Design Patterns - Summary

Semester: Spring 2022

Version: 0.1.0
Date: 2022-05-22 10:09

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

1.3 The Most common causes of redesign

A list of the most common causes of redesign and how you could solve the problem.

1. Creating an object by specifying a class explicitly.
   Try to avoid creating objects explicitly otherwise you are bound to this decision.
   Design Pattern - Abstract Factory, Design Pattern - Factory Method, Design Pattern - Prototype
2. Dependence on specific operations
   Try to avoid to hard code how you want to satisfy a request. Example: Do not hard code how a specific button should perform.
   Design Pattern - Chain of Responsibility, Design Pattern - Command

3. Dependence on hardware and software platform
   Try to avoid implementing against a specific HW or API.
   Example: Linux & Windows does not have the same Windows System. Provide an abstraction in which the underlying system is not relevant.
   Design Pattern - Abstract Factory, Design Pattern - Bridge
4. Dependence on object representations or implementations
   You should never bother how the inside of class works while using it. Otherwise, you might need to change things when the class is changing its inner live.
   Design Pattern - Builder, Design Pattern - Iterator, Design Pattern - Strategy, Design Pattern - Template Method, Design Pattern - Visitor
5. Algorithmic dependencies
   Do not hard code the algorithm. Try to encapsulate the algorithm in a class. Then you can replace it every time you want.
   Design Pattern - Builder, Design Pattern - Iterator, Design Pattern - Strategy, Design Pattern - Template Method, Design Pattern - Visitor
6. Tight coupling
   Try to avoid tight coupling. Otherwise, it is hard to reuse the class.
   Design Pattern - Abstract Factory, Design Pattern - Bridge, Design Pattern - Chain of Responsibility, Design Pattern - Command, Design Pattern - Facade, Design Pattern - Mediator, Design Pattern - Observer
7. Extending functionality by subclassing
   This is very difficult to made it right. Instead, use object composition to extend functionality.
   Design Pattern - Bridge, Design Pattern - Chain of Responsibility, Design Pattern - Composite, Design Pattern - Decorator, Design Pattern - Observer, Design Pattern - Strategy
8. Inability to alter classes conveniently
   Sometimes you can not modify a class (for example closed source library). The following Design Patterns can help to work around this problem.
   Design Pattern - Adapter, Design Pattern - Decorator, Design Pattern - Visitor

2 Design Patterns

2.1 Creational

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 (Design Pattern - 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.

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.

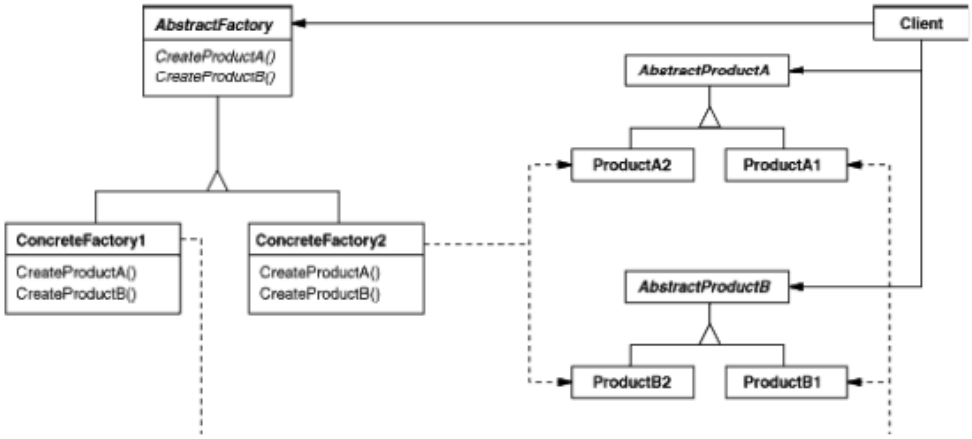

Figure 2: Abstract Factory Class Diagram

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).

