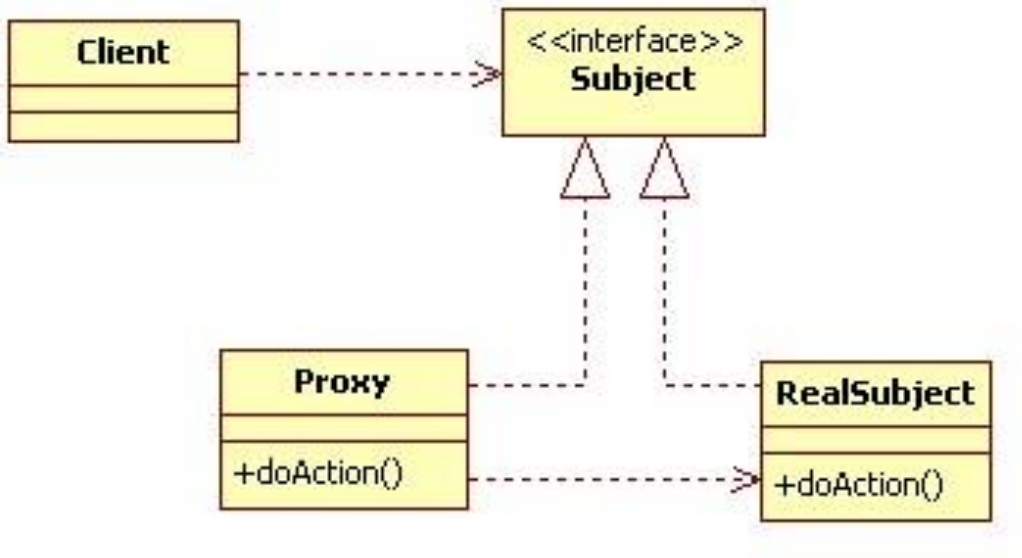
El patrón Proxy se utiliza como intermediario para acceder a un objeto, permitiendo controlar el acceso a él. Para ello obliga que las llamadas a un objeto ocurran indirectamente a través de un objeto proxy, que actúa como un sustituto del objeto original, delegando luego las llamadas a los métodos de los objetos respectivos.



Patrón Estructural – Proxy

Participantes:

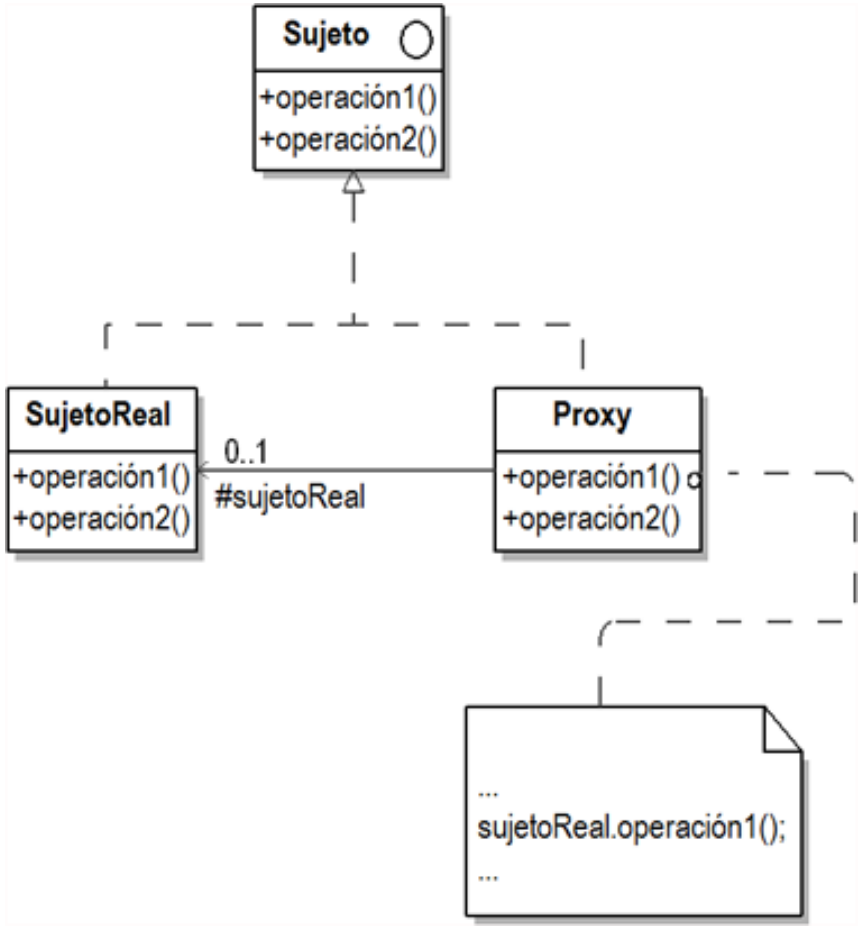
- **Subject:** interfaz o clase abstracta que proporciona un acceso común al objeto real y su representante (proxy).
- **Proxy:** mantiene una referencia al objeto real. Controla la creación y acceso a las operaciones del objeto real.
- **RealSubject:** define el objeto real representado por el Proxy.
- **Cliente:** 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:

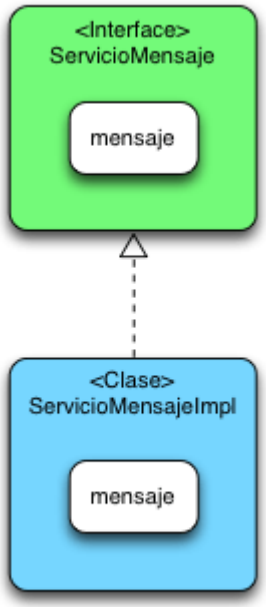
- Podemos modificar funcionalidad sin afectar a los clientes.
- El código se puede volver complicado si es necesario introducir gran cantidad de funcionalidad nueva.
- La respuesta real del servicio puede ser desconocida hasta que realmente se la invoca.

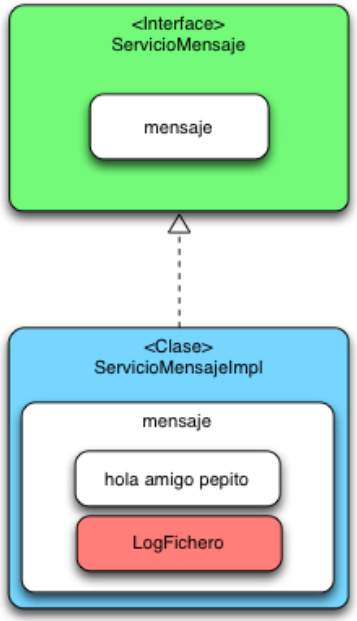


Patrón Estructural – Proxy

Ejemplo:

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





Patrón Estructural – Proxy

Ejemplo:

```java
package com.arquitecturajava.proxy;

public interface ServicioMensaje {
public String mensaje(String persona);
}
```

```java
package com.arquitecturajava.proxy2;

public class ServicioMensajeImpl implements ServicioMensaje {

 public String mensaje(String persona)
{

 return "hola amigo "+ persona;
 }
}
```

```java
package com.arquitecturajava.proxy;

public class Principal {

public static void main(String[] args) {

ServicioMensaje sm= new ServicioMensajeImpl();
System.out.println(sm.mensaje("pepito"));

}

}
```

Patrón Estructural – Proxy

Ejemplo:

```java
package com.arquitecturajava.proxy2;

public class ServicioMensajeProxy implements ServicioMensaje {
private ServicioMensaje sm;


@Override
public String mensaje(String persona) {

System.out.println("log del mensaje para"+ persona);
//mensaje delegado
return sm.mensaje(persona);
}
public ServicioMensajeProxy() {
super();
this.sm = new ServicioMensajeImpl();
}

}
```

```java
package com.arquitecturajava.proxy2;

public class Principal {

public static void main(String[] args) {

ServicioMensaje sm= new ServicioMensajeProxy();
System.out.println(sm.mensaje("pepito"));

}

}
```

Patrón Estructural – Proxy

Ejemplo:

```
/**
 * The Subject interface declares common operations for both RealSubject and the
 * Proxy. As long as the client works with RealSubject using this interface,
 * you'll be able to pass it a proxy instead of a real subject.
 */
interface Subject {
    request(): void;
}

/**
 * The RealSubject contains some core business logic. Usually, RealSubjects are
 * capable of doing some useful work which may also be very slow or sensitive -
 * e.g. correcting input data. A Proxy can solve these issues without any
 * changes to the RealSubject's code.
 */
class RealSubject implements Subject {
    public request(): void {
        console.log('RealSubject: Handling request.');
    }
}
```

```
/**
 * The Proxy has an interface identical to the RealSubject.
 */
class Proxy implements Subject {
    private realSubject: RealSubject;

    /**
     * The Proxy maintains a reference to an object of the RealSubject class. It
     * can be either lazy-loaded or passed to the Proxy by the client.
     */
    constructor(realSubject: RealSubject) {
        this.realSubject = realSubject;
    }

    /**
     * The most common applications of the Proxy pattern are lazy loading,
     * caching, controlling the access, logging, etc. A Proxy can perform one of
     * these things and then, depending on the result, pass the execution to the
     * same method in a linked RealSubject object.
     */
    public request(): void {
        if (this.checkAccess()) {
            this.realSubject.request();
            this.logAccess();
        }
    }

    private checkAccess(): boolean {
        // Some real checks should go here.
        console.log('Proxy: Checking access prior to firing a real request.');

        return true;
    }

    private logAccess(): void {
        console.log('Proxy: Logging the time of request.');
    }
}
```

