# Adapter Pattern

Lutfun Nahar Lota

Lecturer, CSE

# Adapter Pattern

- Adapter pattern works as a bridge between two incompatible interfaces
- Also known as wrapper
- This type of design pattern comes under structural pattern
- 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

# Adapter Pattern

Problem:

We want to reuse the capability of an existing interface that is incompatible

# Adapter Pattern

- It follows Robert C. Martin's Dependency Inversion Principle and enables you to reuse an existing class even so it doesn't implement an expected interface.

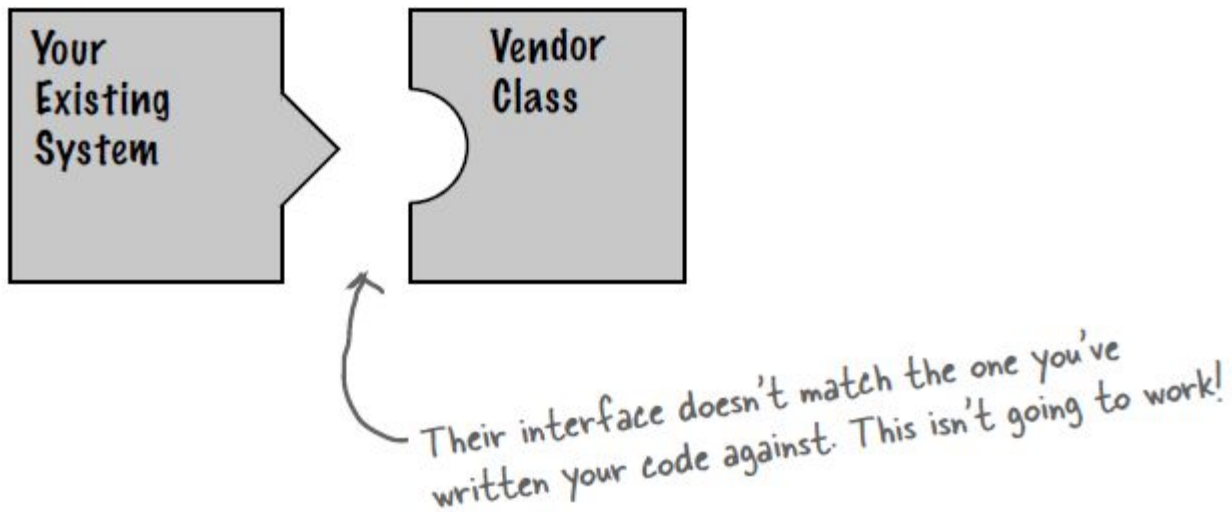
Dependency Inversion Principle

> *"high level modules should not depend on low level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions."*

