① Single Responsibility Principle:

One class should have one and only one responsibility.

② Open/closed Principle:

Software components should be "Open for extension, but closed for modification".

③ Liskov's Substitute Principle:

Derived types must be substitutable for their base type

④ Interface Segregation Principle:

Clients should not forced to implement unnecessary methods which they will not use

⑤ Dependency Inversion Principle

High-level modules should not depend on low-level modules. Both depen on abstraction. Abstractions should not depend on details. Details should depend on abstractions

SOLID Principles

29/10/2024

Authors: Robert C. Martin (Uncle Bob)
Developed the solid principles in his 2000 paper "Design Principles and Design Patterns".
These concepts were later built upon by Michael Feathers. who introduced us to the SOLID acronym.

SOLID principles ensure maintable, scalable, and flexible code by promoting modular design, loose coupling, and focused responsibilities.
They faciliate easy feature addition, reusability, and bug prevention.
Adopting these principles leads to robust and reliable software systems.

These are five principles are:

1. Single Responsibility Principle (SRP) ... S
2. Open / Closed Principle O
3. Liskov's Substitution Principle (LSP) L
4. Interface Segregation Principle (ISP) I
5. Dependency Inversion Principle (DIP) D

The broad goal of the SOLID principles is to reduce dependencies, so that the engineers change one area of software without impacting others. Additionally, they're intended to make design easier to understand, maintain, and extend. Ultimately, using these design principles makes it easier for SW Engs to avoid issue to build adaptive, effective, and agile software.
→ better code for readability, maintainability, design patters, testability

30/12/24

① Single Responsibility Principle:

It states that "One class should have one and only one responsibility".

which specifically means - we should write, change, and maintain a class only for one purpose.

Change class only when you need to change state of one particular object or instance.

Example: POJOs follow SRP

* Suppose we have Employee and Address class, If we want to change the state of Employee then we do not need to modify the class Account and vice-versa.
* If you would have merged both as single POJO, then modification in one field for address (like state) needs to modify whole POJO including Employee.

Worst Design:

which dont follow SRP.

Hitting Database in POJO of Employee class.

That's why we have service layer, DAP layer and Entities separated.

Example: Employee.java

```
public class Employee {
    private int id;
    private String name;

    Employee(int id, String name){
        this.id = id;
        this.name = name;
    }
    public void printMe(){
        System.out.println("Employee ID is" + this.id
            + "Employee name is " + this.name);
    }
}
```

The task of above class is manage Employee state object only

→ If we try to add one more responsibility to the class of emp like Adress related if we add, it will be violation of SRP.

→ So we need to create another Address.java class and maintain Address related only.

Ex: Address.java

```
public class Address {
    private String First;
    private string Second;
    private string City;
    public string getFirst() {
        return First;
    }
    public void setFirst(String first){
        First = first;
    //All getters
    //All setters
}
```

② Open/Closed Principle:

Software components should be "open for extension, but closed for modification"

The term "open for extension" means that we can extend and include extra functionalities in our code without altering or affecting our existing implementation.

The term "closed for modification" means that after we add the extra functionality, we should not modify the existing implementation.

You have noticed that you change something to cater a new requirement and some other functionality breaks because of your change.
To prevent that we have this principle in hand.

Implementation:

The application classes should be designed in such a way that whenever fellow developers want to change the flow of control in specific condition in application, all they need to extend the classes and override some functions and that's it.

Example: Created a pojo employee id, name now new functionality comes which says add Training to, your constructor will fail for employees who didnt do training, better extend employee class, name it Trained Employee class then add constructors.

Employee class:

```
public class Employee {
    private int id;
    private String name;

    Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }
    public void printMe() {
        System.out.println("Employee ID is" + this.id
            + "Employee name is" + this.name);
    }
}
```

→ In Employee class if we add new fields like Trained Employee and will add in constructor also will fail (Emps who didn't have done training).

TrainedEmployees.java

```
public class TrainedEmployees extends Employee {
    private String trainingAreas;
    TrainedEmployees(int id, String name, String trainingAreas)
    {
        super(id, name);
        this.trainingAreas = trainingAreas;
    }
}
```

→ Employee class is closed for modification.
→ Employee class is open for extension.

LSP introduced by - Barbara Liskov in 1987

③ Liskov Substitution Principle

LSP states that the software should not alter the desirable results when we replace a parent type with any of the subtypes". or

Derived types must be completely substitutable for their base types"

LSP means that the classes, fellow developers created by extending our class, should be able to fit in application without failure. This is important when we resort to polymorphic behaviour through inheritance.

Take example, print employee from child or parent references print all required details.

This requires the objects of the subclasses to behave in the same way as the objects of the superclass.
This is mostly seen in places where we do runtime type identification and then cast it to appropriate reference type.

This avoids misusing inheritance.
It helps us conform to the "is-a" relationship.
We can also say that, subclass must fulfill a contract defined by the base class

Wrongly Implementing it can prove real world objects wrong like:
"Square is Rectangle", it is not in real world but its easy to implement through code.
To prevent this "Square is a Rectangle", it is not in real world but its easy to implement through code. To prevent this Better use this principle.