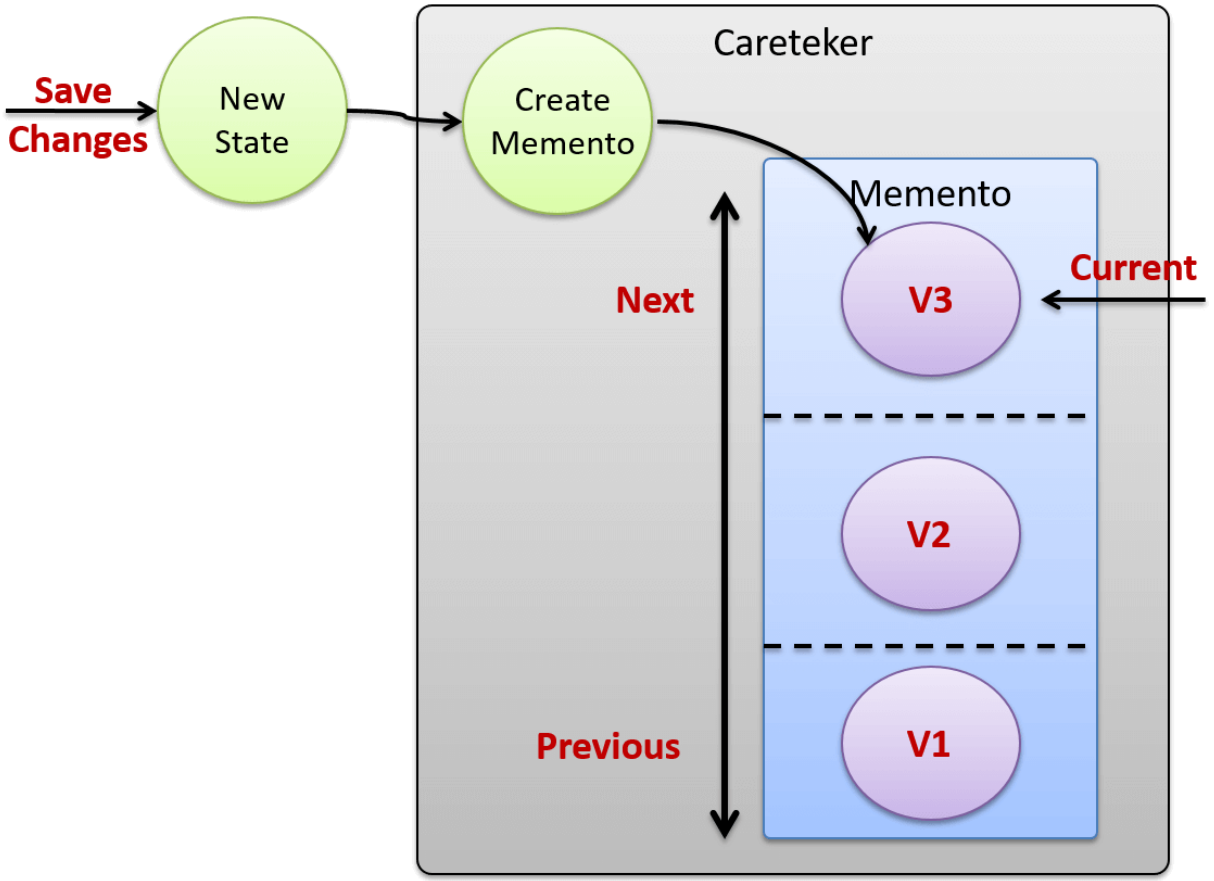
```
/**
 * The client code is supposed to work with all objects (both subjects and
 * proxies) via the Subject interface in order to support both real subjects and
 * proxies. In real life, however, clients mostly work with their real subjects
 * directly. In this case, to implement the pattern more easily, you can extend
 * your proxy from the real subject's class.
 */
function clientCode(subject: Subject) {
    // ...

    subject.request();

    // ...
}

console.log('Client: Executing the client code with a real subject:');
const realSubject = new RealSubject();
clientCode(realSubject);

console.log('');

console.log('Client: Executing the same client code with a proxy:');
const proxy = new Proxy(realSubject);
clientCode(proxy);
```

Output.txt: Resultado de la ejecución

```
Client: Executing the client code with a real subject:
RealSubject: Handling request.

Client: Executing the same client code with a proxy:
Proxy: Checking access prior to firing a real request.
RealSubject: Handling request.
Proxy: Logging the time of request.
```

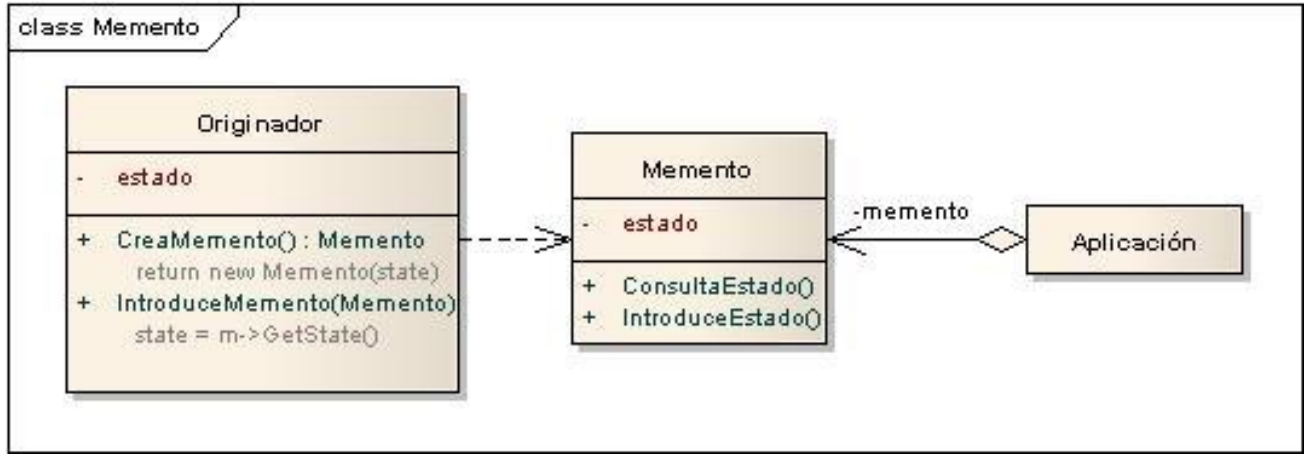
Patrón Memento

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

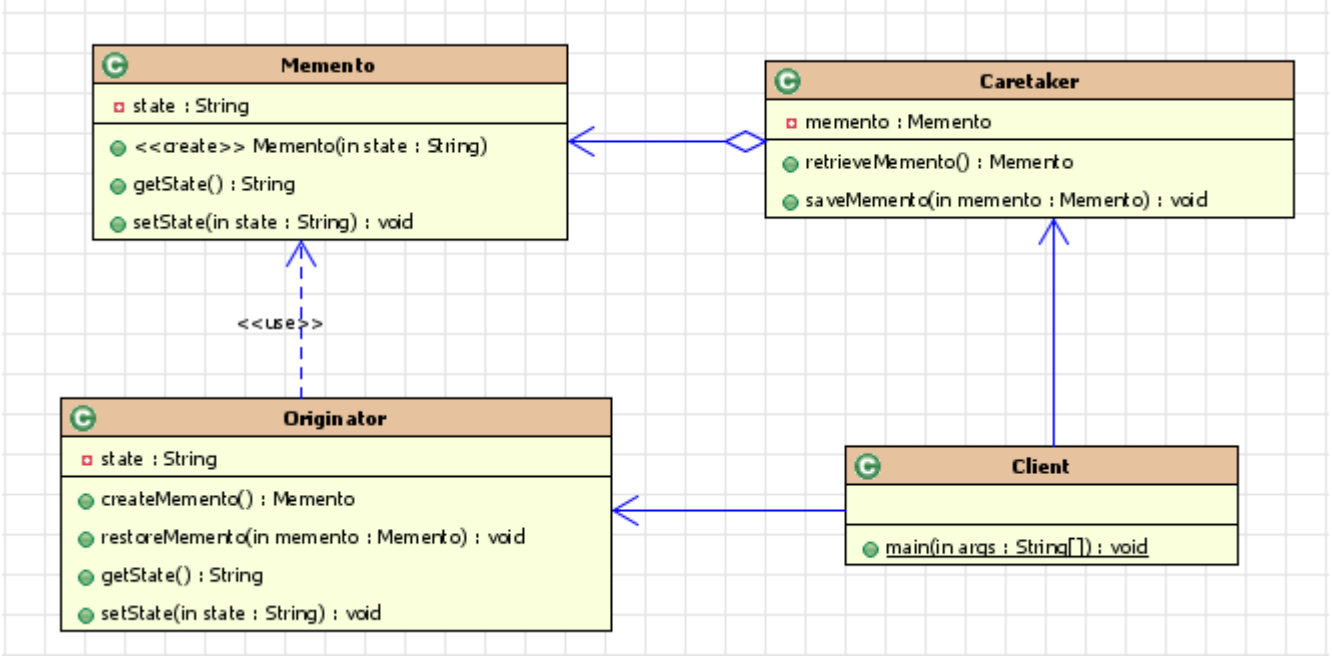
- **Memento**: Almacena el estado de un objeto Originador. Memento almacena todo o parte de Originador. Tiene dos interfaces, una para Aplicación que le permite comunicarse con otros objetos y otra para Originador que le permite almacenar el estado.
- **Originador**: Crea un objeto Memento con una copia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