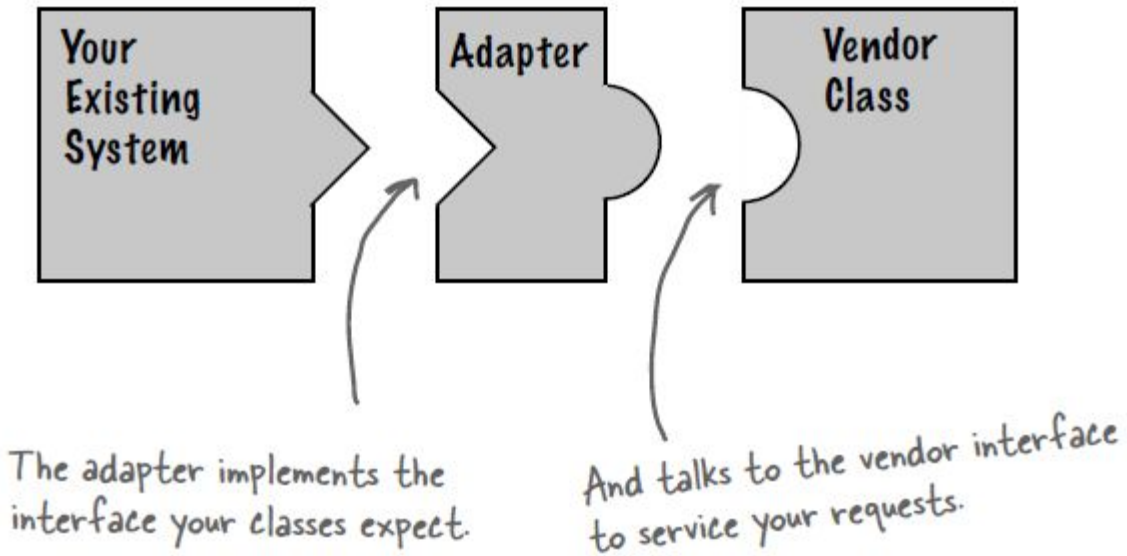
# Homework

- How Adapter pattern ensures ‘Dependency Inversion Principle’?
- Difference between inheritance and composition
- Advantages and disadvantages of Adapter

# Adapter Pattern



# Adapter Pattern



# Adapter Pattern

*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 Pattern

*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
   }
}
```

*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 Pattern

*MediaAdapter.java*

