Decorator Design Pattern

Jonathan Petit

27 octobre 2017

ECAM

Decorator

Context and application

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

Bad structure

- An base class "Windows"

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

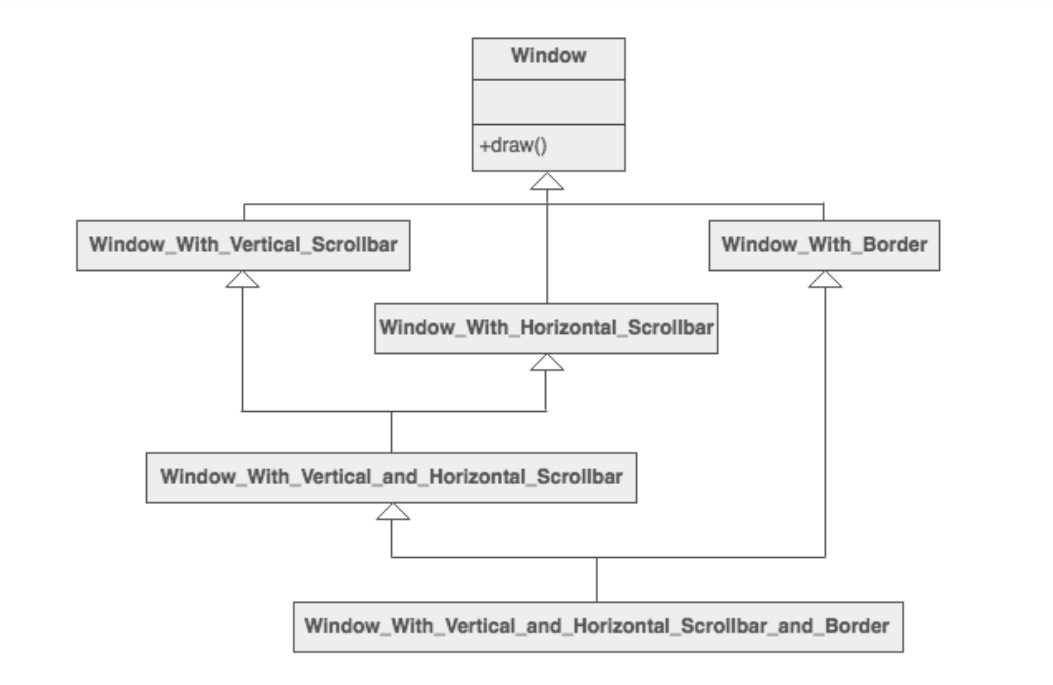

Figure 1 – Bad structure

Good structure

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

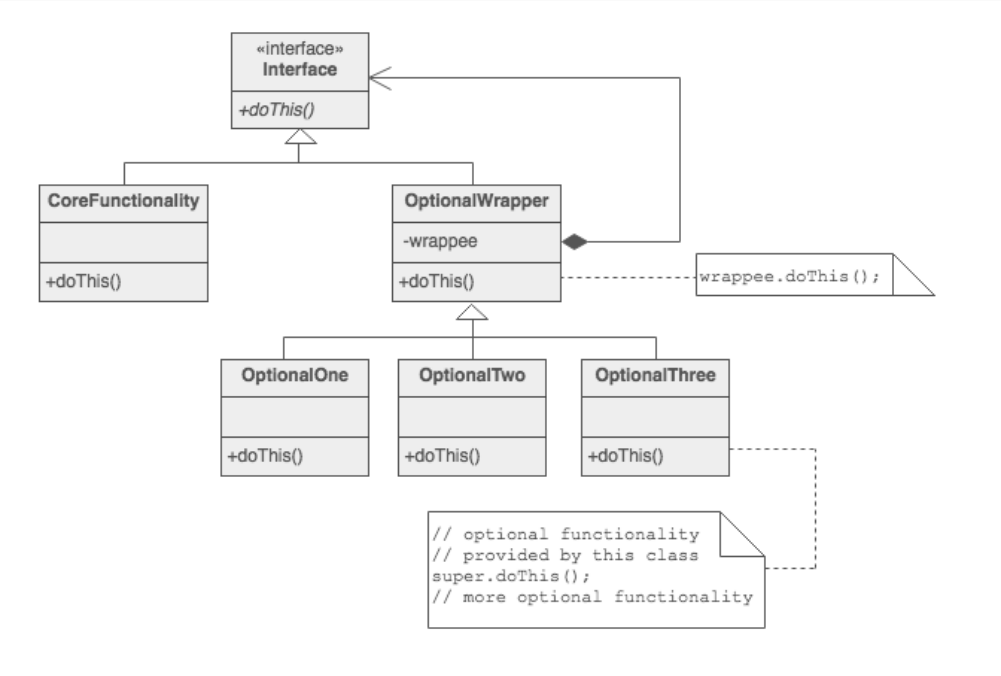

Figure 2 – The Decorator design pattern

The base class(1)