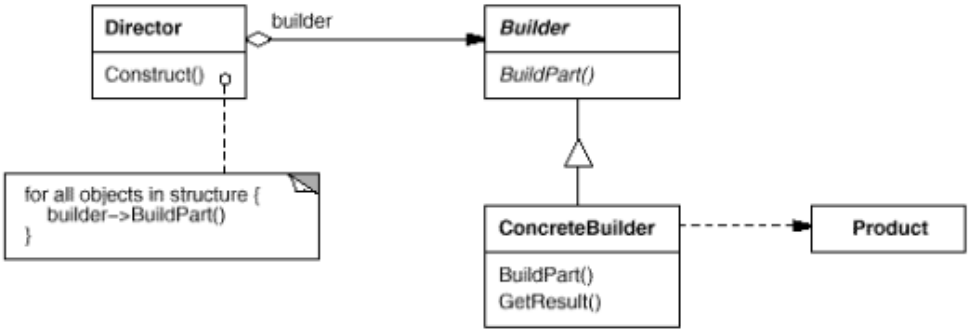

Figure 3: Builder Class Diagram

```
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.

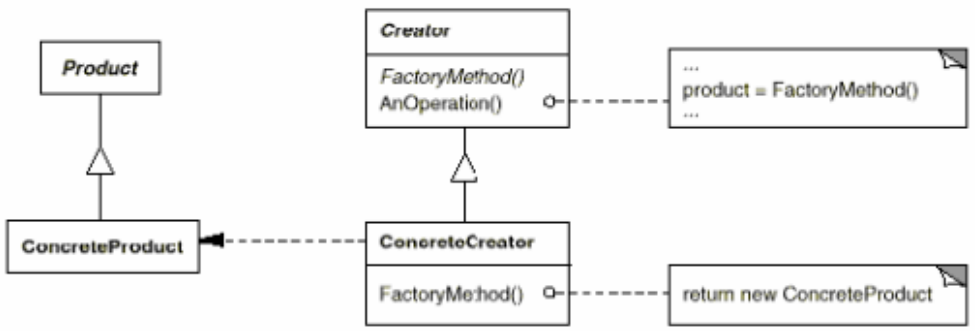

Figure 4: Factory Method Class Diagram

2.1.4 Prototype

The prototype pattern is something similar like the factory method (Design Pattern - 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;
  Wall* _prototypeWall;
  Door* _prototypeDoor;
};

MazePrototypeFactory::MazePrototypeFactory (Maze* m, Wall* w, Room* r,
    Door* d) {
```