```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 Pattern

*AudioPlayer.java*

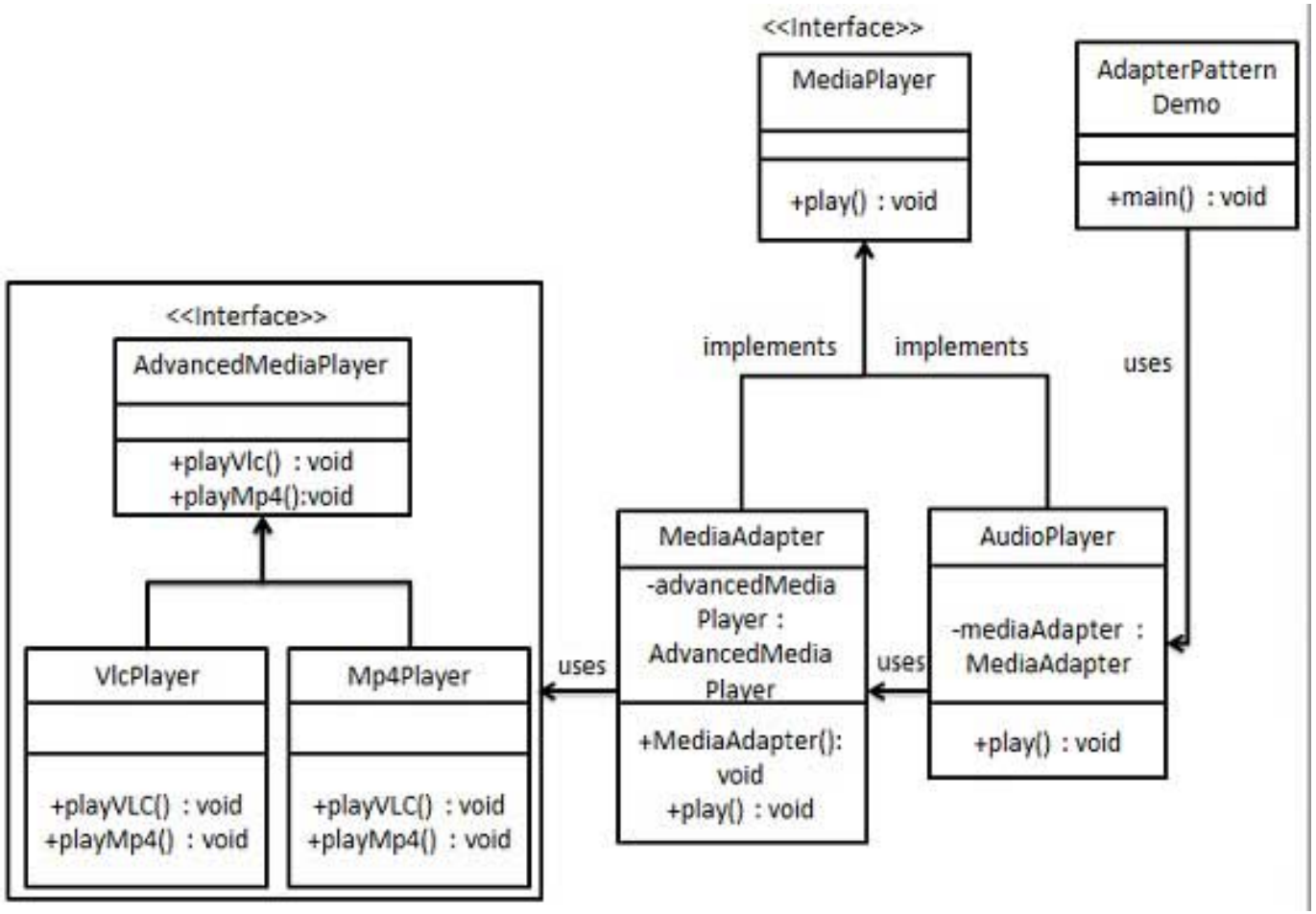
```
public class AudioPlayer implements MediaPlayer {
   MediaAdapter mediaAdapter;

   @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("m
         mediaAdapter = new MediaAdapter(audioType);
         mediaAdapter.play(audioType, fileName);
      }

      else{
         System.out.println("Invalid media. " + audioType + " format not support
      }
   }
}
```

