Software Re-Engineering

Design Patterns

Dr. Syed Muazzam Ali Shah
Department of Software Engineering
National University of Computer & Emerging Sciences
muazzam.ali@nu.edu.pk

Design Pattern

- Design patterns represent the best practices used by experienced object-oriented software developers.
- Design patterns are solutions to general problems that software developers faced during software development.
- These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

Singleton Pattern

- Singleton pattern is one of the simplest design patterns in Java.
- This type of design pattern comes under creational pattern:
  - As this pattern provides one of the best ways to create an object.
- This pattern involves a single class which is responsible to create an object while making sure that only single object gets created.
- This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Singleton Pattern - Implementation

- We're going to create a *SingletonObject* class.
- *SingletonObject* class have its constructor as private and have a static instance of itself.
- *SingletonObject* class provides a static method to get its static instance to outside world.
- *SingletonPatternDemo*, our demo class will use *SingletonObject* class to get a *SingletonObject* object.

Singleton Pattern – Implementation steps

- **Step 1:** Create a Singleton Class: *SingletonObject.java*
- **Step 2:** Get the only object from the singleton class: *SingletonPatternDemo.java*
- **Step 3:** Verify the output.

Singleton Pattern - Example

```
public class SingletonObject {
//create an object of SingletonObject
private static SingletonObject instance = new
SingletonObject();

//make the constructor private so that this class cannot be
instantiated
private SingletonObject() {}
//Get the only object available
public static SingletonObject getInstance() {
return instance;
}

public void showMessage() {
System.out.println("Hello World");
}
}
```

Singleton Pattern - Example

```
public class SingletonPatternDemo {
public static void main(String[] args) {
//illegal construct
//compile time error: the constructor SingletonObject is not
visible
//SingletonObject object = new SingletonObject();

//Get the only object available
SingletonObject object = SingletonObject.getInstance();

//show the message
object.showMessage();
}
}
```

