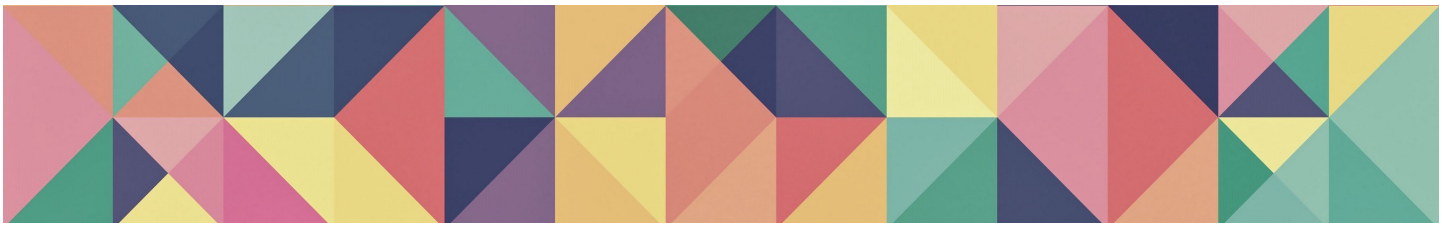
Coding better using Design Patterns

A Quick Reference for a fast reference in case of need

Federico Haag
Dec 30, 2017 · 4 min read



> Design Patterns are formalized best practices used to solve the most common problems that generally occur during OOP Software Development.

Table of Contents

. . .

1. Creational Patterns

Creational patterns deal with **object creation** mechanisms, providing some special features or abstracting some common tasks.

1.1. Singleton Pattern

Singleton Pattern ensures that **a class has only one instance and provides a global point of access to it**. Typical Use Cases are logging classes, database connection classes, file manager classes.

```
public class Singleton{
        //single instance
        final private static Singleton instance = null;

        //hidden constructor
        private Singleton(){}

        //public interface for this class
        public static Singleton get_instance(){
                //create object only if it does not exists
                if(instance==null){
                        this.instance = new Singleton();
                }
                return this.instance;
        }
}
```

singleton.java hosted with ♥ by GitHub view raw

1.2. Factory Pattern

The Factory Pattern should be used when you have to create an instance of a class, but depending on some variable input you may have to create a sub-type A or a sub-type B of such a class. In this case, it's considered good design setting up a class (called Factory) which encapsulate all the logic flows needed to chose which sub-type has to be generate.

Have a look at the following example:

```
abstract class Animal{
    String name;
    String ownerName;
}

class Dog extends Animal {
    // specific attributes and methods...
}

class Cat extends Animal {
```

```
    // specific attributes and methods...
}

class AnimalFactory {
    public Animal createNewAnimal(){

        if( ... ){ return new Dog(); }
        else { return new Cat(); }

    }
}
```

factory.java hosted with ♥ by GitHub view raw

. . .

2. Structural Patterns

Strucutral Patterns deal with the creation of complex object structures through the composition of classes and objects.

2.1. Adapter Pattern

The Adapter Pattern is used in case of need to convert a class interface into another one, for example due to communication issues.

```
public class AdapteeToClientAdapter implements Adapter {

    private final Adaptee instance; //the class to be adapted

    //constructor
    public AdapteeToClientAdapter(final Adaptee instance) {
         this.instance = instance;
    }

    //client method implemented calling a method of the adaptee class
    public void clientMethod() {
       instance.adapteeMethod();
    }

}
```

adapter.java hosted with ♥ by GitHub view raw

2.2. Decorator Pattern

The Decorator Pattern **adds behavior to an object dynamically without affecting the behavior of other objects from the same class**.

It is a **dynamic alternative to subclassing** for extending functionality, particularly when you have to **extend functionality at run-time**.

```
interface BurgerInterface {
  public int getPrice();
}

class Burger implements BurgerInterface {
  public int getPrice(){
    return 2;
  }
}

class BurgerDecorator implements Burger {
  private Burger burger;
  public BurgerDecorator(Burger b){
    this.burger = b;
  }

  public int getPrice(){
    return this.burger.getPrice();
  }
}

class TomatoDecorator extends BurgerDecorator {
  @Override
  public int getPrice(){
    return super.getPrice() + 1;
  }
}

class SaladDecorator extends BurgerDecorator {
  @Override
  public int getPrice(){
    return super.getPrice() + 0.5;
  }
}

// Build a burger with tomato
Burger myBurger = new TomatoDecorator( new Burger() );
System.out.println(myBurger.getPrice()); // prints 3 (2+1)
// Add salad to myBurger
myBurger = new SaladDecorator( myBurger );
```

