Patrones de Diseño: Estructurales

Juan Pablo Sandoval

Patrones de Comportamiento que veremos

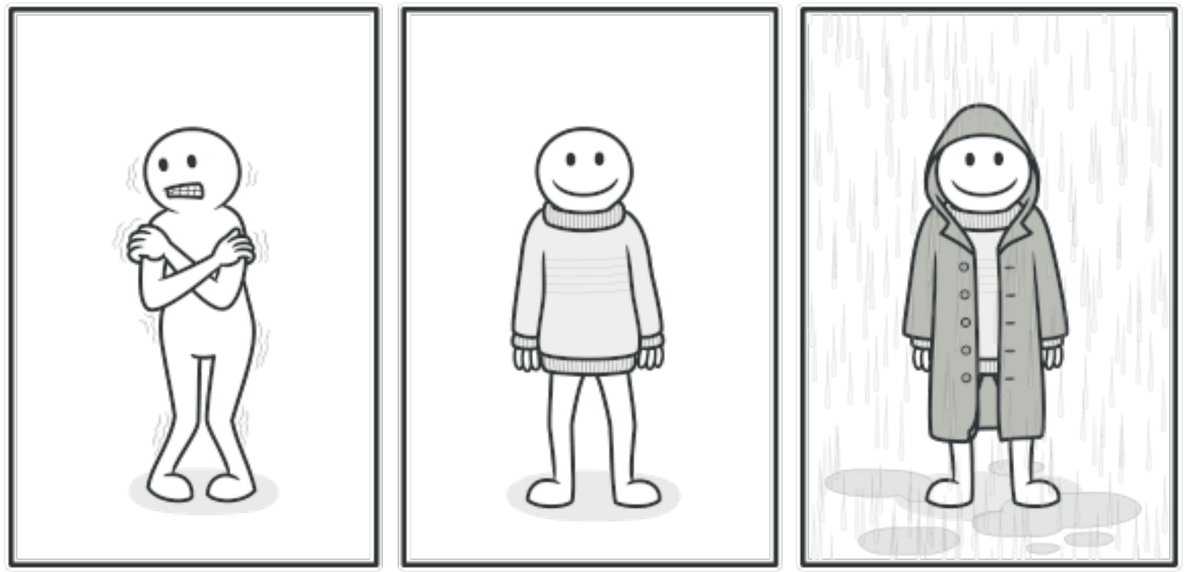
- *Decorator*
- *Composite*

Decorator

Decorator es un patrón de diseño structural que permite agregar (adjuntar) comportamiento nuevo a objetos. Un objeto decorador encapsula el objeto decorado.

La idea es que 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 adicionalmente realiza acciones adicionales.

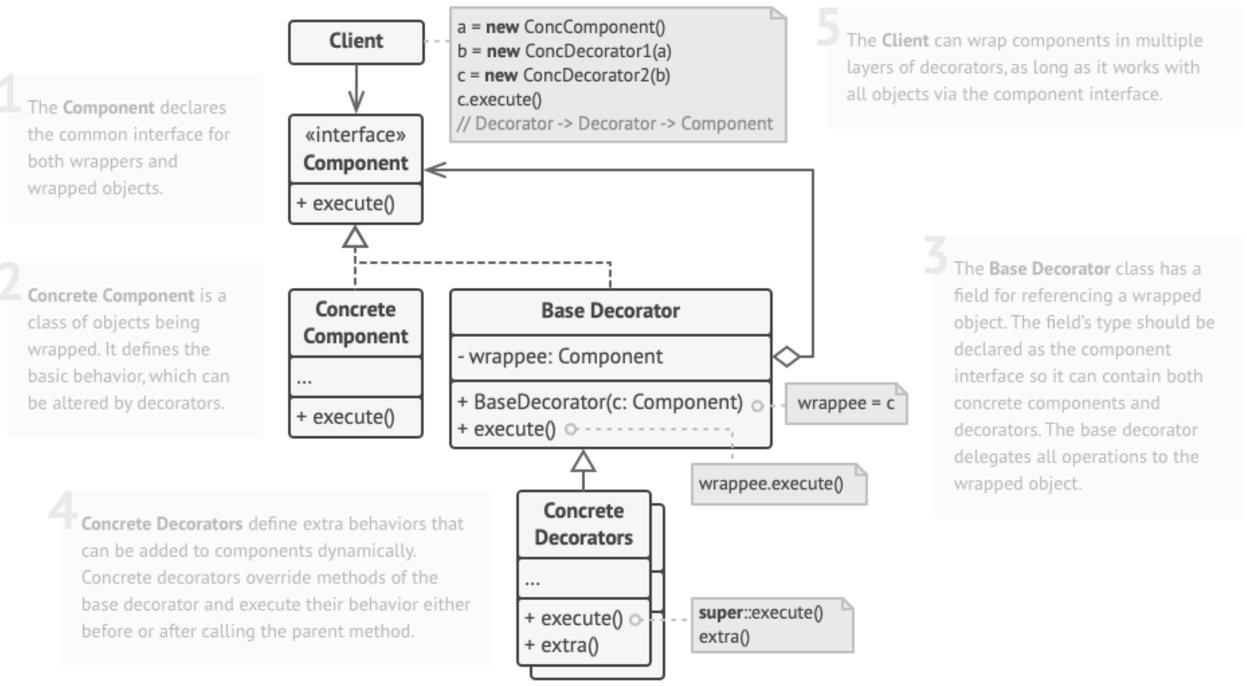
La figura de alado muestra una analogía de la vida real, donde la persona es el objeto decorado y existen dos decoradores el suéter y la chaqueta.



