

Ingeniería de Software

16 -Patrones de Diseño II: Estructurales

IIC2143-3

Josefa España

jpespana@uc.cl

Clase “patrocinada” por Refactoring Guru



Patrones de Diseño

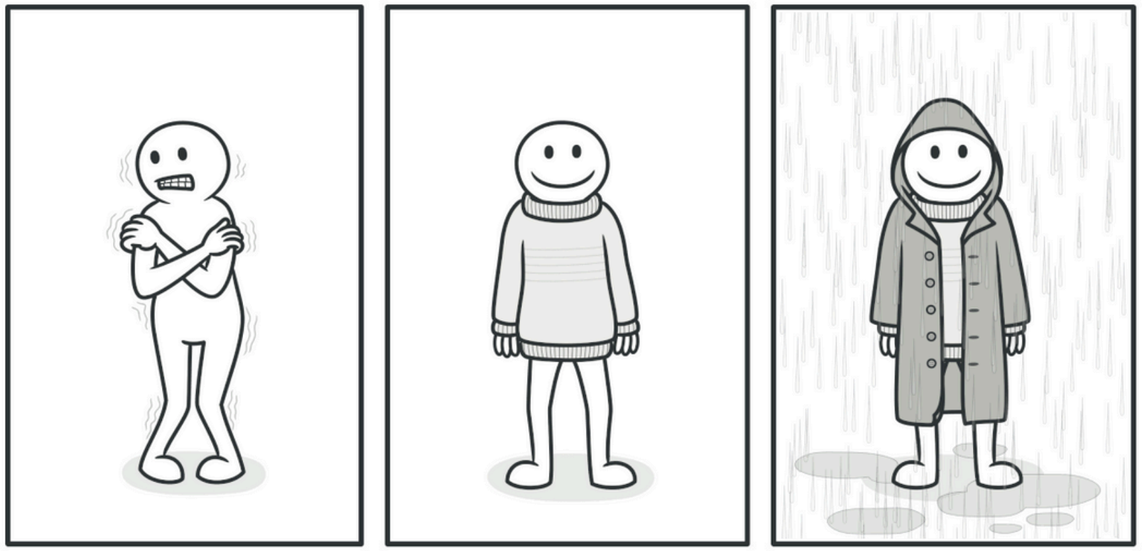
Patrones Estructurales

- Decorator
- Composite
- Adapter
- Proxy

Decorator

Propósito

Decorator es un patrón de diseño estructural que te permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.



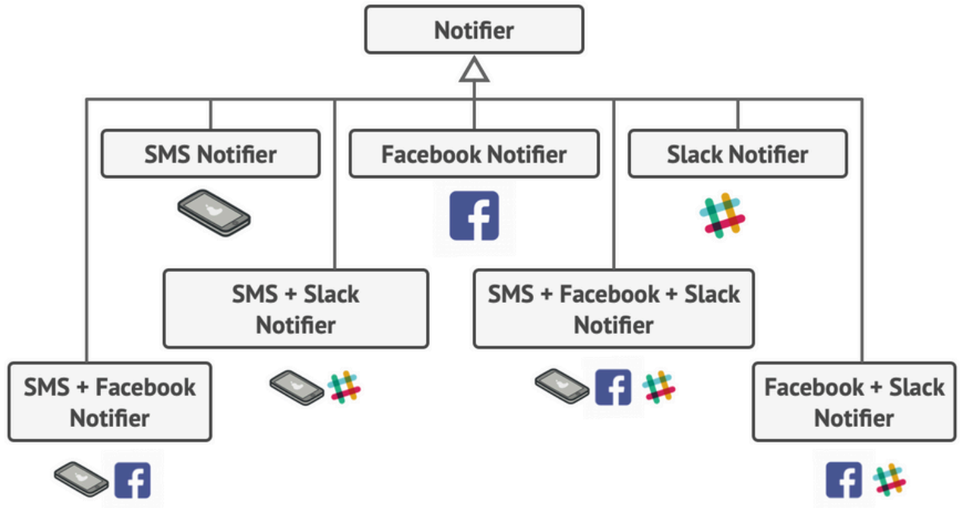
Obtienes un efecto combinado vistiendo varias prendas de ropa.

Decorator

Motivación - Problema

Aplicación de Notificaciones donde solo se puede enviar por un canal: SMS, Slack o Facebook.

- ¿Qué pasa si quiero recibir notificaciones en varios canales a la vez?



Explosión combinatoria de subclases.

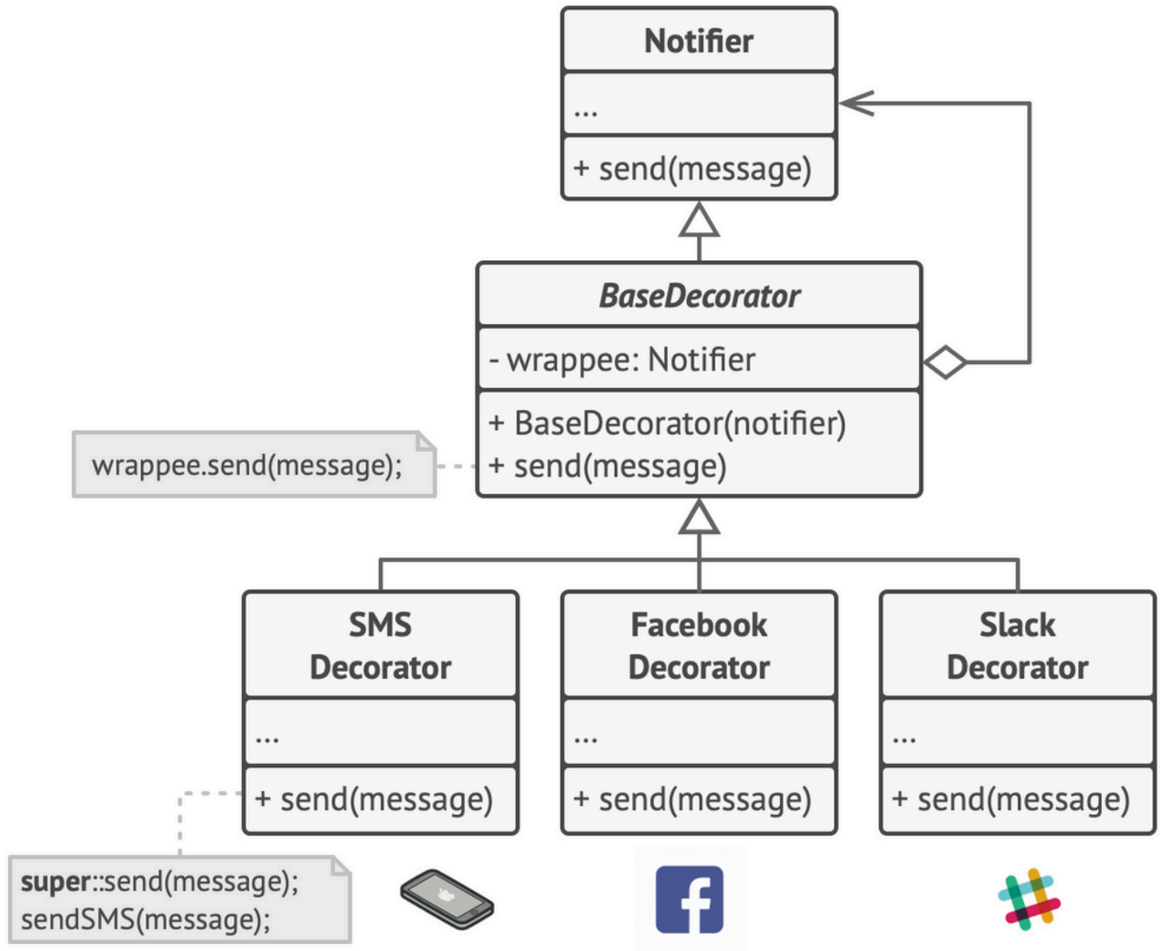
Decorator

Motivación - Solución

- Añadimos Composición: Un objeto tiene referencia a otro y le delega parte del trabajo.
- El objeto decorador tiene como atributo al objeto decorado y normalmente los mismos métodos que el objeto decorado.
- Cuando un método del objeto decorado es llamado, el decorador llama al mismo método del objeto decorado y realiza acciones adicionales.

