Experiment#		Student ID	
Date		Student Name	@KLWKS_BOT THANOS

2. Structural Design Patterns

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.

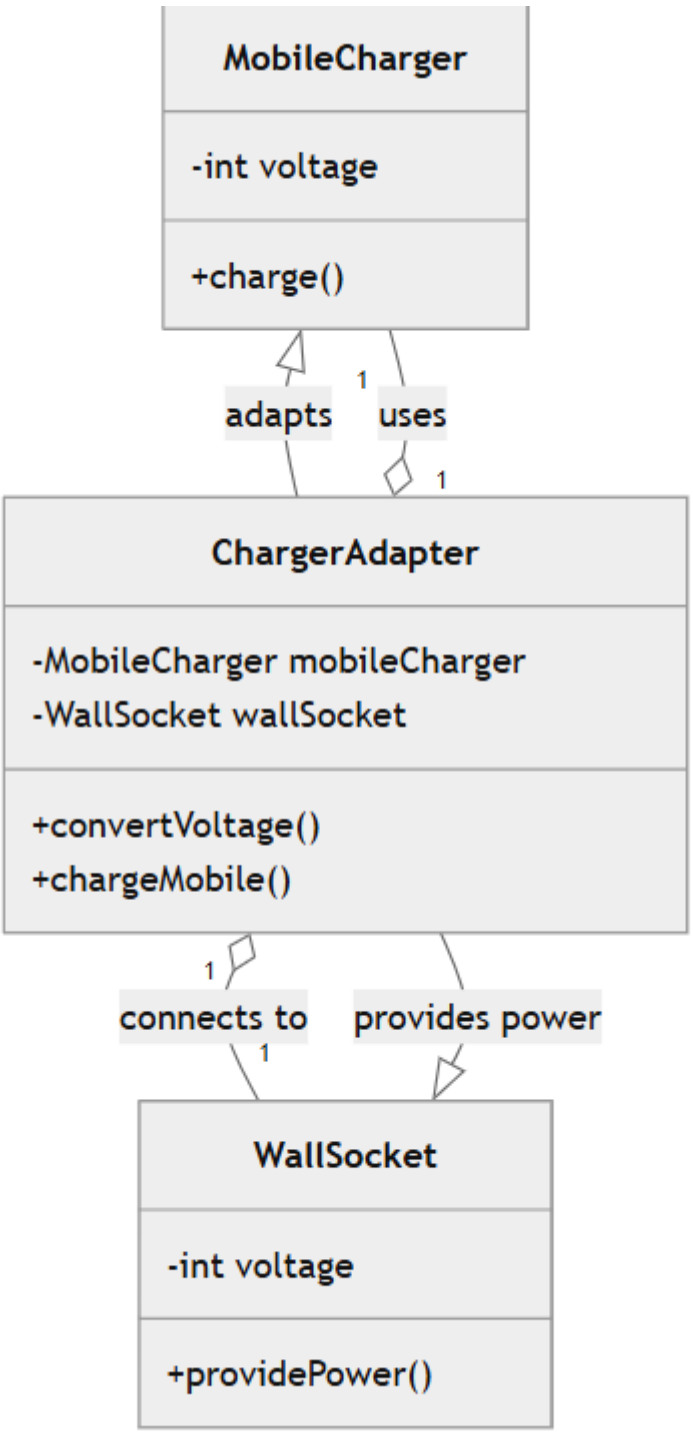
Pre-Requisites: Classes and Objects in JAVA

Tools: Eclipse IDE for Enterprise Java and Web Developers

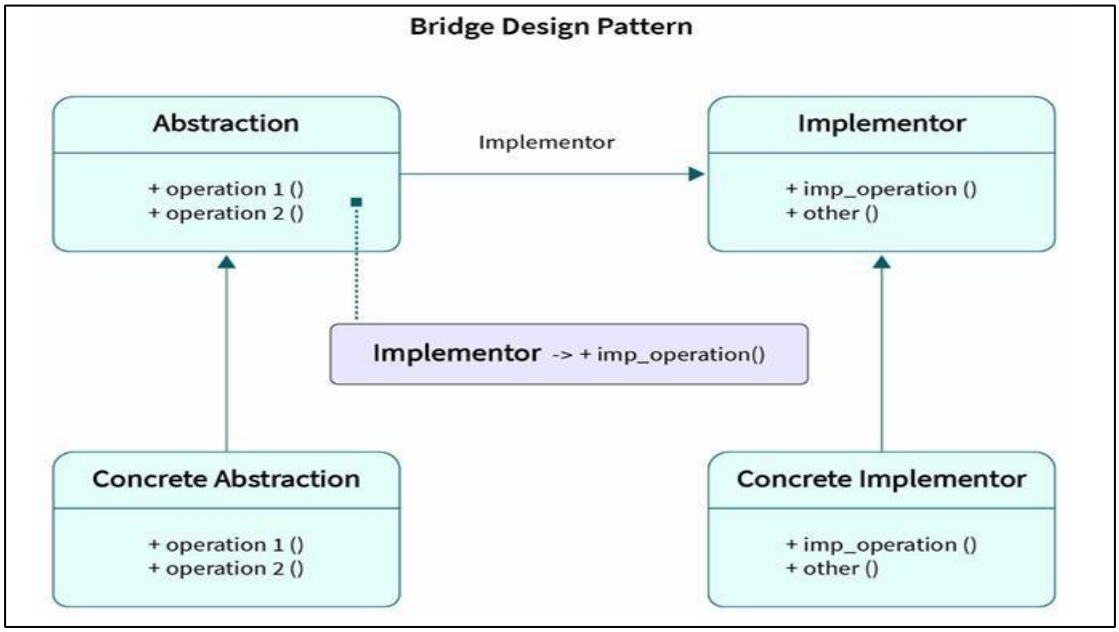
Pre-Lab:

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.

ANSWER


MobileCharger
-int voltage
+charge()
adapts
uses
1
1
ChargerAdapter
-MobileCharger mobileCharger
-WallSocket wallSocket
+convertVoltage()
+chargeMobile()
1
connects to
1
provides power
WallSocket
-int voltage
+providePower()

2. Discuss about the below mentioned structure of Bridge Design Pattern.



ANSWER

Abstraction (Abstract Class)

- It is the core of the bridge design patterns and defines the interface for the abstraction (such as the GUI i.e., the **Graphical User Interface** Layer of an App`).
- This layer isn't supposed to do any real work on its own i.e., it is the platform-independent layer.
- It contains a reference to the implementor.
- Abstraction defines the interface which is used by the client i.e., it is a platform-independent (or abstract) model which is used to provide the functionality to the client via the platform-specific implementations.

Implementor (Interface)

- The implementor provides the actual function which is described in the abstraction. For example, if we are creating a Banking system application, the abstraction may define a function to withdraw some amount of money from a bank account. Here, the implementor will provide a list of operations (such as checking if the account has enough balance or not) that may be useful to carry out the cash withdrawal from the bank account.
- It also defines the interface for implementation classes.

Concrete Abstractions (Normal Class)

- It extends the abstract interface defined by Abstraction.
- The concrete abstractions provide any specialized operation that may be required by the client. For example, the Abstraction may represent and define a basic user interface and its concrete abstraction may define a hand-held device's specialized user interface.

Concrete Implementations (Normal Class)

- It implements the Implementor.
- It provides the actual source code to implement the operations and functionality defined by the implementor interface.

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.

Bridge Pattern: Decouple the music playback functionality from the music source.

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 class MusicStreamingApp {

   interface IMusicSource {
      void play();
      void stop();
   }

   static class LocalMusic implements IMusicSource {
      private String file;

      public LocalMusic(String file) {
         this.file = file;
      }

      public void play() {
         System.out.println("Playing local music from " + file);
      }

      public void stop() {
         System.out.println("Stopping local music playback");
      }
   }

   static class StreamingMusic implements IMusicSource {
      private String service, track;

      public StreamingMusic(String service, String track) {
         this.service = service;
         this.track = track;
      }
```

```
    public void play() {
        System.out.println("Playing track " + track + " from " + service);
    }

    public void stop() {
        System.out.println("Stopping track from " + service);
    }
}

static class RadioMusic implements IMusicSource {
    private String station;

    public RadioMusic(String station) {
        this.station = station;
    }

    public void play() {
        System.out.println("Playing radio from " + station);
    }

    public void stop() {
        System.out.println("Stopping radio playback");
    }
}

static class MusicPlayer {
    private IMusicSource source;

    public void play(IMusicSource source) {
        this.source = source;
        source.play();
    }

    public void stop() {
        if (source != null) source.stop();
    }
}

static class VolumeControl extends MusicPlayer {
    private int volume;

    public VolumeControl(int volume) {
        this.volume = volume;
    }

    @Override
    public void play(IMusicSource source) {
        System.out.println("Setting volume to " + volume);
        super.play(source);
    }
}
```

```
    static class Equalizer extends MusicPlayer {
        private String settings;

        public Equalizer(String settings) {
            this.settings = settings;
        }

        @Override
        public void play(IMusicSource source) {
            System.out.println("Applying equalizer settings: " + settings);
            super.play(source);
        }
    }

    public static void main(String[] args) {
        IMusicSource localMusic = new LocalMusic("track1.mp3");
        IMusicSource streamingMusic = new StreamingMusic("Spotify",
"track123");
        IMusicSource radioMusic = new RadioMusic("http://radio.com");

        MusicPlayer player = new Equalizer("Bass Boost");
        player = new VolumeControl(75);

