

## Garbage Collector

Se trata de un proceso de baja prioridad, el cual es el encargado de liberar la memoria que no se usa. Se basa en rastrear todos los objetos que aún se usan y marca el resto como basura.

**Montículo(heap) -> Donde se encuentran los objetos**

**Pila(Stack) -> Donde se encuentran las invocaciones a métodos y variables locales.**

La **Pila** se usa para la asignación de memoria estática, en ella se encuentran los valores primitivos de un métodos y referencias a objetos que viven en el montículo que son creados por un método.

Una variable local está viva hasta que finalice el método en el que se declaró.

El **Montículo** se usa para la asignación de memoria dinámica para los objetos y las clases de java. La vida de los objetos en el montículo depende de las referencias que se tengan a ese objeto cuando esas referencias lleguen a 0 se elimina el objeto.

Una variable de instancia está viva hasta que finalice el objeto al que pertenece la variable.

## Pila y montículo

Si una variable local es una referencia a un objeto, solo la variable va a la Pila, el objeto en si, va al Montículo

Las variables de instancia que no son de tipo primitivo y son referencias a objetos 'viven' también en el Montículo, dentro del objeto al que pertenecen.

## Constructor

El constructor no es un método de la clase, ya que se trata del código que se ejecuta cuando creamos una instancia de la clase.

### La sobrecarga de constructores:

Una clase puede tener más de un constructor mientras cada uno tenga argumentos diferentes.

## Interfaz

Una interfaz no es una clase, si no se trata de un conjunto de requisitos. En una interfaz se especifican un conjunto de métodos sin cuerpo o variables.

Una clase puede implementar cualquier número de interfaces y cada clase es libre de especificar los detalles su propia implementación como desee.

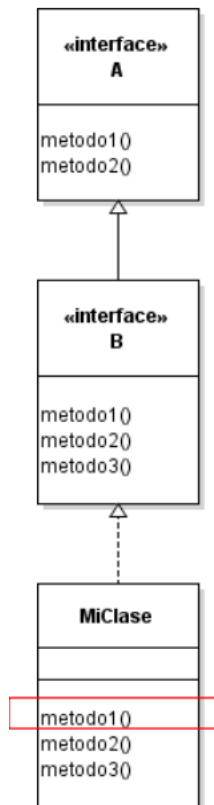
Los métodos de una interfaz a implementar en cualquier clase deben de declararse en las mismas como públicos.

Si una clase no implementa todos los métodos de una interfaz, dicha clase debe declararse como abstracta, ya que no se podrá instanciar ningún objeto de dicha clase. Pero permita que una subclase pueda proporcionar implementación completa de la interfaz.

Cuando creamos una instancia de una clase que implemente una interfaz la variable que almacena dicha instancia puede ser de referencia a la interfaz. Esto se puede dar gracias al polimorfismo.

Cuando se hace referencia a la interfaz solo tenemos conocimiento de los métodos declarados por la interfaz.

Las interfaces pueden ser extensibles usando la palabra clave `extends`.



**Los métodos predeterminados de una interfaz** son métodos que nos permiten realizarles una implementación en la interfaz. Estos métodos existen básicamente para que su implementación en las clases que implementen la interfaz sea opcional. Para definir estos métodos predeterminados en una interfaz se hace añadiendo la palabra clave `default` antes de declarar el método.

```
public interface Vehiculo {
    String getMarca();
    String acelerar();
    String frenar();

    default String encenderAlarma() {
        return "Encendiendo la alarma";
    }

    default String apagarAlarma() {
        return "Apagando la alarma";
    }
}
```

Estos dos métodos están disponibles en la clase Coche pero no es necesario implementarlos explícitamente en esta clase

```
public class Coche implements Vehiculo {
    private String marca;

    // constructores y getters

    @Override
    public String getMarca() {
        return marca;
    }

    @Override
    public String acelerar() {
        return "El coche está acelerando";
    }

    @Override
    public String frenar() {
        return "El coche está frenando";
    }
}
```

**Los métodos estáticos de una interfaz** son métodos que tienen una implementación realizada en la propia interfaz, la cual no se puede volver a implementar en las clases que implementen la interfaz. Por lo que sería un método que tendría la misma implementación en todas las clases que lo implementen. Para hacer uso de los métodos estáticos de la interfaz no hace falta tener una implementación de la interfaz ni tampoco ninguna instancia de la interfaz.

```
public interface Vehiculo {  
    // IGUAL QUE ANTES  
  
    static double getPotenciaHP(int rpm, int parMotor) {  
        return (rpm * parMotor) / 5252;  
    }  
}
```

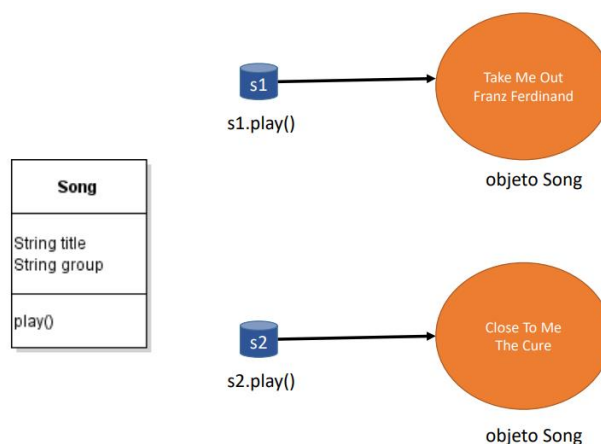
podemos invocarlo así:

```
Vehiculo.getPotenciaHP(2500, 480);
```

## Métodos Instancia vs Métodos Estáticos

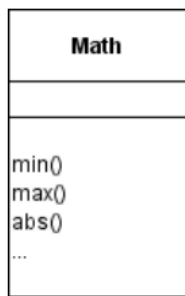
**Los métodos de Instancia** son métodos que se requiere que se cree un objeto de la clase para poder ser invocados. Los métodos de Instancia:

- Pertenecen a un objeto de una clase, no a la clase, es decir solo pueden llamarse tras crear el objeto de dicha clase
- Cada objeto individual creado a partir de una clase tiene su propia copia de los métodos de instancia de esa clase.
- Pueden ser sobrescritos (override).



**Los métodos estáticos** (o de Clase) son los que se pueden llamar sin crear un objeto de clase. Están referenciados por el nombre de la clase en sí o la referencia al objeto de la clase. Los métodos Estáticos:

- Solo pueden llamar directamente a otros métodos estáticos.
- Solo pueden usar variables de instancia estática
- No tiene referencia `this`.



```
public static int min(int a, int b){  
    //devuelve el mínimo de a y b  
}
```

`Math.min(33, 10)`

No hay variables de instancia: se usa el nombre de la clase en vez de usar el nombre de una variable de referencia.

Los métodos Estáticos:

- Están asociados a la clase en la que residen, es decir, se pueden invocar incluso sin crear una instancia de la clase.
- Están diseñados con el objetivo de que sean compartidos entre todos los objetos de una misma clase.
- No pueden ser sobrescritos, pero sí sobrecargados(overloaded).

Comparativa Métodos de Instancia/Método Estáticos:

Los métodos de instancia puede acceder a los métodos de instancia y a las variables de instancia directamente.

Los métodos de instancia puede acceder a variables estáticas y métodos estáticos directamente.

Los métodos estáticos pueden acceder a las variables estáticas y a los métodos estáticos directamente.

Los métodos estáticos no pueden acceder directamente a los métodos de instancia y las variables de instancia. Deben usar referencia a un objeto.

Los método estáticos no puede usar la palabra clave `this`, ya que no hay ninguna instancia a la que 'this' haga referencia.

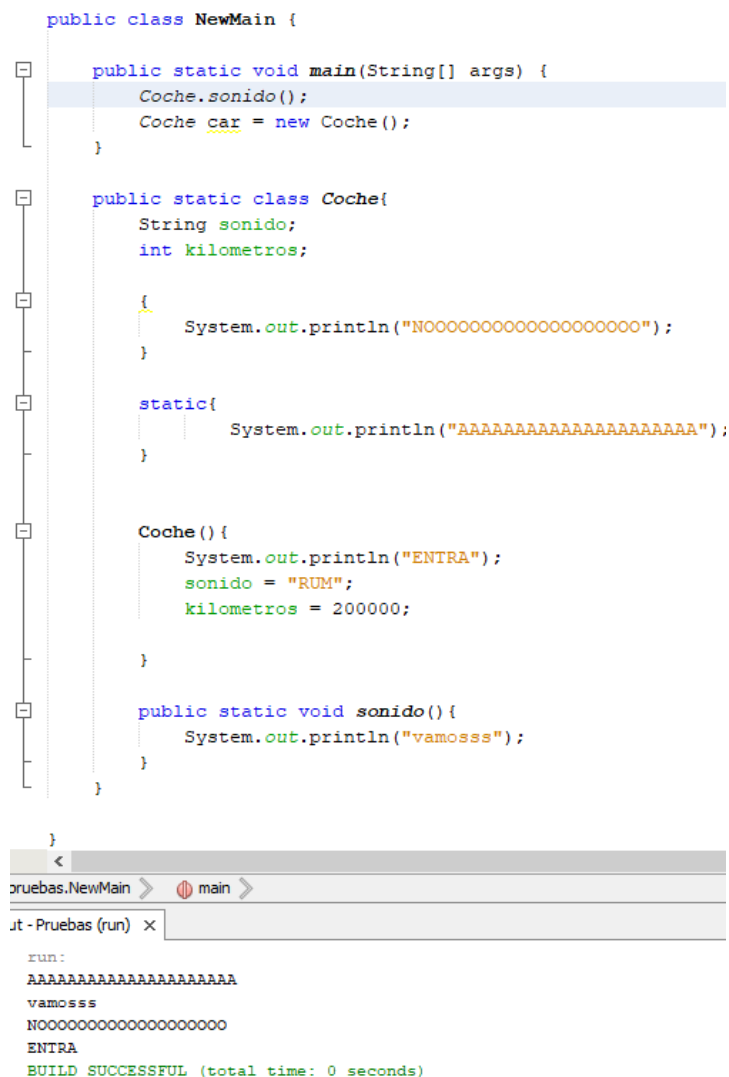
## Bloques estáticos vs Bloques no estáticos

Un **bloque estático** es un bloque que contiene código que se ejecuta en el tiempo de carga de la clase (se ejecuta antes de que la clase se pueda usar para cualquier otro propósito). Se usa la palabra clave `static`. En él solo se puede acceder a variables estáticas de la clase.

