Design Patterns - Summary

Semester: Spring 2022

Version: 0.1.0
Date: 2022-05-23 08:08

School of Computer Science
OST Eastern Switzerland University of Applied Sciences

Contents

1 Introduction

1.1 Most important quotes

In the introduction the basics of object-oriented programming are explained. Therefore, for many this is easy to read. However, some important statements are made.

> Favor object composition over class inheritance

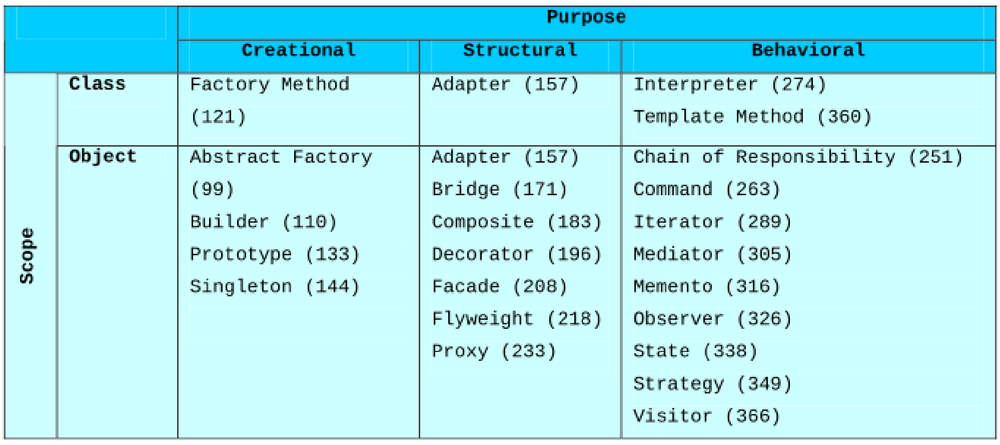
Because it is very difficult to make the correct abstraction often the base class(es) are not complete or have too much in it. With object composition you don't have this problem.

> Program to an interface, not an implementation

If you implement against interfaces you can test your class with unit tests. If you implement against a fixed implementation you can not change this behavior at anytime.

1.2 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.

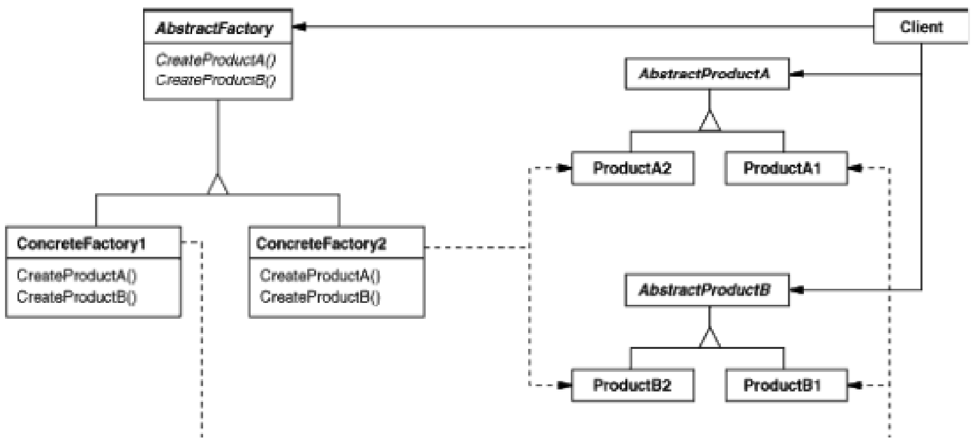


Figure 2: Abstract Factory Class Diagram

1. 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 seperates the construction of a complex class from its representation. Therby you can hidde the creation of complex objects behind functions (see listening 1).

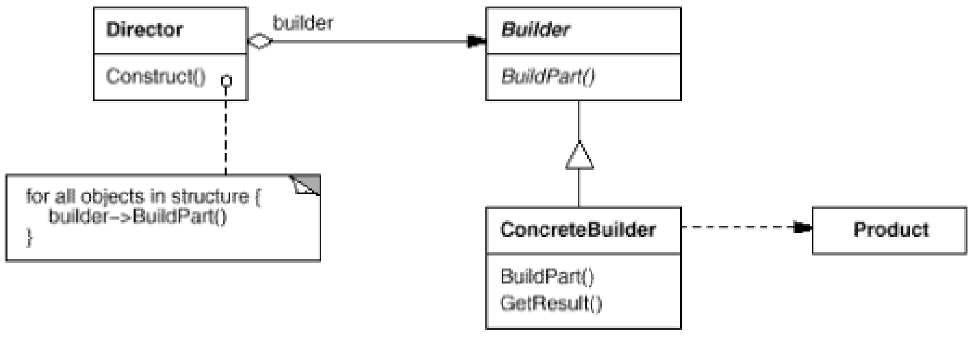


Figure 3: Builder Class Diagram

```
Maze* MazeGame::CreateMaze () {
  Maze* aMaze = new Maze;
  Room* r1 = new Room(1);
  Room* r2 = new Room(2);
  Door* theDoor = new Door(r1, r2);
  aMaze->AddRoom(r1);
  aMaze->AddRoom(r2);
  r1->SetSide(North, new Wall);
  r1->SetSide(East, theDoor);
  r1->SetSide(South, new Wall);
  r1->SetSide(West, new Wall);
  r2->SetSide(North, new Wall);
  r2->SetSide(East, new Wall);
  r2->SetSide(South, new Wall);
  r2->SetSide(West, theDoor);
  return aMaze;
}

// With Builder Pattern
Maze* MazeGame::CreateMaze (MazeBuilder& builder) {
  builder.BuildMaze();
  builder.BuildRoom(1);
  builder.BuildRoom(2);
  builder.BuildDoor(1, 2);
  return builder.GetMaze();
}
```

Listing 1: The Builder Pattern in Action

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.

