RMIT University Vietnam

ISYS2102 - Software Engineering 2

Design Patterns Report

Sprint 1

Lecturer: Kevin Jackson

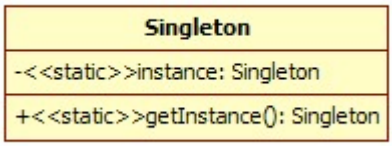
BinMaster team:
Thai Ly Cuong – s3255125
Dang Nguyen Hoang Gia – s3269900
Nguyen Thanh Huy – s3255150

Singleton Design Pattern

1. Definition: The Singleton Pattern ensures a class has only one instance and provides global point of access to it.

2. Description:
   By making the constructor private, the Singleton Pattern ensures that instance of one class can only be made using the static method getInstance(). By modify the getIntance() method, it ensures that only one instance of this class is ever created.

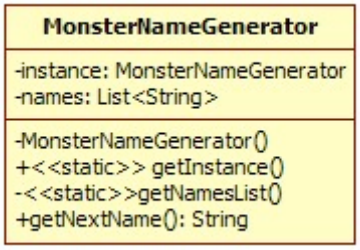
3. Class Diagram:



4. Advantages and Disadvantages:
   a. Advantages:
      Good for provide global access point to a shared resource (not data)
      When there is the need of only one instance of a class, limit the class to create more instance can help the application consumes less resources.
      There are special classes that if two or more instances of them is made, it will break the structure of the program (for example: Thread, Pool).

   b. Disadvantages:
      When using with thread, the Singleton Pattern cannot make sure that the class can only create one instance anymore.
      It is hard to write unit test because Singleton class is a global state.

5. Example:
   Example of using Singleton that will make the program better: this class is taken from the Assignment Dorfle



```
public class MonsterNameGenerator {
    private static List<String> names;
    private static MonsterNameGenerator instance;

    private MonsterNameGenerator(){

    }

    public static MonsterNameGenerator getInstance(){
        if(instance==null){
            instance=new MonsterNameGenerator();
        }
        return instance;
    }

    public String getNextName(String difficulty){
        if(!difficulty.equals(Constant.HARD)){
            //in sprint 2, it will return the actual monster type
            return "Monster";
        }
        if(names==null){
            getNamesList();
        }
        Random r=new Random();
        return names.get(r.nextInt(names.size()))+" the Monster";
    }

    private static void getNamesList(){
        try {
            names=new ArrayList<String>();
            Scanner s = new Scanner(new File(Constant.RES_DIR + "name.csv"));
            while(s.hasNextLine()){
                String line=s.nextLine();
                String[] nameArray=line.split(",");
                names.addAll(Arrays.asList(nameArray));
            }

        } catch (FileNotFoundException ex) {
            Logger.getLogger(MonsterNameGenerator.class.getName()).log(Level.SEVERE, null,
ex);
        }
    }
}
```

In the above class, the Singleton Pattern is applied to it so that in the whole program, there is only one instance of MonsterNameGenerator, thus making the program to read the file containing the list of random name only one. In addition, inside the memory, there is only one List object that holds the whole list of random name. If the Singleton Pattern is not used here, some different developer may create a new MonsterNameGenerator's instance just to get the name of the monster, thus making the program to unnecessarily read the file the second time, and create two List object that holds the random names. In this example, the Singleton Pattern helps reducing the use of resources in the program.

The disadvantages of Singleton Pattern when using in Thread is not presented here because even if two instance of MonsterNameGenerator is made, the program will only consume more resources. There will be no fatal error for the program.

6. Alternative pattern and Justification:

Sometime the Proxy Pattern works better than Singleton Pattern.

As state before, the Singleton Pattern is best used for making a global access point to a shared resource. This shared resource is heavily consume system resource so Singleton Pattern ensures that it will be instantiate only when it is needed and this happens only once. However, sometimes this will not work if the resource is shared across ejb, which will hold different Singleton instance in each context. This time, Proxy server is better because it provides a way to access a shared resource only when it is needed and it will work no matter if the structure of the program is asynchronous or synchronous.