Este bloque solo se accede la primera vez que se carga la clase

```
1 static{
2     // código para el bloque estático
3 }
```

Un **bloque no estático** es un bloque que se ejecuta cuando se crea un objeto y se ejecuta antes del constructor. Se ejecuta cada vez que se crea un objeto de la clase.



```
public class NewMain {
    public static void main(String[] args) {
        Coche.sonido();
        Coche car = new Coche();
    }

    public static class Coche{
        String sonido;
        int kilometros;

        {
            System.out.println("Noooooooooooooooooooo");
        }

        static{
            System.out.println("AAAAAAAAAAAAAAAAAAAA");
        }

        Coche() {
            System.out.println("ENTRA");
            sonido = "RUM";
            kilometros = 200000;
        }

        public static void sonido() {
            System.out.println("vamosss");
        }
    }
}
```

pruebas.NewMain > main >

ut - Pruebas (run) x

run:

```
AAAAAAAAAAAAAAAAAAAA
vamosss
Noooooooooooooooooooo
ENTRA
BUILD SUCCESSFUL (total time: 0 seconds)
```

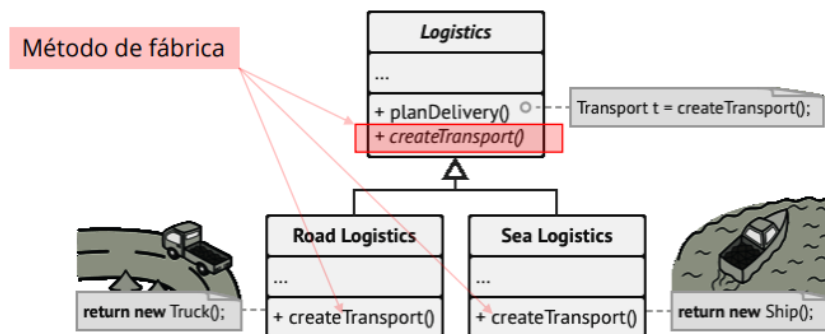
## **Patrones de Diseño**

**Los patrones de diseño son soluciones que agrupan, almacenan y resumen el conocimiento de los expertos en el desarrollo del software que sirven de base para solucionar nuevos problemas. Se identifican las partes del problema que son similares a otros problemas que ha sido solucionados con estos patrones.**

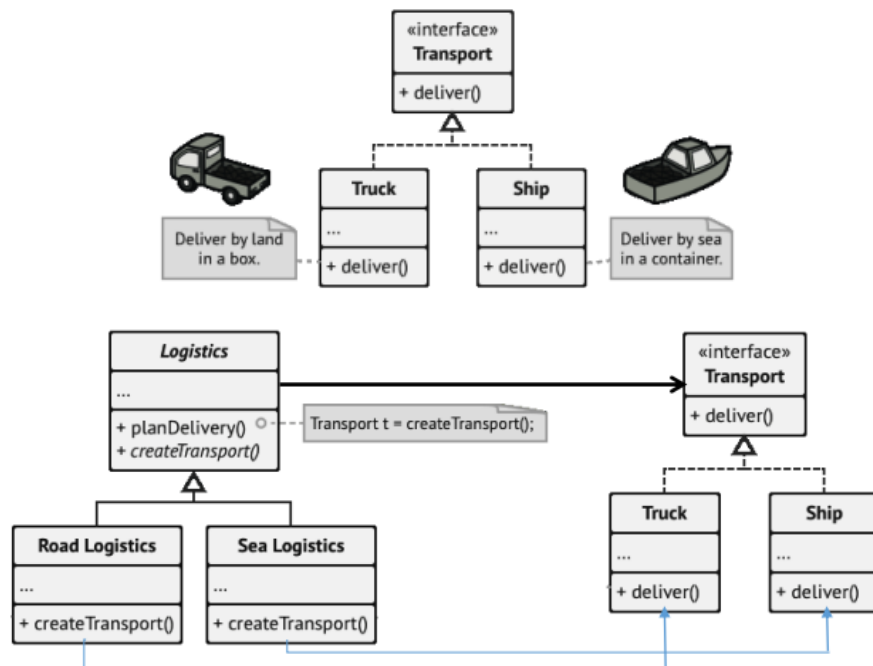
## Factory Method (patrón de diseño de creación)

Se trata de un patrón de diseño que se basa en que una clase delegue la creación de las instancias en un método de la clase. La llamada al constructor se realiza dentro del método.

Esto nos puede ser útil cuando tenemos el método de creación de instancia sobrescrito en varias subclases y de esta forma cambiar los tipos de objetos que se crean. Por ejemplo

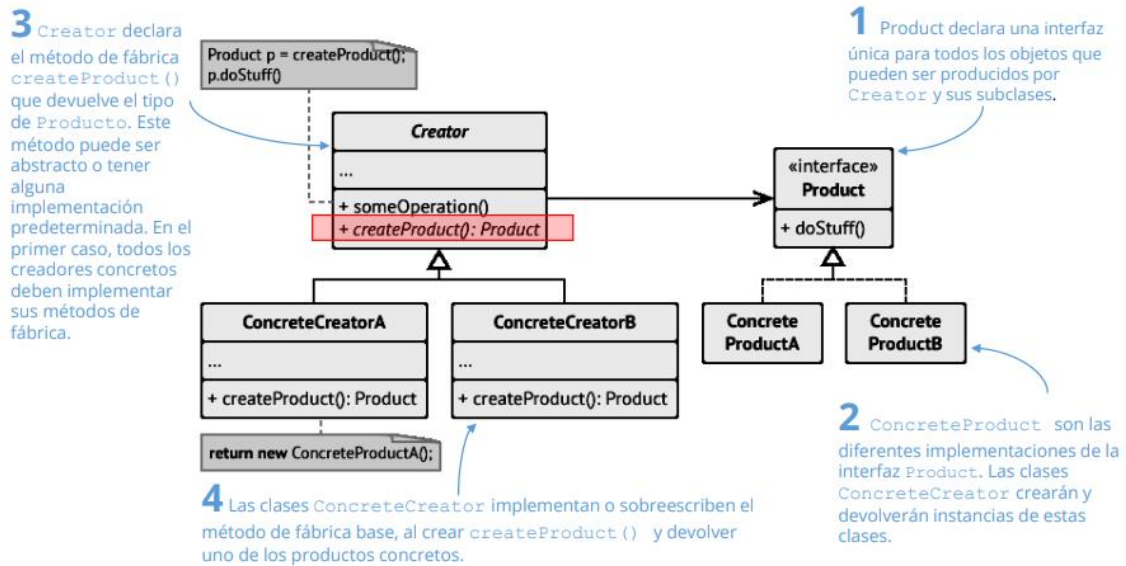


El único problema que aparece con esto es que todos los objetos que se crean deben de tener una interfaz común. Para poder hacer referencia a todos haciendo uso de la interfaz y los métodos de la interfaz.





## Diagrama de Clases genérico del Patrón de Diseño *Factory Method*



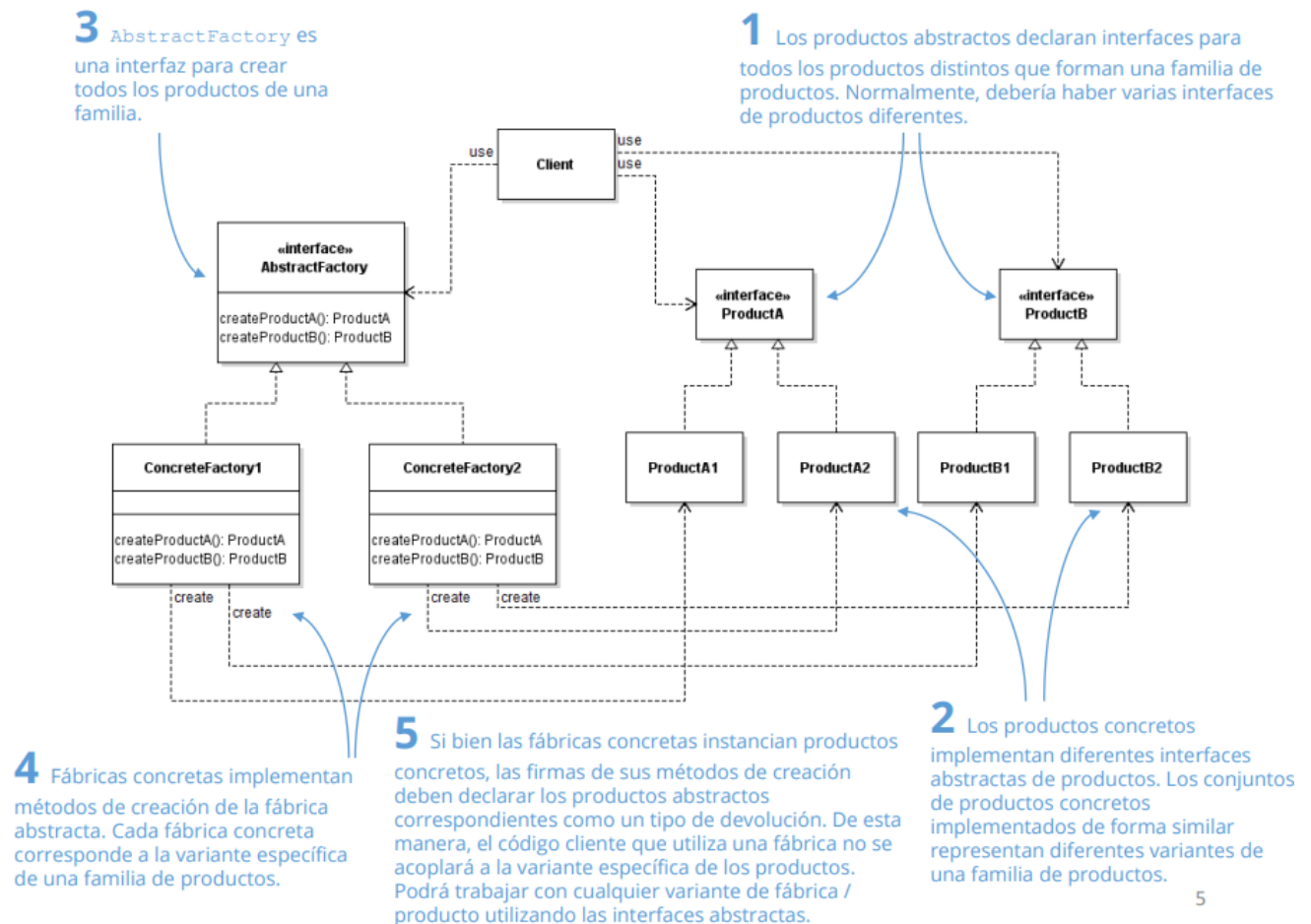
## Abstract Factory (patrón de diseño de creación)

Se trata de un patrón de diseño que nos permite producir familias de objetos relacionados sin especificar sus clases concretas.

Primero tendremos una interfaz por cada producto a crear, la cual van a tener que implementar cada una de las variantes de cada producto. (*Silla, Mesa, Sofa*).

Luego tendremos que crear una interfaz base que declara métodos para crear todos los productos que forman una familia de productos. (*createSilla(), createMesa(), createSofa()*).

Finalmente implementamos cada una de las clases que van a implementar los métodos de creación para los productos de un tipo en particular. (*IKEAFactory(IKEASilla, IKEASofa, IKEAMesa)*).

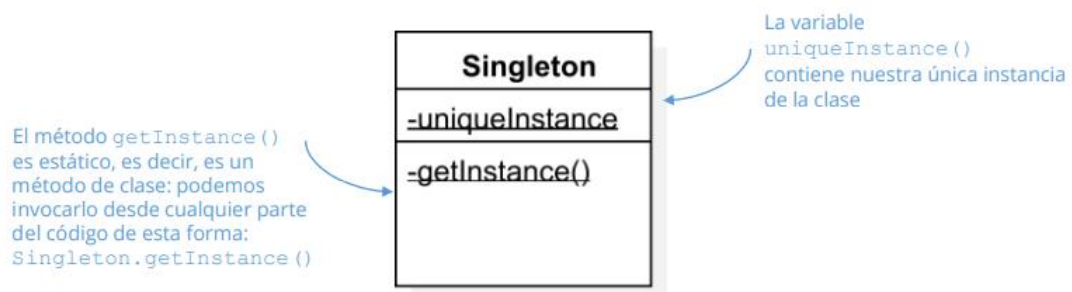


5

## Singleton (patrón de diseño de creación)

En el diseño orientado a objetos, es muy importante para algunas clases tener una sola instancia.

