Experiment#		Student ID	
Date		Student Name	

4. Structural Design Patterns I

Aim/Objective: To analyse the implementation of Adapter and Decorator Design Patterns for the real-time scenario.

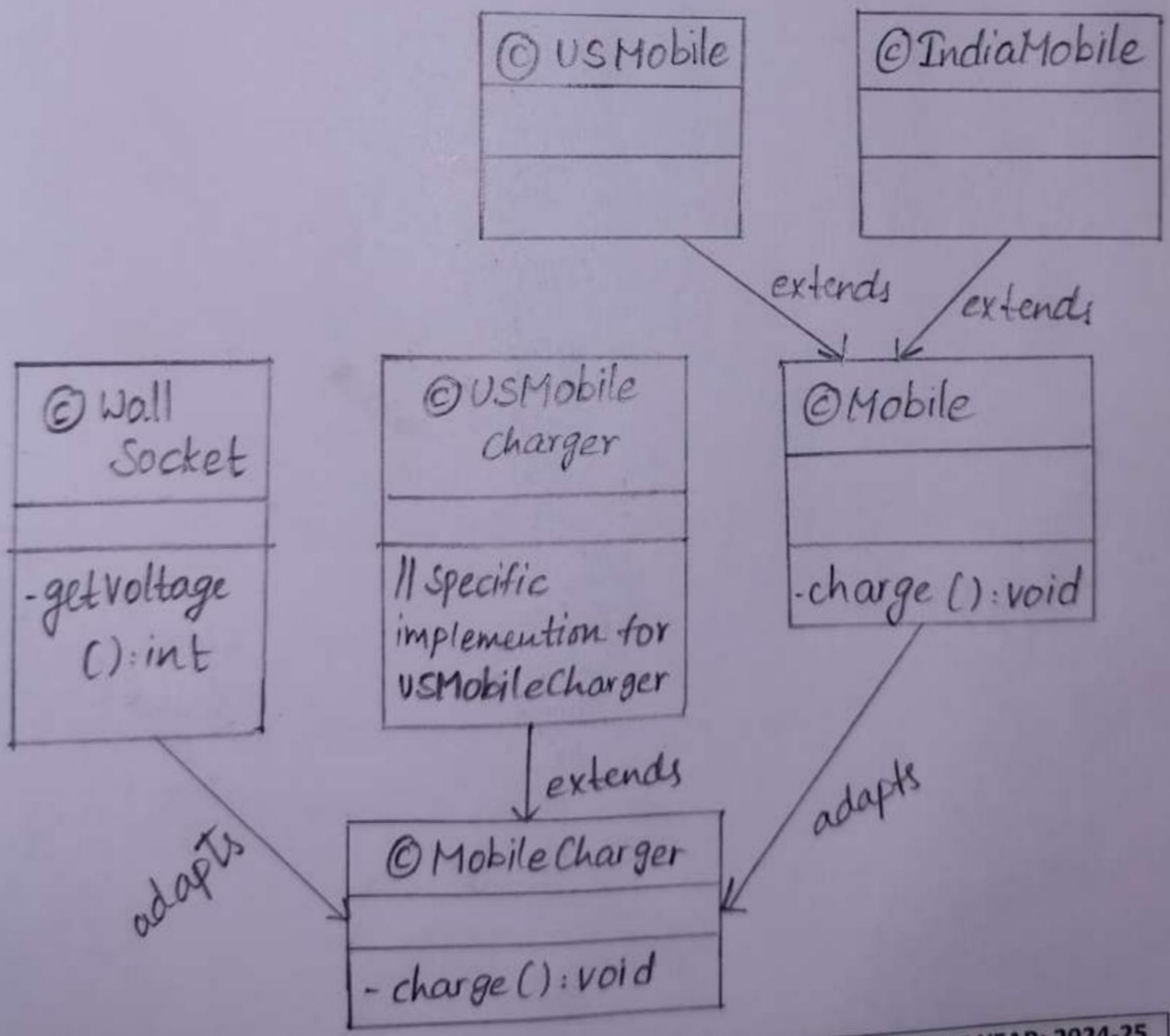
Description: The student will understand the concept of Adapter and Decorator Design Patterns.

Pre-Requisites: Classes and Objects in JAVA

Tools: Eclipse IDE for Enterprise Java and Web Developers

Pre-Lab:

1) Draw the UML Relationship Diagram for Adapter Design Pattern for Mobile charger adapter scenario. Note: Mobile battery needs 3 volts to charge but the normal socket produces either 120V (US) or 240V (India). So, the mobile charger works as an adapter between mobile charging socket and the wall socket.



Experiment#		Student ID	
Date		Student Name	

In-Lab:

1) Develop a music streaming application that can play music from various sources, such as local files, online streaming services, and radio stations. Implement the following design patterns to achieve this functionality:

Adapter Pattern: Adapt different music sources to a common interface.

Decorator Pattern: Add additional features (e.g., equalizer, volume control) to the music playback.

Your task is to design and implement the application using these design patterns to ensure flexibility, scalability, and maintainability.

