Design Patterns - Summary

Semester: Spring 2022

Version: 0.1.0
Date: 2022-06-01 11:29

School of Computer Science
OST Eastern Switzerland University of Applied Sciences

Contents

1 Introduction

1.1 The design patterns

Class Design Patterns deal with the relationships between classes and their subclasses. The Object Design Patterns with the object relationships which can be changed at runtime.

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (121)	Adapter (157)	Interpreter (274) Template Method (360)
	Object	Abstract Factory (99) Builder (110) Prototype (133) Singleton (144)	Adapter (157) Bridge (171) Composite (183) Decorator (196) Facade (208) Flyweight (218) Proxy (233)	Chain of Responsibility (251) Command (263) Iterator (289) Mediator (305) Memento (316) Observer (326) State (338) Strategy (349) Visitor (366)

Figure 1: Design pattern space

2 Design Patterns

2.1 Creational Patterns

2.1.1 Abstract Factory

The Abstract Factory provides an interface for creating families of related or dependent objects without specifying their concrete class. It is a good choice, when you want to create big class hierarchy from a factory. In general the Simple Factory is more common today.

Motivation:

- You want to create concrete objects without knowing which concrete type you get.
- You are only interested in the interface.

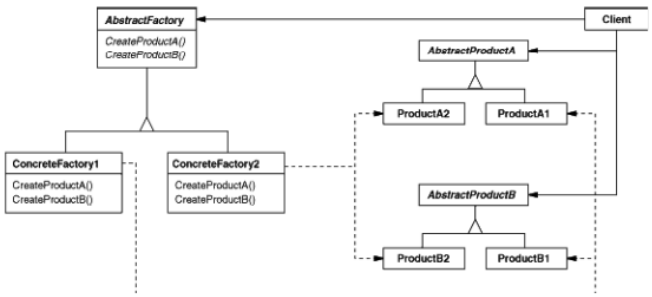


Figure 2: Abstract Factory Class Diagram

Important Notes: The Factory must be created at the very beginning of the application. After creation only this factory must be used to create objects. Otherwise, the benefit of the factory are gone.

2.1.2 Builder

The Builder Pattern separates the construction of a complex class from its representation. Thereby you can hide the creation of complex objects behind functions.

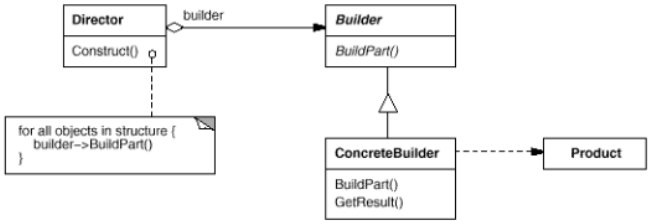


Figure 3: Builder Class Diagram

2.1.3 Factory Method

The Factory Method is used to create an object, but the subclass decides which class is instantiated. With this pattern you can create concrete creators which can decide which concrete product should be created. The user of the creator only has to use the interface.

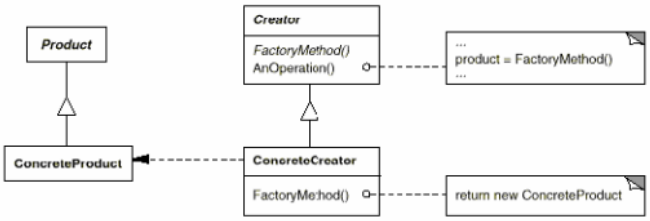


Figure 4: Factory Method Class Diagram

2.1.4 Prototype

The prototype pattern is something similar for objects like the factory method is for classes. Using the prototype pattern you create copies from concrete objects. This is achieved by cloning the prototype.

Important Notes: The crucial part of this pattern is the `Clone()` function. During the implementation you have to decide if you want to do a deep copy or a shallow copy.