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

Decorator

La figura de la derecha muestra la meta estructura de un modelo objeto que implementa el patron decorador.



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

```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
```

El costo de la pizza es el costo de la masa (25), mas el precio del queso (5), mas el queso del jamon(5).

```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
```

El costo de la pizza es el costo de la masa (25), mas el precio del queso (5), mas el queso del jamon(5).

Lo anterior multiplicado por 2.

Pizza - Example

El problema con la implementación actual, es que la forma en que se agregan los toppings es estática.

Por ejemplo, si quieres hacer una pizza con doble queso, tendría que crear otra clase. Si quiero hacer una pizza con doble jamón y doble queso, tendría que crear otra clase. Por cada nueva combinación de pizza (incluso utilizando los ingredientes existentes) tendría que crear una clase.

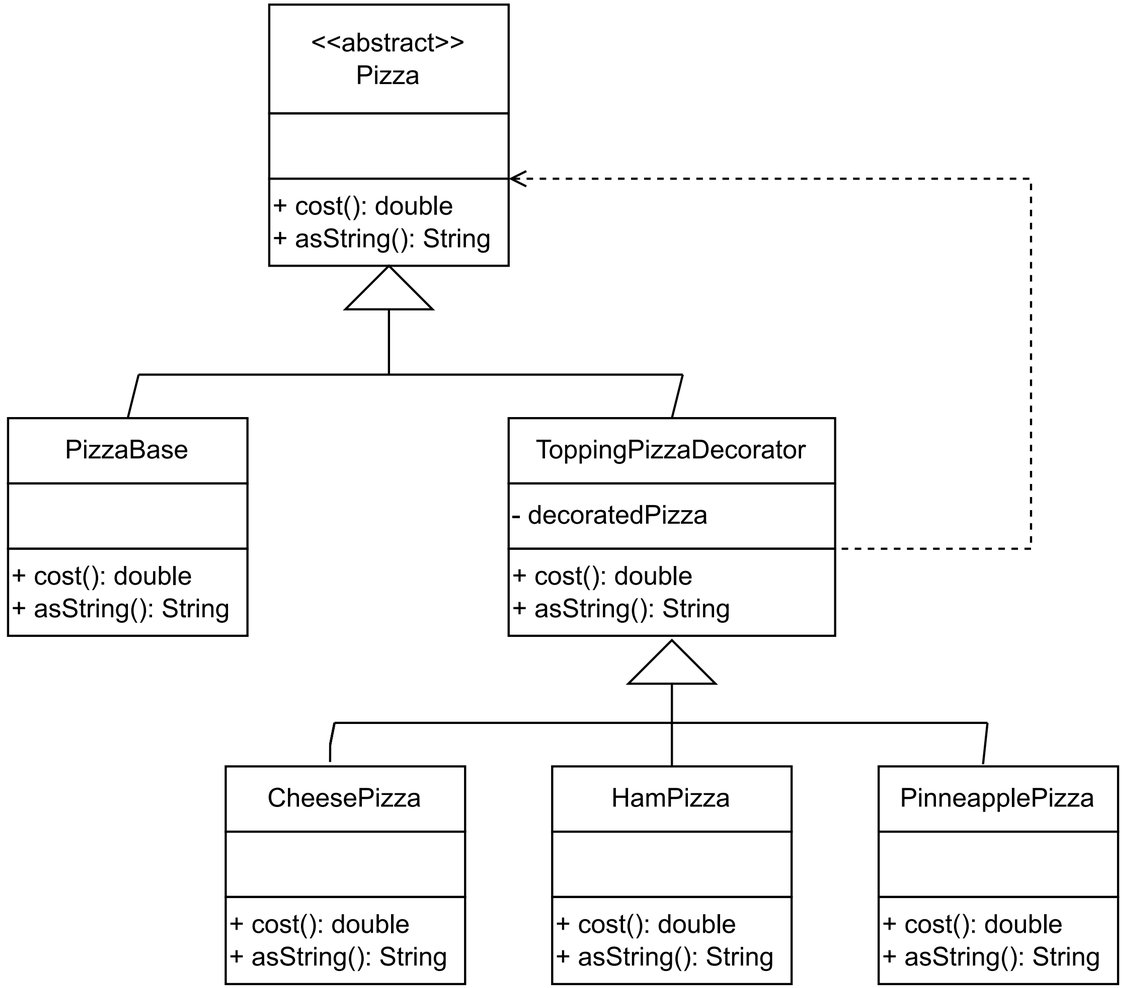
Lo anterior provoca, código duplicado entre los distintos tipos de pizza. Ademas, si tengo por ejemplo 4 diferentes tipos de ingredientes, tendría que tiene 24 clases (una por cada combinación).

```ruby
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}"
```

