IIC 2143 - Ingeniería de Software

Patrones de diseño II

M. Trinidad Vargas
mtvargas1@uc.cl

Clase “auspiciada” por Refactoring Guru

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

Clasificación de patrones

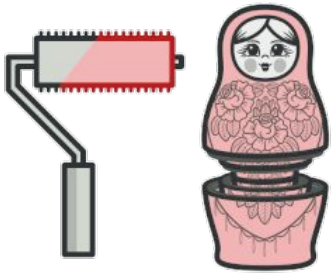
Según su propósito

- **Comportamiento:** brindan flexibilidad para agregar comportamiento.
- **Creacionales:** incrementan la flexibilidad y reusabilidad al momento de crear objetos complejos..
- **Estructurales:** mantiene la flexibilidad y reusabilidad al momento de componer estructuras de objetos.

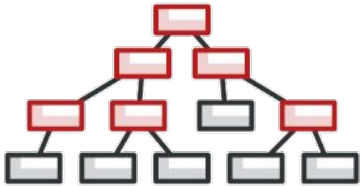
Patrones de diseño

Estructurales

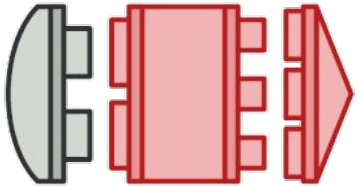
Decorator



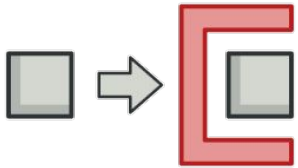
Composite



Adapter

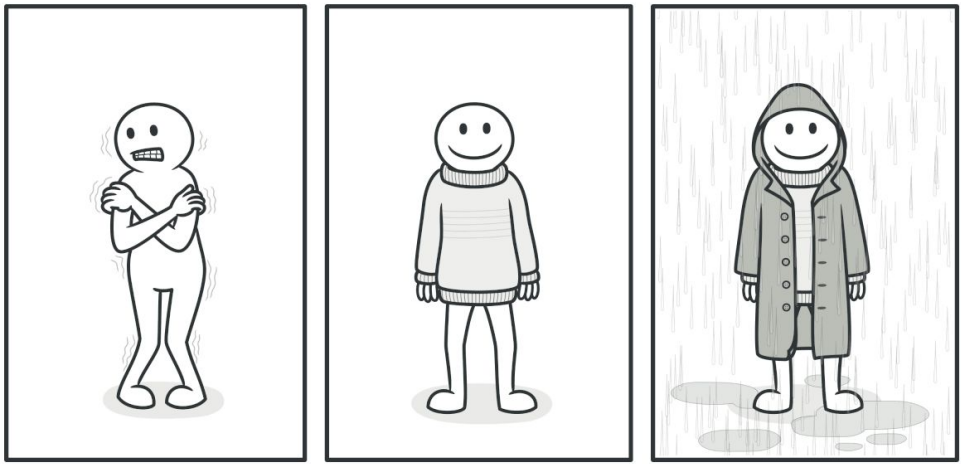


Proxy



Decorator

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.

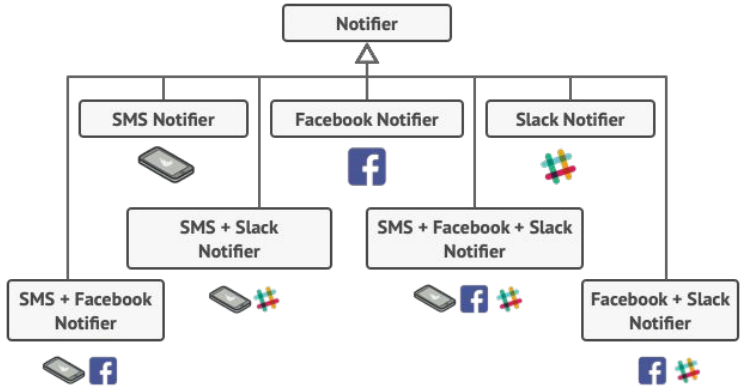


Obtienes un efecto combinado vistiendo varias prendas de ropa.

Decorator

Caso: Una aplicación de notificaciones solo permite notificar por un canal: SMS, Slack o Facebook. Luego te piden que una notificación pueda ser enviada por más de un canal a la vez

Problema: Tu primera solución es extendiendo la clase Notifier pero, rápidamente ves que es inviable crear cada combinación de notificadores.



