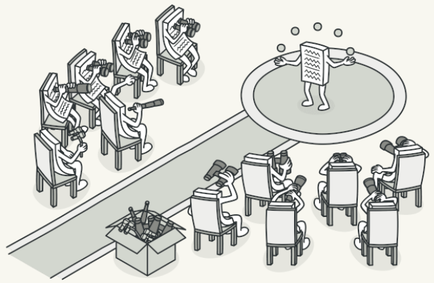
PATRONES DE DISEÑO

Presentado por Domingo Edwards, Fabian Estelle, Sebastián Ramos

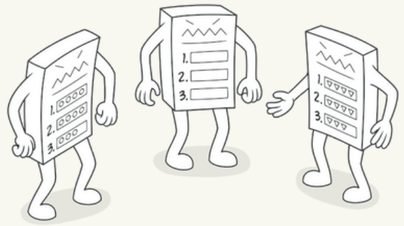
PATRONES DE DISEÑ0:

De Comportamiento

Patrones de Comportamiento

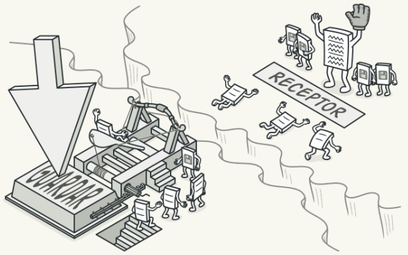


Observer



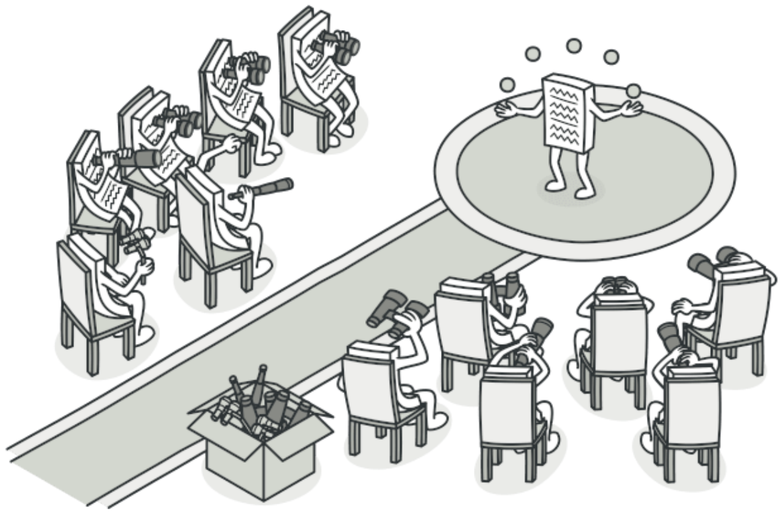
Template

Strategy



Command

Patrón: OBSERVER

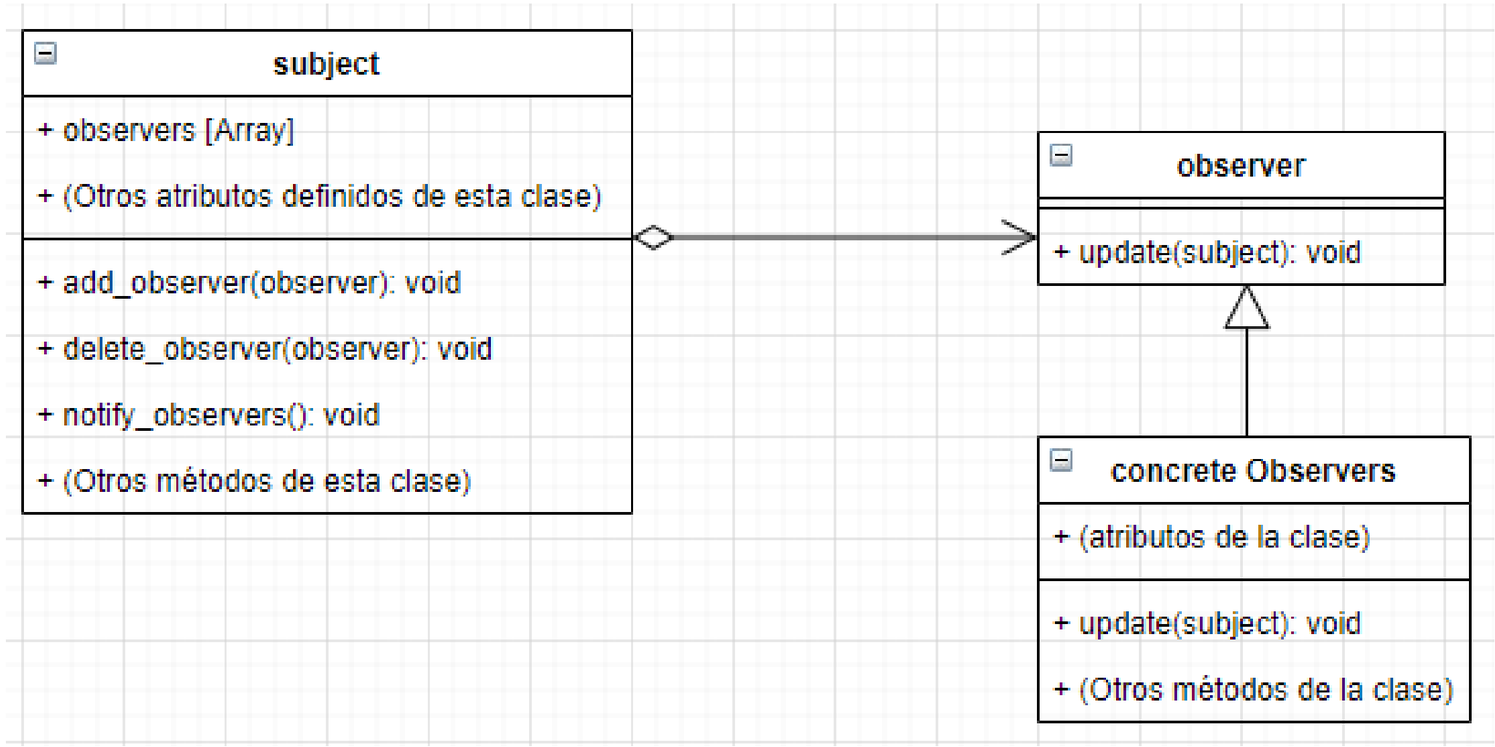


- Definir un mecanismo para notificar a varios objetos (observadores) cuando suceda un evento sobre un objeto en particular (sujeto)

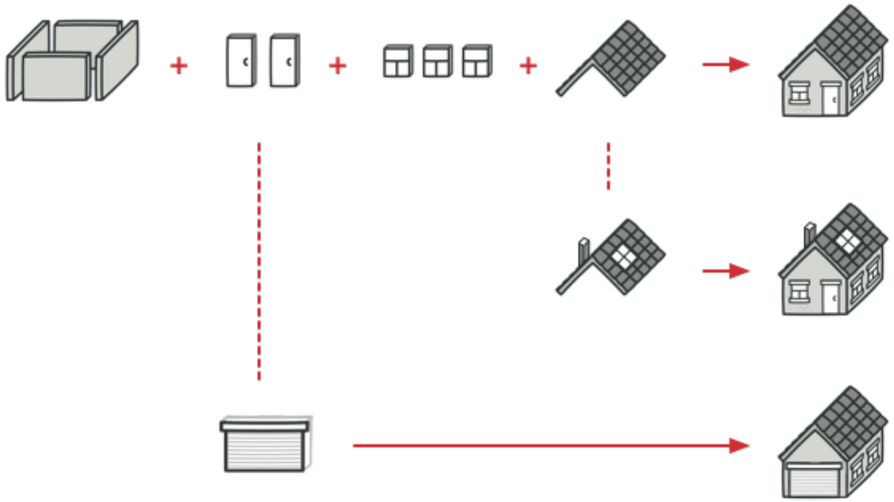
Patrón: OBSERVER

¿Cómo Funciona?

- Cualquier clase se puede transformar en un sujeto u observador, solo hay que agregar algunos atributos y métodos extra

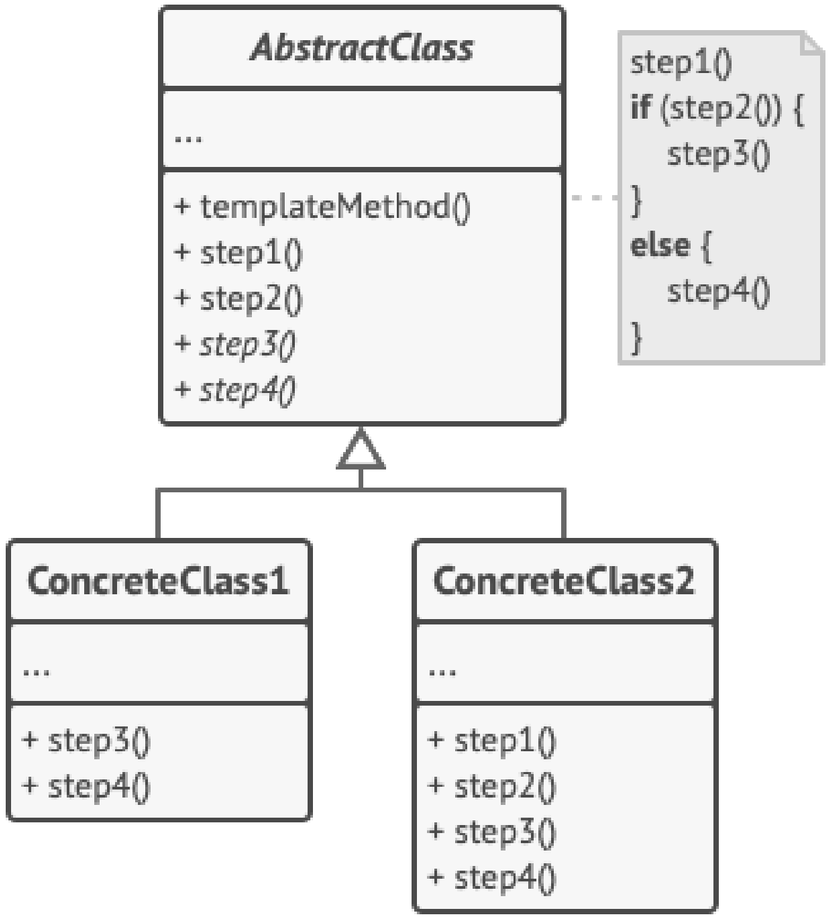


Patrón: TEMPLATE



- Encapsular secuencias de operaciones que se repiten mucho. Define el "esqueleto"

Patrón: TEMPLATE



- Pasos de implementación:
  - Crear una clase abstracta que contenga los métodos (un método por cada paso del algoritmo). Los métodos pueden ser abstractos
  - Crear clases concretas que heredan del template para construir una clase nueva. Pueden sobreescribir los métodos heredados

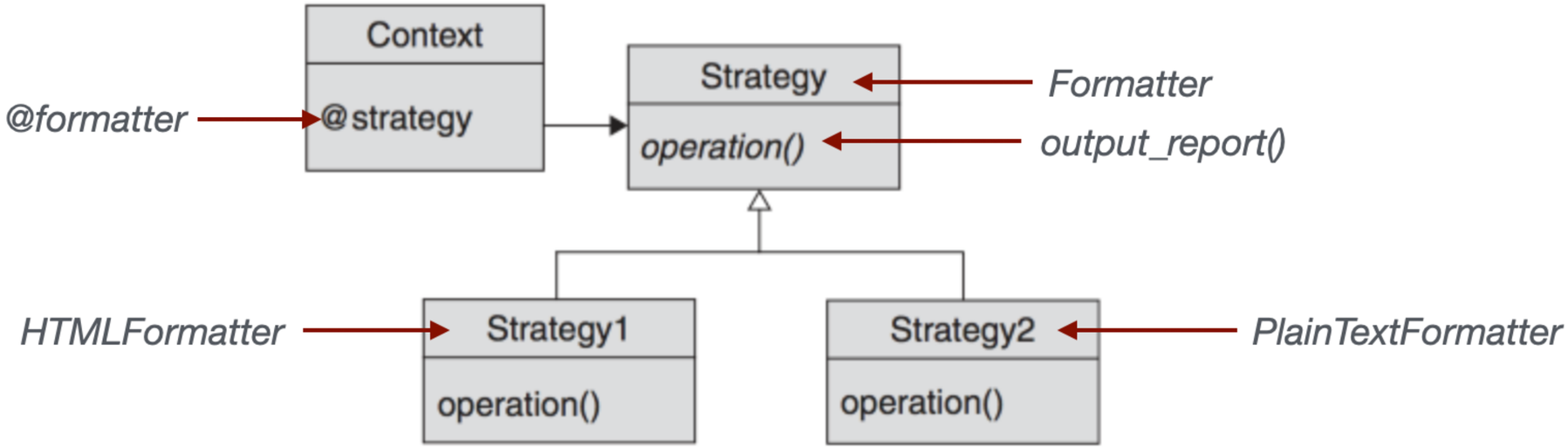
Patrón: STRATEGY

- Permite definir una familia de algoritmos, ponerlos en clases separadas y hacer sus objetos intercambiables

- En el algoritmo base se declara los métodos, en las implementaciones se termina de completar

