Solid Principles

1) Single Responsibility :

Account.Java : A class should have Single Responsibility

Bad Example:

```
public class Account {

    int money;

    public getAccountNumber();
    public setAccountNumber();
    public getName();
    public setName();

    public int deposit(); // Account class
                             Should not take
                             responsibility of
                             Transaction class
}
```

Transaction.Java:

```
public class Transaction {


    public int withdraw();
}
```

Good Example:

Account.Java:

```
public class Account {
    int money;
    public getAccountNumber();
    public setAccountNumber();
    public getName();
    public setName();
    ~~public int deposit()~~;   // Account class
                                 should not take
}                                responsibility of
                                 Transaction class
```

Transaction.Java:

```
public class Transaction {
    public int deposit();
    public int withdraw();
}
```

OPEN / CLOSE:

(Open for Extension, Closed for modification)

Bad Example: Calculator.java:

```
public class calculator {

  switch (operation) {

    case "add":

        return num1 + num2;
    Case " sub" :
        return num1 - num2:
  }

}
```

If we want to add another feature i.e., Multiplication. then we are modifying the existing code

This is not okay.

Good Example:

Calculator.Java:

```
public class Calculator{
    public void perform(int num1, int num2,
                                Operation operation)
    {
      operation.perform(num1, num2);
    }
}
```

Operation.Java:

```
Public interface Operation{
    public void perform(int num1, int num2);
```

AddOperation.Java:

```
public class AddOperation implements
                              operation{
   @override
   public void perform(int a, int b);
       return a+b;
```

Sub operation.Java:

```
public class SubOperation implements
                              operation{
   @override
   public void perform(int a, int b);
       return a-b;
```

We can add multiplication

Liskov Substitution Principle:

Both parent & child class pointers should be interchangeable.

Bad Example:

LoanPayment.Java:

```
public interface LoanPayment {
    public void doPayment();


    public void ForceCloseLoan();
}
```

HomeLoanPayment.Java:

```
public class HomeLoanPayment implements
                                        LoanPayment
   public void doPayment();
   public void doRepayment();
   public void ForceCloseLoan();
}
```

CreditLoanPayment.Java:

```
Public class creditLoanPayment implements
                                        LoanPayment
  public void doPayment();
  public void doRepayment(){throw Error;}
  public void ForceCloseLoan();{
}                                 throw Error}
```

LoanClosureService.Java:

```
public class LoanClosureService{
  LoanPayment homeloan = new
                          HomeLoan();

  homeloan.forceCloseLoan() // no error;

  LoanPayment creditLoan = new
                          creditLoan();

  creditLoan.forcecloseLoan();
                    // Error.
```

Violation of Liskov Substitution Principle

Good Example:

LoanPayment.Java:

```
public interface LoanPayment {
      public void doPayment();
}
```

SecureLoan.Java:

```
public interface SecureLoan extends
                    LoanPayment{
      public void forceCloseLoan();
}
```

HomeloanPayment.Java:

```
public class HomeLoanPayment implements
                                    Secureloan
    public void doPayment();


    public void Forcecloseloan();
}
```

CreditLoanPayment.Java:

```
Public class credit-LoanPayment implements
                                    LoanPayment
    public void doPayment();
 }
```

LoanclosureService.Java

```
public class LoanclosureService {
   public Secureloan secureLoan;
   public LoanclosureService(Secureloan
                                  secureloan)
      this.secureloan = secureloan;
   }

   secureLoan.forcecloseloan();

 }
```

Interface Segregation:

Donot bottleneck the client with multiple implementation. segregate the interfaces with corresponding functionality

Bad Example:

```
public interface DAOInterface {
    public void openConnection();
    public void createRecord();
    public void deleteRecord();
    public void openFile();
}
public class DBImplemention implements
  DAOInterface {

    public void openConnection();
    public void createRecord();
    public void deleteRecord();
    public void openFile()
    {
        DB don't need File
    }
```

```
public class File,Implementtion implements
    DAOInterface {

        public void openFile();
        public void createRecord();
        public void deleteRecord();
        public void Openconnection()
        {
              File dont need DB

        }
```

It's a bottleneck ; Therefore, lets segregate interfaces.

```
public interface operations {
      public void createRecord()
      public void deleteRecord();
}
public interface DBconnection{
      public void openconnection()
}
public interface FileOperation{
      public void openFile()
}
```

```
public class DAO implements
                FileOperation, Operations

public class DBO implements
                DBconnection, Operations
```

Dependency Injection:

one class should not have dependency internally

Bad Example: Calculator.Java:

```
public class Calculator {

  Switch (operation) {

    case "add":
      AddOperation a = new AddOperation()
                a. perform
    Case " Sub" :
      SubOperation a = new SubOperation()
  }             a. perform

}
```

Calculator depends on AddOperation & SubOperation class

Good Example:

Calculator.Java:

```
public class calculator{
    public void perform(int num1, int num2,
                        Operation operation)
    {
        operation.perform(num1, num2);
    }
}
```

Operation. Java:

```
Public interface Operation{
    public void perform(int num1, int num2);
```

Add Operation.Java:

```
public class AddOperation implements
                                Operation{
    @override
    public void perform(int a, inb b);
        return a+b;
```

Sub operation.Java:

```
public class subOperation implements
                                operation{
    @override
    public void perform(int a, inb b);
        return a-b;
```

No dependency.