        System.out.println("Playing Local Music:");
        player.play(localMusic);
        player.stop();

        System.out.println("\nPlaying Streaming Music:");
        player.play(streamingMusic);
        player.stop();

        System.out.println("\nPlaying Radio Music:");
        player.play(radioMusic);
        player.stop();
    }
}
```

OUTPUT

Playing Local Music:
Setting volume to 75
Applying equalizer settings: Bass Boost
Playing local music from track1.mp3
Stopping local music playback

Playing Streaming Music:
Setting volume to 75
Applying equalizer settings: Bass Boost
Playing track track123 from Spotify
Stopping track from Spotify

Playing Radio Music:
Setting volume to 75
Applying equalizer settings: Bass Boost
Playing radio from http://radio.com
Stopping radio playback

- ✓ **Data and Results:**

Data

Music sources include local files, streaming services, and radio stations.

Result

Music playback works seamlessly across all sources with added features.

- ✓ **Analysis and Inferences:**

Analysis

Adapters, bridge, and decorators ensure flexibility and modular design.

Inferences

Design patterns enhance scalability, maintainability, and code reusability effectively.

VIVA-VOCE Questions (In-Lab)

1) State at which situation that we are in need of Bridge Design Pattern.

- When you need to separate abstraction from implementation, allowing both to evolve independently.
- When you have multiple hierarchies that need to vary independently (e.g., devices and their controls).

2) Illustrate the difference between Decorator and Bridge pattern.

Aspect	Decorator	Bridge
Purpose	Adds functionality to an object dynamically.	Decouples abstraction from implementation.
Use Case	Augmenting object behavior at runtime.	Managing multiple abstraction/implementation pairs.

3) Discuss the Pros and Cons of Facade Design Pattern.

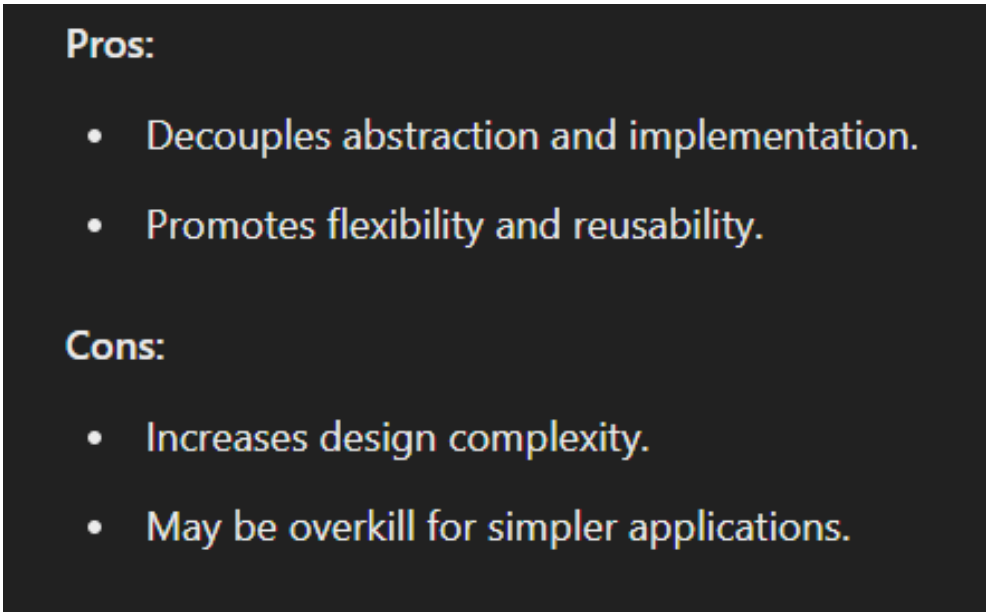
Pros:

- Simplifies complex systems.
- Reduces coupling between clients and subsystems.
- Provides a unified interface.

Cons:

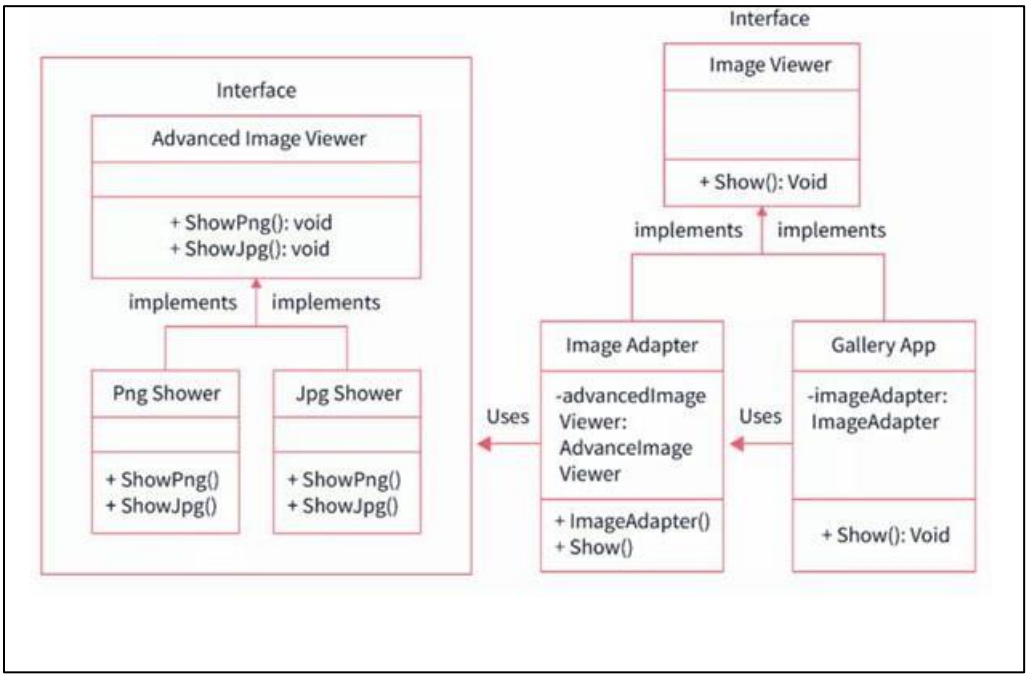
- Can become a "god object" if too large.
- Might limit flexibility by hiding subsystem details.

4) Discuss the Pros and Cons of Bridge Design Pattern.



Post-Lab:

1) Implement the below depicted UML Diagram in Java Program with respective to Adapter Pattern



Procedure/Program:

```
public class AdapterPatternDemo {
  interface ImageViewer {
    void show(String imageFormat, String fileName);
  }

  interface AdvancedImageViewer {
    void showPng(String fileName);
    void showJpg(String fileName);
  }

  static class PngShower implements AdvancedImageViewer {
    @Override
    public void showPng(String fileName) {
      System.out.println("Showing png file. Name: " + fileName);
    }

    @Override
```

```
public void showJpg(String fileName) {}

 }

static class JpgShower implements AdvancedImageViewer {
   @Override
   public void showPng(String fileName) {}

   @Override
   public void showJpg(String fileName) {
      System.out.println("Showing jpg file. Name: " + fileName);
   }
}

static class ImageAdapter implements ImageViewer {
   AdvancedImageViewer advancedImageViewer;

   public ImageAdapter(String imageFormat) {
      if (imageFormat.equalsIgnoreCase("png")) {
         advancedImageViewer = new PngShower();
      } else if (imageFormat.equalsIgnoreCase("jpg")) {
         advancedImageViewer = new JpgShower();
      }
   }

   @Override
   public void show(String imageFormat, String fileName) {
      if (imageFormat.equalsIgnoreCase("png")) {
         advancedImageViewer.showPng(fileName);
      } else if (imageFormat.equalsIgnoreCase("jpg")) {
         advancedImageViewer.showJpg(fileName);
      }
   }
}

static class GalleryApp implements ImageViewer {
   ImageAdapter imageAdapter;

   @Override
```

```
    public void show(String imageFormat, String fileName) {

        if (imageFormat.equalsIgnoreCase("jpeg")) {
            System.out.println("Showing jpeg file. Name: " + fileName);
        } else if (imageFormat.equalsIgnoreCase("png") ||
imageFormat.equalsIgnoreCase("jpg")) {
            imageAdapter = new ImageAdapter(imageFormat);
            imageAdapter.show(imageFormat, fileName);
        } else {
            System.out.println("Invalid image. " + imageFormat + " format not
supported");
        }
    }
  }

  public static void main(String[] args) {
    GalleryApp gallery = new GalleryApp();
    gallery.show("jpeg", "naruto.jpeg");
    gallery.show("png", "sasuke.png");
    gallery.show("jpg", "jiraya.jpg");
    gallery.show("gif", "sakura.gif");
  }
}
```

OUTPUT

```
Showing jpeg file. Name: naruto.jpeg
Showing png file. Name: sasuke.png
Showing jpg file. Name: jiraya.jpg
Invalid image. gif format not supported
```

Experiment#		Student ID	
Date		Student Name	@KLWKS_BOT THANOS

- ✓ **Data and Results:**

Data

The dataset contains images of various formats for testing.

Result

JPEG, PNG, and JPG formats were successfully displayed; GIF failed.

- ✓ **Analysis and Inferences:**

Analysis

Adapter pattern bridges format compatibility effectively in image viewing.

Inferences

The design pattern ensures extensibility and format-specific handling seamlessly.

Evaluator Remark (if Any):	**Marks Secured _____ out of 50**
	Signature of the Evaluator with Date

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