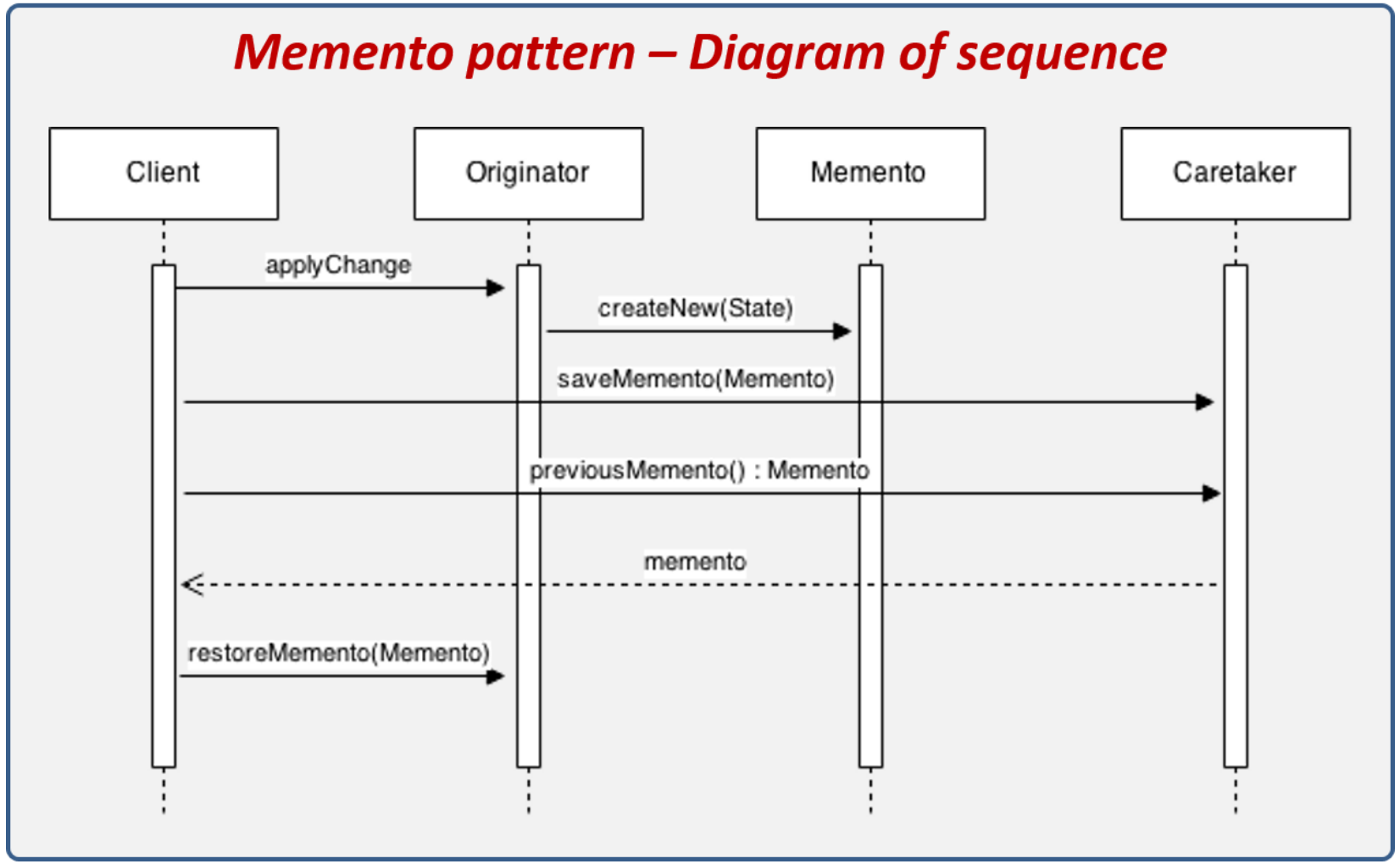


Implementación:

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.



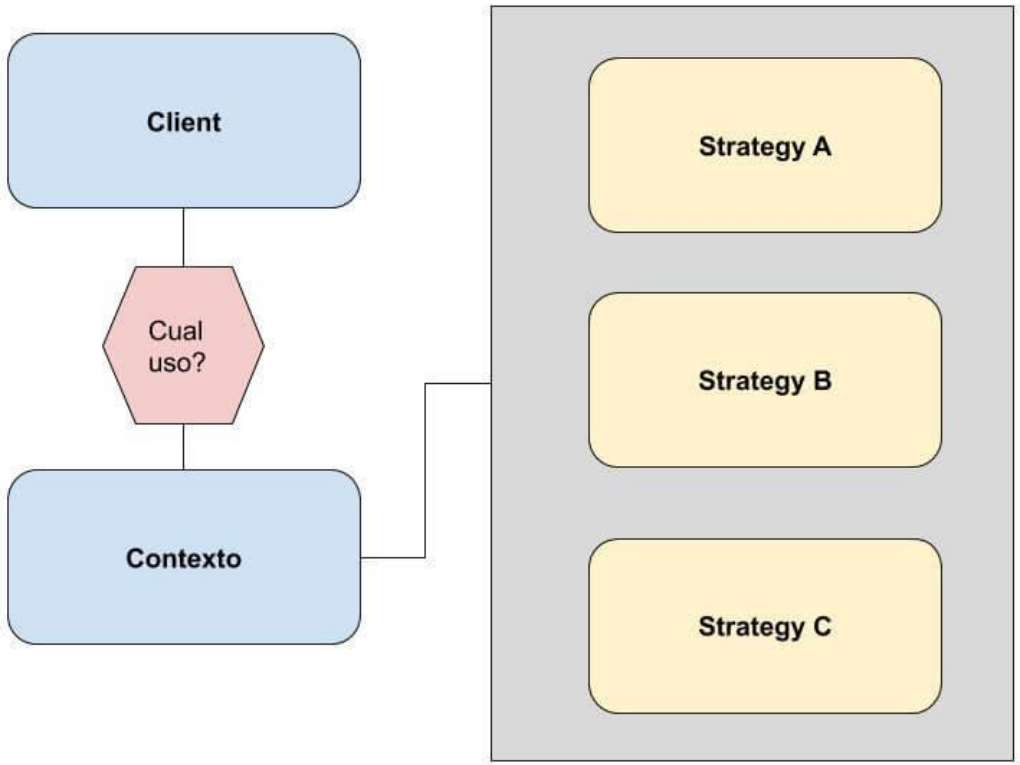

Memento pattern – Diagram of sequence
Client
Originator
Memento
Caretaker
applyChange
createNew(State)
saveMemento(Memento)
previousMemento() : Memento
memento
restoreMemento(Memento)

Patrones

Patrón de comportamiento - STRATEGY

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

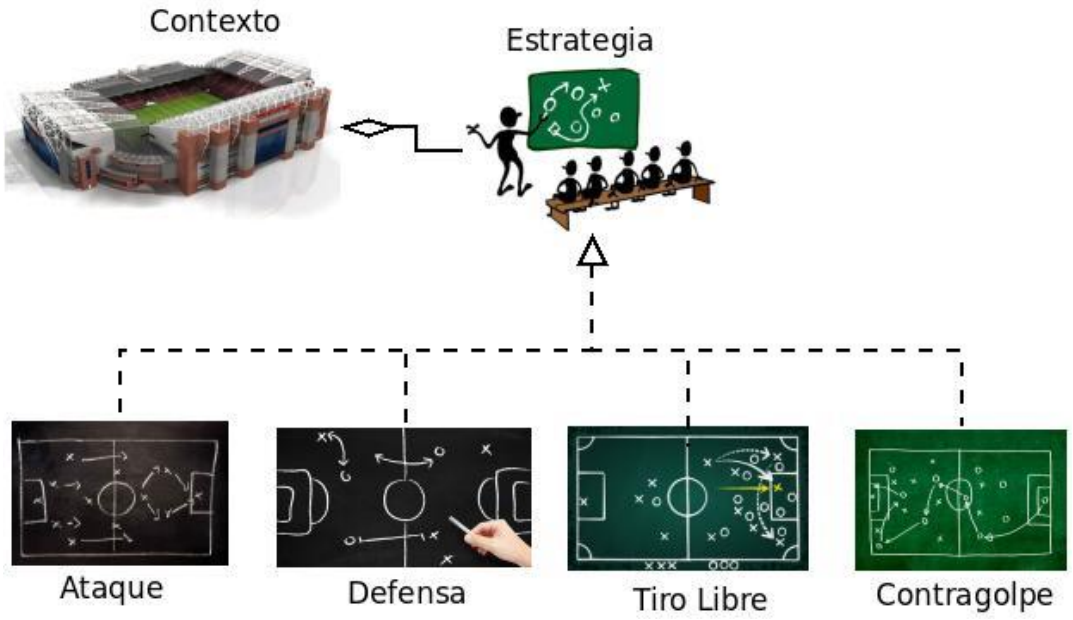
En el patrón Strategy creamos diferentes clases que representan estrategias y que podremos usar según alguna variación o input.