Decorator

Motivación - Solución



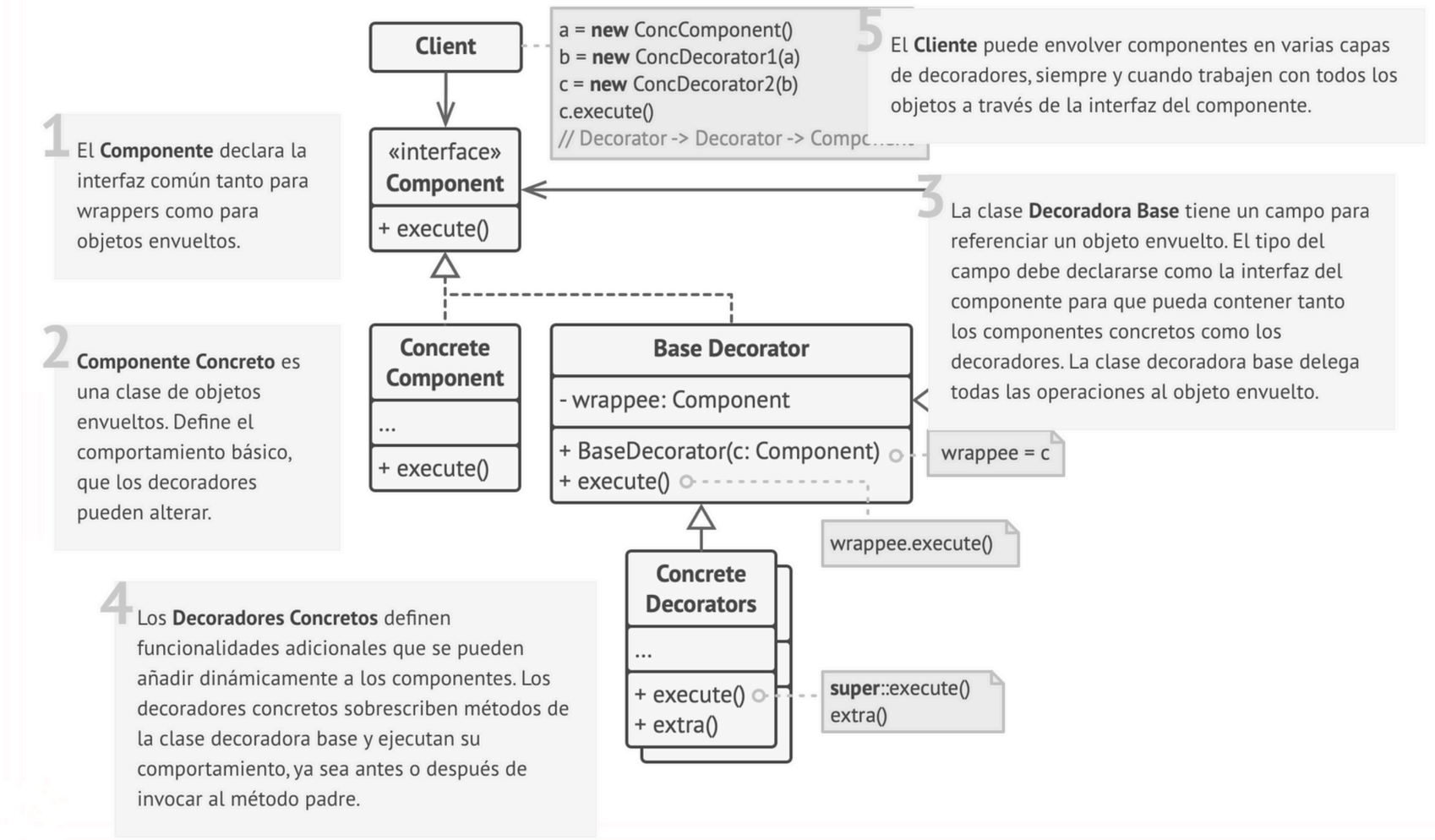
Varios métodos de notificación se convierten en decoradores.

Decorator

Pros y contras

- ✓ Puedes extender el comportamiento de un objeto sin crear una nueva subclase.
- ✓ Puedes añadir o eliminar responsabilidades de un objeto durante el tiempo de ejecución.
- ✓ Puedes combinar varios comportamientos envolviendo un objeto con varios decoradores.
- ✓ *Principio de responsabilidad única*. Puedes dividir una clase monolítica que implementa muchas variantes posibles de comportamiento, en varias clases más pequeñas.
- ✗ Resulta difícil eliminar un wrapper específico de la pila de wrappers.
- ✗ Es difícil implementar un decorador de tal forma que su comportamiento no dependa del orden en la pila de decoradores.
- ✗ El código de configuración inicial de las capas pueden tener un aspecto desagradable.

Decorator


Client
a = new ConcComponent()
b = new ConcDecorator1(a)
c = new ConcDecorator2(b)
c.execute()
// Decorator -> Decorator -> Compo
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.
«interface» Component
+ execute()
1 El Componente declara la interfaz común tanto para wrappers como para objetos envueltos.
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.
Concrete Component
...
+ execute()
Base Decorator
- wrappee: Component
+ BaseDecorator(c: Component)
+ execute()
wrappee = c
wrappee.execute()
2 Componente Concreto es una clase de objetos envueltos. Define el comportamiento básico, que los decoradores pueden alterar.
Concrete Decorators
...
+ execute()
+ extra()
super::execute()
extra()
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.

Decorator - Ejemplo

Motivación - Problema

```ruby
class Pizza
  def print
    raise NotImplementedError
  end

  def cost
    raise NotImplementedError
  end
end
```

```ruby
require_relative 'pizza'

class HamCheesePizza < Pizza
  def print
    puts 'normal bred'
    puts 'cheese'
    puts 'ham'
  end

  def cost
    20 + 5 + 5
  end
end
```

Decorator - Ejemplo

Motivación - Problema

```ruby
class Pizza
  def print
    raise NotImplementedError
  end

  def cost
    raise NotImplementedError
  end
end
```

```ruby
require_relative 'pizza'
class PineableHamCheesePizza < Pizza
  def print
    puts 'normal bred'
    puts 'cheese'
    puts 'ham'
    puts 'pineapple'
  end

  def cost
    (20 + 5 + 5)* 2
    #cada porcion de pineable duplica el costo
  end
end
```

Decorator - Ejemplo

Motivación - Problema

- Actualmente se agregan los toppings de manera estática. Si queremos hacer una pizza de doble queso hay que crear otra clase.

- Tenemos código duplicado entre distintos tipos de pizza.

```
require_relative 'ham_cheese_normal_pizza'
require_relative 'pineapple_ham_cheese_normal_pizza'

hawaii_pizza = PineableHamCheesePizza.new
hawaii_pizza.print
puts "#{hawaii_pizza.cost}"

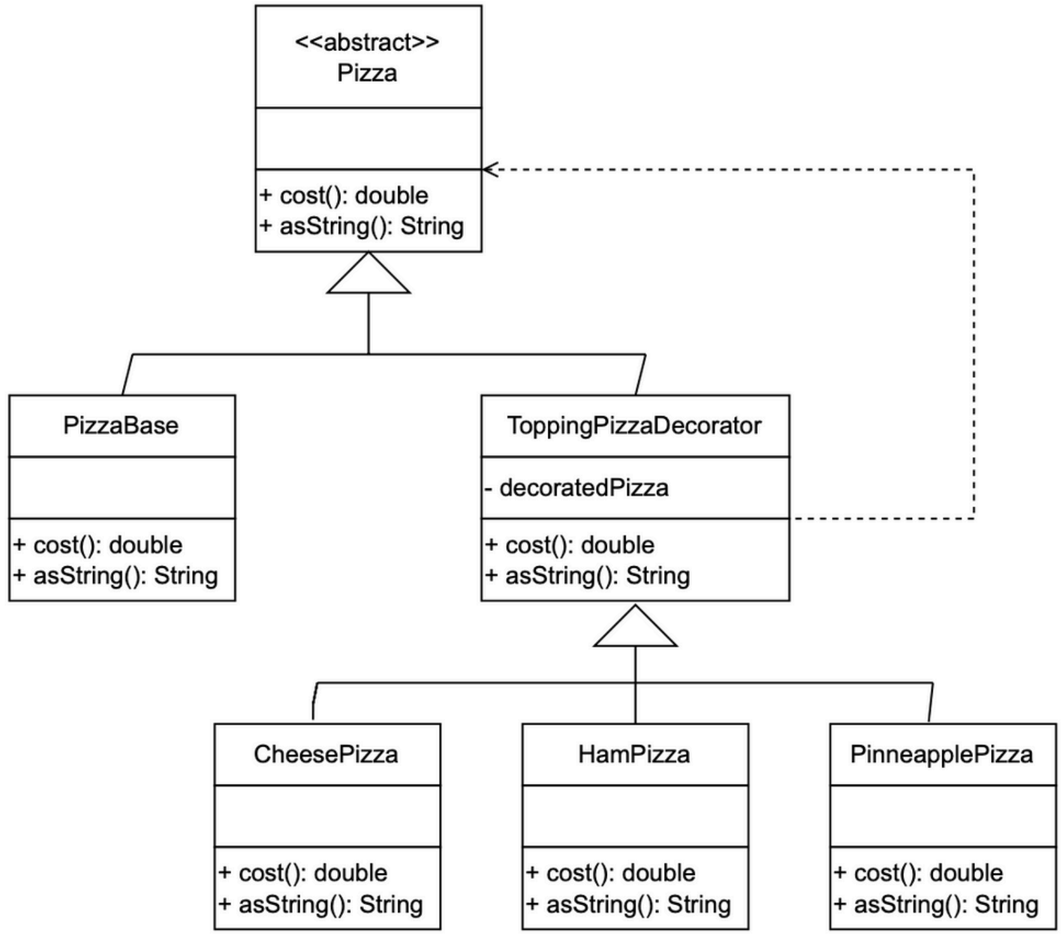

basic_pizza = HamCheesePizza.new
basic_pizza.print
puts "#{basic_pizza.cost}"
```

Decorator - Ejemplo

Motivación - Solución

```ruby
class Pizza
  def print
    raise NotImplementedError
  end

  def cost
    raise NotImplementedError
  end
end
```



