Lesson 10 Mediator and Observer

The Mediator Pattern

1. Intent

   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.

2. Motivation

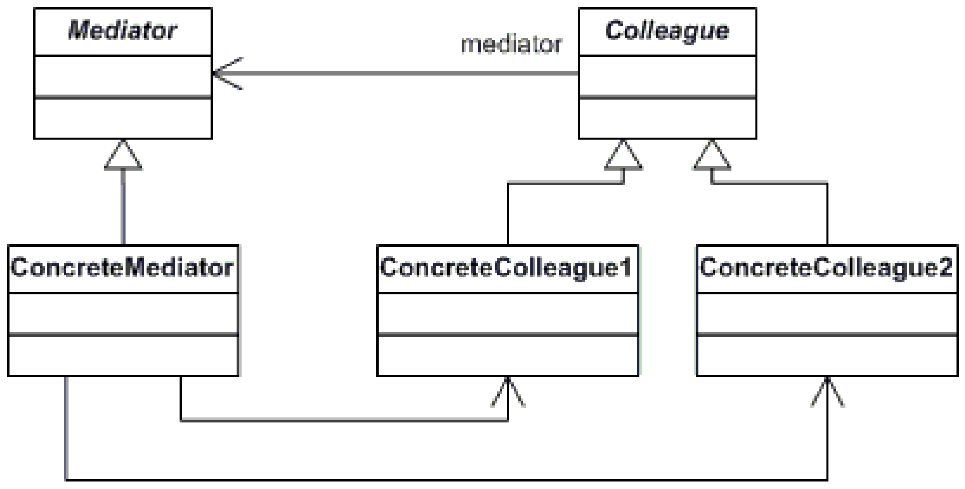
   a. In a system where multiple objects communicate to each other by directly sending messages to others, communication between these objects may become complex.
   b. This makes the program harder to maintain since any change may affect code in other objects or classes.
   c. Use the Mediator Pattern to centralize complex communications and control between related objects.
   d. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently

3. Applicability

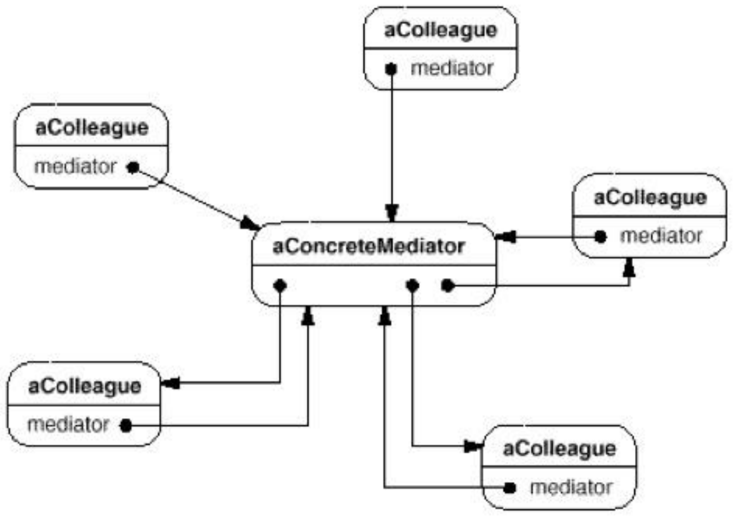
   Use the Mediator pattern when
   a. A set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.

b. Reusing an object is difficult because it refers to and communicates with many other objects.
c. A behavior that's distributed between several classes should be customizable without a lot of subclassing.

4. Structure



A typical object structure looks like this:



5. Participants

a. Mediator
   - defines an interface for communicating with Colleague objects.

b. ConcreteMediator
   - implements cooperative behavior by coordinating Colleague objects.
   - knows and maintains its colleagues.

c. Colleague classes
   - each Colleague class knows its Mediator object.
   - each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.

6. Consequences

The Mediator pattern has the following benefits and drawbacks:

a. A mediator localizes behavior that otherwise would be distributed among several objects. Changing this behavior requires subclassing Mediator only; Colleague classes can be reused as is.

b. It decouples colleagues. A mediator promotes loose coupling between colleagues.

c. It simplifies object protocols. A mediator replaces many-to-many interactions with one-to-many interactions between the mediator and its colleagues.

d. It centralizes control which can make the mediator itself a monolith that's hard to maintain.

