# Decorator Design Pattern

Jonathan Petit

27 octobre 2017

ECAM

## The Decorator design pattern

## Context and application

- Attach features dynamically
  *Add new functionality to an existing object without altering its structure.*
- Single responsibility principle.
  *Divide functionality between classes with unique feature.*
- Embellishment of a core object by recursively wrapping
  *Basic object is envelloped with its different characteristics.*

## Bad structure

- A base class "Windows"

  *A new class inherited when a new windows have others options.*
- A lot of repetition

  *The classes have a lot of resemblance.*

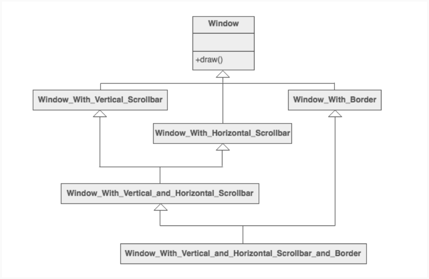

Window
+draw()
Window_With_Vertical_Scrollbar
Window_With_Border
Window_With_Horizontal_Scrollbar
Window_With_Vertical_and_Horizontal_Scrollbar
Window_With_Vertical_and_Horizontal_Scrollbar_and_Border

**Figure 1** – Bad structure

## The solution structure

- A base class (interface)
- Few concretes classes of the base class
- A decorator class
- Few options

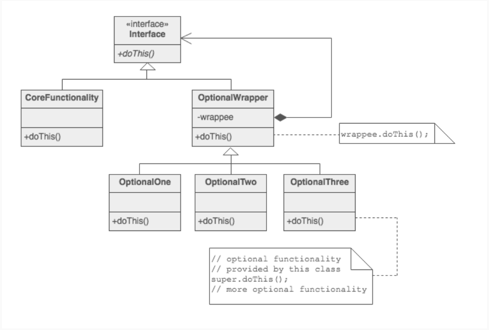


**Figure 2** – The Decorator design pattern

## The base class(1)

- The basic representation of an object
  *Without the characteristics of the options.*

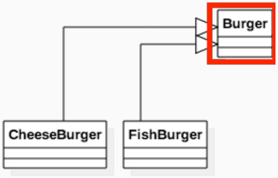


**Figure 3** – The base class

## The base class(2)

```python
class Burger:
    def __init__(self, name, sauce):
        self.name = name
        self.sauce = sauce
        self.element = list()

    def __repr__(self):
        return "{} sauce {}".format(self.name, self.sauce)

    def total(self):
        return sum(elem[1] for elem in self.element)
```

## The concrete class(1)

- The representation of a concrete object
  *Inheritance of the basic object.*
- Specification of the main class **Burger**
  *Object with its own characteristics.*

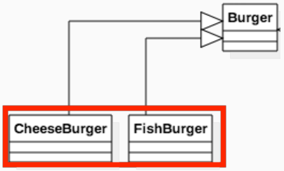


**Figure 4** – The concretes classes

## The concrete class(2)

```python
class CheeseBurger(Burger):
    def __init__(self, sauce):
        self.name = "CheeseBurger"
        super().__init__(self.name, sauce)
        self.element.append(("Bread", 1.00))
        self.element.append(("Cheese", 0.5))
        self.element.append(("Beef", 1.00))
        self.element.append((sauce, 0.5))


class FishBurger(Burger):
    def __init__(self, sauce):
        self.name = "FishBurger"
        super().__init__(self.name, sauce)
        self.element.append(("Bread", 1.00))
        self.element.append(("Fish", 1.50))
        self.element.append((sauce, 0.5))
```

## The decorator class(1)

- An abstract class of the basic object **Burger**
  *Encapsulation of the original object inside an abstract wrapper interface.*
- Giving the abilities to specify
  *Abstract class to attach a combination of features at concrete class.*

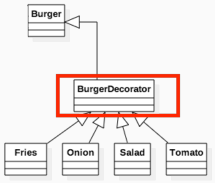


**Figure 5** – The decorator class

## The decorator class(2)

```python
class BurgerDecorator(Burger):
    def __init__(self, burger, supp):
        super().__init__(burger.name, burger.sauce)
        self.burger = burger
        self.supp = supp
        self.element = burger.element

    def __repr__(self):
        string = repr(self.burger)
        if "with" not in string:
            string += " with "
        if self.supp not in string:
            string += "{} ".format(self.supp)
        return string
```