Decorator - Ejemplo

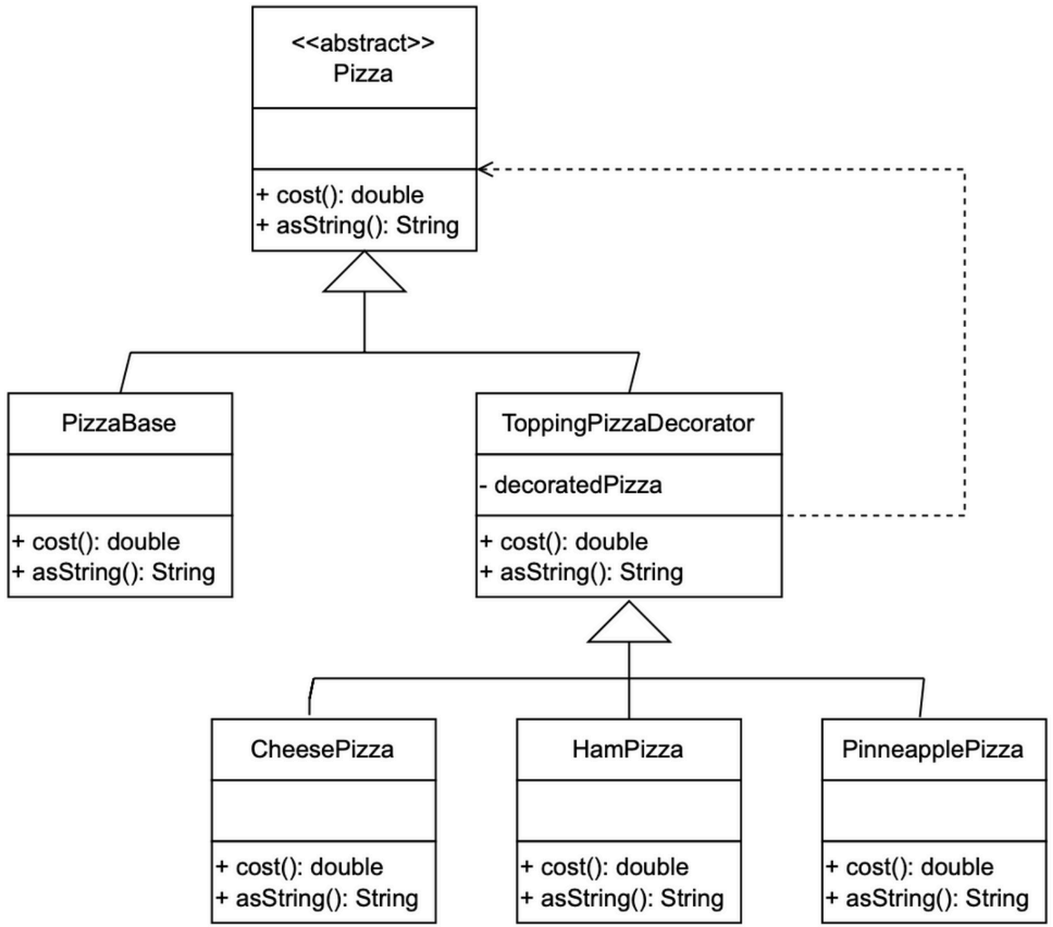
Motivación - Solución

```ruby
require_relative 'pizza'

class BasePizza < Pizza

  def print
    puts 'normal bred'
  end

  def cost
    20
  end
end
```

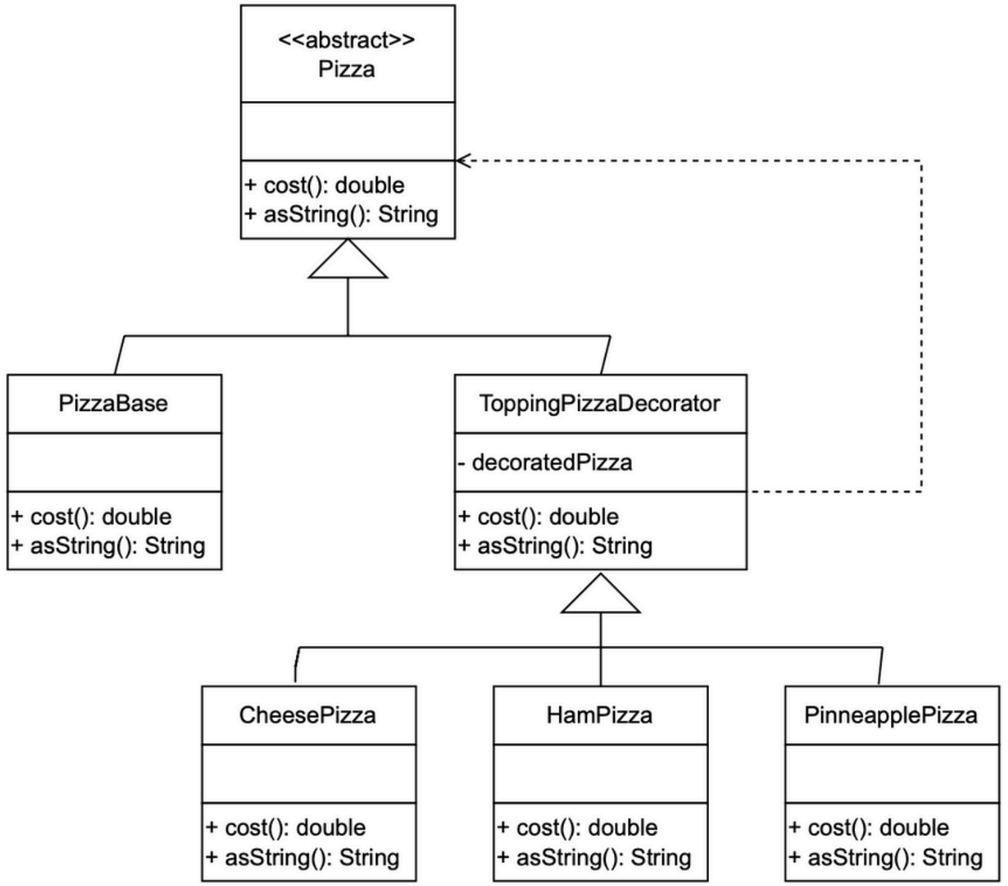


Decorator - Ejemplo

Motivación - Solución

```ruby
require_relative 'pizza'

class ToppingPizzaDecorator < Pizza
  def initialize(decoratedPizza)
    @decoratedPizza = decoratedPizza
  end
  def cost
    @decoratedPizza.cost
  end
  def print
    @decoratedPizza.print
  end
end
```



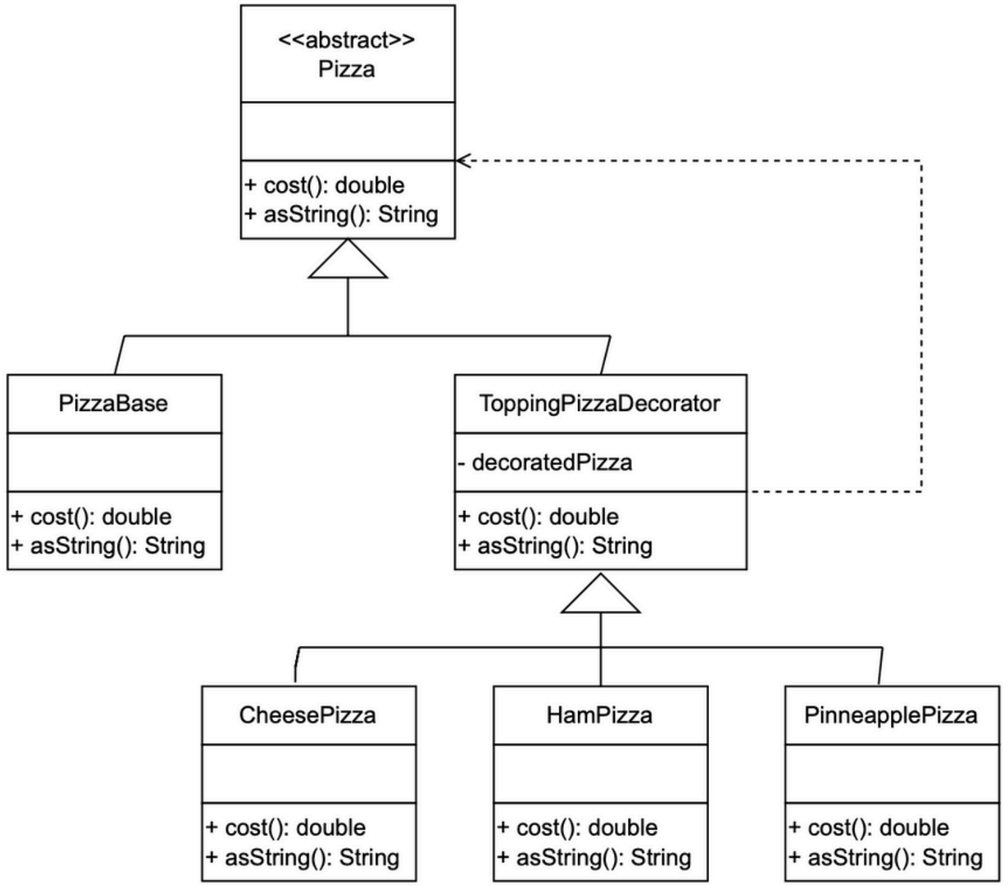
Decorator - Ejemplo

Motivación - Solución

```ruby
require_relative 'topping'

class Cheese < ToppingPizzaDecorator

  def print
    @decoratedPizza.print
    puts 'cheese'
  end
  def cost
    return @decoratedPizza.cost  + 5
  end
end
```