Para esto exista el **patrón de diseño singleton**, el cual garantiza que una clase solo tenga una instancia y le proporcione un punto de acceso global.



Para aplicar el patrón de diseño Singleton, debemos de:

1. Declara el constructor de la clase como privado, para evitar que se hagan otras instancias de la clase.
2. Crear una instancia de la clase o bien durante la carga de la clase en una variable estática o campo estático, o bajo un método estático que primero verifique no existe instancia y que cree la instancia si esta no existe. Finalmente tendremos un método estático el cual nos devolverá la instancia de la clase.

### Opciones de implementación del patrón singleton:

- Usando una **variable static final**:

#### Opción 1:

```
public class EagerSingleton {  
  
    // Constructor privado para prevenir que otros instancien la clase.  
    private EagerSingleton() {}  
  
    // Se crea una instancia de la clase en el tiempo de carga de la misma.  
    private static final EagerSingleton instance = new EagerSingleton();  
  
    // Se proporciona un punto de acceso global a la instancia.  
    public static EagerSingleton getInstance() {  
        return instance;  
    }  
}
```

Tiene la desventaja de que se crea la instancia de la clase, aunque no se haga uso de esta.

- Usando un bloque static:

#### Opción 2:

```
public class EagerStaticBlockSingleton {  
    private static final EagerStaticBlockSingleton instance;  
  
    // Constructor privado, igual que en la Opción 1  
    private EagerStaticBlockSingleton() {}  
  
    // Se crea la única instancia de la clase en un bloque estático  
    static {  
        try {  
            instance = new EagerStaticBlockSingleton();  
        } catch (Exception e) {  
            throw e;  
        }  
    }  
  
    // Igual que en la Opción 1  
    public static EagerStaticBlockSingleton getInstance() {  
        return instance;  
    }  
}
```

Funciona porque el bloque static solo se ejecuta la primera vez que se carga la clase. Tiene la ventaja de manejar excepciones. Tiene la desventaja de que se crea la instancia de la clase, aunque no se haga uso de esta.

- Comprobando si existe una instancia creada:

#### Opción 3:

```
public class LazySingleton {  
  
    private static LazySingleton instance;  
  
    private LazySingleton() {}  
  
    // Inicialización perezosa  
    public static synchronized LazySingleton getInstance() {  
        if(instance == null) {  
            instance = new LazySingleton();  
        }  
        return instance;  
    }  
}
```

Comprueba si la instancia está creada, si no lo está la crea, pero si existe una instancia la devuelve. Se añade la palabra clave `synchronized` para si hay varios hilos trabajando simultáneamente solo puede acceder uno simultáneamente a este método.

- Utilizando una clase interna estática para crear la instancia de la clase.

#### Opción 4:

```
public class LazyInnerClassSingleton {  
  
    private LazyInnerClassSingleton() {}  
  
    // Esta clase interna solo se carga después de que getInstance() se llame.  
    private static class SingletonHelper {  
        private static final LazyInnerClassSingleton INSTANCE =  
            new LazyInnerClassSingleton();  
    }  
  
    public static LazyInnerClassSingleton getInstance() {  
        return SingletonHelper.INSTANCE;  
    }  
}
```

Esta solución es segura (desde el punto de vista de subprocesos) y no requiere ninguna sincronización. Es el enfoque más eficiente entre todas las implementaciones del patrón de diseño Singleton.

## Iterator (Patrón de diseño de comportamiento)

Este patrón de diseño Iterator se usa debemos recorrer colecciones de elementos, independientemente de su implementación.

Ejemplo: Debemos de recorrer dos menús de establecimientos de comida distintos que están implementados de formas distintas. PE: Uno con un array y otro con un ArrayList.

2. Para imprimir los ítems del menú de McDonald's debemos iterar sobre un **ArrayList**:

```
for (int i = 0; i < hamburguesasItems.size(); i++){  
    MenuItem menuItem = (MenuItem) hamburguesasItems.get(i);  
    ...  
}
```

y para imprimir los ítems del menú de Telepizza, debemos iterar sobre un **Array**:

```
for (int i = 0; i < pizzasItems.length(); i++) {  
    MenuItem menuItem = pizzasItems[i];  
    ...  
}
```

8

La solución para recorrer estas dos colecciones independientemente de su implementación es encapsulándolos. Para lograr esto hacemos uso del patrón de diseño iterator.

El Patrón de diseño Iterator se apoya en una interfaz Iterator el cual encapsula en cada una de las implementaciones de dicha interfaz la forma de recorrer dicha colección, pero siempre respetando los métodos de la interfaz, la cual está compuesta por:

- El método **hasNext()** el cual nos dice si hay más elementos en la colección sobre los que iterar.
- El método **next()** el cual devuelve el siguiente objeto en la colección.

```

        public class HamburguesasMenuIterator implements Iterator {
            ArrayList<MenuItem> items;
            int position = 0;

            public HamburguesasMenuIterator(ArrayList<MenuItem> items) {
                this.items = items;
            }

            public boolean hasNext() {
                if (position >= items.size()) {
                    return false;
                } else {
                    return true;
                }
            }

            public Object next() {
                MenuItem item = items.get(position);
                position = position + 1;
                return item;
            }
        }

public class McDonaldsMenu {
    ArrayList<MenuItem> menuItems;

    public McDonaldsMenu() {
        menuItems = new ArrayList<>();

        addItem("Beacon SmokeHouse Burger","Carne, tocino ahumado, cebolla,cheddar",false,3.75);
        addItem("Big Mac","Carne, pepinillo,lechuga,cheddar",false,3.90);
        ...
    }

    public void addItem(String name, String description,boolean vegetarian, double price) {
        MenuItem item = new MenuItem(name, description, vegetarian, price);
        menuItems.add(item);
    }

    public ArrayList<MenuItem> getMenuItems() {
        return menuItems;
    }

    public Iterator createIterator() {
        return new HamburguesasMenuIterator(menuItems);
    }

    // OTROS MÉTODOS DEL MENÚ DE HAMBURGUESAS
}

        public class PizzasMenuIterator implements Iterator {
            MenuItem[] items;
            int position = 0;

            public DinerMenuIterator(MenuItem[] items) {
                this.items = items;
            }

            public Object next() {
                MenuItem menuItem = items[position];
                position = position + 1;
                return menuItem;
            }

            public boolean hasNext() {
                if (position >= items.length || items[position] == null) {
                    return false;
                } else {
                    return true;
                }
            }
        }
    }
}

```

```

public class TelepizzaMenu {
    static final int MAX_ITEMS = 2;
    int numberOfItems = 0;
    MenuItem[] menuItems;

    // CONSTRUCTOR
    // MÉTODO addItem()

    public MenuItem[] getMenuItems() {
        return menuItems;
    }

    public Iterator createIterator() {
        return new PizzasMenuIterator(menuItems);
    }

    // OTROS MÉTODOS DEL MENÚ DE PIZZAS
}

```

No necesitaremos el método getMenuItems() porque, entre otras razones, expone nuestra implementación interna.

```

public class Camarero {
    McDonaldsMenu mcdonaldsMenu;
    TelepizzaMenu telepizzaMenu;

    public Camarero(McDonaldsMenu mcdonaldsMenu, TelepizzaMenu telepizzaMenu) {
        this.mcdonaldsMenu = mcdonaldsMenu;
        this.telepizzaMenu = telepizzaMenu;
    }

    public void imprimeMenu() {
        Iterator hamburguesasIterator = mcdonaldsMenu.createIterator();
        Iterator pizzasIterator = telepizzaMenu.createIterator();
        System.out.println("MENU\n---\nHAMBURGUESAS");
        imprimeMenu(hamburguesasIterator);
        System.out.println("\nPIZZAS");
        imprimeMenu(pizzasIterator);
    }

    private void imprimeMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = (MenuItem)iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " - ");
            System.out.println(menuItem.getDescription());
        }
    }

    // OTROS MÉTODOS
}

```

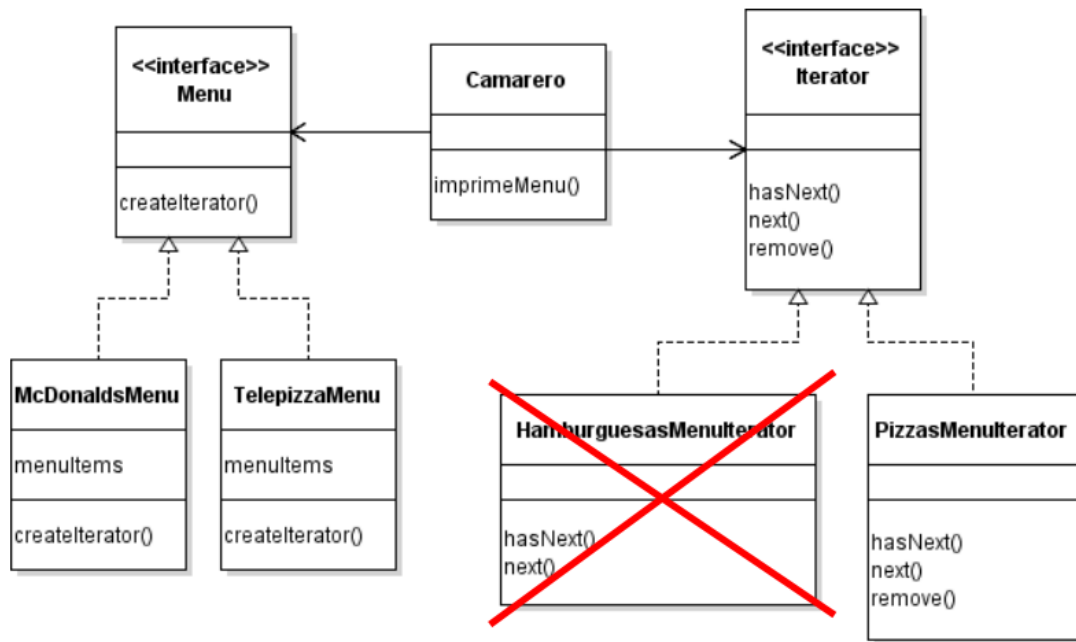
imprimeMenu() crea dos iteradores: uno por cada menú.

Llamada al método sobrecargado imprimeMenu() con cada iterador.

El método sobrecargado usa Iterator para recorrer los elementos del menú e imprimirlos.