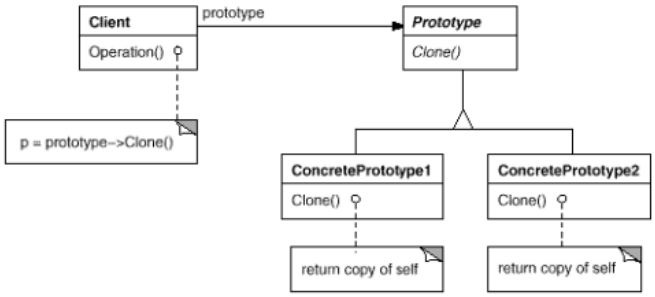


Figure 5: Prototype Class Diagram

2.1.5 Singleton

The Singleton pattern ensures that a class has only one (global) instance. Additionally, it provides easy access to this element.

However, you should **NOT** use it in your software. It is basically a global variable. Therefore, not testable and difficult to maintain.

If you want something like singleton, you should use a simple factory which always returns the same instance.

```
class SingletonFactory {
    private static object myInstance;

    public object createObject() {
        if (SingletonFactory.myInstance == null) {
            SingletonFactory.myInstance = new();
        }
        return SingletonFactory.myInstance;
    }
}
```

Listing 1: Singelton alternative in code

2.2 Structural Patterns

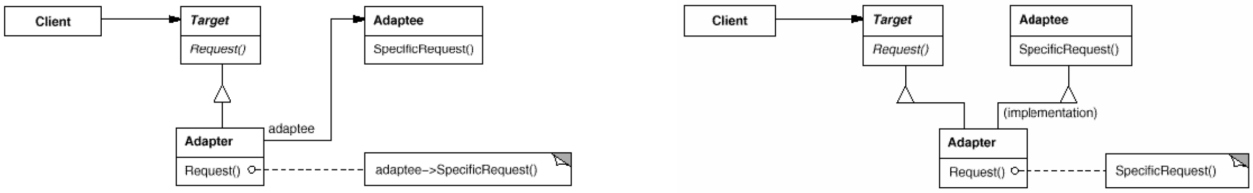
2.2.1 Adapter

The Adapter Design Pattern is used to make a class compatible to others (like a power adapter). The adapter pattern has two different types:

- class adapter
- object adapter

Both patterns do the same thing. The class adapter inherits from the target **AND** from the adaptee (6b). The object adapter inherits only from the target and holds the adaptee as an instance variable (6a).

The class adapter is not often used because many languages do not support multi-inheritance, and you should favor object composition over class inheritance.



(a) Object Adapter Class Diagram (b) Class Adapter Class Diagram

Figure 6: Adapter Class Diagrams

2.2.2 Bridge

The Bridge Design Pattern is used to decouple an abstraction from its implementation so that the two can vary independently.

Example: Your application should support multiple window systems (X, Windows). The client (your application) should be able to create windows, without committing to a concrete implementation. Only your window implementation should depend on the target platform (X, Windows).