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

```java
package patterns.strategy;

public interface CommissionStrategy {
    double applyCommission(double amount);
}
```

```java
package patterns.strategy;

public class FullCommission implements CommissionStrategy {
    @Override
    public double applyCommission(double amount) {
        // do complicated formula of commissions.
        return amount * 0.50d;
    }
}
```

```java
package patterns.strategy;

public class NormalCommission implements CommissionStrategy {
    @Override
    public double applyCommission(double amount) {
        // do complicated formula of commissions.
        return amount * 0.30;
    }
}
```

```java
package patterns.strategy;

public class RegularCommision implements CommissionStrategy {
    @Override
    public double applyCommission(double amount) {
        // do complicated formula of commissions.
        return amount * 0.10;
    }
}
```

Patrón de comportamiento – STRATEGY

```java
package patterns.strategy;

public class Context {

    private CommissionStrategy commissionStrategy;

    public Context(CommissionStrategy commissionStrategy){
        this.commissionStrategy = commissionStrategy;
    }

    public double executeStrategy(double amount){
        return commissionStrategy.applyCommission(amount);
    }
}
```

```java
package patterns.strategy;

public interface CommissionStrategy {
    double applyCommission(double amount);
}
```

Patrón de comportamiento – STRATEGY

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

Patrón de comportamiento – STRATEGY

```
/**
 * The Context defines the interface of interest to clients.
 */
class Context {
    /**
     * @type {Strategy} The Context maintains a reference to one of the Strategy
     * objects. The Context does not know the concrete class of a strategy. It
     * should work with all strategies via the Strategy interface.
     */
    private strategy: Strategy;

    /**
     * Usually, the Context accepts a strategy through the constructor, but also
     * provides a setter to change it at runtime.
     */
    constructor(strategy: Strategy) {
        this.strategy = strategy;
    }

    /**
     * Usually, the Context allows replacing a Strategy object at runtime.
     */
    public setStrategy(strategy: Strategy) {
        this.strategy = strategy;
    }

    /**
     * The Context delegates some work to the Strategy object instead of
     * implementing multiple versions of the algorithm on its own.
     */
    public doSomeBusinessLogic(): void {
        // ...

        console.log('Context: Sorting data using the strategy (not sure how it\'ll do it
        const result = this.strategy.doAlgorithm(['a', 'b', 'c', 'd', 'e']);
        console.log(result.join(','));

        // ...
    }
}
```

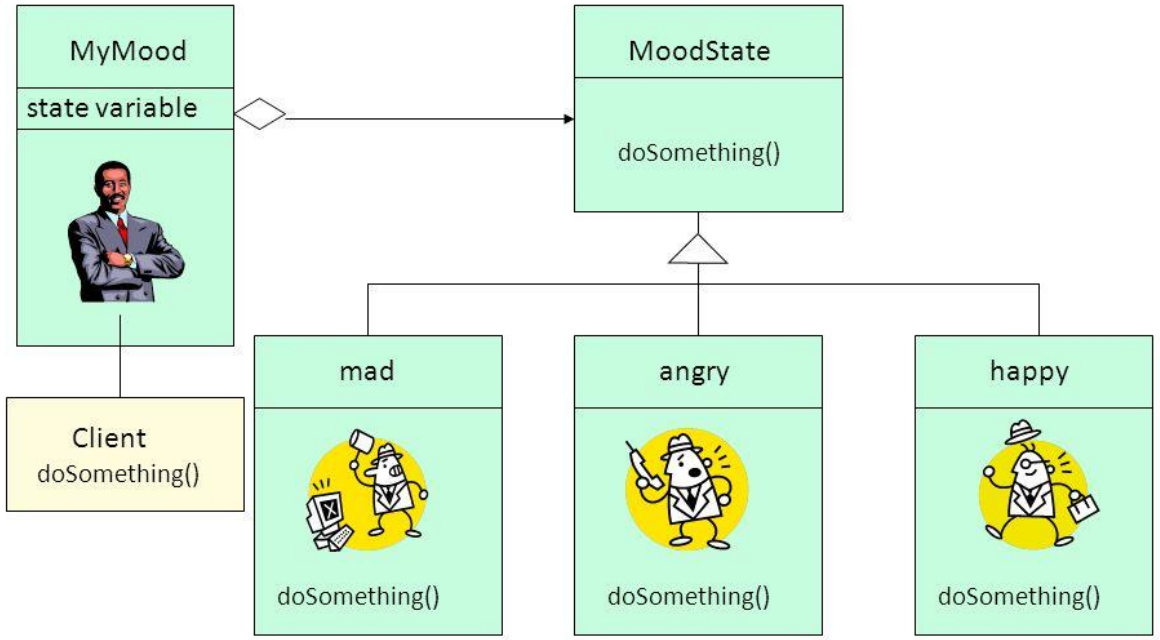
```
/**
 * The Strategy interface declares operations common to all supported versions
 * of some algorithm.
 *
 * The Context uses this interface to call the algorithm defined by Concrete
 * Strategies.
 */
interface Strategy {
    doAlgorithm(data: string[]): string[];
}

/**
 * Concrete Strategies implement the algorithm while following the base Strategy
 * interface. The interface makes them interchangeable in the Context.
 */
class ConcreteStrategyA implements Strategy {
    public doAlgorithm(data: string[]): string[] {
        return data.sort();
    }
}

class ConcreteStrategyB implements Strategy {
    public doAlgorithm(data: string[]): string[] {
        return data.reverse();
    }
}

/**
 * The client code picks a concrete strategy and passes it to the context. The
 * client should be aware of the differences between strategies in order to make
 * the right choice.
 */
const context = new Context(new ConcreteStrategyA());
console.log('Client: Strategy is set to normal sorting.');
context.doSomeBusinessLogic();

console.log('');

console.log('Client: Strategy is set to reverse sorting.');
context.setStrategy(new ConcreteStrategyB());
context.doSomeBusinessLogic();
```

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

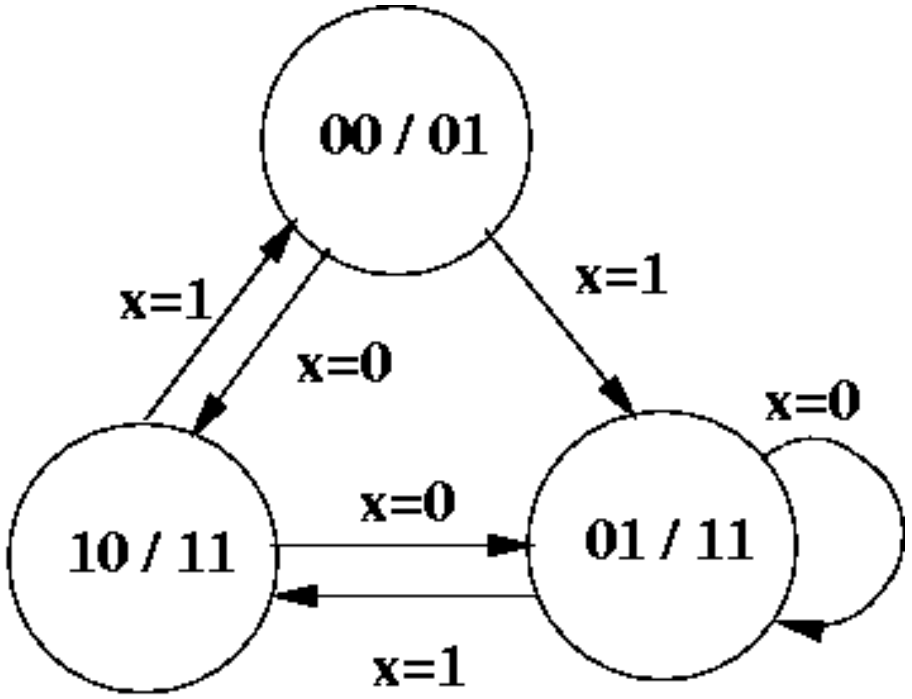
Aplicabilidad:

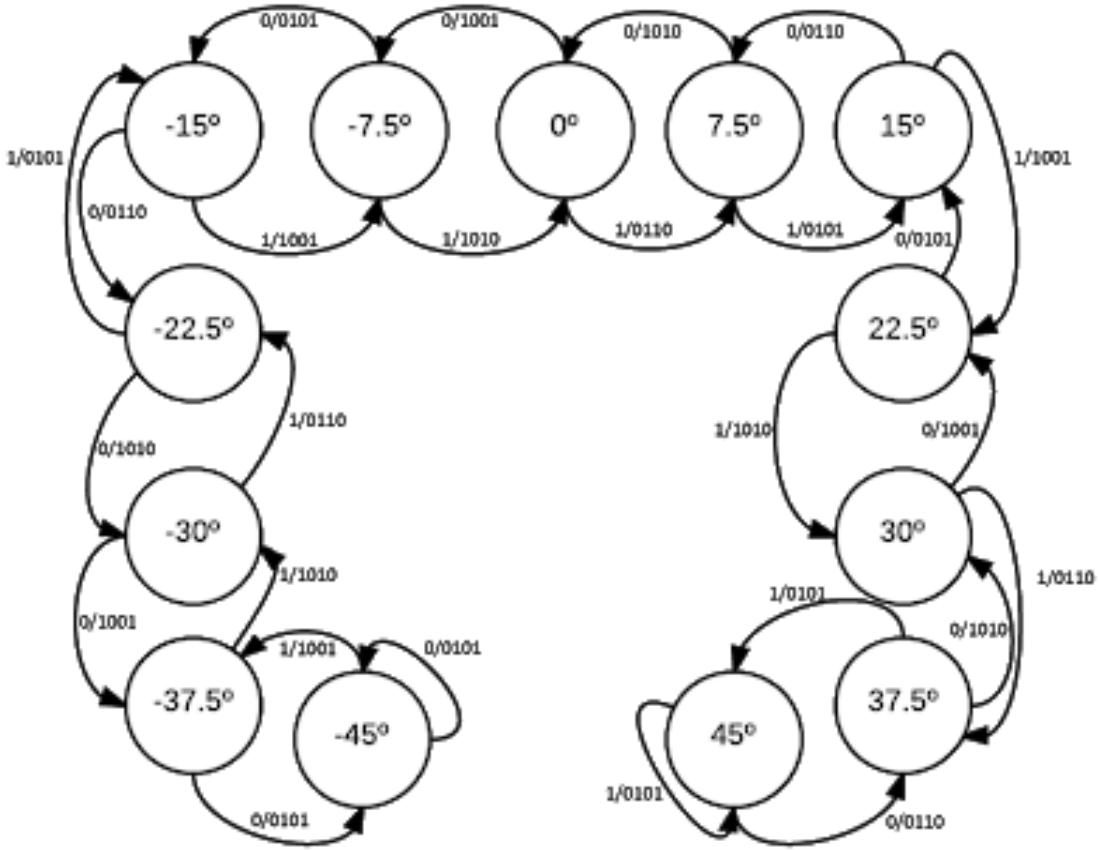
El patrón State se usa en cualquiera de estos casos:

- 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

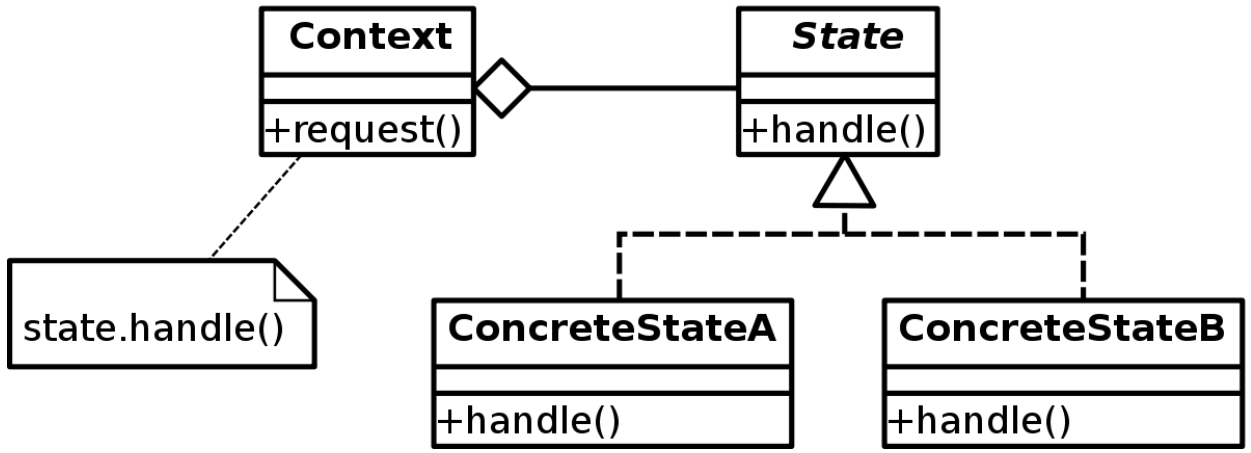




Patrón de comportamiento – state

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.

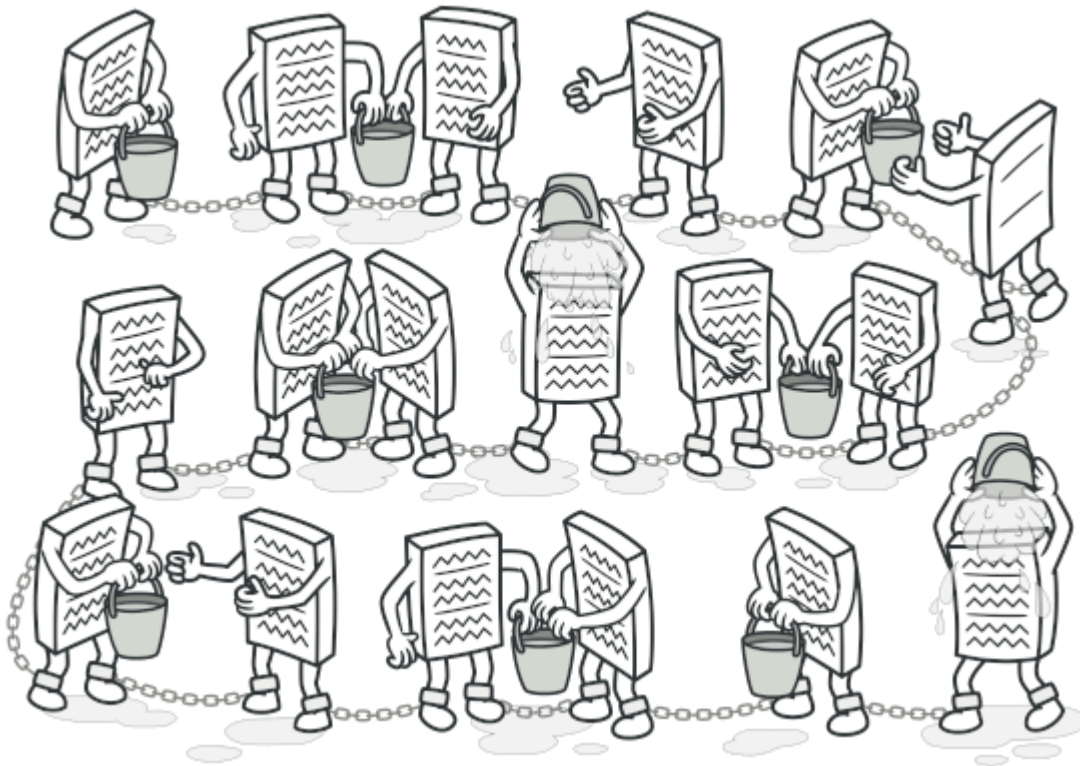


Patrón de comportamiento – state

Ventajas:

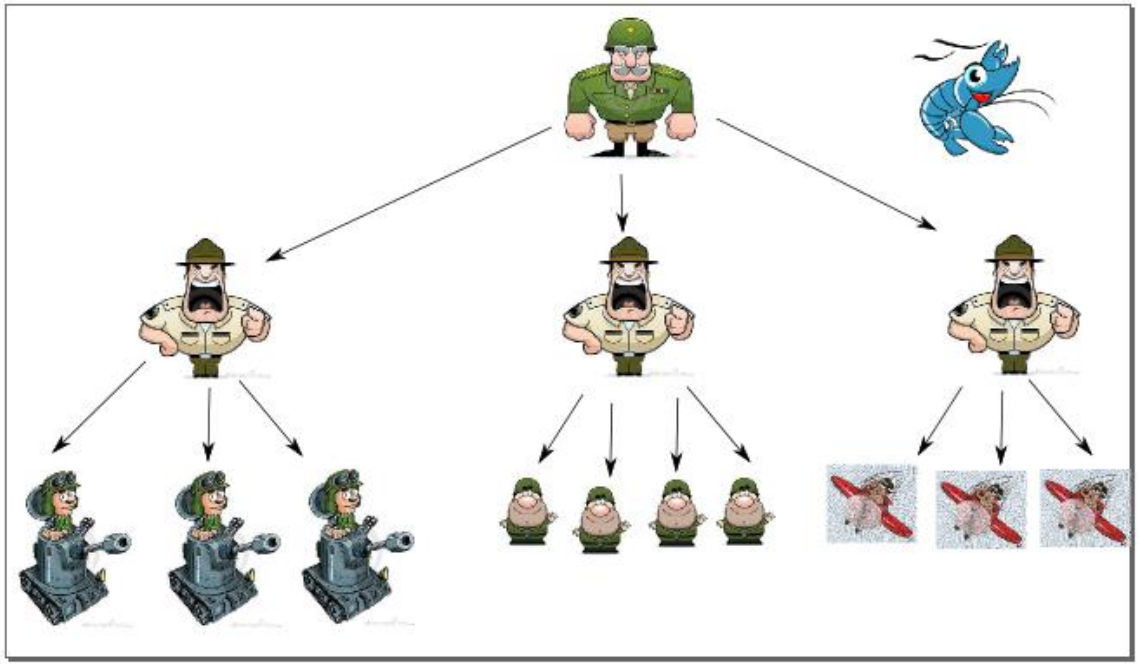
- **Localiza el comportamiento de un estado específico y divide el comportamiento para diferentes estados**: El patrón State pone todo el comportamiento asociado con un estado particular en un objeto. Debido a que todo el código de un estado específico está en un una subclase de State, los nuevos estados 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.

