Advanced Java Programming first ICA

By: Kiril Anastasov
Student in Teesside University Computer Science
13, December 2012

Mediator

Advanced Java Programming
Design Patterns

Lecture Outline

- Introduction to the Mediator pattern
- Examples of the Mediator pattern
- Advantages and Disadvantages
- The relation of the Mediator pattern with the other patterns
- A summary

Mediator Pattern

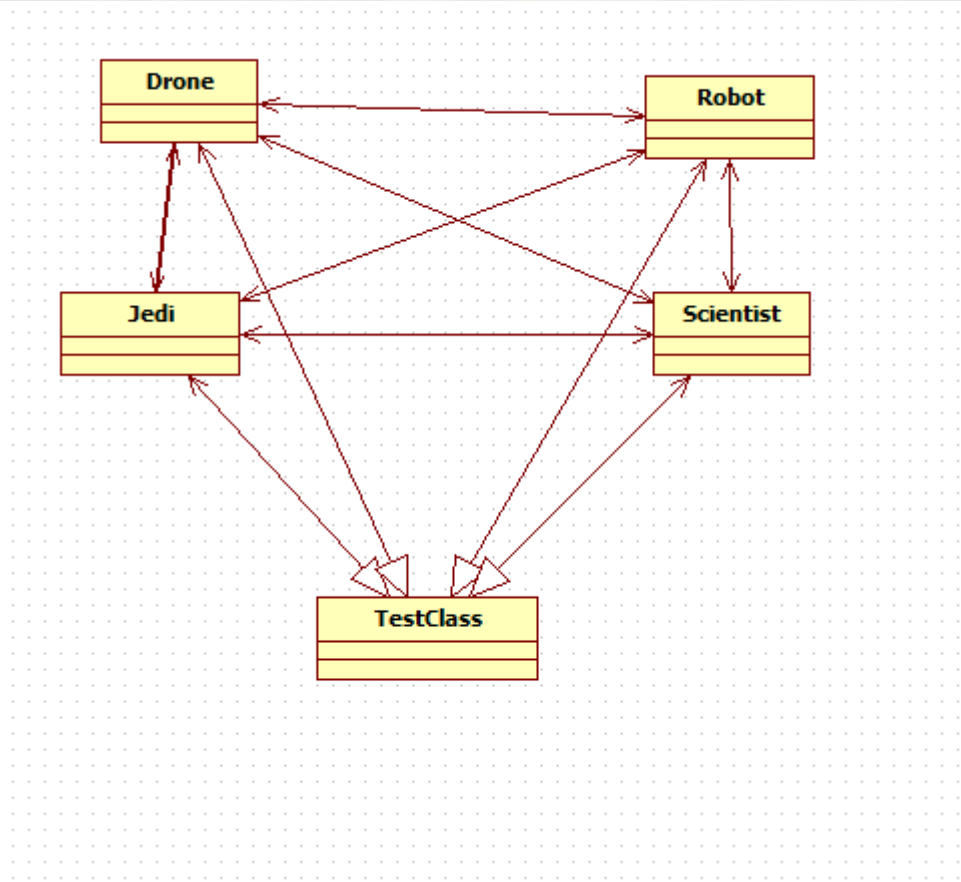
- We have to develop a Mars Based Alpha application
- At the moment we have four classes
  - Jedi which represent a Jedi from a real world
  - DroneConstruction which is responsible for building
  - RobotPressurizedRover which is responsible for exploring
  - Scientist which represent a computer scientist that programs
  - All of them are on a mission to the Mars.

Mediator Pattern

- First we send the pressurized rovers to explore the surface of Mars
- If the rovers sense any danger warn the headquarters
- Headquarters sends the Jedis
- If the negotiations are not successful
- Jedis kills the enemies
- Send the Drones to build defense system and labs
- Scientist start to program

Mediator Pattern

- All the classes needs to know about each other



The GoF pattern

- This is the classic GoF definition of the Mediator pattern:
  - “Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently”.

The GoF pattern

- Mediator is a behavioral pattern
- When we have heavy interaction between components, and they depend on a group of components the logic gets hard to understand and maintain this is where we want to use the mediator
- Help the object to communicate in decoupled manner and reduce complexity