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

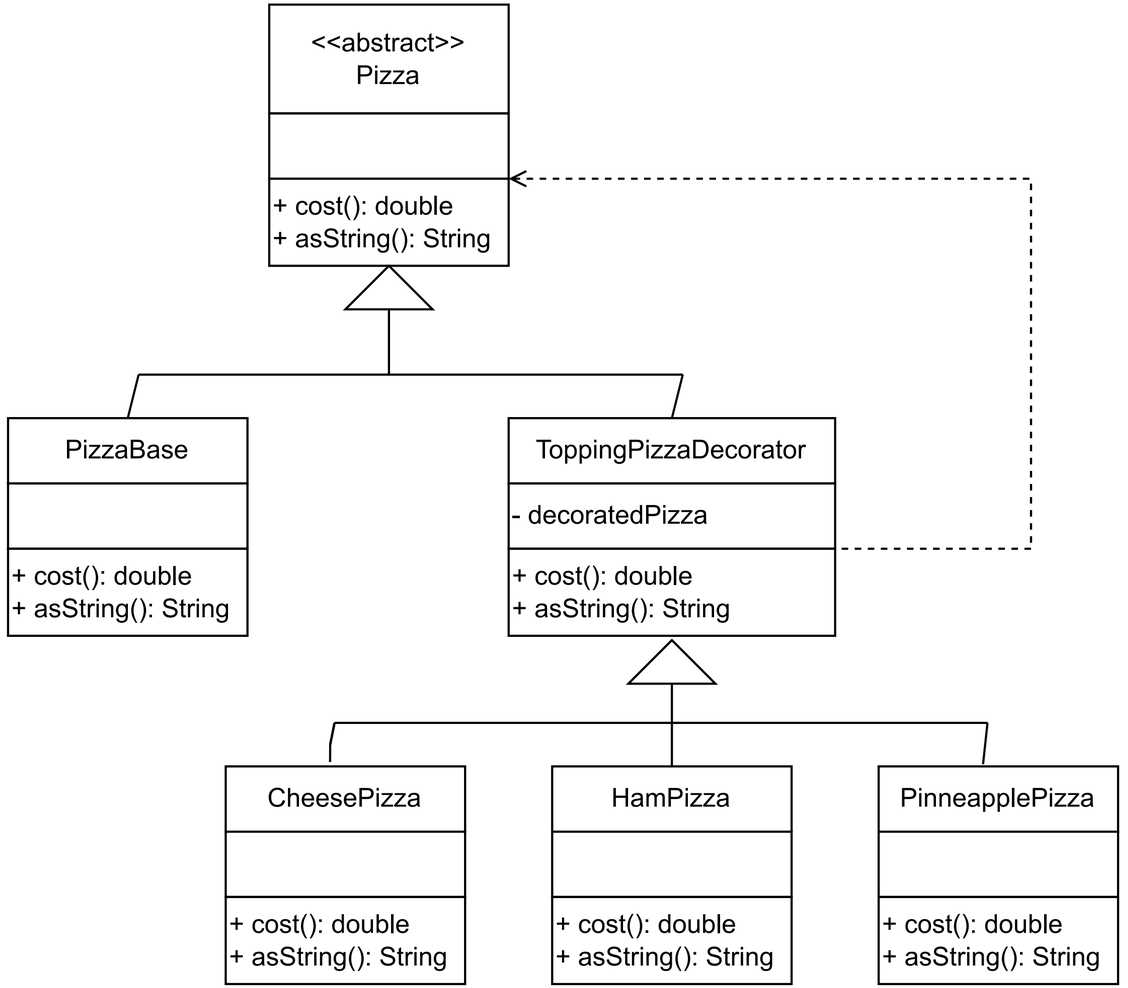
Pizza - Example + Decorator

```ruby
class Pizza
  def print
    raise NotImplementedError
  end

  def cost
    raise NotImplementedError
  end
end
```



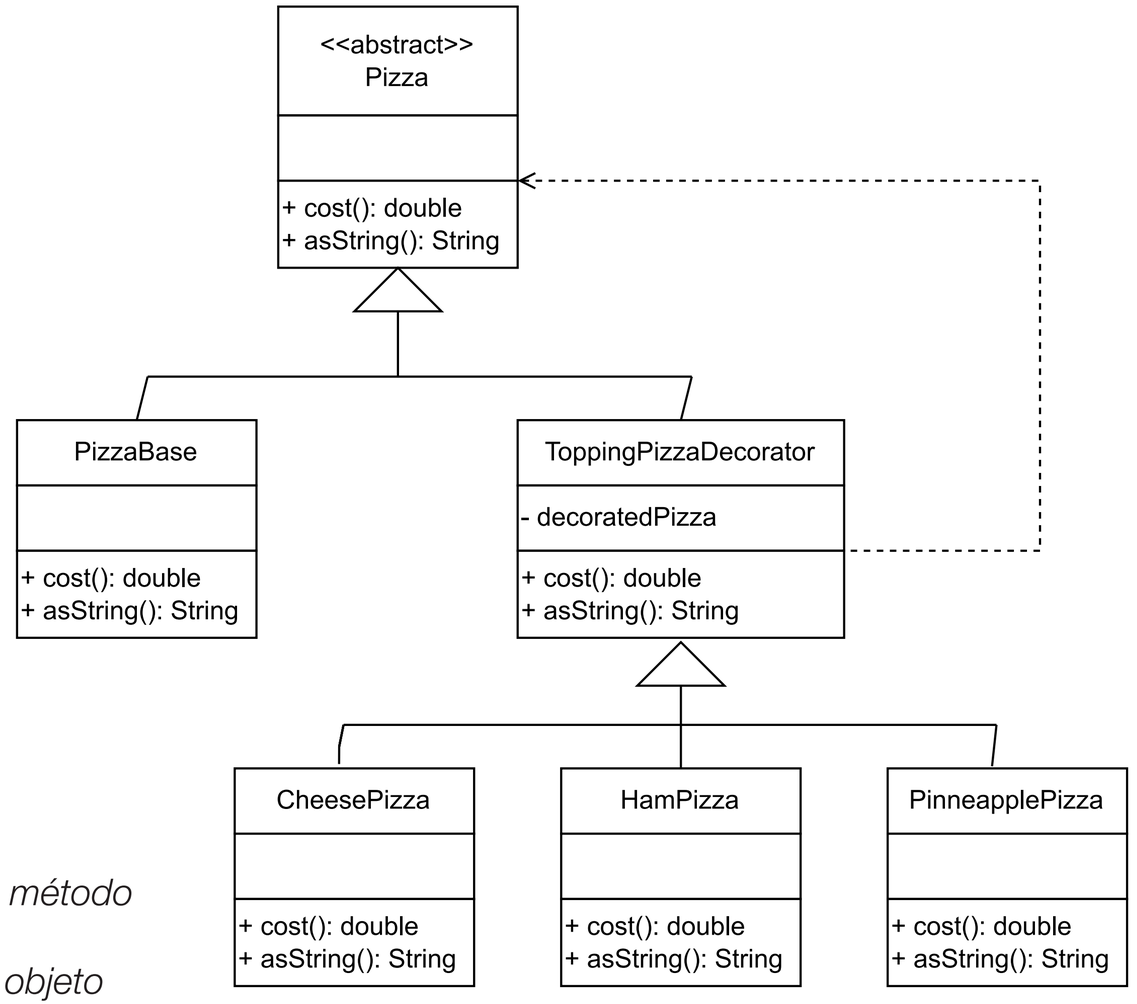
Pizza - Example + Decorator

```ruby
require_relative 'pizza'

class BasePizza < Pizza

  def print
    puts 'normal bred'
  end

  def cost
    20
  end
end
```



