Design Patterns
Observer, Strategy, MVC

B.Tech. (IT), Sem-6,
Applied Design Patterns and Application Frameworks (ADPAF)

Dharmsinh Desai University
Prof. H B Prajapati

The Observer design pattern

- It is behavioral design pattern
- Used when there is one-to-many relationship between objects and
- If one object is modified, its dependent objects are to be notified automatically.

The Observer design pattern

- Problem
  - Suppose a class (say ShapeArchiver) is responsible for archiving information about all the drawn shapes.
  - Canvas is responsible for displaying all drawn shapes
  - Whenever any change in shapes takes place, you need to inform these two classes as to the changed information

The Observer design pattern

```
class Point {
   private int xPos;
   private int yPos;
   public Point(int x, int y) {
      xPos = x;
      yPos = y;
   }
   public String toString() {
      return "(" + xPos + "," + yPos + ")";
   }
}
```


The Observer design pattern

```
public class Circle {
   private Point center;
   private int radius;
   public Circle(int x, int y, int r) {
      center = new Point(x, y);
      radius = r;
   }
   public void setCenter(Point center)
    {
      this.center = center;
      canvas.update(this);
      shapeArchiver.update(this);
   }
```

```
public void setRadius(int radius) {
      this.radius = radius;
      canvas.update(this);
      shapeArchiver.update(this);
  }
private ShapeArchiver
   shapeArchiver;
  public void
   setShapeArchiver(ShapeArchiver
   shapeArchiver) {
     this.shapeArchiver =
   shapeArchiver;
  }
```


The Observer design pattern

```
protected Canvas canvas;
   public void setCanvas(Canvas canvas) {
      this.canvas = canvas;
   }
   public String toString() {
      return "center = " + center + " and radius = " + radius;
   }
}
```

The Observer design pattern

```
public class Canvas {
   public void update(Circle circle) {
      System.out.println("Canvas::update");
      //update implementation
   }
}

public class ShapeArchiver {
   public void update(Circle circle) {
      System.out.println("ShapeArchiver::update");
      // update implementation
   }
}
```

7

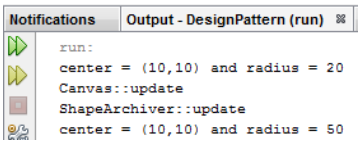
The Observer design pattern

```
public class Test {
   public static void main(String []s) {
      Circle circle = new Circle(10, 10, 20);
      System.out.println(circle);
      circle.setCanvas(new Canvas());
      circle.setShapeArchiver(new ShapeArchiver());
      circle.setRadius(50);
      System.out.println(circle);
   }
}
```

8

The Observer design pattern

Output



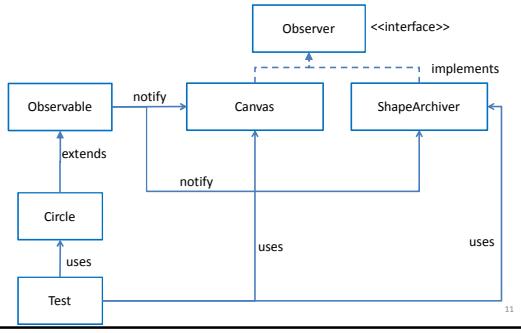
9

The Observer design pattern

- The solution works, but it has a problem.
- There is a tight coupling between the subject (Circle class) and both of the observers
  - ShapeArchiver and
  - Canvas
- Consequences of a tightly coupled design:
  - The subject class (Circle) knows about the specific observer classes. As a result, if you change observer classes, you need to change subject class, too.
  - If you want to add or remove an observer, you cannot do it without changing the subject.
  - You cannot reuse either the subject or the observer classes independently.

10

The Observer design pattern



11

The Observer design pattern

```
class Point {
   private int xPos;
   private int yPos;
   public Point(int x, int y) {
      xPos = x;
      yPos = y;
   }
   public String toString() {
      return "(" + xPos + "," + yPos + ")";
   }
}
```

12

The Observer design pattern

```
import java.util.Observable;
public class Circle extends
    Observable {
  private Point center;
  public void setCenter(Point center)
   {
    this.center = center;
    setChanged();
    notifyObservers(this);
  }
  public void setRadius(int radius) {
    this.radius = radius;
    setChanged();
    notifyObservers(this);
  }
```

```
private int radius;
  public Circle(int x, int y, int r) {
    center = new Point(x, y);
    radius = r;
  }
  public String toString() {
    return "center = " + center + "
   and radius = " + radius;
  }
}
```