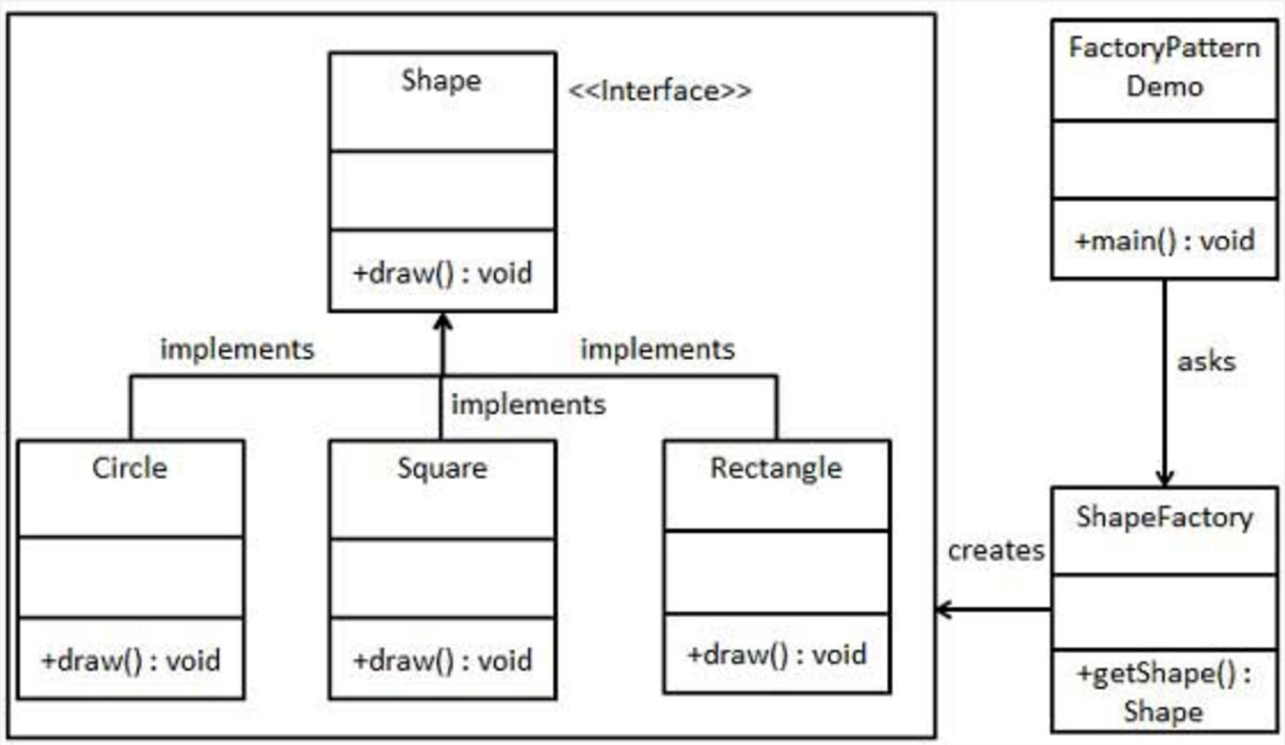
Factory Design Pattern

- Factory pattern is one of the most used design patterns in Java.
- This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Factory Design Pattern

- We're going to create a `Shape` interface and concrete classes implementing the `Shape` interface.
- A factory class `ShapeFactory` is defined as a next step.
- `FactoryPatternDemo`, our demo class will use `ShapeFactory` to get a `Shape` object.
- It will pass information `(CIRCLE/RECTANGLE/SQUARE)` to `ShapeFactory` to get the type of object it needs.

Factory Design Pattern



Factory Design Pattern – Implementation

Step 1

Create an interface.

Shape.java

```
public interface Shape {
void draw();
}
```

Factory Design Pattern – Implementation

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Factory Design Pattern – Implementation

Step 2

Create concrete classes implementing the same interface.

Square.java

```
public class Square implements Shape {


   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Factory Design Pattern – Implementation

Step 2

Create concrete classes implementing the same interface.

Square.java

```
public class Square implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Factory Design Pattern – Implementation

Step 2

Create concrete classes implementing the same interface.

Circle.java

```
public class Circle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

Factory Design Pattern – Implementation

Step 3

Create a Factory to generate object of concrete class based on given information.

ShapeFactory.java

```
public class ShapeFactory {
   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();
      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
            return null;
   }
}
```

Factory Design Pattern – Implementation

Step 4

Use the Factory to get object of concrete class by passing an information such as type.

FactoryPatternDemo.java

```
public class FactoryPatternDemo {
   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();
      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");
      //call draw method of Circle
      shape1.draw();
      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");
      //call draw method of Rectangle
      shape2.draw();
      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");
      //call draw method of square
      shape3.draw();
   }
}
```

Facade Design Pattern

- Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system.
- This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.
- This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

Factory Design Pattern - Benefits

- It hides the logic of instantiation.
- Reduce the duplicity of code.
- Once Place Changes will change everywhere.

Facade Design Pattern

- When you create a simplified interface that performs many other actions behind the scenes.
  - Can I withdrawal $50 from the Bank?
  - Check if the Checking account is valid.
  - Check if the security code is valid.
  - Check if funds are available
  - Make changes accordingly

Facade Design Pattern – Implementation

```
public class WelcomeToBank{

    public WelcomeToBank() {

        System.out.println("Welcome to ABC Bank");

        System.out.println("We are happy to give you your
money if we can find it\n");

    }

}
```

Facade Design Pattern – Implementation

```
public class AccountNumberCheck{
  private int accountNumber = 12345678;
  public int getAccountNumber() { return accountNumber; }
  public boolean accountActive(int acctNumToCheck){
      if(acctNumToCheck == getAccountNumber()) {
          return true;
      } else {
          return false;
      }
  }
}
```

Facade Design Pattern – Implementation