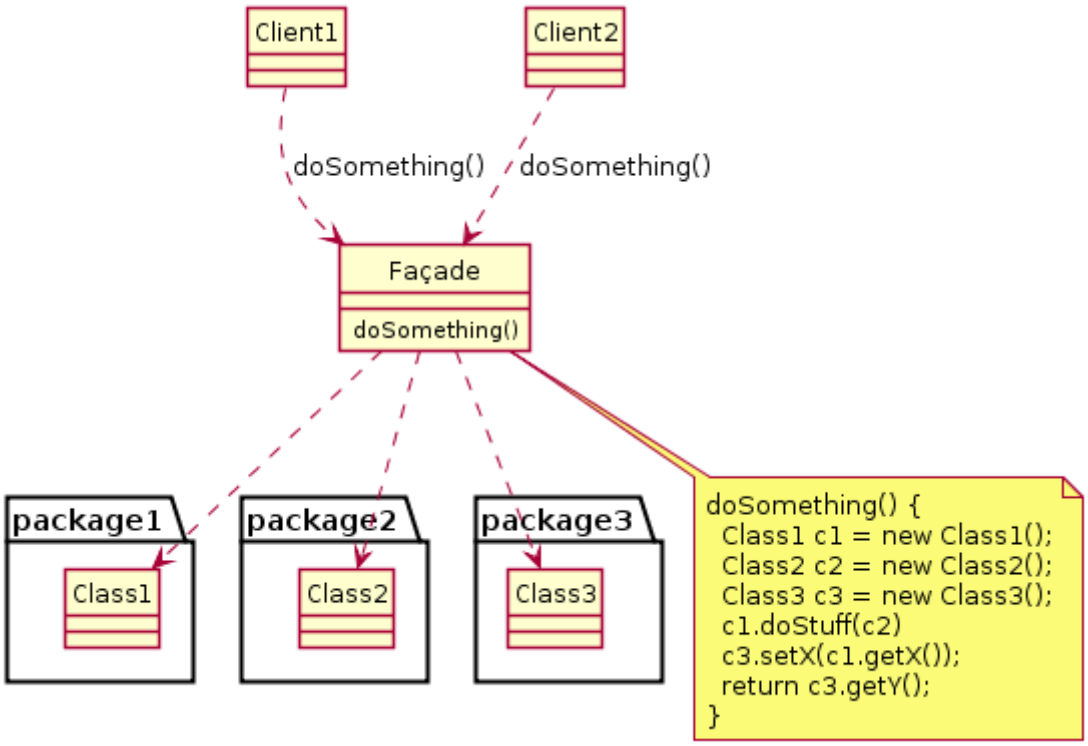
```
41  System.out.println(myBurger.getPrice()); // prints 3.5 ((2+1)+0.5)
```

decorator.java hosted with ♥ by GitHub view raw

2.3. Facade Pattern

The Facade Pattern provides a **simplified interface** to a **larger body of code**.

It typically involves a single **wrapper class** that contains a set of variables and methods required by the client that access the full body of code and **hide the implementation details** in order to **reduce the client perceived complexity of the code.**

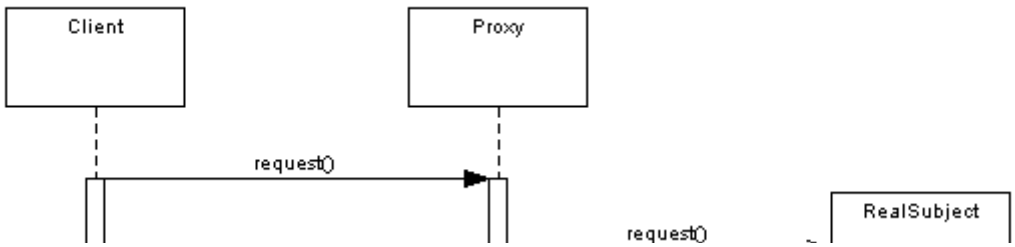


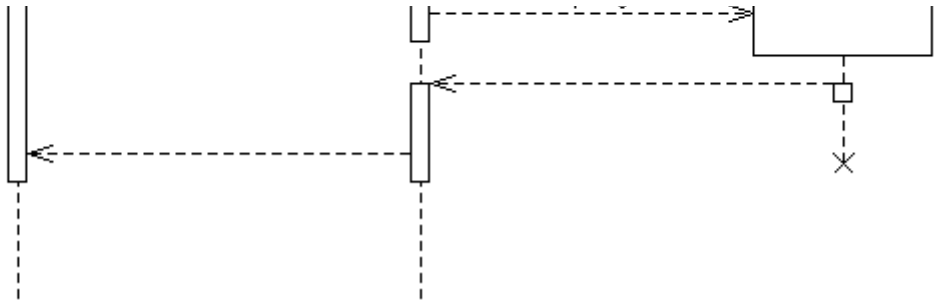
Credits: https://en.wikipedia.org/wiki/File:Example_of_Facade_design_pattern_in_UML.png