# Adapter Pattern

*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");
   }
}
```

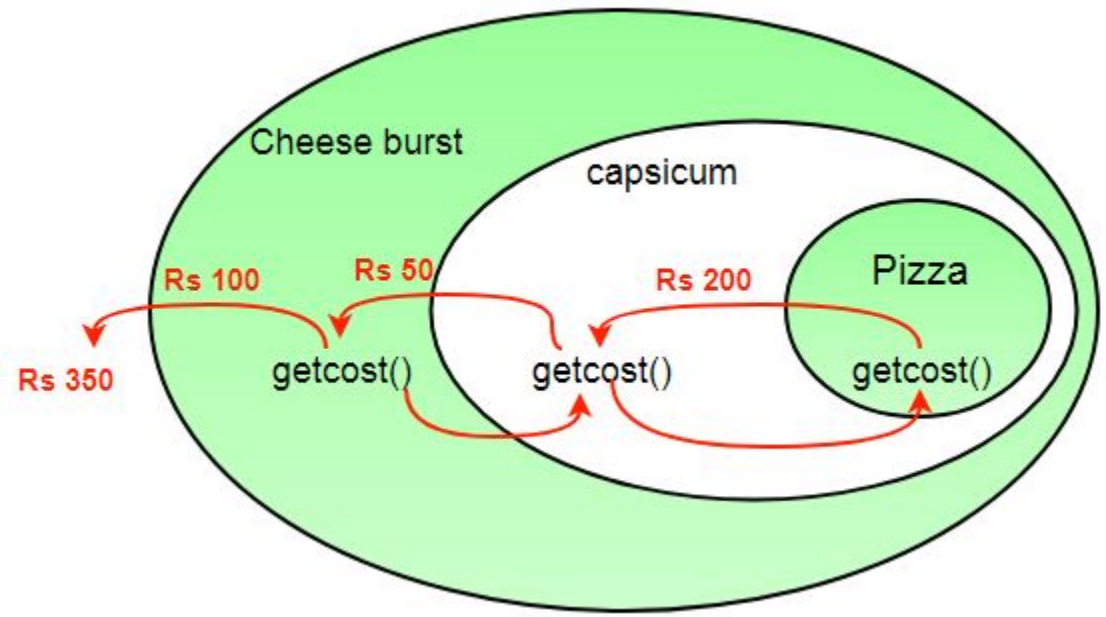
# UML Diagram for Adapter Pattern



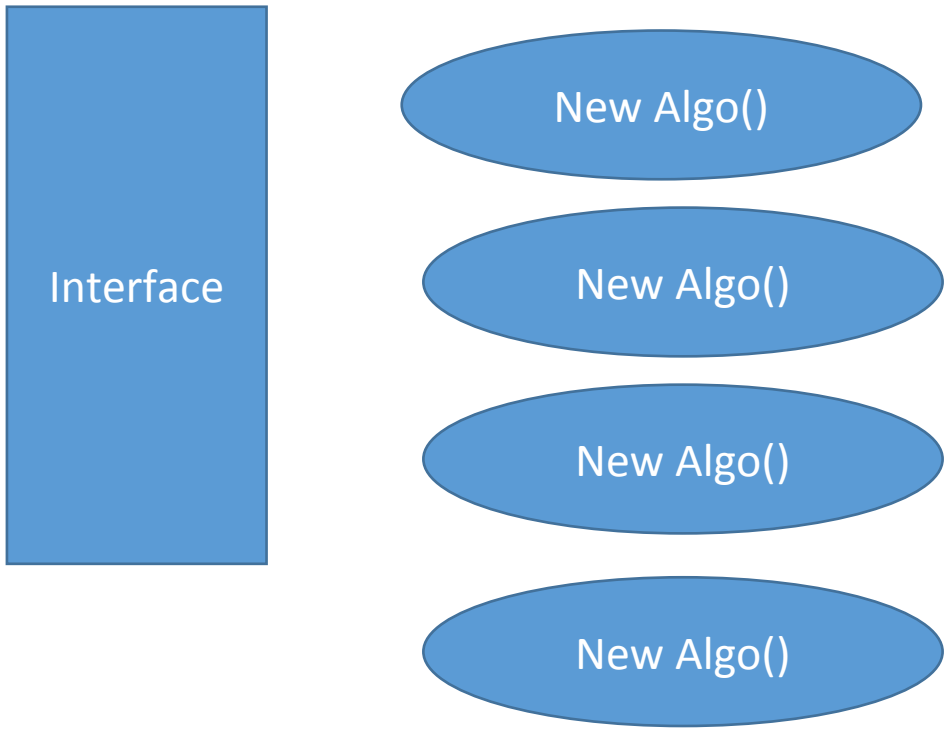
# Decorator Pattern

- Decorator pattern allows a user to add new functionality to an existing object without altering its structure.
- This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.
- This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

# Decorator Pattern



# Strategy



# Decorator Pattern

*Shape.java*

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

*Rectangle.java*

```
public class Rectangle implements Shape {

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

*Circle.java*

```
public class Circle implements Shape {

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

# Decorator Pattern

*ShapeDecorator.java*