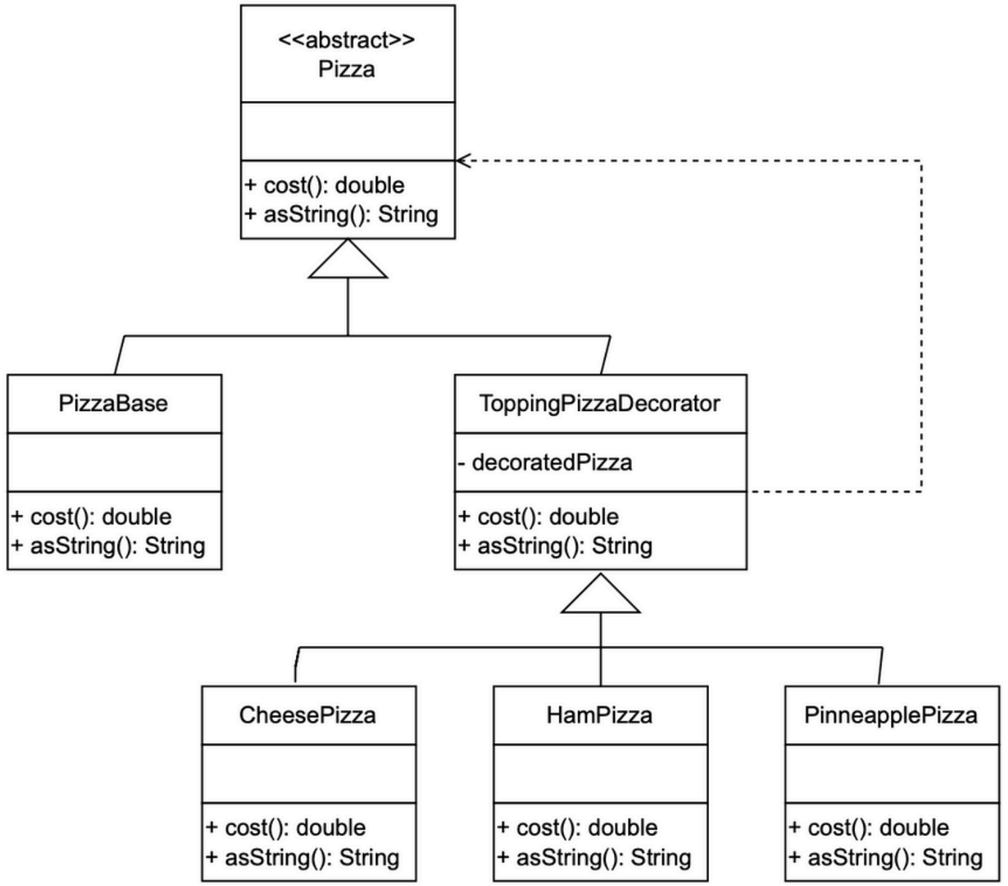
Decorator - Ejemplo

Motivación - Solución

```ruby
require_relative 'topping'

class Pineapple < ToppingPizzaDecorator

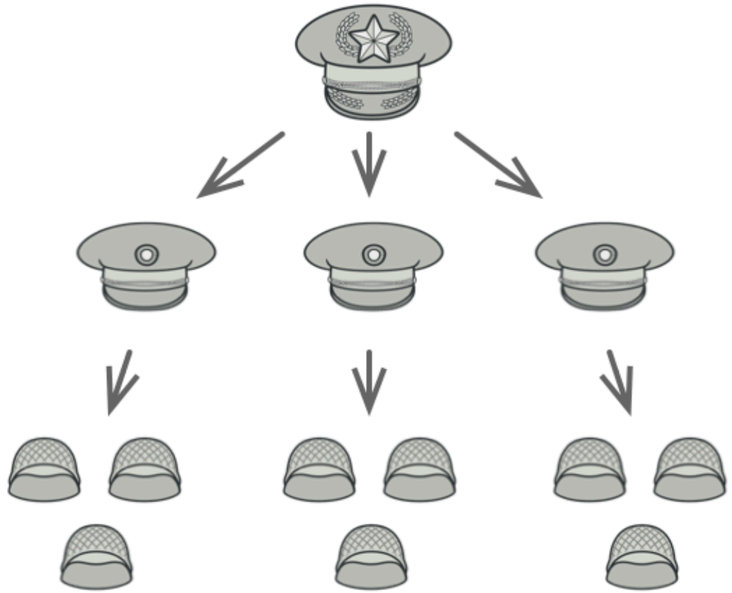
  def print
    @decoratedPizza.print
    puts 'pineapple'
  end
  def cost
    return @decoratedPizza.cost * 2
  end
end
```



Composite

Propósito

Composite es un patrón de diseño estructural que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.



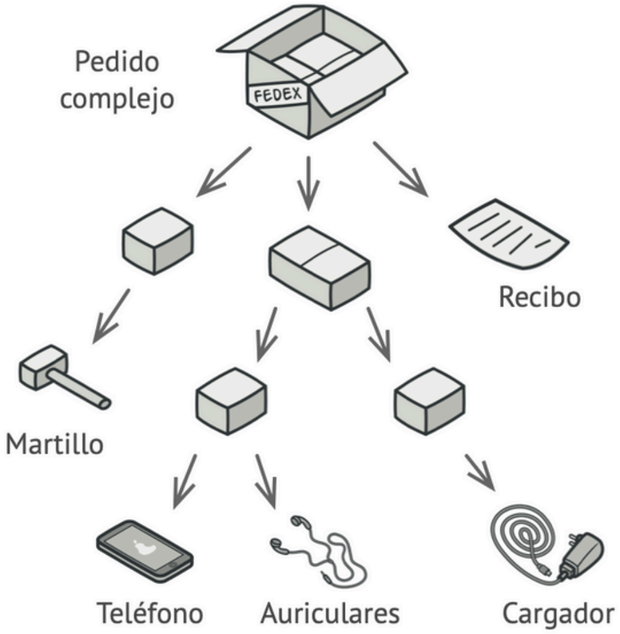
Un ejemplo de estructura militar.

Composite

Motivación - Problema

Este patrón solo tiene sentido si tenemos una estructura de árbol, por ejemplo, tenemos pedidos con cajas y objetos, y una caja puede contener más cajas hasta llegar a un producto.

- ¿Cómo determinamos el precio total del producto?
- En código no es bueno en performance buscar hasta encontrar el producto.

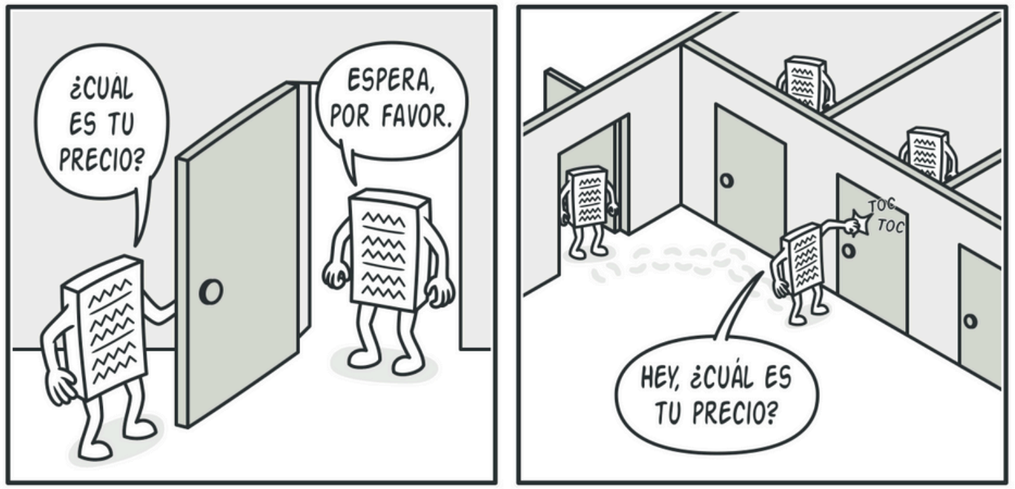


Composite

Motivación - Solución

- Composite sugiere trabajar con Productos y Cajas a través de una interfaz común que declara un método para calcular el precio total

- Para un producto, devuelve el precio del producto.
- Para una caja, recorre cada artículo que contiene la caja, pregunta su precio y devuelve un total por la caja. Si uno de esos artículos fuera una caja más pequeña, esa caja también comenzaría a repasar su contenido y así sucesivamente, hasta que se calcule el precio de todos los componentes internos.



El patrón Composite te permite ejecutar un comportamiento de forma recursiva sobre todos los componentes de un árbol de objetos.

Composite

⚖ Pros y contras

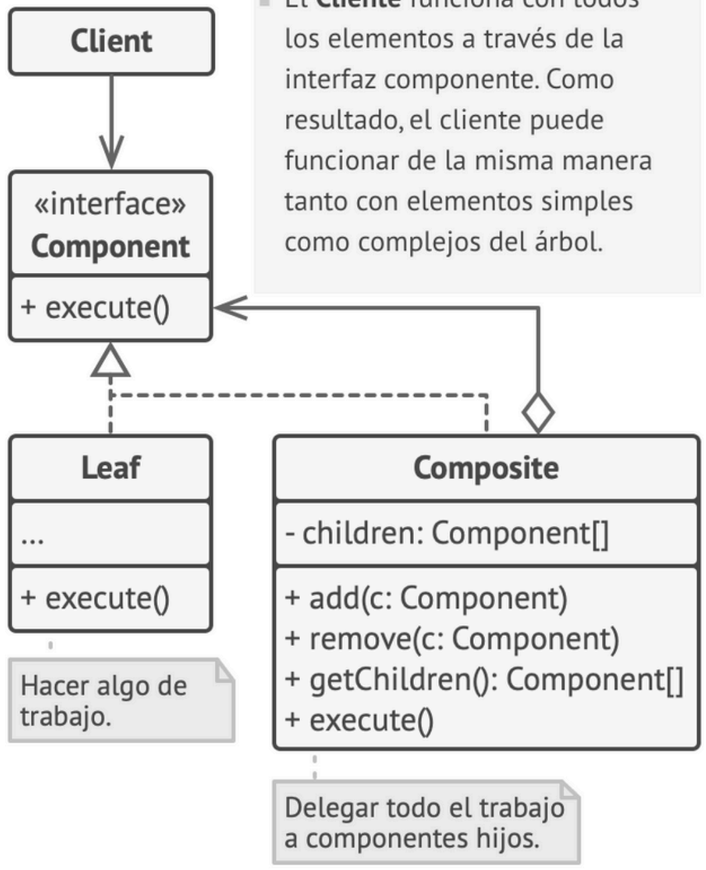
- ✓ Puedes trabajar con estructuras de árbol complejas con mayor comodidad: utiliza el polimorfismo y la recursión en tu favor.
- ✓ *Principio de abierto/cerrado*. Puedes introducir nuevos tipos de elemento en la aplicación sin descomponer el código existente, que ahora funciona con el árbol de objetos.
- ✗ Puede resultar difícil proporcionar una interfaz común para clases cuya funcionalidad difiere demasiado. En algunos casos, tendrás que generalizar en exceso la interfaz componente, provocando que sea más difícil de comprender.