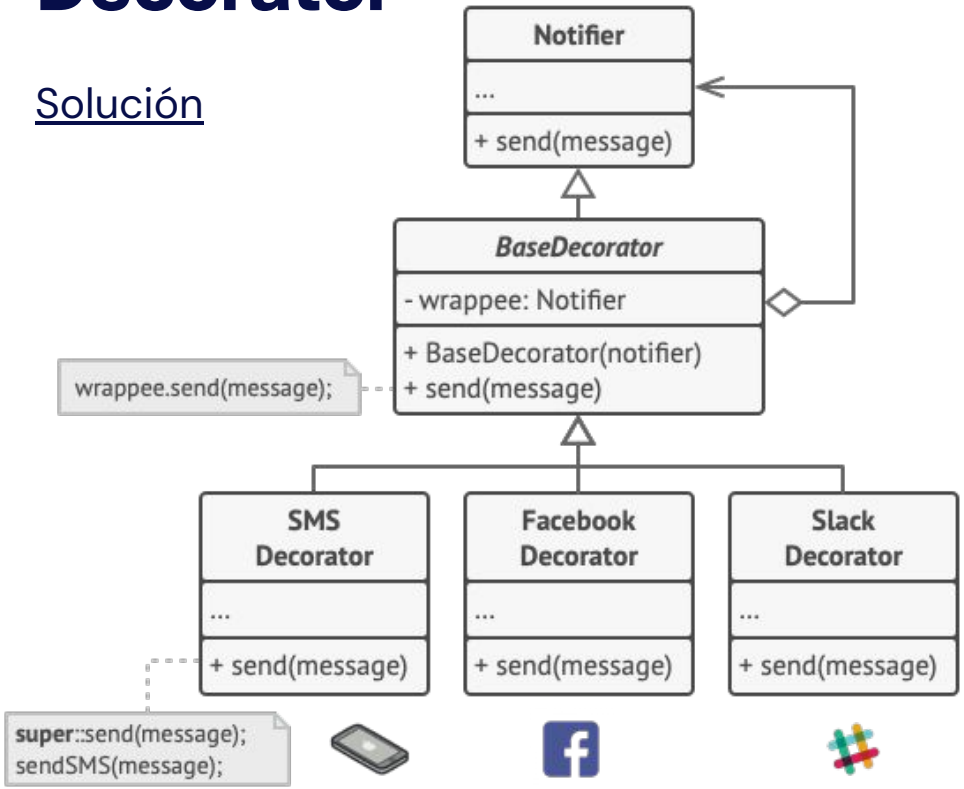
Decorator

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.

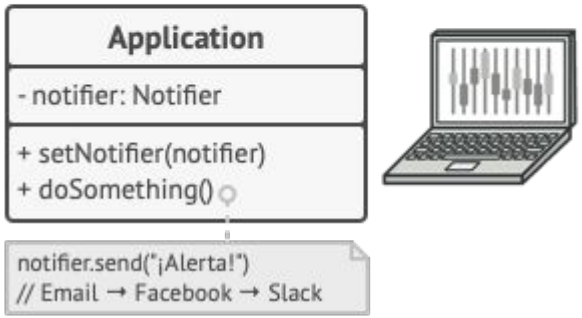
Decorator

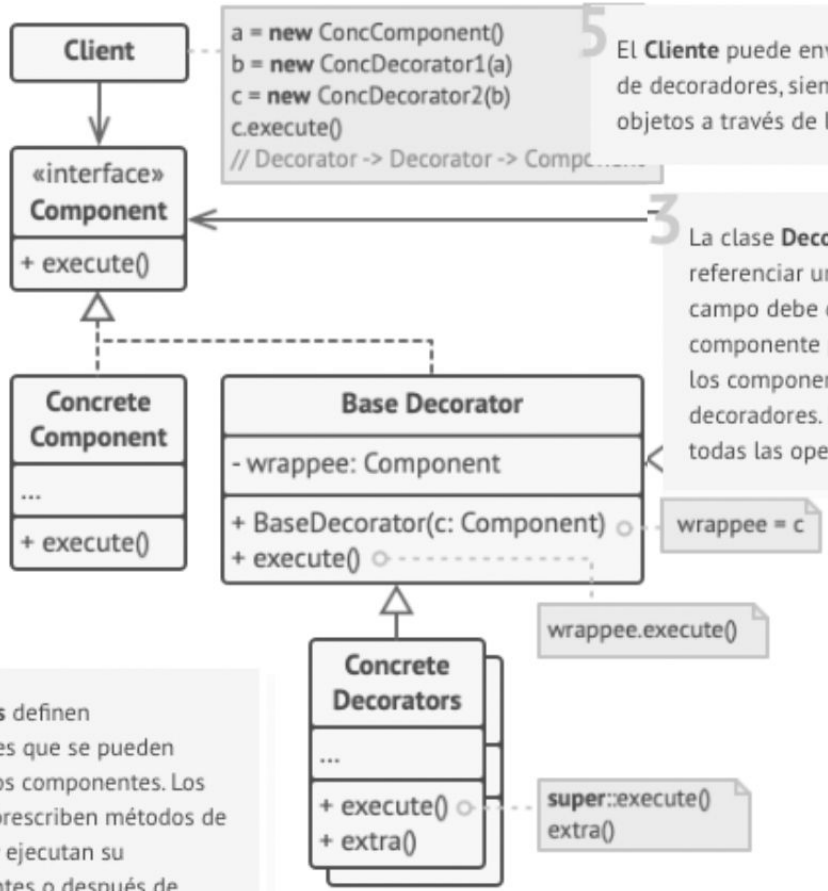
Solución




```
stack = new Notifier()
if (facebookEnabled)
    stack = new FacebookDecorator(stack)
if (slackEnabled)
    stack = new SlackDecorator(stack)

app.setNotifier(stack)
```