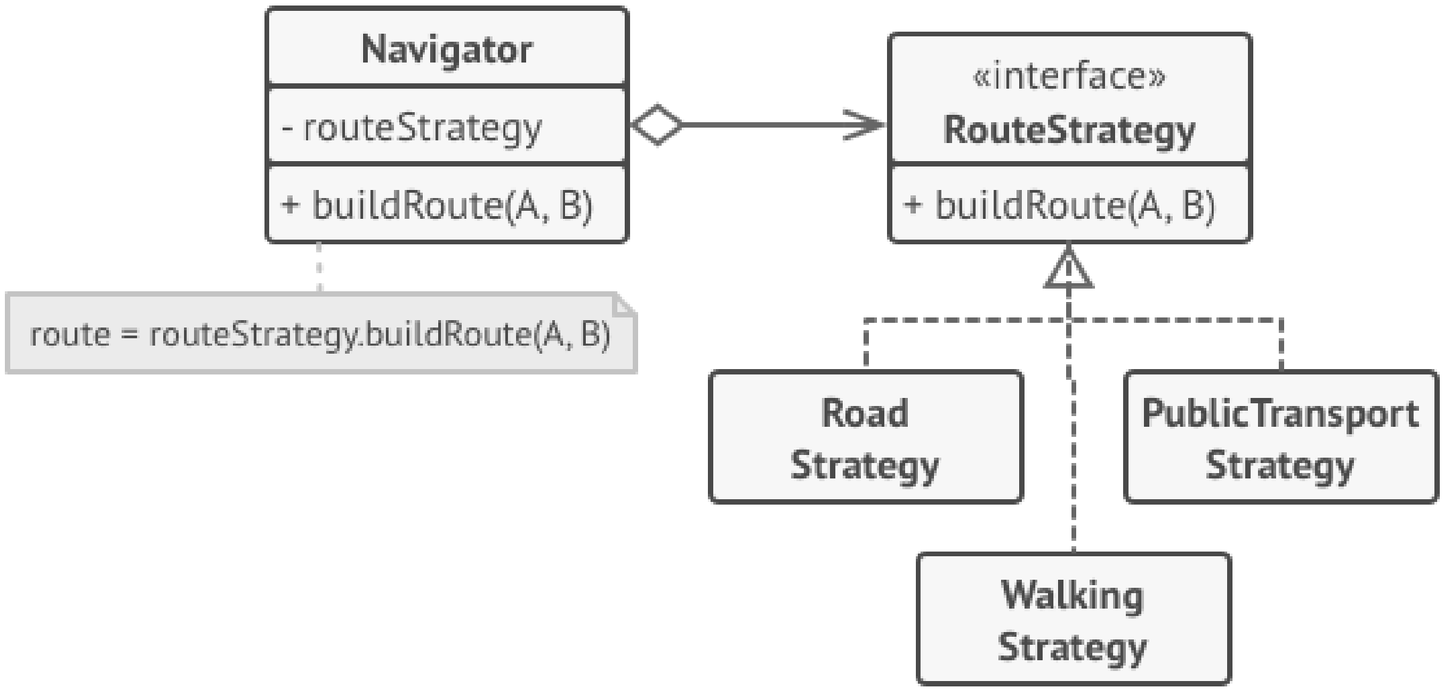
Patrón: STRATEGY

- Diagrama UML

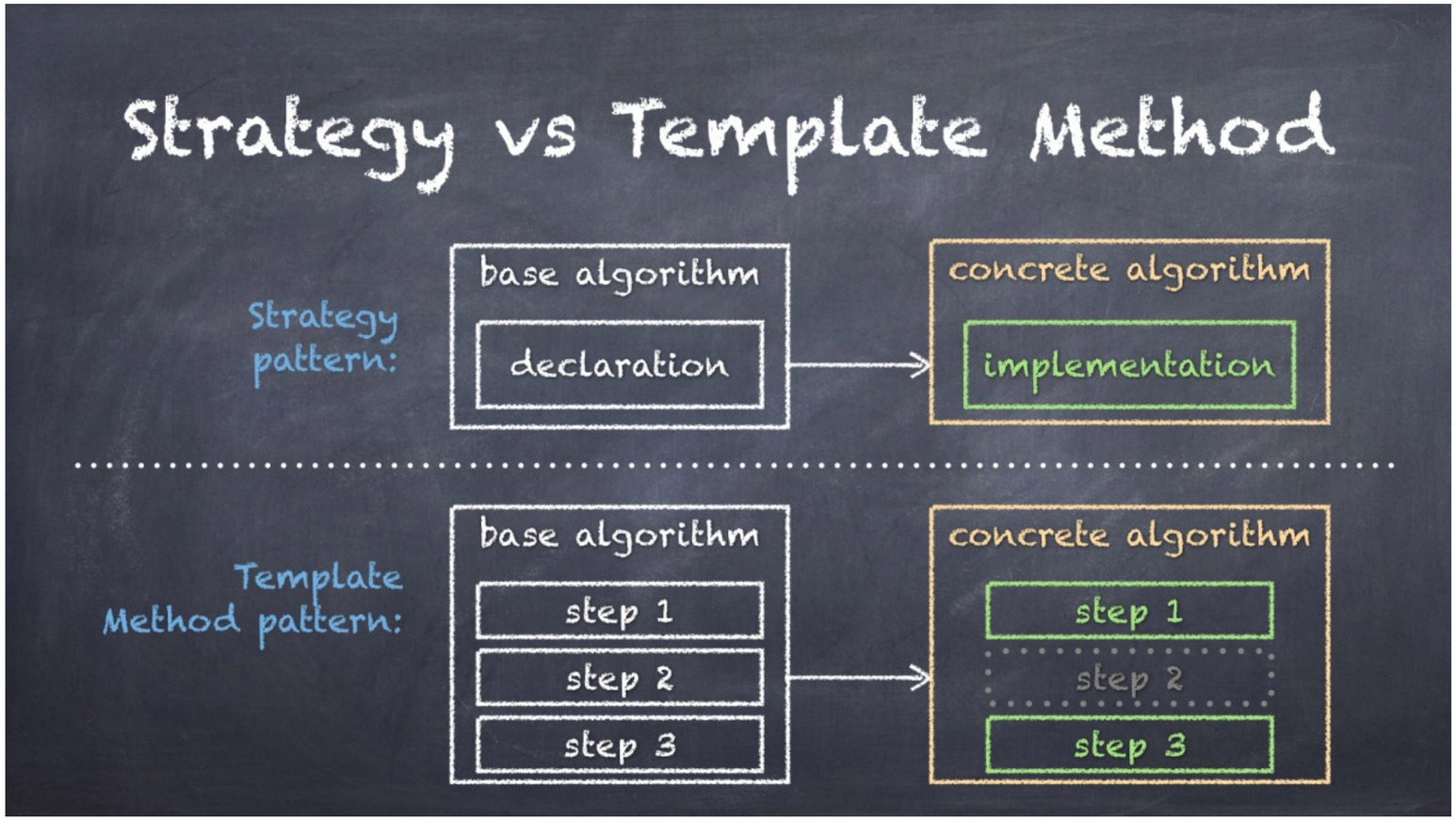


Patrón: STRATEGY

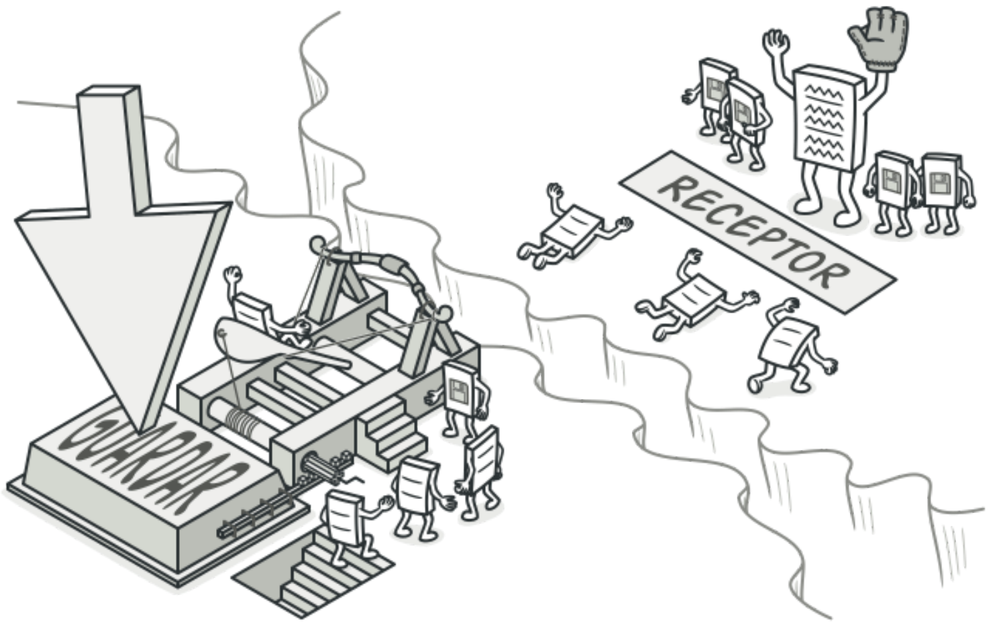
- Ejemplo: Implementación en app que calcular rutas



Patrón: STRATEGY



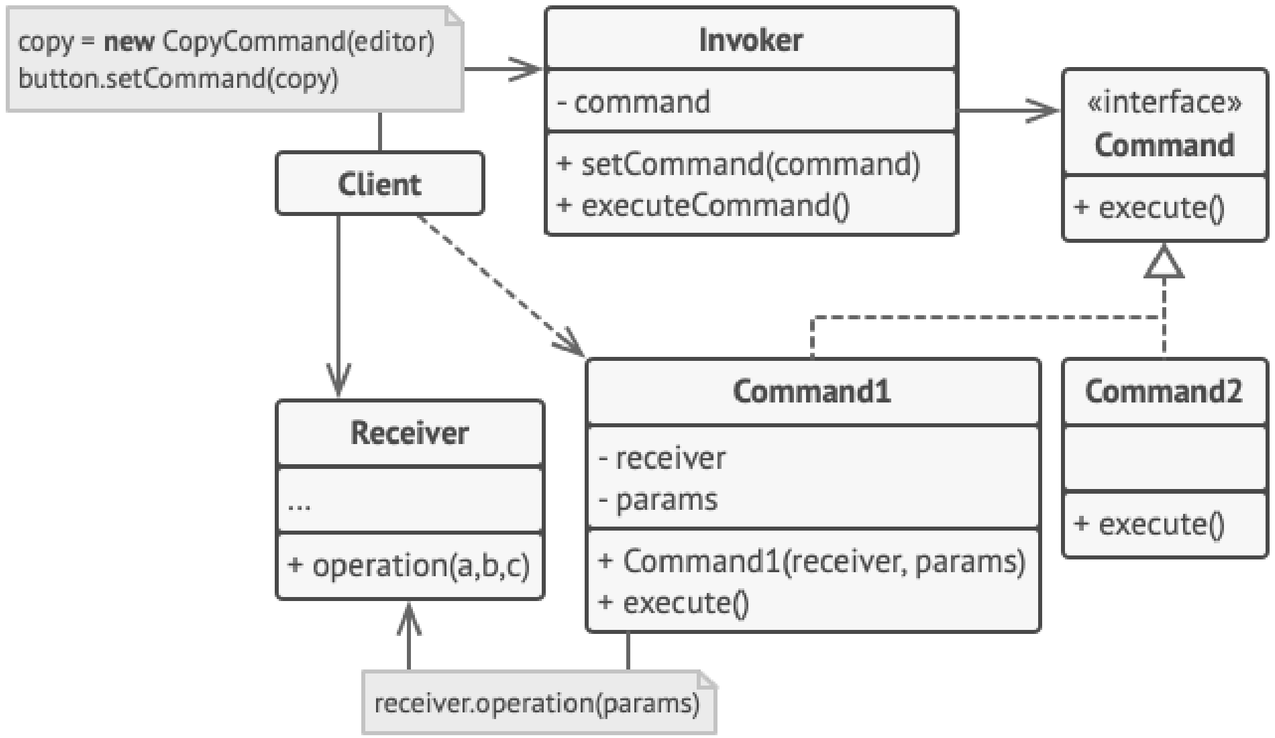
Patrón: COMMAND



- Convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud

Patrón: COMMAND

- Diagrama UML:



EJERCICIO

Pregunta 1 Examen 2020-2

P1 Examen 2020-2

Se ha construido un trozo de código que utiliza un patrón observador para vigilar y actuar frente a cambios en el sujeto que en este caso es un objeto de la clase Employee. Los objetos de esta clase representan empleados y tienen solo 3 atributos: name, job y salary.

Los objetos observadores corresponden a instancias de 2 clases: JobsLog y SalariesLog y, como buenos observadores se activan cuando el sujeto sufre cambios. Para simplificar, supondremos que la reacción de cada uno de ellos ante cambios en el estado del sujeto es muy simple: Un objeto JobsLog imprimirá el nuevo cargo del empleado de la siguiente forma:
*** Pepe es un analista *** (suponiendo que es Pepe quien cambió)
Un objeto SalariesLog imprimirá el nuevo sueldo del empleado de la siguiente forma:
*** el sueldo de Pepe es 1.000.000 *** (suponiendo que es Pepe quien cambió)

a) Escriba las clases Employee, SalariesLog y JobsLog y dibuje el diagrama de clases correspondiente

P1 Examen 2020-2

A continuación un trozo de código que prueba el funcionamiento:

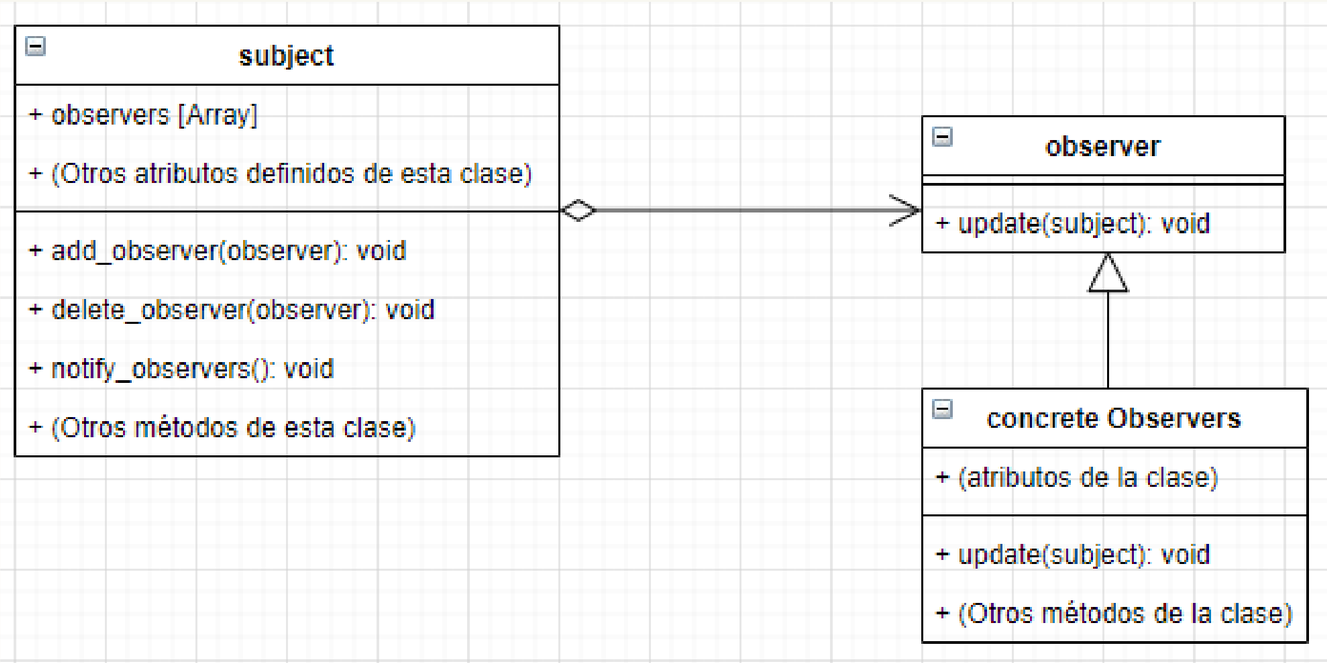
```
jaime = Employee.new('Jaime', 'Programador Junior', 1000000)
jobslog = JobsLog.new
salarieslog = SalariesLog.new
jaime.add_observer( jobslog )
jaime.add_observer( salarieslog )
jaime.job = "Programador"
jaime.salary = 1500000
jaime.delete_observer( jobslog )
jaime.job = "Programador Senior"
jaime.salary = 2000000
```