13

The Observer design pattern

```
import java.util.Observable;
import java.util.Observer;
public class Canvas implements Observer {
  @Override
  public void update(Observable arg0, Object arg1) {
    System.out.println("Canvas::update");
    Circle c=(Circle)arg1;
    System.out.println("In Canvas::update, circle = "+c);
  }
}
```

14

The Observer design pattern

```
import java.util.Observable;
import java.util.Observer;
public class ShapeArchiver implements Observer{
  @Override
  public void update(Observable arg0, Object arg1) {
    System.out.println("ShapeArchiver::update");
    Circle c=(Circle)arg1;
    System.out.println("In ShapeArchiver::update, circle = "+c);
  }
}
```

15

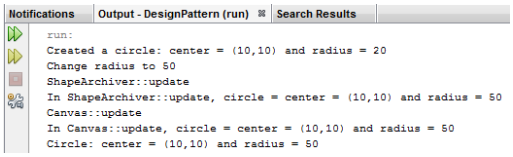
The Observer design pattern

```
public class Test {
  public static void main(String []s) {
    Circle circle = new Circle(10, 10, 20);
    System.out.println("Created a circle: "+circle);
    circle.addObserver(new Canvas());
    circle.addObserver(new ShapeArchiver());
    System.out.println("Change radius to 50");
    circle.setRadius(50);
    System.out.println("Circle: "+circle);
  }
}
```

16

The Observer design pattern

Output



17

The Observer design pattern

- Advantages
  - The subject—Circle class—does not know about the concrete observer classes, and
  - the observers do not know about the concrete subject.
  - Both the subject and observers can now be used independently and changed independently.
  - Furthermore, you can add and remove observers from the subject without changing the subject class.
- Principle
  - Observer defines a one-to-many dependency between objects so that when one object changes state, all its dependencies are notified and updated automatically.

18

The Observer design pattern

- Java supports an abstract class with the name Observable and
- An interface named Observer(both provided in the java.util package) to implement the Observer design pattern.
- Whenever the state of subject is changed, you call the setChanged() method followed by notifyObservers(),which is implemented in the Observable class.
- The notifyObservers() method calls all observers registered earlier for that subject.

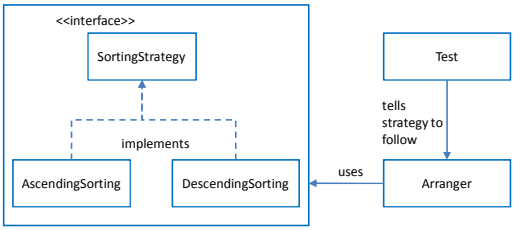
19

The Strategy design pattern

- It is of type behavioral design pattern.
- This design pattern allows changing a class behavior or its algorithm at run time.
- Various strategies are created in different objects.
- The strategy object changes the executing algorithm of the context object.

20

The Strategy design pattern



21

Example-The Strategy design pattern

- SortingStrategy (Interface for Strategy operation)

```
public interface SortingStrategy {
   public void sort(int[] numbers);
}
```

22

Example-The Strategy design pattern

- AscendingSorting (implementation of strategy: ascending sorting)

```
package designpattern.strategy;
public class AscendingSorting implements SortingStrategy{
  @Override
  public void sort(int[] numbers) {
    int numberCount=numbers.length;       int temp;
    for (int i = 0; i < numberCount-1; i++) {
      for (int j = i+1; j < numberCount; j++) {
        if(numbers[i]>numbers[j])
          temp=numbers[j];   numbers[j]=numbers[i];  numbers[i]=temp;
      }
    }
  }
}
```

23

Example-The Strategy design pattern

- DescendingSorting (implementation of strategy: descending sorting)

```
package designpattern.strategy;
public class DescendingSorting implements SortingStrategy{
  @Override
  public void sort(int[] numbers) {
    int numberCount=numbers.length; int temp;
    for (int i = 0; i < numberCount-1; i++) {
      for (int j = i+1; j < numberCount; j++) {
        if(numbers[i]<numbers[j])
          temp=numbers[j]; numbers[j]=numbers[i]; numbers[i]=temp;
      }
    }
  }
}
```

24

Example-The Strategy design pattern

- Arranger (it takes implementation of strategy as input)

```
package designpattern.strategy;
public class Arranger {
   private SortingStrategy sortStrategy;
   public Arranger(SortingStrategy sortStrategy){
      this.sortStrategy=sortStrategy;
   }
   public void arrange(int[] numbers){
      sortStrategy.sort(numbers);
   }
}
```