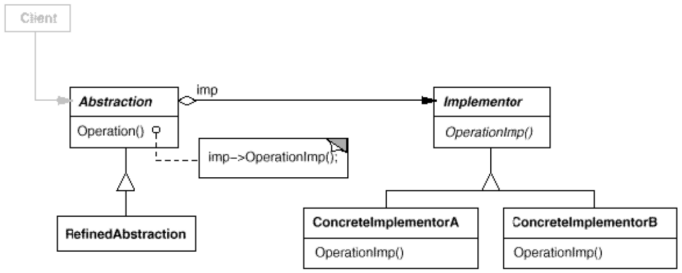
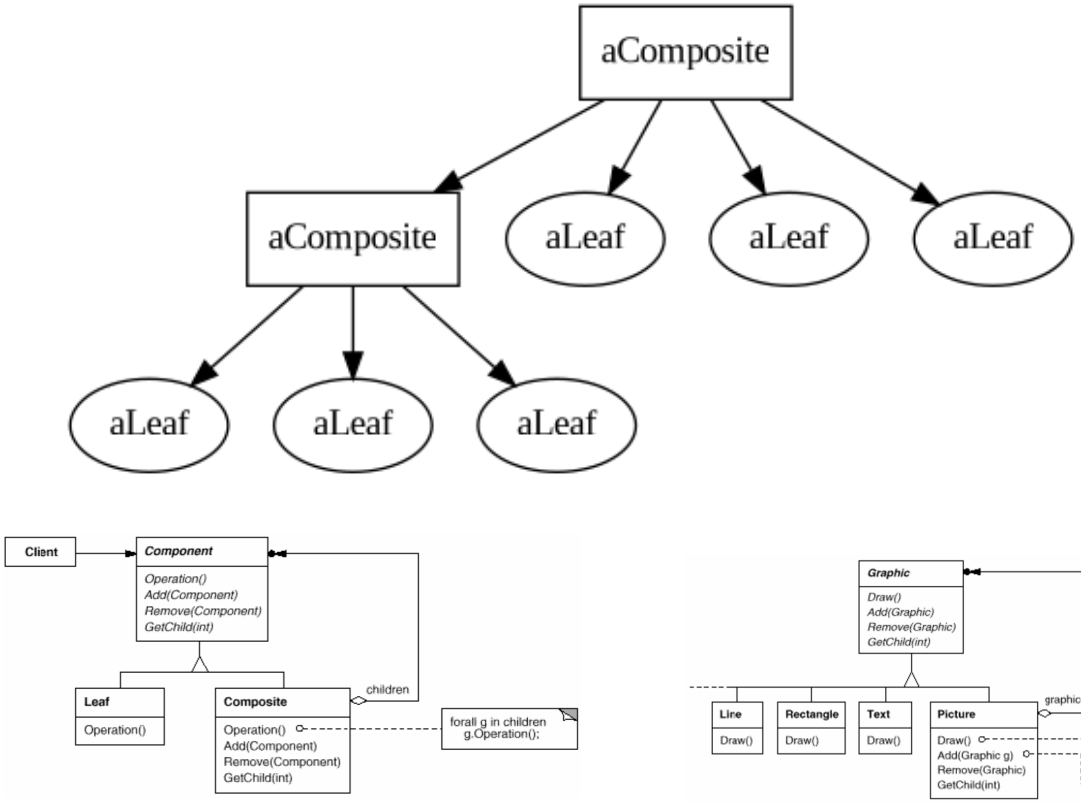


Figure 7: Bridge Class Diagram

2.2.3 Composite

The Composite Design Pattern is used to model a part-whole hierarchy. The pattern let clients treat individual and compositions of objects uniformly.



(a) Composite Class Diagram (b) Example of Composite

Figure 8: Composite Diagrams

Example: It exists many kinds of graphics and forms (Line, Rectangle, Picture, …). If the client wants to draw any graphic, it does not care how draw. Therefore, we need one function (`draw`) for all kind of graphics. However, a picture consists of many lines, rectangles and more graphics. This picture class has some more functions (`Add`, `Remove`, `GetChild`). The `Draw` function iterates over all children and calls their `Draw` function.

2.2.4 Decorator

The Decorater Pattern is used to attach additional responsibility (features) to an object dynamically. For example, a class implements only the login mechanism. Using the

decorater pattern exception handling can be done in a separate class.

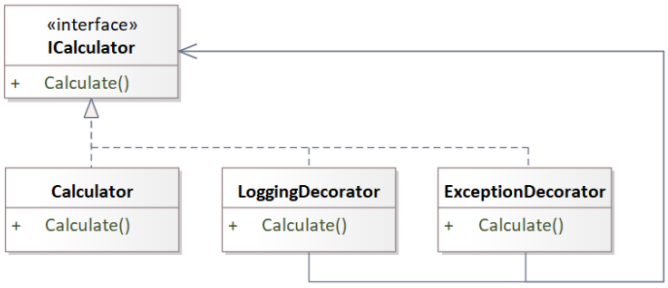


Figure 9: Decorater Diagram

2.2.5 Facade

The Facade Design Pattern is used to provide a simple interface to a set of interfaces (subsystem). For example the Compiler class provide an easy-to-use interface for the whole compiler subsystem (figure 11).