7. Alternative method:

Instead of using Singleton Pattern to make sure a class has only one instance, we can just make everything in that class static(in Java). For example

```
public class Resource{
        private static String[] someResources={“a”,”b”,”c”,”d”};

        private Resource(){
        }

        public String[] getSomeResources(){
                return someResources;
        }
}
```

The only bad thing about this is that the class Resource may not be used at all but the memory still store the someResources as variable in the class. However, it fixes the asynchronous problem.

To sum up, Singleton Pattern is used to ensure that a specific class will have only one instance. This pattern is good for creating a global access point to some heavy resources. However, using this pattern in an asynchronous program is dangerous and not recommended. In this case, another pattern or different method will work better.

Interpreter pattern

1. Problem:

Our programs sometimes need to define a "small language" with small set of rules. That language can be used to help users to enter input easier and friendlier. It can also help us to map a well-known domain into our program. Then, Interpreter pattern can help us to specify how to interpret sentences or statements in the "small language". For example, if we want to allow user to move a game character by typing in: "player move left move up", we can use Interpreter pattern to translate that sentence into objects or methods that our program can use.

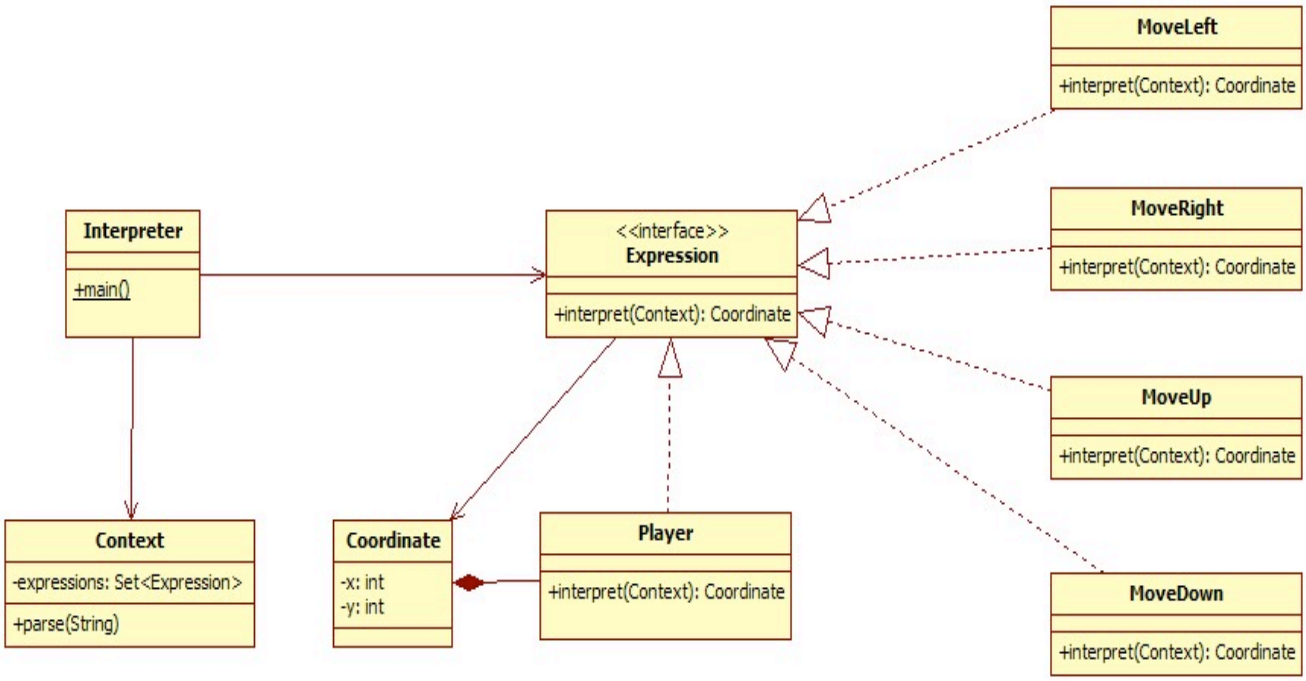
2. Description:

As a simple language, it has simple grammar. In Interpreter pattern, each element in the language is called as an expression and each expression has a method called "interpret" to execute the meaning of the expression. For example, in MoveLeft expression, the interpret method will get the player and change his coordinate so that he will move left.

Interpreter pattern also introduces a concept called Context which basically contain sentences of the new language which will be translated and run. That context can be created from input of users.