Composite

1 La interfaz **Componente** describe operaciones que son comunes a elementos simples y complejos del árbol.

2 La **Hoja** es un elemento básico de un árbol que no tiene subelementos.

Normalmente, los componentes de la hoja acaban realizando la mayoría del trabajo real, ya que no tienen a nadie a quien delegarle el trabajo.



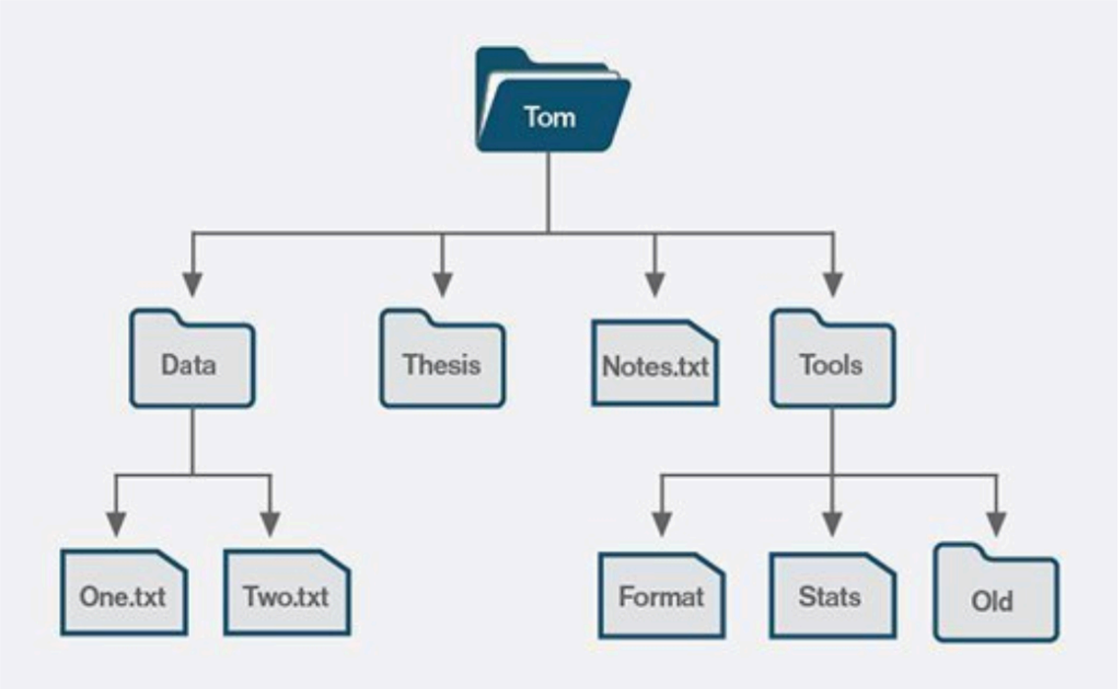
4 El **Cliente** funciona con todos los elementos a través de la interfaz componente. Como resultado, el cliente puede funcionar de la misma manera tanto con elementos simples como complejos del árbol.

3 El **Contenedor** (también llamado *compuesto*) es un elemento que tiene subelementos: hojas u otros contenedores. Un contenedor no conoce las clases concretas de sus hijos. Funciona con todos los subelementos únicamente a través de la interfaz componente.

Al recibir una solicitud, un contenedor delega el trabajo a sus subelementos, procesa los resultados intermedios y devuelve el resultado final al cliente.

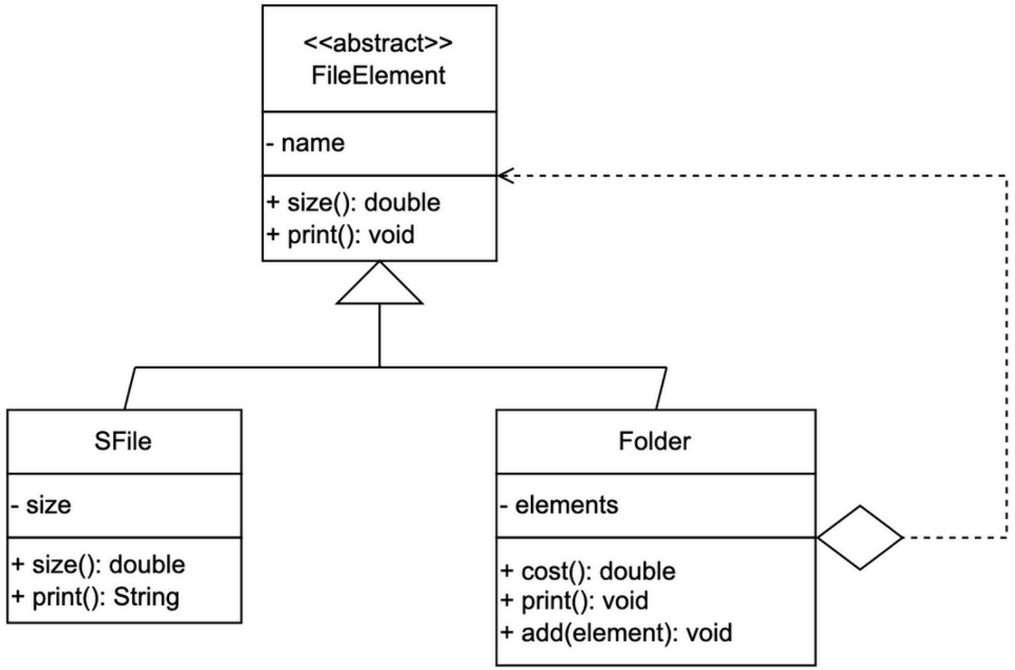
Composite - Ejemplo

- Cada elemento del sistema de archivos puede ser un archivo simple o un folder.
- El folder a su vez puede tener sub-folders o archivos simples.
- Por lo anterior se puede ver que la estructura del file sistema está compuesta.



Composite - Ejemplo

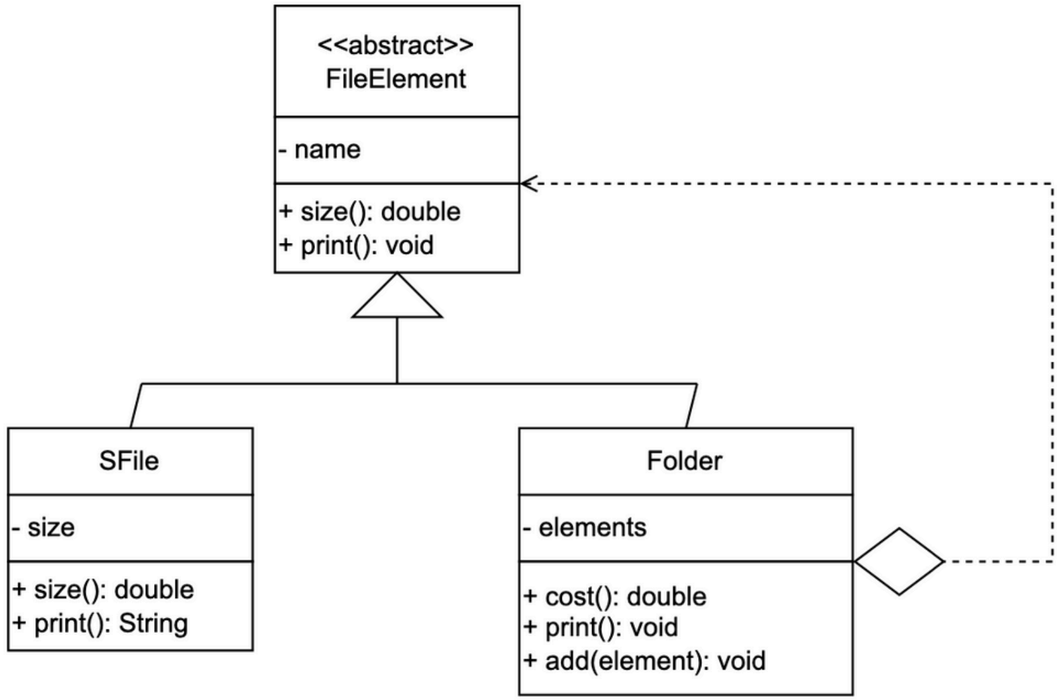
```
class FileElement
  def print
    raise NotImplementedError
  end
  def size
    raise NotImplementedError
  end
end
```



Composite - Ejemplo

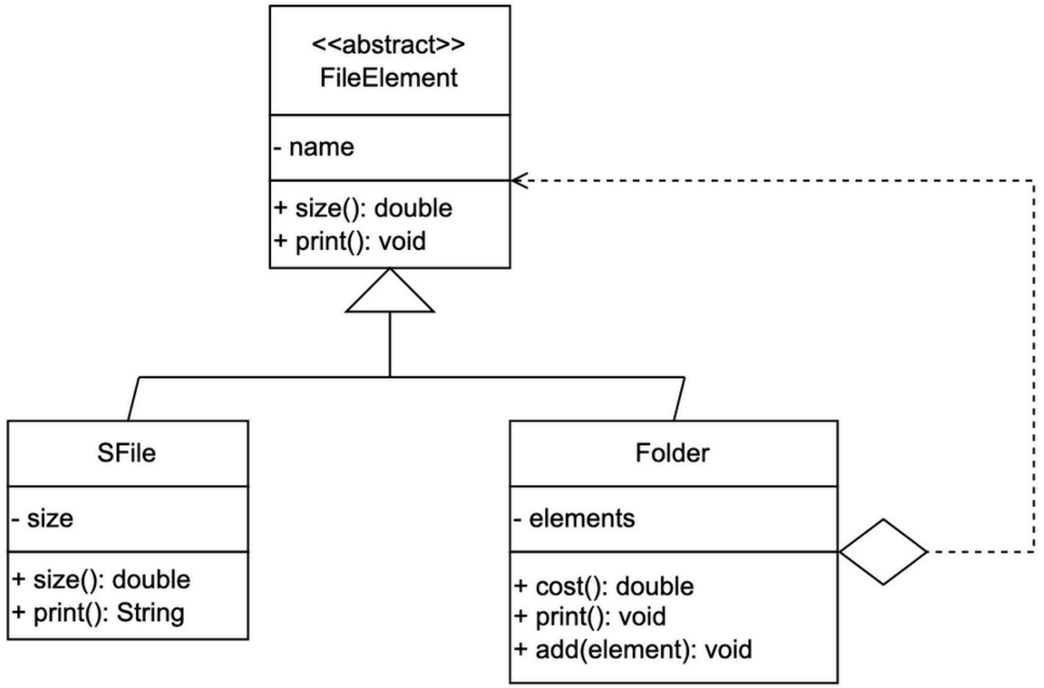
```ruby
require_relative 'element'

class SFile < FileElement
  def initialize(name,size)
    @size = size
    @name = name
  end
  def size
    @size
  end
  def print
    puts "#{@name} : #{@size}"
  end
end
```



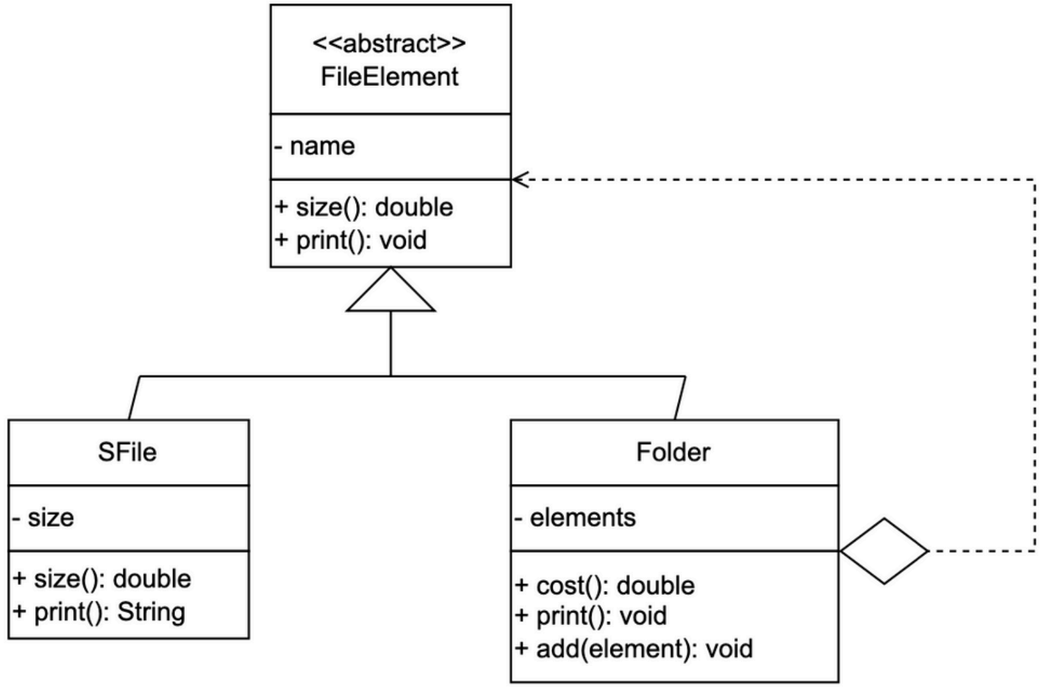
Composite - Ejemplo

```
class FileElement
  def print
    raise NotImplementedError
  end
  def size
    raise NotImplementedError
  end
end
```



Composite - Ejemplo

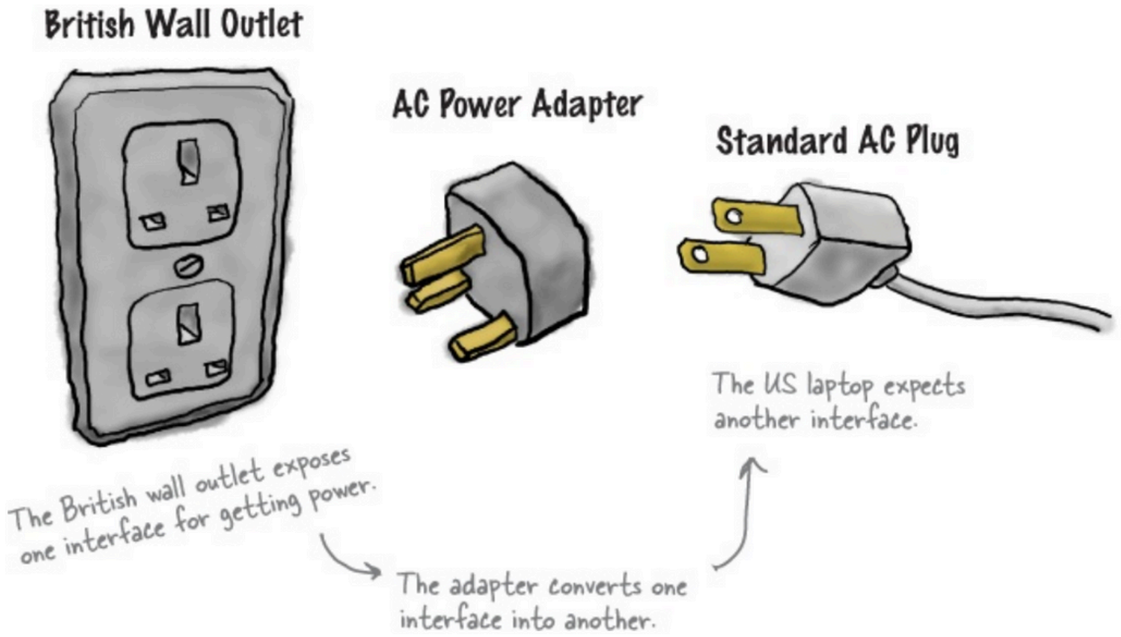
```ruby
class SFolder < FileElement
  ...
  def print
    puts "#{@name}"
    @elements.each do |each|
      each.print
    end
  end
  def size
    total = 0
    @elements.each do |each|
      total = total + each.size
    end
    total
  end
end
```



Adapter

Propósito

Adapter es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles.

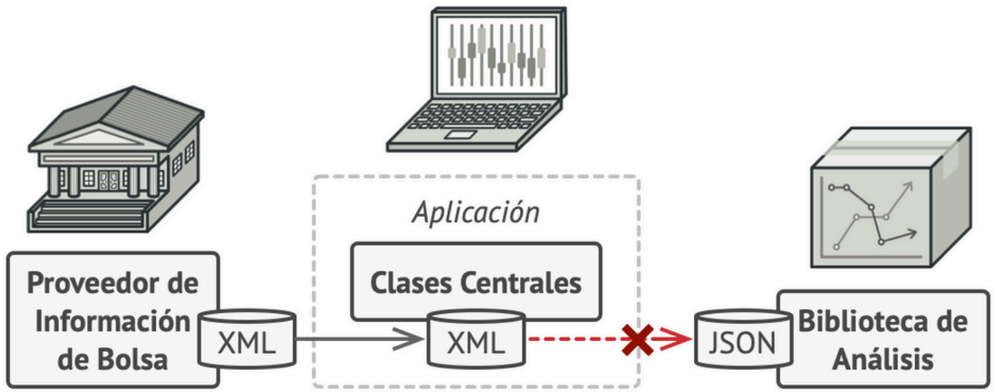


Adapter

Motivación - Problema

- Aplicación de monitoreo del mercado de valor. Descarga información XML para mostrarla en gráficos.
- Versión 2: Integración con biblioteca de análisis que solo funciona con JSON.