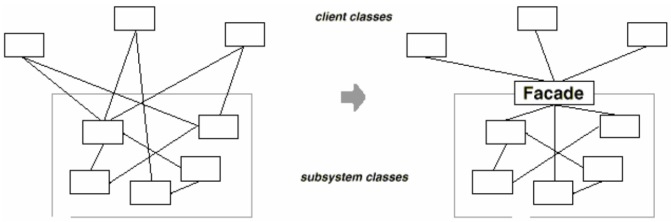


Figure 10: Facade Use Case

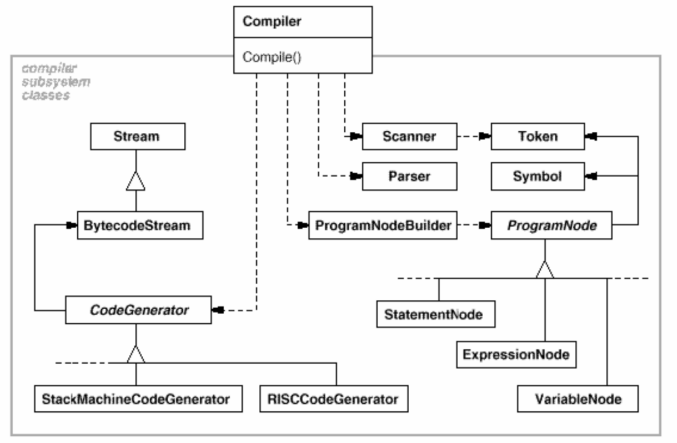


Figure 11: Facade Example

2.2.6 Flyweight

The Flyweight pattern is used to support large number of objects efficiently. For example, you can store every character in its own object. In a document with 1000 characters you need more than 1000 objects. The Flyweight pattern helps here.

In the Flyweight class is only state independent stuff stored. Therefore, the Flyweight is shareable. Instead, creating every time a new object with the character ”a” you always reference to the same object.

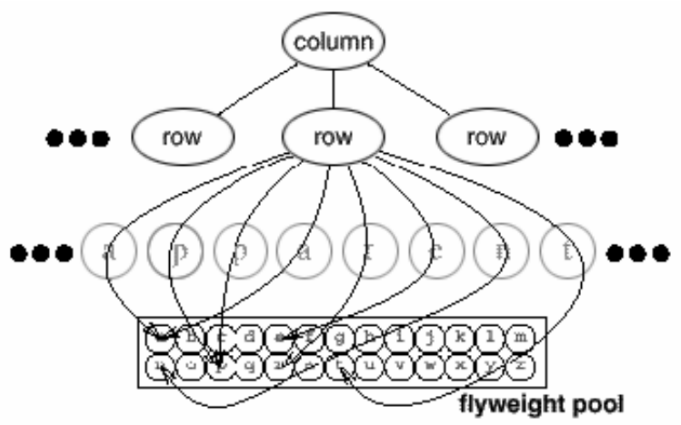


Figure 12: Flyweight Example

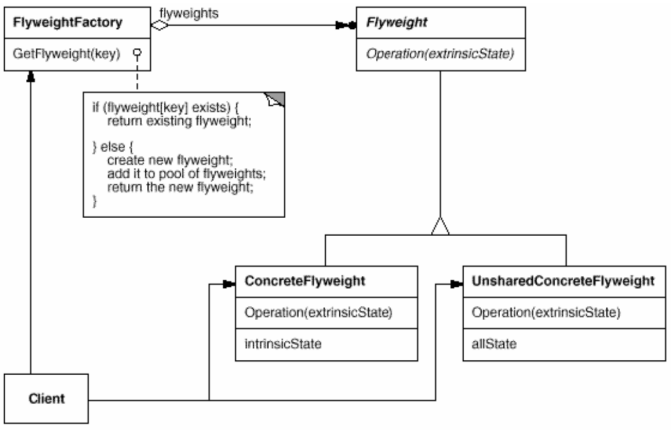


Figure 13: Flyweight Class Diagram

2.2.7 Proxy

The Proxy Design pattern provides a surrogate or placeholder for another object to control access to it.