```
public class SecurityCodeCheck {
  private int securityCode = 1234;
  public int getSecurityCode() { return securityCode; }
  public boolean isCodeCorrect(int secCodeToCheck){
      if(secCodeToCheck == getSecurityCode()) {
          return true;
      } else {
          return false;
      }
  }
}
```

Facade Design Pattern – Implementation

```
public class FundsCheck {
  private double cashInAccount = 1000.00;
  public double getCashInAccount() { return cashInAccount; }
  public void decreaseCashInAccount(double cashWithdrawn) { cashInAccount -=
cashWithdrawn; }
  public void increaseCashInAccount(double cashDeposited) { cashInAccount +=
cashDeposited; }
  public boolean haveEnoughMoney(double cashToWithdrawal) {
      if(cashToWithdrawal > getCashInAccount()) {
          System.out.println("Error: You don't have enough money");
          System.out.println("Current Balance: " + getCashInAccount());
          return false;
      } else {
          decreaseCashInAccount(cashToWithdrawal);
          System.out.println("Withdrawal Complete: Current Balance is " +
getCashInAccount());
          return true;
      }
  }
  public void makeDeposit(double cashToDeposit) {
      increaseCashInAccount(cashToDeposit);
      System.out.println("Deposit Complete: Current Balance is " +
getCashInAccount());
  }
}
```

Facade Design Pattern – Implementation

```
// The Facade Design Pattern decouples or separates the client
// from all of the sub components
// The Facades aim is to simplify interfaces so you don't have
// to worry about what is going on under the hood
public class BankAccountFacade {
  private int accountNumber;
  private int securityCode;

  AccountNumberCheck acctChecker;
  SecurityCodeCheck codeChecker;
  FundsCheck fundChecker;
  WelcomeToBank bankWelcome;

 public BankAccountFacade(int newAcctNum, int newSecCode){
 AccountNumberCheck accountNumber = newAcctNum;
      securityCode = newSecCode;
      bankWelcome = new WelcomeToBank();
      acctChecker = new AccountNumberCheck();
      codeChecker = new SecurityCodeCheck();
      fundChecker = new FundsCheck();
  }
```

Facade Design Pattern – Implementation

```
    public int getAccountNumber() { return accountNumber; }
    public int getSecurityCode() { return securityCode; }

    public void withdrawCash(double cashToGet){
        if(acctChecker.accountActive(getAccountNumber()) &&
                codeChecker.isCodeCorrect(getSecurityCode()) &&
                fundChecker.haveEnoughMoney(cashToGet)) {
                    System.out.println("Transaction Complete\n");
                } else {
                    System.out.println("Transaction Failed\n");
                }
    }
    public void depositCash(double cashToDeposit){
        if(acctChecker.accountActive(getAccountNumber()) &&
                codeChecker.isCodeCorrect(getSecurityCode())) {
                    fundChecker.makeDeposit(cashToDeposit);
                    System.out.println("Transaction Complete\n");
                } else {
                    System.out.println("Transaction Failed\n");
                }
    }
}
```

Facade Design Pattern – Implementation