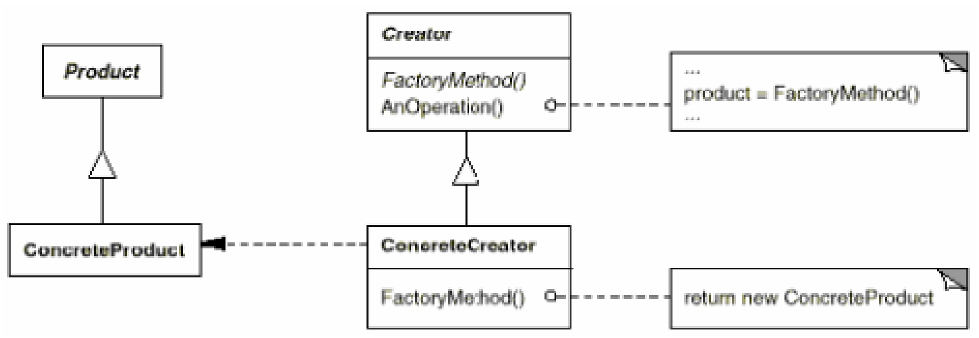


Figure 4: Factory Method Class Diagram

2.1.4 Prototype

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

1. 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.

   ```
   class MazePrototypeFactory : public MazeFactory {
   public:
     MazePrototypeFactory(Maze*, Wall*, Room*, Door*);
     virtual Maze* MakeMaze() const;
     virtual Room* MakeRoom(int) const;
     virtual Wall* MakeWall() const;
     virtual Door* MakeDoor(Room*, Room*) const;
   private:
     vMaze* _prototypeMaze;
     Room* _prototypeRoom;
   ```

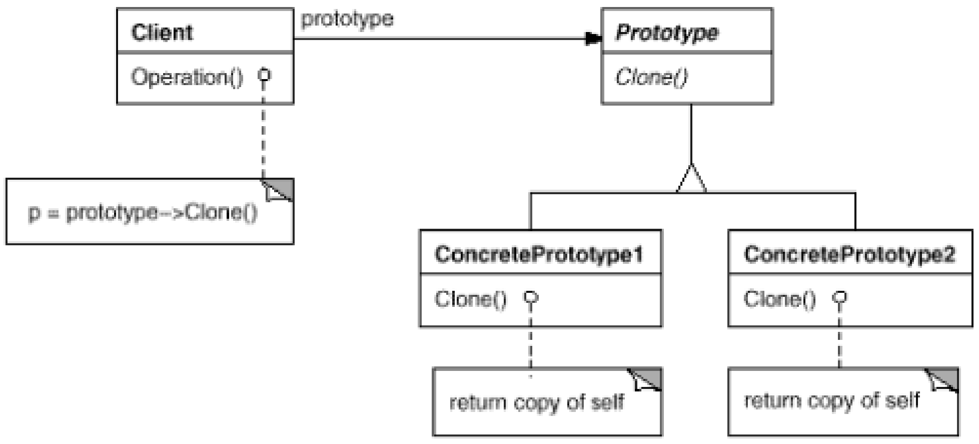


Figure 5: Prototype Class Diagram

```
    Wall* _prototypeWall;
    Door* _prototypeDoor;
};

MazePrototypeFactory::MazePrototypeFactory (Maze* m, Wall* w, Room* r,
      Door* d) {
    _prototypeMaze = m;
    _prototypeWall = w;
    _prototypeRoom = r;
    _prototypeDoor = d;
}

Maze MazePrototypeFactory::MakeMaze () {
    return this->_prototypeMaze;
}
```

Listing 2: Prototype Design Patter in Action

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;

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

Listing 3: Singelton alternative in code

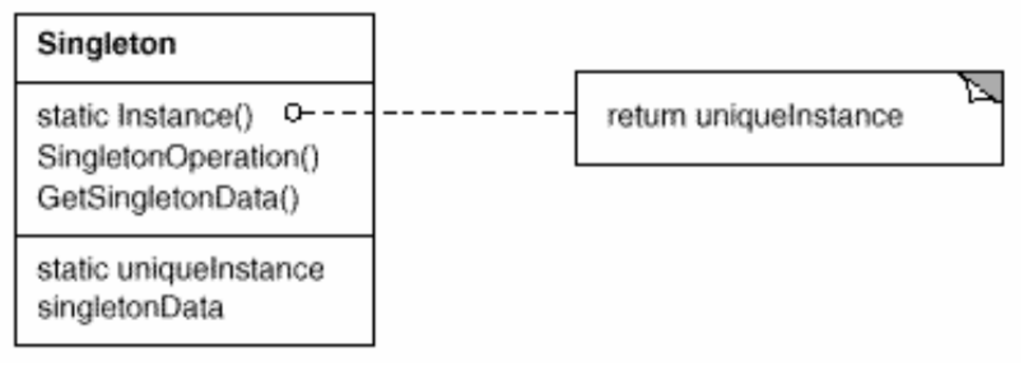


Figure 6: Singelton Class Diagram

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 (7). The object adapter inherits only from the target and holds the adaptee as an instance variable (8).

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

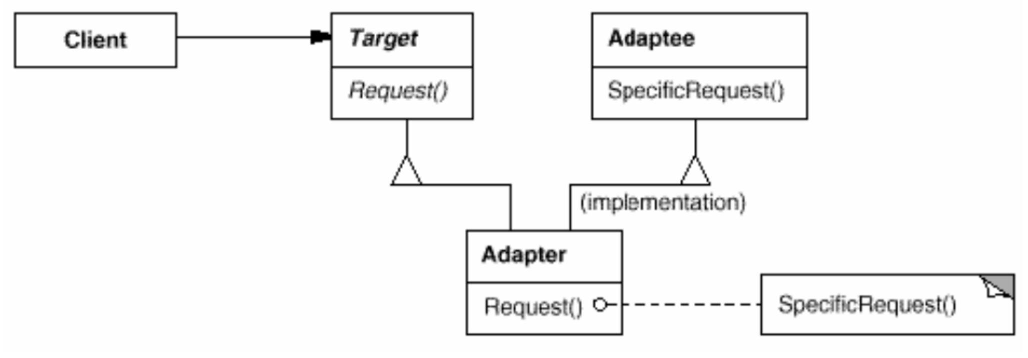


Figure 7: Class Adapter Class Diagram

2.2.2 Bridge

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

1. 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).