El output que produce es el siguiente:

```
*** Jaime es un Programador ***
*** el sueldo de Jaime es 1000000 ***
*** Jaime es un Programador ***
*** el sueldo de Jaime es 1500000 ***
*** el sueldo de Jaime es 1500000 ***
*** el sueldo de Jaime es 2000000 ***
```

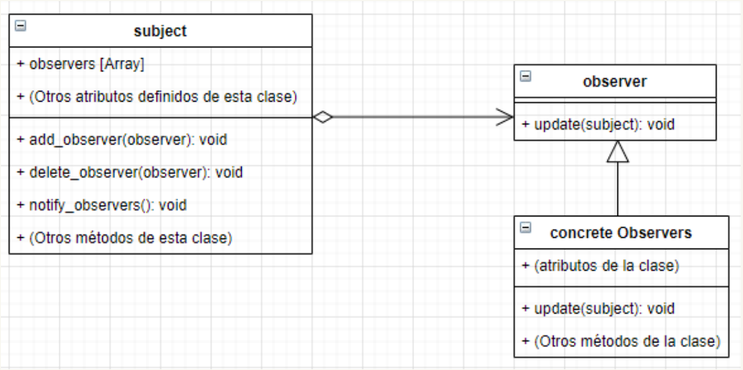
P1 Examen 2020-2

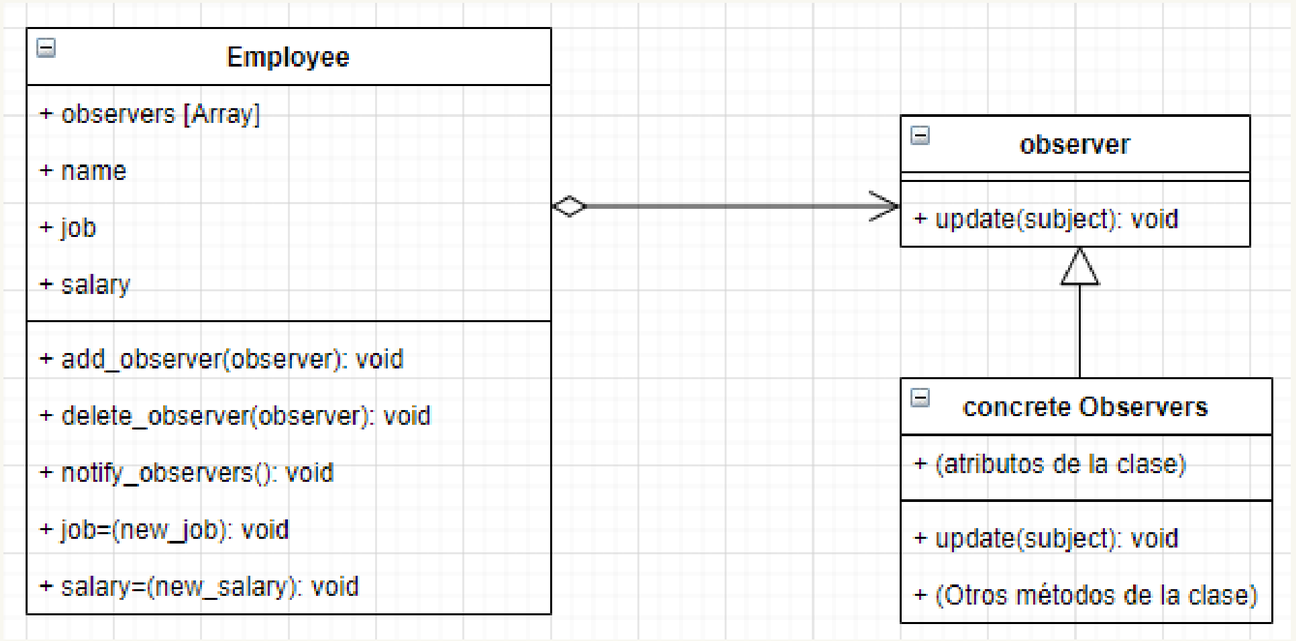
Se ha construido un trozo de código que utiliza un patrón observador para vigilar y actuar frente a cambios en el sujeto que en este caso **es un objeto de la clase Employee**. Los objetos de esta clase representan empleados y tienen **solo 3 atributos: name, job y salary.**



P1 Examen 2020-2

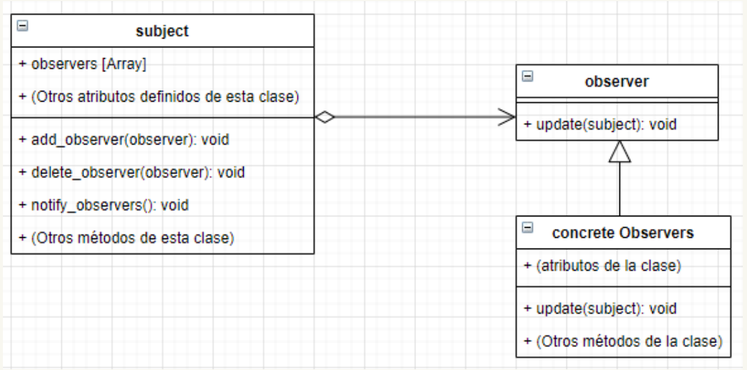
Se ha construido un trozo de código que utiliza un patrón observador para vigilar y actuar frente a cambios en el sujeto que en este caso **es un objeto de la clase Employee**. Los objetos de esta clase representan empleados y tienen **solo 3 atributos: name, job y salary.**

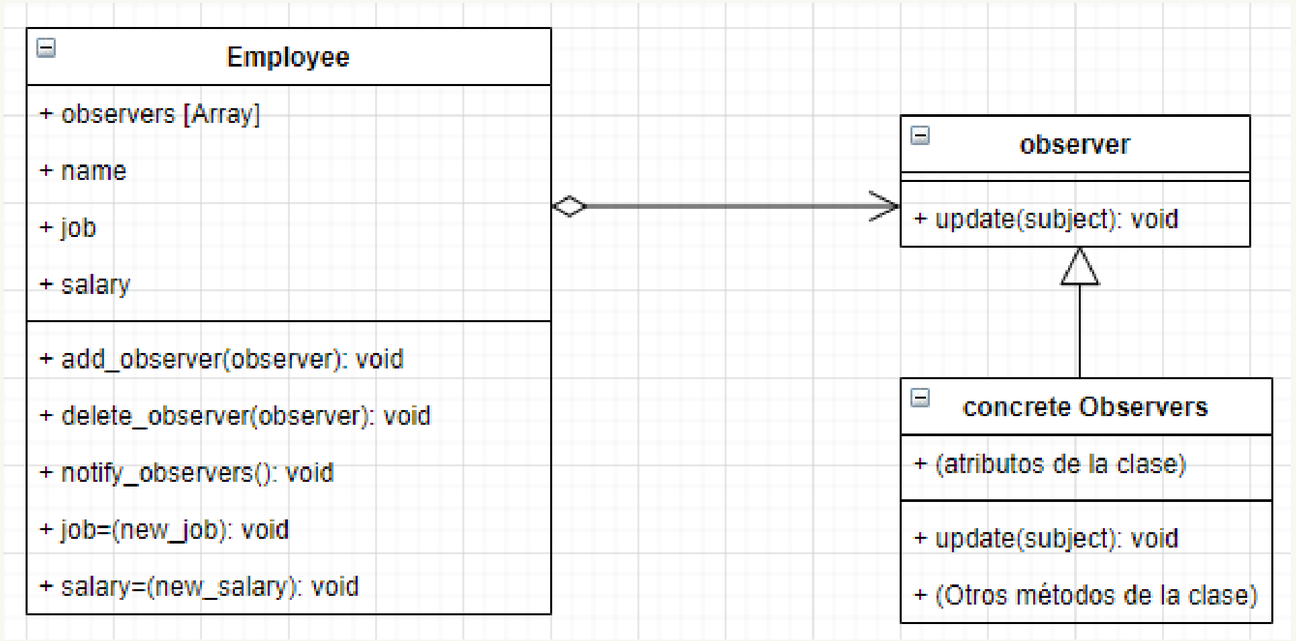




P1 Examen 2020-2

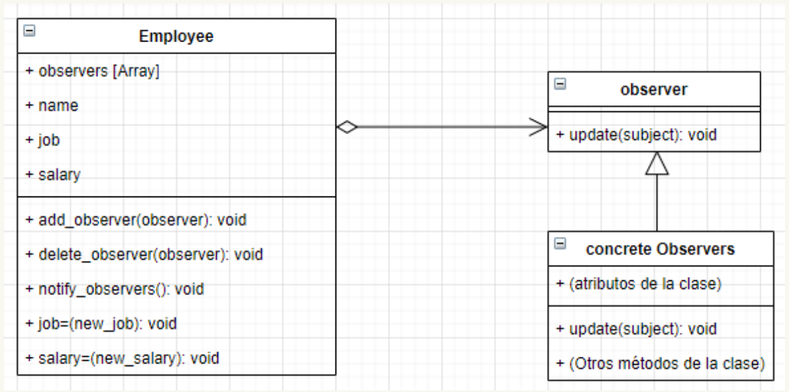
Los objetos observadores corresponden a instancias de 2 clases: **JobsLog y SalariesLog** y, como buenos observadores se activan cuando el sujeto sufre cambios. Para simplificar, supondremos que la reacción de cada uno de ellos ante cambios en el estado del sujeto es muy simple: Un objeto JobsLog **imprimirá el nuevo cargo** del empleado y un objeto SalariesLog **imprimirá el nuevo sueldo** del empleado

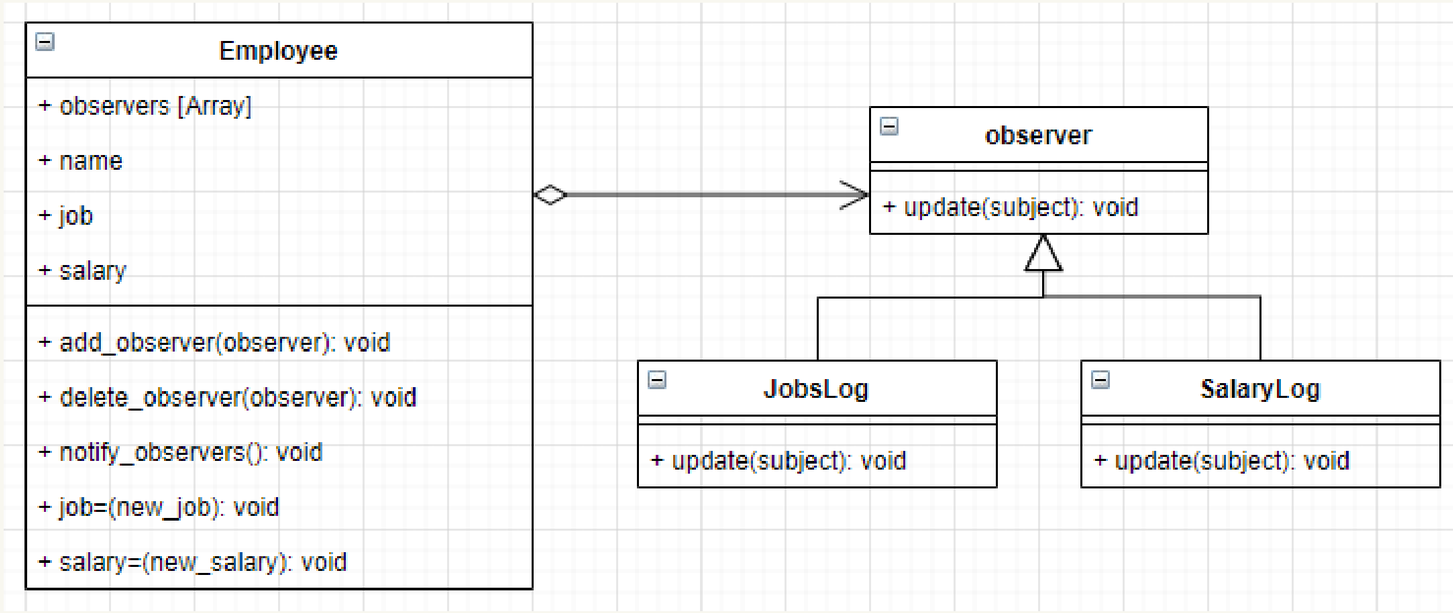




P1 Examen 2020-2

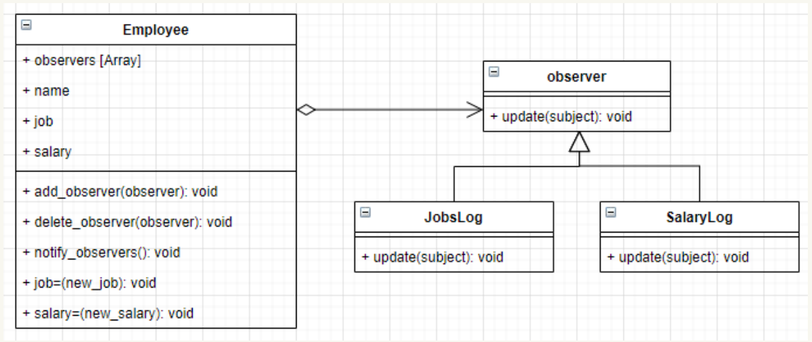
Los objetos observadores corresponden a instancias de 2 clases: **JobsLog y SalariesLog** y, como buenos observadores se activan cuando el sujeto sufre cambios. Para simplificar, supondremos que la reacción de cada uno de ellos ante cambios en el estado del sujeto es muy simple: Un objeto JobsLog **imprimirá el nuevo cargo** del empleado y un objeto SalariesLog **imprimirá el nuevo sueldo** del empleado

