01/03/2017

Report Tutorial Course 2

Algorithmics and Advanced Programming

WU Zheng

Report Tutorial Course 2

Algorithmics and Advanced Programming

2. Recognizing design patterns

2.1. Recognizing a design from code

Here I extended the example by creating another MazeGame subclass, EnchantedMazeGame:

```java
package com.darwindev.mazegame;

import com.darwindev.mazegame.elements.*;

class EnchantedMazeGame extends MazeGame {
    public Room makeRoom(int n) {
        return new EnchantedRoom(n);
    }

    public Wall makeWall() {
        return new SecretPassageWall();
    }

    public Door makeDoor(Room r1, Room r2) {
        return new DoorWithSpell(r1, r2);
    }
}
```

Then I implemented the Factory Pattern by creating the MazeGameFactory single instance:

```java
package com.darwindev.mazegame;

public class MazeGameFactory extends Object {
    public MazeGame create(String type) throws Exception {
        switch (type) {
            case "BombedMazeGame":
                return new BombedMazeGame();
            case "EnchantedMazeGame":
                return new EnchantedMazeGame();
            default:
                throw new Exception("MazeGame " + type + " is unknown.");
        }
    }

    public static void main(String[] args) {
        MazeGameFactory mazeGameFactory = new MazeGameFactory();
        try {
            MazeGame mazeGame = mazeGameFactory.create("BombedMazeGame");
            mazeGame.makeRoom(1);
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println(mazeGameFactory);
    }
}
```

The UML graph of this maze game is shown below.

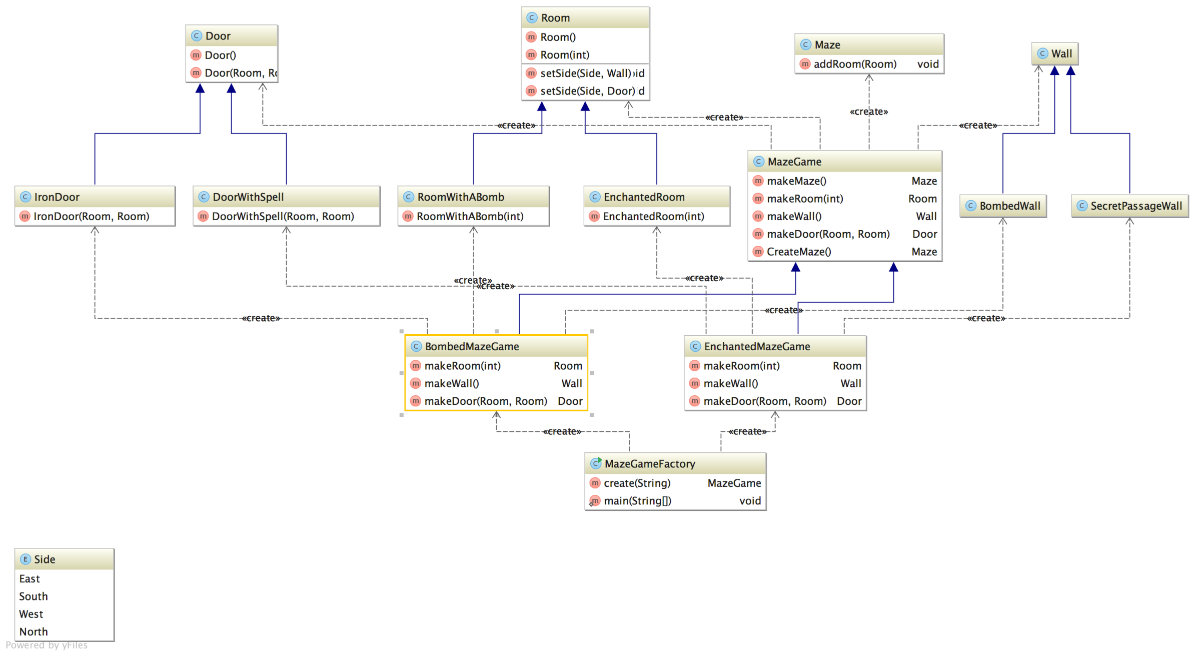

Figure 2.1 – UML Graph of MazeGame

2.2. Recognizing from a diagram

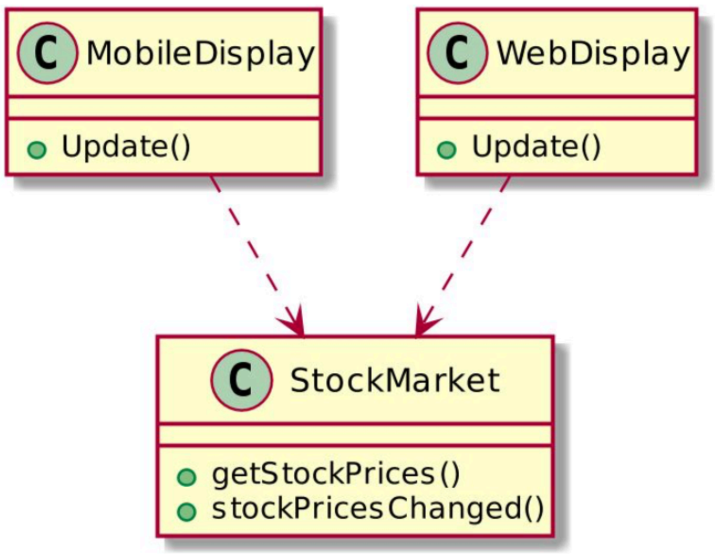

Figure 2.2 – Observer Pattern Example

The design above uses Observer Pattern. MobileDisplay and WebDisplay are both subscribed to the StockMarket as observers. Each time when stock prices changed, the stockPricesChanged() method will be called, which notifies all observers and give its new state to them.

2.3. Differentiate 2 similars Patterns

Pattern A is a Strategy Pattern. The Context is the subject in Strategy Pattern, which is the interface to outside. While <<*interface*>> is the strategy, which is the common interface for different algorithms.

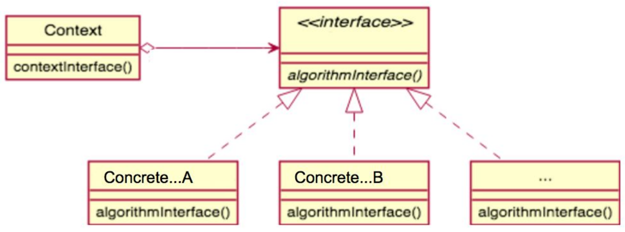

Figure 2.3 – Strategy Pattern

Pattern B is a State Pattern. It lets an object show other methods after a change of internal state.

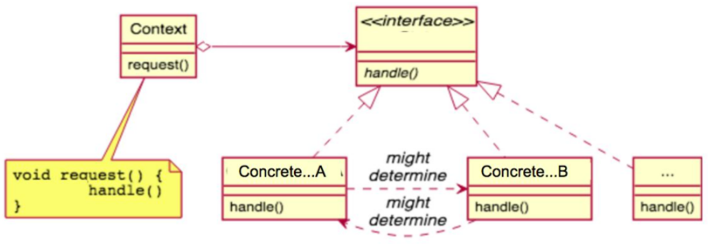

Figure 2.4 – State Pattern

The Strategy pattern is about having a different implementation that accomplishes the same thing, so that one implementation can be replaced with the other as the strategy requires. For example, you might have different sorting algorithms in a strategy pattern. The callers to the object does not change based on which strategy is being employed, but regardless of strategy the goal is the same (sort the collection).

The State pattern is about doing different things based on the state, while leaving the caller relieved from the burden of accommodating every possible state. So for example you might have a *getStatus()* method that will return different statuses based on the state of the object, but the caller of the method doesn't have to be coded differently to account for each potential state.

Example

Here is an example on StackOverflow. Consider an IVR (Interactive Voice Response) system handling customer calls. You may want to program it to handle customers on:

- Work days
- Holidays

To handle this situation you can use a State Pattern.

- Holiday: IVR simply responds saying that 'Calls can be taken only on working days between 9am to 5pm'.
- Work days: it responds by connecting the customer to a customer care executive.

This process of connecting a customer to a support executive can itself be implemented using a Strategy Pattern where the executives are picked based on either of:

- Round Robin
- Least Recently Used
- Other priority based algorithms

In conclusion, the strategy pattern decides on '**how**' to perform some action and state pattern decides on '**when**' to perform them.

3. Embrace Peer Review and Feedbacks

I have created a Github Gist at: https://gist.github.com/Lessica/7ad227c6ac288c8d913c21b7f0f4188a, and I commented on it by myself.

The original design did not follow any design pattern. It was a combination of functions handling different tasks. Which causes confusion and increases the complexity. To improve the design, I decided to apply Strategy Pattern to algorithm switching, and Factory Pattern to graph generating.

Here is my review list:

- https://github.com/jxy0119/TP1/commit/46e2d9e8ef1a1c5cec5c8e54c5c2fd61fc6145d1
- https://github.com/yerimJu/AdvancedAlgorithm_TP1/commit/59ad12ee367e1f52925fc99110ed3e82c74f7e9e
- https://github.com/Alexandrecorre/Tp_Algo_Progra/commit/11793c09f49c64ec3ffa5b5a7a4da48896fcabed
- https://github.com/haiyinyin/time-complexity/commit/c262b2790f6956a0d34f88680f6a661345084b49
- https://github.com/aurelienshz/ii.2415-algo-prog/commit/afb5d40f82469ad2a22475de11ae4f879cc34ede
- ...