7. How to implement the Mediator pattern?

a. Mediator interface

```java
public interface ChatroomMediator {
      public void sendMessage(String msg, User user);
      public void addUser(User user);
}
```

b. Colleague interface/abstract class

```java
public abstract class User {
    protected ChatroomMediator mediator;
    protected String name;

    public User(ChatroomMediator med, String name){
        this.mediator=med;
        this.name=name;
    }

    public abstract void send(String msg);
    public abstract void receive(String msg);
}
```

c. Concrete mediator

```java
public class ChatroomMediatorImpl implements ChatroomMediator {

      private List<User> users;
      public ChatroomMediatorImpl() {
            this.users = new ArrayList<User>();
      }

      @Override
      public void addUser(User user) {
            this.users.add(user);
      }

      @Override
      public void sendMessage(String msg, User user) {
            for (User u : this.users) {
                  // message should be sent to all colleagues except
                  him/herself
                  if (u != user) {
                        u.receive(msg);
                  }
            }
      }
}
```

d. Concrete colleague

```java
public class UserImpl extends User {
        public UserImpl(ChatroomMediator med, String name) {
                super(med, name);
        }

        @Override
        public void send(String msg) {
                System.out.println(this.name + ": Sending Message=" + msg);
                mediator.sendMessage(msg, this);
        }

        @Override
        public void receive(String msg) {
                System.out.println(this.name + ": Received Message:" +
msg);
        }

}
```

e. Client

```
public class ChatClient {

    public static void main(String[] args) {
        ChatroomMediator mediator = new ChatroomMediatorImpl();
        User user1 = new UserImpl(mediator, "Pan");
        User user2 = new UserImpl(mediator, "Lis");
        User user3 = new UserImpl(mediator, "Sar");
        User user4 = new UserImpl(mediator, "Dav");
        mediator.addUser(user1);
        mediator.addUser(user2);
        mediator.addUser(user3);
        mediator.addUser(user4);

        user1.send("Hi All");

    }

}
```

Lab 10-1

1. Consider a board game of Reversi. The problem of deciding whether a player can occupy a position on the board can be delegated to the position itself or to a mediator. The mediator encapsulates how the positions interact in a move of the game.
   Step 1: Design and write the program with a Mediator pattern.
   Step 2: Develop a GUI that 2 players can play the game with. (optional)

2. (optional) I recently spoke with a Compro student who had an interview with Amazon. He was asked to develop a minesweeper game. Are you going to use Mediator to implement it? Why or why not? If you want to do this one as the lab, you do not have to do question 1.

The Observer Pattern

1. Intent

   Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

2. Motivation

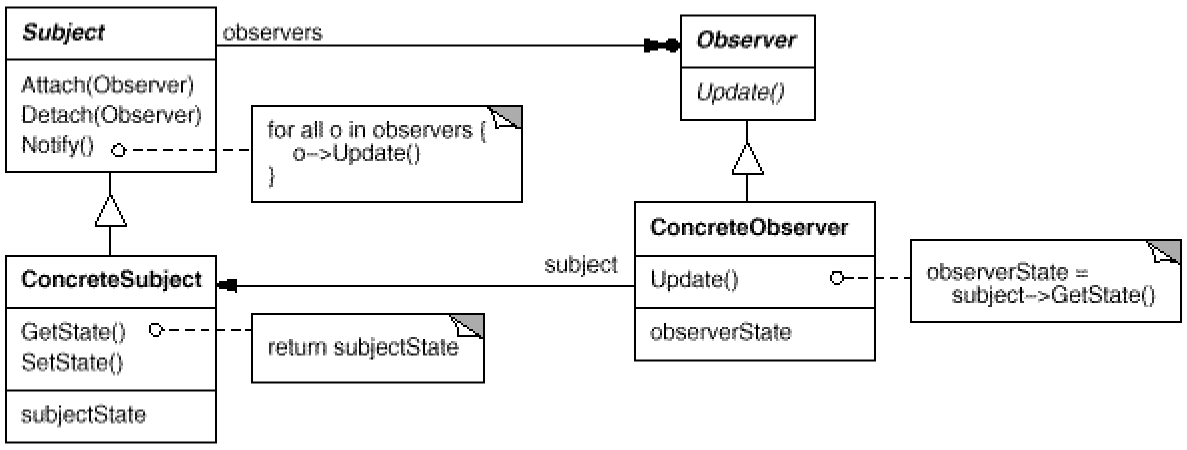
   a. Sometimes we want to model a 1-to-many publisher-subscriber relationship in our applications.
   b. The multiple subscribers are dependent on the state of the publisher; therefore they need to be notified of any change of state with the publisher.
   c. In order to reuse the publisher and subscriber classes, their relationship has to be decoupled.

3. Applicability

   Use the Observer pattern when

   a. When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
   b. When a change to one object requires changing others, and you don't know how many objects need to be changed.
   c. When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

4. Structure



5. Participants

a. Subject
- knows its observers. Any number of Observer objects may observe a subject.
- provides an interface for attaching and detaching Observer objects.

b. Observer
- defines an updating interface for objects that should be notified of changes in a subject.

c. ConcreteSubject
- stores state of interest to ConcreteObserver objects.
- sends a notification to its observers when its state changes.

d. ConcreteObserver
- maintains a reference to a ConcreteSubject object.
- stores state that should stay consistent with the subject's.
- implements the Observer updating interface to keep its state consistent with the subject's.

