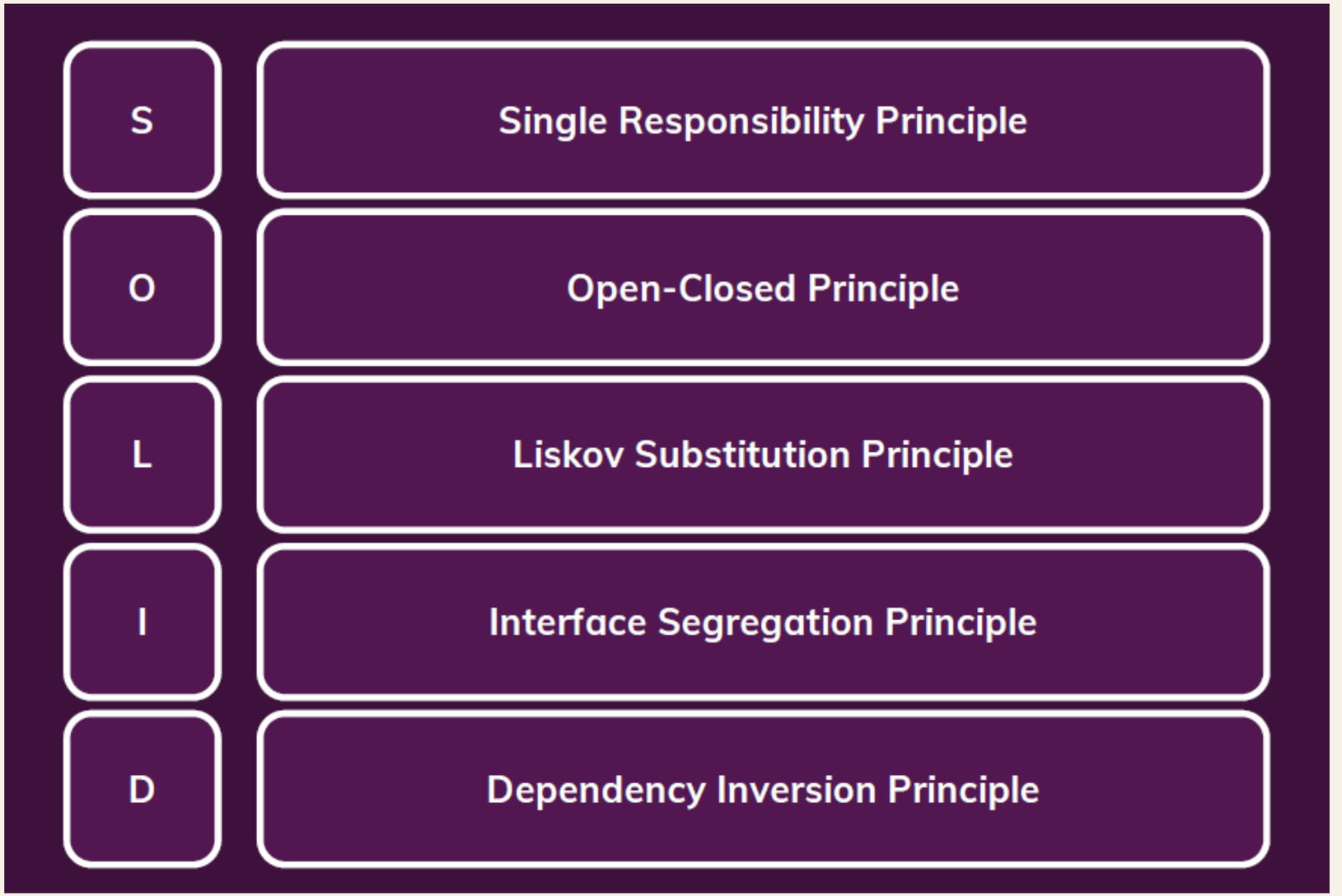
SOLID PRINCIPLES

Use case: Review Reward System

By Nebil.V
Trainee

SOLID PRINCIPLES



Single Responsibility Principle

Classes should have a single responsibility – a class shouldn't change for more than one reason.