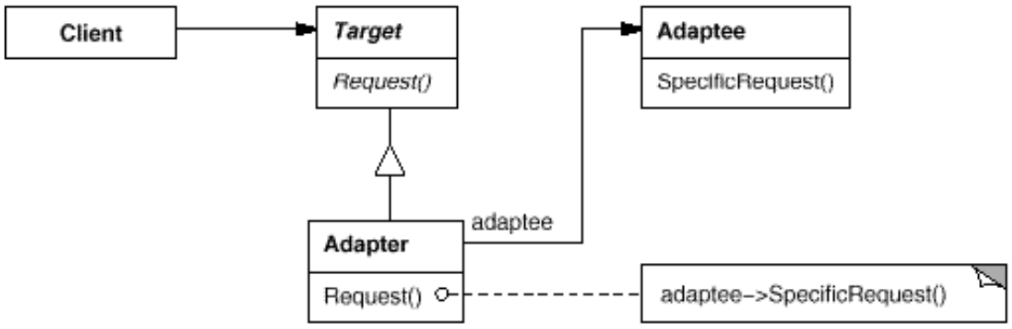


Figure 8: Object Adapter Class Diagram

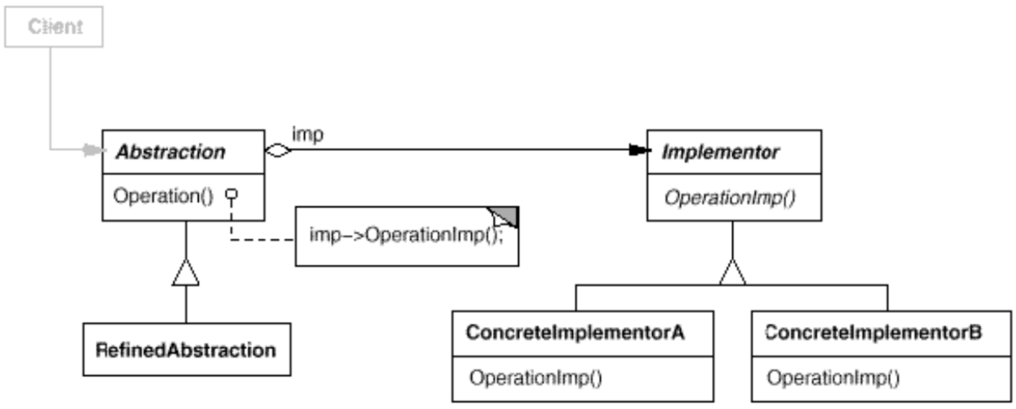


Figure 9: Bridge Class Diagram

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.

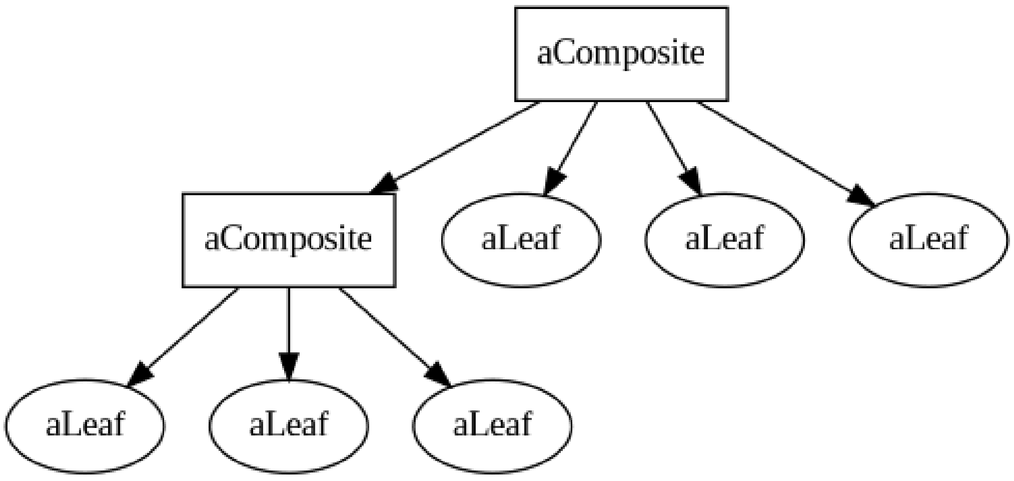


Figure 10: A Composite Structure

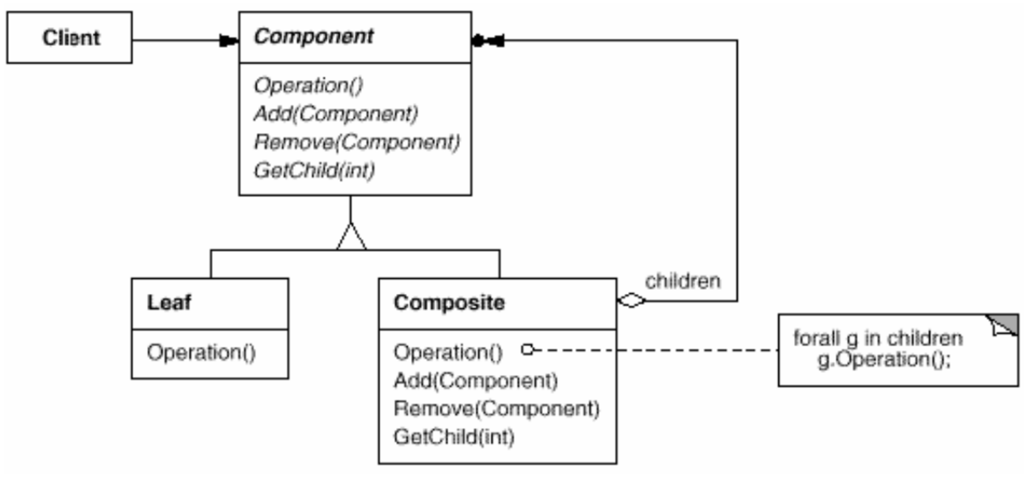


Figure 11: Composite Diagram

1. 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.