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.

Ejemplo

```ruby
class Pizza
 def print
   raise NotImplementedError
 end
 def cost
   raise NotImplementedError
 end
end

require_relative 'pizza'
class HamCheesePizza < Pizza
 def print
   puts 'normal bred'
   puts 'cheese'
   puts 'ham'
 end
 def cost
   20 + 5 + 5
 end
end
```

```ruby
require_relative 'ham_cheese_pizza'

class PineableHamCheesePizza <
HamCheesePizza
 def print
   puts 'normal bred'
   puts 'cheese'
   puts 'ham'
   puts 'pineapple'
 end
 def cost
   (20 + 5 + 5)* 2
 end
end
```

```ruby
require_relative
'ham_cheese_pizza'
require_relative
'pineapple_ham_cheese_pizza'

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

Ejemplo usando Decorator

```ruby
require_relative 'pizza'
class PizzaTopping < Pizza
 def initialize(decoratedPizza)
   @decorated = decoratedPizza
 end
 def cost
   @decorated.cost
 end
 def print
   @decorated.print
 end
end
```

```ruby
require_relative 'pizza'
class BasePizza < Pizza
 def print
   puts 'normal bred'
 end
 def cost
   20
 end
end
```

```ruby
require_relative 'topping'
class Ham < PizzaTopping
 def print
   @decorated.print
   puts 'ham'
 end
 def cost
   return @decorated.cost + 5
 end
end
```

```ruby
require_relative 'topping'
class Cheese < PizzaTopping
 def print
   @decorated.print
   puts 'cheese'
 end
 def cost
   return @decorated.cost + 5
 end
end
```

```ruby
require_relative 'topping'
class Pineapple < PizzaTopping
 def print
   @decorated.print
   puts 'pineapple'
 end
 def cost
   return @decorated.cost * 2
 end
end
```