Single Responsibility Principle

```
class PointReward extends Reward {

    private int requiredPoints;


    public PointReward(String rewardName, int requiredPoints) {

        super(rewardName);
        this.requiredPoints = requiredPoints;
    }

    public boolean isRedeemable(int customerPoints) {
        return customerPoints >= requiredPoints;
    }

}
```

Open Closed Principle

A class should be open for extension but closed for modification.

Open Closed Principle

```java
public class Reward {
    private String rewardName;

    public Reward(String rewardName) {
        this.rewardName = rewardName;
    }

    public String getRewardName() {
        return rewardName;
    }

    public void setRewardName(String rewardName) {
        this.rewardName = rewardName;
    }
}
```

Open Closed Principle

```
class PointReward extends Reward {

    private int requiredPoints;


    public PointReward(String rewardName, int requiredPoints) {

        super(rewardName);
        this.requiredPoints = requiredPoints;
    }

    public boolean isRedeemable(int customerPoints) {
        return customerPoints >= requiredPoints;
    }

}
```

Open Closed Principle

The existing code of the **Reward** class remains untouched, and you can introduce new reward types by creating more subclasses like **PointReward** without modifying the Reward class.

Liskov Substitution Principle

Objects should be replaceable with instances of their subclasses without altering the behavior.

Liskov Substitution Principle

```
public class Customer {
    private String name;
    private int loyaltyPoints;

    public Customer(String name) {
        this.name = name;
        this.loyaltyPoints = 0;
    }

    public void earnLoyaltyPoints(int points) {
        this.loyaltyPoints += points;
    }

    public int getLoyaltyPoints() {
        return loyaltyPoints;
    }

}
```

Liskov Substitution Principle

```java
public class PremiumCustomer extends Customer {
    public PremiumCustomer(String name) {
        super(name);
    }
public void earnDoubleLoyaltyPoints(int points) {
        this.earnLoyaltyPoints(points*2);
    }
}
```

PremiumCustomer is a subclass of Customer, and it extends the functionality by adding a method (earnDoubleLoyaltyPoints) without modifying or breaking the behavior of the existing methods in the Customer class.You can replace an instance of Customer with an instance of PremiumCustomer without introducing errors or unexpected behavior.

Interface Segregation Principle

Many client-specific interfaces are better than one general purpose interface.

Interface Segregation Principle

```java
public interface TextReview {
     void submitTextReview(String reviewText);
}
```

```java
public interface Rating {
    void submitRating(double rating);
}
```

```java
public class RatingOnlyReview implements Rating{
    @Override
    public void submitRating(double rating) {
        System.out.println("Only Rating submitted: " + rating);
    }
}
```

Interface Segregation Principle

```
public class OverallReview implements Rating, TextReview {

    @Override
    public void submitRating(double rating) {
        System.out.println("Rating submitted: " + rating);
    }

    @Override
    public void submitTextReview(String reviewText) {
        System.out.println("Text review submitted: " + reviewText);
    }
}
```

In this example, **RatingOnlyReview** implements only the **Rating** interface, and **OverallReview** implements both Rating and **TextReview** interfaces.
Thus classes can implement interfaces that are relevant to their specific functionality.

Dependency Inversion Principle

You should depend upon abstractions, not concretions.

Dependency Inversion Principle

```java
public interface ConvertRating {
    void convertRating(double rating);
}
```

```java
public class RatingConverter implements ConvertRating{

    public void convertRating(double rating) {
        System.out.println("Converting rating with value: " + rating*20+"%");
    }
}
```

Dependency Inversion Principle

```
public class ReviewService {
    private ConvertRating ratingConverter;

    public ReviewService(ConvertRating ratingConverter) {
        this.ratingConverter = ratingConverter;
    }

    public void convertRating(double rating) {
        ratingConverter.convertRating(rating);
    }
}
```

Here the **ReviewService** class depends on the interface **CalculateRating** that serves as an abstraction ,rather than a concrete implementation.

Thank you