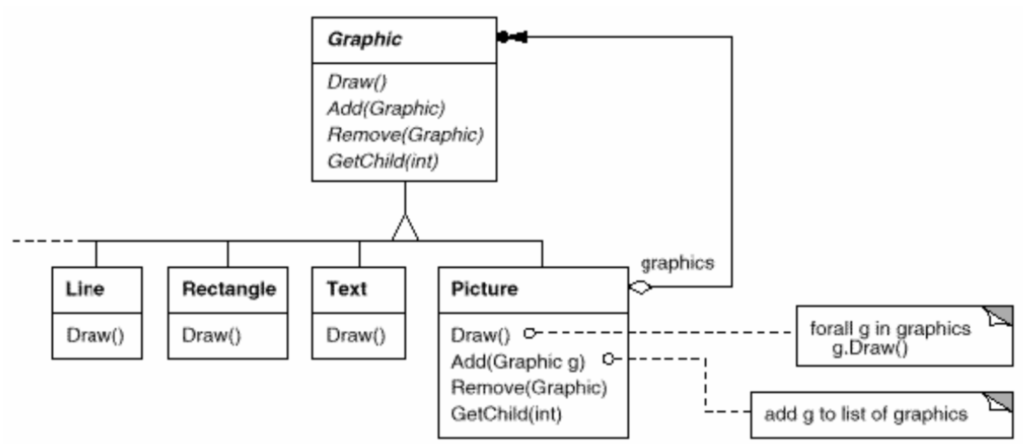


Figure 12: Example of Composite

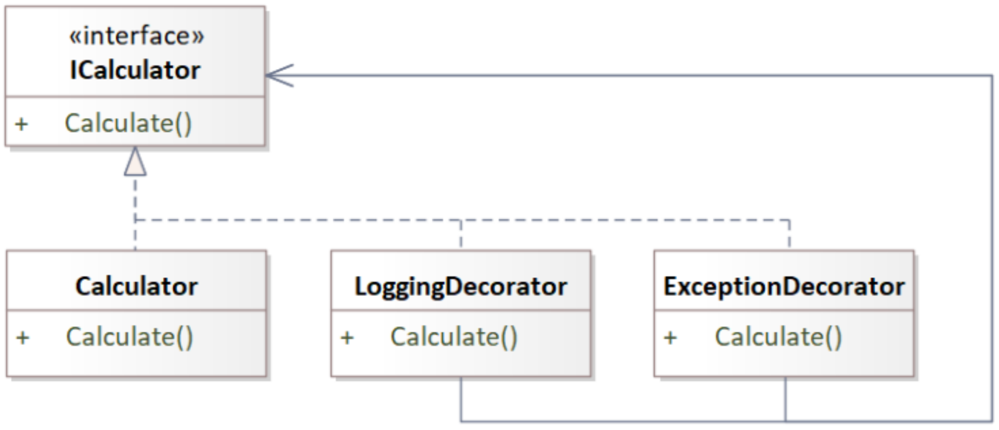


Figure 13: Decrater Diagram

2.2.5 Facade

The Facade Design Pattern is used to provide a simple interface to set of interfaces (subsystem). For example the Compiler class provide an easy to use interface for the whole compiler subsystem (figure 15).