```ruby
require_relative 'pizza'
require_relative 'cheese'
require_relative 'pineapple'
require relative 'ham'
require_relative 'base_pizza'
pizza = Cheese.new(
      Ham.new(BasePizza.new))
pizza.print
puts "#{pizza.cost}"
```

Decorator

¿Cuándo aplicar Decorator?

- Cuando necesites asignar funcionalidades adicionales a objetos durante el tiempo de ejecución sin descomponer el código que utiliza esos objetos.
- Cuando resulte extraño o no sea posible extender el comportamiento de un objeto utilizando la herencia.

Decorator

Pros

Puedes extender el comportamiento de un objeto sin crear una nueva subclase.

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.

Contras

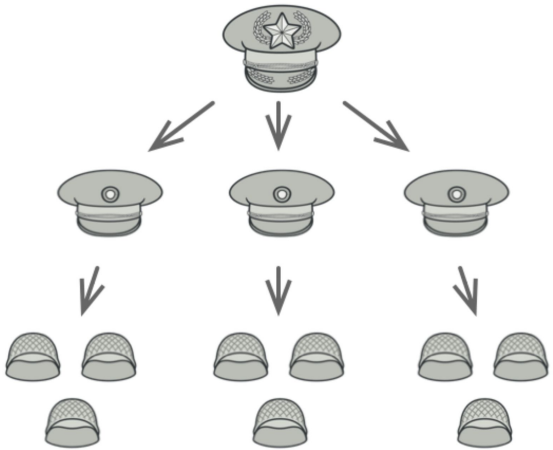
Resulta difícil eliminar un wrapper específico de la pila de wrappers.

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.

Composite

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.

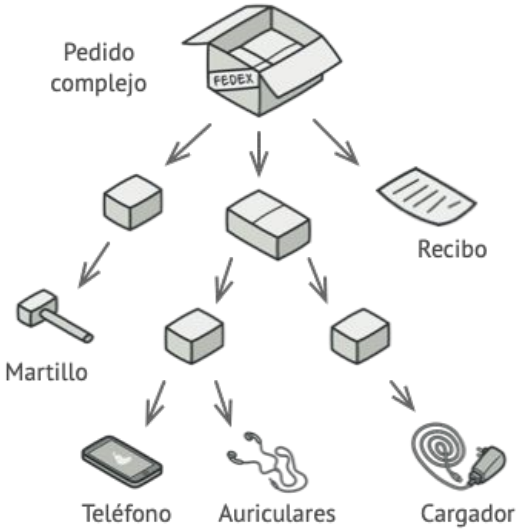


Un ejemplo de estructura militar.

Composite

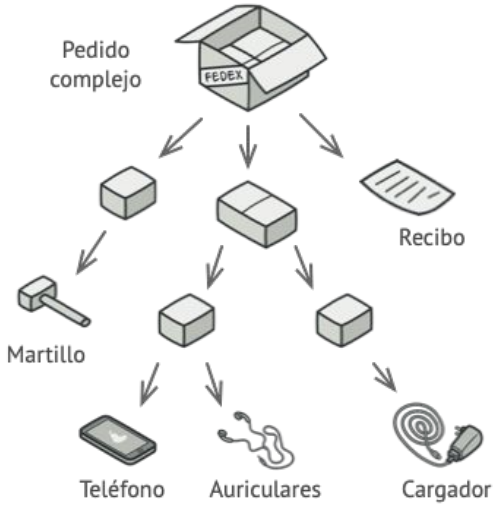
Caso: Tenemos dos tipos de objetos: productos y cajas. Una caja puede contener varios productos como también cierto número de cajas más pequeñas. Estas cajas también pueden contener productos o incluso cajas más pequeñas y así sucesivamente.

Considera que creas un sistema de pedidos con estas clases. Los pedidos pueden contener productos individuales o cajas llenas de productos y otras cajas ¿Cómo calculas el precio?



Composite

Problema: La solución directa es desenvolver todas las cajas y calcular el total, pero necesitas conocer de antemano las clases de productos y cajas e iterar.



Un pedido puede incluir varios productos empaquetados en cajas, que a su vez están empaquetados en cajas más grandes y así sucesivamente. La estructura se asemeja a un árbol boca abajo.

Composite

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

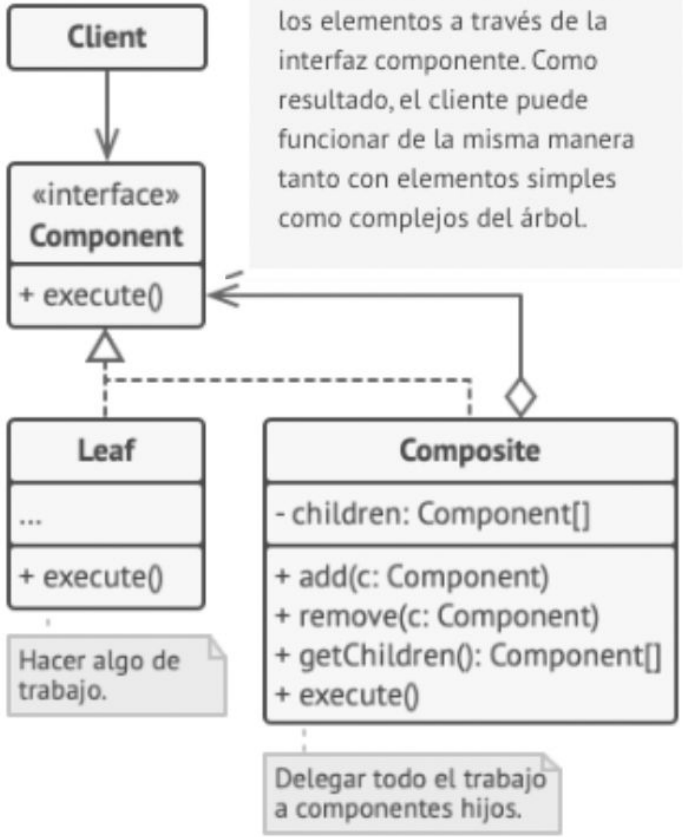
Composite

Solución

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.

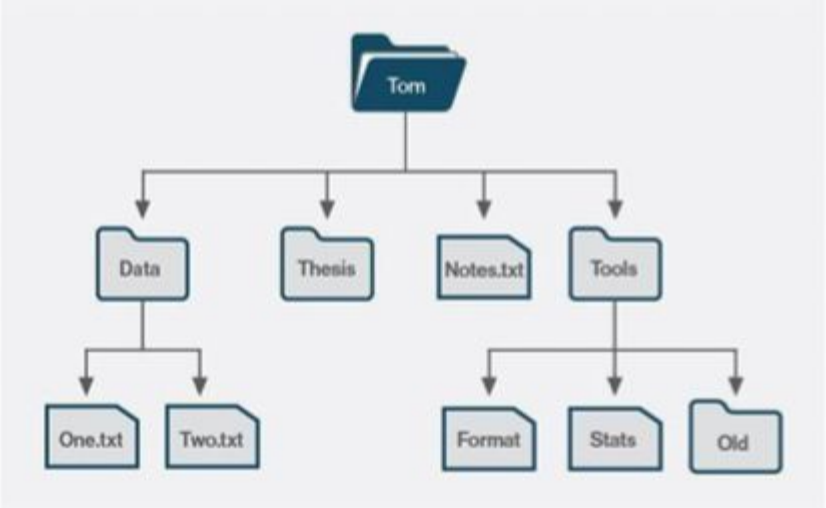
Al recibir una solicitud, un contenedor delega el trabajo a sus subelementos, procesa los resultados intermedios y devuelve el resultado final al cliente.

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.



Ejemplo

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