2.4. Proxy Pattern

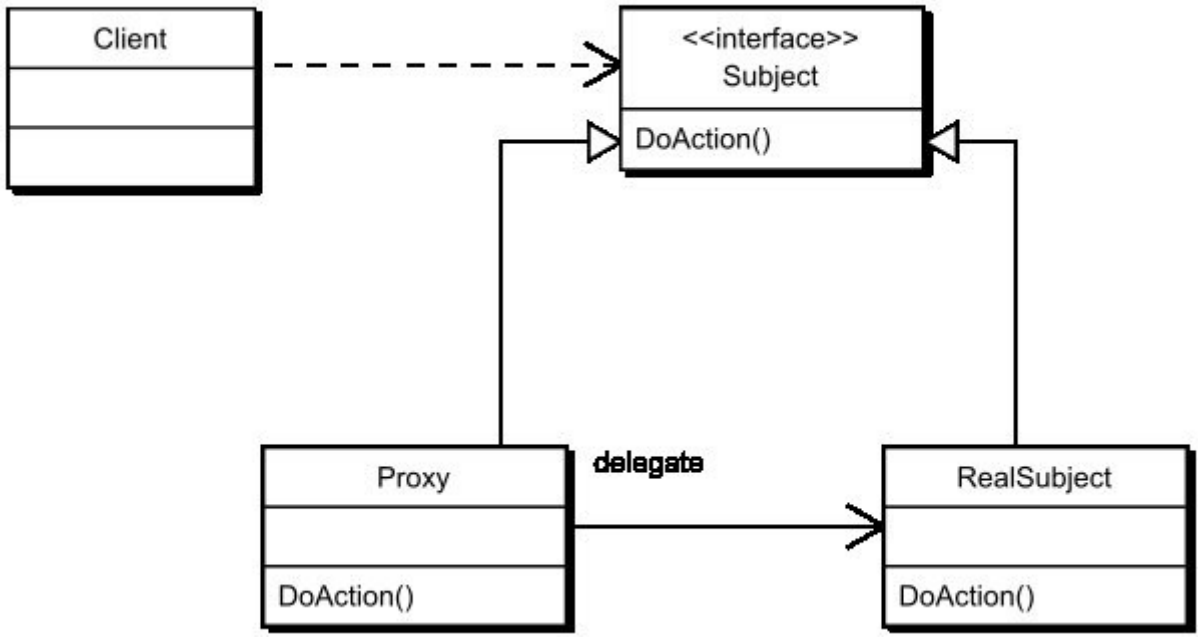
The Proxy Pattern is used to **postpone or avoid instantiation of an heavy object** using a **lighter object that has the same interface**.

The Proxy Pattern is often used in combination with preprocessing techniques, caching, requests queuing.





Sequence Diagram



UML Diagram

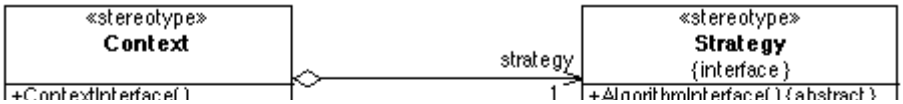
. . .

3. Behavioral Patterns

Behavioral patterns deal with the **communication between objects.**

3.1. Strategy Pattern

The Strategy Pattern is used to **select at runtime** which algorithm has to be executed. It's useful when a class has different behaviors depending on a variable input. A bad design to solve this use case would be building a huge switch where you insert the all possible algorithms based on the variable input. This solution is considered the best choice as a matter of good OOP design.