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.

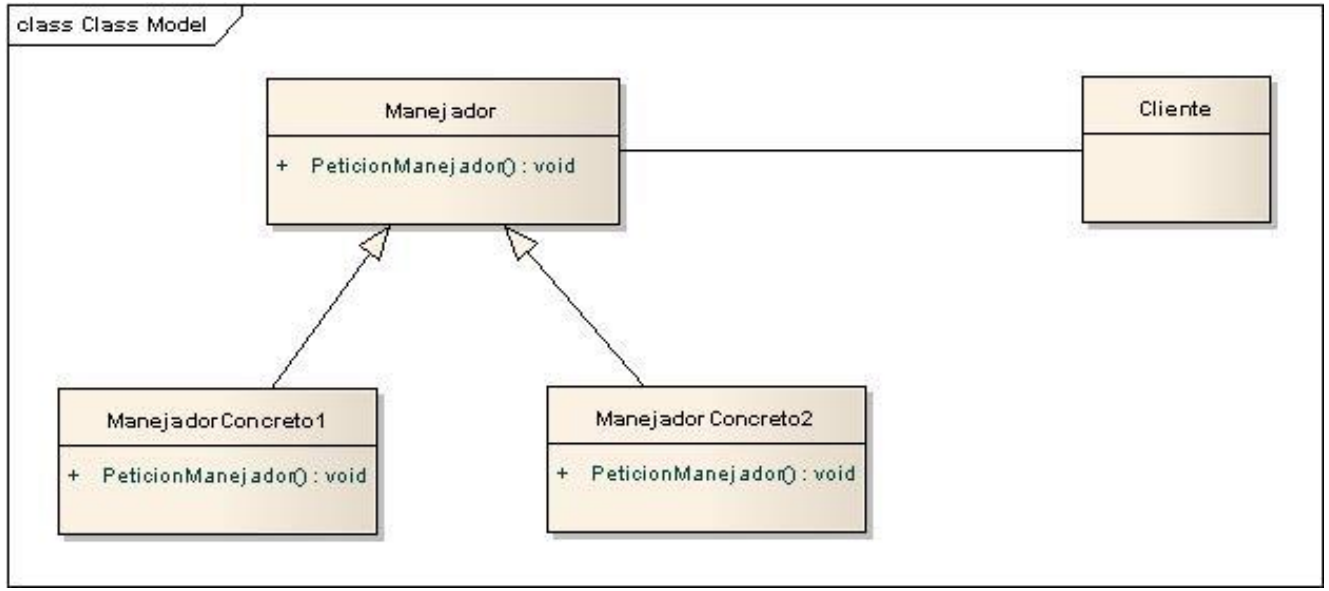


Participantes

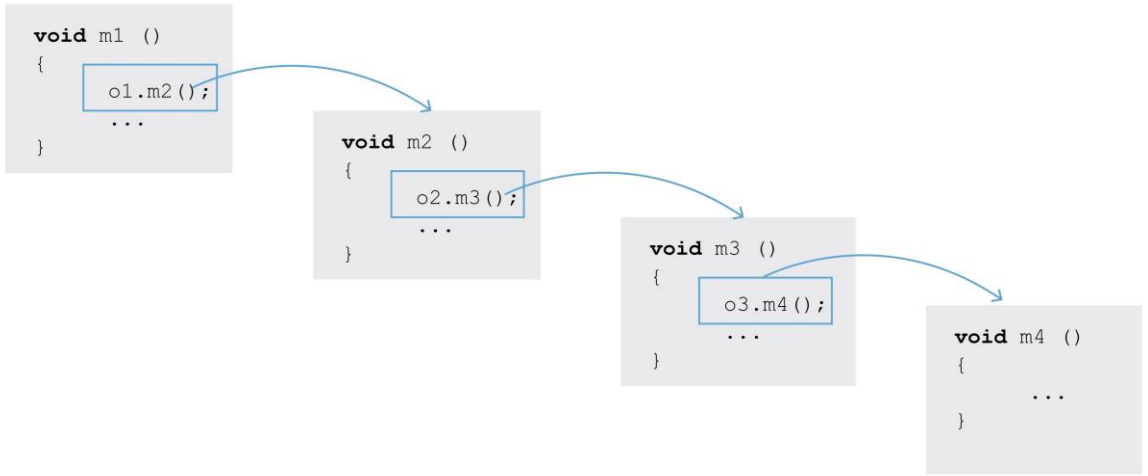
Cliente: Inicia la petición que llega a la cadena en busca del responsable.

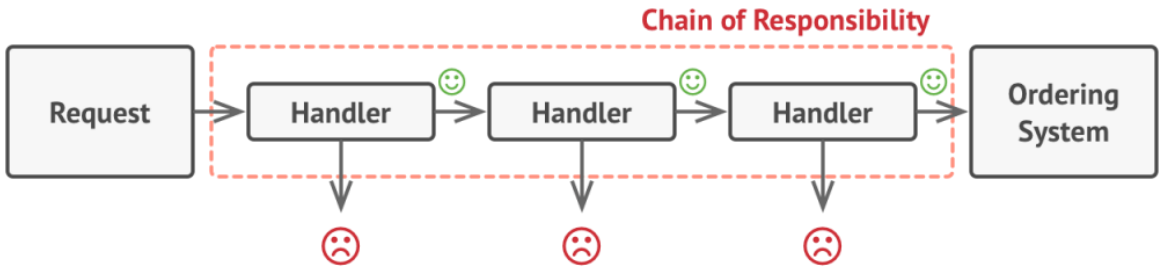
Manejador: Define una interfaz para manejar peticiones.

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, está se propaga por el Manejador hasta que un Manejador Concreto asume la responsabilidad de procesarla.