Pizza - Example + Decorator

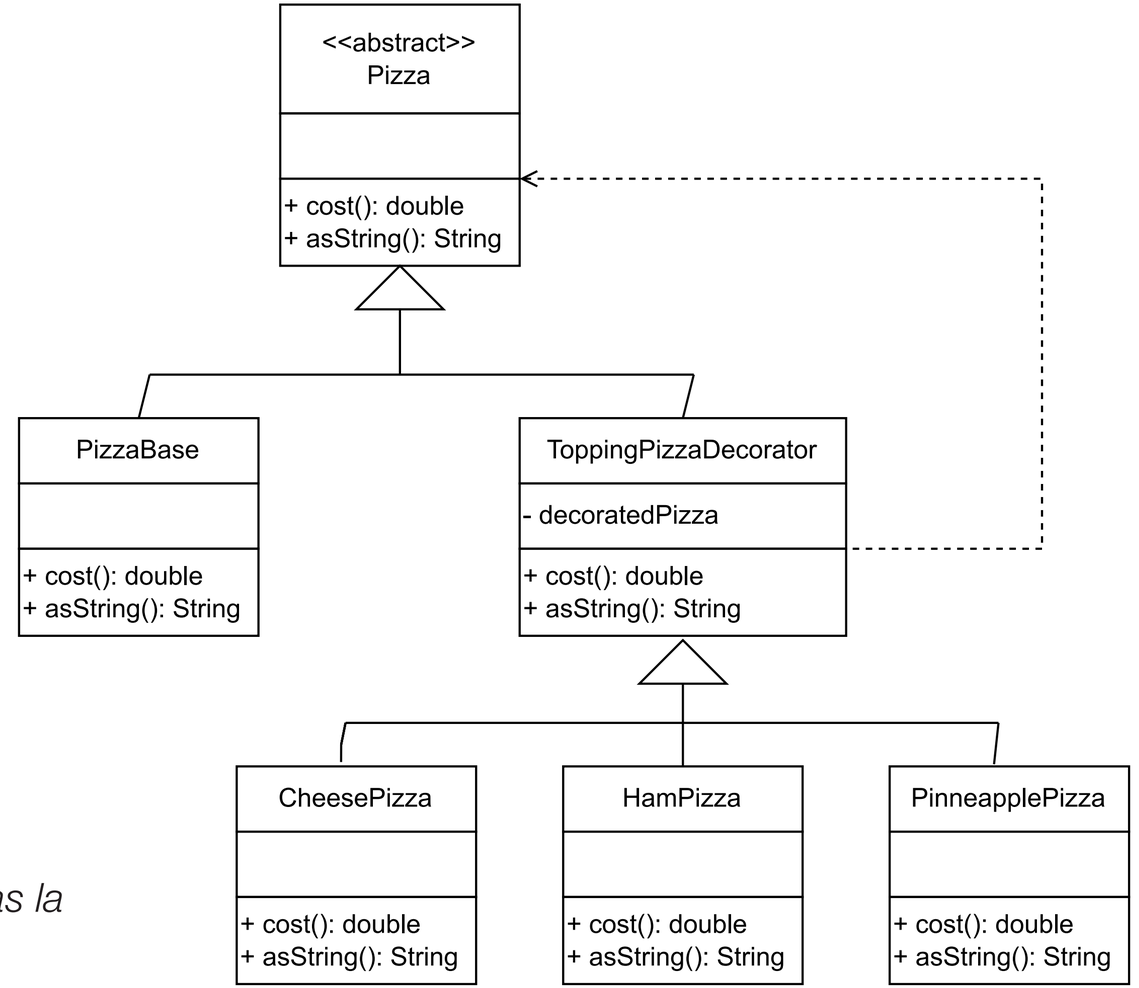
```ruby
require_relative 'pizza'

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



La clase Decorator, recibe el objeto a decorar en el constructor, para cada método de la pizza el decorador ejecuta el mismo método del objeto decorado.
Esta clase en realidad no agrega funcionalidad, solo hace lo mismo que el objeto decorado.

Pizza - Example + Decorator