4. Make design generics

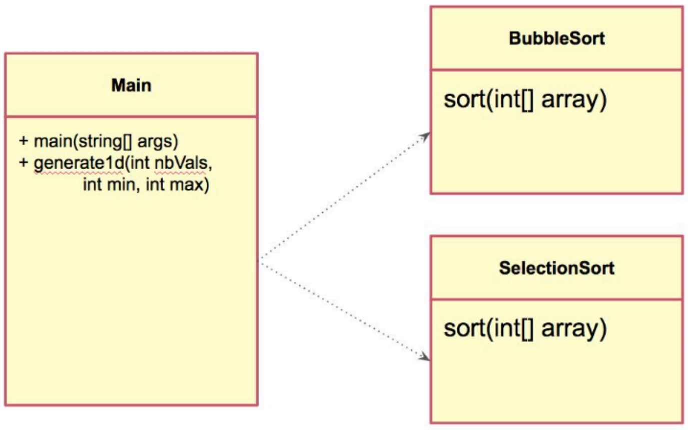

Figure 4.1 – Class diagram of the first class

This design is not generic, because when we want to add another benchmark, we have to modify *main()* and create many duplicated codes. Also, the code of generating graphs are written in the *main()* method, which seems ugly. It is not open for extension too. Sorting methods are hard coded, we cannot add other sorting methods dynamically. A better model I have designed is shown in next chapter.

5. Refactoring

We would like to support the creation of multiple sorting algorithms, while the time measurements of the algorithms have the same task – calculating and showing the time usages when the array sizes varies.

Different sorting algorithms have different implementations, but they have no direct connection to the time measuring task. We should hide their implementations and create a generic interface for the caller. That is why Strategy Pattern can be applied to the algorithm switching. Considering there are many classes implemented for these sorting algorithms, we should use Factory Pattern to avoid specifying their classes.

So I created AlgoFactory (Algorithm Factory) to create subclasses of algorithm being tested and ApplicationFactory to create TimeMeasurement or TimeComparison application.

I modified the TP1 code to follow the diagram. I created two Enums for the name list of Sorting Algorithms and Applications.

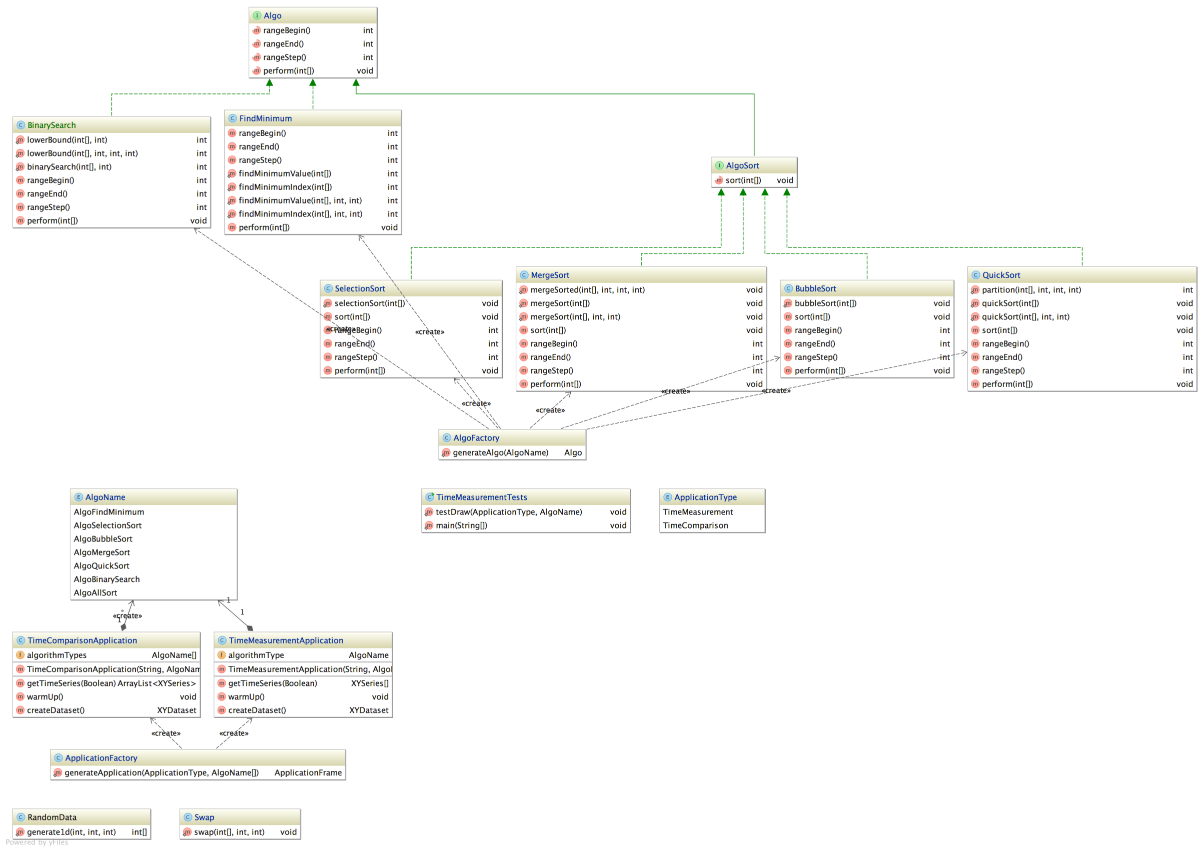

Figure 5.1 – Refactoring My Design

6. Adapter (Bonus)

The implement of StackAdapter and all source code in previous chapters can be found here: https://github.com/Lessica/Advanced-Algorithm