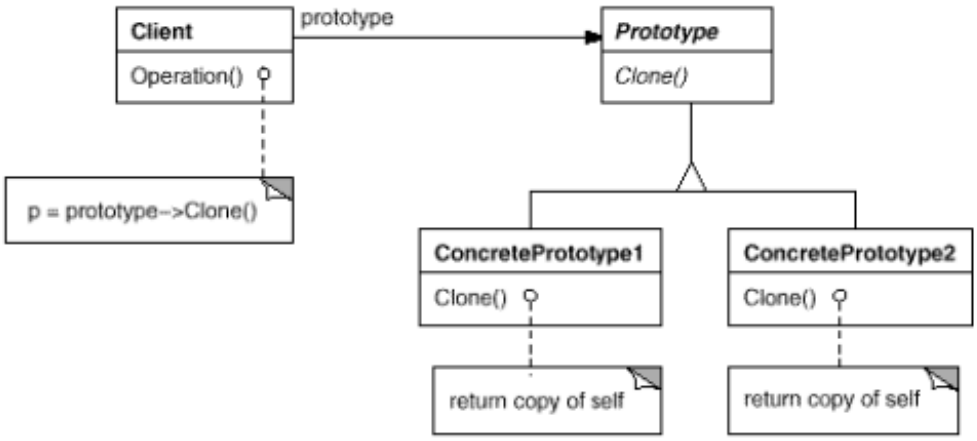

Figure 5: Prototype Class Diagram

```
    _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 (Design Pattern - 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

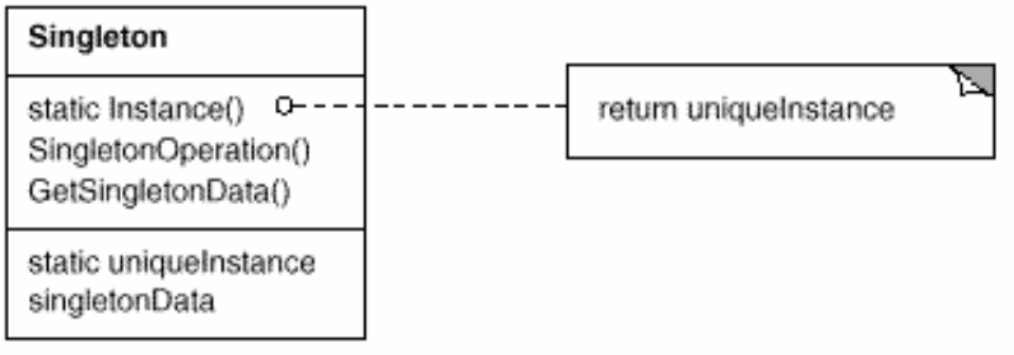

Figure 6: Singelton Class Diagram

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.

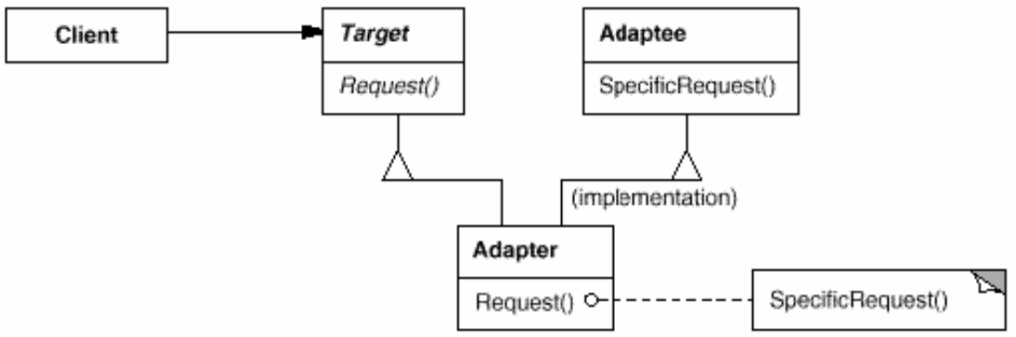

Figure 7: Class Adapter Class Diagram

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).

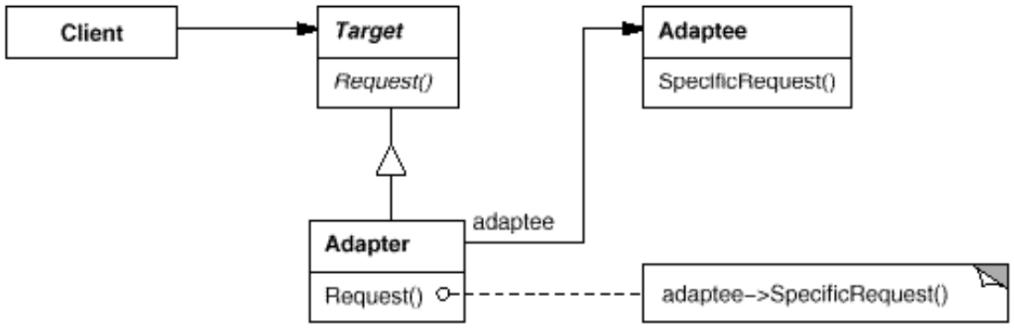

Figure 8: Object Adapter Class Diagram

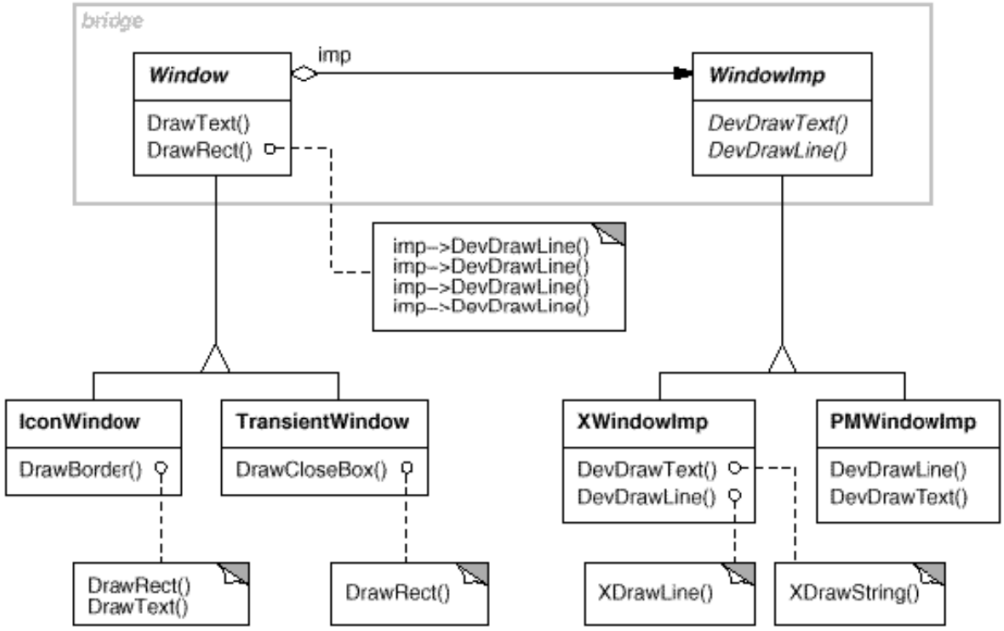

Figure 9: Bridge Design Pattern Example

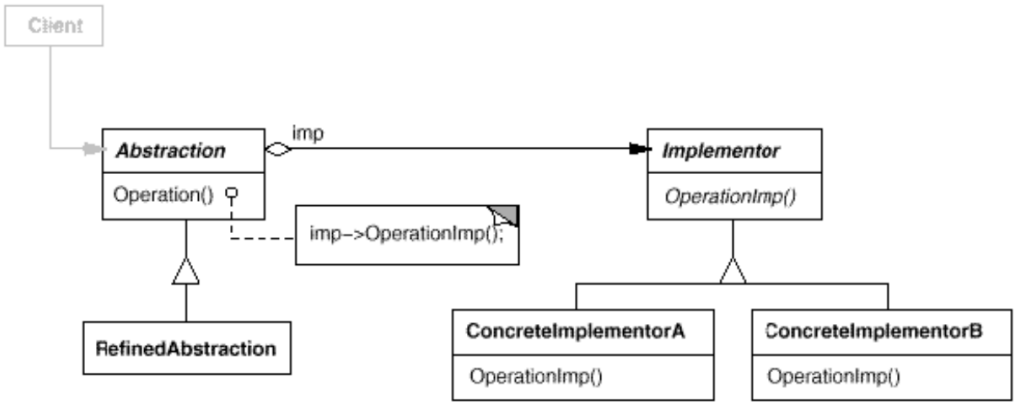

Figure 10: Bridge Class Diagram

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.

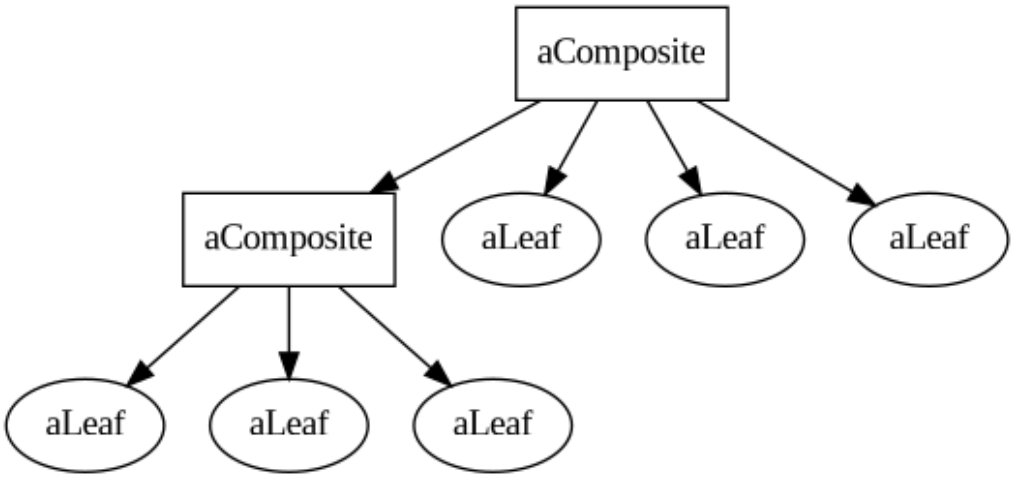

Figure 11: A Composite Structure

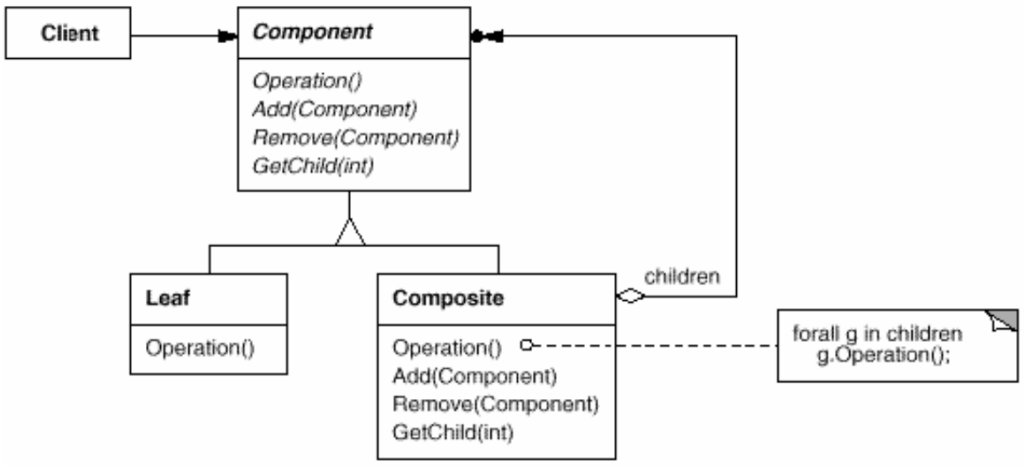

Figure 12: Composite Diagram

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.

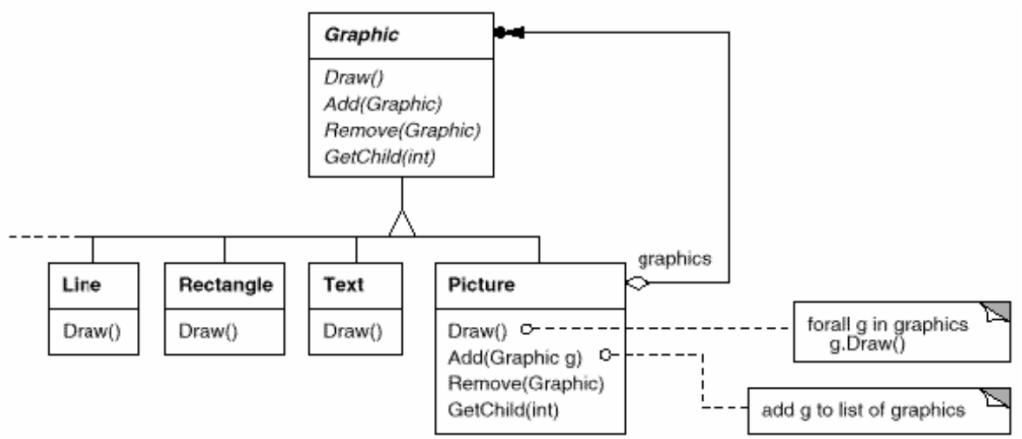

Figure 13: Example of Composite

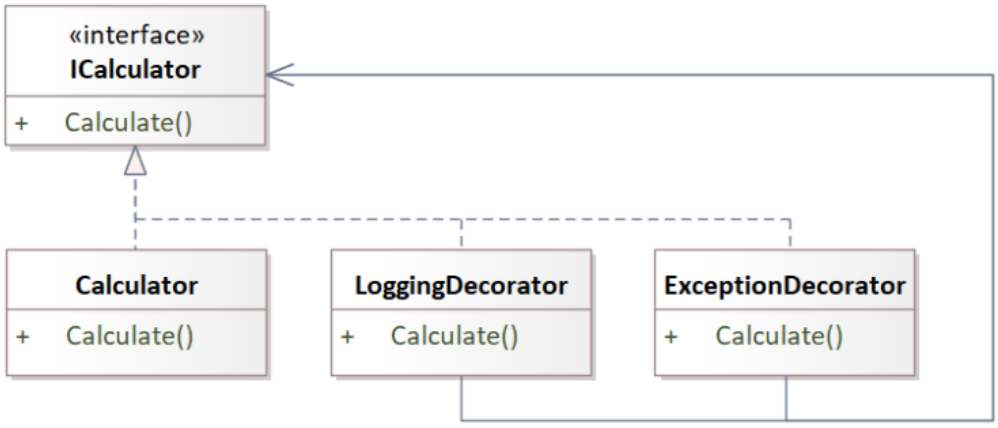

Figure 14: Decrater Diagram