Problema: Incompatibilidad de formatos



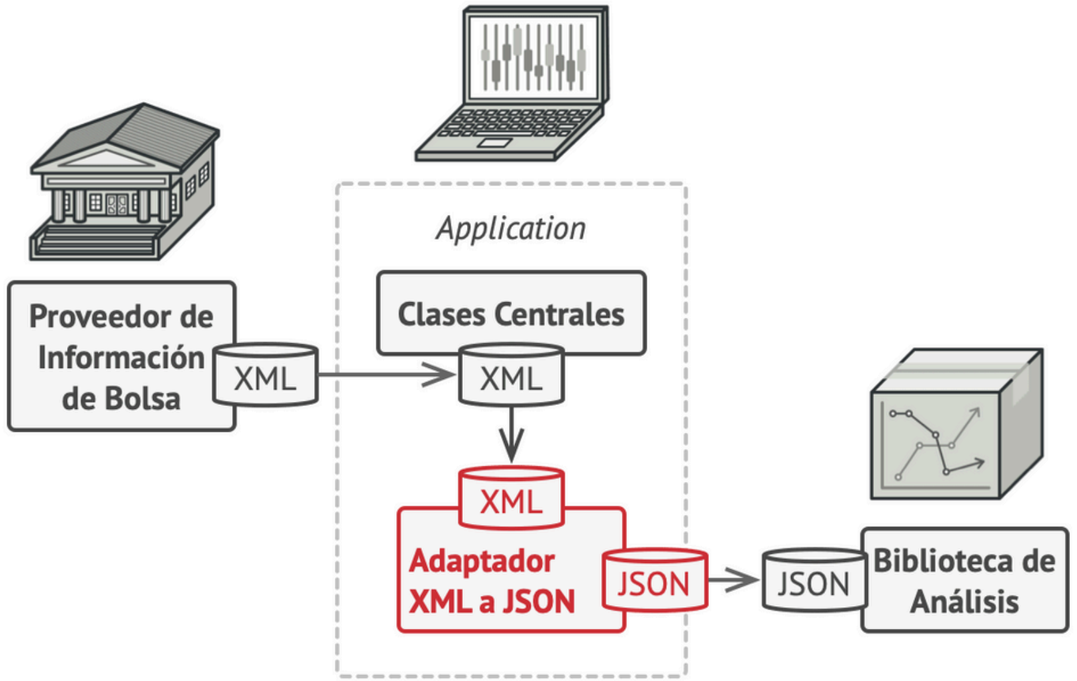
No puedes utilizar la biblioteca de análisis "tal cual" porque ésta espera los datos en un formato que es incompatible con tu aplicación.

Adapter

Motivación - Solución

- Crear un adaptador para convertir la interfaz de un objeto, de tal forma que otro objeto pueda entenderla.

1. El adaptador obtiene una interfaz compatible con uno de los objetos existentes.
2. El objeto existente puede invocar con la interfaz los métodos del adaptador.
3. Al recibir una llamada, el adaptador pasa la solicitud al segundo objeto, pero en el formato que este objeto espera.



Adapter

⚖ Pros y contras

✓ *Principio de responsabilidad única*. Puedes separar la interfaz o el código de conversión de datos de la lógica de negocio primaria del programa.

✓ *Principio de abierto/cerrado*. Puedes introducir nuevos tipos de adaptadores al programa sin descomponer el código cliente existente, siempre y cuando trabajen con los adaptadores a través de la interfaz con el cliente.

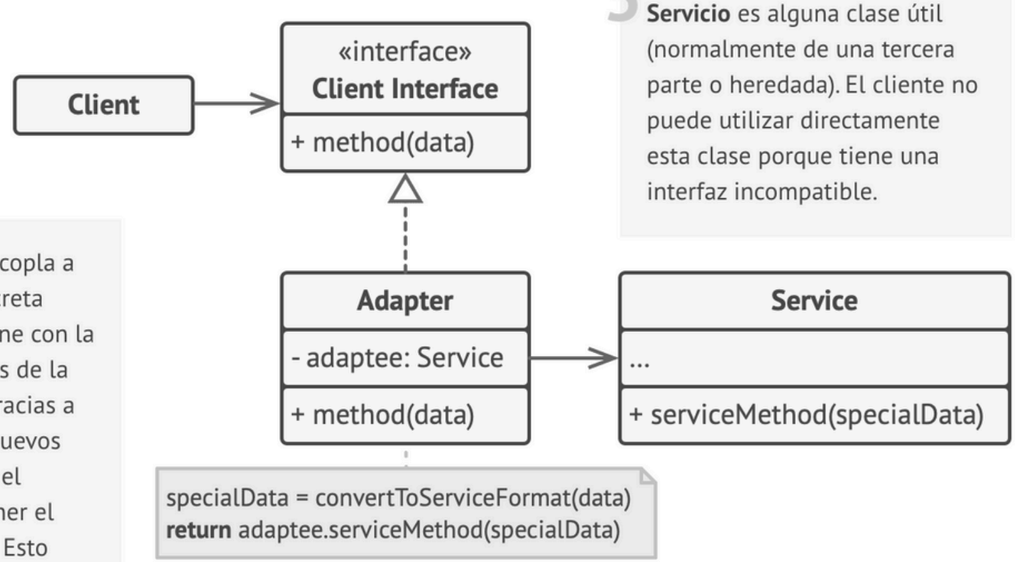
✗ La complejidad general del código aumenta, ya que debes introducir un grupo de nuevas interfaces y clases. En ocasiones resulta más sencillo cambiar la clase de servicio de modo que coincida con el resto de tu código.

Adapter

2 La **Interfaz con el Cliente** describe un protocolo que otras clases deben seguir para poder colaborar con el código cliente.

1 La clase **Cliente** contiene la lógica de negocio existente del programa.

3 **Servicio** es alguna clase útil (normalmente de una tercera parte o heredada). El cliente no puede utilizar directamente esta clase porque tiene una interfaz incompatible.



5 El código cliente no se acopla a la clase adaptadora concreta siempre y cuando funcione con la clase adaptadora a través de la interfaz con el cliente. Gracias a esto, puedes introducir nuevos tipos de adaptadores en el programa sin descomponer el código cliente existente. Esto puede resultar útil cuando la interfaz de la clase de servicio se cambia o sustituye, ya que puedes crear una nueva clase adaptadora sin cambiar el código cliente.

4 La clase **Adaptadora** es capaz de trabajar tanto con la clase cliente como con la clase de servicio: implementa la interfaz con el cliente, mientras envuelve el objeto de la clase de servicio. La clase adaptadora recibe llamadas del cliente a través de la interfaz de cliente y las traduce en llamadas al objeto envuelto de la clase de servicio, pero en un formato que pueda comprender.

Adapter - Ejemplo

Motivación - Problema

```ruby
class ConsoleClient
    def initialize(service)
        @service = service
    end
    def run
        rut = "22.456.456-2"
        clave = "secreta"
        if @service.login(rut,clave) then
            puts "autenticacion exitosa"
        else
            puts "autenticacion fallida"
        end
    end
end
```

```ruby
class LoginService
    def initialize()
        @users = { "22456456-2" => "secreta",
                   "11789789-3" => "password",
                   "33456123-4" => "clave" }
    end
    def login_user(rut, pass)
        return @users[rut.strip] == pass.strip
    end
end
```

```ruby
service = LoginService.new
client = ConsoleClient.new(service)
client.run
```

Adapter - Ejemplo

Motivación - Solución

```
class ConsoleClient
    def initialize(service)
        @service = service
    end
    def run
        rut = "22.456.456-2"
        clave = "secreta"
        if @service.login(rut,clave) then
            puts "autenticacion exitosa"
        else
            puts "autenticacion fallida"
        end
    end
end
```

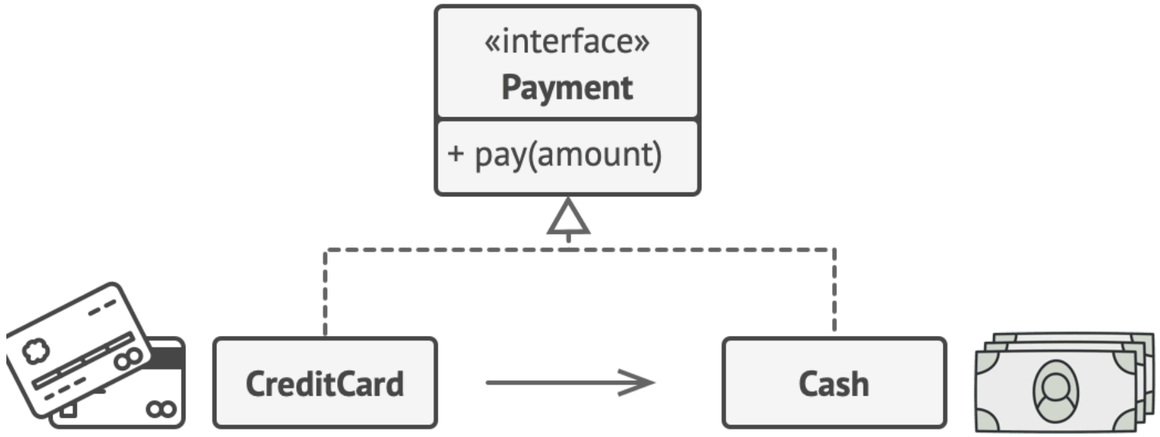
```
class LoginServiceAdapter
    def initialize(originalService)
        @originalService = originalService
    end
    def login(rut, pass)
        adaptedrut = rut.gsub(".", "")
        return   @originalService.login_user(adaptedrut, pass)
    end
end
```

```
service = LoginService.new
adaptador = LoginServiceAdapter.new(service)
client = ConsoleClient.new(adaptador)
client.run
```

Proxy

Propósito

Proxy es un patrón de diseño estructural que te permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original.



Las tarjetas de crédito pueden utilizarse para realizar pagos tanto como el efectivo.

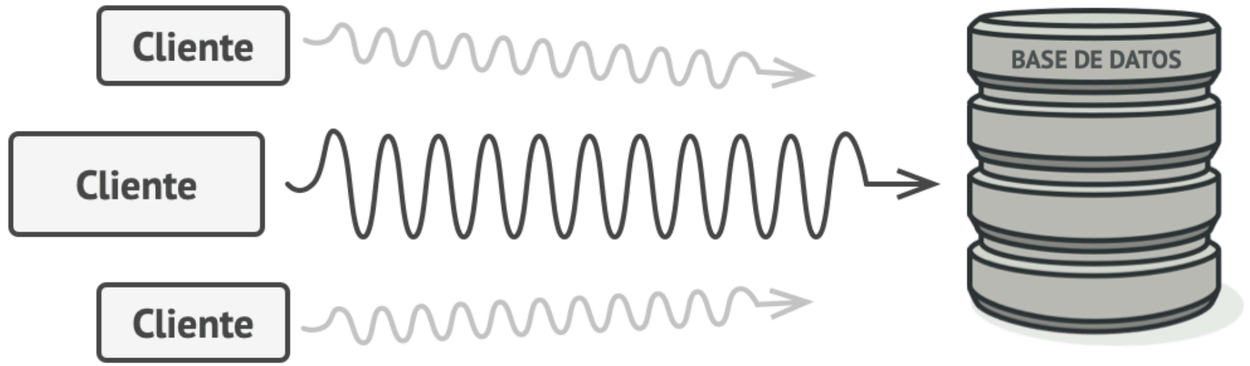
Proxy

Motivación - Problema

- Objeto enorme que consume una gran cantidad de recursos del sistema. Lo necesitas a veces, pero no siempre; como una base de datos

Opciones:

- Crear el objeto solo cuando sea necesario.
  - Genera mucho código duplicado.



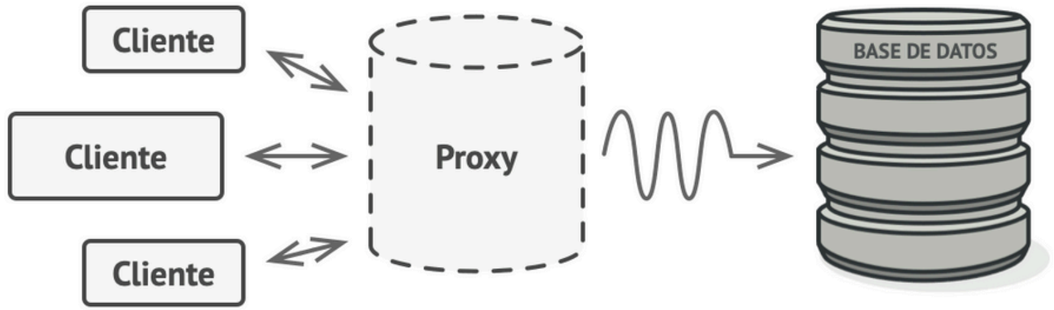
Las consultas a las bases de datos pueden ser muy lentas.

Proxy

Motivación - Solución

- Crear una nueva clase “Proxy” con la misma interfaz que el objeto original. Luego se actualiza la aplicación para pasar el objeto proxy a todos los clientes del objeto original.

- Ventajas: Si necesitamos ejecutar algo entremedio del objeto, no es necesario cambiar la clase.



El proxy se camufla como objeto de la base de datos. Puede gestionar la inicialización diferida y el caché de resultados sin que el cliente o el objeto real de la base de datos lo sepan.

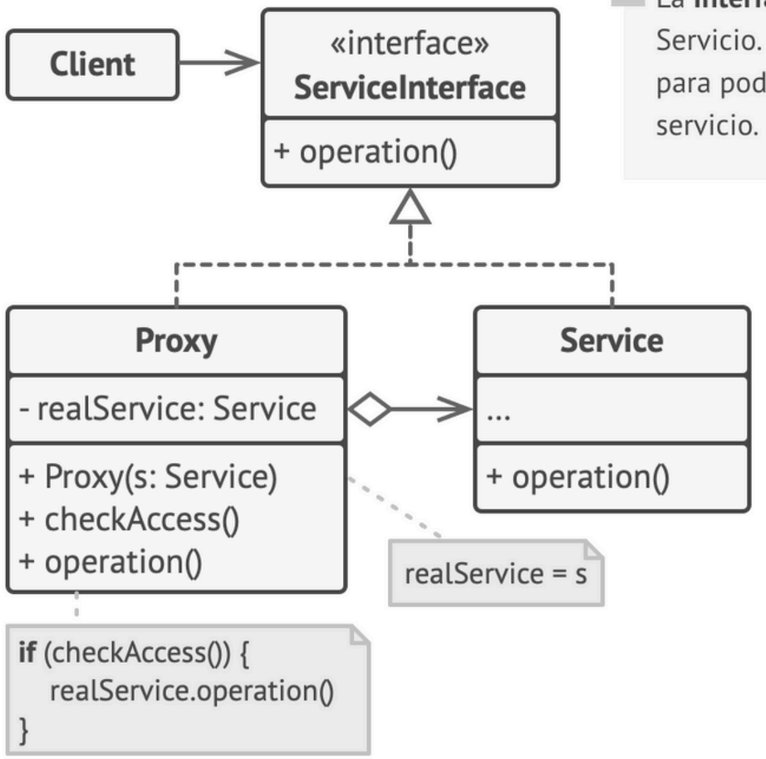
Proxy

⚖ Pros y contras

- ✓ Puedes controlar el objeto de servicio sin que los clientes lo sepan.
- ✓ Puedes gestionar el ciclo de vida del objeto de servicio cuando a los clientes no les importa.
- ✓ El proxy funciona incluso si el objeto de servicio no está listo o no está disponible.
- ✓ *Principio de abierto/cerrado*. Puedes introducir nuevos proxies sin cambiar el servicio o los clientes.
- ✗ El código puede complicarse ya que debes introducir gran cantidad de clases nuevas.
- ✗ La respuesta del servicio puede retrasarse.

Proxy

4 El **Cliente** debe funcionar con servicios y proxies a través de la misma interfaz. De este modo puedes pasar un proxy a cualquier código que espere un objeto de servicio.



1 La **Interfaz de Servicio** declara la interfaz del Servicio. El proxy debe seguir esta interfaz para poder camuflarse como objeto de servicio.

3 La clase **Proxy** tiene un campo de referencia que apunta a un objeto de servicio. Cuando el proxy finaliza su procesamiento (por ejemplo, inicialización diferida, registro, control de acceso, almacenamiento en caché, etc.), pasa la solicitud al objeto de servicio.

Normalmente los proxies gestionan el ciclo de vida completo de sus objetos de servicio.

2 **Servicio** es una clase que proporciona una lógica de negocio útil.

Proxy

Motivación - Problema

```ruby
class ConsoleClient
    def initialize(service)
        @service = service
    end
    def run
        rut = "22456456-2"
        clave = "secreta"
        if @service.login(rut,clave) then
            puts "autenticacion exitosa"
        else
            puts "autenticacion fallida"
        end
    end
end
```

```ruby
class LoginService
    def initialize()
        @users = { "22456456-2" => "secreta",
                   "11789789-3" => "password",
                   "33456123-4" => "clave" }
    end
    def login_user(rut, pass)
        return @users[rut.strip] == pass.strip
    end
end
```

```ruby
service = LoginService.new
client = ConsoleClient.new(service)
client.run
```

¿Cómo añadir un mensaje en consola (Log) cada vez que un cliente hace un login fallido? Requisito es no tocar ninguna de las dos clases.

Proxy

```ruby
class ConsoleClient
    def initialize(service)
        @service = service
    end
    def run
        rut = "22456456-2"
        clave = "secreta"
        if @service.login(rut,clave) then
            puts "autenticacion exitosa"
        else
            puts "autenticacion fallida"
        end
    end
end
```



```ruby
class LoginProxy
    def initialize(objetoOriginal)
        @objetoOriginal = objetoOriginal
    end
    def login(rut,pass)
        result = @objetoOriginal.login(rut,pass)
        if result == false then
            puts "failed login attempt #{rut}"
        end
        return result
    end
end
```



```ruby
class LoginService
    def initialize()
        @users = { "22456456-2" => "secreta2",
            "11789789-3" => "password",
            "33456123-4" => "clave" }
    end
    def login(rut, pass)
        return @users[rut.strip] == pass.strip
    end
end
```

Bibliografía

Enlaces útiles

- Adapter: https://refactoring.guru/es/design-patterns/adapter
- Proxy: https://refactoring.guru/es/design-patterns/proxy
- Decorator: https://refactoring.guru/es/design-patterns/decorator
- Composite: https://refactoring.guru/es/design-patterns/composite

Ejercicio con décimas

Parte I:
Considere el código de la pizza implementada con el patrón decorador. Agregue un método que devuelva un string con la composición de la pizza. Por ejemplo, el sigu

```
basic_pizza = Pineapple.new(Cheese.new(Ham.new(Ham.new(BasePizza.new))))
puts basic_pizza.toString()
basic_pizza = Pineapple.new((Ham.new(BasePizza.new)))
puts basic_pizza.toString()
```

Debería verse el siguiente código en consola:

```
ruby pizza-improved/main.rb
Pineaple(Cheese(Ham(Ham(BasePizza))))
Pineaple(Ham(BasePizza))
```

Ejercicio con décimas

Parte II:
Considere el código del file-system implementado con el patrón composite.
Agregue uno o los métodos necesarios para imprimir el path de todos los archivos en consola:
Debería verse:

```
/main/a
/main/b
/main/docs/c
/main/docs/d
/main/docs/x
```

```
4  main = SFolder.new('main')
5  main.add(SFile.new('a',4))
6  main.add(SFile.new('b',6))
7
8  doc = SFolder.new('docs')
9  main.add(doc)
10 doc.add(SFile.new('c',2))
11 doc.add(SFile.new('d',3))
12 doc.add(SFile.new('x',10))
13
14 main.pprint()
```

Ejercicio con décimas

Entrega Individual:

- Miércoles 15 de Mayo a las 22:00 hrs.
- Entrega vía Canvas en formato PDF.

¿Consultas?



Ingeniería de Software

16 -Patrones de Diseño II: Estructurales

IIC2143-3

Josefa España

jpespana@uc.cl