Procedure/Program:

```
public interface MusicPlayer {
    void play();
    void stop(); }
public class LocalMusicPlayer {
public void playLocalMusic () {
System.out.println ("Playing local music...");
} public void stopLocalMusic () {
System.out.println ("stopping local music...");
  }
}
  Public class OnlineStreamingPlayer {
Public void playonlineMusic () {
System.out.println ("Playing online music...");
} public void stoponlineMusic () {
```

```
System.out.println("stopping online music...");
}
}
public class RadioStationPlayer {
  public void playRadio(){
System.out.println("Playing radio station...");
} public void stopRadio(){
System.out.println("Stopping radio station...");
  }
}
public class LocalMusicPlayerAdapter implements
                                    MusicPlayer {
private LocalMusicPlayer localPlayer;
Public LocalMusicPlayerAdapter(LocalMusicPlayer
                                    localPlayer) {
  this.localPlayer = localPlayer;
} public void play() {
  localplayer.stopLocalMusic(); }
}
public class OnlinestreamingAdapter implements
```

```
MusicPlayer {
private onlineStreamingPlayer onlinePlayer;
public onlineStreamingPlayerAdapter (OnlineStreaming
        Player onlinePlayer) {
  this.onlinePlayer = onlinePlayer;
}
  public void play() {
  onlinePlayer.PlayonlineMusic(); }
  public void stop() {
   onlinePlayer.stopOnlineMusic();
  } }
Public class RadioStationPlayerAdapter implements
                                MusicPlayer {
  private RadioStationPlayer radioPlayer;
public RadioStationPlayerAdapter (RadioStationPlayer
                radioPlayer) {
  this.radioPlayer = radioPlayer;
  } public void play() {
  }  radioPlayer.playRadio();
```

```
public void stop () {
    radioPlayer. stopRadio (); }
Public abstract class MusicPlayerDecorator
                    implements MusicPlayer {
  protected MusicPlayer decoratedPlayer;
Public MusicPlayerDecorator (MusicPlayer
                              decoratedPlayer) {
this.decoratedPlayer = decoratedPlayer;
} public void play () {
    decoratedPlayer. play ();
} public void stop () {
    decoratedPlayer. stop ();
} }
public class EqualizerDecorator extends MusicPlayer
                                        Decorator {
  public EqualizerDecorator (MusicPlayer decoratedPlayer) {
    super (decoratedPlayer);
} public void play () {
    super. play ();
```

```
setEqualizer();
} private void setEqualizer() {
System.out.println("Applying equalizer settings...");
}}
public class VolumeControlDecorator extends
                    MusicPlayerDecoator {
public VolumeControlDecorator(MusicPlayer decorated
                                    player) {
  super(decoratedPlayer);
} public void play(){
  super.play();
  setVolumeControl();
} private void setVolumeControl(){
System.out.println("Adjusting volume...");
  }
}
```

Experiment#		Student ID	
Date		Student Name	

✓ **Data and Results:**

Playing with enhanced local player:

Playing local music...

Applying equalizer settings...

Adjusting volume...

✓ **Analysis and Inferences:**

The output demonstrates how the application handles different music sources using the Adapter Pattern and enhances functionality with the Decorator Pattern.

Experiment#		Student ID	
Date		Student Name	

VIVA-VOCE Questions (In-Lab)

1) State how the adapter pattern allows the objects with incompatible interfaces to collaborate.

Adapter pattern enables collaboration between objects with incompatible interfaces by providing a wrapper that translates one interface to another.

2) Discuss about Adapter Pattern by considering the scenario of a USB to Ethernet adapter.

In the scenario of USB to Ethernet adapter, the adapter pattern allows devices with incompatible interfaces (USB and Ethernet) to work together seamlessly by translating communication between them.

3) Illustrate the difference between Class Adapter and Object Adapter.

Class Adapter extends the target class and adapts the adaptee through inheritance. Object Adapter uses composition, containing the adaptee as a member, allowing for more flexible adaptation.