The GoF pattern

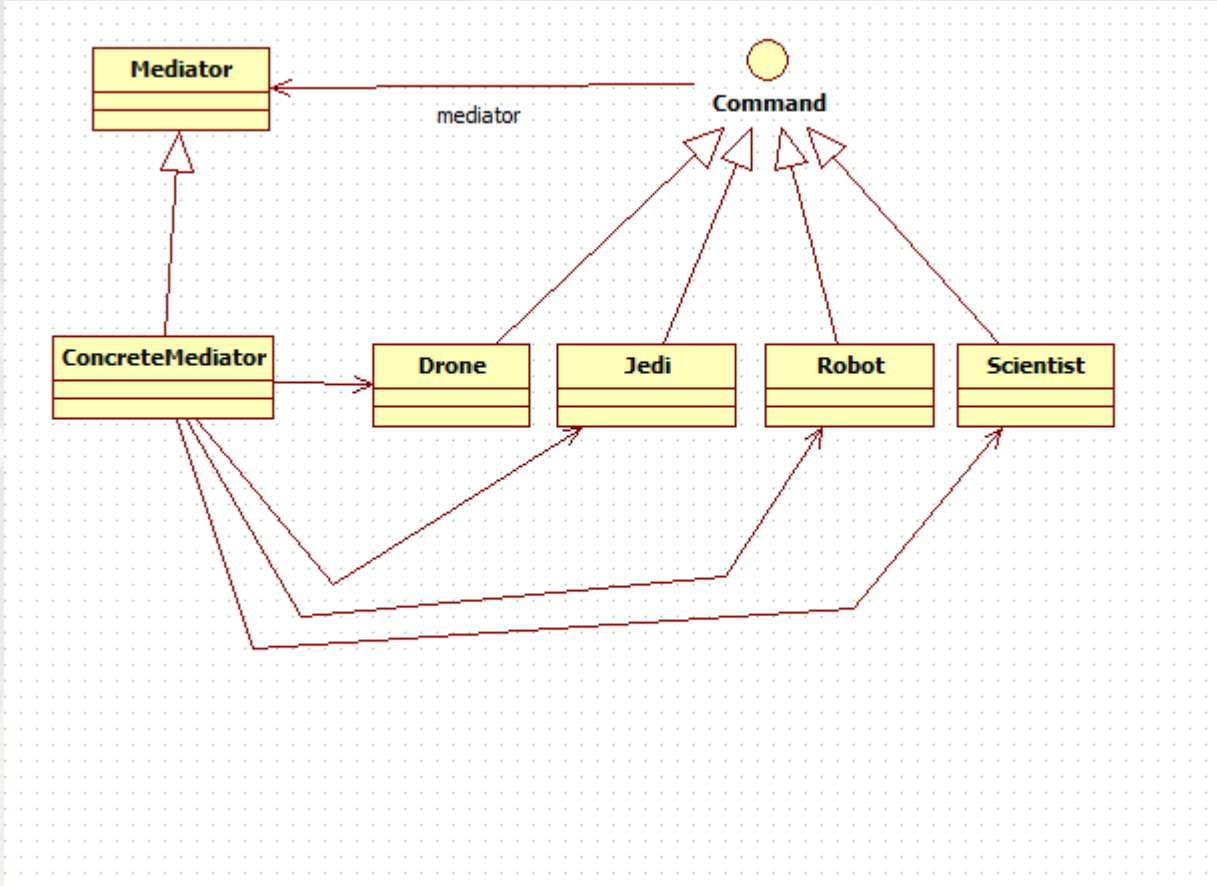
- Use the mediator pattern to encapsulate the navigation code out of the separate classes and place it into a mediator object instead
- From then on each class has to communicate with the mediator when changes occur and the mediator knows what class to send the information to
- If all of the appliance objects need to know about each other and they are all tightly coupled use the Mediator pattern to centralize complex communications and control between related objects

Applying the Mediator

- Jedi/Robot/Drone/Scientist are the concrete classes that communicate with each other via the mediator
- Mediator is the mediator class that needs objects from all the concrete classes and defines methods that can be called by from them
- Command is interface responsible for communication between the Mediator and the concrete classes
- In TestClass is the main method and correspond with the concrete classes through the mediator

Applying the Mediator

- Now the classes don't need to interact with each other but only with the Mediator through the Command interface



Applying the Mediator

- We create a mediator object
- In the constructors we initialize the mediator object
- That is how the mediator is going to communicate with the DroneConsctuction
- In the handle() method we call the handleDrones through the mediator

```
public class DroneConstruction implements Command {

    Mediator mediator;

    DroneStates state;

    public DroneConstruction(Mediator mediator) {
        this.mediator = mediator;

    }

     public DroneStates getState() {
        return state;
    }

    public void setState(DroneStates state) {
        this.state = state;
    }

    @Override
    public void handle() {
        mediator.handleDrones(state);
    }
}
```

Applying the Mediator

- We create a mediator object
- In the constructors we initialize the mediator object
- That is how the mediator is going to communicate with the Jedi
- In the handle() method we call the handleJedis through the mediator

```
public class Jedi implements Command {

    private Mediator mediator;
    private JediStates state;

    public Jedi(Mediator mediator) {
        this.mediator = mediator;
    }
    public JediStates getState() {
        return state;
    }

    public void setState(JediStates state) {
        this.state = state;
    }

    @Override
    public void handle() {
        mediator.handleJedis(state);
    }
}
```