```ruby
require_relative 'topping'

class Cheese < ToppingPizzaDecorator

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

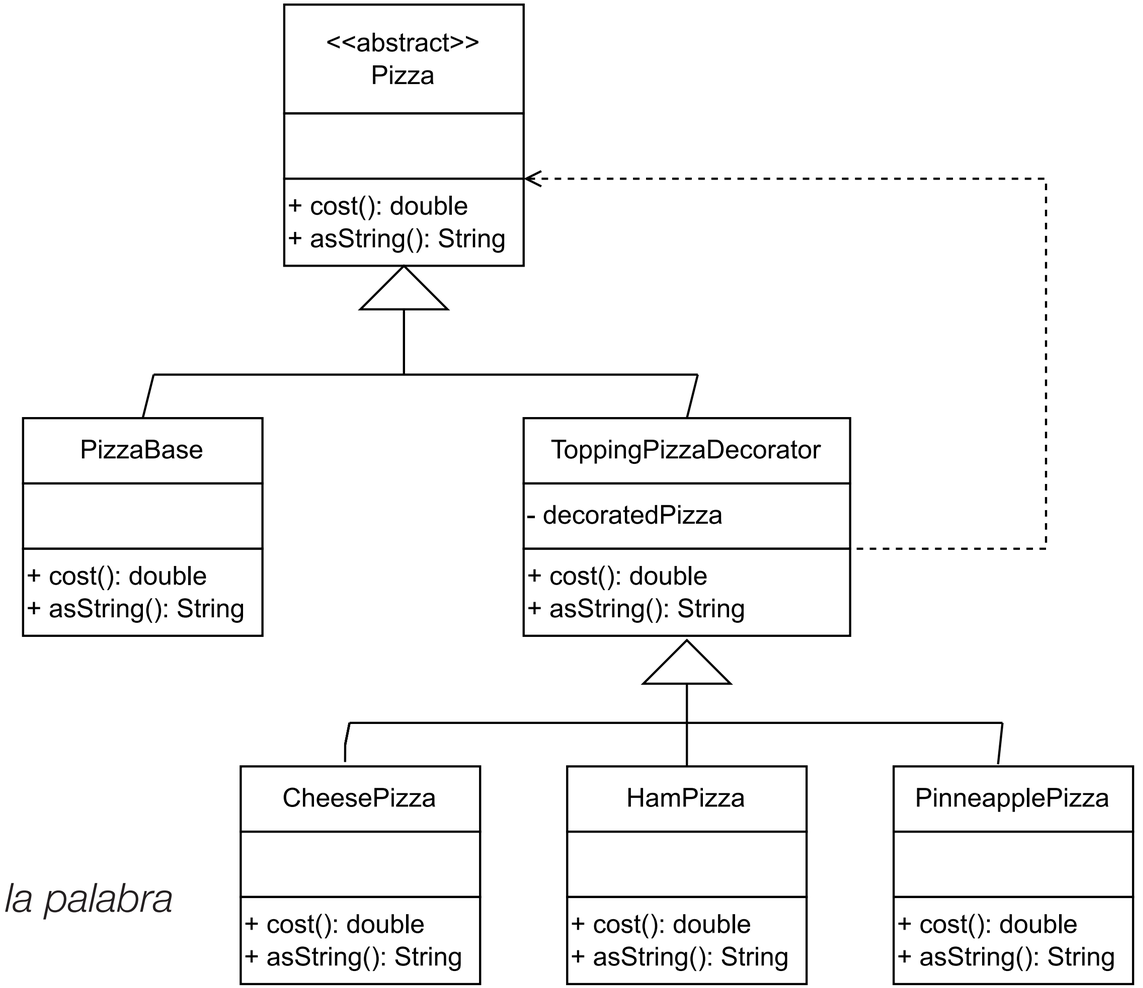


El Decorador de queso, imprime lo mismo que la pizza decorada mas la palabra cheese (nuevo comportamiento).

El costo es lo mismo que la pizza decorada mas cinco.