```
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }

   public void draw(){
      decoratedShape.draw();
   }
}
```

# Decorator Pattern

*RedShapeDecorator.java*

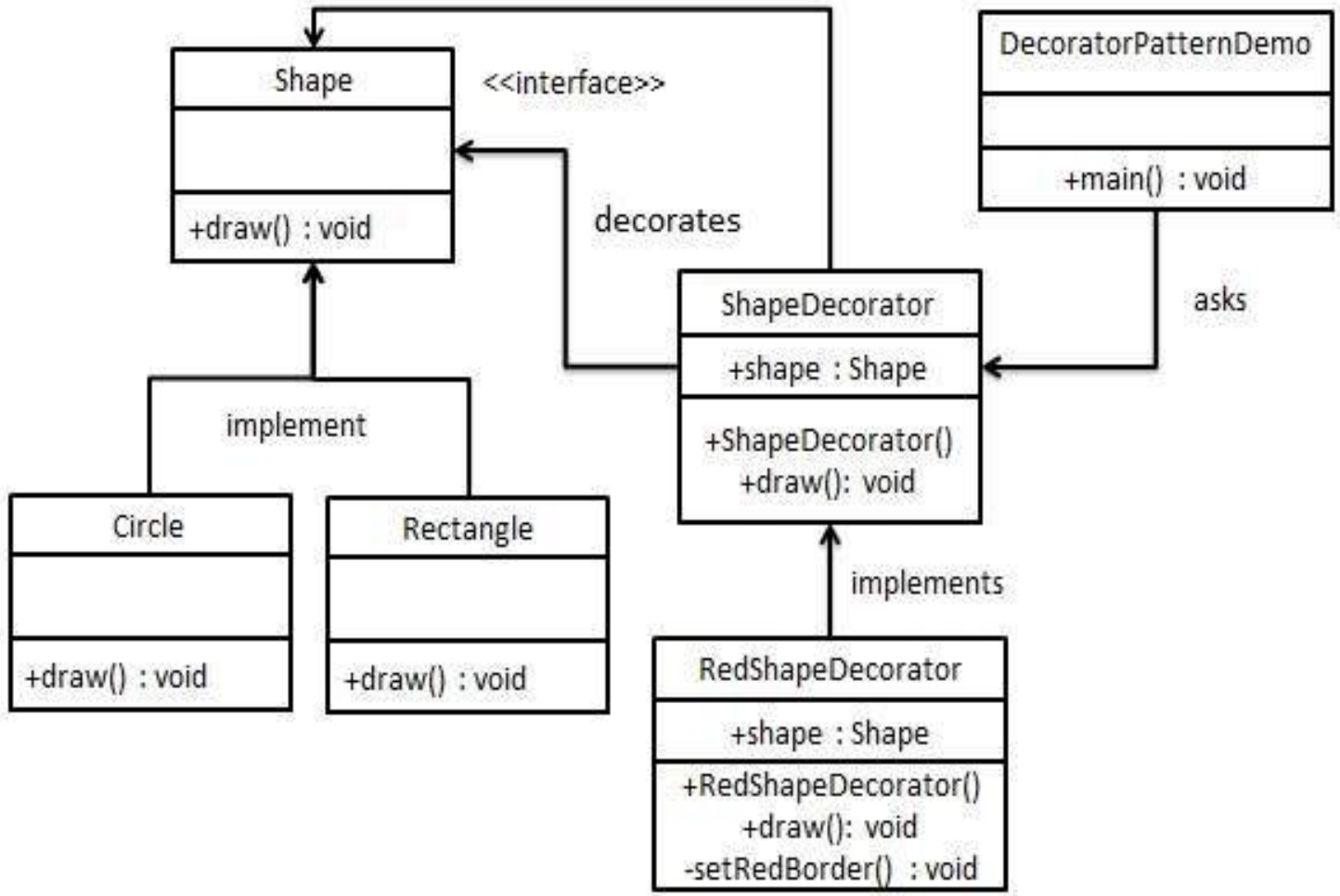
```
public class RedShapeDecorator extends ShapeDecorator {

   public RedShapeDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape){
      System.out.println("Border Color: Red");
   }
}
```

# Decorator Pattern

*DecoratorPatternDemo.java*

```
public class DecoratorPatternDemo {
   public static void main(String[] args) {

      Shape circle = new Circle();

      Shape redCircle = new RedShapeDecorator(new Circle());

      Shape redRectangle = new RedShapeDecorator(new Rectangle());
      System.out.println("Circle with normal border");
      circle.draw();

      System.out.println("\nCircle of red border");
      redCircle.draw();

      System.out.println("\nRectangle of red border");
      redRectangle.draw();
   }
}
```

# UML Diagram of Decorator Pattern



# HW

- Which Pattern is Similar to Decorator?

# HW

- Which Pattern is Similar to Decorator?

Ans: Strategy

- Why decorator seems similar to each other?
- What are the differences?
- Are they (strategy and Decorator) can be used alternatively?