```ruby
require_relative 'file'
fileA = SFile.new('a',4)
puts "#{fileA.size}"
fileA.print
```

Ejemplo usando Composite

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

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
```

```ruby
require_relative 'element'
class SFolder < FileElement
 def initialize(name)
   @name = name
   @elements = []
 end
 def add(element)
   @elements.push(element)
 end
 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
```

```ruby
require_relative 'folder'
require_relative 'file'
main = SFolder.new('main')
main.add(SFile.new('a',4))
main.add(SFile.new('b',6))
doc = SFolder.new('docs')
doc.add(SFile.new('c',2))
doc.add(SFile.new('d',3))
main.add(doc)
doc.add(SFile.new('x',10))
puts "#{main.size}"
main.print
```

Composite

¿Cuándo aplicar Composite?

- Cuando tengas que implementar una estructura de objetos con forma de árbol.
- Cuando quieras que el código trate elementos simples y complejos de la misma forma.

Composite

Pros

Puedes trabajar con estructuras de árbol complejas con mayor comodidad: utiliza el polimorfismo y la recursión en tu favor.

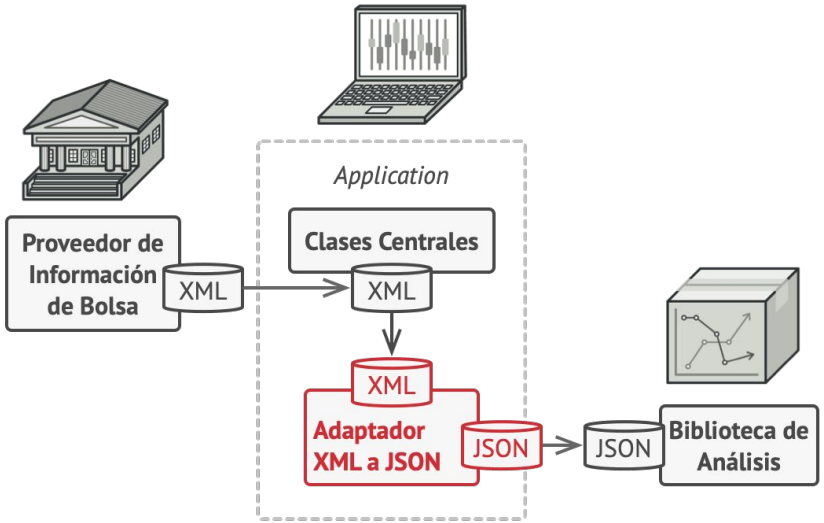
Principio de abierto/cerrado. Puedes introducir nuevos tipos de elemento en la aplicación sin descomponer el código existente, que ahora funciona con el árbol de objetos.

Contras

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.

Adapter

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

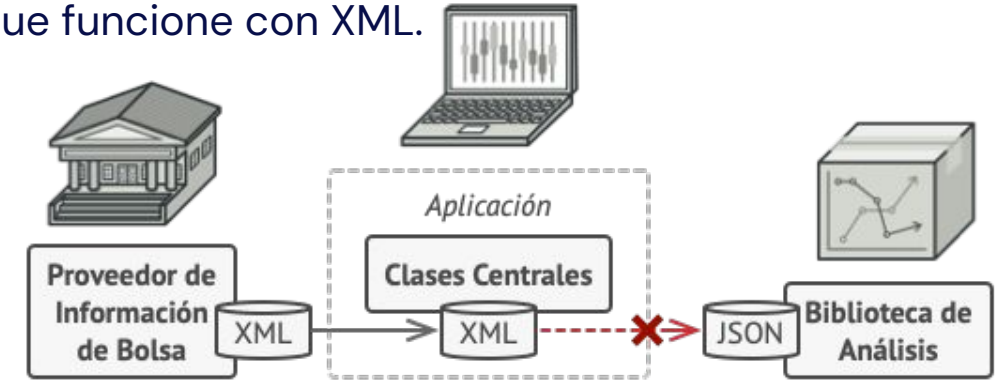


Adapter

Caso: Considera una aplicación de monitoreo del mercado de valores que descarga información de la bolsa de distintas fuentes en formato XML para presentar al usuario gráficos y diagramas.