Ex:
```
// Base class for shapes
class Rectangle {
    protected double width;
    protected double height;
    public Rectangle(double w, double h) {
        this.width = w;
        this.height = h;
    }
    public double area() {
        return width * height;
    }
    public double getWidth() {
        return width;
    }
    public double getHeight() {
        return height;
    }
```

```
public void setWidth(double w){
    this.width = w;
}
public void setHeight(double w){
    return height; this.light = h;
}
}
// Derived class for squares

class Square extends Rectangle {
  public Square(double size){
      super(size, size);
  }
@Override
  public void setWidth(double w){
      this.width = this.height = w; //Ensure both
  }                                 width and height remain the
}                                   same.
```

To see potential violation of LSP consider what would happen if you where to use the square class in a context expecting Rectangle

If you substitute a square where a Rectangle is expected, changing just the width or height would lead to unexpected results because it will change both dimensions.

4. Interface Segregation Principle

It states that "clients should not be forced to implement unnecessary methods which they will not use".

ISP is applicable to interfaces as a single responsibility principle holds to classes.
ISP states that we should split our interfaces into smaller and more specific ones.

Ex: PizzaApp.java

```
public interface PizzaApp {
    public void acceptOrdersOnline();
    public void acceptWalkOrders();
    public void acceptTelephonicOrders();
    public void acceptPaymentsOnline();
    public void acceptOnlyCash();
}
```

If we implementing above interface.

```
interface Payment {
    void payByCard();
    void payByCash();
    void payByDigitalWallet();
    void payByNetBanking();
}
// online payment system forced to implement
irrelevant methods.

class OnlinePayment implements Payment {
  @Override
    public void payByCard() {
      S.o.p(" paying online by card.");
  @Override
    public void payByCash() {
      throw new UnsupportedOperationException("Cash
    payment not supported for online.");

  @Override
  public void payByDigitalWallet() {
    S.o.p("Paying online using digital wallet.");
  }
@Override
  public void payByNetBanking() {
    S.o.p("Paying online via net banking.");
  }
}
```

// Walk-in payment system forced to implement irrelevant methods

```
class WalkInPayment implements Payment {
  @Override
    public void payByCard() {
      S.o.p(" Paying at counter by card.");
    }
  @Override
    public void payByCash() {
      S.o.p(" Paying at counter by cash.");
    }
  @Override
    public void payByDigitalWallet() {
      throw new UnsupportedOperationException("Digital
        wallet payment cannot be accepted in walk-ins.");
    }
  @Override
    public void PayByNetBanking() {
      throw new UnsupportedOperationException(" NetBanking
      payment cannot be accepted in walk-ins.");
    }
}
```

The above code violates LSP principle implementing unnecessary methods.

Using LSP principle:

OnlinePayment only implements CardPayment, DigitalWalletPayment, and NetBanking Payment. avoiding unnecessary methods

WalkingPayment only implements CardPayment and CashPayment making the system more modular and easier to maintain.

Dependency Inversion Principle

① High-level modules should not depend on low-level modules. Both should depend on abstractions.
② Abstractions should not depend on details. Details should depend on abstractions.

This principle ensures loose coupling b/w different layers of application.

```
// Low-level classes
class CreditCardPayment {
    public void processPayment(double amount) {
        S.o.p("Processing credit card payment" + amount);
    }
}

class PayPalPayment {
    public void processPayment(double amount) {
        S.o.p("Processing PayPal payment of" + amount);
    }
}

// High-level class directly depend on low-level class
class PaymentService {
    private CreditCardPayment creditCardPayment =
            new CreditCardPayment();
    private PayPalPayment payPalPayment =
            new PayPalPayment();
    public void pay(String method, double amount)
    {
```

```
        if (method.equals("creditCard")) {
            creditCardPayment.processPayment(amount);
        } else if (method.equals("paypal")) {
            payPalPayment.processPayments(amount);
        }
    }
}
```

Issues:

1. Tight coupling: The PaymentService class is tightly coupled with specific payment methods.
2. Hard to Extend: Adding a new payment method (e.g., UPI) requires modifying the PaymentService class.

With Dependency Inversion Principle we introduced an abstraction (PaymentMethod) that both the high-level module (PaymentService) and low-level modules (CreditCardPayment, PayPalPayment) depend on.

Example:

```
// Abstraction for payment methods
interface PaymentMethod {
    void processPayment(double amount);
}

// Low-level class depend on the abstraction
class PayPalPayment implements PaymentMethod {
    @Override
    public void processPayment(double amount) {
        S.o.p("Processing PayPal payment of $" + amount);
    }
}
```

```
class CreditCardPayment implements PaymentMethod {
@Override
  public void processPayment(double amount) {
    S.o.p("Processing credit card payment of $"+amount
  }
}
class UPIPayment implements PaymentMethod {
@Override
  public void processPayment(double amount) {
    S.o.p("Processing UPI payment of $"+amount);
  }
}

// High-level class depends on the abstraction
  class PaymentService {
    private PaymentMethod paymentMethod;
    public PaymentService(PaymentMethod paymentMethod) {
      this.paymentMethod = paymentMethod;
    }
    public void pay(double amount) {
      paymentMethod.processPayment(amount);
    }
  }
```

Benefits:

Loose Coupling: 1. The PaymentService class is decoupled from specific payment methods.

2. Extensibility: Adding new payment methods (e.g. UPI) only requires implementing the PaymentMethod interface without modifying existing code.

3. Testability: Mock implementations of PaymentMethod can be created for testing purposes.