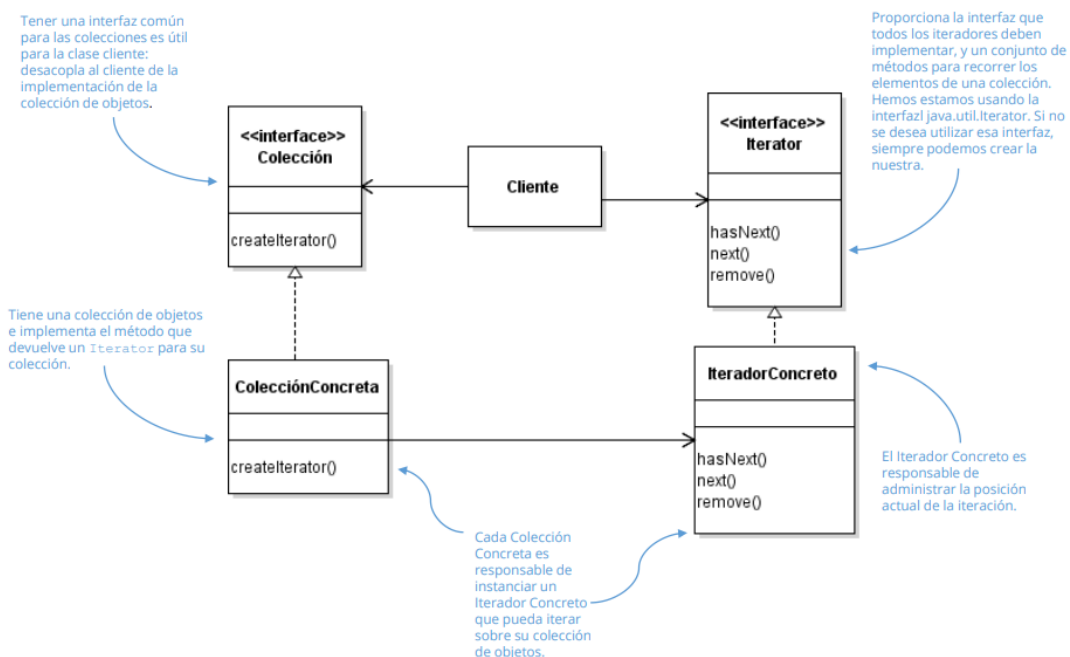
17

**Esto se podría mejorar, ya que realmente ya ArrayList tiene su propio método iterator(). Por lo que solo debemos de utilizar la implementación del Iterator para el apollado en el Array.**



Por lo tanto:

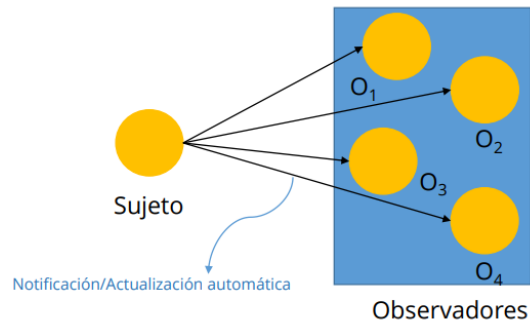
**El Patrón de diseño Iterator nos proporciona una forma de recorrer los elementos/objetos de una colección de forma secuencial sin exponer su implementación interna.** Es decir, este patrón ofrece una forma de recorrer los elementos de una colección sin tener que saber cómo se representan las cosas internamente.





## Observer (Patrón de diseño de comportamiento)

Este patrón de diseño define una dependencia “uno a muchos” entre los objetos, de modo que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.

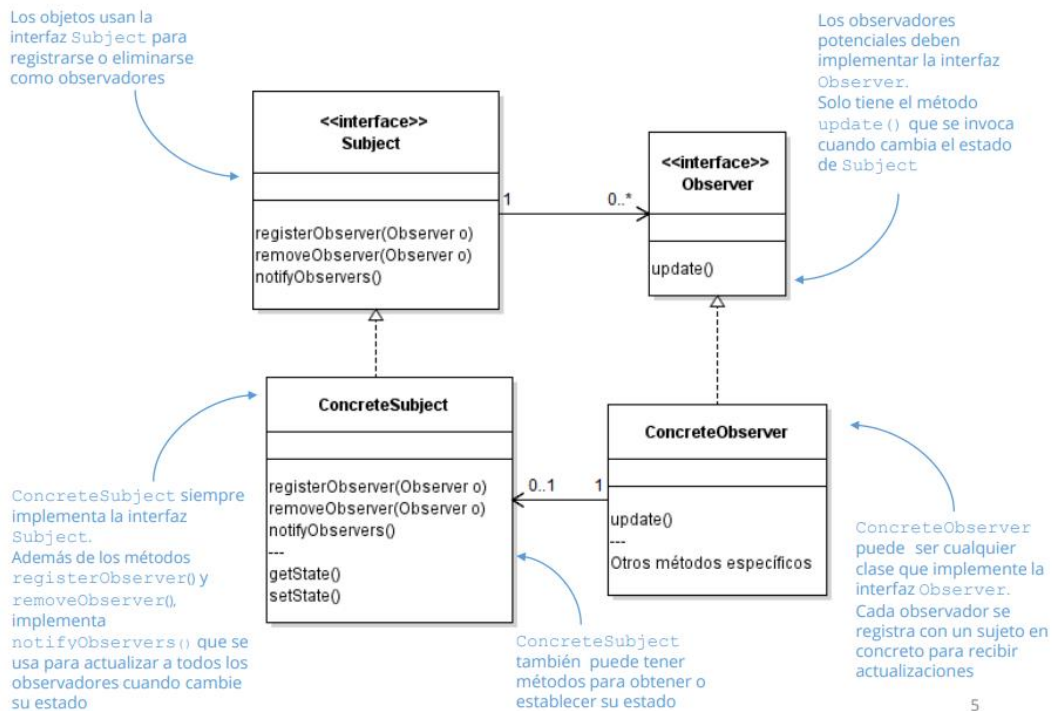


El Sujeto y los Observadores definen una relación “uno a muchos”.

- Los Observadores dependen del sujeto, de modo que cuando el estado del Sujeto cambia, los observadores reciben una notificación.
- Dependiendo del estilo de notificación, los Observadores también pueden actualizarse con nuevos valores.

La forma de implementar el patrón Observer es en base al diseño de clases que incluyen las interfaces **Subject** y **Observer**.

## Diagrama de Clases genérico del Patrón de Diseño *Observer*



5

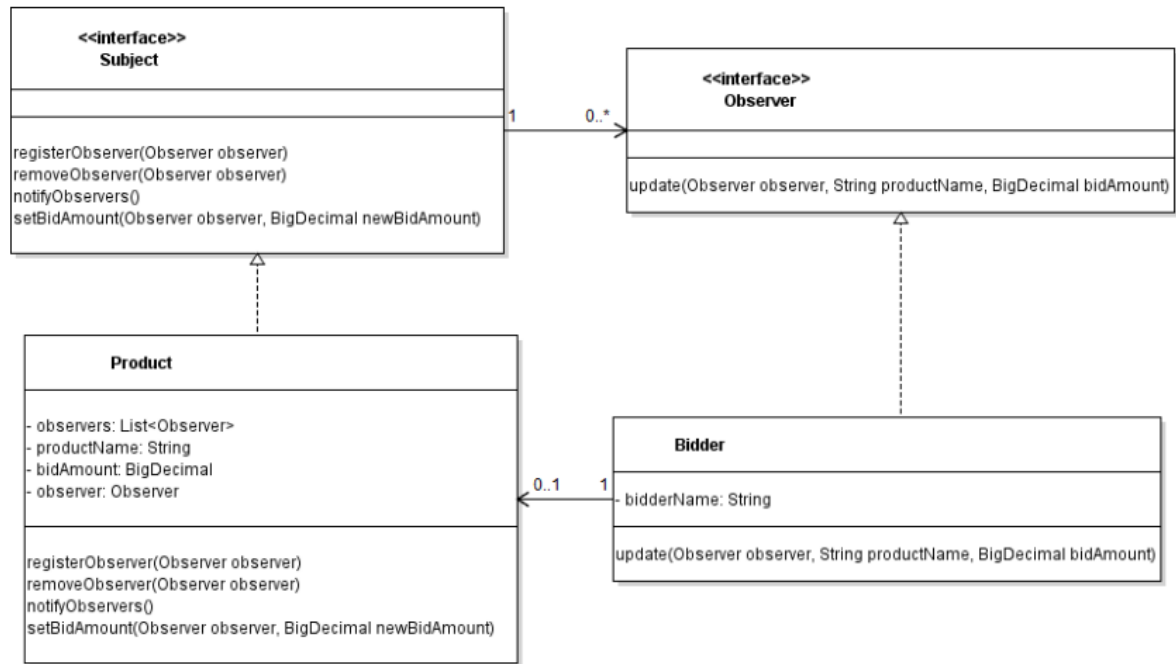
### EJEMPLO:

**Subject:** Será un producto en subasta. Su estado (precio) cambiará cuando se haga una nueva oferta. Se trata de la interfaz *Subject* que declara el contrato de definición de productos que nos permitirá añadir o eliminar objetos *Observer*.

**ConcreteSubject:** Modelaremos los productos reales mediante la creación de la clase concreta *Product* que implementa la interfaz *Subject*. La clase *Product* es la encargada de notificar a sus *Observers* cuando cambia su estado.

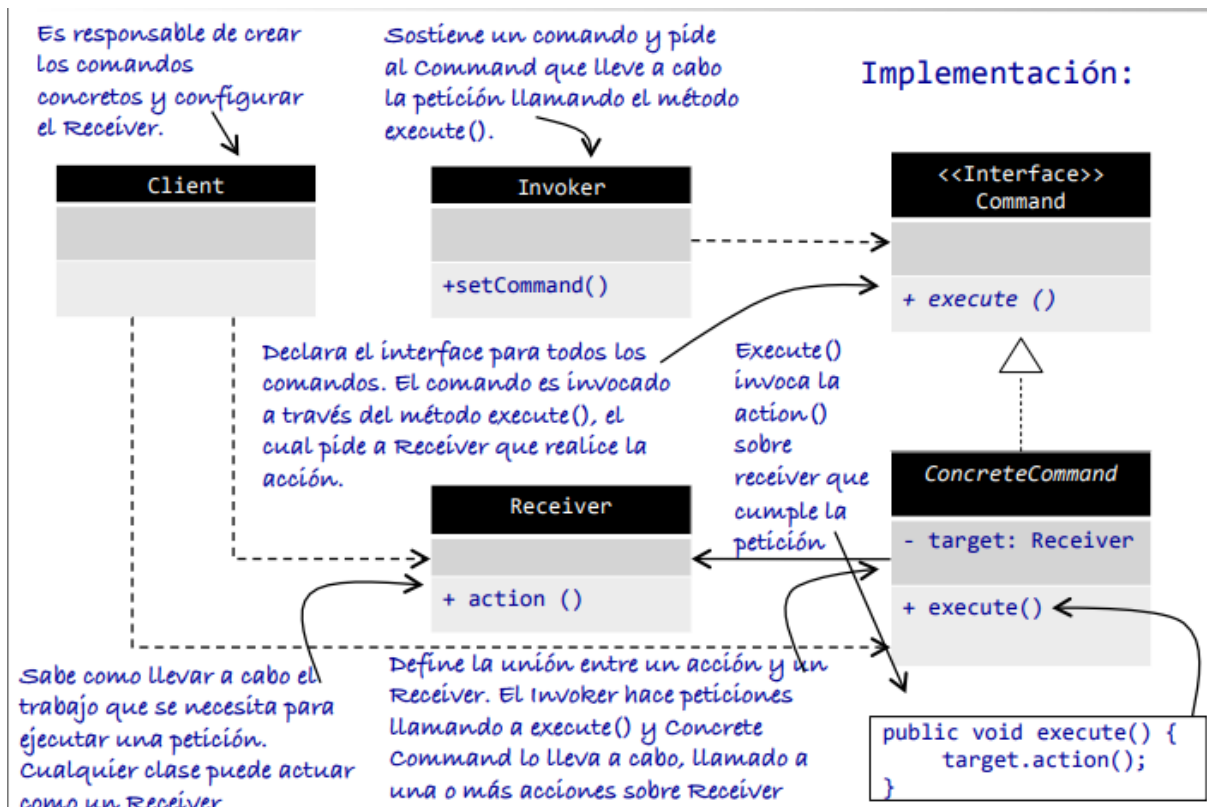
**Observer:** Proporciona una interfaz para los objetos que deben ser notificados de los cambios de un sujeto.