A Feed Reader must load the news from a (slow) server. When a frontend want to display the news before the data are available you have to provide a loading screen. This can be easily done using the Proxy pattern.

The Proxy accepts the request, checks if the data are available. If not, it provides the login screen. If the news are loaded, it returns the news.

2.3 Behavioral Patterns

2.3.1 Chain of Responsibility

The Chain of Responsibility is used to decouple the sender of a request to its receiver by giving more than one class the change to handle the request. The first object takes the

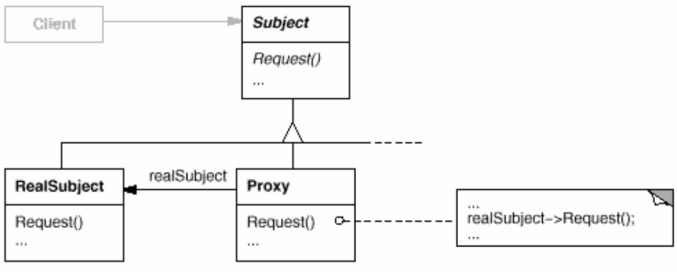


Figure 14: Proxy Class Diagram

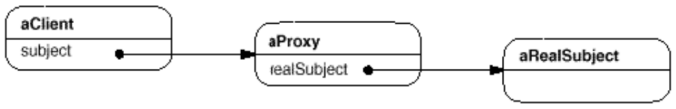
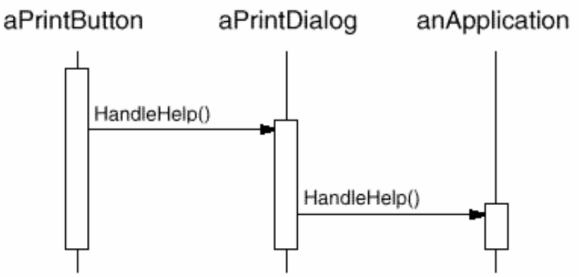
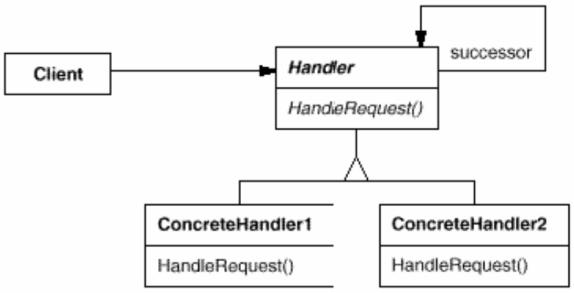


Figure 15: Object Diagram for proxy

request, check if it can handle. When yes, then handle it. If not, forward the request to the parent / successor.



(a) CoR Sequence



(b) CoR Class Diagram

Figure 16: Chain of Responsibilty Diagrams

2.3.2 Command

The command pattern is used to encapsulate actions / request inside an object. For example, the design of a framework does not know which action the button should perform. Therefore, the button is created using a command as parameter. As soon as the user clicks on the button, the button executions the `command.Execute()` function.