Pizza - Example + Decorator

```ruby
require_relative 'topping'

class Pineapple < ToppingPizzaDecorator

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

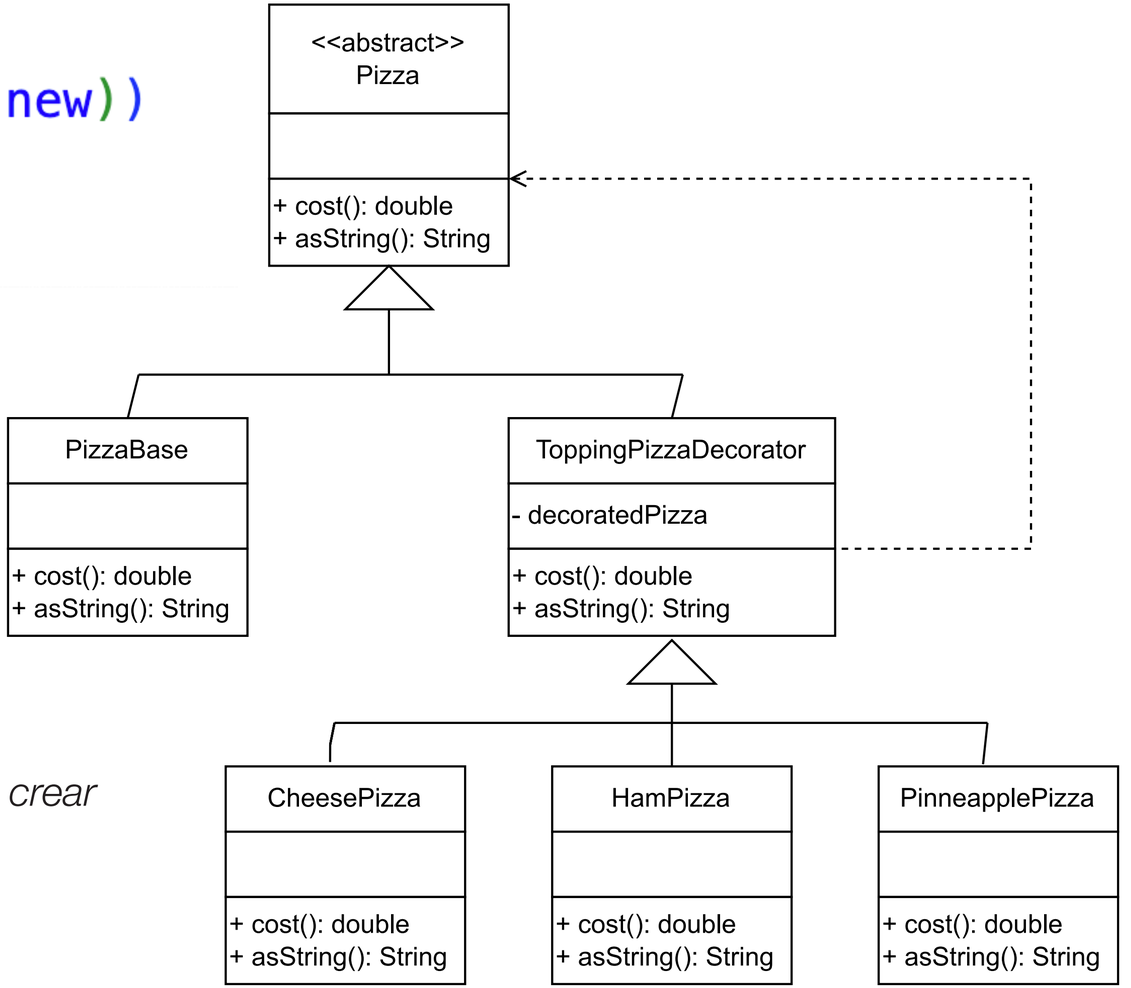


El Decorador de piña, imprime lo mismo que la pizza decorada mas la palabra piña (nuevo comportamiento).

El costo es lo que cuesta la pizza decorada por dos.

Pizza - Example + Decorator