UML Diagram

```java
public interface Strategy {
   public int doOperation(int num1, int num2);
}

public class OperationAdd implements Strategy{
   public int doOperation(int num1, int num2) {
      return num1 + num2;
   }
}

public class OperationSubstract implements Strategy{
   public int doOperation(int num1, int num2) {
      return num1 - num2;
   }
}

public class Context {
   private Strategy strategy;

   public Context(Strategy strategy){
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}
```

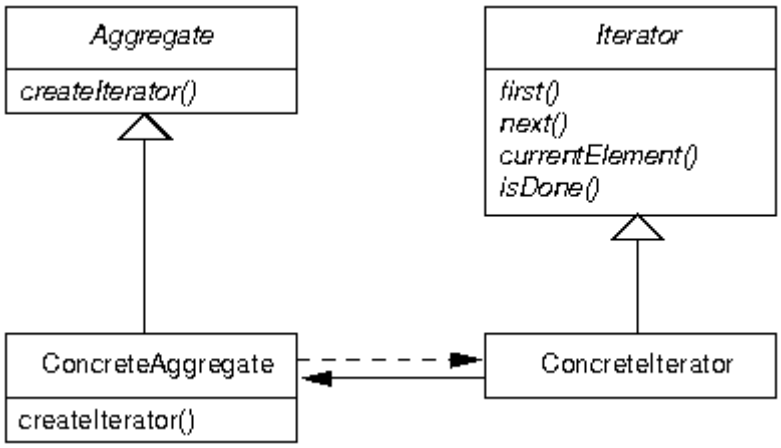
strategy.java hosted with ♥ by GitHub view raw

3.2. Iterator Pattern

The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Iterators decouple the iteration algorithms from the original data structure making the code lighter and more modular, enabling also multiple concurrents

iterations (one iteration <->one iterator).

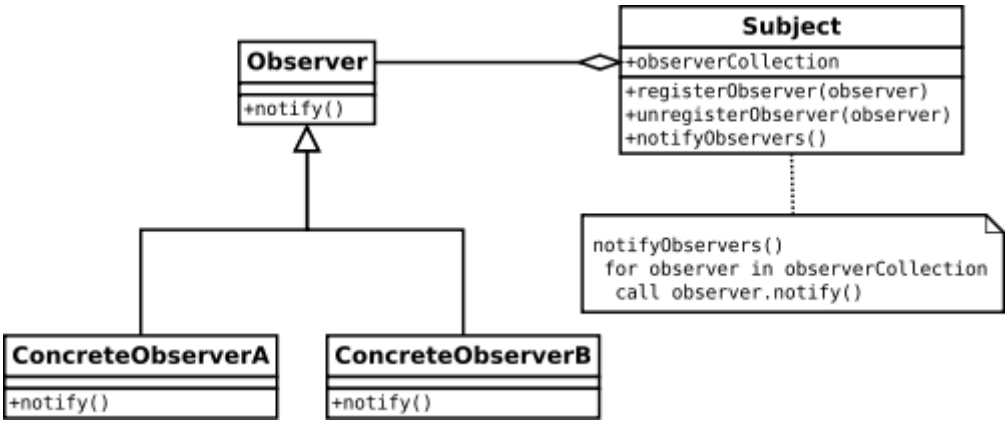


UML Diagram

3.3. Observer Pattern

The Observer Pattern defines a one-to-many dependance between objects (subject — listeners) in order to notify all the dependents when a change of state occurs in the subject.

> ***Warning:*** *this pattern cause a problem known as "Lapsed listener problem". The solution is using weak references between subject and listeners.*



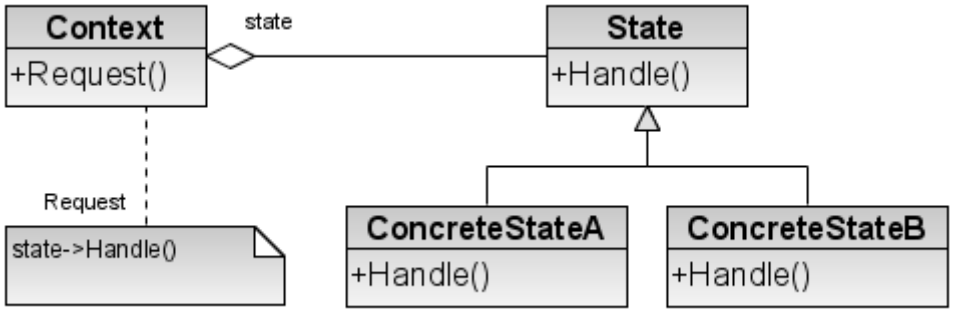
UML Diagram

3.4. State Pattern

Applying the State Pattern you can create an object with a state variable on which the object behaviors depend.

More eaisly: it's the object-oriented implementation of a "state machine".

Unlike the strategy pattern, where the choice on the algorithm to be applied is took base on an input and smoothly applied, the state pattern changes the object behaviors by it self depending on the current object state and as a consequence of a method invocation or internal timing.



UML Diagram

4. Other Patterns

The current best reference you can refer to is **Head First Design Patterns.**

Programming Software Development Software Engineering Design Patterns

About Help Legal