**ConcreteObserver:** Modelaremos los observadores reales como objetos de la clase *Bidder*(Postor) que implementa la interfaz *Observer* para recibir notificaciones de *Subject* y mantener su estado consistente con el de *Subject*.



## Command (Patrón de diseño de Comportamiento)

Se trata de un patrón de diseño el cual se basa en encapsular un comando en un objeto de tal forma que pueda ser pasado, almacenado y devuelto igual que cualquier objeto.



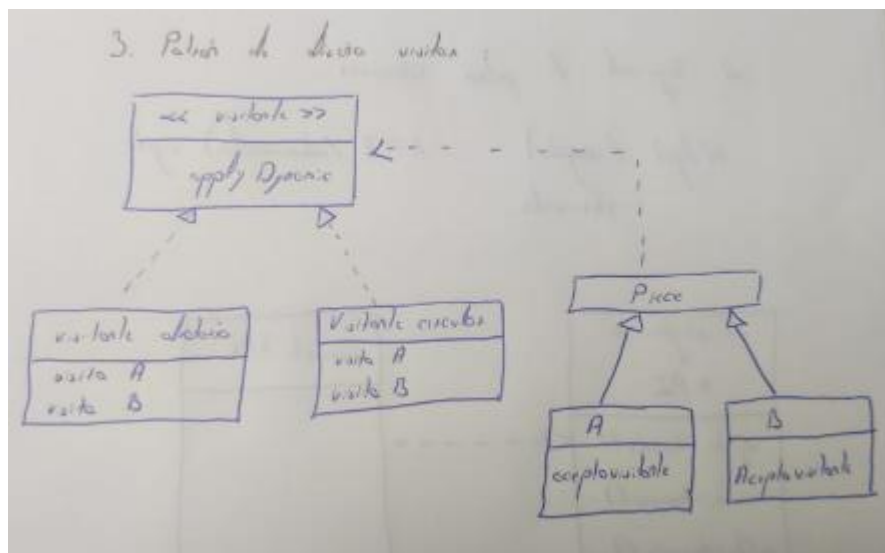
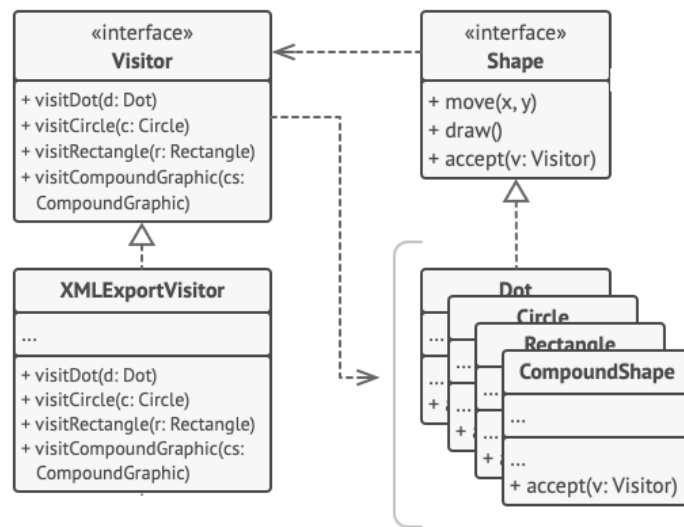
## Visitor (Patrón de diseño de Comportamiento)

El patrón de diseño Visitor permite agregar operaciones adicionales a los objetos sin cambiar las clases de los objetos sobre los que opera. Es muy útil cuando se tienen varias clases con estructuras similares y se quiere realizar una operación diferente sobre cada una de ellas.

Se consta de una interfaz visitante, que define la operación a realizar y una clase concreta para cada tipo de objeto. Los objetos aceptan visitantes y llaman a la operación apropiada en el visitante. De esta manera, las operaciones se encapsulan en las clases de los visitantes, en lugar de en las clases de los objetos visitados.

El patrón Visitor sugiere que coloques el nuevo comportamiento en una clase separada llamada *visitante*, en lugar de intentar integrarlo dentro de clases existentes. El objeto que originalmente tenía que realizar el comportamiento se pasa ahora a uno de los

métodos del visitante como argumento, de modo que el método accede a toda la información necesaria contenida dentro del objeto.

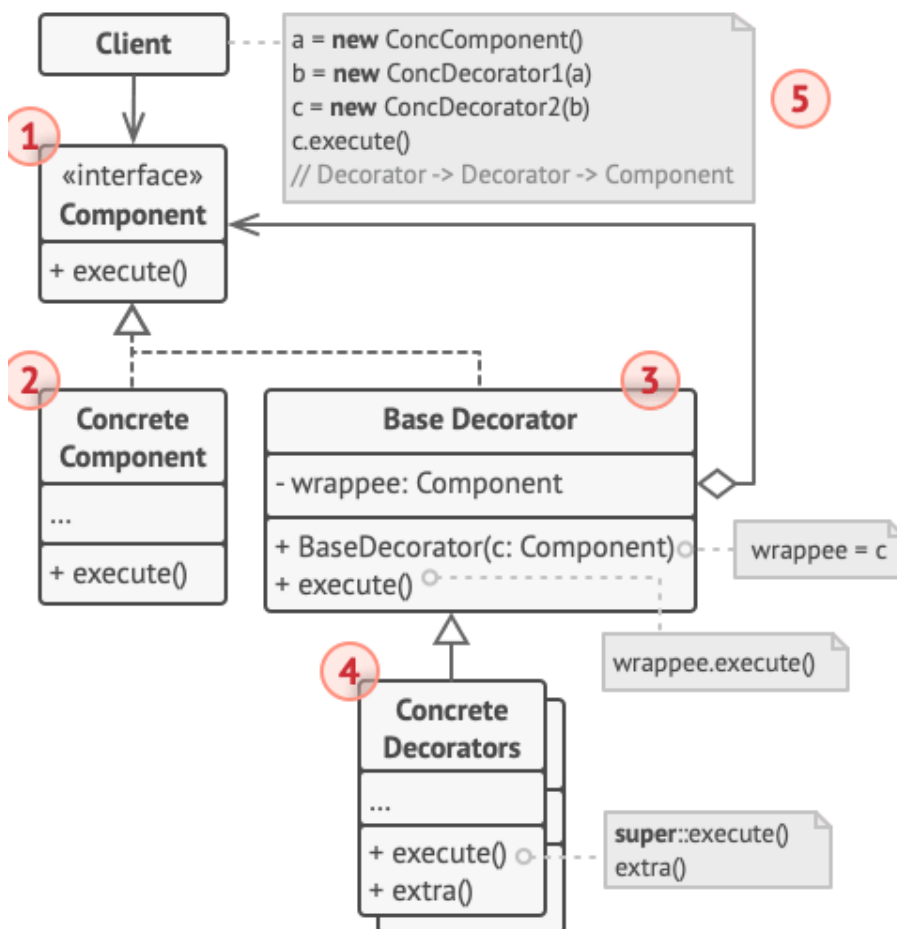


## Decorator (Patrones de diseño estructurales) (Wrapper)

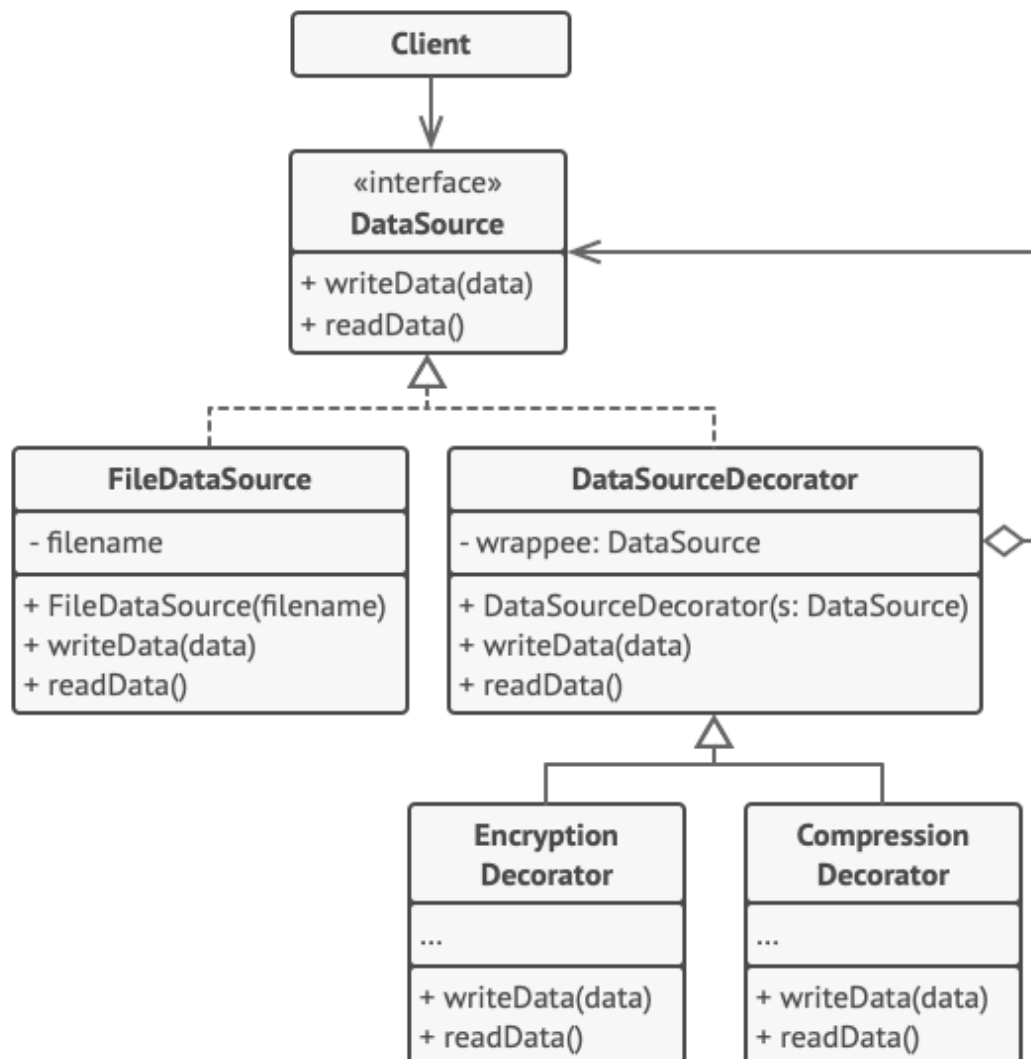
El patrón decorator permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.

Para hacer esto tenemos que hacer que el objeto decorador implemente la misma interfaz que el objeto concreto.

Se basa en el uso de otras clases llamadas clases decoradoras para añadirle funcionalidades a una clase ya creada sin tener que alterarla. Para hacer esto debemos de crear la clase haciendo que contenga el mismo grupo de métodos que la clase a decorar y que le delegue todas las solicitudes que recibe. No obstante, la clase decoradora puede alterar el resultado haciendo algo antes o después de pasar la solicitud al objetivo.