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

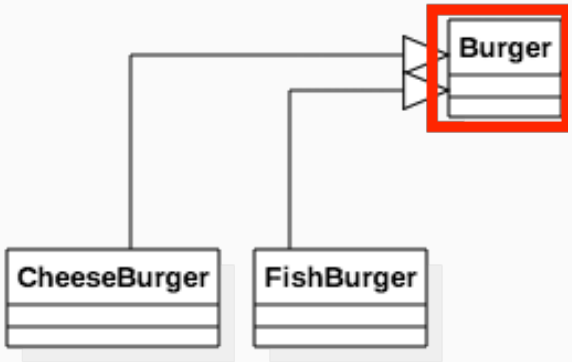

Figure 3 – The base class

The base class(2)

```
1  class Burger:
2      def __init__(self, name, sauce):
3          self.name = name
4          self.sauce = sauce
5          self.element = list()
6
7      def __repr__(self):
8          return "{} sauce {}".format(self.name, self.sauce)
9
10     def total(self):
11         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 it's own characteristics.

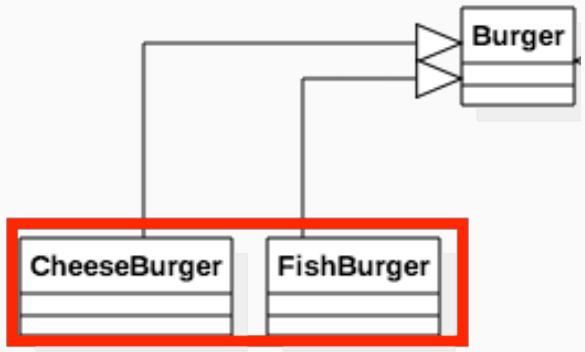

Figure 4 – The concretes classes

The concrete class(2)

```
1  class CheeseBurger(Burger):
2      def __init__(self, sauce):
3          self.name = "CheeseBurger"
4          super().__init__(self.name, sauce)
5          self.element.append(("Bread", 1.00))
6          self.element.append(("Cheese", 0.5))
7          self.element.append(("Beef", 1.00))
8          self.element.append((sauce, 0.5))
9
10
11 class FishBurger(Burger):
12     def __init__(self, sauce):
13         self.name = "FishBurger"
14         super().__init__(self.name, sauce)
15         self.element.append(("Bread", 1.00))
16         self.element.append(("Fish", 1.50))
17         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.

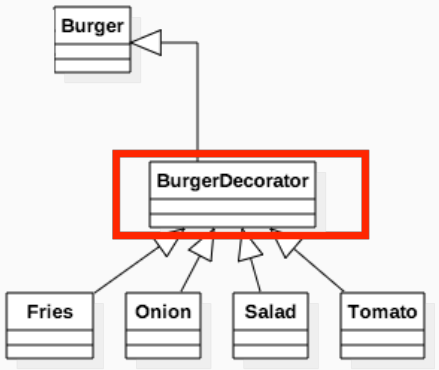

Figure 5 – The decorator class

The decorator class(2)

```
1  class BurgerDecorator(Burger):
2      def __init__(self, burger, supp):
3          super().__init__(burger.name, burger.sauce)
4          self.burger = burger
5          self.supp = supp
6          self.element = burger.element
7
8      def __repr__(self):
9          string = repr(self.burger)
10         if "with" not in string:
11             string += " with "
12         if self.supp not in string:
13             string += "{} ".format(self.supp)
14         return string
```

The options classes(1)

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

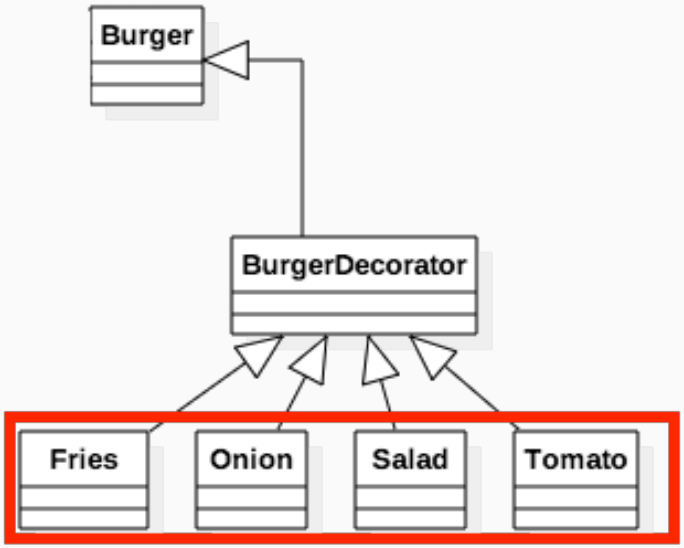

Figure 6 – The options classes

The options classes(2)

```
1  class Tomato(BurgerDecorator):
2      def __init__(self, burger):
3          self.supp = "tomato"
4          super().__init__(burger, self.supp)
5          self.burger.element.append((self.supp, 0.2))
6
7
8  class Salad(BurgerDecorator):
9      def __init__(self, burger):
10         self.supp = "salad"
11         super().__init__(burger, self.supp)
12         self.burger.element.append((self.supp, 0.2))
13
14
15 class Oinon(BurgerDecorator):
16     def __init__(self, burger):
17         self.supp = "oinon"
18         super().__init__(burger, self.supp)
19         self.burger.element.append((self.supp, 0.2))
```

Example

```
1 if __name__ == '__main__':
2     cheese_burger = CheeseBurger("Ketchup")
3     print(cheese_burger)
4     cheese_burger_with_tomato = Tomato(cheese_burger)
5     print(cheese_burger_with_tomato)
6     fish_burger = Fries(Tomato(Salad(FishBurger("Tartar"))))
7     print(fish_burger)
```

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

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