Applying the Mediator

- We create a mediator object
- In the constructors we initialize the mediator object
- That is how the mediator is going to communicate with the RobotPressurizedRover
- In the handle() method we call the handleRobots through the mediator

```
public class RobotPressurizedRover implements Command {

    Mediator mediator;
    RobotStates state;

    public RobotPressurizedRover(Mediator mediator) {
        this.mediator = mediator;
    }

    public RobotStates getState() {
        return state;
    }

    public void setState(RobotStates state) {
        this.state = state;
    }

    @Override
    public void handle() {
        mediator.handleRobots(state);
    }
}
```

Applying the Mediator

- We create a mediator object
- In the constructors we initialize the mediator object
- That is how the mediator is going to communicate with the Scientist
- In the handle() method we call the handleScientist through the mediator

```
public class Scientist implements Command {

    Mediator mediator;
    ScientistStates state;

    public Scientist(Mediator mediator) {
        this.mediator = mediator;
    }

    public ScientistStates getState() {
        return state;
    }

    public void setState(ScientistStates state) {
        this.state = state;
    }

    @Override
    public void handle() {
        mediator.handleScientist(state);
    }
}
```

Applying the Mediator

- We create objects from each concrete class and initialize them in the constructor
- Then we handle the drones in handleDrones

```
public class Mediator {
    private Jedi jedi;
    private DroneConstruction construction;
    private RobotPressurizedRover pressuruzedRover;
    private Scientist computerScientist;

    public Mediator() {
        jedi = new Jedi(this);
        construction = new DroneConstruction(this);
        pressuruzedRover = new RobotPressurizedRover(this);
        computerScientist = new Scientist(this);
    }

    public String handleDrones(DroneStates state) {
        if (state == DroneStates.BUILDHOUSES) {
            return "Drones building houses for jedis!";
        } else if (state == DroneStates.BUILDLABS) {
            return "Drones building a laboratories for scientist.";
        } else if (state == DroneStates.BUILDBUNKERS) {
            return "Drones building a bunkers for defence.";
        } else {
            return null;
        }
    }
```

Applying the Mediator

- According to the corresponding state of the Jedi or Robot we return the appropriate message
- This is where we implement the logic of the concrete classes

```
public String handleJedis(JediStates state) {
    if (state == JediStates.ATTACK) {
        return "Jedis are charging!";
    } else if (state == JediStates.DEFEND) {
        return "Jedis are in protection mode.";
    } else if (state == JediStates.HOLD) {
        return "Jedis are in hold position.";
    } else if (state == JediStates.NEGOTIATE) {
        return "Jedis are waiting for advice from the computer scie
    } else {
        return null;
    }
}

public String handleRobots(RobotStates state) {
    if (state == RobotStates.DANGER) {
        return "Huston we have a problem!, big danger comming.";
    } else if (state == RobotStates.EXPLORE) {
        return "Robot exploring the environment.";
    } else if (state == RobotStates.REPORT) {
        return "Robot report to the headquarters.";
    } else if (state == RobotStates.SEARCH) {
        return "Robot searching.";
    } else {
        return null;
    }
}
```

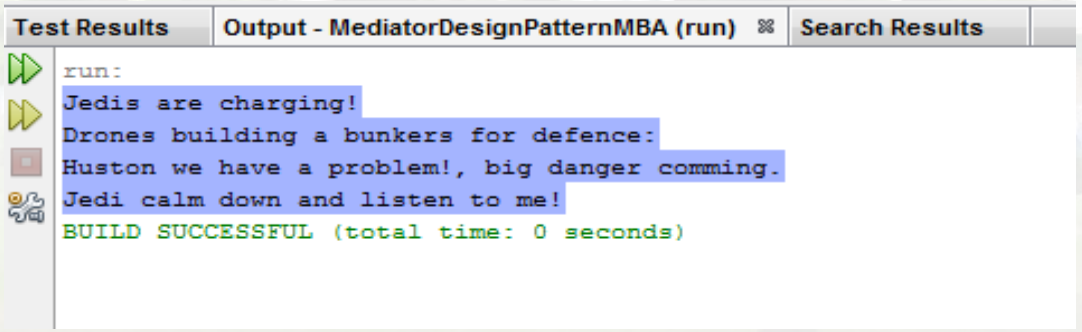
Applying the Mediator

- We create a mediator object
- We call the methods which are implemented in the Mediator
- So we do not have to deal with each class individually

```
public class MediatorDesignPatternMBA {

    public static void main(String[] args) {

        final Mediator mediator = new Mediator();
        mediator.handleJedis(JediStates.ATTACK);
        mediator.handleDrones(DroneStates.BUILDBUNKERS);
        mediator.handleRobots(RobotStates.DANGER);
        mediator.handleScientist(ScientistStates.ADVICEJEDI);
    }
}
```