Un día decides integrando una biblioteca de análisis de una tercera persona. Sin embargo, la biblioteca de análisis solo funciona con datos en formato JSON.

Podrías cambiar la biblioteca para que funcione con XML.
¿Cuáles son los riesgos de cambiar la biblioteca?



Problema:

Incompatibilidad de formatos

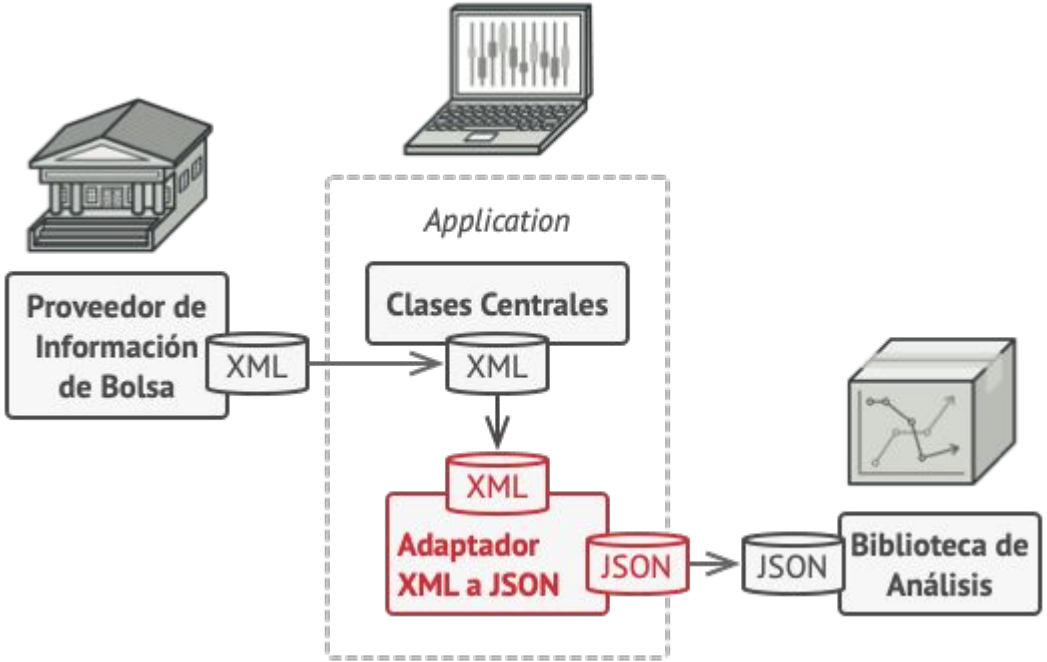
Adapter

Solución

- Crear un adaptador para convertir la interfaz de un objeto, de tal forma que otro objeto pueda entenderla.
    - .El adaptador obtiene una interfaz compatible con uno de los objetos existentes.
    - El objeto existente puede invocar con la interfaz los métodos del adaptador.
    - Al recibir una llamada, el adaptador pasa la solicitud al segundo objeto, pero en el formato que este objeto espera.

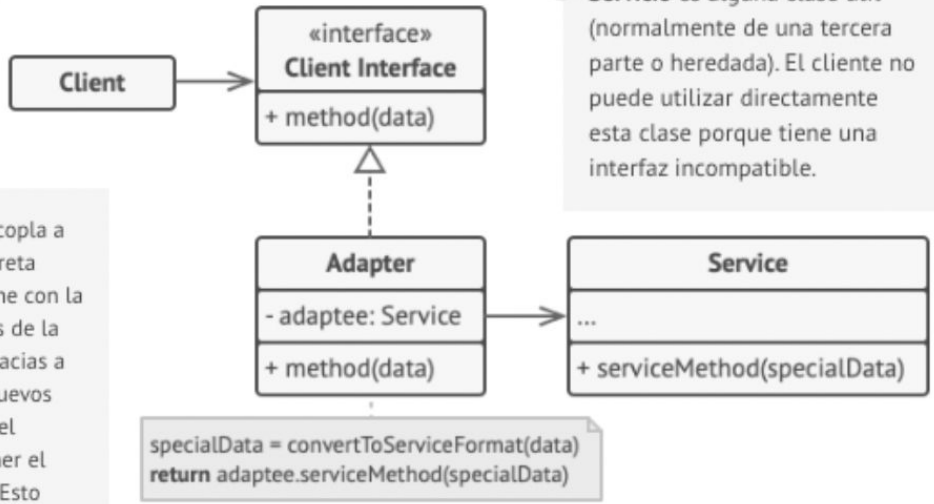
Adapter

Solución



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

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



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.