25

Example-The Strategy design pattern

```
package designpattern.strategy;
import java.util.Arrays;
public class Test {
   public static void main(String[] args) {
      int[] numbers1={55, 11, 44, 99, 66, 77, 22};
      int[] numbers2={55, 11, 44, 99, 66, 77, 22};
      System.out.println("Numbers in Original Order
    :"+Arrays.toString(numbers1));

      Arranger ascendingArranger=new Arranger(new AscendingSorting());
      ascendingArranger.arrange(numbers1);

      System.out.println("Numbers in Ascending Order  :" +
     Arrays.toString(numbers1));
```

26

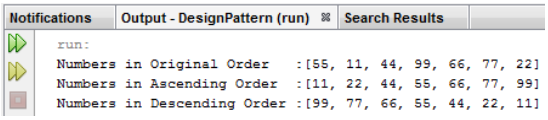
Example-The Strategy design pattern

```
   Arranger descendingArranger=new Arranger(new DescendingSorting());
      descendingArranger.arrange(numbers2);

      System.out.println("Numbers in Descending Order :"+
     Arrays.toString(numbers2));
   }
}
```

27

Running the Example-The Strategy design pattern



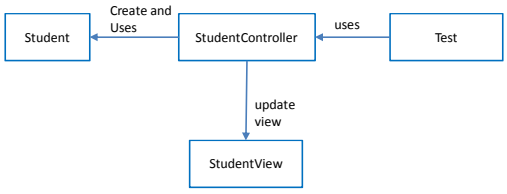
28

The MVC-Model-View-Controller design pattern

- It is of type architectural design pattern.
- This design pattern allows separating different responsibilities to different entities.
  - Loose coupling (Do not combine data with its representation)
  - Maintainable (Different type of code in different entity)
  - Reusability (Data can be reused)
- The design pattern consists of three entities
  - Model: It contains data and business logic
  - View: It creates visualization/presentation of the data (model)
  - Controller:
    - It keeps View and Model separate
    - It controls the data flow into the model object.
    - It updates the view whenever data changes.

29

The MVC-Model-View-Controller design pattern



30

Example-The MVC-Model-View-Controller design pattern

- Student (A model class)

```
package designpattern.mvc;
public class Student {
   private String name;
   private int rollNo;
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
```

31

Example-The MVC-Model-View-Controller design pattern

```
   public int getRollNo() {
      return rollNo;
   }
   public void setRollNo(int rollNo) {
      this.rollNo = rollNo;
   }
}
```

32

Example-The MVC-Model-View-Controller design pattern

- Student View(A view class)

```
package designpattern.mvc;
public class StudentView {
   public void displayDetails(String name, int rollNo){
      System.out.println("Student # Name:"+name+" ,
   Roll No:"+rollNo);
   }
}
```

33

Example-The MVC-Model-View-Controller design pattern

- Student Controller (A controller class)

```
package designpattern.mvc;
public class StudentController {
   private Student model;
   private StudentView view;
   public StudentController(Student model, StudentView view){
      this.model=model;
      this.view=view;
   }
   public void setStudentName(String name){
      model.setName(name);
   }
```

34

Example-The MVC-Model-View-Controller design pattern

```
   public String getStudentname(){
      return model.getName();
   }
   public void setStudentRollNo(int rollNo){
      model.setRollNo(rollNo);
   }
   public int getStudentRollNo(){
      return model.getRollNo();
   }
   public void updateView(){
      view.displayDetails(model.getName(), model.getRollNo());
   }
}
```

35

Example-The MVC-Model-View-Controller design pattern

```
package designpattern.mvc;
public class Test {
   public static void main(String[] args) {
      Student student=getStudentFromDB();

      StudentView view=new StudentView();
      StudentController controller=new StudentController(student, view);
      controller.updateView();
      System.out.println("Update student's Roll No.");
      controller.setStudentRollNo(1);
      controller.updateView();
   }
```

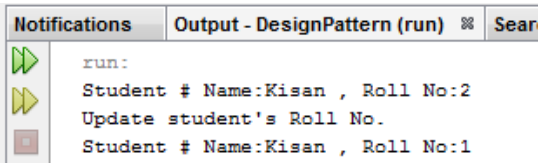
36

Example-The MVC-Model-View-Controller design pattern

```
  public static Student getStudentFromDB(){
     Student st=new Student();
     st.setName("Kisan");
     st.setRollNo(2);
     return st;
  }
}
```

37

Running the Example-The MVC-Model-View-Controller design pattern



38

References

- Oracle Certified Professional Java SE 7 Programmer Exams 1Z0-804 and 1Z0-804 (A Comprehension OCPJP 7 Certification Guide), by S G Ganesh and Tushar Sharma, publisher Apress
- Java Design Patterns, problem solving approaches, tutorials point, www.tutorialspoint.com

39