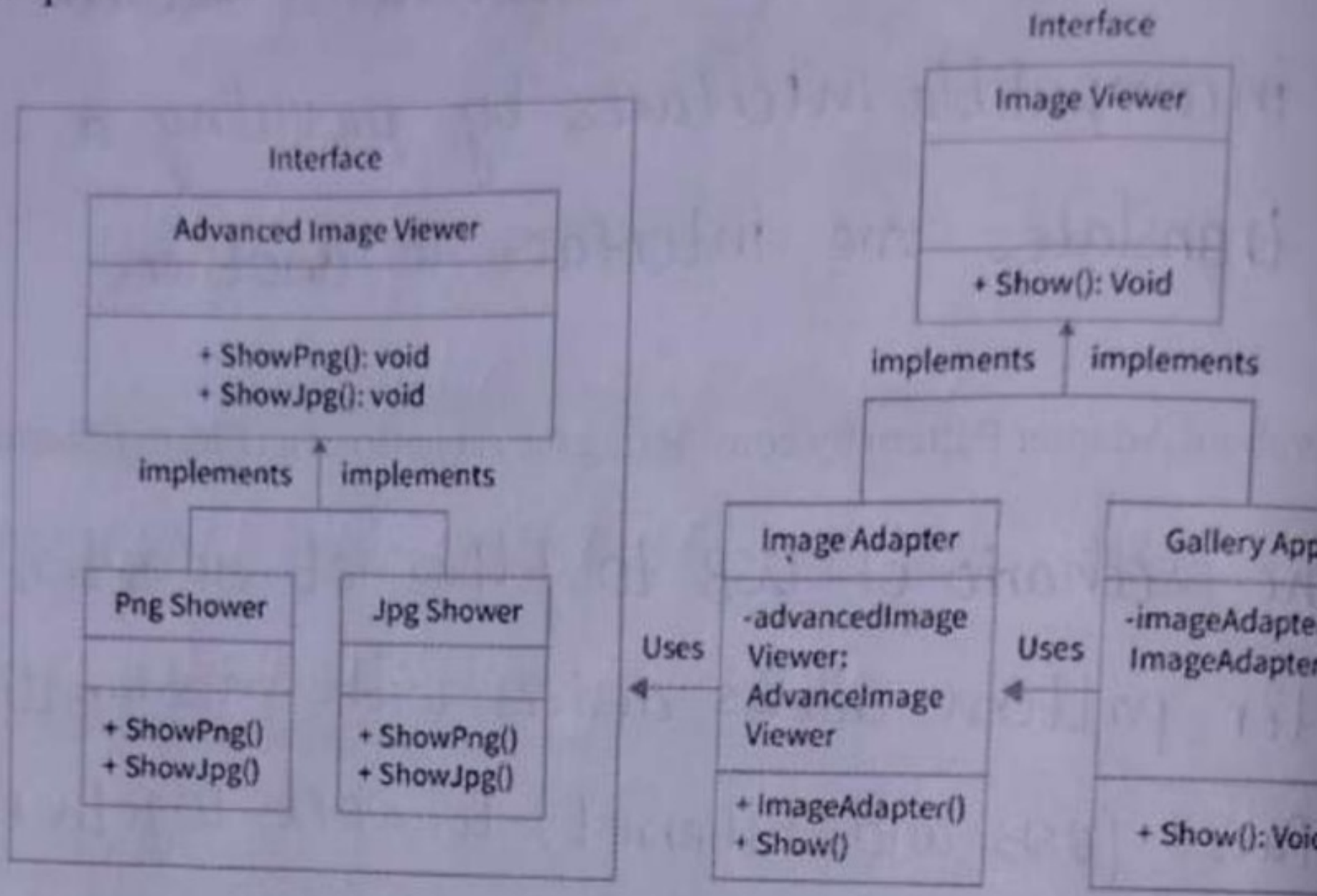
4) Which design problems does the adapter design pattern solve?

Adapter pattern solves the problem of incompatible interfaces b/w classes or objects, enabling them to collaborate and work together without modification.

Post-Lab:

1) Implement the below depicted UML Diagram in Java Program wit

Adapter Pattern



Procedure/Program:

```
interface AdvancedImageViewer {
    void showPng();
    void showJpg();
}
interface ImageViewer {
    void show();
}
class PngShower implements AdvancedImageViewer
{
```

```
public void showPng(){
System.out.println("Displaying PNG image");
} public void showJpg(){
System.out.println("Displaying JPG image");
    }
}
class JpgShower implements AdvancedImageViewer{
  public void showPng(){
System.out.println("Cannot display PNG image in
                    JPG view");
  }
    public void showJpg(){
System.out.println("Displaying JPG image");
    }
}
interface ImageAdapter extends ImageViewer{ }
class ImageAdapterImp implements ImageAdapter{
private AdvancedImageViewer advancedImageViewer;
public ImageAdapterImp(AdvancedImageViewer
                    advancedImageViewer){
```

```
this.advancedImageViewer = advancedImageViewer;
} public void show() {
    advancedImageViewer.showPng();
    }
} class GalleryApp {
  private ImageAdapter imageAdapter;
public GalleryApp(ImageAdapter imageAdapter){
    this.imageAdapter = imageAdapter;
}
 public void displayImage(){
     imageAdapter.show();
    }
 }
 public class Main {
public static void main(String[] args) {
AdvancedImageViewer pngShower = new PngShower();
AdvancedImageViewer jpgShower=new JpgShower();
ImageAdapter pngAdapter=new ImageAdapterImp
(pngShower);
```

```
GalleryApp galleryApp = new GalleryApp(pngAdapter);
    galleryApp.displayImage();
ImageAdapter jpgAdapter=new ImageAdapterImp
                              (jpgShower);
    galleryApp=new GalleryApp(jpgAdapter);
    galleryApp.displayImage();
}
```

Experiment#		Student ID	
Date		Student Name	

✓ **Data and Results:**

Displaying PNG image

Cannot display PNG image in JPG viewer

✓ **Analysis and Inferences:**

The code uses Adapter pattern to allow a 'GalleryApp' to handle multiple image formats seamlessly. It achieves flexibility, encapsulation and maintainability by abstracting the specifics of image handling through adapters.

Evaluator Remark (if Any):	
	Marks Secured: _____out of 50

Evaluator MUST ask Viva-voce prior to signing and posting marks for each exp