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.

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.

Ejemplo

```ruby
class ConsoleClient
 def initialize(service)
   @service = service
 end
 def run
   puts "Ingrese RUT(con puntos y guion):"
   rut = gets
   puts "Ingrese ClaveUnica:"
   clave = gets
   if @service.login(rut,clave) then
     puts "autenticacion exitosa"
   else
     puts "autenticacion fallida"
   end
 end
end
```

```ruby
class LoginService
 def initialize()
   @users = { "10001-2" => "123",
              "10002-3" => "333",
              "12117-4" => "444" }
 end
 def login_user(rut, pass)
   return @users[rut.strip] == pass.strip
 end
end


require_relative 'login_service'
require_relative 'client'
service = LoginService.new
client = ConsoleClient.new(service)
client.run
```

Ejemplo usando Adapter

```ruby
class ConsoleClient
 def initialize(service)
   @service = service
 end
  def run
   puts "Ingrese RUT(con puntos y guion):"
   rut = gets
   puts "Ingrese ClaveUnica:"
   clave = gets
   if @service.login(rut,clave) then
     puts "autenticacion exitosa"
   else
     puts "autenticacion fallida"
   end
 end
end
```

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

class LoginService
 def initialize()
   @users = { "10001-2" => "123",
              "10002-3" => "333",
              "12117-4" => "444" }
 end
 def login_user(rut, pass)
   return @users[rut.strip] == pass.strip
 end
end
```

```ruby
require_relative 'login_service'
require_relative 'adapter'
require_relative 'client'
 service = LoginService.new
adaptador =
LoginServiceAdapter.new(service)
client = ConsoleClient.new(adaptad
client.run
```

Adapter

¿Cuándo aplicar Adapter?

- Cuando quieras usar una clase existente, pero cuya interfaz no sea compatible con el resto del código.
- Cuando quieras reutilizar varias subclases existentes que carezcan de alguna funcionalidad común que no pueda añadirse a la superclase.

Adapter

Pros

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

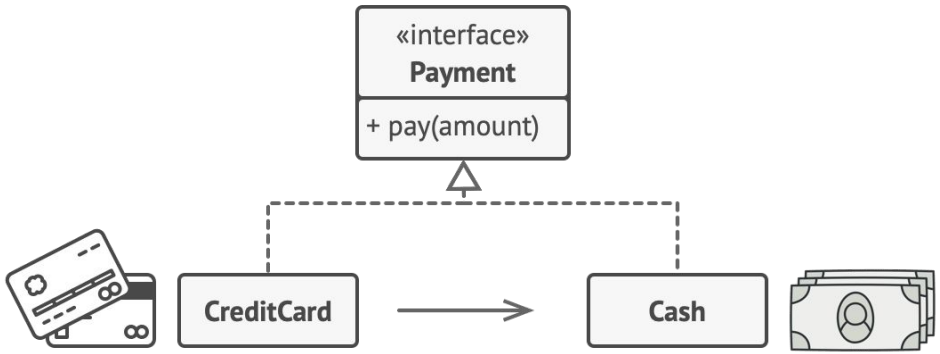
Principio de abierto/cerrado. Puedes introducir nuevos tipos de adaptadores al programa sin descomponer el código cliente existente, siempre y cuando trabajen con los adaptadores a través de la interfaz con el cliente.

Contras

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.

Proxy

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.

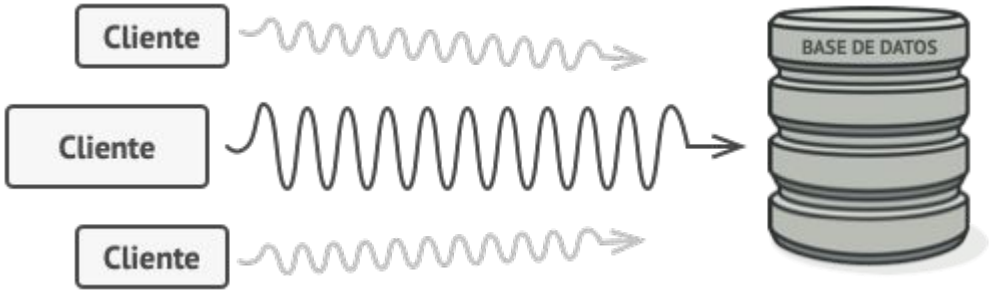


Las tarjetas de crédito pueden utilizarse para realizar pagos tanto como el efectivo.

Proxy

Caso: Tienes un objeto enorme que consume una gran cantidad de recursos del sistema. Lo necesitas a veces, pero no siempre; como una base de datos

Problema: Tu primera opción es crear el objeto solo cuando sea necesario pero esto genera mucho código duplicado.



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

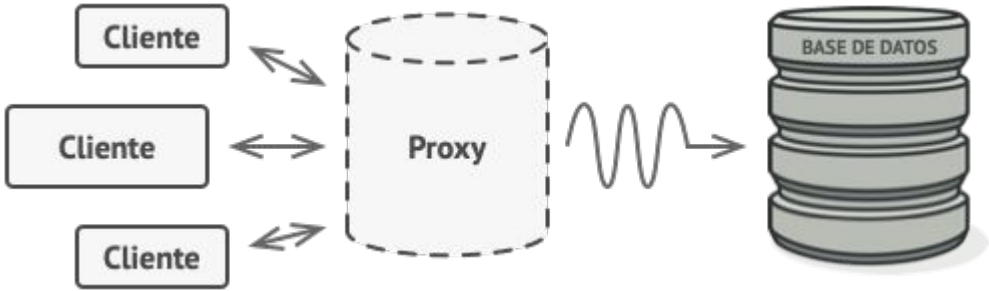
Proxy

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

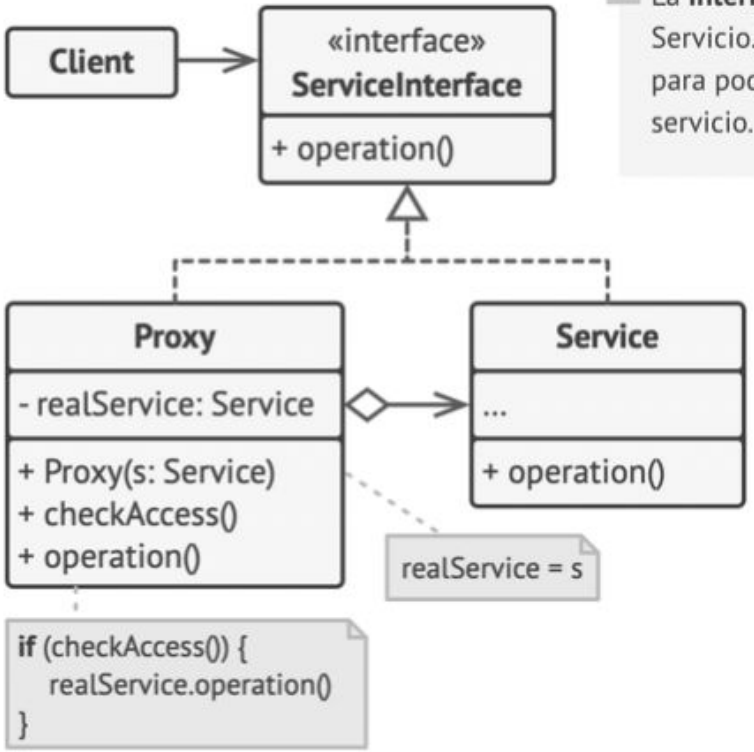
Proxy

Solución



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.

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.

Ejemplo

Desafío: ¿Cómo añadir un mensaje en consola (Log) cada vez que un cliente hace un login fallido?

Restricción: No puedes modificar ninguna de las dos clases