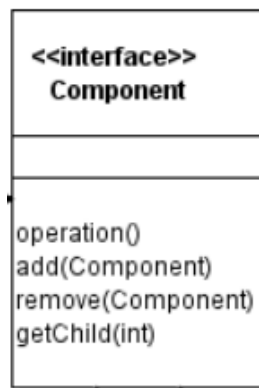


1. El **Componente** declara la interfaz común tanto para wrappers como para objetos envueltos.
2. **Componente Concreto** es una clase de objetos envueltos. Define el comportamiento básico, que los decoradores pueden alterar.
3. La clase **Decoradora Base** tiene un campo para referenciar un objeto envuelto. El tipo del campo debe declararse como la interfaz del componente para que pueda contener tanto los componentes concretos como los decoradores. La clase decoradora base delega todas las operaciones al objeto envuelto.
4. Los **Decoradores Concretos** definen funcionalidades adicionales que se pueden añadir dinámicamente a los componentes. Los decoradores concretos sobrescriben métodos de la clase decoradora base y ejecutan su comportamiento, ya sea antes o después de invocar al método padre.
5. El **Cliente** puede envolver componentes en varias capas de decoradores, siempre y cuando trabajen con todos los objetos a través de la interfaz del componente.



## Composite (Patrón de diseño Estructurales)

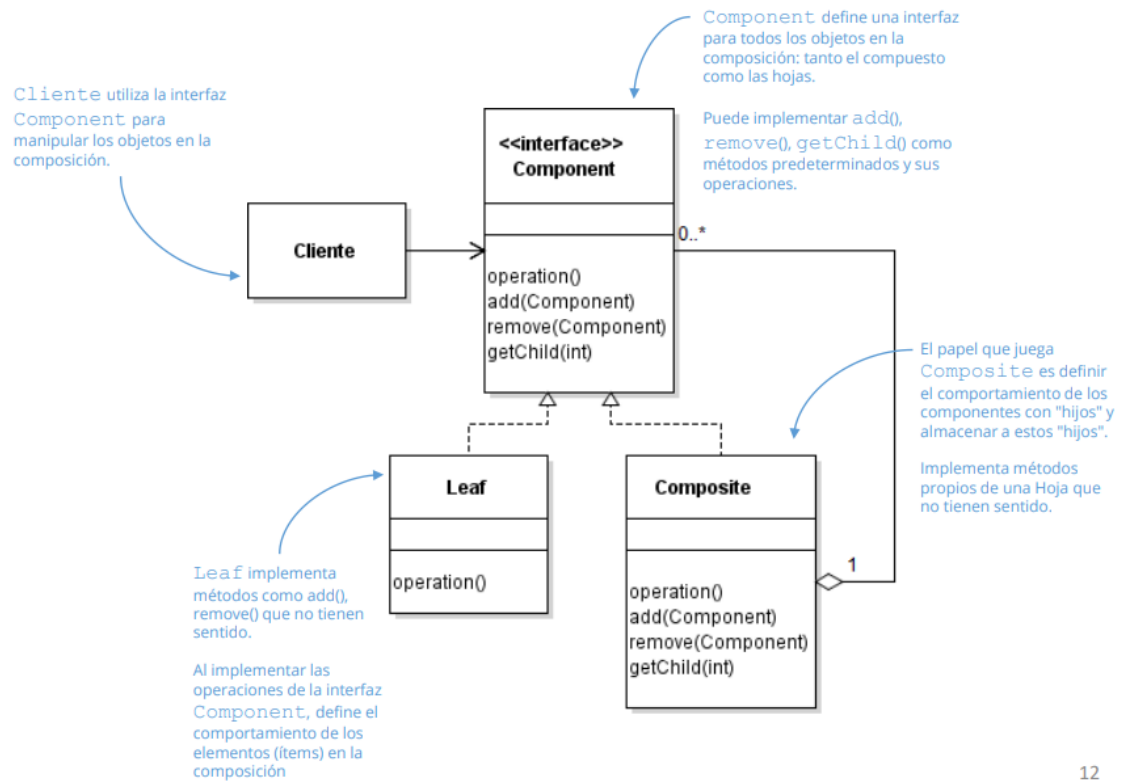
El **patrón de diseño composite** nos permite componer objetos en estructuras de árbol para representar jerarquías del tipo “parte-todo” (sigue la teoría de que las partes forman el todo). Composite **permite a los clientes tratar los objetos individuales y a las composiciones de objetos de la misma forma**. Es decir, nos permite crear una jerarquía flexible en estructura de árbol, en la cual todos los elementos de dicha jerarquía **tienen la misma interfaz**.



Se sigue la teoría de que las partes forman el todo, teniendo en cuenta de que las partes pueden tener dentro otras partes. Por lo tanto, nuestra estructura dos tipos de componentes:

- **Compuestos** por otros tipos de componentes:
  - Los cuales tendrá métodos que vienen definidos en la interfaz para añadir y eliminar otros componentes.
  - También implementará métodos inútiles que son propios de una hoja, pero deben de ser implementados ya que tenemos la misma interfaz para ambos componentes.
- **O componentes hoja:**
  - Implementa las operaciones de la interfaz Component, con la que define el comportamiento de los elementos en la composición.
  - También implementa los métodos de añadir y eliminar componentes que no tienen sentido en un nodo hoja.





12

Ejemplo:

```

package contactostest;

public interface Component {
    public void mensaje(String texto);
}
  
```

Interfaz Component

## Clase Persona

```
package contactostest;

public class Persona implements Component {
    private final String nombrePersona;
    private final long telefono;

    public Persona(String nombre, long telefono) {
        this.nombrePersona = nombre;
        this.telefono = telefono;
    }

    public String getName() {
        return nombrePersona;
    }

    public long getTelefono() {
        return telefono;
    }

    @Override
    public void mensaje(String texto) {
        System.out.println("MENSAJE A: " + nombrePersona
            + "(" + telefono + ")");
        System.out.println(texto);
    }
}
```

## Clase Grupo

```
package contactostest;

import java.util.ArrayList;
import java.util.List;

public class Grupo implements Component {
    private final String nombreGrupo;
    List<Component> grupo = new ArrayList<>();

    public Grupo(String nombreGrupo) {
        this.nombreGrupo = nombreGrupo;
    }

    public String getNombreGrupo() {
        return nombreGrupo;
    }

    @Override
    public void mensaje(String texto) {
        for (Component c : grupo) {
            System.out.println("MENSAJE A GRUPO: " + nombreGrupo);
            c.mensaje(texto);
        }
    }

    public void add(Component component) {
        grupo.add(component);
    }
}
```

## Principios de Diseño (SOLID)

Los principios de diseño son los encargados de guiar el desarrollo de software para que el sistema creado sea fácil de mantener y ampliar con el tiempo.

Los principios de diseño tienen el **objetivo de crear estructuras Software de Calidad:**

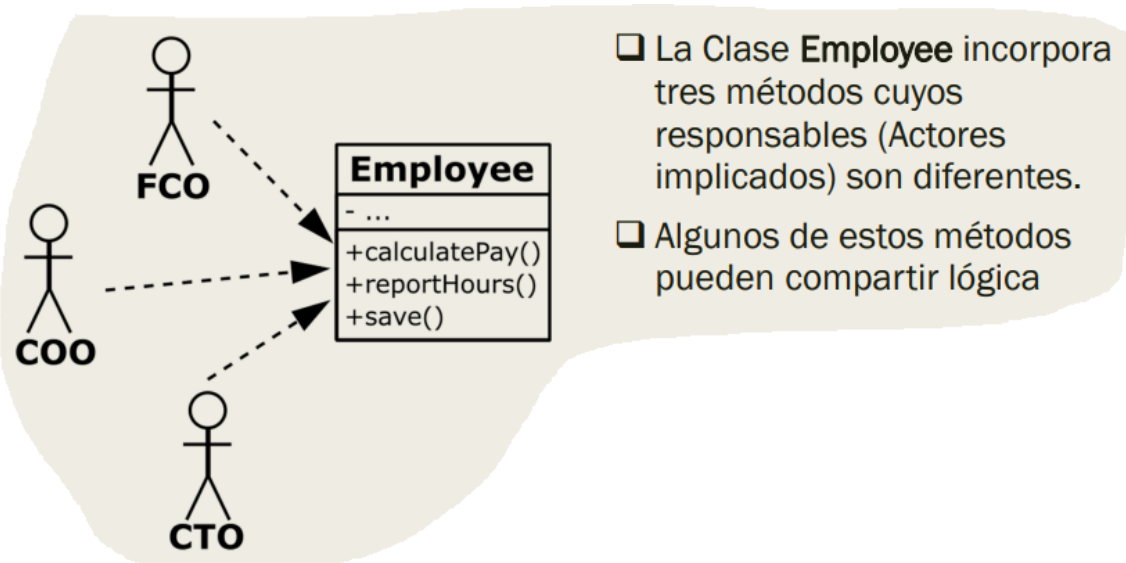
- Que toleren cambios y re combinaciones
- Que sean fáciles de comprender
- Que el sistema sea evolucionable

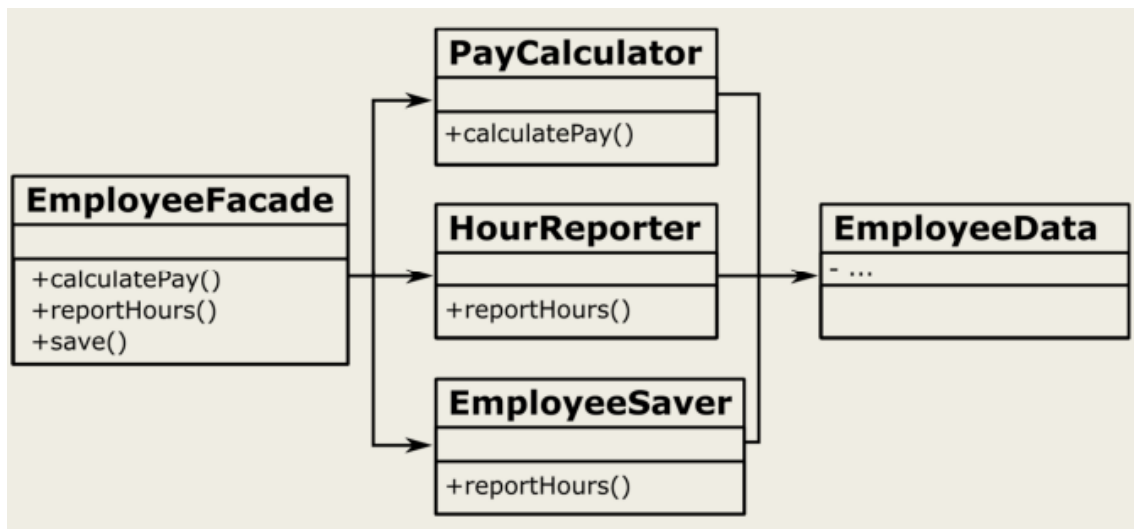
### Responsabilidad Única

El principio de diseño de la responsabilidad única se basa en que un módulo, clase o función del código deberá tener una única razón por la que cambiar, esto quiere decir que un modulo debe tener una única responsabilidad.