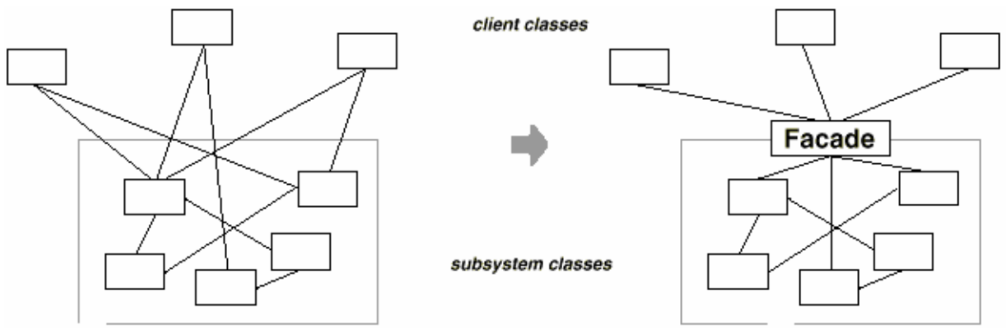


Figure 14: Facade Use Case

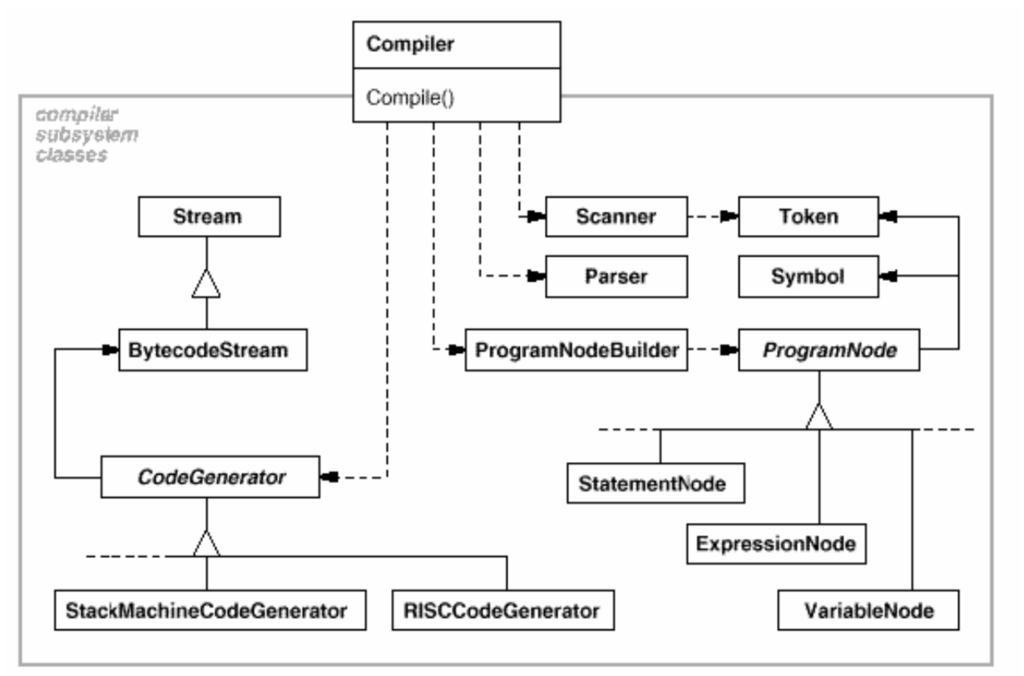


Figure 15: Facade Example

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 indecent 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.

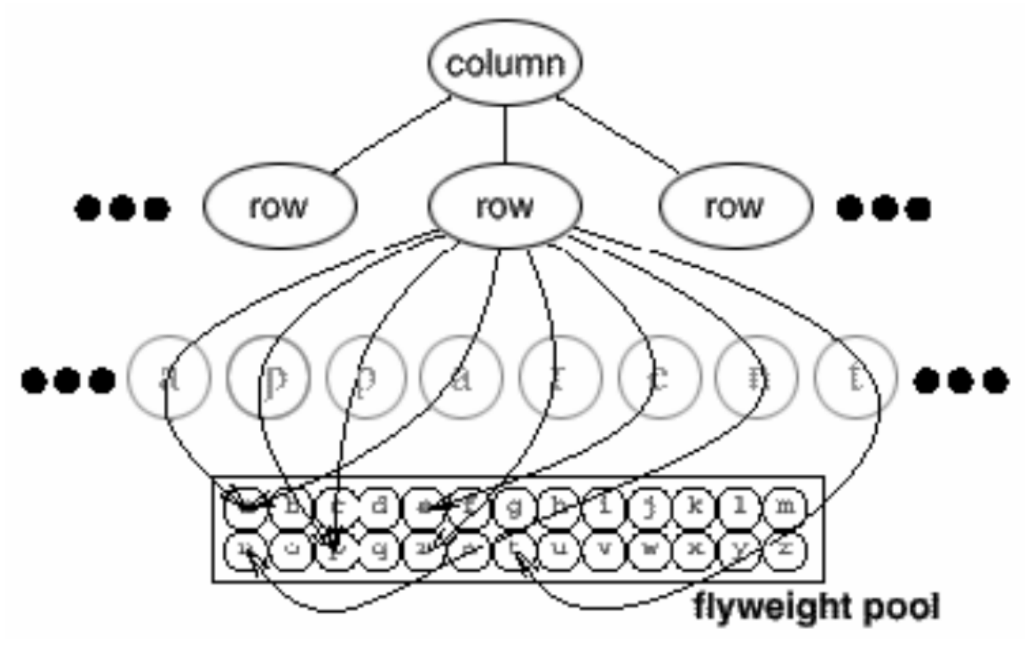


Figure 16: Flyweight Example

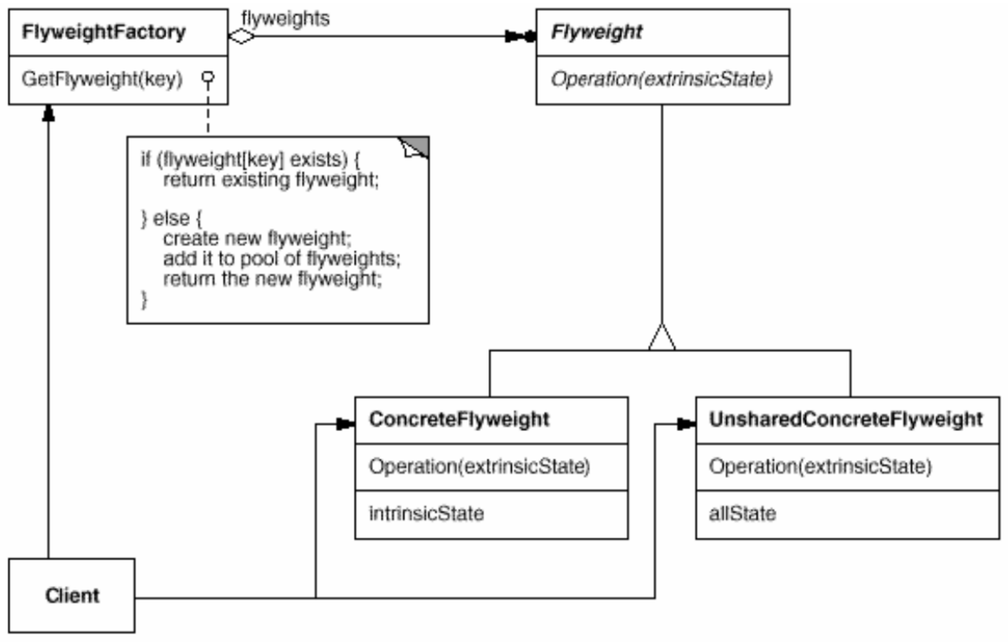

FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

Figure 17: 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 availabel you have to provide a loading screen. This can be easely done using the ProxyPattern.