As a result, the client code will have association with Context and the "Expression" interface that is implemented by every expression in our new language. In order to provide a clear idea about this pattern, a class diagram was created.

3. Diagram:



4. Advantages and Disadvantages:

a. Advantage:

This pattern helps us create our new language easily. When you want to have a new expression, you need to add a class that implements Expression interface and may change the Context to use it.

b. Disadvantages:

This pattern is good for a simple language. When your language becomes complicated with so many rules, you will end up with a lot of expressions and change the Context many times to make it work.

To implement this pattern, two interfaces and other expression classes will be added. So, the program will be more complicated.

It will take sometimes for users and programmers who write client code to learn your new language. When the language is changed, you either need to change your code or notify your users.

5. Example:

On my Dungeon game, I want to allow user to control their characters by entering command lines, such as "player at 3 3 move left". Then, the character will move left and at position (2, 3). Even further, they can play the game totally by command lines. So, I need a controller that can take user input, interpret it and call the model to change accordingly.

In this scenario, the Interpreter pattern fits perfectly. When user type "player at 3 3 move left", a context will be created which contains the Player object and MoveLeft expression. In order to do that, the Context class has a method that can parse the input String into a collection of expressions.

After having a Context, client code need to get the player and trigger the interpretation. Then, interpret method inside each expressions will be called. The method will find the player who is contained in the context and change its position.

A project was created to demonstrate this example. It can be found here: https://github.com/rmitvn-se2/rmit-se2-2011B-BinMaster/tree/master/docs/Reports/Intepreter/Intepreter Example

6. Alternative pattern and Justification:

The main goal of Interpreter pattern is to interpret an input into a list of objects that can be consumed by our program. We have another alternative to get the same result.

In the above example, we can use Builder pattern. It is used to create a player that contains a list of commands. That player will be returned to the client code. Then, the client code will simply call the player to execute all the predefined commands. So, Builder pattern can be used to achieve the same goal.

Which one is better? Well, I think it depends on your design. If your language has a simple list of rules and will not get more complex, Interpreter would be a good candidate. If you are more familiar with Builder pattern and want to simplify your program, a Builder would be enough.

7. Alternative method:

For a simple problem that Interpreter pattern tries to solve, a simple parser method can be used. Basically, that method will get a String parameter and perform accordingly to the input. If the input contains "player at 1 1", the parser will create a new player at that position. If

the input contains "move left", parser will set the coordinate of that player to a new position.

By doing like this, we can simplify our program, avoid adding additional interfaces and classes. However, you may need to split that parser method into smaller ones so that it will not become ugly.

Mediator Pattern

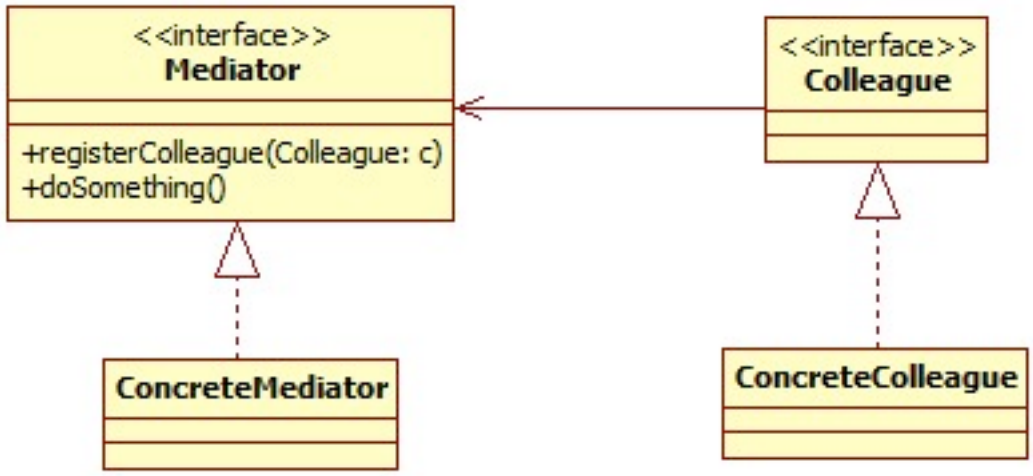
1. Problem:

   When colleague (peer) objects interact with each others, this will introduce high coupling between objects. The problem will become bigger when the complexity of the object increases. This will lead to hard to read and maintain because when you change something in one class, it will affect the others. Therefore, we should find a way to decouple those objects.