6. Consequences

- it supports layering of software applications.
- it supports broadcast communication without having to know all observers. The subject only knows that it has to update a list of observers whenever there is a change of state.

7. Implementation

a. Subject interface

```
public interface Subject {
	public void attach(Observer observer);
	public void detach(Observer observer);
	public void notifyObservers();
}
```

b. Concrete subject

```
public class StockData implements Subject {
	private String symbol;
	private float close;
	private float high;
	private float low;
	private long volume;
	private List<Observer> observers;
	private final Object MUTEX = new Object();

	public StockData() {
		observers = new ArrayList<Observer>();
	}

	@Override
	public void attach(Observer observer) {
		synchronized (MUTEX) {
			if (!observers.contains(observer))
				observers.add(observer);
		}
	}

	@Override
	public void detach(Observer observer) {
		synchronized (MUTEX) {
```

```
            int i = observers.indexOf(observer);
            if (i >= 0)
                observers.remove(i);
        }
    }

    @Override
    public void notifyObservers() {
        synchronized (MUTEX){
            int n = observers.size();
            for (int i = 0; i < n; ++i) {
                Observer observer = (Observer)
                    observers.get(i);
                observer.update(symbol, close, high, low,
                    volume);
            }
        }
    }

    public void sendStockData() {
        notifyObservers();
    }

    public void setStockData(String symbol, float close, float
            high, float low, long volume) {
        this.symbol = symbol;
        this.close = close;
        this.high = high;
        this.low = low;
        this.volume = volume;
        sendStockData();
    }

    public String getSymbol() {
        return symbol;
    }

    public float getClose() {
        return close;
    }

    public float getHigh() {
        return high;
    }

    public float getLow() {
```

```
		return low;
	}

	public long getVolume() {
		return volume;
	}
}
```

c. Observer interface

```
public interface Observer {
	public void update(String symbol, float close, float high, float
		low, long volume);
}
```

d. Concrete observers

```
public class BigBuyer implements Observer {
	private String symbol;
	private float close;
	private float high;
	private float low;
	private long volume;

	public BigBuyer(StockData stockData) {
		stockData.attach(this);
	}

	public void update(String symbol, float close, float high, float
			low,  long volume) {
		this.symbol = symbol;
		this.close = close;
		this.high = high;
		this.low = low;
		this.volume = volume;
		display();
	}

	public void display() {
		DecimalFormatSymbols dfs = new DecimalFormatSymbols();
		DecimalFormat volumeFormat = new
			DecimalFormat("###,###,###,###", dfs);
		DecimalFormat priceFormat = new DecimalFormat("###.00",
			dfs);
		System.out.println("Big Buyer reports... ");
		System.out.println("\tThe lastest stock quote for " +
			symbol + " is:");
		System.out.println("\t$" + priceFormat.format(close)
				+ " per share (close).");
		System.out.println("\t$" + priceFormat.format(high)
				+ " per share (high).");
		System.out.println("\t$" + priceFormat.format(low)
				+ " per share (low).");
		System.out.println("\t" + volumeFormat.format(volume)
				+ " shares traded.");
```

```
            System.out.println();
        }
}



public class SmallBuyer implements Observer {
        private String symbol;
        private float close;

        public SmallBuyer(StockData stockData) {
                stockData.attach(this);
        }

        public void update(String symbol, float close, float high, float
                low, long volume) {
                this.symbol = symbol;
                this.close = close;
                display();
        }

        public void display() {
                DecimalFormatSymbols dfs = new DecimalFormatSymbols();
                DecimalFormat priceFormat = new DecimalFormat("###.00",
                        dfs);
                System.out.println("Trading Fool says... ");
                System.out.println("\t" + symbol + " is currently trading
                        at $" + priceFormat.format(close) + " per share.");
                System.out.println();
        }
}
```

e. Client

```
public class Client {
        public static void main(String[] args) {
                System.out.println();
                System.out.println("-- Stock Quote Application --");
                System.out.println();
                StockData stockData = new StockData();
                // register observers...
                new SmallBuyer(stockData);
                new BigBuyer(stockData);
                // generate changes to stock data...
                stockData.setStockData("JUPM", 16.10f, 16.15f, 15.34f,
                        (long) 481172);
                stockData.setStockData("SUNW", 4.84f, 4.90f, 4.79f, (long)
                        68870233);
                stockData.setStockData("MSFT", 23.17f, 23.37f, 23.05f,
                        (long) 75091400);
        }
```

Lab 10-2

Develop a program that displays persons’ names on a GUI. The names are stored in a collection object. As you add/remove names at runtime, your GUI should be synchronized with the content change in the data collection. Use the Observer Pattern to design/write the program.