```
basic_pizza = Cheese.new(Ham.new(BasePizza.new))
basic_pizza.print
puts "#{basic_pizza.cost}"
```



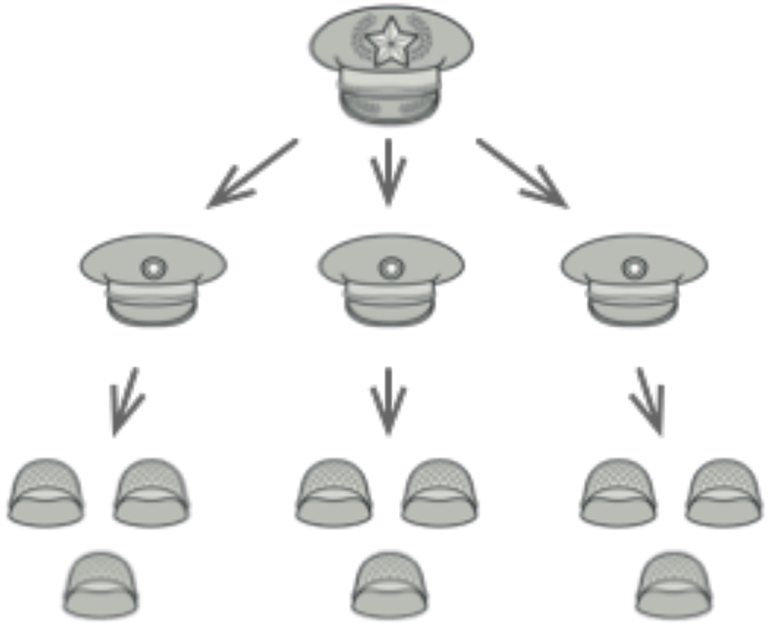
Con lo anterior ya es posible crear diferentes tipos de pizza sin tener que crear nuevas clases. Se posible crear todas las combinaciones.

Composite

El patron composite permite componer objetos que siguen una estructura de árbol. Cada objeto en la estructura puede tratado como un sub-árbol o un objeto individual.

Por ejemplo, considere la linea de mando en el ejercicio. Cada rango tiene a su cargo elementos de menor rango que su vez tiene a cargo otros elementos.

Lo unites que no tienen nadie bajo su mando son los soldados rasos. Siendo estos los únicos elementos no compuestos.

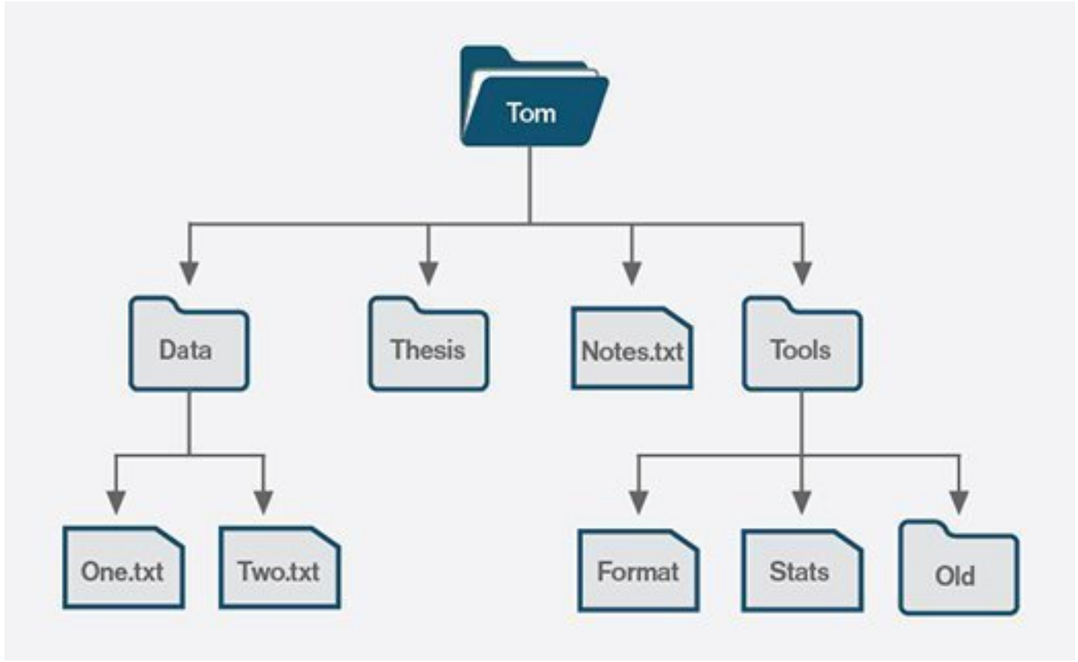


https://refactoring.guru/design-patterns/composite

Ejemplo - FileSystem

En este 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.

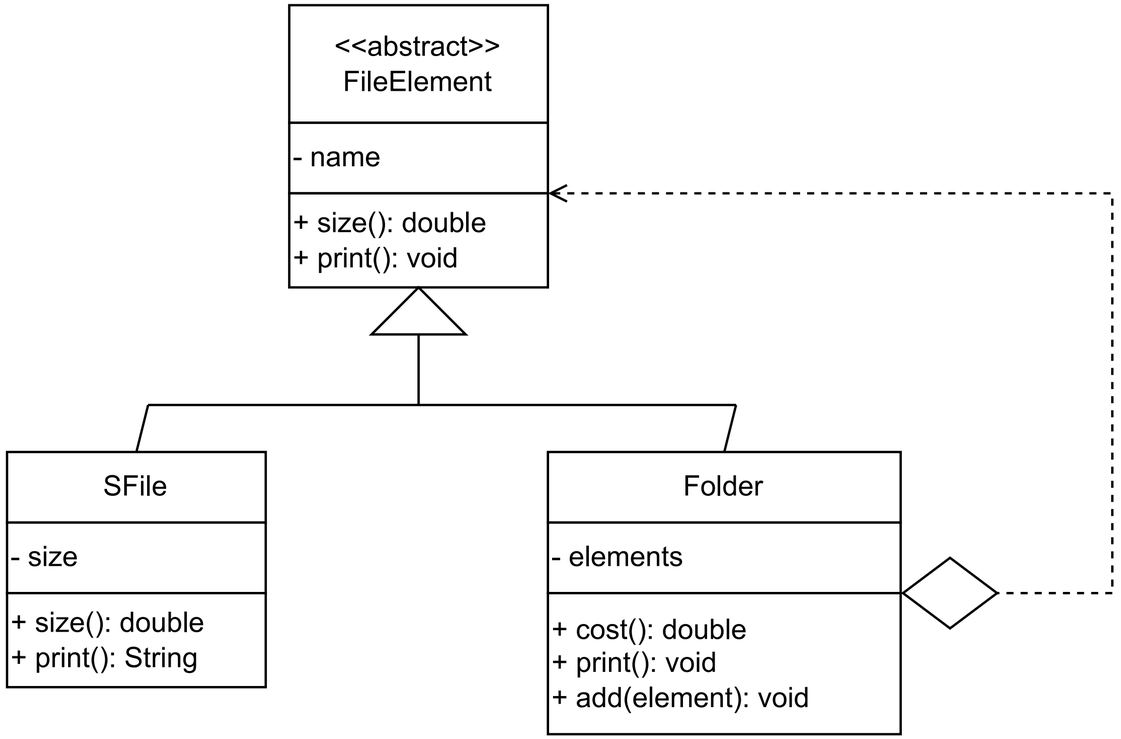
Por lo anterior se puede ver que la estructura del file sistema esta compuesta.



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

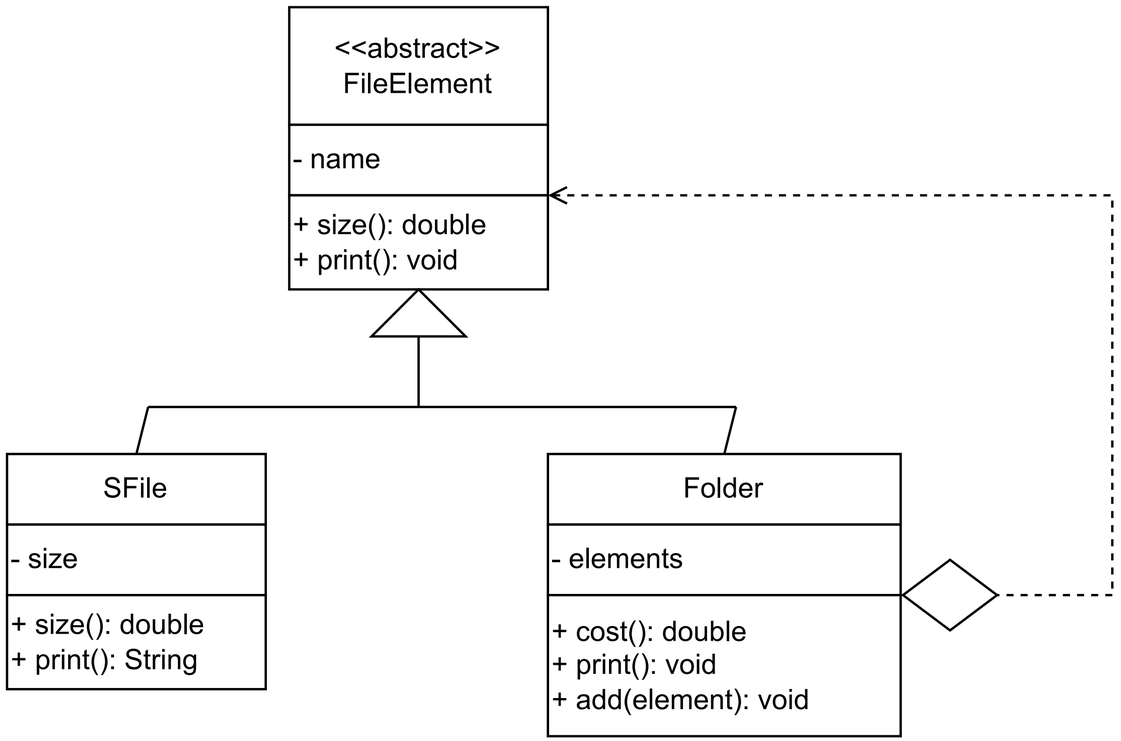