2. Description:

   Mediator will try to encapsulate the communication between objects into a Mediator object. Mediator object produces an abstraction layer among colleague objects so that any change in one object wont affect the others.

3. Diagram:



4. Advantages and Disadvantages
   a. Advantages:

      Decoupling objects supported by mediator.

      Reduce the complexity of calling each other within the peer objects.

      Simplify the maintenance process by centralizing control logic in mediator object.

b. Disadvantages:

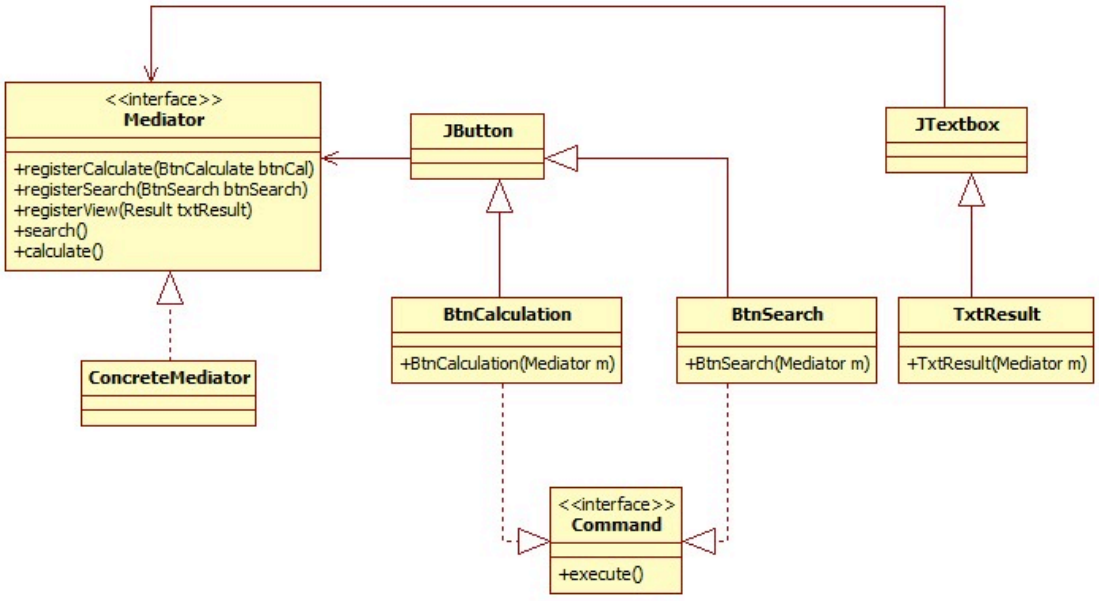
The mediator itself will be complex because it will contain many other objects and control sending requests among them.

The multi-direction communication makes the design more complex compare to Observer pattern. Observer introduces the concept of observer and subject, hence there is one way of sending request.

5. Example Scenario:

When developing GUI components for a desktop application, we will see the communication between components (JButton, JLabel) becomes very complicated. For example, in Management System at super market, users of this system need to have a lot of information and controls on the screen and these components need to be interact with each others to limit and reduce confusion for user. There is the case when an action listener is trigger, one button will call the label to edit its text and disable another button, or if there is no products then menu will be enabled and disable it when products are added into the system. At this point, you will see the high coupling between components, changing in one component could lead to others need to be edited. Furthermore, because the logic to control interaction is spread across components, maintenance would be extremely painful and wait of time. Therefore, mediator pattern will help you to centralize the interaction of components.

Mediator Pattern Illustration:



In the diagram, you can see, the interaction between of objects (btnCalculate and btnSearch) now moved to Mediator. You just need to register a component when it is constructed and call appropriate method in mediator. The mediator will control interactions among Buttons and Textbox.