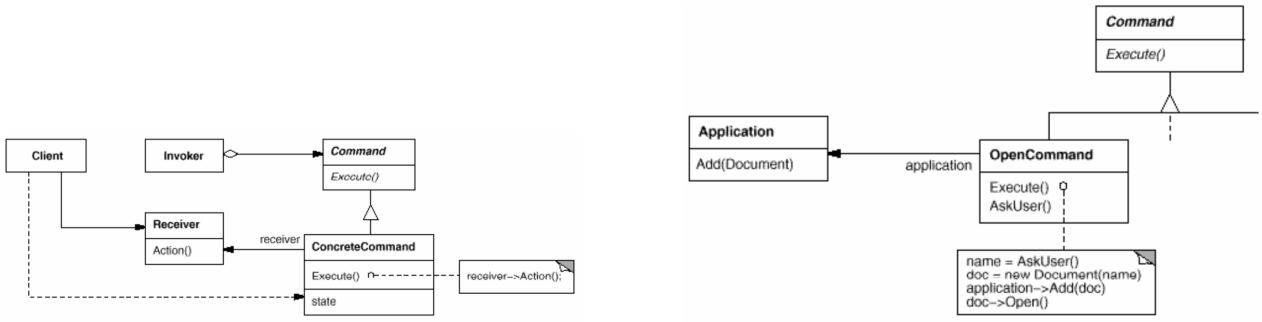
2.3.3 Interpreter

The Interpreter Design Pattern is used to interpret a simple grammar / language. Every grammar rule is modeled as one subclass.

My thoughts: Before you implement your own interpreter consider a specialized library for this.

2.3.4 Iterator

The iterator design pattern is used to access the items in a Data structure (Aggregate in figure 19). The benefit of iterator is that it hides the implementation details of the data structure.



(a) Command Class Diagram (b) Example of Command

Figure 17: Command Class Diagrams

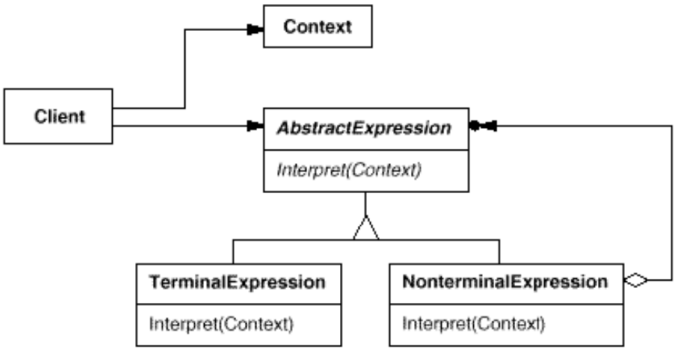


Figure 18: Interpreter Class Diagram

2.3.5 Mediator

The Mediator Design Pattern is used to encapsulate how a set of objects interact with each other. The different objects do not have to reference each other explicit.

How does it work: The mediator holds a reference to every object. The user selected some item from the list box (object A). Object A notifies the mediator that the value has changed. The mediator know notifies the other objects.

2.3.6 Memento

The Memento Design Pattern is used to store an internal state. This is often used for restoring the internal state of an object.

The object creates a memento (a copy of the state itself) and returns it to the world.

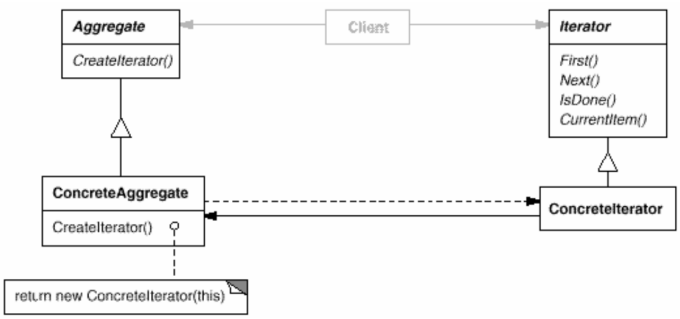
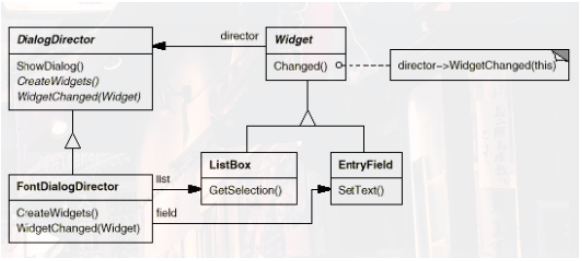
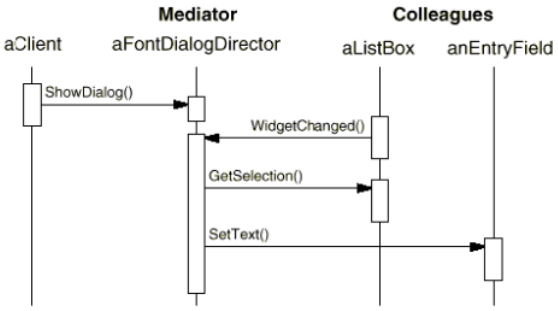


