SOLID Principle

The SOLID principles are a set of five design principles introduced by Robert C. Martin. They are one of the object-oriented approaches used in software development, intended to create quality software. The broad goal of the SOLID principles is to reduce dependencies, so that developers can change one area of the software without affecting others. Furthermore, they are intended to make designs easier to understand, maintain, reuse, and extend.

Single responsibility principle (SRP)

- SRP states that, a class should have only one reason to change, meaning it should have a single responsibility.
- This principle encourages you to create classes that do one thing and do it well.
- Lots of responsibilities make the class highly coupled, harder to maintain and harder to understand.

For an example, consider the BankAccount class below:

```java
public class BankAccount {
    private double balance;
    private String accountNo;
    private String accountType;

    //constructor
    public BankAccount(double balance, String accountNo, String acco
        this.balance = balance;
        this.accountNo = accountNo;
        this.accountType = accountType;
    }

    public void deposit() {
        //code to deposit amount
    }

    public void withdraw(double amount) {
        //code to withdraw amount
    }

    public double calculateInterest() {
        //code to calculate interest
    }

    public void saveBankAccountDetails() {
        //save account information to database
    }

    public void sendSmsNotification() {
        //code to send SMS notification to customer
    }
}
```

In the context of BankAccount class, managing deposits, withdrawals and interest are reasonable and related responsibilities to account management. But saveBankAccountDetails and sendSmsNotification methods are not related to bank account management's behavior. Hence, this class is violating SRP. The easiest way to fix this problem is to create separate classes for managing bank accounts, save information to database and send SMS notifications, so that each class having only one responsibility.

```
// BankAccount class will handle account related responsibilities
public class BankAccount {
    private double balance;
    private String accountNo;
    private String accountType;
    private SQLBankAccountRepository sqlBankAccountRepository;
    private NotificationService notificationService;

    //constructor
    public BankAccount(double balance, String accountNo, String acc
        this.balance = balance;
        this.accountNo = accountNo;
        this.accountType = accountType;
        this.sqlBankAccountRepository = sqlBankAccountRepository;
        this.notificationService = notificationService;
    }

    public void deposit() {
        //code to deposit amount
    }

    public void withdraw(double amount) {
        //code to withdraw amount
    }

    public double calculateInterest() {
        //code to calculate interest
    }

    // SQLBankAccountRepository class will handle database related r
    public class SQLBankAccountRepository {
        public void saveBankAccountDetails(BankAccount bankAccount)
            //save account information to database
        }
    }

    // NotificationService class will handle notification related re
    public class NotificationService {
        public void sendSmsNotification(BankAccount bankAccount) {
            //code to send SMS notification to customer
        }
    }
}
```

Open closed principle (OCP)

- OCP states that, software entities (such as classes, modules, functions, etc.) should be open for extension but closed for modification.

- In other words, you should be able to add new functionality or behavior to a system without altering the existing code.
- Adding a new feature to software entities by modifying it, can lead new bugs, poor readability and hard to maintain.

For an example consider the calculateInterest method of BankAccount class.

```
public class BankAccount {
    private double balance;
    private String accountNo;
    private String accountType;
    private SQLBankAccountRepository sqlBankAccountRepository;
    private NotificationService notificationService;

    //constructor
    public BankAccount(double balance, String accountNo, String acc
        this.balance = balance;
        this.accountNo = accountNo;
        this.accountType = accountType;
        this.sqlBankAccountRepository = sqlBankAccountRepository;
        this.notificationService = notificationService;
    }

    public void deposit() {
        //code to deposit amount
    }

    public void withdraw() {
        //code to withdraw amount
    }

    public double calculateInterest() {
        if (this.accountType.equals("Savings"))
            return this.balance * 0.03;
        else if (this.accountType.equals("Checking"))
            return this.balance * 0.01;
        else if (this.accountType.equals("FixedDeposit"))
             return this.balance * 0.05;
    }
}
```

There is a problem with the calculateInterest method. What if there is a new account type introduced with new interest requirement? We have to add another if condition in the calculateInterest method. It violates OCP. The easiest way to fix this problem is creating a common interface for all account types and implement it for every account types.

```java
public interface BankAccount {
    void deposit();
    void withdraw(double amount);
    double calculateInterest();
}


public class SavingsBankAccount implements BankAccount {
    // attributes and constructor
    // deposit and withdraw method declarations

    @Override
    public double calculateInterest() {
        return this.balance * 0.03;
    }
}


public class CheckingBankAccount implements BankAccount {
    // attributes and constructor
    // deposit and withdraw method declarations

    @Override
    public double calculateInterest() {
        return this.balance * 0.01;
    }
}


public class FixedDepositBankAccount implements BankAccount {
    // attributes and constructor
    // deposit and withdraw method declarations

    @Override
    public double calculateInterest() {
        return this.balance * 0.05;
    }
}
```

With this implementation, we can calculate interests without modifying the underlying logic. The class is open for extension (new account classes can be added) but closed for modification (existing calculateInterest methods remain untouched).

Liskov Substitution Principle (LSP)

- LSP states that objects of a derived class should be able to replace objects of the base class without affecting the correctness of the program.
- In other words, if a class is a subclass of another class, it should be able to substitute its parent class without causing problems.
- This principle ensures that inheritance relationships are well-designed and that the derived class adheres to the contract of the base class.