FileSystem - Example + Composite

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



FileSystem - Example + Composite

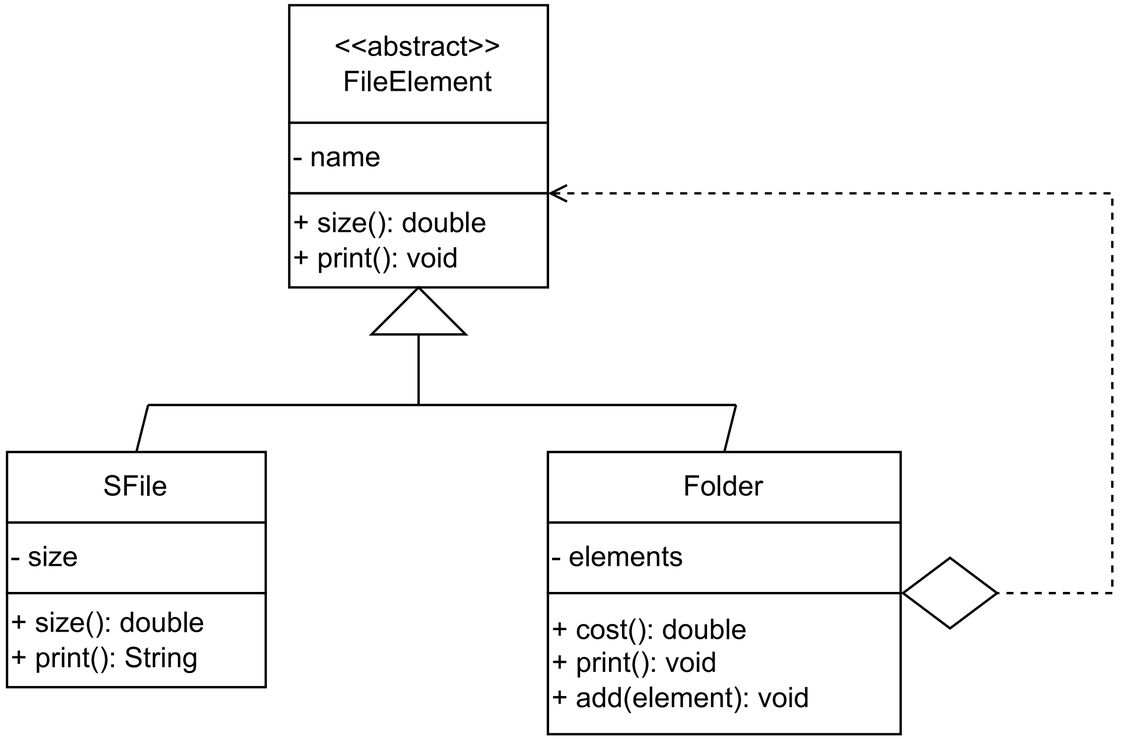
```ruby
require_relative 'element'

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



Los métodos del objeto que no esta compuesto resuelven la tarea (normalmente) de forma independiente.

FileSystem - Example + Composite

```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
```



Los métodos del objeto compuesto, normalmente resuelven las tareas llamando al mismo método de sus sub-elementos.

Ejercicio 1

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 siguiente código:

```
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 mostrar el siguiente texto en consola:

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

Ejercicio 2

Considere el código del file-system implementado con el patrón composite. Agregue un o los métodos necesarios para imprimir el path de todos los archivos en consola. Por ejemplo, el código de la derecha,

debería imprimir lo siguiente:

```
/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()
```