Para respetar este principio las funcionalidades de un modulo deben de pasar por las siguientes preguntas:

- ¿Quién? (Es el actor responsable de la funcionalidad)
- ¿Qué? (Tiene que hacer dicha funcionalidad)
- Como (Hago la lógica de la funcionalidad)





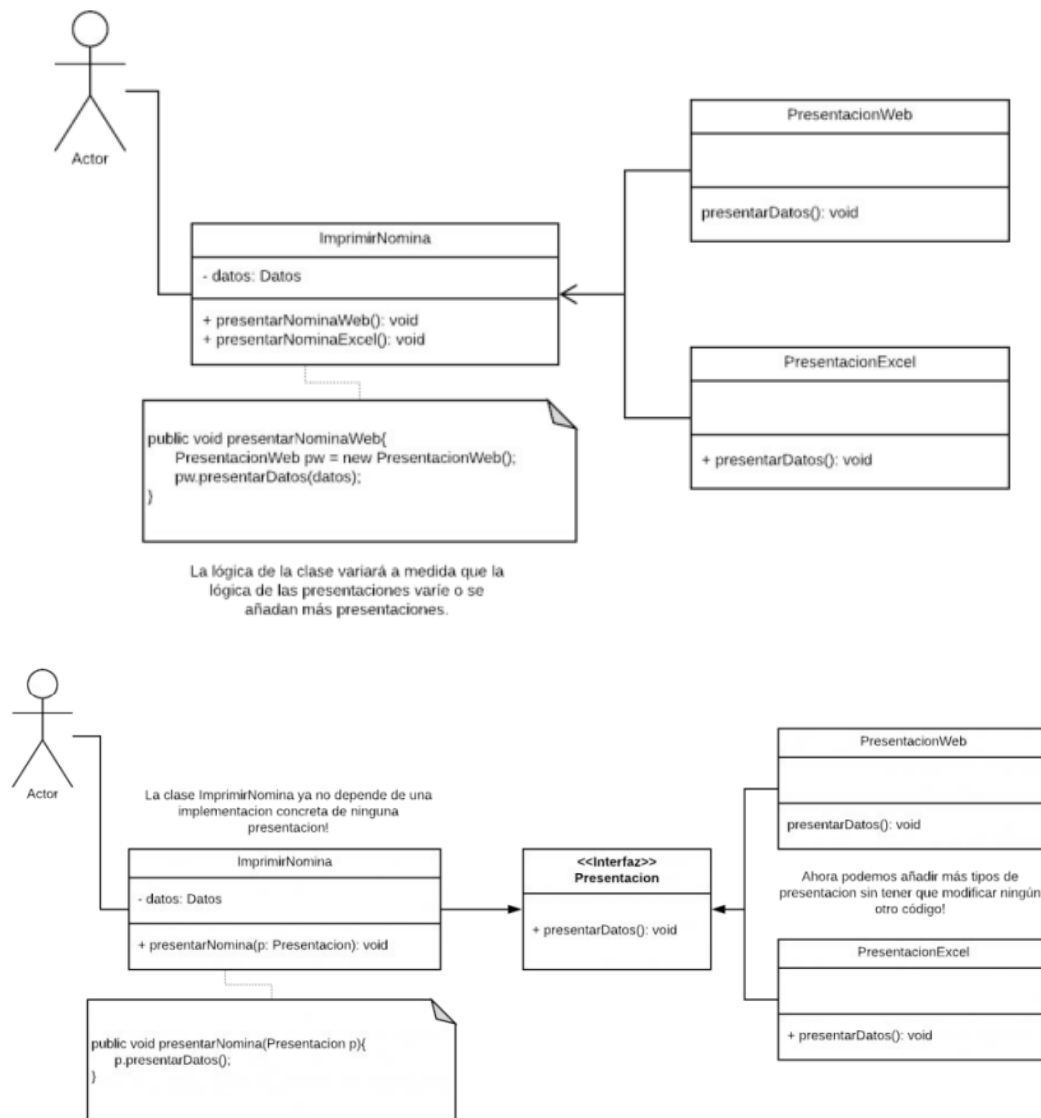
## Principio de diseño Abierto-Cerrado

El principio de diseño de Abierto y Cerrado se basa en que un módulo, clase o función del software tiene que estar abierto para su extensión, pero cerrado para su modificación. Es decir, el comportamiento de un módulo, clase o función de un software debe ser extensible, sin tener que modificar dicho módulo, clase o función.

¿Cómo se pueden cumplir ambas propiedades a la vez?

Pues realmente parecen que entran en conflicto ya que la forma más común de extender un comportamiento del software es hacer cambios en el mismo.

Pues la clave es crear abstracciones que sean fijas, como pueden ser clases bases (clases abstractas) o interfaces. Gracias a esto se pueden crear un número ilimitado de comportamientos con clases que deriven de la clase base o interfaz. Como además las clases pueden realizar su trabajo manipulando sólo la abstracción está cerrada para sus modificaciones ya que depende de dicha abstracción y simplemente el comportamiento puede ser extendido.



## Principio de diseño Sustitución de Liskov

El principio de diseño de sustitución de Liskov se basa en si por cada objeto **Obj1** del **tipo S** hay otro objeto **Obj2** del **tipo T**, y para todos los programas **P** definidos en términos de **T**, el comportamiento de **P** no cambia cuando **Obj1** sustituye a **Obj2**, por lo que **S** es un subtipo de **T**.

```

public static void main( String[] args ) {
    Rectangle rectangle = new Rectangle(10, 15);
    Square square = new Square(10);
    if (square.isSquare()) System.out.println("square is square");
    else System.out.println("square isn't square");
    f(square);
    if (square.isSquare()) System.out.println("square is square");
    else System.out.println("square isn't square");
    g(square);
}
private static void f(Rectangle rectangle) {
    rectangle.setHeight(5);
}
private static void g(Rectangle rectangle) {
    rectangle.setWidth(5);
    rectangle.setHeight(4);
    if(RectangleAreaCalculator.calculateArea(rectangle) == 20)
        System.out.println("área Ok");
    else System.out.println("área Nok");
}

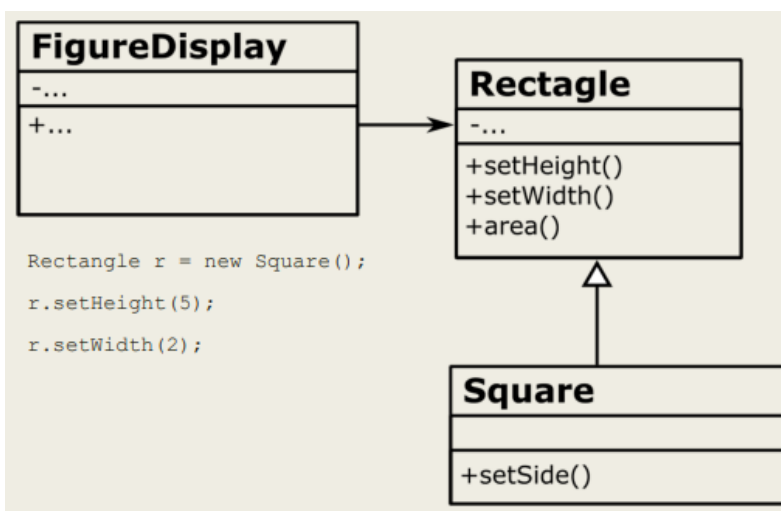
```

Cuando el comportamiento del software depende del tipo de objeto con el que se está trabajando: el software no cumple el principio de diseño de la sustitución de Liskov.

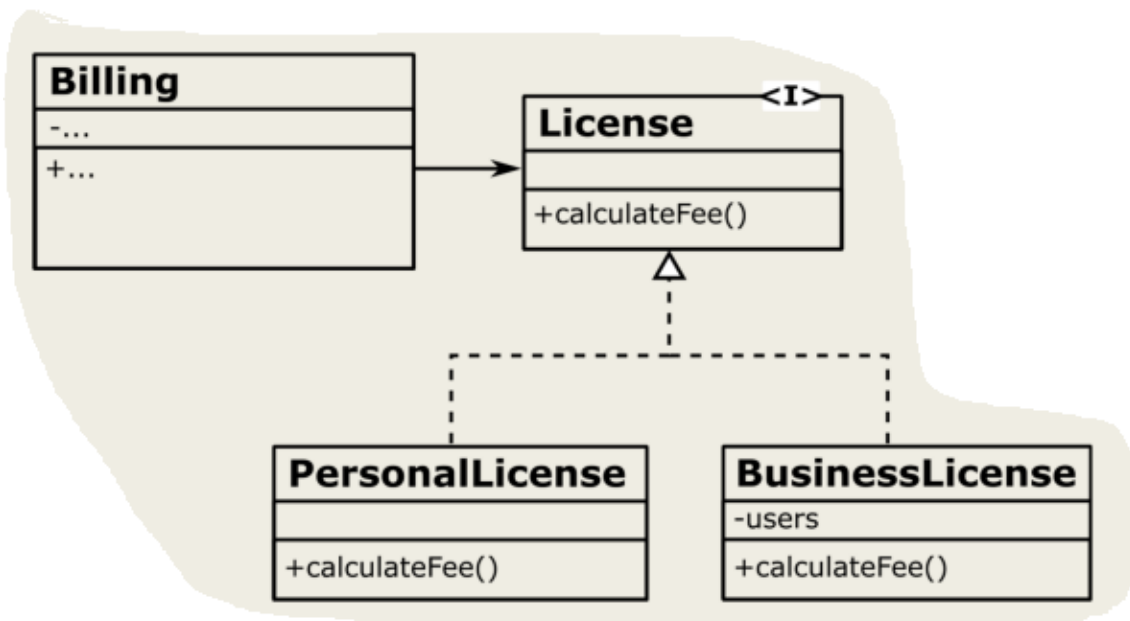
```

public class Square extends Rectangle {
    public Square(int side) {
        super(side, side);
    }
    public void setHeight(int height) {
        super.setHeight(height);
        super.setWidth(height);
    }
    public void setWidth(int width) {
        super.setHeight(width);
        super.setWidth(width);
    }
    public boolean isSquare() {
        return this.getWidth() == this.getHeight();
    }
}

```



De esta forma si se cumple el principio de sustitución de Liskov.

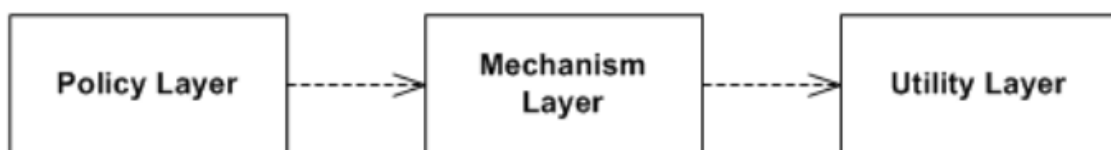


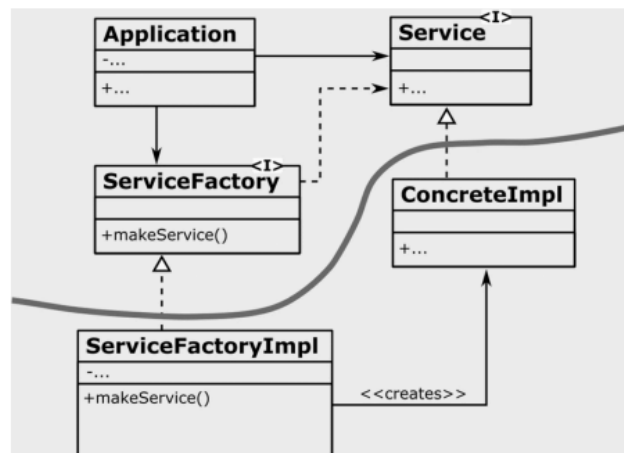
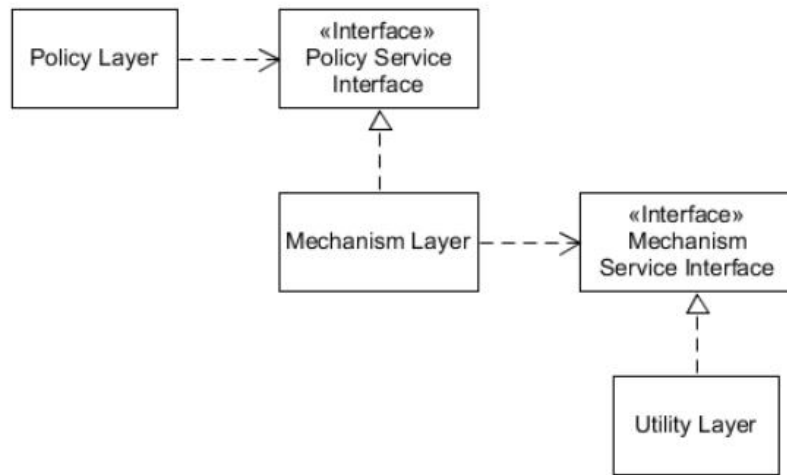
## Principio de diseño Inversión de la Dependencia

El principio de diseño de la inversión de la dependencia se basa en depender de las abstracciones como pueden ser las interfaces y no de las implementaciones de dicha clase. Los sistemas son más escalables y mantenibles si las dependencias de un módulo dependen de abstracciones y no de elementos concretos.

