INDIAN INSTITUTE OF INFORMATION TECHNOLOGY NAGPUR

Software Architecture Mini Project

Implementation of Design Patterns For

Emotion Based Music Player

Raj Aryan	BT19CSE043
Sarthak Gupta	BT19CSE054
Aditi Sahu	BT19CSE057
Aseem Ranjan	BT19CSE085
Krish Rustagi	BT19CSE089
Naveen Rathore	BT19CSE117

Creational Pattern

Factory pattern =>

A Factory Pattern or Factory Method Pattern says that just **define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.** In other words, subclasses are responsible to create the instance of the class.

```
public MediaAdapter(String audioType){
    //Implementing factory pattern to create instances of VlcPlayer and Mp4Player

    if(audioType.equalsIgnoreCase("vlc") ){
      advancedMusicPlayer = new VlcPlayer();

    }else if (audioType.equalsIgnoreCase("mp4")){
      advancedMusicPlayer = new Mp4Player();
    }
  }
```

```
package adapterpattern;
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
  }
}
```

```
package adapterpattern;
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);
  }
```

Builder pattern

Builder Pattern says that construct a complex object from simple objects using step-by-step approach.

```
public class FileLoader {
   String name;
   int dur;
   String artist;
   String movieName;
   //implementing builder pattern to create song file instance
   public FileLoader withName(String name) {
      this.name = name;
      return this;
    }

    public FileLoader withDur(int dur) {
       this.dur = dur;
       return this;
    }

    public FileLoader withArtist(String artist) {
       this.artist = artist;
       return this;
    }

    public FileLoader withMovieName(String movieName) {
       this.movieName = movieName;
       return this;
    }

   public File loadFile(String fileName) {
      this.name = fileName;
      return new File(fileName);
   }
}
```

In parent Class

```
        File musicFile = loader.withName("SongName").withArtist("artistName")
                               .withDur(100)
                               .withMovieName("Song Movie Name")
                               .loadFile(fileName);
```

Without builder pattern the calling of constructor will be like:

```
File musicFile =
          loader.laodFile("SongName","ArtistName",100,"SongMoviewName");
```

which can be less cleaner if we add more attributes in it

Structural Pattern

FACADE PATTERN

Without facade pattern, to play mp3 music file, the client has to understand the interfaces of all modules and understand the sequence in which it has to use the module before playing the music file.
But while using façade pattern, Client will deal with Music Client File only.

```
public class MusicClient{
   public static void main(String args[]) {
       MusicPlayerFacade player = new MusicPlayerFacade();
       System.out.println("calling player");
       player.play("mp3","Song Link");
   }
}
```

```
public class MusicPlayerFacade {
   Player player = new Player();
   // FileLoader loader = new FileLoader();
   FileLoader loader = new FileLoader();
   AudioPlayer audioPlayer = new AudioPlayer();

  //giving all responsibility to this single method
   public void play(String ext, String fileName) {
       System.out.println("Starting");
       File musicFile =
           loader.withName("SongName").withArtist("artistName")
                           .withDur(100)
                           .withMovieName("Song Movie Name")
                           .loadFile(fileName);

       PlayPauseFunctionality ppf = new PlayPauseFunctionality();
       ppf.playPause();
       //action listeners to be used in playpausefunctionality
       Decoder decoder = new Decoder();
       byte[] stream = decoder.decode(musicFile);
       audioPlayer.play(ext, fileName);
       player.init();
       player.startStreaming(stream);
       player.showTitle(fileName);
       player.showDuration(15);


   }
}
```

Adapter pattern

This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces.

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.

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

```
public class MediaAdapter implements MediaPlayer {

   AdvancedMediaPlayer advancedMusicPlayer;

   public MediaAdapter(String audioType){
    //Implementing factory pattern to create instances of VlcPlayer and Mp4Player

     if(audioType.equalsIgnoreCase("vlc") ){
       advancedMusicPlayer = new VlcPlayer();

     }else if (audioType.equalsIgnoreCase("mp4")){
       advancedMusicPlayer = new Mp4Player();
     }
   }
   // STRATEGY PATTERN
   @Override
   public void play(String audioType, String fileName) {

     if(audioType.equalsIgnoreCase("vlc")){
       advancedMusicPlayer.playVlc(fileName);
     }
     else if(audioType.equalsIgnoreCase("mp4")){
       advancedMusicPlayer.playMp4(fileName);
     }
   }
}
```

Behavioral Pattern

Strategy Pattern

A Strategy Pattern says that "defines a family of functionality, encapsulate each one, and make them interchangeable".

When the multiple classes differ only in their behaviors we use strategy pattern

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

```
public class MediaAdapter implements MediaPlayer {
   // STRATEGY PATTERN
   @Override
   public void play(String audioType, String fileName) {

      if(audioType.equalsIgnoreCase("vlc")){
         advancedMusicPlayer.playVlc(fileName);
      }
      else if(audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer.playMp4(fileName);
      }
   }
}
```

```
public class AudioPlayer implements MediaPlayer {
   MediaAdapter mediaAdapter;
   @Override
   public void play(String audioType, String fileName) {

      if(audioType.equalsIgnoreCase("mp3")){
         System.out.println("Playing mp3 file. Name: " + fileName);
      }

     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");
      }
   }
}
```

Command Pattern

A Command Pattern says that "encapsulate a request under an object as a command and pass it to invoker object. Invoker object looks for the appropriate object which can handle this command and pass the command to the corresponding object and that object executes the command".

It is used when you need parameterize objects according to an action perform and when you need to create and execute requests at different times.

```
public interface Command {
   public void execute();
}
```

```
public class PauseMusicCommand implements Command{
   private Song song;
   public PauseMusicCommand(Song song) {
       this.song = song;
   }
    @Override
    public void execute() {
       song.pause();
   }
}
```

```
public class PlayMusicCommand implements Command{
   private Song song;
   public PlayMusicCommand(Song song) {
      this.song = song;
   }
   @Override
   public void execute() {
      song.play();
   }
}
```

```
package playpausecommand;
public class PlayPauseFunctionality {
   public void playPause() {
      Song song = new Song();

      Command clickPlay = new PlayMusicCommand(song);
      Command clickPause = new PauseMusicCommand(song);
```

```
        clickPlay.execute();
        clickPause.execute();
    }
}
```

```
public class Song {
    public void play(){
     System.out.println("Song Played");
 }
 public void pause(){
    System.out.println("Song Paused");
 }
}
```