For an example, assume that, we have a superclass A and three subclasses B, C, and D.

```
A obj1 = new B();

A obj2 = new C();

A obj3 = new D();
```

To ensure a valid use of LSP, all of the above 3 statements should run perfectly without interrupting the program flow.

Let's take a more detailed example:

```
class Bird {
    public void eat() {
        System.out.println("This bird can eat.");
    }

    public void fly() {
        System.out.println("This bird can fly.");
    }
}


class Parrot extends Bird {
}


class Penguin extends Bird {
    @Override
    public void fly() {
        throw new FlyException("Penguins cannot fly");
    }
}
```

The Penguin class overrides the fly() method from the base class, but the behavior is fundamentally different from what's expected by the base class. This is an LSP violation because when we try to substitute an instance of Penguin for a generic Bird, it will not behave as a typical bird in terms of flying. This could lead to unexpected behavior in the code.

To resolve this LSP violation, you should restructure the class hierarchy and ensure that derived classes confirm to the contract of the base class. One way to fix this issue is to use composition or interfaces to handle behaviors that don't fit the base class's contract.

```
class Bird {
    public void eat() {
        System.out.println("This bird can eat.");
    }
}


class FlyingBird extends Bird {
    public void fly() {
        System.out.println("This bird can fly.");
    }
}


class Parrot extends FlyingBird {
}


class Penguin extends Bird {
}
```

Interface segregation principle (ISP)

- ISP states that, clients should not be forced to depend on interfaces they do not use.
- This principle encourages you to create specific, fine-grained interfaces rather than large, monolithic ones, to avoid forcing clients to implement methods they don't need.

For an example consider the `withdraw()` method of LoanBankAccount class that implements previously discussed BankAccount class.

```java
public interface BankAccount {
    void deposit();
    void withdraw(double amount);
    double calculateInterest();
}


public class SavingsBankAccount implements BankAccount {
    // attributes and constructor
    // deposit and calculateInterest method declarations

    public void withdraw(double amount) {
        if (this.balance < amount)
            this.balance -= amount;
    }
}


public class CheckingBankAccount implements BankAccount {
    // attributes and constructor
    // deposit and withdraw method declarations

    public void withdraw(double amount) {
        if (this.balance < amount)
            this.balance -= amount;
    }
}


public class LoanBankAccount implements BankAccount {
    // attributes and constructor
    // deposit and withdraw method declarations

    public double withdraw() {
        //empty method - cannot withdraw from loan accounts
    }
}
```

Here, withdraw method in SavingsBankAccount and CheckingBankAccount classes working fine. But LoanBankAccount have an empty withdraw method, because in loan account withdrawing process not allowed. The implementation classes should use only the methods that are required. We should not force the client to use the methods that they do not want to use. That's why the principle states that the larger interfaces split into smaller ones.

```
public interface BankAccount() {
    void deposit();
    double calculateInterest();
}
```

```
public interface Withdrawable {
    void withdraw();
}
```

```
public class SavingsBankAccount implements BankAccount,  Withdrawab
    //deposit, calculateInterest, withdraw methods definitions
}
```

```
public class CheckingBankAccount implements BankAccount, Withdrawab
    // deposit, calculateInterest, withdraw methods definitions
}
```

```
public class LoanBankAccount implements BankAccount {
    // deposit, calculateInterest methods definitions
}
```

Here, we created BankAccount interface for deposit and calculateInterest and `Withdrawable` interface for withdraw. So that implementation classes can implement necessary interfaces according to its need.

Dependency inversion principle (DIP)

The principle states that we must use abstraction (abstract classes and interfaces) instead of concrete implementations. The DIP states that:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.
- In simpler terms, the principle encourages you to rely on interfaces or abstract classes to decouple your code and make it easier to extend, maintain, and test.

Let’s understand the principle through an example.

```
class PDFReportGenerator {
    public void generatePDFReport() {
        // PDF generation logic
    }
}


class HTMLReportGenerator {
    public void generateHTMLReport() {
        // HTML generation logic
    }
}


class ReportService {
    private PDFReportGenerator pdfGenerator;
    private HTMLReportGenerator htmlGenerator;

    public ReportService() {
        pdfGenerator = new PDFReportGenerator();
        htmlGenerator = new HTMLReportGenerator();
    }

    public void generatePDFReport() {
        pdfGenerator.generatePDFReport();
    }

    public void generateHTMLReport() {
        htmlGenerator.generateHTMLReport();
    }
}
```

In the above code, ReportService directly depends on concrete implementations of report generators (PDFReportGenerator and HTMLReportGenerator). This leads to high coupling between the high-level and low-level modules. To adhere to the Dependency Inversion Principle, you should introduce abstractions (interfaces or abstract classes) and rely on those abstractions instead.

```
interface ReportGenerator {
    void generateReport();
}
```

```
class PDFReportGenerator implements ReportGenerator {
    public void generateReport() {
        // PDF generation logic
    }
}


class HTMLReportGenerator implements ReportGenerator {
    public void generateReport() {
        // HTML generation logic
    }
}


class ReportService {
    private ReportGenerator reportGenerator;

    public ReportService(ReportGenerator generator) {
        this.reportGenerator = generator;
    }

    public void generateReport() {
        reportGenerator.generateReport();
    }
}
```

In this updated code, we introduced the ReportGenerator interface, and the ReportService now depends on this abstraction rather than concrete implementations. This decouples the high-level module from low-level modules, and you can easily swap out different report generators without modifying the ReportService class.

Resources

- https://refactoring.guru/