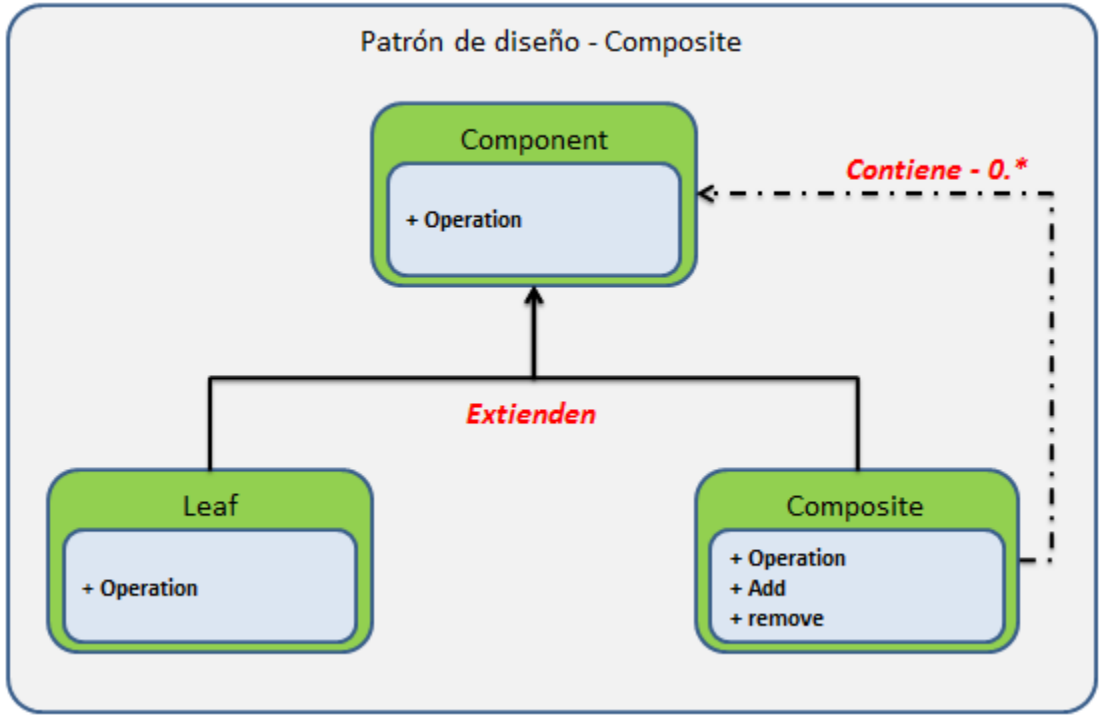




Patrones

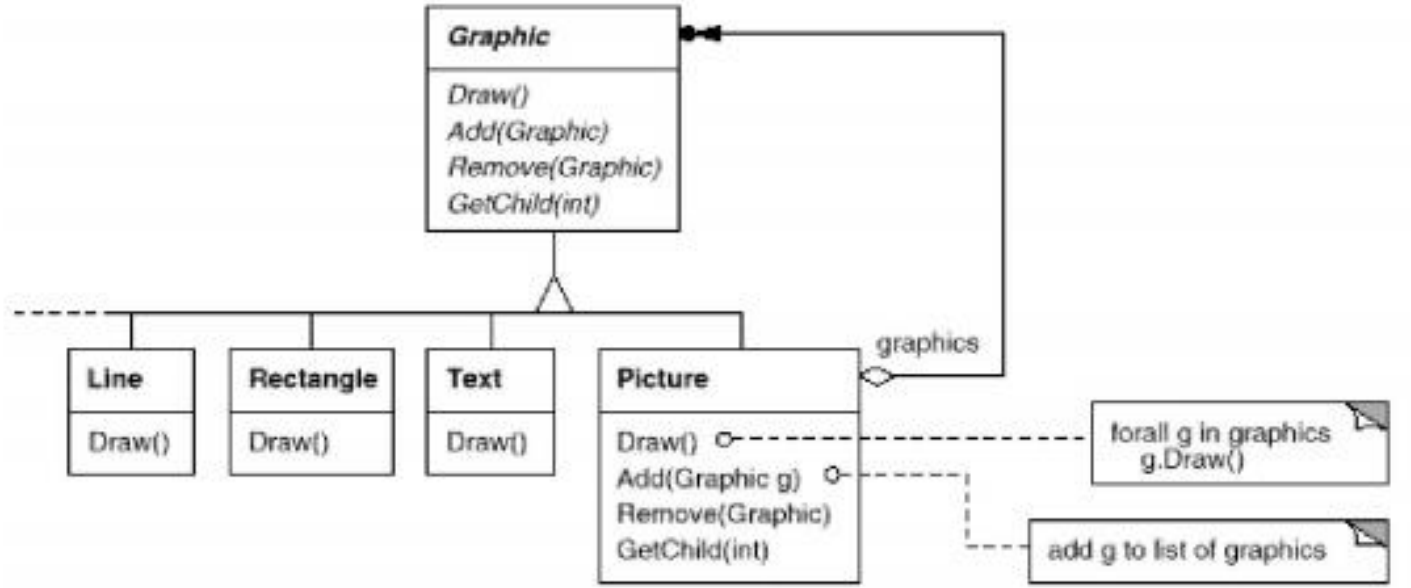
Patrón estructura – Composite

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.

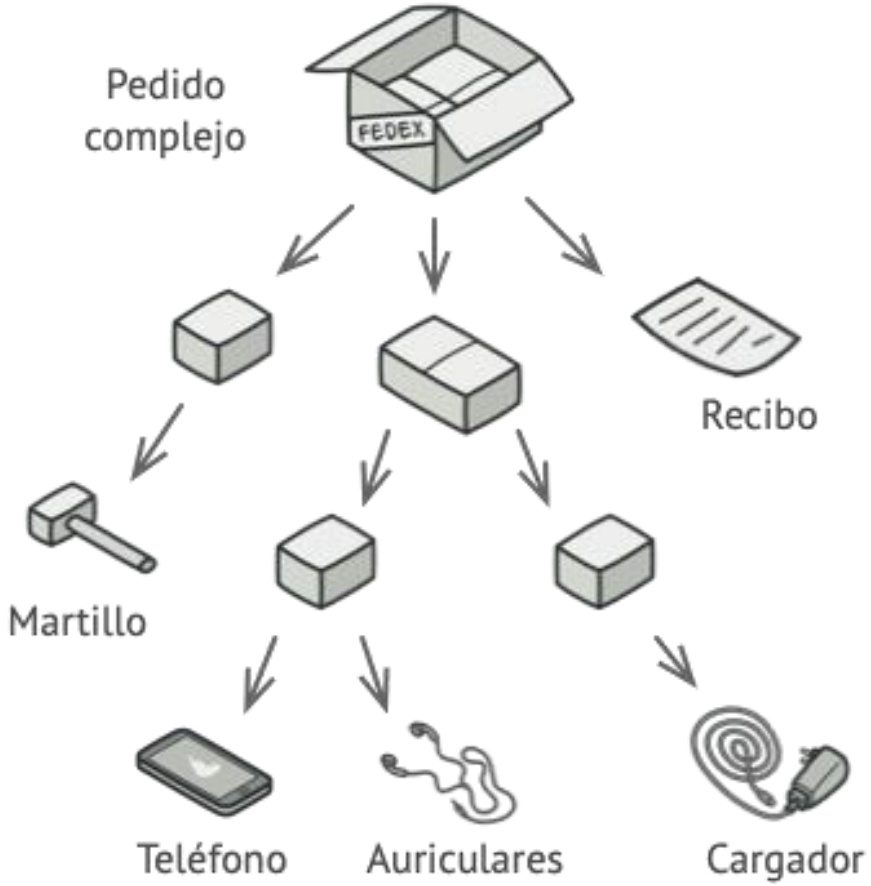
Para el cliente, resulta totalmente indiferente de qué tipo de objeto se trate (parte o todo) ya que solo se comunica con la interfaz.

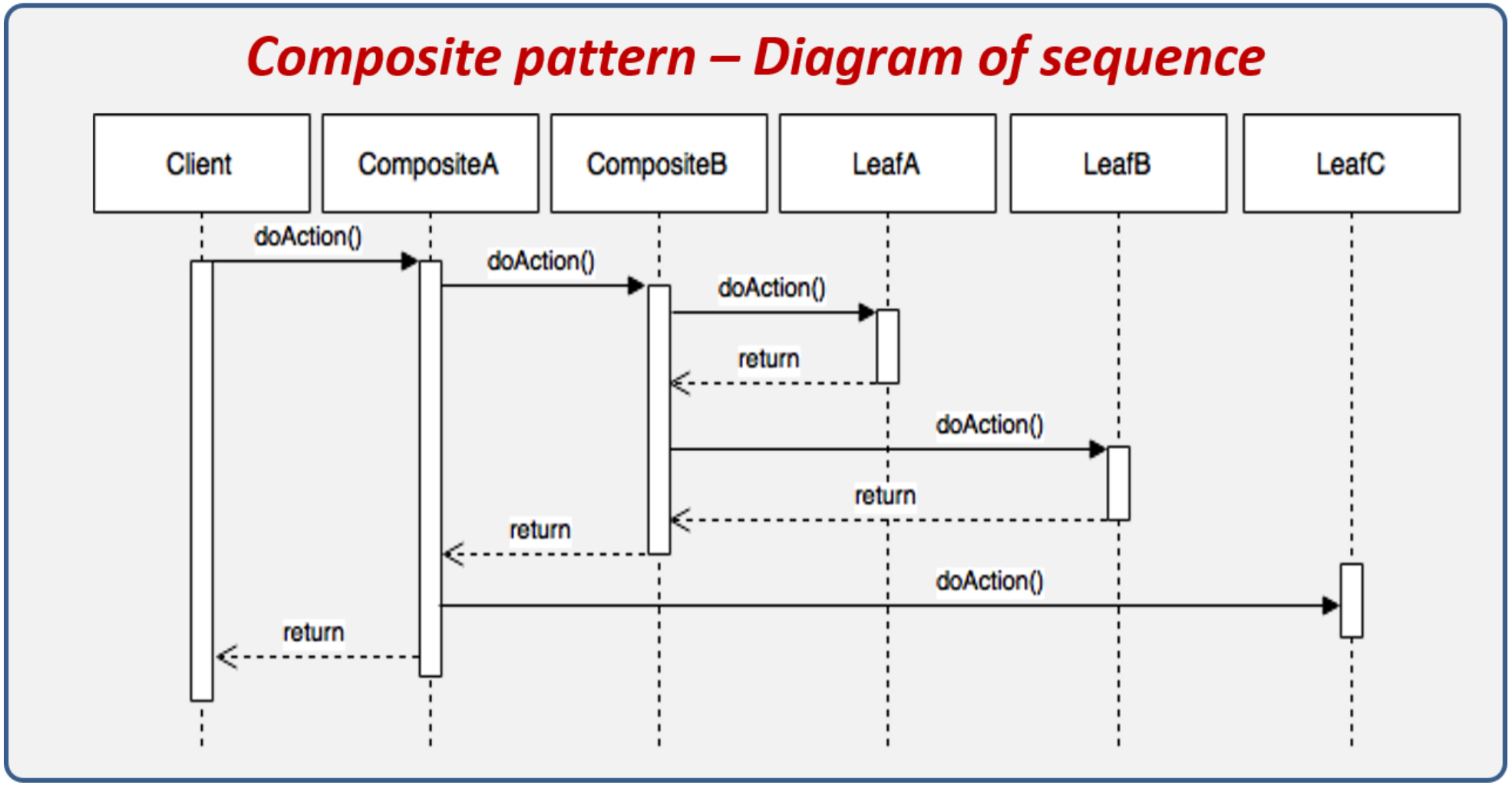


Aplicabilidad

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
Client
CompositeA
CompositeB
LeafA
LeafB
LeafC
doAction()
doAction()
doAction()
return
doAction()
return
return
doAction()
return