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

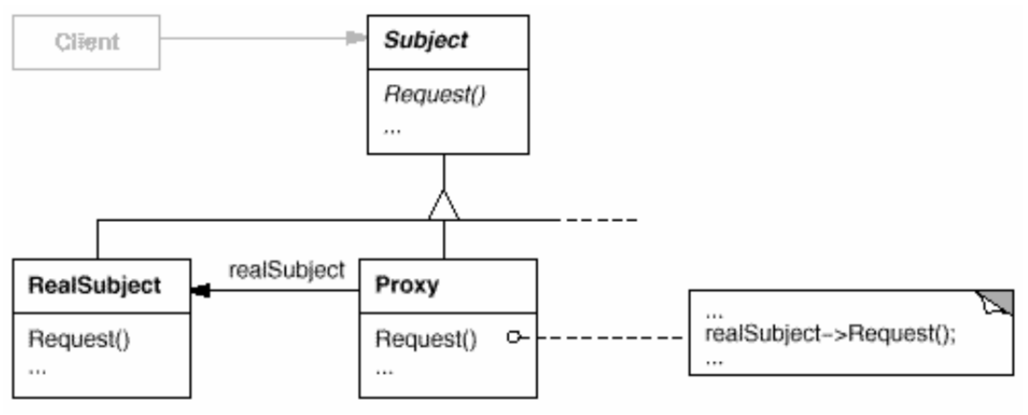


Figure 18: Proxy Class Diagram

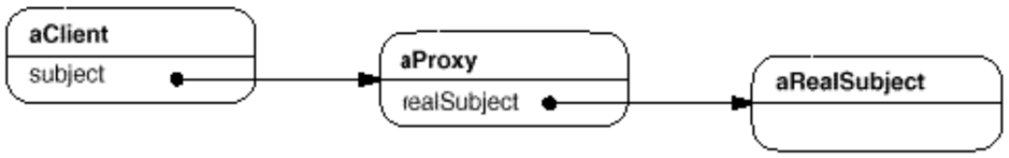


Figure 19: Object Diagram for proxy

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

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.