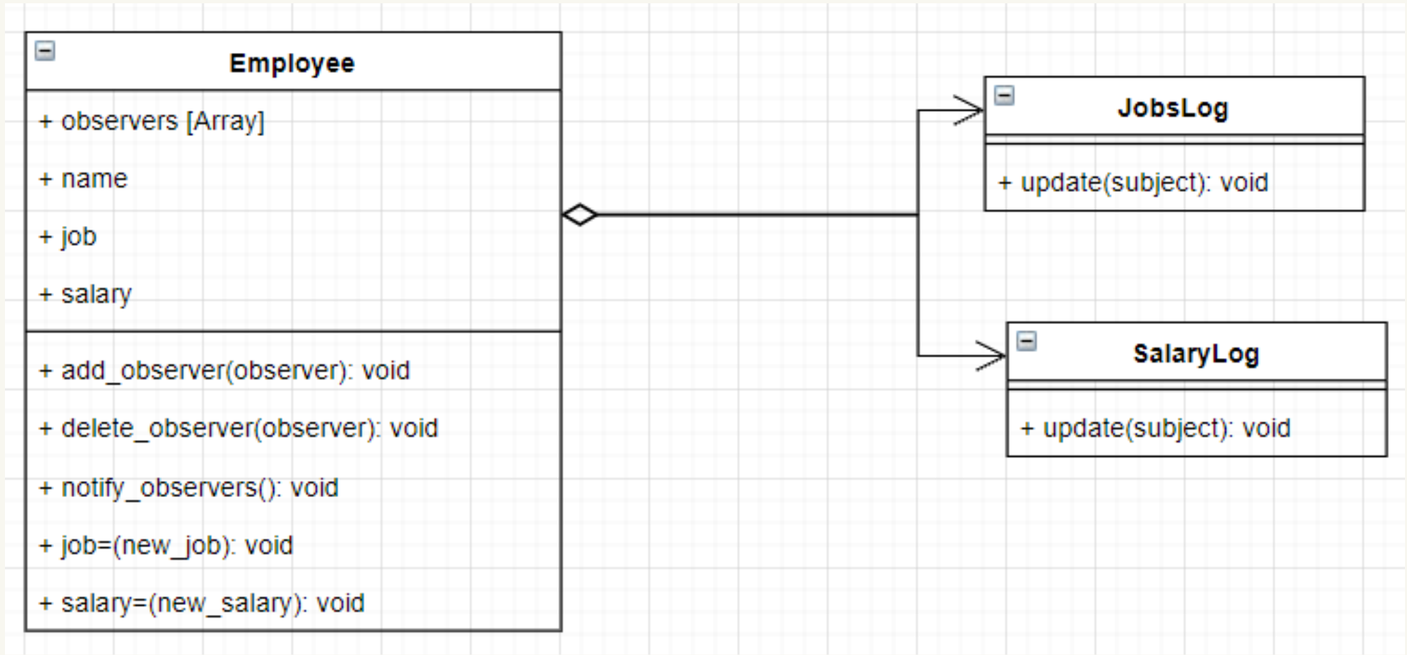




P1 Examen 2020-2

Los objetos observadores corresponden a instancias de 2 clases: **JobsLog y SalariesLog** y, como buenos observadores se activan cuando el sujeto sufre cambios. Para simplificar, supondremos que la reacción de cada uno de ellos ante cambios en el estado del sujeto es muy simple: Un objeto JobsLog **imprimirá el nuevo cargo** del empleado y un objeto SalariesLog **imprimirá el nuevo sueldo** del empleado





P1 Examen 2020-2

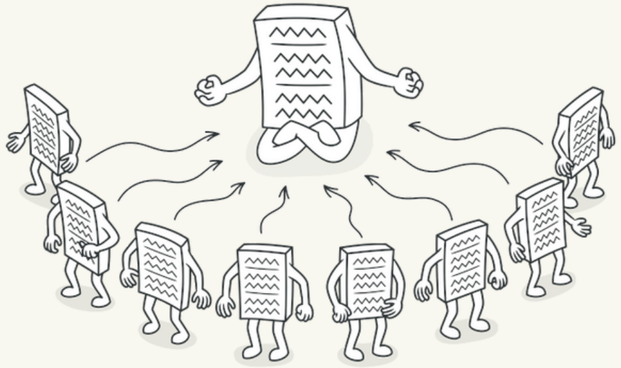
a) **Escriba las clases Employee, SalariesLog y JobsLog** y dibuje el diagrama de clases correspondiente

Implementación en Ruby

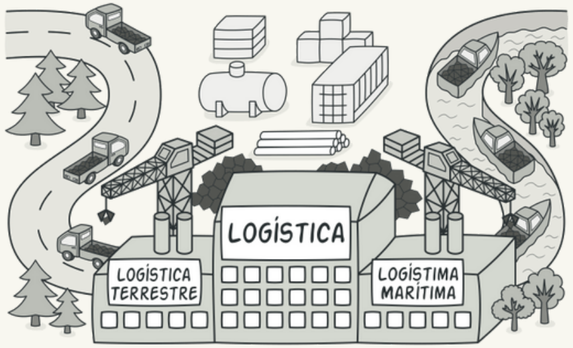
PATRONES DE DISEÑO:

Creacionales

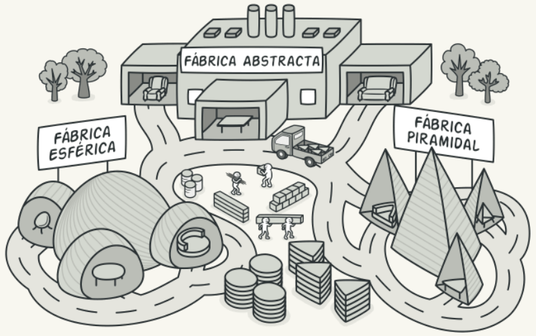
Patrones Creacionales



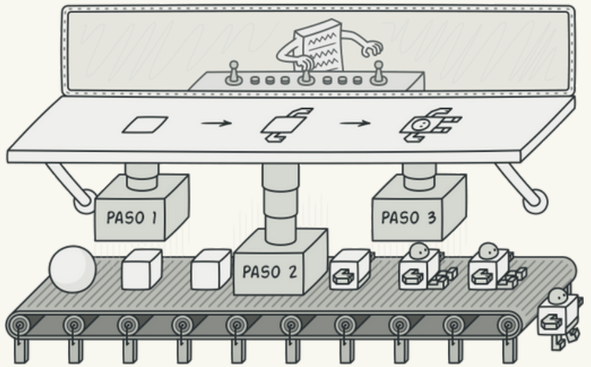
Singleton



Factory Method

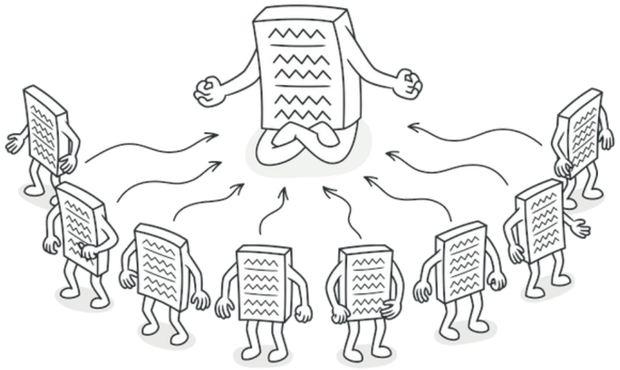


Abstract Factory



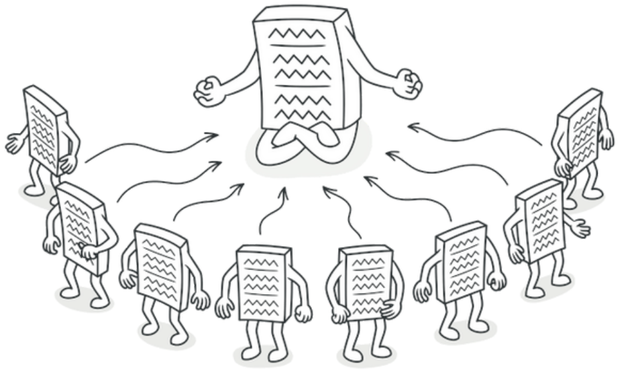
Builder

Patrón: SINGLETON



- Permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

Patrón: SINGLETON

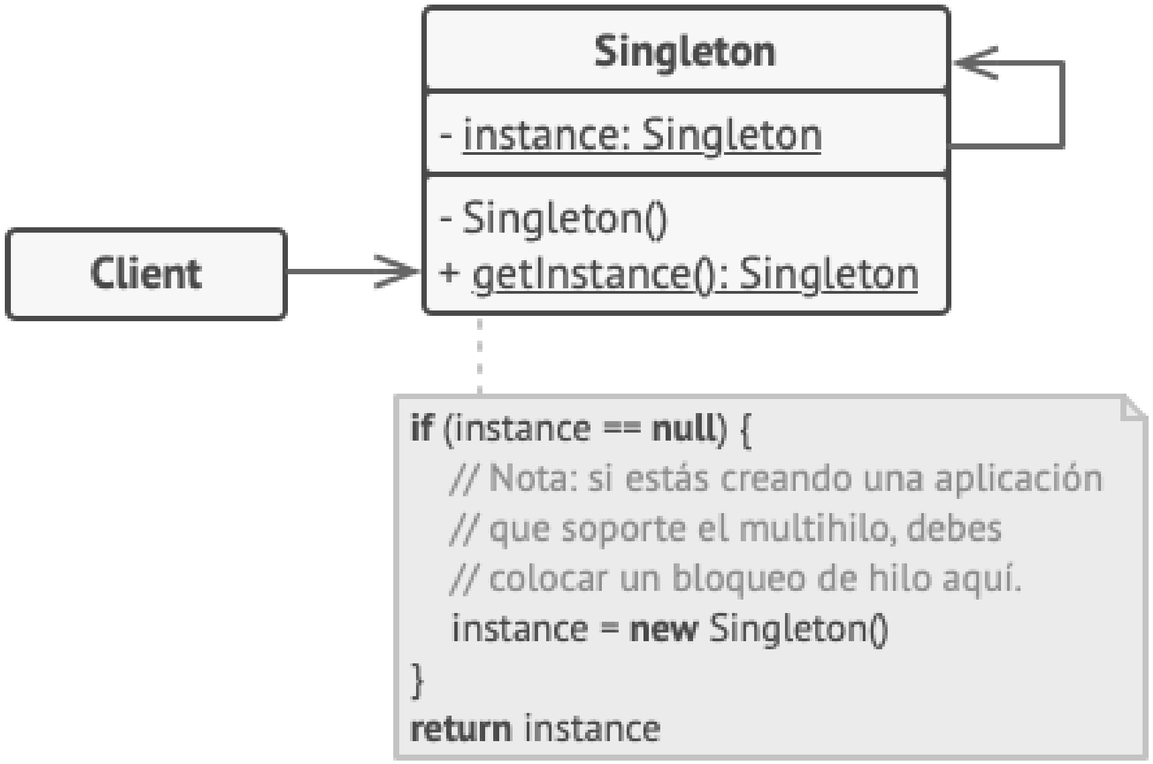


Problemas que soluciona:

- Controlar el acceso de recursos compartidos
- Sobreescribir variables globales

Patrón: SINGLETON

Estructura:




```
class Tablero
    attr_accessor :status

    @@instance = Tablero.new

    def self.instance
      return @@instance
    end

    private_class_method :new
end
```

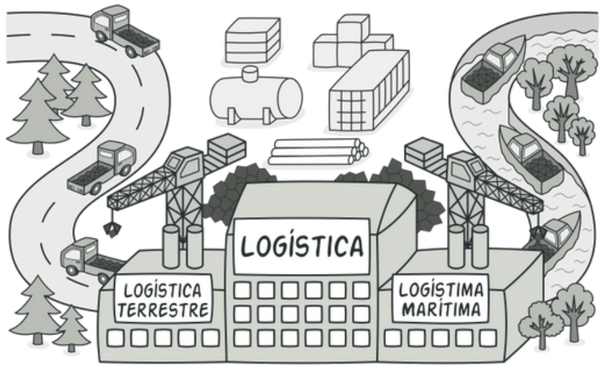
variable de clase guarda la única instancia

método de clase retorna la única instancia

método de clase new se hace privado

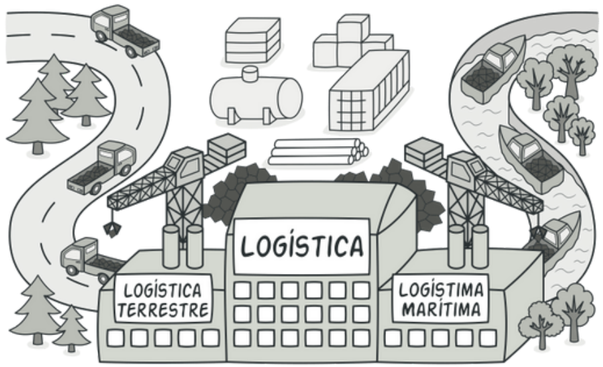
```
tablero1 = Tablero.instance
tablero1.status = "on"
puts tablero1.status
tablero2 = Tablero.instance    # devuelve el mismo tablero
puts tablero2.status
```

Patrón: FACTORY METHOD



- Permite crear objetos en una superclase, mientras que las subclases que heredan de la superclase pueden alterar el tipo de objetos que serán creados