```
public class TestBankAccount {
  public static void main(String[] args){
      BankAccountFacade accessingBank = new
BankAccountFacade(12345678, 1234);
      accessingBank.withdrawCash(50.00);
      accessingBank.depositCash(2000.00);
      accessingBank.withdrawCash(3000.00);
  }
}
```

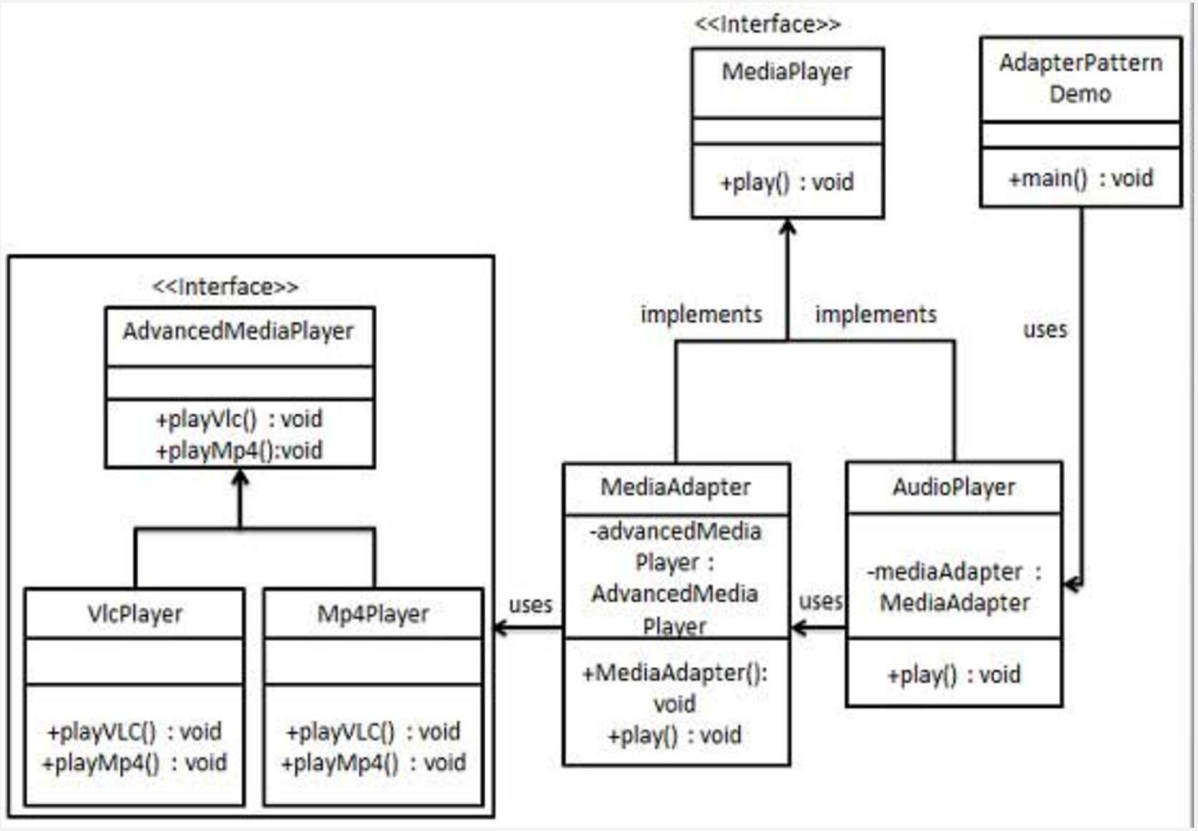
Adapter Design Pattern

- Adapter pattern works as a bridge between two incompatible interfaces.
- This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.
- This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces.
- A real life example could be a case of card reader which acts as an adapter between memory card and a laptop.
- You plugin the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.
- We are demonstrating use of Adapter pattern via following example in which an audio player device can play mp3 files only and wants to use an advanced audio player capable of playing vlc and mp4 files.

Adapter Design Pattern - Implementation

- We have a *MediaPlayer* interface and a concrete class *AudioPlayer* implementing the *MediaPlayer* interface.
- *AudioPlayer* can play mp3 format audio files by default.
- We are having another interface *AdvancedMediaPlayer* and concrete classes implementing the *AdvancedMediaPlayer* interface.
- These classes can play vlc and mp4 format files.
- We want to make *AudioPlayer* to play other formats as well.
- To attain this, we have created an adapter class *MediaAdapter* which implements the *MediaPlayer* interface and uses *AdvancedMediaPlayer* objects to play the required format.
- *AudioPlayer* uses the adapter class *MediaAdapter* passing it the desired audio type without knowing the actual class which can play the desired format.
- *AdapterPatternDemo*, our demo class will use *AudioPlayer* class to play various formats.

Adapter Design Pattern - Implementation



Adapter Design Pattern - Implementation

Step 1

Create interfaces for Media Player and Advanced Media Player.

MediaPlayer.java

```
public interface MediaPlayer {
 public void play(String audioType, String fileName);
}
```

AdvancedMediaPlayer.java

