SOLID PRINCIPLES

JOEL C RAJU

SINGLE RESPONSIBILITY PRINCIPLE

The Single Responsibility Principle states that a class should have only one reason to change. Each class should have one responsibility or should encapsulate a single functionality

Example: The Entertainment class is responsible for storing information related to entertainment, such as ID, title, release date, etc. It follows SRP by having a single responsibility focused on managing entertainment data.

SINGLE RESPONSIBILITY PRINCIPLE

```java
public abstract class Entertainment{ {
 private int id;
 private String title;
 private String releaseDate;
 private String poster;
 private String overview;
 public Entertainment( int id, String title, String releaseDate, String poster, String overview){) {
  this.id = id;
  this.title = title;
  this.releaseDate = releaseDate;
  this.poster = poster;
  this.overview = overview;
}
}


}
```

Open/Close Principle

- The Open/Closed Principle states that a class should be open for extension but closed for modification. This means that the behavior of a class can be extended without altering its source code, promoting flexibility and ease of maintenance.
- **Example:** The Entertainment class is open for extension as new attributes or methods can be added in its subclasses (Movie and TvShow) without modifying the existing code.

Open/Close Principle

```
public class Movie extends Entertainment{
 private int duration;
 private double boxOffice;
 public Movie( int id, String title, String releaseDate, String poster, String overview, int duration, double
   boxOffice){
  super(id, title, releaseDate, poster, overview);
  this.duration = duration;
  this.boxOffice = boxOffice;
 }
}


}}
```

Liskov Substitution Principle

The Liskov Substitution Principle states that objects of a superclassshould be replaceable with objects of its subclasses without affecting the correctness of the program. Subclasses should behave in such a way that they don't violate the expected behavior of the superclass.

Example: The Entertainment class can accept instances of the base class (Entertainment) and its subclasses (Movie, TvShow) interchangeably, demonstrating that they adhere to LSP.

Liskov Substitution Principle

```
Entertainment entertainment = new Movie(movieId,movieTitle, releaseDate,
  posterUrl, overview, movieDuration, boxOffice);))
```

Interface Segregation Principle

The Interface Segregation Principle states that a class should not be forced to implement interfaces it does not use. In other words, it suggests breaking large interfaces into smaller, focused ones to avoid unnecessary dependencies.

Example: The TrailerDisplay interface segregates the methods into smaller interfaces (displayMainTrailer and displaySeasonTrailer), allowing classes to implement only the methods relevant to their context. The TvTrailerDisplay class implements only the methods it needs.

Interface Segregation Principle

```
// Interface Segregation Principle
public interface TrailerDisplay {
  void displayMainTrailer(Entertainment entertainment);
  void displaySeasonTrailer(TvShow tvshow);
}

public class TvTrailerDisplay implements TrailerDisplay {
  @Override
  public void displayMainTrailer(Entertainment entertainment) {
    // Display main trailer for TV show
  }

  @Override
  public void displaySeasonTrailer(TvShow tvshow) {
    // Display trailer of all seasons for TV show
  }
}
```

Dependency Inversion Principle

The Dependency Inversion Principle states that high-level modules (e.g., business logic) should not depend on low-level modules (e.g., data access details). Instead, both should depend on abstractions. This promotes flexibility and ease of testing.

Example: The TrailerDisplayer class depends on an abstraction (MainTrailer interface), and it can work with any implementation of that interface. This allows for flexibility and easier testing, adhering to DIP.

Dependency Inversion Principle

```
// Dependency Inversion Principle
public class TrailerDisplayer {
    private MainTrailer mainTrailer;

    public TrailerDisplayer(MainTrailer mainTrailer) {
        this.mainTrailer = mainTrailer;
    }

    public void displayTrailer(Entertainment entertainment) {
        mainTrailer.displayTrailer(entertainment);
    }
}
```