## The options classes(1)

- The features to wrap a concrete object
  *Inheritance of the abstract class decorator.*

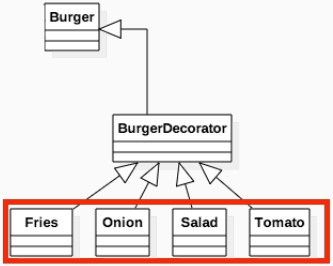


**Figure 6** – The options classes

## The options classes(2)

```python
class Tomato(BurgerDecorator):
    def __init__(self, burger):
        self.supp = "tomato"
        super().__init__(burger, self.supp)
        self.burger.element.append((self.supp, 0.2))


class Salad(BurgerDecorator):
    def __init__(self, burger):
        self.supp = "salad"
        super().__init__(burger, self.supp)
        self.burger.element.append((self.supp, 0.2))


class Oinon(BurgerDecorator):
    def __init__(self, burger):
        self.supp = "oinon"
        super().__init__(burger, self.supp)
        self.burger.element.append((self.supp, 0.2))
```

## Example(1)

```python
if __name__ == '__main__':
    cheese_burger = CheeseBurger("Ketchup")
    print(cheese_burger)
    cheese_burger_with_tomato = Tomato(cheese_burger)
    print(cheese_burger_with_tomato)
    fish_burger = Fries(Tomato(Salad(FishBurger("Tartar"))))
    print(fish_burger)
```

- CheeseBurger sauce Ketchup
- CheeseBurger sauce Ketchup with tomato
- FishBurger sauce Tartar with salad tomato fries

## Example(2)

- The complete diagram of the application
  *The main class **Burger** with concrete classes and the options to wrap and decorate concrete classes.*

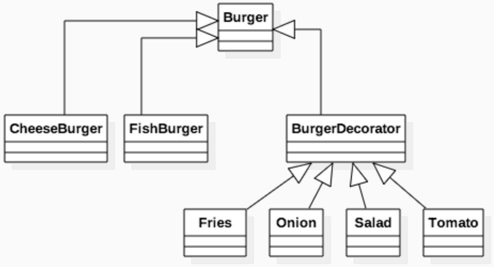
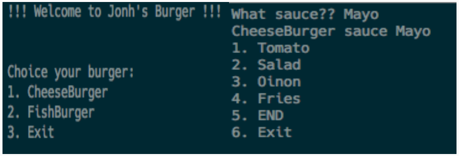


**Figure 7** – The complete diagram

## Application

- Output of the little application (main.py)

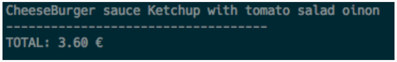


```
!!! Welcome to Jonh's Burger !!!


Choice your burger:
1. CheeseBurger
2. FishBurger
3. Exit
```

```
What sauce?? Mayo
CheeseBurger sauce Mayo
1. Tomato
2. Salad
3. Oinon
4. Fries
5. END
6. Exit
```

**Figure 8** – The menus of main.py (cf : GitHub)



```
CheeseBurger sauce Ketchup with tomato salad oinon
----------------------------------
TOTAL: 3.60 €
```

**Figure 9** – The ouput of main.py (cf : GitHub)

## Conclusion

The decorator pattern used when :

- A base object have multiple derivates
  *A **Burger** may be a **CheeseBurger** or **FishBurger**.*
- Derivates may be wrap with same or differentes features
  ***CheeseBurger** with tomato or **FishBurger** with salad and tomato.*
- Decorator pattern allows to add new derivates or options easily
  *A new concrete class **VegetarianBurger** or a new option **pickels**.*

## Enjoy !

The application burger is available on GitHub :

- `https://github.com/JonathanPetit/Decorator-design-pattern`
- The manual is the README.md

## Bibliography i

- https://sourcemaking.com/design_patterns/decorator
- https://en.wikipedia.org/wiki/Decorator_pattern
- https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm
- Architecture logiciel slides - Mr. Combéfis
- https://github.com/matze/mtheme