```
public interface AdvancedMediaPlayer {
   public void playVlc(String fileName);
   public void playMp4(String fileName);
}
```

Adapter Design Pattern - Implementation

Step 2

Create concrete classes implementing the *AdvancedMediaPlayer* interface.

VlcPlayer.java

```
public class VlcPlayer implements AdvancedMediaPlayer{
 @Override
 public void playVlc(String fileName)
 {
    System.out.println("Playing vlc file. Name: "+ fileName);
 }
 @Override
 public void playMp4(String fileName)
 {
    //do nothing
 }
}
```

Adapter Design Pattern - Implementation

Step 2

Mp4Player.java

```
public class Mp4Player implements AdvancedMediaPlayer
{
 @Override
 public void playVlc(String fileName) {
 //do nothing
 }
 @Override
 public void playMp4(String fileName) {
 System.out.println("Playing mp4 file. Name: "+ fileName);
 }
}
```

Adapter Design Pattern - Implementation

Step 3

Create adapter class implementing the *MediaPlayer* interface.

MediaAdapter.java

```
public class MediaAdapter implements MediaPlayer {
 AdvancedMediaPlayer advancedMusicPlayer;
 public MediaAdapter(String audioType){
  if(audioType.equalsIgnoreCase("vlc")){
    advancedMusicPlayer = new VlcPlayer();
  }
  else if (audioType.equalsIgnoreCase("mp4")){
   advancedMusicPlayer = new Mp4Player();
  }
 }
 @Override public void play(String audioType, String fileName) {
   if(audioType.equalsIgnoreCase("vlc"))
   {
     advancedMusicPlayer.playVlc(fileName);
   }
   else if(audioType.equalsIgnoreCase("mp4")){
     advancedMusicPlayer.playMp4(fileName);
   }
  }
}
```

Adapter Design Pattern - Implementation

Step 3

Create adapter class implementing the *MediaPlayer* interface.

MediaAdapter.java

```
public class MediaAdapter implements MediaPlayer {
 AdvancedMediaPlayer advancedMusicPlayer;
 public MediaAdapter(String audioType){
  if(audioType.equalsIgnoreCase("vlc")){
    advancedMusicPlayer = new VlcPlayer();
  }
  else if (audioType.equalsIgnoreCase("mp4")){
   advancedMusicPlayer = new Mp4Player();
  }
 }
 @Override public void play(String audioType, String fileName) {
   if(audioType.equalsIgnoreCase("vlc"))
   {
     advancedMusicPlayer.playVlc(fileName);
   }
   else if(audioType.equalsIgnoreCase("mp4")){
     advancedMusicPlayer.playMp4(fileName);
   }
  }
}
```

Adapter Design Pattern - Implementation

Step 4

Create concrete class implementing the *MediaPlayer* interface.

AudioPlayer.java

```
public class AudioPlayer implements MediaPlayer {
 MediaAdapter mediaAdapter;
 @Override
 public void play(String audioType, String fileName) {
 //inbuilt support to play mp3 music files
 if(audioType.equalsIgnoreCase("mp3")){
  System.out.println("Playing mp3 file. Name: " + fileName);
 }
 //mediaAdapter is providing support to play other file formats
 else if(audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")){
  mediaAdapter = new MediaAdapter(audioType);
  mediaAdapter.play(audioType, fileName);
 }
 else{
  System.out.println("Invalid media. " + audioType + " format not supported");
 } } }
```

Adapter Design Pattern - Implementation

Step 5

Use the AudioPlayer to play different types of audio formats.

AdapterPatternDemo.java

```
public class AdapterPatternDemo {
public static void main(String[] args) {
   AudioPlayer audioPlayer = new AudioPlayer();
   audioPlayer.play("mp3", "beyond the horizon.mp3");
   audioPlayer.play("mp4", "alone.mp4");
   audioPlayer.play("vlc", "far far away.vlc");
   audioPlayer.play("avi", "mind me.avi");
  }
}
```

End of Lecture
Any Question ?