Patrón: FACTORY METHOD

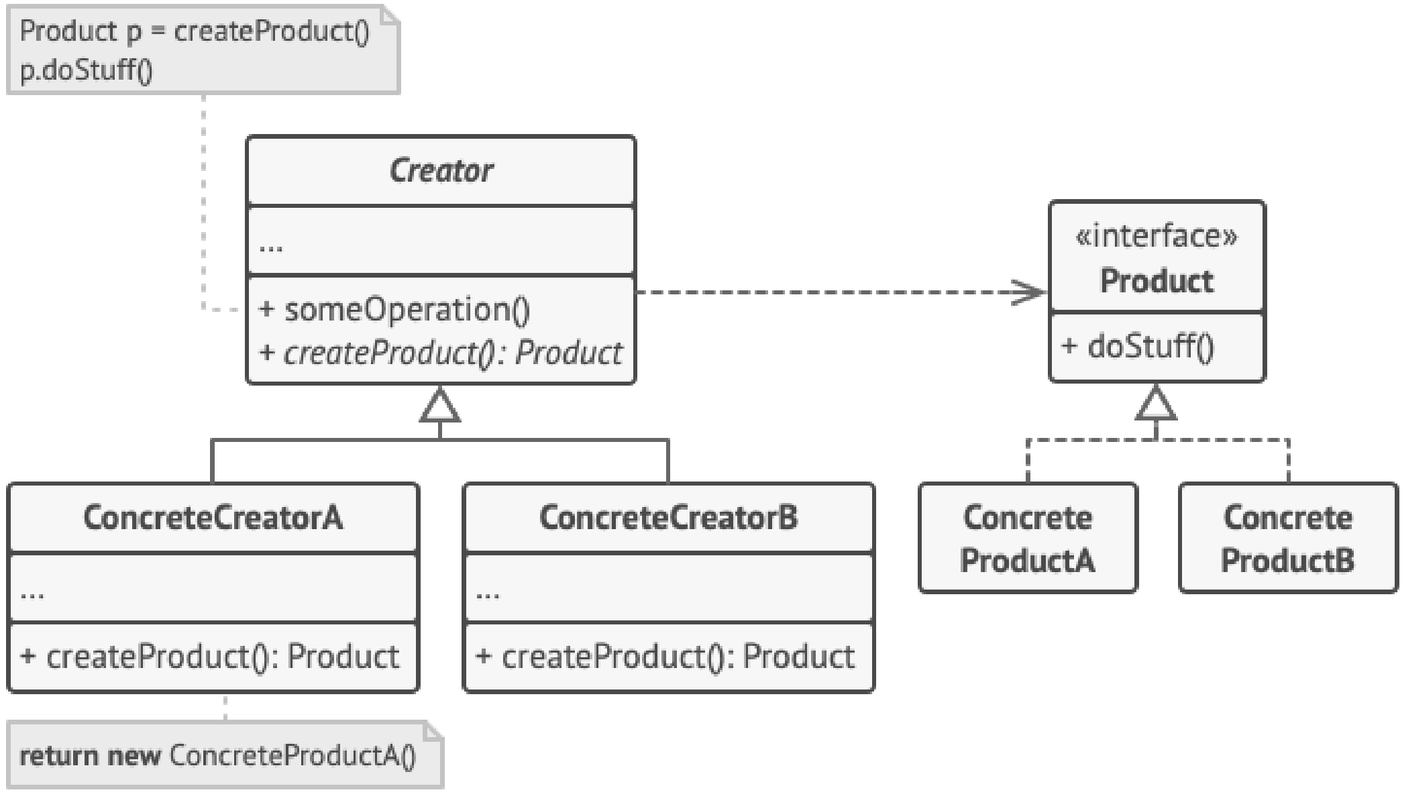


Problemas que soluciona:

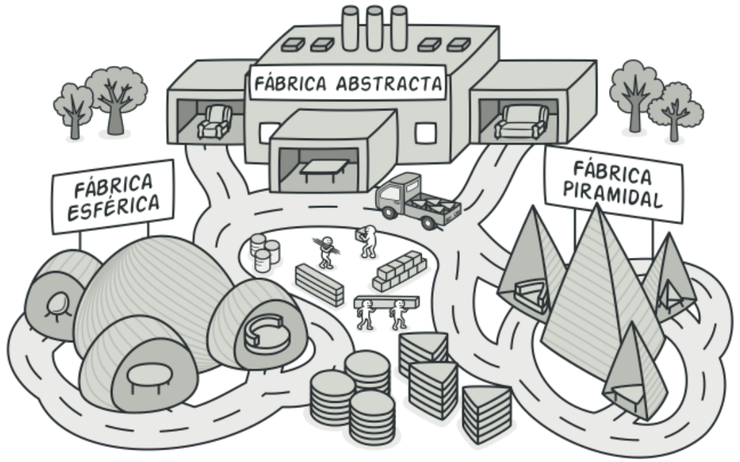
- Bajo acoplamiento entre creador y productos
- Se pueden incorporar nuevos tipos de productos sin tener que descomponer el código existente

Patrón: FACTORY METHOD

Estructura:



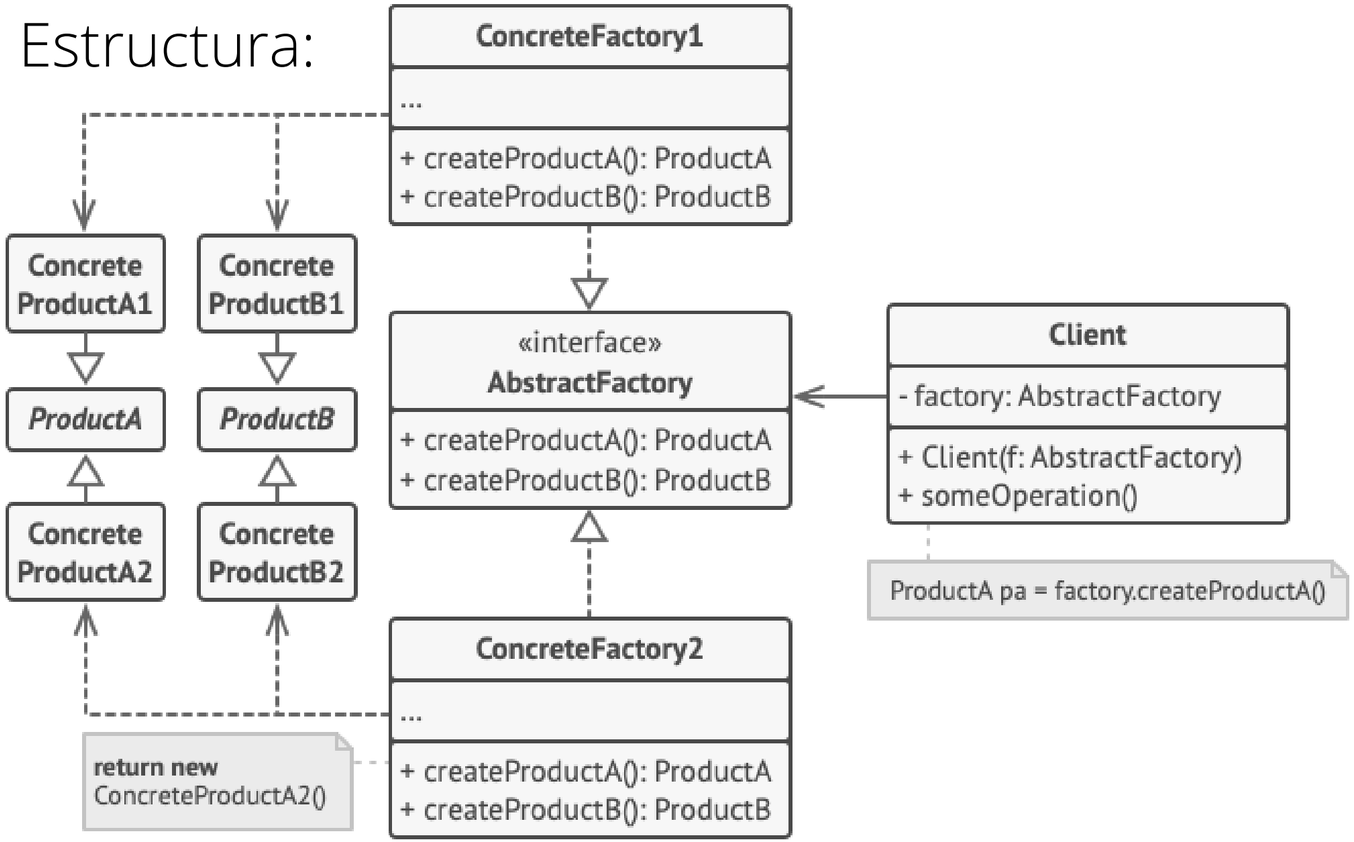
Patrón: ABSTRACT FACTORY

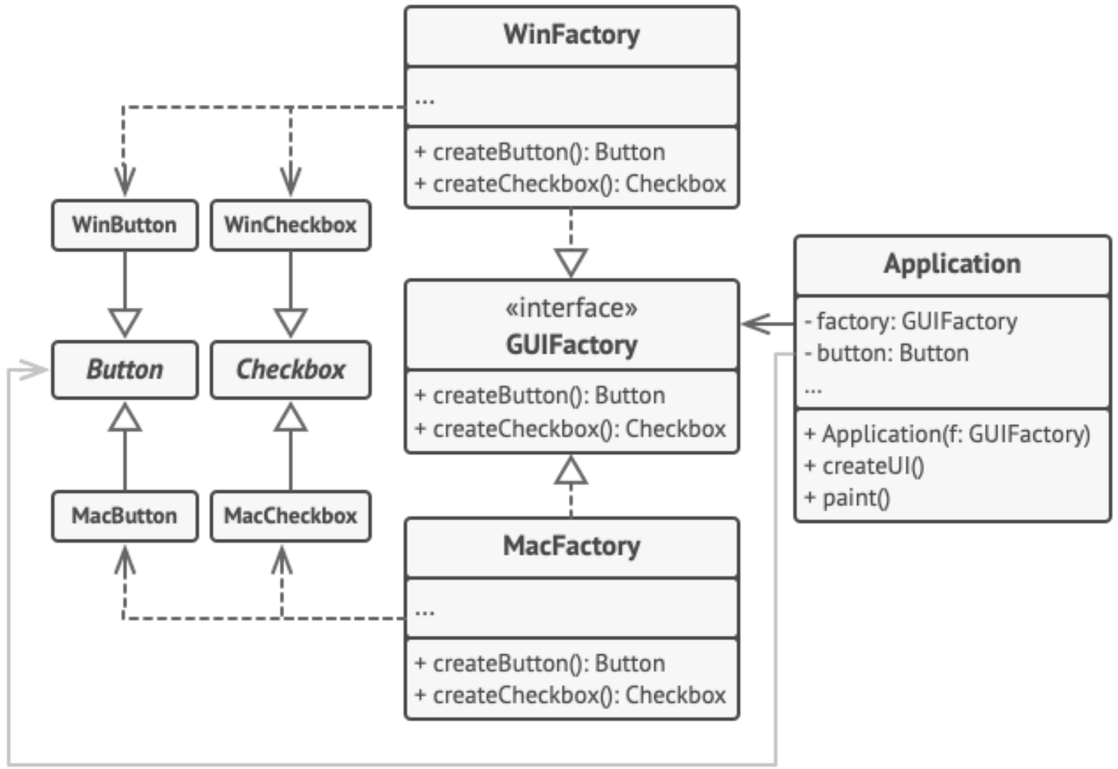


- Permite producir familias de objetos relacionados sin especificar sus clases concretas.

Patrón: ABSTRACT FACTORY

Estructura:





FACTORY METHOD V/S ABSTRACT FACTORY

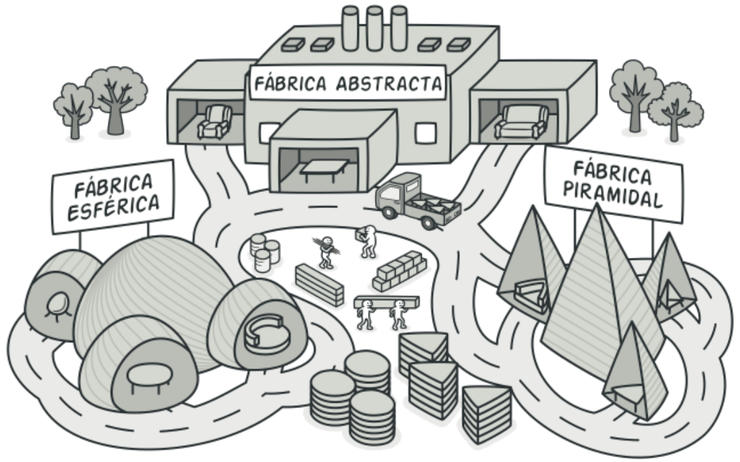
Factory method

- Se usa la misma fábrica para instanciar objetos
- Crea un solo tipo de producto
- Se usa en diseños sencillos

Abstract factory

- Se usan distintas fábricas para instanciar distintos tipos de objetos
- Crea una familia de productos
- Se usa en diseños más complejos
- Pueden haber varios factory method en la misma clase

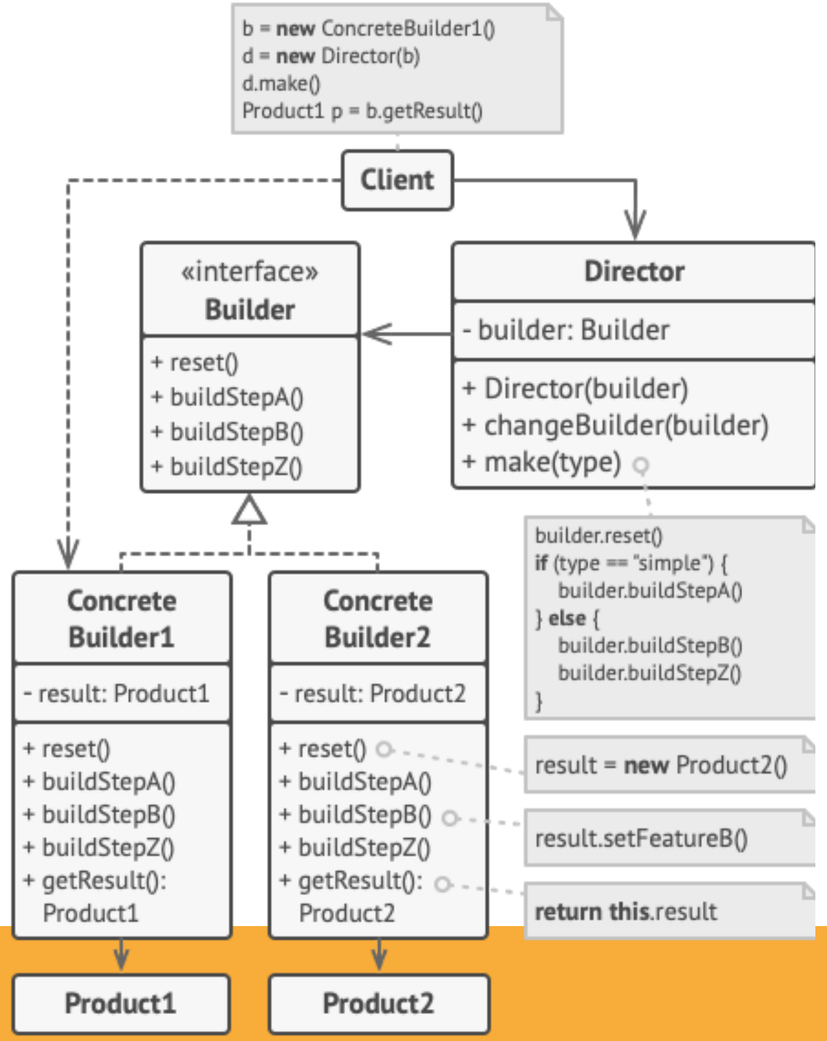
Patrón: BUILDER



- Permite construir objetos complejos paso a paso. El patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.