Esto se basa en que las abstracciones apenas requieren cambios mientras que las implementaciones concretas tienden a cambiar con más facilidad. Esto implica que todos los módulos dependan de elementos concretos sean propensos al cambio y esto dificulta el mantenimiento.

De esta forma hacemos que los módulos de alto nivel sean independientes de los detalles de implementación de los módulos de bajo nivel.





## Principio de diseño Inyección de Dependencias

El principio de diseño de la inyección de la dependencia se basa en hacer que una modulo, clase o método reciba referencias a los componentes que necesita para funcionar, en lugar de permitir que sea ellos mismo quienes los instancien de forma directa. Estas dependencias tienen que ser inyectadas por abstracciones(interfaces), para que facilite el reemplazo de componentes.



```

1 public class InvoiceServices
2 {
3     ...
4     public void Remove(int invoiceId)
5     {
6         using (var invoiceRepository = new InvoiceRepository())
7         {
8             var removed = invoiceRepository.Remove(invoiceId);
9             if (removed)
10            {
11                var notifier = new EmailNotifier();
12                notifier.NotifyAdmin($"Invoice {invoiceId} removed");
13            }
14        }
15    }
16 }

```

```

public class InvoiceServices: IInvoiceServices
{
    private readonly IInvoiceRepository _invoiceRepository;
    private readonly INotifier _notifier;

    public InvoiceServices (IInvoiceRepository invoiceRepository, INotifier notifier)
    {
        _invoiceRepository = invoiceRepository;
        _notifier = notifier;
    }

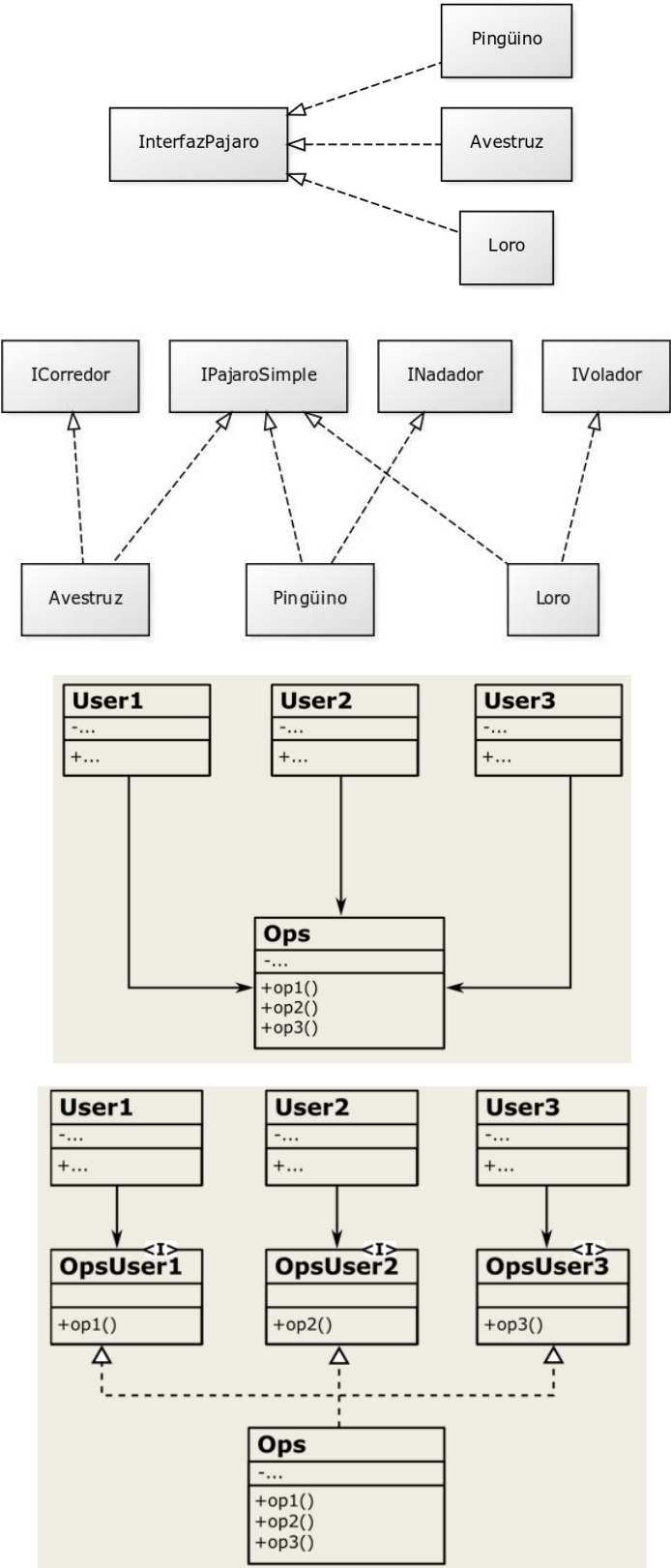
    ...
    public void Remove(int invoiceId)
    {
        var removed = _invoiceRepository.Remove(invoiceId);
        if (removed)
        {
            _notifier.NotifyAdmin($"Invoice {invoiceId} removed");
        }
    }
}

```

## Principio de diseño Segregación de la Interfaz

El principio de diseño de Segregación de la Interfaz se basa en que ninguna clase debería depender de métodos que no usa. Por tanto a la hora de crear interfaces que definan el comportamiento, es importante estar seguros de que todas las clases que implementen esas interfaces vayan a necesitar y ser capaces de implementar todos

los métodos. Por lo tanto, siempre es preferible tener varias interfaces más pequeñas.



## Principio de diseño Ley de Demeter

El principio de diseño de la Ley de Demeter se basa en reducir el acoplamiento entre clases. Esto quiere decir que un modulo no debe conocer como está hecho por dentro los objetos que manipula. Por lo tanto, el método no debería invocar métodos de los objetos devueltos.

EJ:

No cumple la ley de demeter:

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

Cumple la ley de demeter:

```
Options opts = ctxt.getOptions();  
File scratchDir = opts.getScratchDir();  
final String outputDir = scratchDir.getAbsolutePath();
```

## Fundamentos de software

Son las características básicas que debe poseer todo software que son:

- Modularidad
- Abstracción
- Acoplamiento
- Cohesión

### Modularidad

La modularidad es la propiedad de poder dividir una aplicación en partes más pequeñas, las cuales deben ser tan independientes como sea posible. También ser recombinales para construir diferentes aplicaciones de la misma familia.

Un modulo es una parte de un programa que tiene una funcionalidad en específico.

## Diseño por contrato (USO DE INTERFAZ)

Los componentes se diseñan cumpliendo ciertas condiciones de entrada y garantizando ciertas condiciones de salida. El contrato sería la interfaz.

## Sustituibilidad (Sustitución de Liskov)

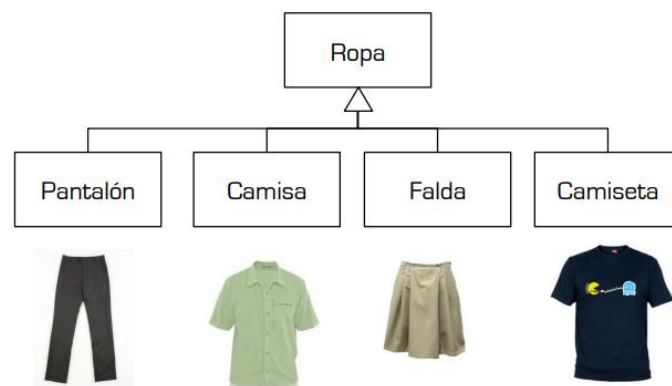
El diseño debe realizarse permitiendo usar con total libertad cualquier subtipo de una clase.

## Encapsulamiento

Se basa en que los módulos se deben crear de tal forma que la información que contiene debe ser inaccesible a los otros módulos que no la necesiten.

## Abstracción

La abstracción se trata de aislar la información que no es relevante y captar las características esenciales de un objeto, así como su comportamiento. Básicamente se trata de generalizar una serie de objetos a sus características base que los une a todos.



## **Polimorfismo**

El polimorfismo se trata de una propiedad que posee los objetos, la cual se trata de la capacidad de representar un objeto de varias formas distintas. Por ejemplo, si una clase tiene varias subclases, entonces los objetos de esas subclases también pueden ser representados como objetos de la superclase.

Es la capacidad que tiene los objetos de una clase para dar respuestas distintas en función de los parámetros usados durante la invocación.

Mueble mueble = new Silla();

## **Acoplamiento**

El acoplamiento es una medida de la interconexión entre los módulos de un programa

A menor acoplamiento mejor, ya que se reduce la propagación de errores y fomenta la reutilización de los módulos.

## **Cohesión**

La cohesión se basa en crear módulos que ejecuten tareas sencillas. A mayor cohesión mejor, ya que el módulo será más sencillo de diseñar, programar y probar.

## **Dependencias**

### **Independencia funcional**

Es que cada módulo se debe centrar en realizar una función con una interfaz sencilla.

### **Dependencias Circulares**

Son dependencias nocivas ya que:

- Se reduce o hace imposible la reutilización del código
- Un cambio en un módulo afecta a los otros.
- Puede causar filtraciones de memoria y recursiones infinitas.

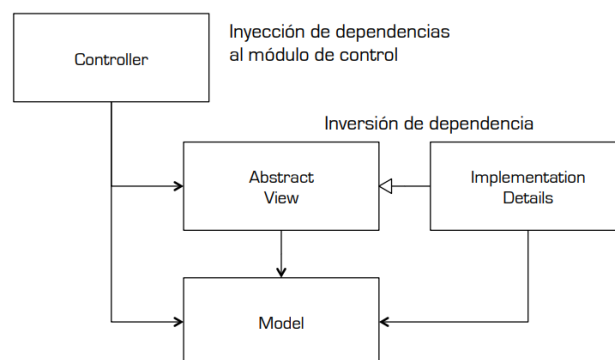
## Estilo Arquitectónico

Se tratan de una colección de decisiones sobre la organización estructural de un Sistema Software.

Tipos de estilos:

Modelo del Dominio, Documento/Vista, MVC, MVP, MVVM, Hexagonales, Lambda...

MVC -> La responsabilidad la tiene la vista.



- 1 **Model.** Módulos que representan los datos.
- 2 **View.** Módulos que gestionan el modelo con otros sistemas/usuario
- 3 **Controller.** Módulos que gestionan la lógica de control

MVP -> La responsabilidad la tiene el modelo, ya que es el presenter quien le quita responsabilidad a la vista.