Figure 19: Iterator Class Diagram



(a) Mediator Class Diagram



(b) Mediator Sequence Diagram

Figure 20: Mediator Diagrams

The state of the object can be restored using such a memento. You can not change the memento and the inner live is unknown to the outer world.

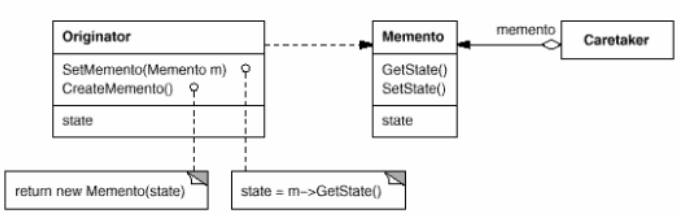


Figure 21: Memento Class Diagram

2.3.7 Observer

The goal of the Observer Pattern is to resolve a cyclic dependency. The Pattern consists of two objects:

- Subject: is monitored (e.g. a model)
- Observer: monitors the subject (e.g. a view)

The observer register itself on the subject. The only requirement is that the subject implements a specific interface. Using this approach the domain does not have to need anything from the view (excepted the interface).

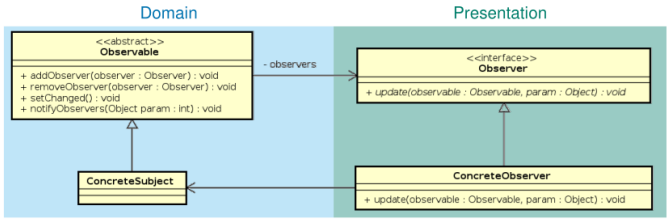
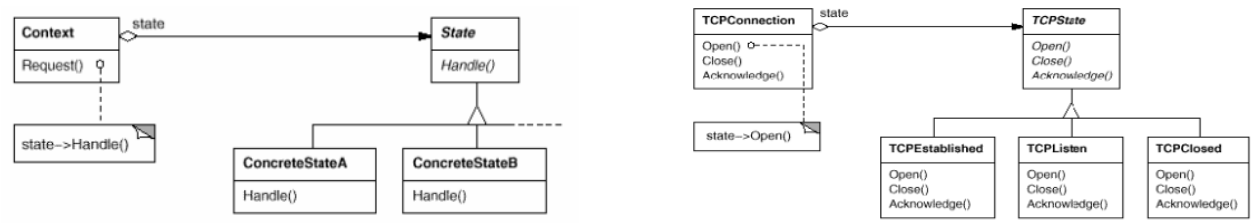


Figure 22: Observer Class Diagram

2.3.8 State

The state pattern is used allow an object to alter its internal behavior depending on its state. The object will appear to change its class.



(a) State Class Diagram (b) TCP Connection with State

Figure 23: State Class Diagrams

2.3.9 Strategy

The Strategy Pattern is used to encapsulate an algorithm in an object. With this approach you can change the algorithm at runtime.

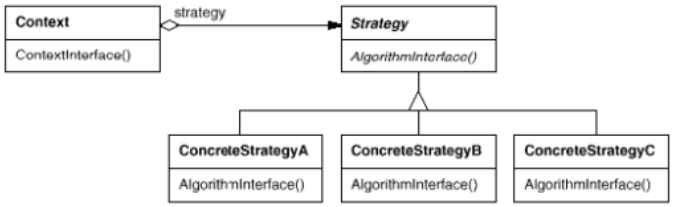


Figure 24: Strategy Class Diagram