Patrón: BUILDER

Estructura:



EJERCICIO

Pregunta 4 Examen 2021-1

Se quiere simular el funcionamiento de varios posibles negocios de venta de comida: una pizzería, una hamburguesería y una ensaladería. Para simplificar supongamos que en cada caso solo se fabrican dos productos:

- Pizzería: pizza de pepperoni y pizza vegetariana
- Hamburguesería: regular y not_meat
- Ensaladería: cesar y mediterranea

Queremos sacar partido del patrón Abstract Factory y para ello se pide:

a) Definir tres fábricas abstractas: una para las pizzas una para las hamburguesas y una para las ensaladas. Además de escribir el código de las fábricas escriba el código de los productos (lo más simple posible)

b) Definir una clase Negocio con un método llamado simular que recibe una fábrica y el número de productos de cada tipo y procede a hacer la simulación como muestra el ejemplo. Su código debe funcionar exactamente de la misma manera.

a) Definir tres fábricas abstractas: una para las pizzas una para las hamburguesas y una para las ensaladas. Además de escribir el código de las fábricas escriba el código de los productos (lo más simple posible)

```
class Pizzeria
    def new_p1(number)
        Pepperoni.new(number)
    end
    def new_p2(number)
        Vegetarian.new(number)
    end
end
class Hamburgueseria
    def new_p1(number)
        Regular.new(number)
    end
    def new_p2(number)
        Not_meat.new(number)
    end
end
class Ensaladeria
    def new_p1(number)
        Cesar.new(number)
    end
    def new_p2(number)
        Mediterranean.new(number)
    end
end
```

a) Definir tres fábricas abstractas: una para las pizzas una para las hamburguesas y una para las ensaladas. Además de escribir el código de las fábricas escriba el código de los productos (lo más simple posible)

```
class Pepperoni
    def initialize (number)
        @name = 'pizza peperoni ' + number
    end
    def reveal
        return @name + ' saliendo'
    end
end
class Vegetarian
    def initialize (number)
        @name = 'pizza vegetariana ' + number
    end
    def reveal
        return @name + ' saliendo'
    end
end
class Regular
    def initialize (number)
        @name = 'hamburguesa regular ' + number
    end
    def reveal
        return @name + ' saliendo'
    end
end
```

a) Definir tres fábricas abstractas: una para las pizzas una para las hamburguesas y una para las ensaladas. Además de escribir el código de las fábricas escriba el código de los productos (lo más simple posible)

```
class Not_meat
    def initialize (number)
        @name = 'hamburguesa not_meat ' + number
    end
    def reveal
     return @name + ' saliendo'
    end
end
class Cesar
    def initialize (number)
        @name = 'ensalada cesar ' + number
    end
    def reveal
        return @name + ' saliendo'
    end
end
class Mediterranean
    def initialize (number)
        @name = 'ensalada mediterranea ' + number
    end
    def reveal
        return @name + ' saliendo'
    end
end
```

b) Definir una clase Negocio con un método llamado simular que recibe una fábrica y el número de productos de cada tipo y procede a hacer la simulación como muestra el ejemplo. Su código debe funcionar exactamente de la misma manera.

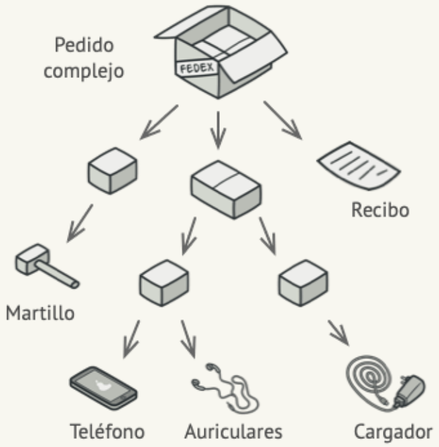
```
class Negocio
    def initialize(meal_factory, number_product_1, number_product_2)
        @the_factory = meal_factory
        @p1s = []
        number_product_1.times do |i|
            p1 = @the_factory.new_p1("#{i+1}")
            @p1s << p1
        end
        @p2s = []
        number_product_2.times do |i|
            p2 = @the_factory.new_p2("#{i+1}")
            @p2s << p2
        end
    end
    def simular
        @p1s.each {|p1| puts(p1.reveal)}
        @p2s.each {|p2| puts(p2.reveal)}
    end
end

(Negocio.new(Hamburgueseria.new,3,1)).simular
(Negocio.new(Pizzeria.new,2,3)).simular
(Negocio.new(Ensaladeria.new,4,1)).simular
```

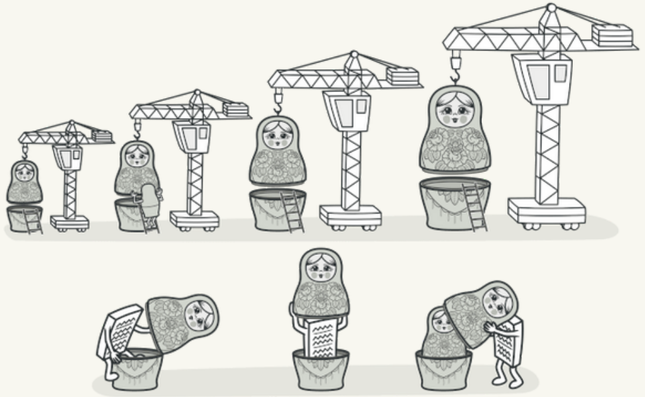
PATRONES DE DISEÑ0:

Estructurales

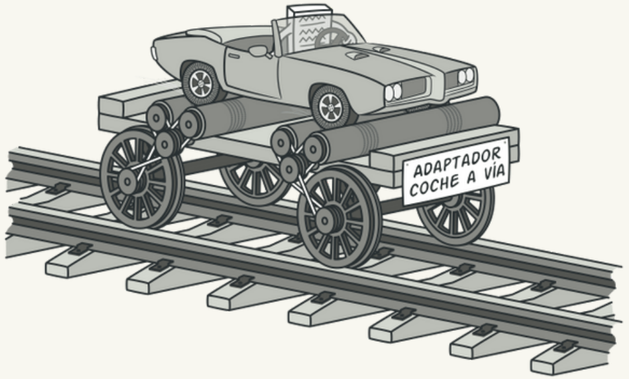
Patrones de Comportamiento



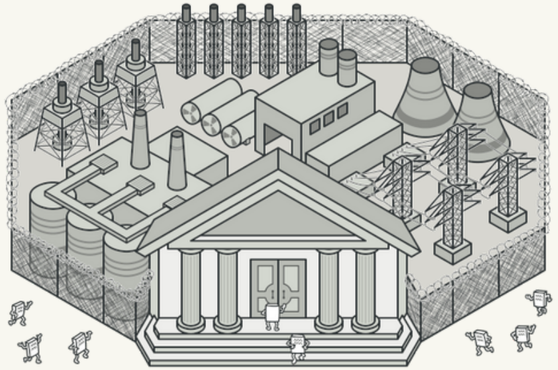
Composite



Decorator

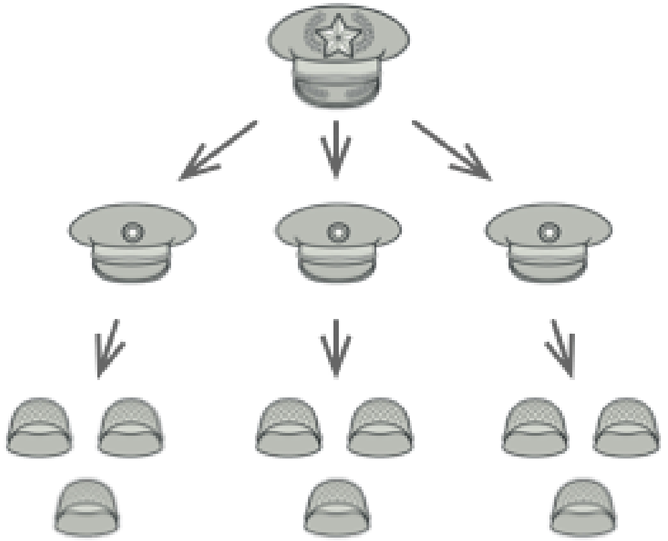


Adapter



Facade

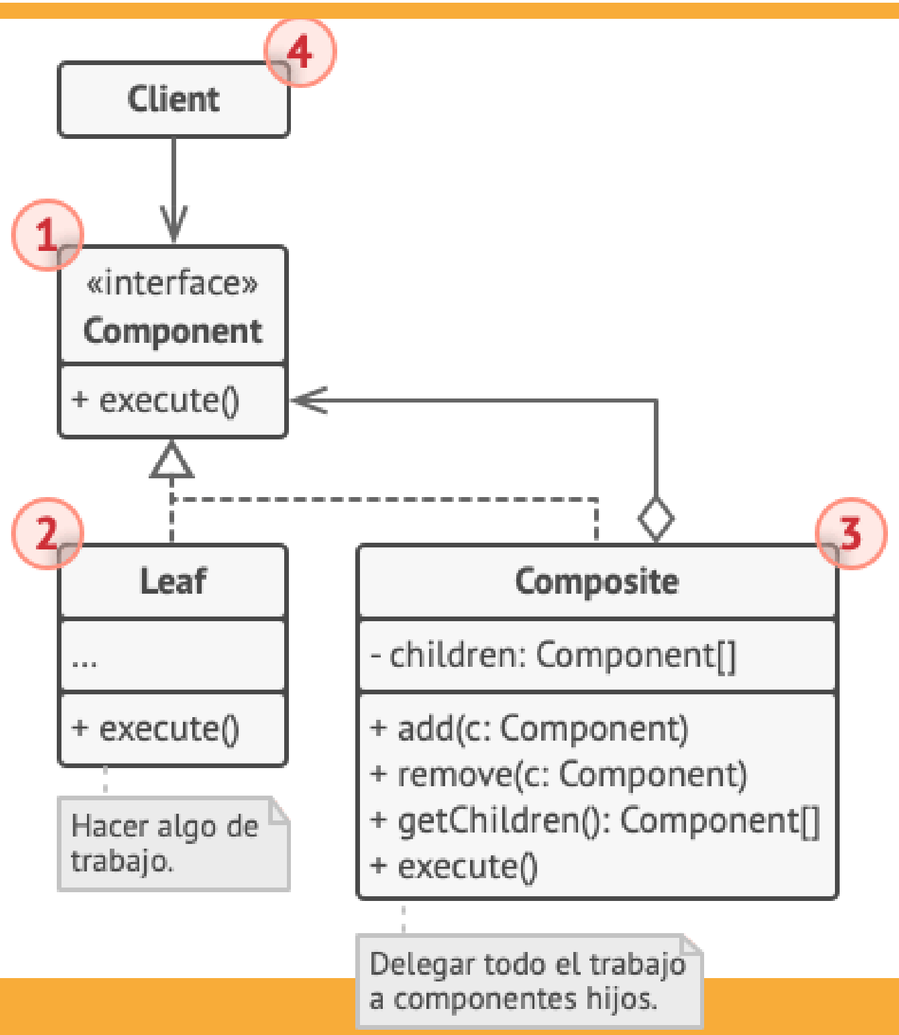
Patrón: Composite



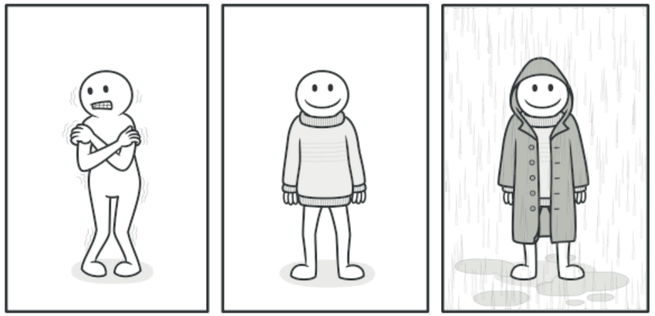
- Permite construir problemas que se pueden modelar como árboles, y genera un método común a todas las partes del árbol para realizar operaciones de forma sencilla. Así solo llamando a la raíz del árbol accedes facilmente a todos los sub-elementos.