Sample Code:

```
  Public class Main extends JFrame implements ActionListener{
        Public Main(){
          Mediator mediator=new Mediator();
          Add(new BtnCalculation(mediator));
          Add(new BtnSearch(mediator));
          Add(new TxtResult(mediator));
        }
        public void actionPerformed(ActionEvent e){
                Command cmd=(Command) e.getSource();
                cmd.execute();
        }
        public static void main(String[] args){
            new Main();
        }
   }
public class BtnCalculation extends JButton implements Command{
    public BtnCalculation(Mediator me){
            me.registerCalculate(this);
    }
     public void execute(){
            me.calculate();
    }
}
```

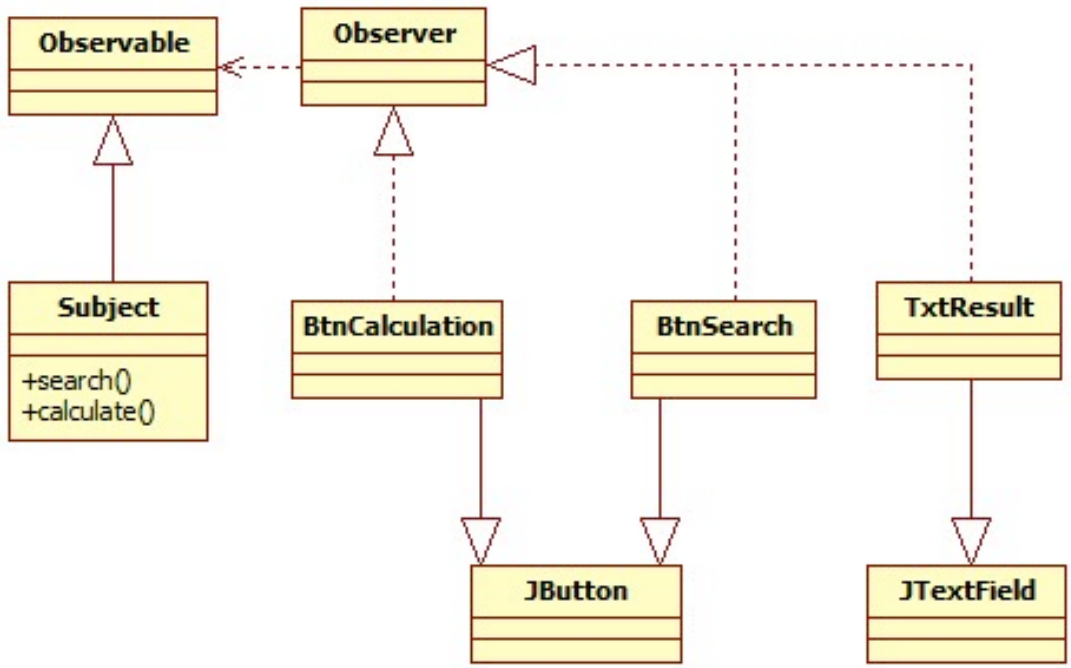
6. Alternative Pattern and Justification:

    - Façade Pattern:
      This pattern is quite similar to Mediator in term of centralizing the control logic of other objects and decoupling them. However, in some scenarios Façade pattern would outweigh the mediator. For example, when we have an external system communicate with a sub-system, we should use a façade instead of mediator to talk with sub-system to reduce the complexity. On the other hand, mediator just acts as the center point to interact with colleague objects. If we use the mediator object as an interface for external system, this would lead to mixing codes of internal and external system, and again, the object is not cohesive.

    - Observer Pattern:
      The pattern does the same thing as the mediator does. However, the observer pattern uses observer term for the object that is notified when another object state changes (the observable or subject). The interaction between observer and observable are in one direction whereas the mediator and peer objects have two references between them. In most of the case, using observer pattern is better because we usually need to be notified when state of an object changes. In addition, because we usually have a subject to notify for the observers to update, reusability is easier in observer pattern whereas in mediator you will have to modify new mediator so that it can register new object.

7. Alternative Methods to achieve the same goal

    We can use the observer pattern to implement above example by changing mediator to subject object and colleague objects are observers. New class diagram can be seen as the following:



You can see now the subject (like the mediator) is simplified by removing the register methods. So when a method in subject is called, it will notify all observers to pull new data and update themselves. This action will put the renderer UI code toward UI components e.g setText in TxtResult.

- Advantages:
    - Simplify the Subject object by removing the register methods and providing data source for observers
    - Separate data source from the UI renderer
    - Reusable the observable objects and observer objects
- Disadvantages:
    - Subject class cant extend another class