```ruby
class ConsoleClient
 def initialize(service)
   @service = service
 end
 def run
   rut = "19892631-9"
   clave = "secreta"
   if @service.login(rut,clave) then
     puts "autenticación exitosa"
   else
     puts "autenticación fallida"
   end
 end
end
```

```ruby
class LoginService
 def initialize()
   @users = { "19892631-9" => "123",
              "20112651-2" => "333",
              "22892119-3" => "444" }
 end
 def login_user(rut, pass)
   return @users[rut.strip] == pass.strip
 end
end


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

Ejemplo usando Proxy

```ruby
class ConsoleClient
 def initialize(service)
   @service = service
 end
 def run
   rut = "19892631-9" clave = "secreta"
   if @service.login(rut,clave) then
     puts "autenticación exitosa"
   else
     puts "autenticación fallida"
   end
 end
end

class LoginService
 def initialize()
   @users = { "19892631-9" => "123", …}
 end
 def login_user(rut, pass)
   return @users[rut.strip] == pass.strip
 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
```

Proxy

Pros

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.

Contras

El código puede complicarse ya que debes introducir gran cantidad de clases nuevas.

La respuesta del servicio puede retrasarse.

Ejercicios adicionales

1. ¿Cómo se diferencian un Proxy y un Decorator, dado que ambos implementan la misma interfaz?
2. ¿Qué tan acoplado queda el sistema al usar Adapters? ¿Cómo se puede minimizar ese acoplamiento?
3. ¿Puedes identificar un caso en el que usar Adapter sería una mala elección?
4. Implementa en Ruby el ejemplo de Composite.
5. Crea los diagramas de clase para los ejemplos en Ruby

Material Adicional

Existen muchos patrones estructurales

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