Patrón: Composite

Estructura:

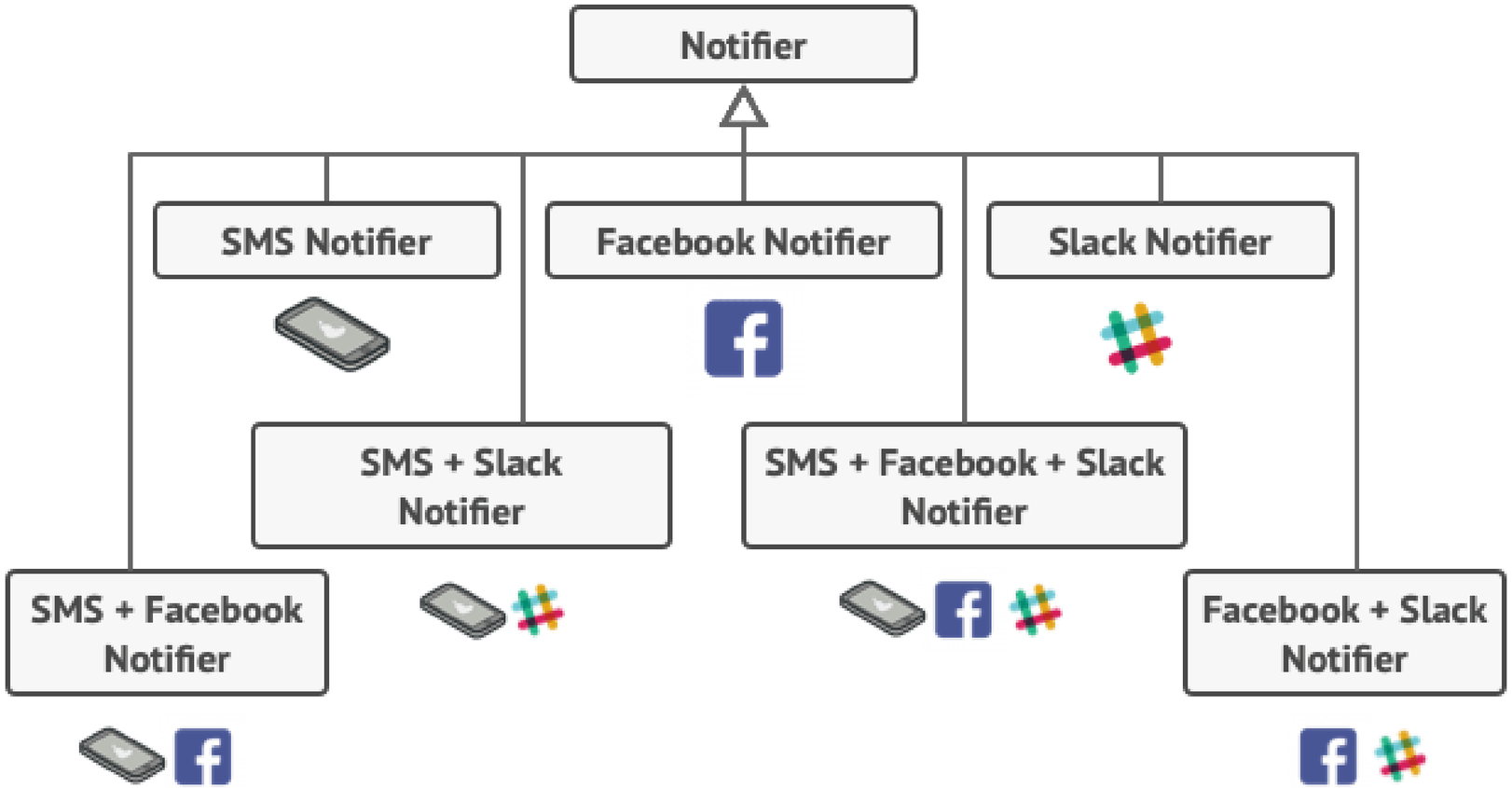


Patrón: Decorator

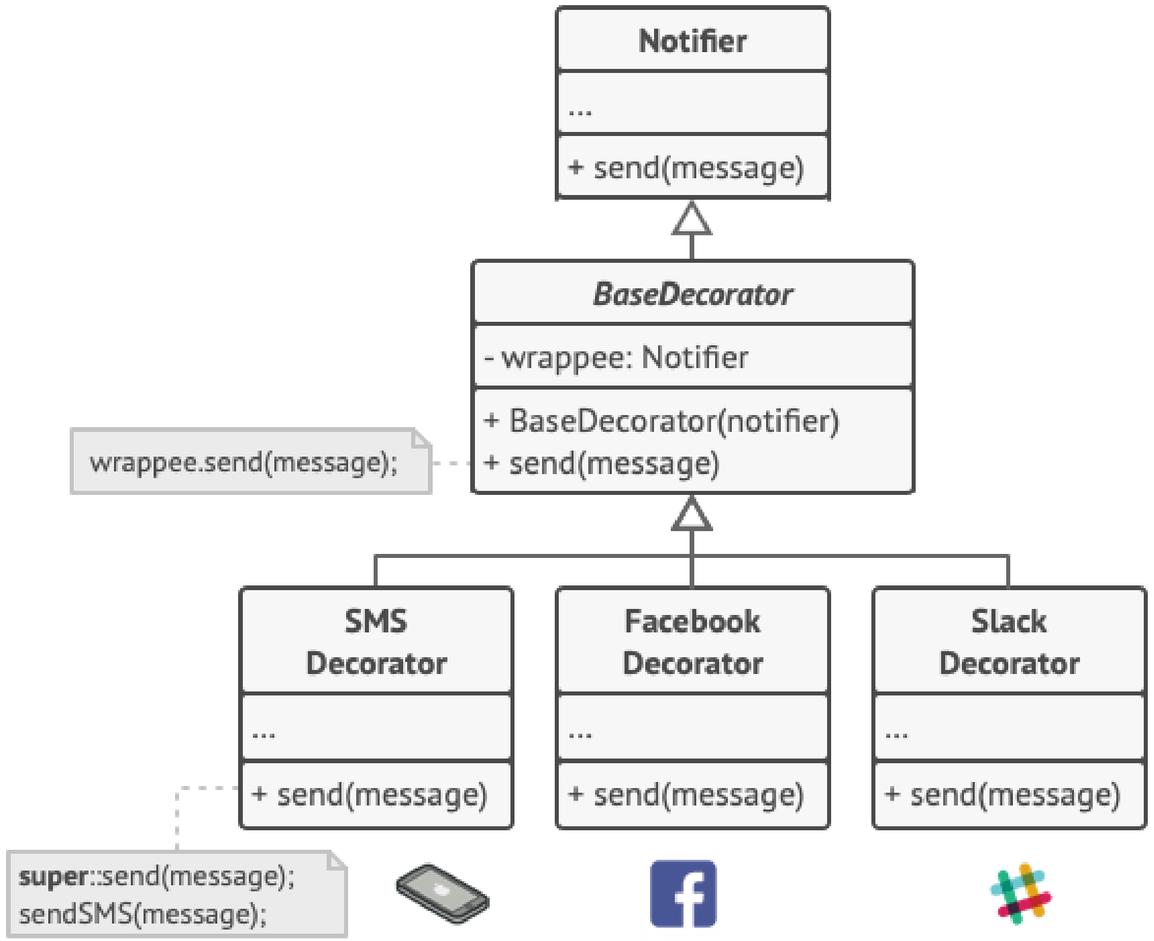


- Este patrón aprovecha la composición sobre la herencia, al componer en capas los objetos te permite tener un objeto con múltiples funcionalidades sin tener que tratar con multiherencias grandes.

Patrón: Decorator

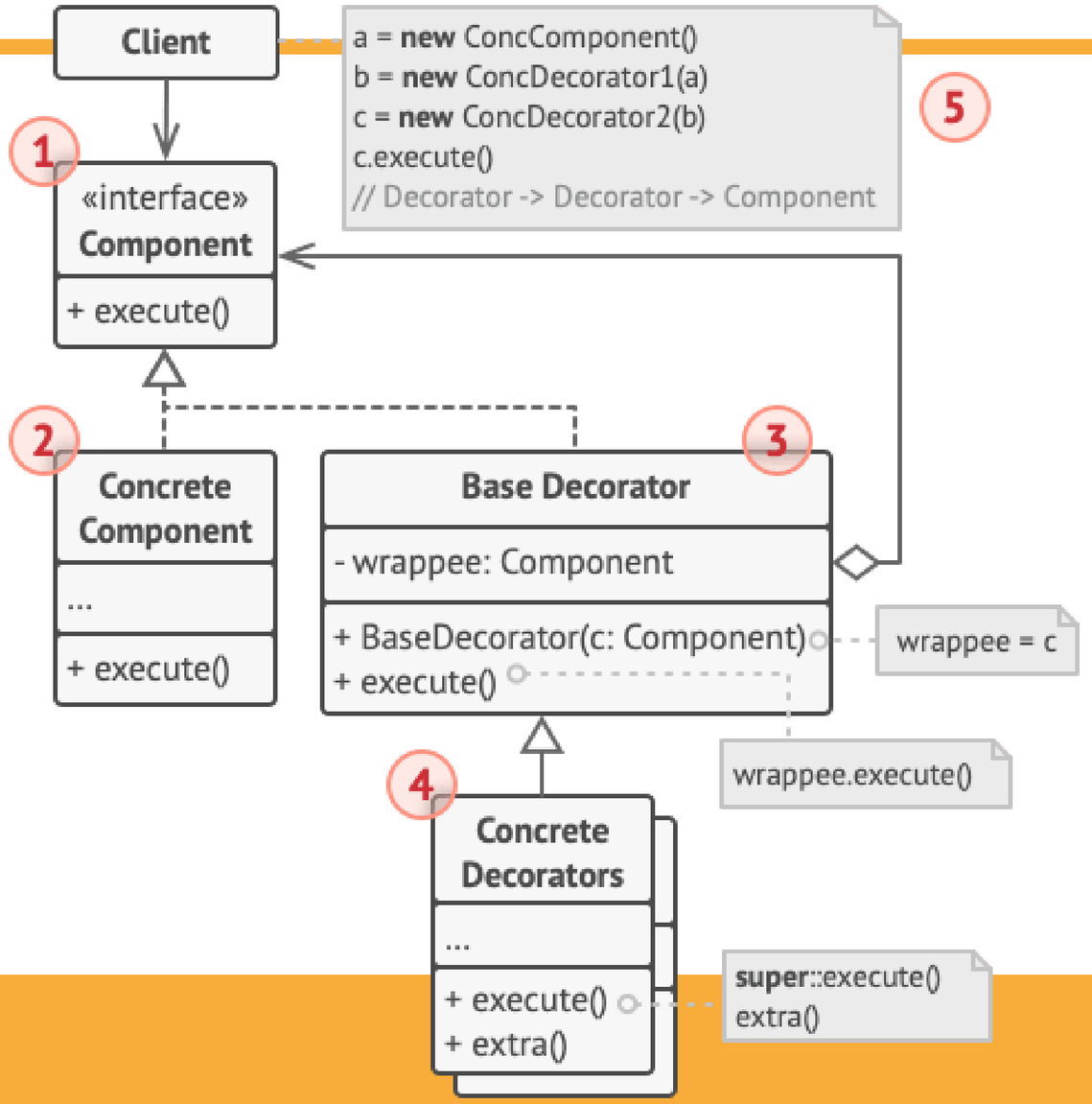


Patrón: Decorator

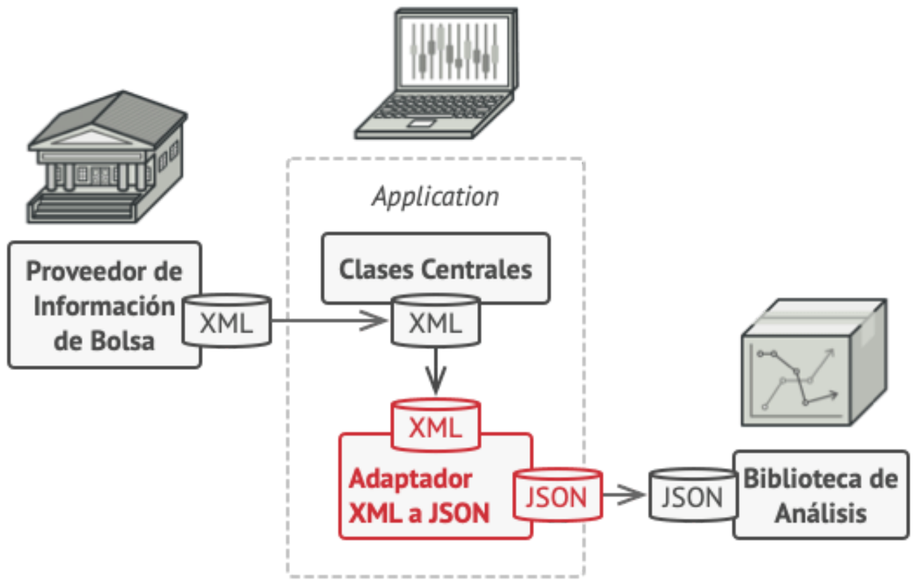


Patrón: Decorator

Estructura:



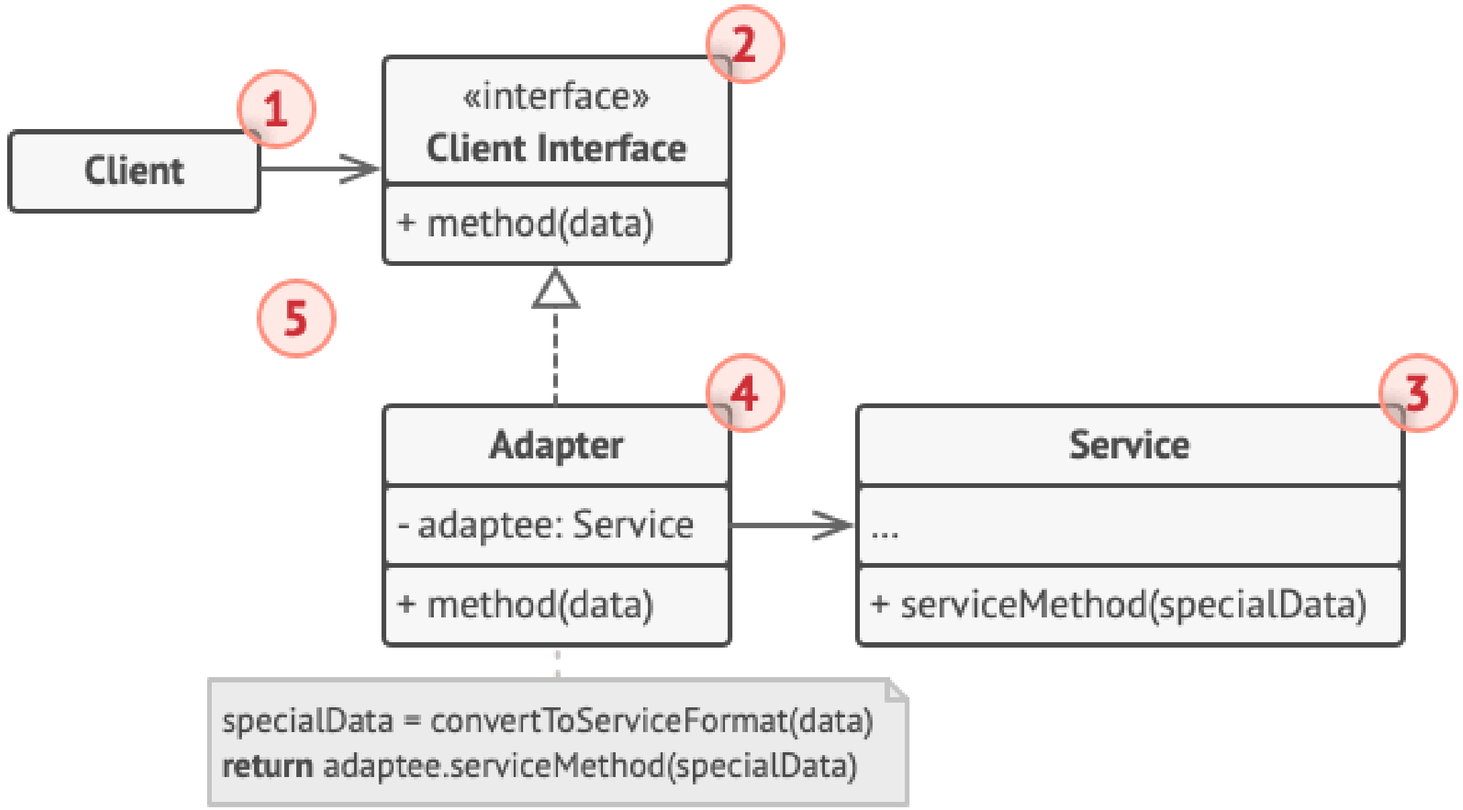
Patrón: Adapter



- Este patrón busca crear una interfaz intermedia entre dos interfaces incompatibles, para así poder integrar diferentes funcionalidades resolviendo las incompatibilidades.