Examples with the Mediator

- java.util.Timer(all scheduleXXX() methods)
- java.util.concurrent.Executor#execute()
- java.util.concurent.ExecutorService(the invokeXXX() and the submit methods)
- java.util.concurent.ScheduledExecutorService(all scheduleXXX() methods)
- java.lang.reflect.Method#invoke()
- The mediator pattern is used in many GUI framework applications and in the Chat applications

Advantages of the Mediator

- While we are maintaining our project we need to add more functionalities. So our project needs to be updated regularly for example we need to add weather station to our project which will alert drones, rovers, jedi and scientist for the weather on Mars
- So because of the Mediator pattern that we have used there is only WeatherStation <==> Mediator relationship to implement
- Instread of:
- WeatherStation <==> Robot
- WeatherStation <==> Drone
- WeatherStation <==> Jedi
- WeatherStation <==> Scientist

Advantages of the Mediator

- If we need to add twenty more classes like Chef to prepare the food on Mars and Waiter to server the food and Politician who will be in charge the complexity will grow exponentially and it is far harder to implement it and maintain it without the Mediator
- Limits subclasses, localizes behavior that would otherwise be distributed among several objects
- The Mediator promotes high level of re usability: So we do not violate the DRY rule

Advantages of the Mediator

- Comprehension, the mediator encapsulate the logic of mediation between the concrete classes. From this reason it is more easier to understand the logic since it is kept in only one class the mediator.

- Decoupled Classes - The concrete classes are totally decoupled. Adding a new class is relatively easy due to this decoupling level.

- Due to loose coupling, both the mediator and concrete classes can be reused independently of each other

Advantages of the Mediator

- Increases the re usability of the objects supported by the Mediator by decoupling them from the system
- Simplifies maintenance of the system by centralizing control logic
- Simplifies and reduces the variety of messages sent between objects in the system
- Partition a system into pieces or small objects
- Centralize control to manipulate participating objects
- Most of the complexity involved in managing dependencies is shifted from other objects to the mediator object. This makes other objects easier to implement and maintain

Disadvantages of the Mediator

- Usually not a good practice for a relatively small group of objects.
- Without proper design, the Mediator object itself can become overly complex. In practice the mediators tends to become more complex. A good practice is to make sure the mediator classes are responsible only for the communication part.
- Centralized control mediator encapsulates protocols and is more complex than individual classes, as a result it might become harder to maintain.

SOLID

- The code we used is a good example of SOLID
- Every class and method have exactly one responsibility. Jedi class deals only with Jedis and the handleJedis method in the Mediator handles only Jedis. A bad example would be to have one method in the Mediator that deals with everything.
- We also follow the Open-closed principle which is open for extension and closed for modification. So if we modify our code we will need to test it again which is cost effective and not desirable.
- Liskov substitution principle states that functions that use references to base classes must be able to use objects of derived classes without knowing it

SOLID and GRASP

- Interface segregation principle: splits interfaces which are very large into smaller and more specific ones in order to ensure that the clients will only have to use the methods that are necessary for them
- Dependency Inversion Principle: High-level modules should not depend on low-level modules. Both should depend on abstraction and abstraction should not depend upon details. Details should depend upon abstractions.
- A lot of G.R.A.S.P principles are covered in SOLID like low coupling, high cohesion, polymorphism and pure fabrication

Summary

- You should use the mediator pattern when you have to centralize complex communications and control between related objects.
- There are a few design patterns that are closely related to the Mediator pattern and are often used in conjunction with it.
- Facade Patter: a simplified mediator becomes a facade pattern if the mediator is the only active class and the concrete classes are passive. A facade pattern is just an implementation of the mediator pattern where mediator is the only object triggering and invoking actions on passive concrete classes. The Facade is being call by some external classes.

Summary

- Adapter Pattern: the mediator patter just "mediate" the requests between the concrete classes. It is not supposed to change the messages it receives and sends. If it alters those messages then it is an Adapter pattern.
- Observer Pattern: the observer and mediator are similar patterns, solving the same problem. The main difference between them is the problem they address. The observer pattern handles the communication between observers and subjects or subject. It's very probable to have new observable objects added. On the other side in the mediator pattern the mediator class is the the most likely class to be inherited.

References

- 1. Gang of Four August 1994
- 2. Head First Design Patterns 2004 O'REILLY
- 3.Design Patterns for Dummies 2006 by Wiley Publishing, Inc., Indianapolis, Indiana
- 4. http://www.slideshare.net/devel123/solid-grasp-9816996#btnNext
- 5.http://snehaprashant.blogspot.co.uk/2009/01/mediator-pattern-in-java.html
- 6.http://www.oodesign.com/index2.php?option=com_content&do_pdf=1&id=24
- 7. http://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns