SOLID PRINCIPLES

Aneeka Geo

SINGLE RESPONSIBILITY PRINCIPLE

- A class should have only one responsibility, focusing on a single piece of functionality.
- Encourages code that is modular, enhancing readability and maintainability.
- Discourages classes from having multiple reasons to change.
- Promotes a design that is focused, adaptable, and less prone to errors.

Example:The Game class is responsible for representing the data associated with a game, such as its title, price, and genre.

It provides methods (getTitle, setPrice, getGenre, etc.) to access and modify the attributes of a game.

Its primary responsibility is to serve as a data container and encapsulate the state of a game object.

By adhering to the Single Responsibility Principle, the Game class remains focused on a single concern: managing the data related to a game. This makes the class more modular, easier to understand, and less prone to changes if modifications are needed in the future, promoting maintainability and flexibility in the codebase.

SINGLE RESPONSIBILITY PRINCIPLE

```
public class Game {
        private String title;
  private double price;
  private String genre;

        public String getTitle() {
                return title;
        }
        public void setTitle(String title) {
                this.title = title;
        }
        public double getPrice() {
                return price;
        }
        public void setPrice(double price) {
                this.price = price;
        }
public String getGenre() {
                return genre;
        }
        public void setGenre(String genre) {
                this.genre = genre;
        }
        public Game(String title, double price, String genre) {
                super();
                this.title = title;
                this.price = price;
                this.genre = genre;
        }
}
```

OPEN CLOSED PRINCIPLE

- Software entities (classes, modules, functions) should be open for extension.
- They should be closed for modification, ensuring existing, working code remains unchanged.
- New functionality can be added by introducing new code (classes or modules).
- Encourages the use of interfaces and abstract classes for flexibility and extensibility.
- Aims to minimize the risk of introducing bugs in existing code while efficiently accommodating changes and expansions.

Example:The GameValidator interface defines the validation contract, and two classes (DefaultGameValidator and ExtendedGameValidator) provide specific implementations with additional validation rules. The ExtendedGameValidator leverages the DefaultGameValidator for default validation, showcasing a form of composition in validation logic. The validation results are printed to the console.

OPEN CLOSED PRINCIPLE

```java
public interface GameValidator {
  void validate(Game game);
}
```

```java
public class DefaultGameValidator implements GameValidator {
  @Override
  public void validate(Game game) {
     if (game.getPrice() <= 0) {
        System.out.println("Price must be greater than zero.");
     }
  }
}
```

```java
public class ExtendedGameValidator implements
GameValidator {
  @Override
  public void validate(Game game) {
     if (game.getGenre() == null ||
game.getGenre().isEmpty()) {
        System.out.println("Genre must be specified.");
     }
     new DefaultGameValidator().validate(game);
  }
}
```

LISKOV SUBSTITUTION PRINCIPLE(LSP)

- States that objects of a superclass should be replaceable with objects of a subclass 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:In this code, the Liskov Substitution Principle (LSP) is illustrated through the use of a common interface called Discountable. This interface defines a shared method, calculateDiscount(Game game), that any discount calculator must implement. The DiscountableGame class serves as a container for a discount calculator and relies on the Discountable interface, allowing different discount calculators, like DefaultDiscountCalculator and PremiumDiscountCalculator, to be easily interchanged without causing issues. This interchangeability showcases LSP, demonstrating that specific implementations can seamlessly replace the general interface, maintaining the expected behavior. The code promotes a consistent and predictable usage of discount calculations, emphasizing the adherence to Liskov Substitution.

LISKOV SUBSTITUTION PRINCIPLE(LSP)

```
public interface Discountable {
  double calculateDiscount(Game game);
}
```

```
public class DiscountableGame {
  private Discountable discountable;
  public DiscountableGame(Discountable discountable) {
     this.discountable = discountable;
  }
  public double applyDiscount(Game game) {
     return discountable.calculateDiscount(game);
  }
}
```

```
public class DefaultDiscountCalculator implements Discountable {
  @Override
  public double calculateDiscount(Game game) {
     return game.getPrice() * 0.1;
  }
}
```

```
public class PremiumDiscountCalculator implements Discountable {
  @Override
  public double calculateDiscount(Game game) {
     return game.getPrice() * 0.2;
  }
}
```

INTERFACE SEGREGATION PRINCIPLE(ISP)

- The Interface Segregation Principle (ISP) suggests that classes should not be forced to implement interfaces with methods they don't need.
- It promotes creating smaller, focused interfaces tailored to specific functionalities, allowing classes to implement only what is relevant.

Example:In the below code, the Interface Segregation Principle (ISP) is applied by creating focused interfaces: AnalyticsProvider for tracking analytics and PaymentProcessor for processing payments. The SimplePaymentProcessor class implements only the PaymentProcessor interface, adhering to ISP by implementing only what's relevant. The GamePaymentProcessor class implements both interfaces, showcasing ISP compliance by allowing selective implementation based on functionality needs. This approach ensures that classes are not forced to implement unnecessary methods, aligning with ISP's goal of creating modular and efficient interfaces.

INTERFACE SEGREGATION PRINCIPLE

```java
public interface AnalyticsProvider {
  void trackAnalytics();
}
```

```java
public interface PaymentProcessor {
  void processPayment(double amount);
}
```

```java
public class SimplePaymentProcessor implements PaymentProcessor {
  @Override
  public void processPayment(double amount) {
    System.out.println("Payment Processed");
  }
}
```

```java
public class GamePaymentProcessor implements PaymentProcessor, AnalyticsProvider {
  @Override
  public void processPayment(double amount) {
          System.out.println("Payment Processed");
  }
  @Override
  public void trackAnalytics() {
          System.out.println("Analytics Tracked");
  }
}
```

DEPENDENCY INVERSION PRINCIPLE(DIP)

- The Dependency Inversion Principle (DIP) in object-oriented design advocates that high-level and low-level modules should depend on abstractions, not on concrete details.
- It emphasizes the importance of defining clear abstractions (interfaces or abstract classes) that serve as contracts, allowing for flexibility in system design.
- Concrete implementations should then depend on these abstractions, facilitating the introduction of new implementations without disrupting existing code.

Example:In the below code, the Dependency Inversion Principle (DIP) is implemented by introducing an abstraction (NotificationService interface) that defines the contract for sending notifications. The NotificationManager class then depends on this abstraction, receiving a NotificationService instance through its constructor. This adheres to the principle of depending on abstractions rather than concrete details. The introduction of EmailNotificationService as a concrete implementation showcases the flexibility of DIP, allowing new notification services to be added without modifying existing code. This design promotes the inversion of dependencies, where high-level NotificationManager depends on the abstraction, and concrete implementations depend on the same abstraction.

DEPENDENCY INVERSION PRINCIPLE(DIP)

```
public interface NotificationService {
  void sendNotification(String message);
}


public class NotificationManager {
  private NotificationService notificationService;
  public NotificationManager(NotificationService
notificationService) {
      this.notificationService = notificationService;
  }
  public void notifyUser(String message) {
      notificationService.sendNotification(message);
  }
}
```

```
public class EmailNotificationService implements NotificationService {
  @Override
  public void sendNotification(String message) {
           System.out.println("Email sent!"+message);
  }
}
```