Patrón: Adapter

Estructura:

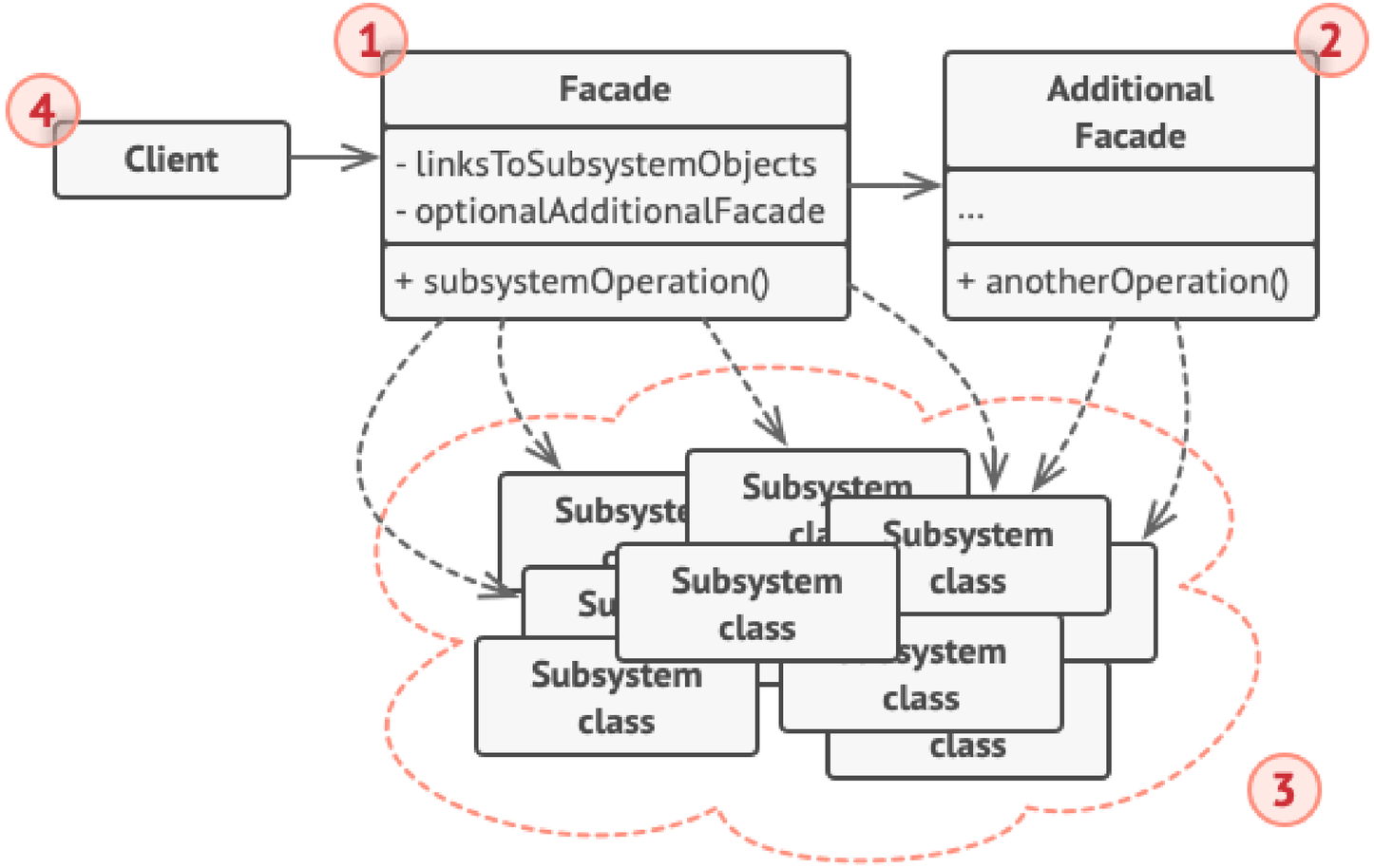


Patrón: Facade

- Este patrón busca simplificar la funcionalidad de un sistema complejo, la gracia está en dar varios métodos sencillos de usar, y que por debajo la fachada ejecute toda la parte compleja, haciendo más fácil usar nuestro sistema.

Patrón: Facade

Estructura:



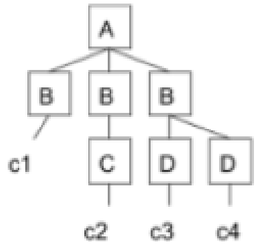
EJERCICIO

Pregunta 5 I2 2021-1

Pregunta 5

Aún no calificado / 25 pts

(25 puntos) Un documento xml tiene estructura de árbol. Un elemento xml (etiqueta) puede contener otros elementos xml o bien solo contenido. Por ejemplo, el árbol de la figura corresponde al documento xml que sigue mas abajo.



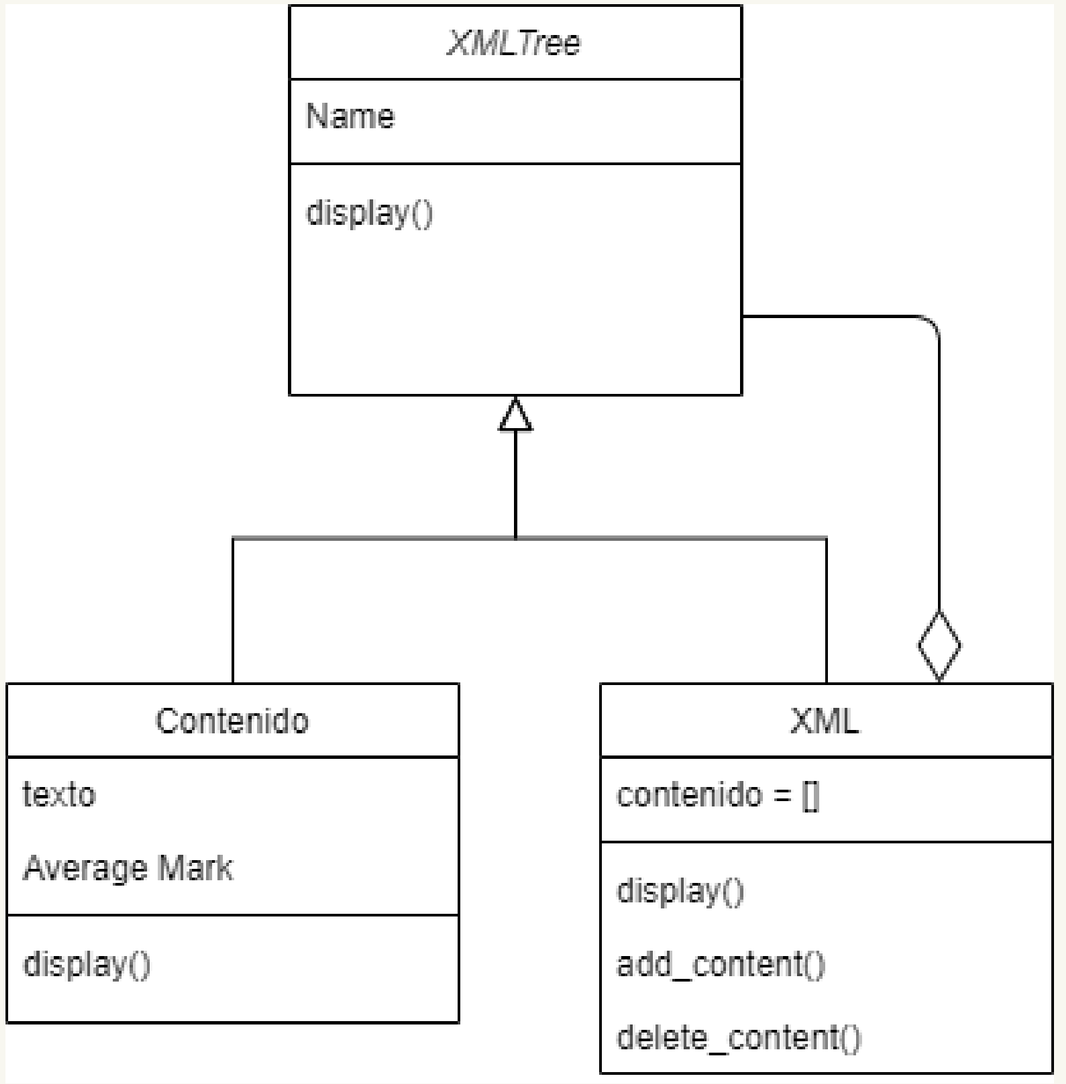

```
<A>
  <B> c1 </B>
  <B>
    <C> c2 </C>
  </B>
  <B>
    <D> c3 </D>
    <D> c4 </D>
  </B>
</A>
```

a) (15 puntos) Utiliza el patrón de diseño conocido como *Composite* para implementar una clase XmlTree con un método *display*que genere la versión en texto del árbol. Haga un diagrama de clases con todo lo necesario (métodos y atributos) y escriba (Ruby) el método display. No te preocupes por identación o espacios, puedes mostrar el árbol anterior como:

```
<A> <B> c1 </B> <B> <C> c2 </C> </B> <B> <D> c3 </D> <D> c4 </D> </B> </A>
```

b) (10 puntos) Escribe un trozo de código Ruby que construye el árbol de la figura como un objeto compuesto y luego lo imprime como documento *xml* usando el método display

Diragrama_I2_pregunta_composite.png



```ruby
class Component
    def display
        raise "Abstract method display"
    end
end
```

```ruby
class Content < Component
    def initialize(text)
        @text = text
    end

    def display()
        print "#{@text}"
    end
end
```

```ruby
class XMLTree < Component
    def initialize(tag_name)
        @tag_name = tag_name
        @contents = []
    end

    def add_content(content)
        @contents << content
    end

    def remove_content(content)
        @contents.delete(content)
    end
    def display
        print "<#{@tag_name}> "
        @contents.each() do |component|
            content = component.display()
        end
        print " </#{@tag_name}>"
    end
end
```

```ruby
# Creates The tree and leafs
a = XMLTree.new("A")

b = XMLTree.new("B")
a.add_content(b)
content_1 = Content.new("c1")
b.add_content(content_1)

b = XMLTree.new("B")
a.add_content(b)

c = XMLTree.new("C")
b.add_content(c)
content_2 = Content.new("c2")
c.add_content(content_2)

b3 = XMLTree.new("B")
a.add_content(b3)
d = XMLTree.new("D")
b3.add_content(d)
content_3 = Content.new("c3")
d.add_content(content_3)
d = XMLTree.new("D")
content_4 = Content.new("c4")
d.add_content(content_4)
```

Y la parte linda, ahoras solo llamo a un método para que se muestre todo

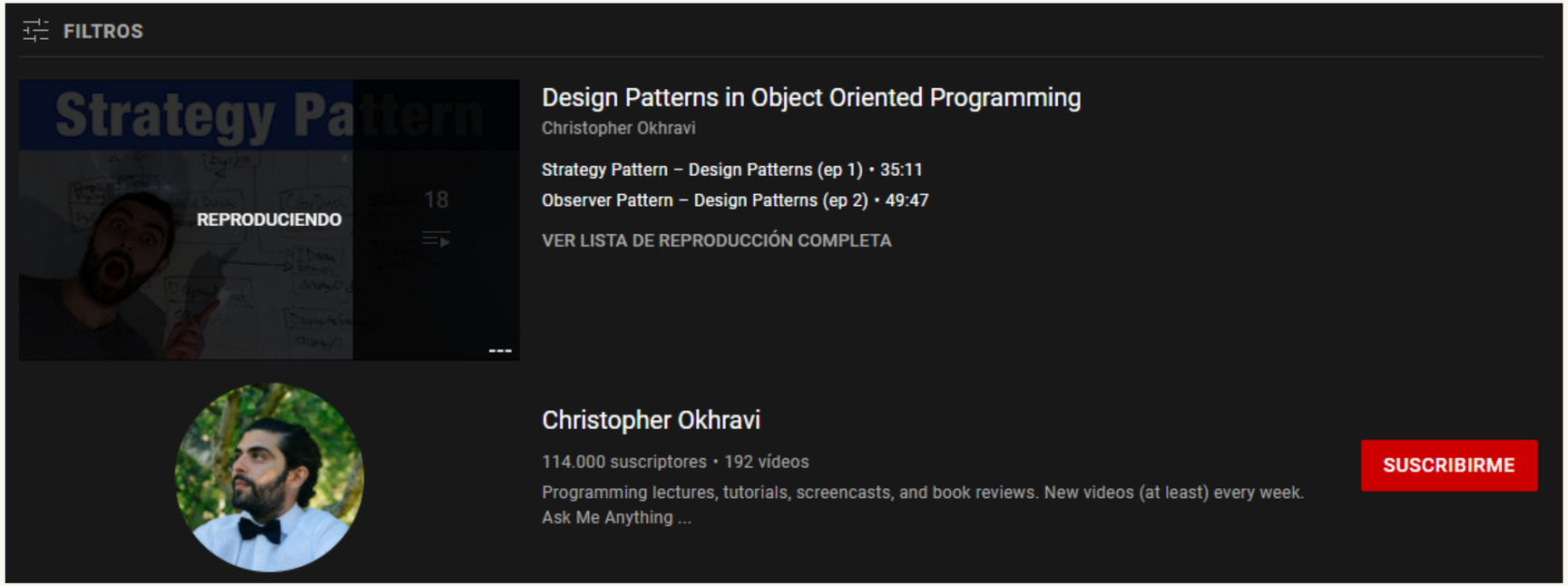
```ruby
# Runs the display to show the
a.display()
```

Bibliografía y links útiles

Refactoring Guru

https://refactoring.guru/es/design-patterns

Christopher Okharvi



https://www.youtube.com/watch?v=v9ejT8FO-7I&list=PLrhzvIcii6GNjpARdnO4ueTUAVR9eMBpc