Patrón composición

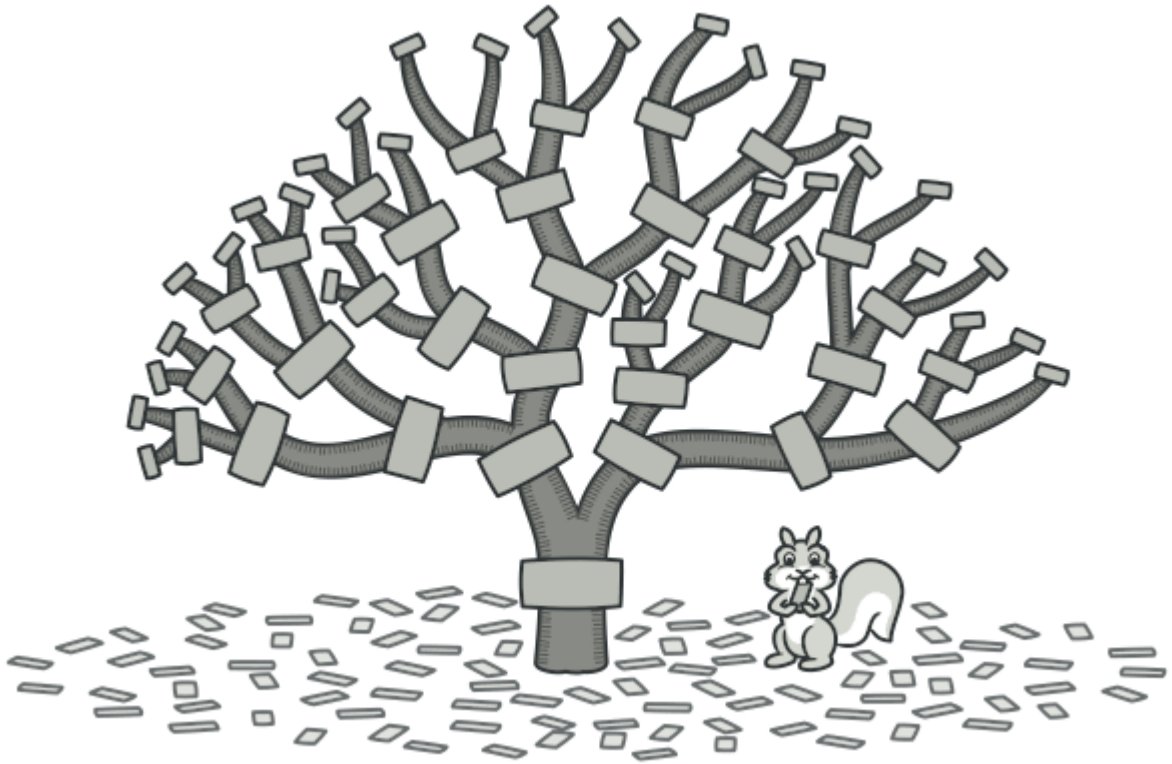
Ventajas:

- **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.

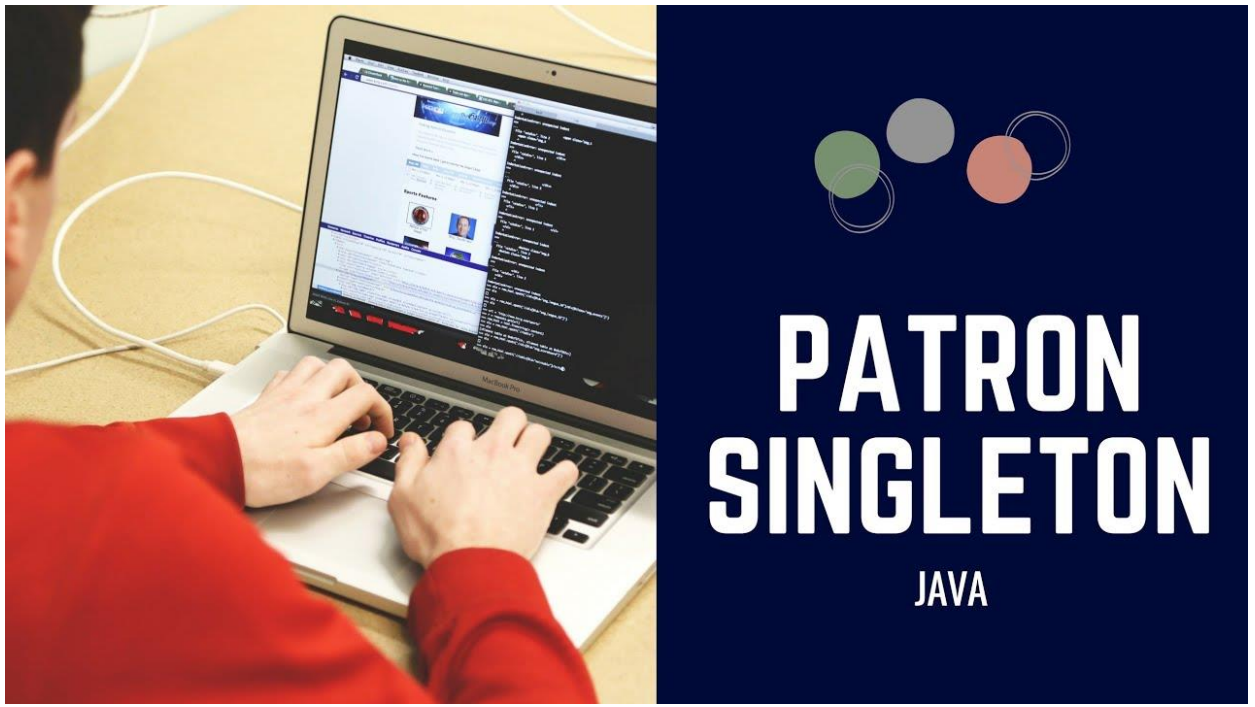
Patrón composición

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.



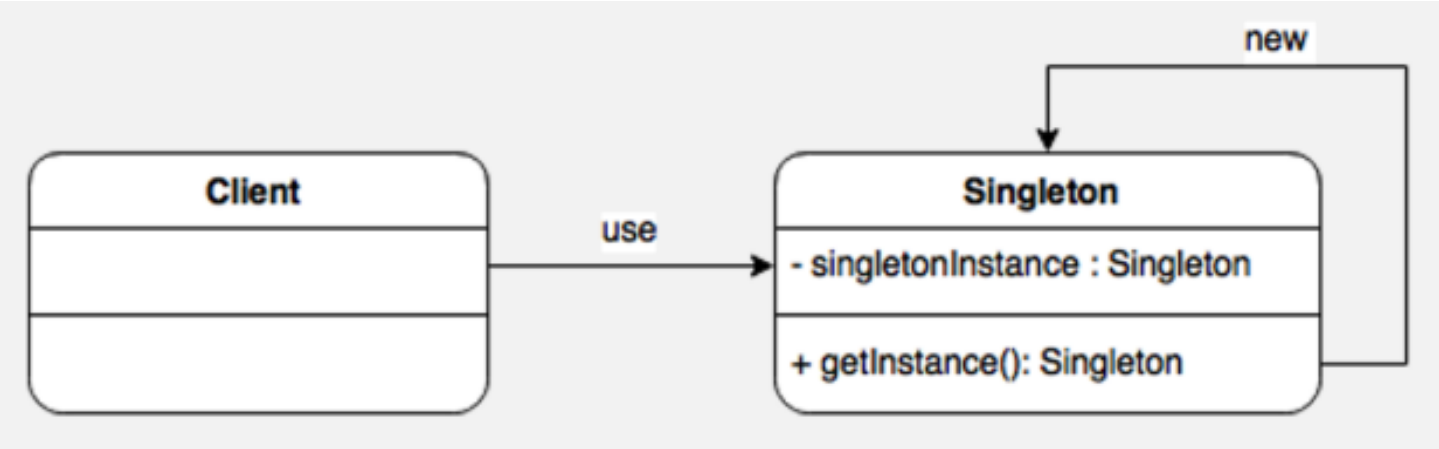
Patrones



Patrón de 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 Typescript es la siguiente:

```
/**
 * The Singleton class defines the `getInstance` method that lets clients access
 * the unique singleton instance.
 */
class Singleton {
    private static instance: Singleton;

    /**
     * The Singleton's constructor should always be private to prevent direct
     * construction calls with the `new` operator.
     */
    private constructor() { }

    /**
     * The static method that controls the access to the singleton instance.
     *
     * This implementation let you subclass the Singleton class while keeping
     * just one instance of each subclass around.
     */
    public static getInstance(): Singleton {
        if (!Singleton.instance) {
            Singleton.instance = new Singleton();
        }

        return Singleton.instance;
    }

    /**
     * Finally, any singleton should define some business logic, which can be
     * executed on its instance.
     */
    public someBusinessLogic() {
        // ...
    }
}
```