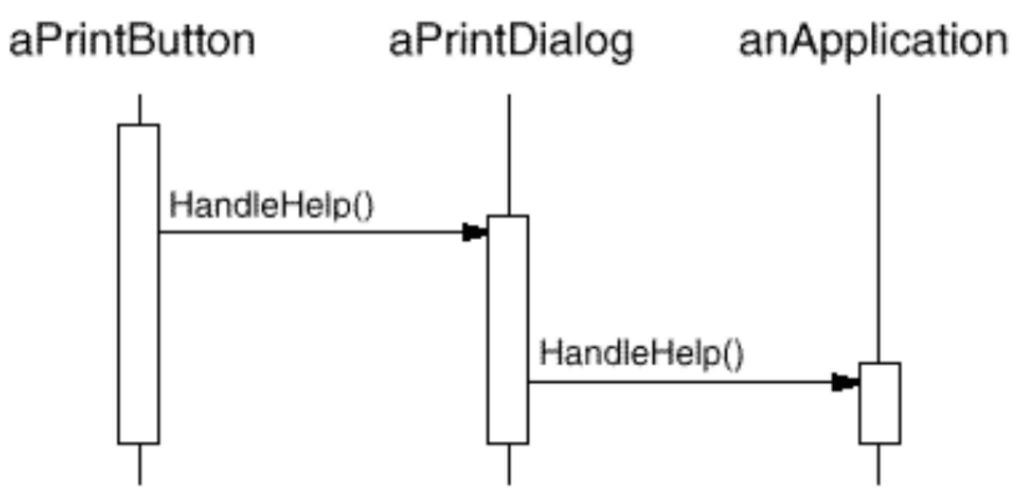


Figure 20: The Sequence of a Chain of Responsibility

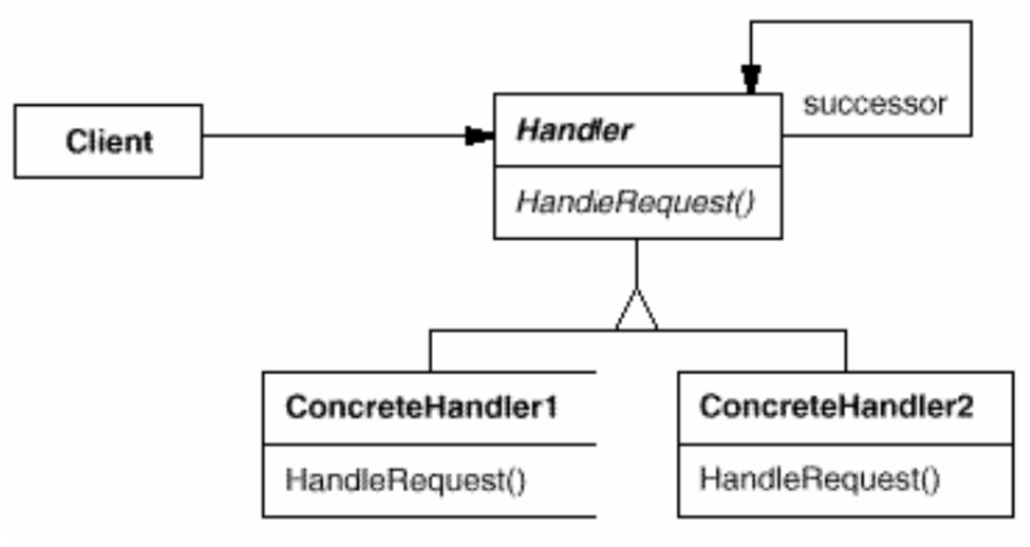


Figure 21: Chain of Responsibility Class Diagram

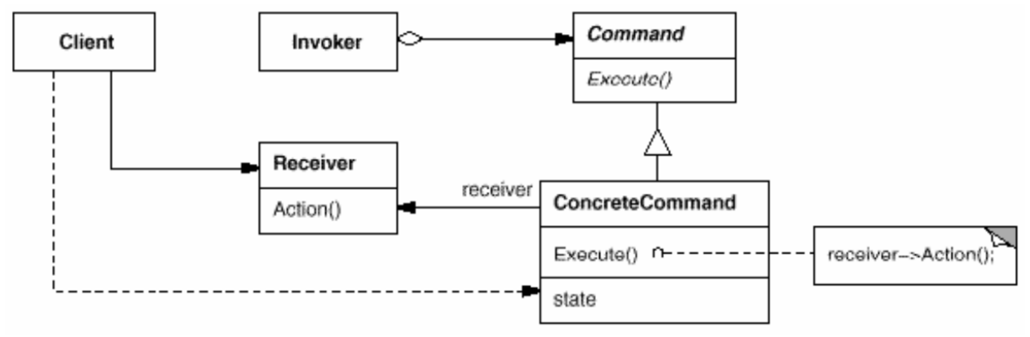


Figure 22: Command Class Diagram

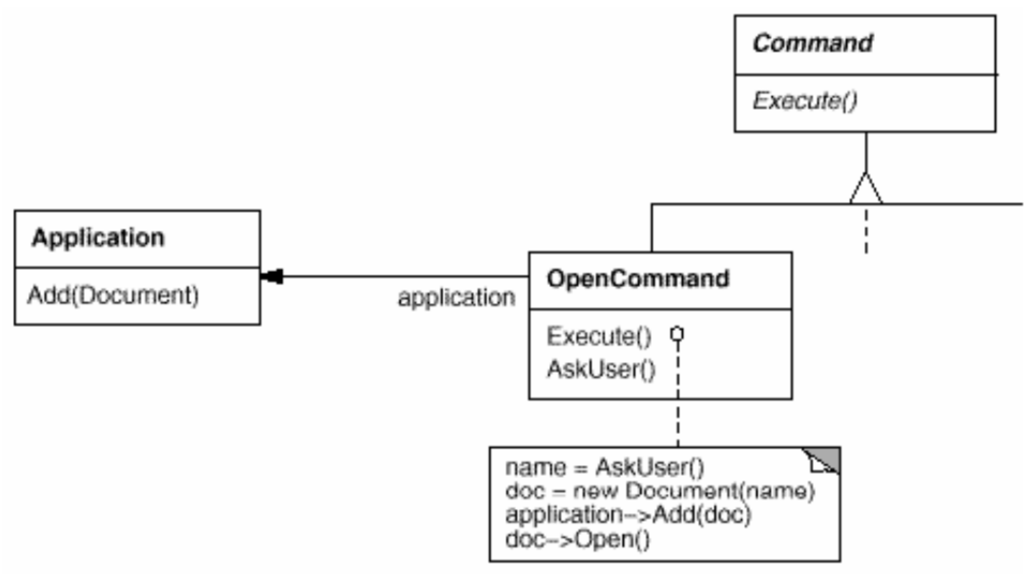


Figure 23: Example of Command Pattern

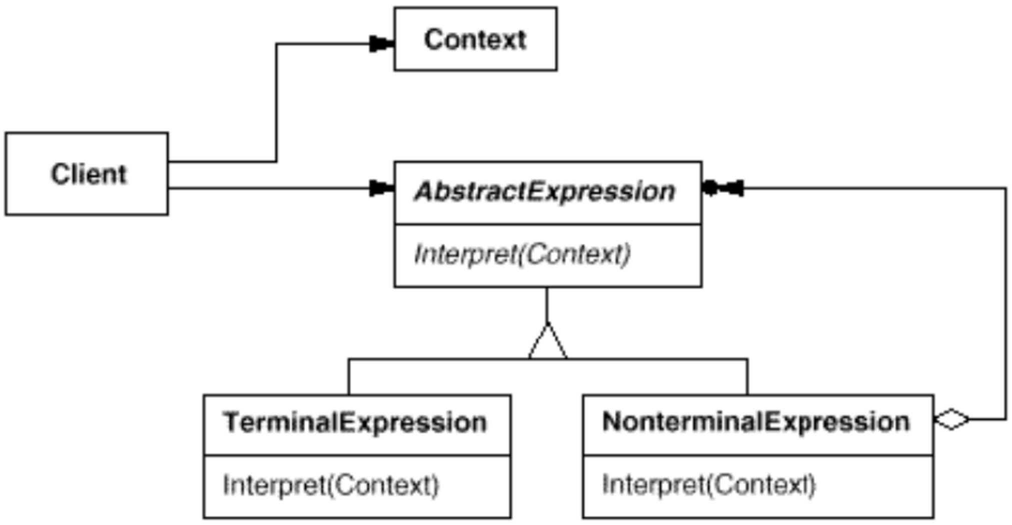


Figure 24: Interpreter Class Diagram

1. Example

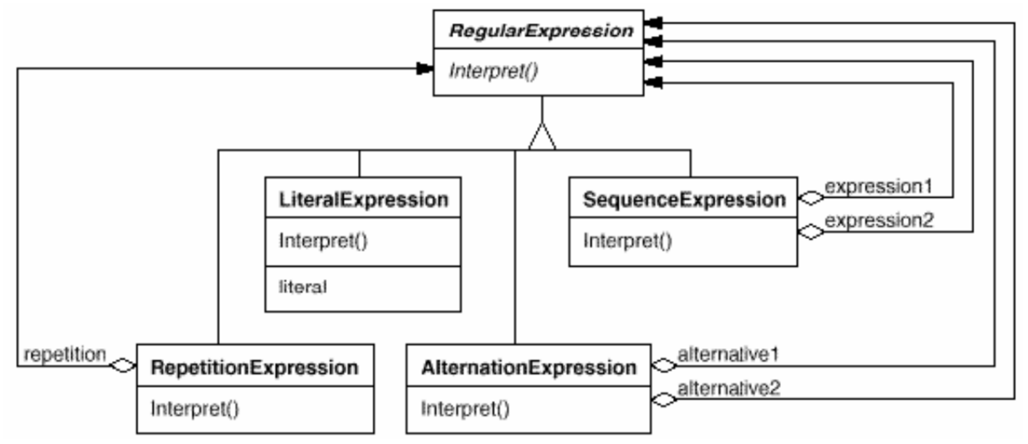


Figure 25: Interpreter Class Diagram for RegEx

2. 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 27). The benefit of iterator is that it hides the implementation details of the data structure.

```
let my_vec = vec![1, 10];
let iter = my_vec.iter();
let _ = iter.next();
let _ = iter.next();
// ...
```

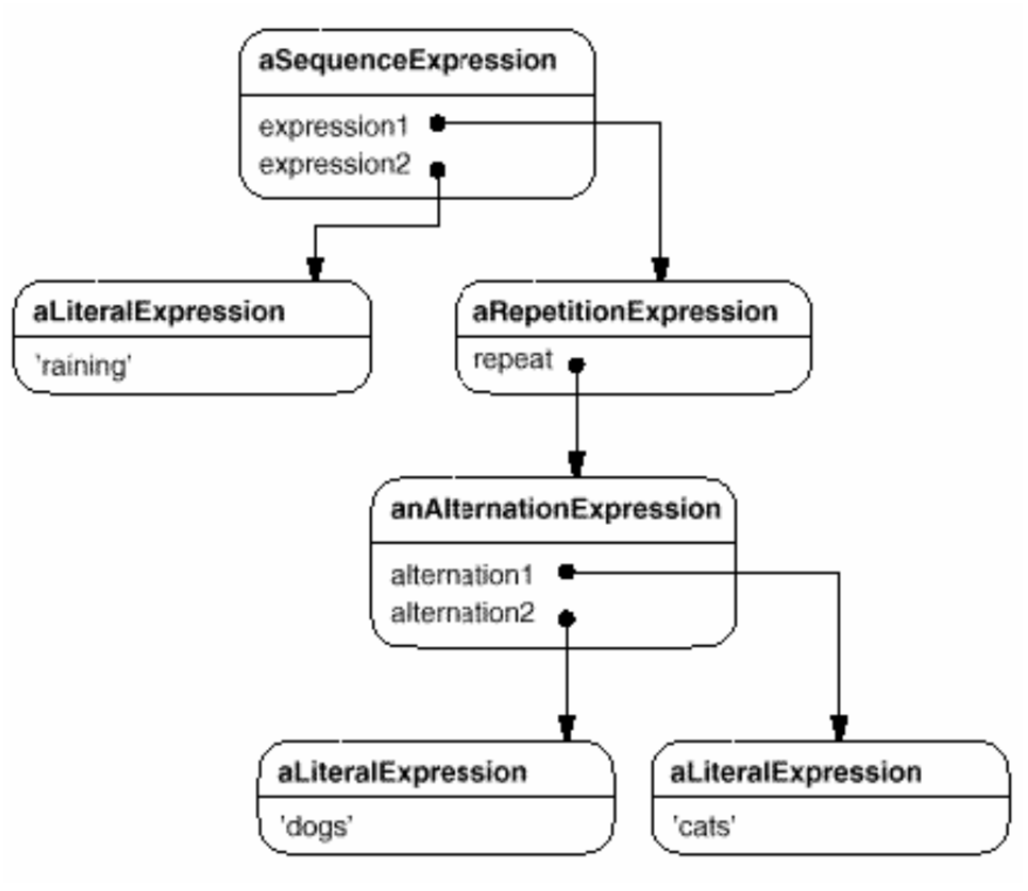


Figure 26: AST for RegEx ”raining & (dogs | cats) *”

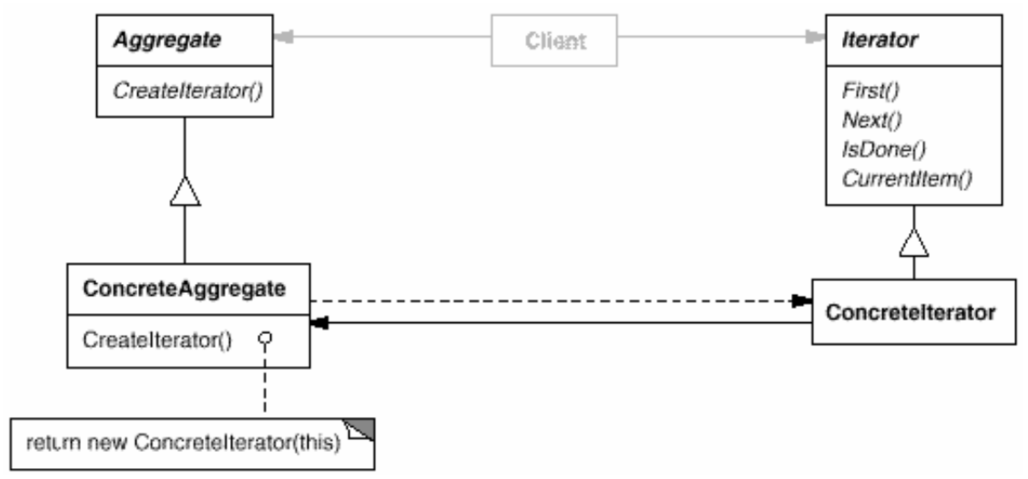


Figure 27: Iterator Class Diagram